Feature Based Visualization

Dem Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
genehmigte

Dissertation
von
Younis O. Hijazi

Thesis supervisors:
Prof. Dr. Hans Hagen, TU Kaiserslautern
Prof. Dr. Charles Hansen, University of Utah

President of the Ph.D. committee:
Prof. Dr. Klaus Madlener, TU Kaiserslautern

Dean:
Prof. Dr.-Ing. Reinhard Gotzhein, TU Kaiserslautern

Ph.D. defense: 14. December 2007

Kaiserslautern, February 2008

Abstract

In this thesis we apply powerful mathematical tools such as interval arithmetic for applications in computational geometry, visualization and computer graphics, leading to robust, general and efficient algorithms. We present a completely novel approach for computing the arrangement of arbitrary implicit planar curves and perform ray casting of arbitrary implicit functions by jointly achieving, for the first time, robustness, efficiency and flexibility. Indeed we are able to render even the most difficult implicits in real-time with guaranteed topology and at high resolution. We use subdivision and interval arithmetic as key-ingredients to guarantee robustness. The presented framework is also well-suited for applications to large and unstructured data sets due to the inherent adaptivity of the techniques that are used. We also approach the topic of tensors by collaborating with mechanical engineers on comparative tensor visualization and provide them with helpful visualization paradigms to interpret the data.

Contents

Remerciements

J'aimerais remercier Hans Hagen de m'avoir offert l'opportunité de préparer ma thèse de doctorat dans d'excellentes conditions, au sein du programme doctoral international IRTG 1131 (financé par le DFG), de m'avoir guidé tout au long de ces trois années et surtout de m'avoir offert une grande liberté dans mes travaux de recherche. Je suis très reconnaissant de la confiance qu'il m'a accordée. Hans a toujours su habilement me conseiller, surtout dans les moments critiques, ce qui m'a permis d'aller sans cesse de l'avant. J'aimerais également remercier Thomas Breuel avec qui j'ai beaucoup appris. C'est Thomas qui m'a initié en particulier à l'arithmétique des intervalles, outil qui s'est avéré être le pilier de mon travail. Je le remercie aussi pour les nombreux conseils utiles qu'il m'a donnés.

Un grand merci à Chuck Hansen, mon co-directeur de thèse, avec qui ça a été un plaisir de collaborer. Chuck a toujours été très réactif lorsque j'ai sollicité son soutien. Merci d'avoir relu mon travail et participé à mon jury de thèse. Je remercie également Klaus Madlener d'avoir accepté de présider mon jury. Merci également à Ken Joy, Bernd Hamann, Wolfgang Kollmann et Ingrid Hotz pour le temps qu'ils m'ont accordé lorsque j'étais à UC Davis et pour leur intérêt envers mon travail. Merci à Georg Umlauf et Gerald Farin pour les nombreuses discussions scientifiques concernant les méthodes de subdivision, notamment à propos de mon algorithme de "sweeps". Merci à Stefanie Hahmann et Georges-Pierre Bonneau pour leur soutien concernant la préparation de l'après-thèse.

Je voudrais tout particulièrement remercier Aaron Knoll avec qui j'ai largement collaboré, surtout dans la deuxième moitié de ma thèse. J'ai découvert avec lui et Chuck un tout nouveau domaine de recherche: le lancer de rayon temps-réel. En combinant nos travaux de recherche respectifs, nous sommes arrivés - sans grande ambition au départ - à des résultats spectaculaires, fruits de nombreuses semaines (et nuits) de travail incessant. Je remercie Aaron pour la patience dont il a fait preuve afin de rendre son domaine de recherche accessible et surtout de m'avoir montré comment rédiger un article de recherche avec style.

Je remercie toutes les personnes ayant contribué de près ou de loin à ce travail, que ce soit des collègues de la TU Kaiserslautern comme Burkhard Lehner - qui m'a beaucoup aidé dans l'implantation de mon premier algorithme (de "sweeps") - Marco Schneider,

Christoph Garth, Oliver Ruebel, Michael Schlemmer, Gerd Reis, Torsten Bierz, Rouven Mohr, Tom Bobach, Frank Michel, Ariane Middel (pour les Haribo); du DFKI, Christoph Lampert ou des universités américaines partenaires, UC Davis et University of Utah: Oliver Kreylos, Xavier Tricoche, Brian Budge, Tony Bernardin, Sung Park, Mathias Schott, Andrew Kensler, Ingo Wald. Un grand merci aussi à Thomas Lewiner qui a toujours pris le temps de répondre à mes questions.

Je tiens à exprimer ma gratitude envers les personnes qui ont toujours été là dès mon arrivée à Kaiserslautern, notamment Peter Dannenmann, Mady Gruys, Inga Scheler, qui m'ont toujours aidé quand j'en avais besoin et qui ont contribué à une atmosphère agréable au quotidien dans l'équipe de recherche AG Hagen durant mes trois années de thèse. Je tiens aussi à remercier les "Français de KL": Hary, Vinz, Dédé, Bernard et Aurel avec qui j'ai toujours bien pu décompresser en dehors du travail. Merci à Christopher Tuot pour son amitié.

Finalement, je souhaiterais remercier ma famille et mes amis de Nancy (Ben, Jean, Simon, Léo, Mike, Maud...) pour m'avoir apporté le soutien nécessaire à l'accomplissement de ce travail. Je remercie tout particulièrement mon père, Oussama, qui a cru en moi et grâce à qui j'en suis arrivé là. Il a tout fait pour que je puisse continuer mes études aussi longtemps qu'il le fallait et a toujours été présent quand j'en avais besoin. Merci à mon frère Yann pour les moments forts et complices que nous avons eus et ma sœur Luna pour son dynamisme et sa joie de vivre. Merci à ma chérie Cynthia d'avoir été à mes côtés et de m'avoir soutenu durant les différentes étapes de ma thèse, surtout en 2007, année riche en évènements. Merci de nous avoir offert la petite Adèle - le 11 octobre 2007 - qui a été adorable de laisser une semaine de plus à son papa pour finir sa thèse.

Acknowledgments

I would like to thank Hans Hagen for giving me the opportunity to prepare my Ph.D. thesis in excellent conditions at the international graduate school IRTG 1131 (financed by the DFG), for guiding me along those three years and for giving me the "carte blanche" to do my research. Above all I thank him for trusting me. He was always of good advice especially during critical situations. I would also like to thank Thomas Breuel from whom I learned a lot; in particular Thomas first introduced me to interval arithmetic, a tool which happened to be the foundation of my work. I also thank him for his numerous helpful advice.

A big thank to Chuck Hansen, my co-advisor, with whom it has been a pleasure to collaborate. Chuck has always been present when I needed his support. Thanks for having proof read my work and having participated to my Ph.D. committee. I also would like to thank Klaus Madlener for being the Chairman of my Ph.D. defense committee. Thanks to Ken Joy, Bernd Hamann, Wolfgang Kollmann and Ingrid Hotz for their time when I was in UC Davis and for their interest in my work. Thanks to Georg Umlauf and Gerald Farin for the many scientific discussions about subdivision methods, particularly about my "sweeps" algorithm. Thanks to Stefanie Hahmann and Georges-Pierre Bonneau for their support concerning the preparation of the "after thesis."

I would particularly like to thank Aaron Knoll with whom I largely collaborated especially in the second half of my thesis. I discovered - with him and Chuck - a completely new research topic (to me): real-time ray tracing. By combining our respective research works we obtained - without particular ambition at the start - impressive results which were the accomplishment of many weeks (and nights) of hard work. I thank Aaron for his patience while making his research area accessible and especially for showing me how to write a paper with slyle.

I thank all the people who contributed from close or far to this work. This includes colleagues from the TU Kaiserslautern such as Burkhard Lehner - who helped me a lot in the implementation of my first algorithm (of "sweeps") - Marco Schneider, Christoph Garth, Oliver Ruebel, Michael Schlemmer, Gerd Reis, Torsten Bierz, Rouven Mohr, Tom Bobach, Frank Michel, Ariane Middel (for the Haribo); from DFKI, Christoph Lampert

or from the american partner universities, UC Davis and University of Utah: Oliver Kreylos, Xavier Tricoche, Brian Budge, Tony Bernardin, Sung Park, Mathias Schott, Andrew Kensler, Ingo Wald. A big thank also to Thomas Lewiner who always took time to answer my questions.

I would like to express my gratitude to the people who helped me since my venue in Kaiserslautern, including Peter Dannenmann, Mady Gruys, Inga Scheler who have always helped me when I needed it and who contributed to the pleasant daily atmosphere in the research group AG Hagen during the three years of my thesis. I would also like to thank the "French guys of KL": Hary, Vinz, Dédé, Bernard and Aurel with whom I've always had fun outside work. Thanks to Christopher Tuot for his friendship.

Last but certainly not least I thank my family and my friends from Nancy (Ben, Jean, Simon, Léo, Mike, Maud...) for their support, which helped me for the accomplishment of this work. I particularly thank my father, Oussama, who believed in me and brought me here. He did everything so that I could keep studying as long as needed and has always been there when needed. Thanks to my brother Yann for the intense moments we had and to my sister Luna for her dynamisn and vividness. Thank you Cynthia for supporting me during the different steps of my thesis, especially in 2007, a year rich in events. Thank you for the sweetest present: our little Adèle - born on October 11th, 2007 - who was so kind to give a one-week delay to her father for completing his thesis.

Introduction

Visualization is a novel research topic which has gained more and more attention in the past decade. It consists of pointing out regions of interest to help understanding and interpreting data. Visualization follows the spirit of "A picture is worth a thousand words". Due to the continuous increasing amount of data generation (and storage) there is a huge need of visualization paradigms to try to understand the data. Visualization intrinsically relies on underlying features and therefore we need to define those features.

In this thesis we mainly consider implicit functions (sometimes referred as "implicits") and how to represent them. We focus on curves, static surfaces and dynamic surfaces. The implicit representation offers great flexibility for modeling $3D$ objects or animations and is also very well suited for representing Constructive Solid Geometry objects. Visualizing arbitrary implicit functions in an efficient way is a challenging task that we solve in this thesis. We also provide several visualization tools to enhance the understanding of these implicits, e.g. transparency, shadows, reflections, etc. In our work we demonstrate how powerful mathematical tools such as interval arithmetic can be applied for practical applications in computational geometry, visualization and computer graphics. We are concerned with the design of robust, general and efficient algorithms in the context of *Feature Based Visualization* and large and unstructured data sets. The new algorithms that we present in this thesis are indeed well-suited for large and unstructured data sets thanks to the inherent adaptivity of the techniques used, i.e. subdivision, interval arithmetic and ray casting. Our algorithms are very general: we make (almost) no assumptions on their input within the class of implicit functions; they are also robust and elegantly simple thanks to the combination of subdivision and interval arithmetic; finally, our algorithms are very efficient.

Interval analysis is still a little known theory from the Computer Science community. It is widely used in scientific fields where interactivity is not necessary, e.g. in Pattern Recognition and Image Understanding. It has been previously applied for several applications in Computer Graphics and Visualization but, though robust, the resulting methods were slow and therefore impacted less interest in those communities. In our work we have shown that both robustness and efficiency can be jointly achieved using interval techniques and recent optimization techniques.

Subdivision methods have gained more attention from computer scientists in the past years. For example, for the task of computing arrangements of curves or surfaces, previous methods were all based on sweeping a line (for curves) or plane (for surfaces) across the domain and registered the intersections between the sweeping line/plane and interesting objects such as end points and intersection points. Also these methods had the disadvantage that the complexity of the algorithm increases with the complexity of the input; indeed, for an arrangement of curves, they are very efficient for the class of lines but start getting much more complicated for conics, cubics or algebraic curves of higher degrees. These methods are restricted to algebraic curves and don't handle transcendental functions. In this thesis we present a completely new approach [HB07] based only on subdivision and interval arithmetic to robustly compute the arrangement of arbitrary implicit curves. By 'arbitrary' we mean that we can handle *any* implicit curve, including algebraic functions, transcendentals, compositions of those functions, etc.

Previous methods for rendering arbitrary implicit functions were limited, either in performance, correctness or flexibility. They were especially slow. In this thesis we present, for the first time, algorithms which combine those three features simultaneously. Our first contribution in the context of rendering arbitrary implicit functions is an efficient ray casting algorithm that doesn't need any particular hardware [KHH*07] and performs at interactive rates; it can perfectly run on common laptops. Our second contribution for this topic is a real-time ray casting algorithm which uses the latest graphics hardware [KHK*08]. These two algorithms are respectively the first ones for ray casting arbitrary implicit functions interactively and real-time, both robustly and flexibly.

Finally, we present a completely novel approach for visualizing a 'difference' tensor field [MBH*07]. Little work has been carried in the literature for comparative tensor visualization. Indeed most methods examine the tensor field itself and do not consider a 'difference' field. This work has been motivated by a practical mechanical engineering problem and the results we obtained with our new visualization schemes have been very helpful to the engineers.

In our work we have mainly examined the computation and visualization of features in scalar fields. The first features that we have studied are intersections of arbitrary implicit curves. As input we have n implicit curves and we want to find as accurately and robustly as possible their intersections and an abstraction of those curves. The problem of computing the arrangement of arbitrary implicit planar curves [HB07] was our first opportunity to experiment with subdivision methods together with interval arithmetic. Indeed all previous methods considered sweeping a line and their complexity increased with the complexity of the input curves. We had the idea of representing all the information using intervals, i.e. the starting domain (bounding box) and the curves, and subdivide until a

certain precision. Thus our method doesn't make any assumption on the input and so doesn't restrict to a particular class of curves.

A different kind of feature that has taken our attention - still in the context of scalar fields - is the intersection between a ray and the image of an implicit function (e.g. surface in $3D$ or $4D$ hyper-surface). Following the same philosophy as for the arrangement of curves we have intended to design a 100% interval arithmetic-based method for ray casting arbitrary implicit objects. Robustness is one desired goal for this task and is achieved using interval techniques (and has been explored in previous work). More importantly we want our algorithms to be efficient, i.e. at least interactive. Therefore we use the latest techniques and tools of the interactive ray casting and the graphics communities (together with Aaron Knoll) such as fast ray traversal, SSE optimizations or GPU programming. Thus we have been able to perform ray casting of arbitrary implicit functions, first interactively on the CPU [KHH*07] and then real-time using the latest graphics hardware [KHK*08]. With this framework we are able to render even the most difficult implicits in real-time with guaranteed topology (given a certain precision) and at high resolution.

As previously mentioned we have also approached the topic of features in tensor fields by collaborating with mechanical engineers on comparative tensor visualization [MBH*07]. Tensors are natural objects having a huge potential with applications in various fields that still need to be explored; they often have been ignored due to their lack of intuition in practical applications. We have provided two main paradigms for visualizing features in a difference tensor field. In this work the interesting features are important changes from one tensor field to another.

In Part I we review interval arithmetic (Chapter 1), some of its extensions (Chapter 2) and successful applications of interval arithmetic based techniques (Chapter 3). This part corresponds to a survey paper [HHHJ07].
Part II is our main contribution to the topic *Feature Based Visualization*. It consists of four chapters. In Chapter 4 we provide a state of the art for computing the arrangement of planar curves [Hij06], followed by our new contribution, a new approach for computing the arrangement of arbitrary implicit planar curves [HB07]. Chapter 5 is a brief overview of implicit functions rendering methods, particularly focusing on ray casting ones. Chapter 6 details our first contribution to rendering implicit functions robustly and efficiently [KHH*07]. Chapter 7 extends this approach using the latest graphics hardware and demonstrates a real-time ray tracer of implicit functions [KHK*08], our second contribution to this task.
In Part III we demonstrate new visualization paradigms for comparative tensor visualization. We detail our application problem and motivate the need for visualization. In Chapter 8 we provide two new contributions to comparative tensor visualization [MBH*07]; one reduces the tensor field to a vector field and the other one uses the tensor's invariants.

Part I

A brief survey of interval arithmetic and its applications

Introduction

Interval arithmetic (IA) was introduced by Ramon Moore [Moo66] in the 1960s as an approach to putting bounds on rounding errors in mathematical computation. The whole theory of interval analysis emerged considering the computation of both true solution and error term as a single entity, i.e. the interval. Though a simple idea, it is a very powerful technique with numerous applications in mathematics, computer science, and engineering. We survey interval arithmetic and some of its extensions in the context of self-validated arithmetics. An immediate first-order extension of interval arithmetic (IA) is known as affine arithmetic (AA) and many algorithms using interval techniques confront these two approaches; in practice there is a trade-off to find between accuracy and speed. Moreover, there are a lot more interval-based alternatives to IA or AA such as Taylor-based arithmetics [Neu03] or extensions of AA that are numerically more stable (e.g. Messine et al. [Mes02]) that we will briefly review in this chapter. Our central interest is to show how these interval techniques can be used in practical applications in computer science to provide robust algorithms without necessarily sacrificing speed, as interactivity or real-time are often desired especially in Computer Graphics & Visualization.

Chapter 1

Interval arithmetic

1.1 A brief history

This section is largely inspired from G. W. Walster's article *Introduction to Interval Arithmetic* [Wal97], as it perfectly introduces how IA emerged.

Ramon E. Moore conceived interval arithmetic in 1957, while an employee of Lockheed Missiles and Space Co. Inc., as an approach to putting bounds on rounding errors in mathematical computation. Forty years later, at the April 19, 1997 kick-off meeting of Sun Microsystems' interval arithmetic university R & D program, he explained his thinking as follows: in 1957 he was considering how scientists and engineers represent measurements and computed results as $\tilde{x} \pm \varepsilon$, where $\tilde{x}$ is the measurement (or result) and ε is the error tolerance.

While representing fallible values using the $\tilde{x} \pm \varepsilon$ notation is convenient, computing with them is not, even in a case as simple as calculating the area of a room. If the errors due to finite precision arithmetic are simultaneously taken into account, complexity increases further. Error analyses of large scientific, engineering and commercial algorithms are sufficiently complex and labor intensive that they are often not conducted. The result is that machine computing with floating-point arithmetic is not tightly linked to mathematics, science, commerce or engineering.

Moore had a better idea. He reasoned that since $\tilde{x} \pm \varepsilon$ consists of two numbers, $\tilde{x}$ and ε, why not use two different numbers to represent exactly the same information? That is, instead of $\tilde{x} \pm \varepsilon$, use $\tilde{x} - \varepsilon$ and $\tilde{x} + \varepsilon$, which define the endpoints of an interval containing the true quantity in question, i.e. x. It was this simple, yet profound, idea that started interval arithmetic and interval analysis, the branch of applied mathematics developed to numerically analyze interval algorithms.

One of the most famous references on IA is probably Moore's *Interval Analysis* book [Moo66] but there are also several more recent surveys introducing IA, e.g. [CMN02, Wal97].

1.2 What is interval arithmetic?

The same way classical arithmetic operates on real numbers, interval arithmetic defines a set of operations on intervals. We denote an interval as $\overline{\underline{x}} = [\underline{x}, \overline{x}]$, and the base arithmetic operations are as follows:

$$\overline{\underline{x}} + \overline{\underline{y}} = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$$

$$\overline{\underline{x}} - \overline{\underline{y}} = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$$

$$\overline{\underline{x}} \times \overline{\underline{y}} = [min(\underline{xy}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{xy}), max(\underline{xy}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{xy})]$$

$$\overline{\underline{x}} \,/\, \overline{\underline{y}} = \overline{\underline{x}} \times [\frac{1}{\overline{y}}, \frac{1}{\underline{y}}]$$

Note that in the case of division, $\overline{\underline{y}}$ must not contain zero. In the case of an interval containing zero, special care is needed, the same way as for real-number floating point arithmetic.

Three properties of intervals and interval arithmetic make it possible to precisely link the fallible observations of science and engineering to mathematics and floating-point arithmetic ([Wal97]):

1. Any contiguous set of real numbers (a continuum) can be represented by a containing interval,

2. Intervals provide a convenient and mechanical way to represent and compute guaranteed error bounds using fallible data,

3. All the important properties of infinite precision interval arithmetic can be preserved using finite precision numbers and directed rounding, which is commonly available on most computers (indeed, any machines supporting the IEEE 754 floating-point standard).

1.3 Why interval arithmetic?

There are usually three sources of error while performing numerical computations: rounding, truncation and input errors. In the following examples (taken from [CMN02]) we show how IA is meant to keep track of them.

- *rounding errors*:
  Consider the expression $f(x) = 1-x+\frac{x^2}{2}$ with $x = 0.531$, i.e. with 10^{-3} precision. Computing this expression with classical arithmetic gives the result $f(x) = 0.610$. Now, if we perform the computations using IA, we get

  $$f(x) = 0.469 + \frac{0.531^2}{2} \in 0.469 + \frac{[0.281, 0.282]^2}{2}$$

  and so

  $$f(x) \in 0.469 + [0.140, 0.141] = [0.609, 0.610]$$

  This guarantees that the true result is within the interval $[0.609, 0.610]$.

- *truncation errors*:
  We are now interested in a Taylor series of the exponential function:

  $$e^x = 1 + x + \frac{x^2}{2!}e^t$$

  where $t \in [0,x]$. For $x < 0$, $e^x \in 1 + x + \frac{x^2}{2!}[0,1]$. In particular, with $x = -0.531$, we get

  $$e^{-0.531} \in 1 - 0.531 + \frac{(-0.531)^2}{2!}[0,1]$$
  $$= 0.469 + [0.140, 0.141][0,1] = [0.469, 0.610]$$

  This example illustrates how IA keeps track both of rounding and truncation errors.

- *input errors*:
  Suppose that given data uncertainty our input is $x \in [-0.532, -0.531]$. If we evaluate the previous expression we obtain

  $$e^x \in 1 + [-0.532, -0.531] + \frac{[-0.532, -0.531]^2}{2}[0,1]$$

  $$= [0.468, 0.470] + \frac{[0.280, 0.284]^2}{2}[0,1]$$
  $$= [0.468, 0.470] + [0, 0.142] = [0.468, 0.612]$$

  This final example illustrates how IA can keep track of all error types simultaneously.

Moreover IA doesn't suffer from any restriction to class of functions that it can be applied to. Indeed Moore's fundamental theorem of interval arithmetic [Moo66] states that for any function f defined by an arithmetical expression, the corresponding interval evaluation function F is an *inclusion function* of f:

$$F(\overline{x}) \supseteq f(\overline{x}) \;=\; \{f(x) \;\mid\; x \in \overline{x}\}$$

Given a n-dimensional box B, i.e. product of n intervals, and an implicit function f we have a very simple and reliable rejection test for the box B not intersecting the image of the function f (e.g. surface or volume),

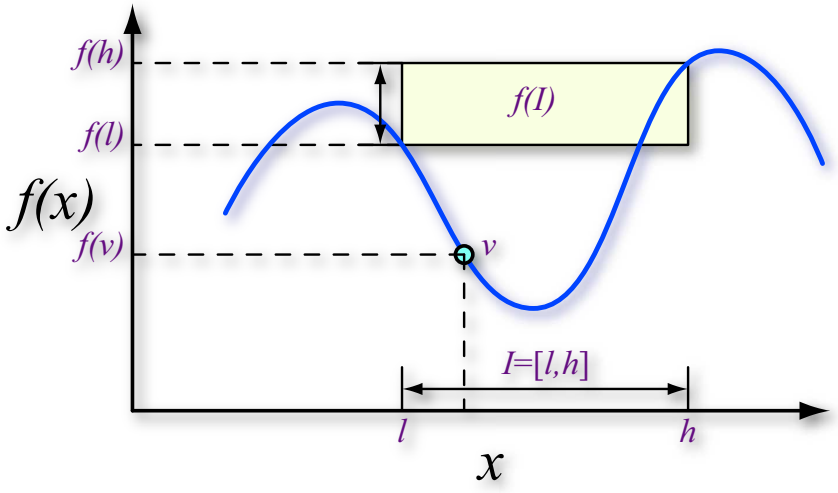
$$0 \notin F(B) \Rightarrow 0 \notin f(B)$$

This property can be used in ray casting or mesh extraction for identifying and skipping empty regions of space. Note, however, that although $0 \notin F(B)$ guarantees the absence of a root on an interval B, that the converse does not necessarily hold: one can have $0 \in F(B)$ without B intersecting f. When $F(B)$ loosely bounds the convex hull, as illustrated in Figure 1.1(b), IA makes for a poor (though still reliable) rejection test. This overestimation problem is a well-known disadvantage, and is fatal to algorithms relying on iterative evaluation of non-diminishing intervals.

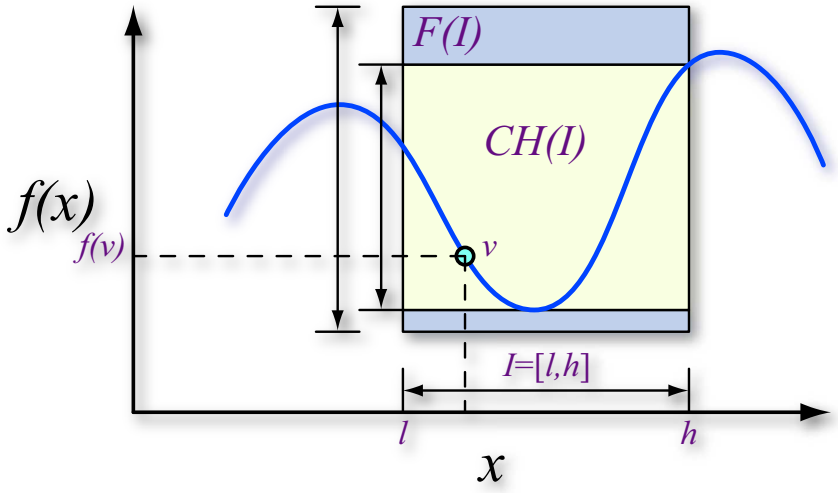
Fortunately, overestimation error is proportional to domain interval width; therefore IA guarantees convergence to the correct solution when interval domains diminish. This is the case in algorithms such as sweeping computation of hierarchically subdivided domains [Duf92, HB07], and ray casting algorithms involving recursive interval bisection [Mit90, CHMS00, KHH*07] as we will see in Part II. Though the overestimation problem affects the efficiency of these algorithms, recursive IA methods robustly detect the zeros of a function, given an adequate termination criterion such as a sufficiently small precision ε over the domain, or tolerance δ over the range.

Effectively, it suffices to implement a library of these IA operators, and substitute them for the real operators, producing an *interval extension* F. If each component operator preserves the inclusion property, then arbitrary compositions of these operators will as well. As a result, literally any computable function may be expressed as interval arithmetic. Some operations are ill-defined, such as empty-set or infinite intervals. However, these are easily handled in a similar fashion to real-number floating point arithmetic.

In the literature we often read "inclusion algebra" or "self-validated arithmetic" when referring to IA and the IA extension is often referred to as the *natural inclusion function*, but it is neither the only mechanism for defining an inclusion algebra, nor always the best. Particularly in the case of multiplication, it greatly overestimates the actual bounds of the range. To overcome this, it is necessary to represent intervals with higher-order approximations.



(a)



(b)

Figure 1.1: *Inclusion property of interval arithmetic.* (a) Floating point arithmetic is insufficient to guarantee a convex hull over the range. (b) IA is much more robust by encompassing all minima and maxima of the function within that interval. Ideally, $F(I)$ is equal or close to the bounds of the convex hull, $CH(I)$.

Chapter 2

Extensions of interval arithmetic

In this Chapter we discuss state-of-the-art extensions of IA, especially focusing on first-order arithmetics that can be alternatives to IA, for example when higher accuracy in the computations is needed.

2.1 Affine arithmetic

Affine arithmetic (AA) was developed by Comba & Stolfi [CS93] to address the bound overestimation problem of IA. Intuitively, if IA approximates the convex hull of f with a bounding box, AA employs a piecewise first-order bounding polygon, such as a parallelogram (Figure 2.1).

An affine quantity $\hat{x}$ takes the form $\hat{x} = x_0 + \sum_{i=1}^{n} x_i e_i$ where the $x_i, \forall i \geq 1$ are the *partial deviations* of $\hat{x}$, and $e_i \in [-1,1]$ are the *error symbols*. An affine form is created from an interval as follows:

$$x_0 = (\overline{x} + \underline{x})/2$$
$$x_1 = (\overline{x} - \underline{x})/2$$
$$x_i = 0, \;\; i > 1$$

and can equally be converted into an interval $\overline{\underline{x}} = [x_0 - rad(\hat{x}), x_0 + rad(\hat{x})]$ where the *radius* of the affine form is given as

$$rad(\hat{x}) = \sum_{i=1}^{n} |x_i|$$

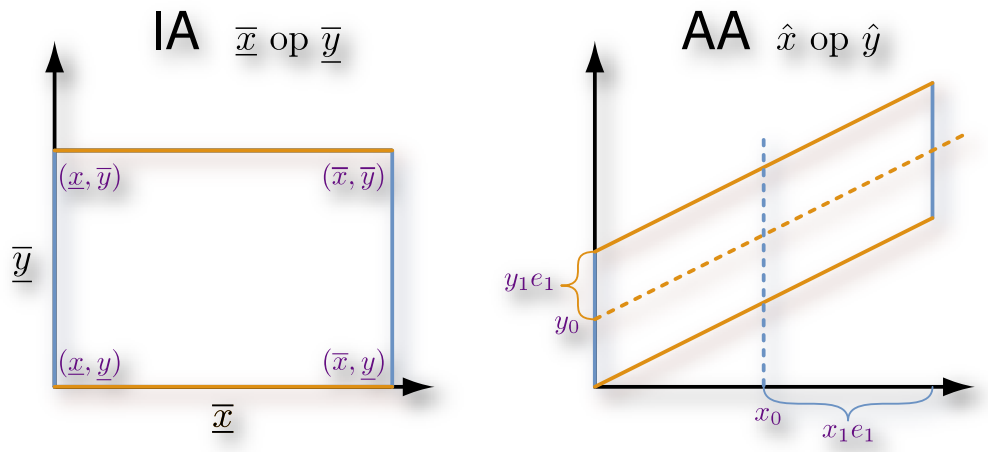


Figure 2.1: *Bounding forms of interval and affine arithmetic operations.*

.

Base affine operations in AA are as follows:

$$\mathbf{c} \times \hat{x} = \mathbf{c}x_0 + \mathbf{c}\sum_{i=1}^{n} x_i e_i$$

$$\mathbf{c} \pm \hat{x} = (\mathbf{c} \pm x_0) + \sum_{i=1}^{n} x_i e_i$$

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) \pm \sum_{i=1}^{n} (x_i \pm y_i) e_i$$

However, *non-affine* operations in AA cause an additional error symbol e_z to be introduced. This is the case in multiplication between two affine forms,

$$\hat{x} \times \hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_i y_0 + y_i x_0) e_i + rad(\hat{x}) rad(\hat{y})$$

Other operations in AA, such as square root and transcendentals, approximate the range of the IA operation using a regression curve – a slope bounding a minimum and maximum estimate of the range. These operations are also non-affine, and require a new error symbol.

The chief improvement in AA comes from maintaining correlated error symbols as orthogonal entities. This effectively allows error among correlated symbols to diminish, as

opposed to always increasing monotonically in IA. Figure 3.1(b) of Chapter 3 Section 3.1 shows how AA can be much more accurate than IA (by providing tighter bounds) e.g. in curve approximation.

Unfortunately, as the number of non-affine operations increases, the number of non-correlated error symbols increases as well. Despite computing tighter bounds, standard AA ultimately is inefficient in both computational and memory demands.

2.2 Extended affine arithmetic

Affine arithmetic is only one example of first-order interval-based approximation and this topic is still an active research area for finding a trade-off between accuracy and computational cost. Other popular first-order arithmetics are E. R. Hansen's generalized interval arithmetic [Han75], its centered form variant [Neu03] and first-order Taylor arithmetic [Neu03]. There are (infinitely) many possible extensions of IA or AA; here we will develop only one. In the following example we present the so-called AF1 formulation from Messine et al. [Mes02] in which, for some constant n, a truncated affine form is given as:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i e_i + x_{n+1} e_{n+1}$$

The arithmetic operations are given by:

$$\mathbf{c} \times \hat{x} = (\mathbf{c}x_0) + \sum_{i=1}^{n} \mathbf{c}x_i e_i + |cx_{n+1}| e_{n+1}$$

$$\mathbf{c} \pm \hat{x} = (\mathbf{c} \pm x_0) + \sum_{i=1}^{n} x_i e_i + |x_{n+1}| e_{n+1}$$

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i) e_i + (x_{n+1} + y_{n+1}) e_{n+1}$$

$$\hat{x} \times \hat{y} = (x_0 y_0) + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i) e_i + (|x_0 y_{n+1}| + |y_0 x_{n+1}| + \mathrm{rad}(\hat{x})\mathrm{rad}(\hat{y})) e_{n+1}$$

In practice we noticed that this formulation is far more efficient than pure AA, providing decent results even when using $n = 1$ or $n = 2$. Messine also developed an extension of AF1 denoted by AF2; for details see [Mes02].

Previous introduced interval-based approaches were all zero- or first-order. Note that there are also higher-order methods such as the quadratic form (QF) from Messine et al. [Mes02] (whose formulation becomes more and more complicated) or Taylor-based arithmetics (see Gavriliu's PhD thesis [Gav05]).

2.3 Discussion on IA-based techniques

After introducing these different interval algebras we might wonder which one should be used in practice. In this section we try to give a feeling for this question (based on experience) and provide pointers to existing IA-like implementations.

In practice we noticed that despite providing less accurate results, IA is much more robust than reduced AA and has better numerical stability. As previously mentioned IA is especially bad when computing interval products, in which case simple AA can double performance for the same computational cost. But for other operations, the difference might not always be that relevant. An interesting approach might be having a hybrid IA/AA technique which would always use the optimal one. Indeed we often have to make a trade-off in practice between accuracy and efficiency. In short, IA has a somehow linear convergence - which is considered slow in most applications - but is robust whereas AA algebras have a better accuracy at a more important computational cost and less stability than IA.

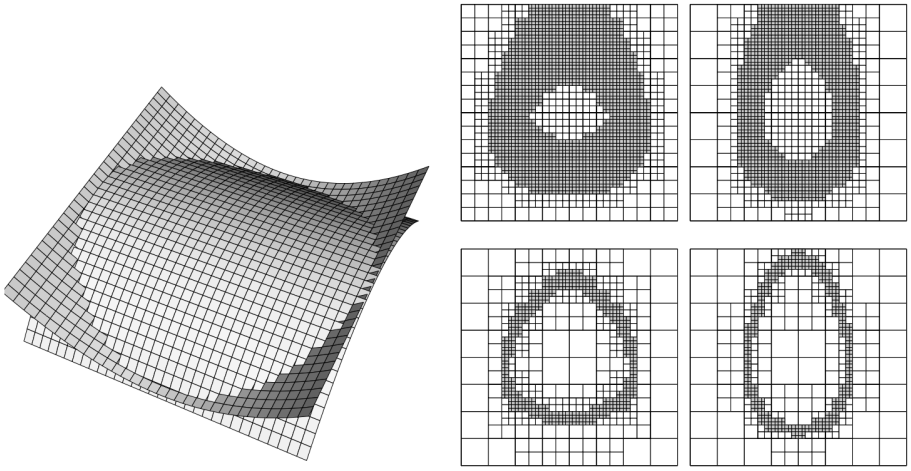
There are many IA-like libraries available and the most popular ones are written in C++, e.g. Filib++ [LTWVG*06] for Interval Arithmetic and LibAffa [GCH00] for Affine Arithmetic. Those libraries are convenient for experimenting with IA - as one basically needs to replace `float` or `double` by `interval` (of `aaform`) - but they can be slow as carrying a lot of unnecessary operations for the desired application; in this case, we might prefer to implement our own library, e.g. as for the interactive ray casting algorithm using IA [KHH*07] or its GPU extension [KHK*08]. Note that function derivatives are needed in a particular application, IA can be combined with automatic differentiation (AD) and thus evaluate the interval function and its interval derivative at the same time.

Chapter 3

Successful interval arithmetic applications in Computer Science

Successful IA-based algorithms abound. In this brief state of the art, we review only some of them by selecting four areas of computer science: pattern recognition, computational geometry, mesh extraction and finally ray casting.

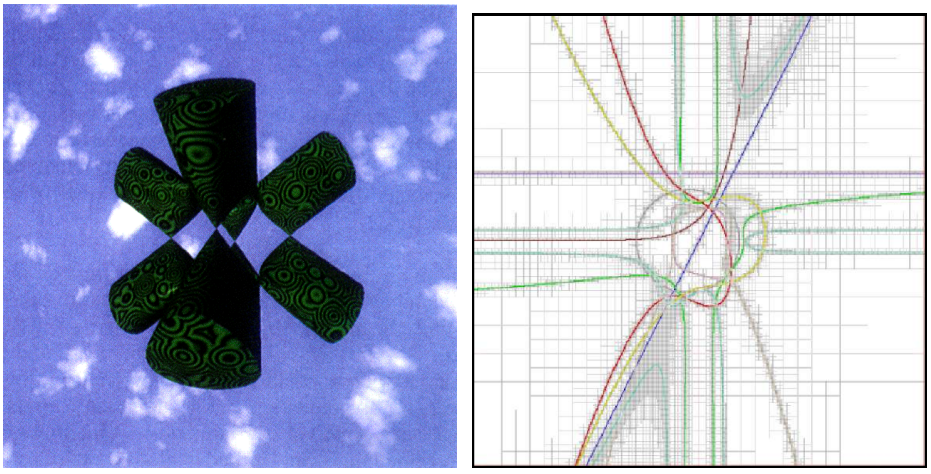
3.1 Pattern Recognition & Computational Geometry



3.1(a) 3.1(b)

Figure 3.1: *[dF96]: (a) Surface intersection using AA. (b) IA (top) versus AA (bottom).*

In computer vision, exploration of arrangements by subdivision methods has been used for computing globally optimal solutions to geometric matching problems under bounded error using interval arithmetic as a key ingredient in the algorithm [Bre03b]. In computational geometry, e.g. for computing the intersection of curves or surfaces, IA has been employed for robustly finding those roots. In [dF96] de Figueiredo applies AA for robustly intersecting parametric surfaces (Fig. 3.1(a)). He uses a quadtree decomposition of the domain for the output. He also compares IA and AA in performance (Fig. 3.1(b)) and comes to the conclusion that AA is better suited than IA for this particular task.



3.2(a) 3.2(b)

Figure 3.2: *(a) [Duf92]: CSG ray casting using IA.* (b) [HB07]: Arrangement of curves using IA.

Other computational geometry examples using interval arithmetic-based recursive subdivisions are CSG operations for implicit surfaces [Duf92] (see Figure 3.2(a)) and arrangements of implicit curves [HB07] (see Figure 3.2(b)). In their paper, Hijazi et al. provide a method for computing arrangements of implicitly defined curves. The new method for computing arrangements (introduced in Part II Chapter 4 Section 4.5) is an adaptation of methods successfully used for the exploration of large, higher dimensional, non-algebraic arrangements in computer vision. While broadly similar to subdivision methods in computational geometry, its design and philosophy are different; for example, it replaces exact computations by subdivision and interval arithmetic computations and prefers data-independent subdivisions. It can be used (and is usually used in practice) to compute well-defined approximations to arrangements, but can also yield exact answers for specific problem classes.

3.2 Mesh extraction

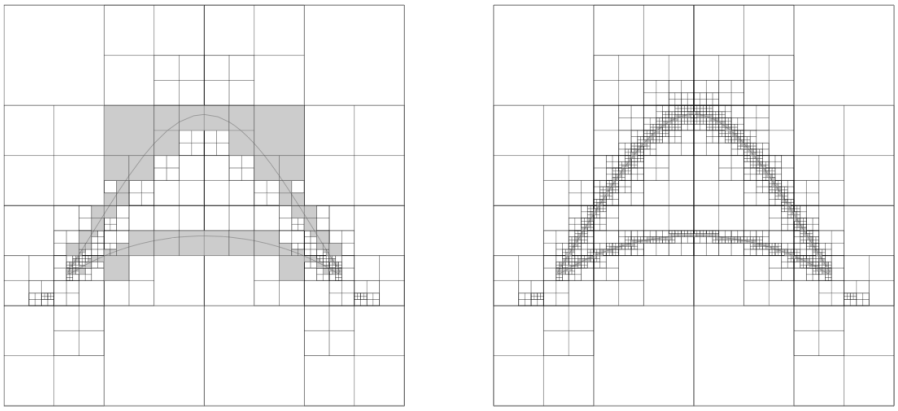


Figure 3.3: *[LOdF01]: Bicorn curve approximation using IA*. Left: using spatial adaption. Right: using geometrical adaption.

Lopes et al. [LOdF01] present an algorithm for computing a robust adaptive polygonal approximation of an implicit curve in the plane. The approximation is adapted to the geometry of the curve as the length of the edges varies with the curvature of the curve (see Figure 3.3). Robustness is achieved by combining interval arithmetic and automatic differentiation. This work has been extended to robust surface approximation by Paiva et al. [PLLdF06], leading to a dual marching-cube octree-based algorithm using IA and providing topological guarantees (given a certain precision). Figure 3.4 illustrates this concept: green regions are topologically guaranteed whereas red regions are uncertain; in this illustration, the precision is pixel-based.

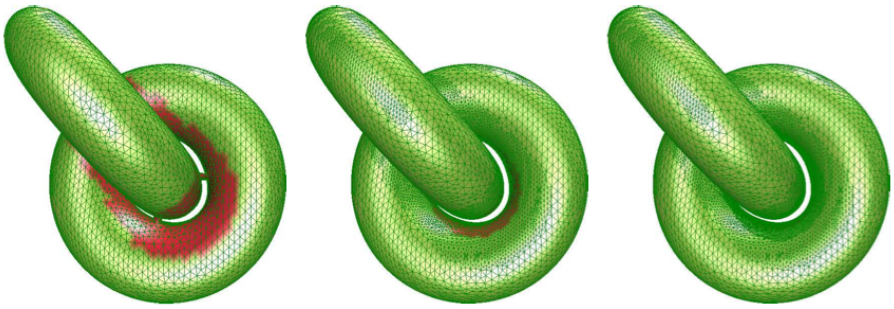
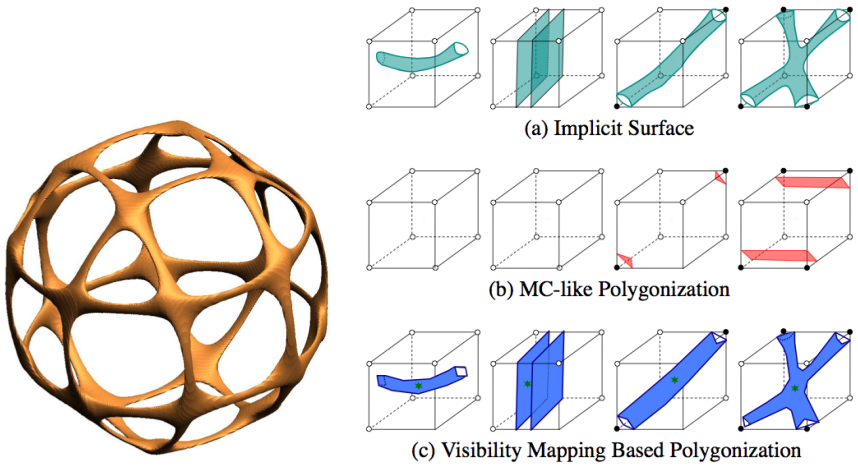


Figure 3.4: *[PLLdF06]: Linked tori approximation using IA*.

Another marching-cube like algorithm algorithm relying on IA and providing topological guarantees is from Varadhan et al. [VKZM06]. A decocube obtained with this technique is shown in Figure 3.5(a). On top of interval arithmetic, the method uses a visibility map and dual contouring for extracting the mesh and compares the results with classical marching cubes (MC) in Figure 3.5(b).



3.5(a) 3.5(b)

Figure 3.5: *[VKZM06]: (a) Decocube. (b) Marching cubes.*

3.3 Ray casting

3.3.1 A brief overview

There is a large number of IA-based ray tracers and here we only review some of them. A complete overview will be presented in Part II Chapter 5 Section 5.2. The first ones who introduced IA techniques for ray casting were Toth [Tot85] and Mitchell [Mit90].

In his survey [Mit91] Mitchell applied Moore's algorithm (see Figure 3.6(a)) for root-finding using IA in the context of ray casting. Indeed, ray casting often reduces to a root-finding problem [Mit91]. This concept is illustrated by Figure 3.6(b) showing how ray casting implicit surfaces reduces to solving a one-dimensional root-finding problem.

Since the 90s many other publications appeared in this area. Capriani et al. [CHMS00] combined interval bisection with various other iterative schemes, including the Interval Newton method. De Cusatis Junior et al. [dCJdFG99] used affine arithmetic to address the bound overestimation problem of pure interval arithmetic. Sanjuan-Estrada et al. [SECG03] compared performance of two hybrid interval methods with implementations of the Interval Newton and a recursive point-sampling subdivision method in the POV-Ray framework (see Figure 3.7). Heidrich & Seidel [HS98] used AA for ray casting procedural displacement shaders.

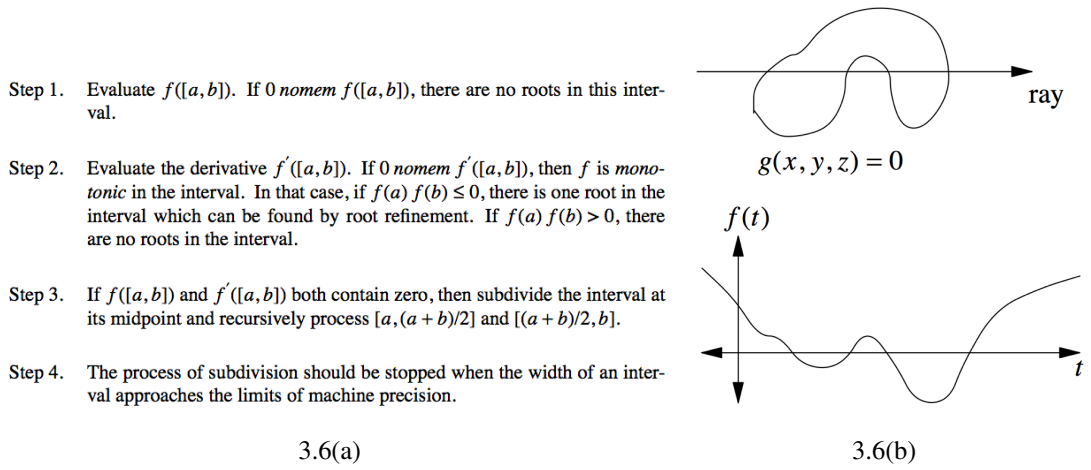


Figure 3.6: *[Mit91]: (a) Moore's root-finding algorithm. (b) Ray casting and root-finding.*

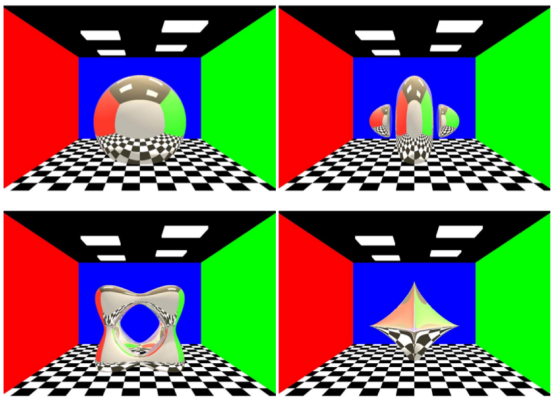


Figure 3.7: *[SECG03]: Up-left to down-right: sphere, Mitchell, tangle and superellipsoid.*

3.3.2 Recent results

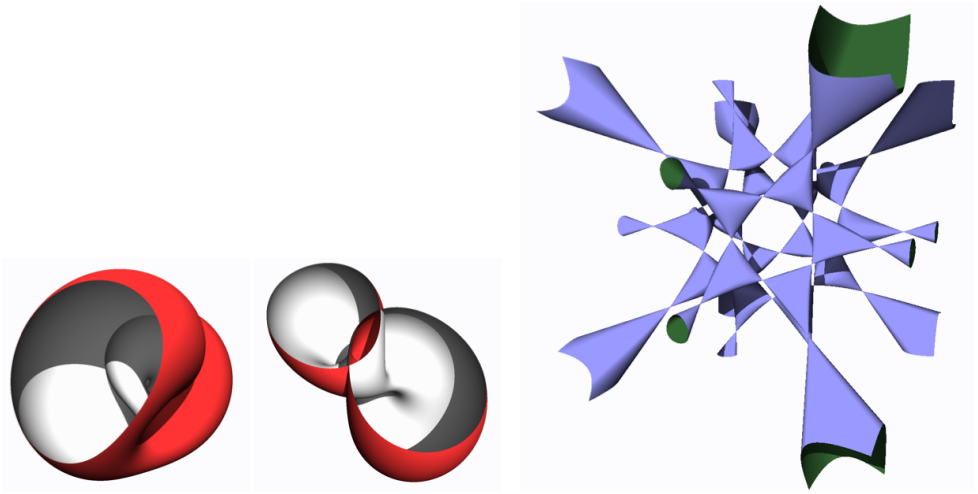
Recently, Gamito & Maddock [GM07] applied reduced affine arithmetic for ray casting implicit fractal surfaces. Though efficient, the proposed reduced affine arithmetic (RAA) method only preserves inclusion under specific circumstances and can only be applied to a certain class of functions. Figure 3.8 shows a procedural planet modeled using their technique.

Knoll & Hijazi et al. [KHH*07] applied interval arithmetic for interactively ray casting arbitrary implicit functions. This is the first time robustness and interactivity have been jointly achieved for ray casting implicits. This new algorithm will be described in Part II Chapter 6 Section 6.2 of this thesis. Figures 3.9(a) and 3.9(b) show selected implicits ray-

Figure 3.8: *[GM07]: Procedural modeling using RAA.*

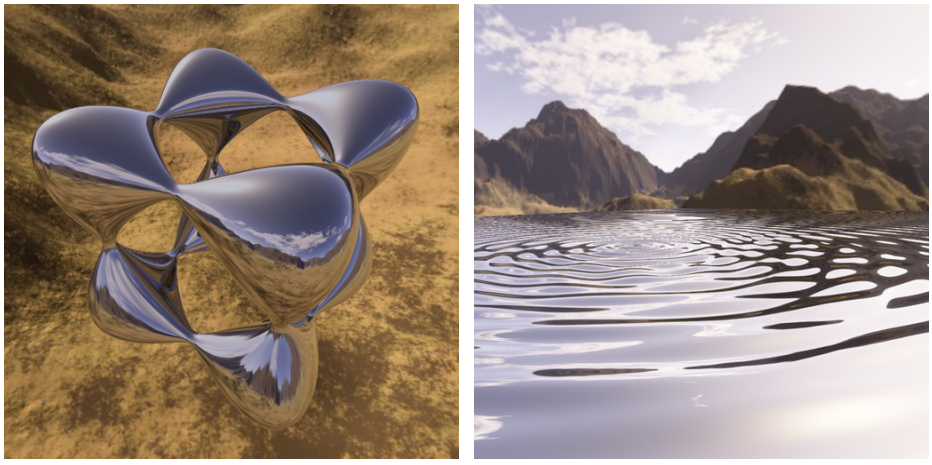
traced using Knoll & Hijazi et al.'s technique. In their paper the authors present a practical and efficient algorithm for interactively ray casting arbitrary implicit surfaces where IA is used both for robust root computation and guaranteed detection of topological features. In conjunction with ray casting, this allows for rendering literally any programmable implicit function simply from its definition. The proposed method requires neither special hardware, nor preprocessing or storage of any data structure. Efficiency is achieved through SIMD optimization of both the interval arithmetic computation and coherent ray traversal algorithm, delivering interactive results even for complex implicit functions. Because they neither pre-compute an explicit representation of the object, nor a physical acceleration structure in memory, they have great flexibility in rendering dynamically changing N-dimensional implicits. Examples of such morphing will be demonstrated in Part II Chapter 6 Section 6.4.4.

More recently, Knoll & Hijazi et al. [KHK*08] developed a new stackless ray traversal algorithm optimized for modern graphics hardware, and a correct inclusion-preserving reduced affine arithmetic (RAA) suitable for fragment shader languages. They also demonstrate multi-bounce effects, such as shadows, depth peeling and reflections, which are useful for visualizing complicated implicit functions. With their system they are able to render even complex implicits correctly, in real-time at high resolution. This new algorithm will be described in Part II Chapter 5 Section 7.2 of this thesis. Figures 3.10(a) and 3.10(b) illustrate their technique.



3.9(a) 3.9(b)

Figure 3.9: *[KHH*07]: (a) Klein bottle (4.0 fps). (b) Barth-sextic implicit (6.1 fps).*



3.10(a) 3.10(b)

Figure 3.10: *[KHK*08]: (a) The tangle with up to six reflection rays (44 fps). (b) Sinusoid procedural geometry for dynamic simulation of water (37 fps).*

Conclusion

As we demonstrated in this brief survey, interval-based techniques can be very useful in computer science, e.g. in computer vision, computational geometry, mesh extraction or ray-tracing. Ten years ago, the first ray-tracing algorithms using IA as a key ingredient appeared and took advantage of IA's inherent robustness and adaptivity by applying it to concrete problems. Though robust, many of those algorithms suffered from a lack of speed and thus implying little interest in the graphics community. More recently, with the increasing computational power of CPUs and GPUs, interval techniques are gaining attention as being now able to provide both fast and robust IA-based algorithms. We hope that the Computer Science community, especially in Computer Graphics & Visualization, will from now on tend to increasingly include IA techniques in their algorithms.

Part II

Features in scalar fields

Introduction

In this Part we consider features in scalar fields. The first features we have studied are intersections of arbitrary implicit curves. As input we have n implicit curves and we want to find as accurately and robustly possible their intersections and an abstraction of those curves. The problem of computing the arrangement of arbitrary implicit planar curves [HB07] was our first opportunity to experiment with subdivision methods together with interval arithmetic. Indeed all previous methods considered sweeping a line and their complexity increased with the complexity of the input curves. We had the idea of representing all the information using intervals, i.e. the starting domain (bounding box) and the curves, and subdivide until a certain precision. Our method doesn't make any assumption on the input and so doesn't restrict to a particular class of curves.

A different kind of feature that has taken our attention is the intersection between a ray and the image of an implicit function (e.g. surface in $3D$ or $4D$ hyper-surface). Following the same philosophy as for the arrangement of curves we have intended to design a 100% interval arithmetic-based method for ray casting arbitrary implicit objects. Robustness is one desired goal for this task and is achieved using interval techniques. More importantly we would like our algorithms to be at least interactive. Therefore we used the latest techniques and tools of the interactive ray casting and the graphics communities (together with Aaron Knoll) such as fast ray traversal, SSE optimizations or GPU programming. Thus we have been able to perform ray casting of arbitrary implicit functions, interactively on the CPU [KHH*07] and real-time using the latest graphics hardware [KHK*08]. With this framework we are able to render even the most difficult implicits in real-time with guaranteed topology (given a certain precision) and at high resolution.

Part II is our main contribution to the topic *Feature Based Visualization*. It consists of four chapters. In Chapter 4 we provide a state of the art for computing the arrangement of planar curves [Hij06], followed by our contribution, a new approach for computing the arrangement of arbitrary implicit planar curves [HB07]. Chapter 5 is a brief overview of implicit functions rendering methods, particularly focusing on ray casting ones. Chapter 6 details our first contribution to rendering implicits robustly and efficiently [KHH*07]. Chapter 7 extends this approach using the latest graphics hardware and demonstrates a real-time ray tracer of implicits [KHK*08].

Chapter 4

Computing arrangements of arbitrary implicit planar curves

Computing arrangements of curves is a fundamental and challenging problem in computational geometry as leading to many practical applications in a wide range of fields, especially in robot motion planning and computer vision. In this chapter we present first, a state of the art for computing the arrangement of planar curves considering various classes of curves; then, we review applications of arrangements; and finally, we present the CAPS algorithm, a new approach - inspired from computer vision - for computing the arrangement of arbitrary implicit planar curves. This new contribution has been done in collaboration with Thomas Breuel (DFKI, Kaiserslautern).

4.1 Introduction

Arrangements are subdivisions of Euclidean space created by multiple lower-dimensional surfaces and are widely used in robotics, computer graphics, molecular modeling, and computer vision. For survey papers on arrangements, see [Hal97, AS00, Hij06]. The output of an arrangement algorithm of planar curves is often the *arrangement graph*, the planar graph in which nodes correspond to intersections of curves and edges correspond to curve segments joining the nodes [BEW03].

The first algorithms for computing planar arrangements of lines were based on *sweeps*, an enumeration of the geometric structures encountered when a line–often parallel to the y-axis–is moved (swept) across the plane [BO79, EG89]. Recent results tend to generalize such methods to more general classes of curves such as conics/cubics [EKSW04], and algebraic curves [MS06]. Subdivision methods have also recently found increasing inter-

est for the computation of arrangements [LMP06, WM06, CdFC98]. Closely related to subdivision methods for the computation of arrangements are subdivision methods for the computation of implicit curves and/or surfaces [PV04, PLLdF06], intersections of Bézier curves [Yap06], intersections of surfaces [BK90], and ray casting [Mit90, Duf92].

In computer vision, exploration of arrangements by subdivision methods has been used for computing globally optimal solutions to geometric matching problems under bounded error [Bre92a, Bre03b].

The study of arrangements started with simple classes of geometric objects such as lines and - in current active research - tends to generalize to much more general classes such as algebraic objects of arbitrary degree or even completely arbitrary curves.

In Section 4.2 we present the current state of the art for computing the arrangement of planar curves. We first consider lines, which are already of great interest regarding the application side, for instance by dualizing problems involving points into line problems. We focus then on cubics - including the particular class of conics - before ending on latest results about algebraic curves and even more general curves.

4.1.1 Background

Following are some mathematical definitions for several types of curves which are studied in this Chapter. For an introduction on algebraic curves, see [Gri85].

Algebraic curve An *algebraic curve* is an algebraic variety of dimension one. A planar algebraic curve can be represented as $P(x,y) := \sum_i \sum_j p_{ij} x^i x^j = 0$, where $p_{ij} \in \mathbb{R}$. The degree d of an algebraic curve is $d = max_{i,j}(i+j)$.

For instance, the algebraic curve defined by the equation $2x^3y + 5xy^2 - y - 1 = 0$ is of degree 4. We are now able to give the definitions of conics and cubics:

Conics and cubics A *conic (or conic curve)* is an algebraic curve of degree at most 2 and a *cubic (or cubic curve)* is an algebraic curve of degree at most 3.

Named for the French mathematician Pierre Bézier, a Bézier curve is a curved line defined by mathematical formulas. Mathematically, we can formulate this as follows:

Bézier curve Given $n+1$ control points P_0, P_1, ..., P_n, the Bézier curve (of degree n) is defined by: $B(t) = \sum_{i=0}^{n} P_i b_{i,n}(t)$ with $t \in$ [0,1] and where $b_{i,n}$ are known as Bernstein polynomials.

4.1.2 Arrangement structures

Arrangement structures are of great interest in computational geometry as expressed in [Hal97] and [AS00] surveys. Early work focused especially on the arrangement of hyperplanes, with the particular case of $2D$ lines. Many efforts were invested to extend the study to more general objects, i.e. conics, cubics, and even arbitrary algebraic curves, being useful on the application side by matching real world applications.

4.1.2.1 What is an arrangement?

Given a collection C of geometric objects, the arrangement $A(C)$ is the decomposition of $\mathbb{R}^d$ into connected open cells of dimensions 0, 1, ..., d induced by C. Considering a collection of curves in $2D$, we have 0-cells (vertices - which are the intersection points), 1-cells (edges), and 2-cells (faces). An arrangement can be represented as a graph whose nodes are the 0-cells and its edges, the 1-cells. The graph gives two main information: the geometry (position of the cells), and the topology (connectivity of the cells) of the collection of the considered objects.

4.1.2.2 Why arrangements?

There are many practical applications of arrangements which will be studied in Section 4.3 but at this stage we only give a flavor to motivate our interest in arrangement structures. For instance, by considering the simple class of lines, many problems involving points can be transformed into problems involving lines, thanks to a duality transform. The task of determining whether any three points of a planar point set are collinear could be determined in $O(n^3)$ time by brute-force checking of each triple. However, if the points are dualized into lines, then this reduces to the question of whether there is a vertex of degree greater than 4 in the arrangement, which can be computed in $O(n^2)$ time.

4.2 Computing planar arrangements: a state of the art

For each considered class of curves we are first interested in counting the number of intersections. Indeed, the number of curves intersections will highly vary whether we consider lines or arbitrary algebraic curves. For n lines, we know that in a simple configuration we will have exactly $\frac{n(n-1)}{2}$ intersection points, which is also an upper bound for all possible configurations. For algebraic curves we can use Bezout's upper bound result which says that the number of intersection points of two algebraic curves of degrees respectively p

and q is bounded by the product of the degrees, i.e. $d = pq$. For n algebraic curves, this implies a combinatorial complexity (counting the cells) both quadratic in n, the number of curves, and d, the degrees of the curves.

One often meets the term "sweep" when referring to arrangements. Usually, sweeping a planar arrangement means sweeping with a vertical line (see Fig.4.1) and updating the event points where curves start, end or cross.

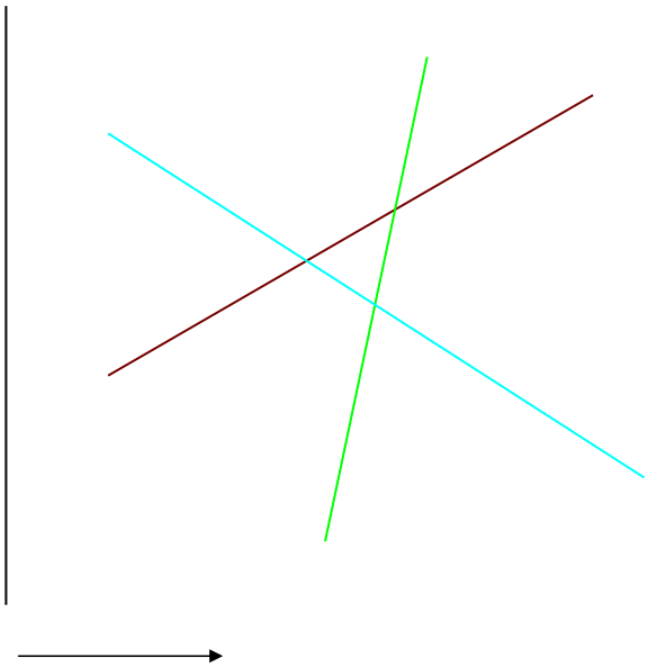


Figure 4.1: *Sweeping an arrangement with a vertical line.*

The sweep line doesn't necessarily need to be straight as demonstrated in [EG89] by the use of a topological line (Fig.4.2). One can even think about the term "sweep" more generally, as for example addressed by Breuel in [Bre92b]: Cass' algorithm in [Cas90] - called Critical Point Sampling (CPS) - is equivalent to a sweep of the arrangement generated by the feasible sets implied by all correspondences between model and image points, regarding computational geometry. Also, in addition to sweep line approaches, subdivision methods are emerging for the task of computing arrangements.

4.2.1 Lines

Arrangements of lines (more generally, hyperplanes) were already studied in the 19th century. Now - and since a long time - everything is known for the arrangement of straight

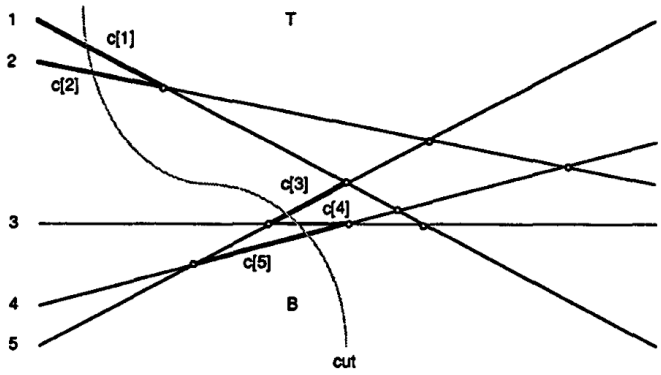


Figure 4.2: *[EG89]: Sweeping an arrangement with a topological line.*

lines. In particular, the arrangement of lines defined by rational numbers can be computed exactly, i.e. without any numerical error. Complete, exact, and efficient implementations for the arrangement of lines can be found in LEDA [MN00].

As already mentioned, the combinatorial complexity for lines arrangement is $O(n^2)$. Due to $\log n$ time operations the "naive" overall time complexity for computing an arrangement of n lines is $O(n^2 logn)$ which can be reduced to $O(n^2)$ time by using a topological line sweep as in [EG89]. This result is proved to be the best one can achieve for lines.

4.2.2 Cubics

A lot of work has been carried to study arrangements of non-linear objects, starting with conics and cubics. As conics are special cases of cubics and as similar approaches are used for computing their arrangement, only cubics will be discussed in our study. Also as our concern remains on planar curves, arrangements of $3D$ quadrics (i.e. quadric surfaces) will be skipped, despite their importance in the literature. Conics and cubics are very interesting classes as we know such curves quite well from the mathematical point of view and they already provide a wide range of applications.

In [ESW02] and [EKSW04] the authors use a Bentley-Ottmann sweep-line algorithm [BO79] to compute the arrangement. Their algorithm in [EKSW04] is complete (handles all possible degeneracies), exact (provides the mathematically correct result), and efficient (in terms of complexity).

4.2.3 (Semi-) Algebraic and Bézier curves

There are several papers studying the arrangement of semi-algebraic, algebraic and Bézier curves and methods to succeed in this goal vary considerably. In [Wol03] Wolpert presents an approach that extends the Bentley-Ottmann sweep-line algorithm - used in [EKSW04] for cubics - to the exact computation of the topology of arrangements induced by non-singular algebraic curves of arbitrary degrees. This paper overcomes the problem of detection and location of tangential intersection points of two curves using only rational arithmetic, and extending the concept of Jacobi curves [Wol02]. The result is an output-sensitive algorithm.

A different approach - but also for computing the arrangement of semi-algebraic curves - is presented in [MS06] where a vertical sweep line is being used together with a module that computes approximate crossing points of the input curves. The authors provide an implementation for semi-algebraic curves based on numerical equation solver. The running time of their algorithm is $O(Vlogn)$ for n curves with N crossings and k inconsistencies where $V = 2n+N+min(3kn, n^2/2)$. One claim of the paper is that the algorithm performs much better than the best known published results (1000 cubics in 100 seconds, versus 2000 seconds).

In [Yap06] Yap follows a new direction and presents the first complete subdivision algorithm for the intersection of two Bézier curves, possibly with tangential intersections. The adaptive algorithm uses a robust subdivision scheme based on geometric separation bounds, using a criterion for detecting non-crossing intersection of curves, and avoids manipulation of algebraic numbers and resultant computations.

In [WM06] Wintz et al. describe a new subdivision method to construct the arrangement of implicit planar algebraic curves and provide an incremental dynamic algorithm maintaining the solution of the problem as the input objects are inserted, without preliminary knowledge on the input data. The subdivision scheme is based on the bounding boxes of the input data - rather than using a classical hierarchical quadtree - keeping though advantage of quadtrees' adaptivity feature. The method combines the multivariate Bernstein's basis with Descarte's Law of Sign and uses an algebraic criterion to decide whether further subdividing a box. Experiments have been run on the algebraic modeling platform AXEL using the algebraic computation library SYNAPS [MPT*07]. The technique proved reliable and can be extended to higher dimensions and different kinds of objects.

4.2.4 Arbitrary curves

Little work has been done for computing the planar arrangement of arbitrary planar curves as they are unpredictable objects and not well known mathematically. Nevertheless, to our

knowledge, there have been two attempts in this direction using a divide-to-conquer interval arithmetic-based approach. In [CdFC98] the authors are interested in determining the exact topological adjacency structure of the planar subdivision induced on a rectangle by a set of curves given in implicit form. They use a recursive method based on estimates provided by interval arithmetic, together with conditions guaranteeing the topological correctness of the arrangement.

In [HB07] the authors present a method to compute the planar arrangement of implicitly defined curves using an interval arithmetic-based recursive algorithm. In their method, interval arithmetic is used both for reliable numerical computations and as an integral part of the search. They use an adaptive subdivision of the domain: interval arithmetic is used to reliably classify each rectangle according to whether it is empty, contains a curve, or contains multiple curves and/or intersections. The resulting decomposition is used to construct a topologically well-characterized representation of the arrangement together with algebraic representations of the cell boundaries. Their algorithm also generalizes to higher dimensional spaces.

4.3 Applications of arrangements

This section is inspired from the survey of Agarwal & Sharir [AS00], as being the reference for this topic. The reader is invited to consult the survey for details. The following non-exhaustive list of applications involves planar arrangements: range searching, transversals, geometric optimization (slope selection, distance selection, segment center, minimum-width annulus, geometric matching, center point, Ham sandwich cuts) and robotics. We develop some of them.

4.3.1 Range searching

Geometric range searching [AE98] is the problem of:

> Preprocessing a set S of n points in $\mathbb{R}^d$, so that all points of S lying in a query region can be counted quickly.

Range searching has important applications in Geographic Information Systems (GIS), computer graphics, spatial databases, and time-series databases. Range searching and arrangements are strongly related as point location in hyperplane arrangements can be used for range searching.

Mathematically, we can formulate this as follows: By defining the dual of a point $p = (a_1, ..., a_d)$ to be the hyperplane $p^* : x_d = -a_1x_1 - ... - a_{d-1}x_{d-1} + a_d$, and the dual of a hyperplane $h : x_d = b_1x_1 + ... + b_{d-1}x_{d-1} + b_d$ to be the point $h^* = (b_1, ..., b_d)$, then p lies above h if and only if the hyperplane p^* lies above the point h^*. Hence, halfspace range searching has the following equivalent "dual" formulation: Preprocess a set Γ of n hyperplanes in $\mathbb{R}^d$ so that the hyperplanes of H lying below a query point can be reported quickly. Using the point-location data structure for hyperplane arrangements provided in [Cha93], the level of a query point can be computed in $O(logn)$ time using $O(\frac{n^d}{log^d n})$ space.

4.3.2 Geometric optimization

In this subsection we focus on geometric optimization problems and how they are related to arrangements. As we will see, the area of geometric optimization is a natural extension and a good application area of the study of arrangements. We examine a sample of them starting with slope selection.

- slope selection:

  > Given a set S of n points in $\mathbb{R}^2$ and an integer k, find the line with the kth smallest slope among the lines passing through pairs of points of S.

  If points of S are dualized to a set Γ of lines of $\mathbb{R}^2$, the problem becomes that of computing the kth leftmost vertex of the arrangement $A(\Gamma)$.

- minimum-width annulus:

  > Compute the annulus of smallest width that encloses a given set of n points in the plane.

  This problem arises in fitting a circle through a set of points in the plane, though involving arrangements.

- geometric matching:

  > Given two sets S_1 and S_2 of n points in the plane, compute a minimum-weight matching in the complete bipartite graph S_1xS_2, where the weight of an edge (p,q) is the Euclidian distance between p and q.

  One can use the underlying geometric structure of these graphs in order to obtain faster algorithms than those available for general abstract graphs. In term of complexity, geometric matching problems can be seen as sweeping or exploring a geometric arrangement generated by constraint sets [Bre92b] [Bre03a].

- Ham sandwich cuts:

  > Let $S_1, S_2, ..., S_d$ be d sets of points in $\mathbb{R}^d$, each containing n points, where n is assumed being even. A *ham sandwich cut* is a hyperplane h so that each open halfspace bounded by h contains at most $n/2$ points of S_i, for $i = 1, ..., d$.

  It is known [Ede87] that such a cut always exists. Let Γ_i be the set of hyperplanes dual to S_i. Then the problem reduces to computing a vertex of the intersection of $A_{n/2}(\Gamma_1)$ and $A_{n/2}(\Gamma_2)$, i.e. involving arrangements.

4.3.3 Robotics

Motion planning for a robot system has been a major motivation for the study of arrangements. The problem can be seen as:

> Let B be a robot system with d degrees of freedom, which is allowed to move freely within a given two- or three-dimensional environment cluttered with obstacles. Given two placements I and F of B, determine whether there exists a collision-free path between these placements.

This problem reduces to determining whether I and F lie in the same cell of arrangement of the family Γ of "contact surfaces" in $\mathbb{R}^d$, regarded as the configuration space of B. Other problems in robotics that have exploited the theory of arrangements to lead to efficient algorithms include assembly planning, fixturing, micro electronics mechanical systems (MEMS), path planning with uncertainty, and manufacturing.

4.4 Brief summary

Computing arrangements has been of great interest for researchers starting in the 19th century with the study of lines and many results already arised from this simple class of curves. Naturally came the curiosity of studying non-linear objects such as conics and cubics, as quite well understood mathematically, which concretized e.g. in [EKSW04].

In current research, people are also interested in computing the arrangement of algebraic curves and arbitrary ones. We can distinguish between three main approaches to overcome this problem: an approximate method [MS06] based on numerical equation solver; an exact method [Wol03] using rational arithmetic; and finally subdivision schemes where

Yap provides a complete algorithm for intersecting two Bézier curves [Yap06], Wintz et al. [WM06] compute the arrangement of implicit planar algebraic curves based on algebraic criterion, [HB07] et al. provide an output-sensitive algorithm based on interval arithmetic for computing the arrangement of arbitrary planar curves.

This interest in arrangements is motivated by a wide range of real world applications and therefore a need of more general methods being both theoretically and practically well-defined. There are still many open problems as addressed in [Yap06] and a compromise is to be found between exactness and efficiency as the complexity of the input data increases.

4.5 A new algorithm for computing arrangements

This new contribution has been done in collaboration with Thomas Breuel (DFKI, Kaiserslautern); see Hijazi et al.[HB07].

This section describes a new algorithm (CAPS) for the computation of exact or approximative arrangement graphs of a collection of implicitly defined curves in the plane using a subdivision method and interval arithmetic; only the static case is considered. Our method for computing arrangements is an adaptation of methods successfully used for the exploration of large, higher dimensional, non-algebraic arrangements in computer vision. While broadly similar to subdivision methods in computational geometry, its design and philosophy are different; for example, it replaces exact computations by subdivision and interval arithmetic computations and prefers data-independent subdivisions. It can be used (and is usually used in practice) to compute well-defined approximations to arrangements, but can also yield exact answers for specific problem classes.

4.5.1 The CAPS algorithm

In this section, we present the CAPS (Curves Arrangements by Planar Subdivisions) algorithm which computes the arrangement of curves, i.e. a subdivision of the plane that consists of vertices (0-cells), edges (1-cells), and faces (2-cells).

We assume that curves are defined implicitly using equations $f_i(x,y) = 0$ and that their inclusion functions are convergent [Moo66].

An *implicit curve* is the set of zeroes of a function $f : \Omega \subseteq \mathbb{R}^2 \to \mathbb{R}$ (where Ω is an open subset of $\mathbb{R}^2$). Evaluations of f will be carried out in terms of interval arithmetic, so $z = f_{[\,]}(x,y)$ (sometimes written as just $f(x,y)$) stands for $[\underline{z},\overline{z}] = f_{[\,]}([\underline{x},\overline{x}],[\underline{y},\overline{y}])$. Here, $f_{[\,]}(x,y)$ is a natural inclusion function [Moo66] corresponding to a real-valued function $f(x,y)$; it satisfies $f_{[\,]}(x,y) \supseteq \{z : z = f(\xi,\eta), \xi \in x, \eta \in y\}$, and any sufficiently well-

behaved f converges towards the real value as its argument intervals shrink. Note that f has a zero within a box $[\underline{x},\overline{x}] \times [\underline{y},\overline{y}]$ only if $0 \in [\underline{z},\overline{z}] = f_{[\,]}(x,y)$. Our general approach towards exploring the arrangement will be to evaluate f on nested families of intervals, excluding boxes from further consideration if we can prove that they do not contain zero.

The input of the CAPS algorithm consists of a collection of implicit curves and an initial bounding box (optionally, additional curves may be added for handling intersections at infinity), and recursively invokes the `classify` function. This function then computes the set of curves that might cross the bounding box.

If provably none of the curves do, the function does not recurse further. If there is exactly one candidate that may intersect the box, the algorithm attempts to prove that this candidate actually intersects the box (below), and, depending on the result, marks the box as either empty, containing one curve, or indeterminate. If there are more than one candidate curve intersecting the box, and the algorithm has not reached the maximal recursion depth, the algorithm subdivides the box and recurses; subdivision is by simple bisection of each dimension of the bounding box (more complex or data-dependent schemes are possible but do not seem to improve performance in preliminary experiments). If the search has reached the maximum recursion depth, it does not subdivide further and instead terminates and marks the box as indeterminate.

Proving that a box contains zero curves can be done simply by demonstrating that $f_{[\,]}(x,y)$ does not contain zero for any of the curves. In order to determine that a curve actually intersects a box, it is not sufficient to show that $f_{[\,]}(x,y)$ contains zero for the box; rather, we need to show that the real-valued function $f_{[\,]}(x,y)$ actually changes sign somewhere within the box. We can do this by identifying a subregion where the function is provably positive ($\underline{z} > 0$) and another subregion where the function is provably negative ($\overline{z} < 0$). Therefore, the algorithm recursively subdivides the box until it either finds that all subregions are strictly positive or strictly negative (non-intersecting), until it obtains an example of a strictly positive and a strictly negative region (intersecting) as shown in Figure 4.3, or until the box size is below a threshold (indeterminate).

As part of its search, the box classification algorithm constructs a quad-tree decomposition of the original box, with each leaf in the quad-tree labeled as containing zero curves, one curve, or indeterminate (see Figure 4.4).

For each leaf, the algorithm also records the curves that intersect or potentially intersect the corresponding box. In order to compute the actual arrangement graph, we view this labeled quad-tree as a subdivision of the plane and apply a standard region labeling and region adjacency graph algorithm for quad-trees. The label for each quad-tree node consists of the set of curves for which the CAPS algorithm could not prove that the curve fails to fall outside that node. In this process, all adjacent quad-tree nodes with the same label (i.e., the same set of curves) are grouped together into regions (for the purposes of this ex-

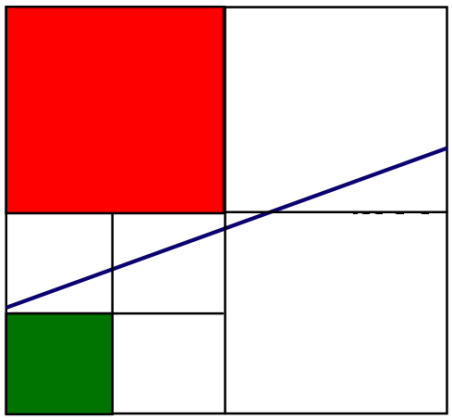


Figure 4.3: *Subdivision test within a box to determine whether it contains exactly one curve or not.*

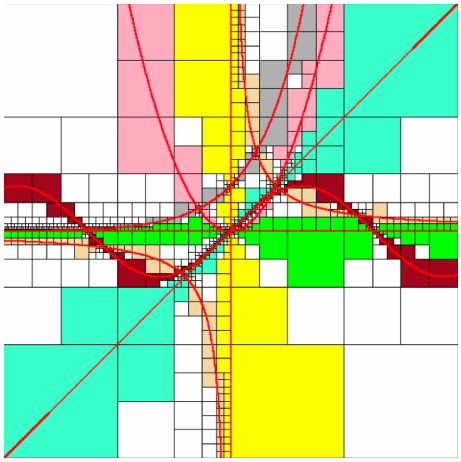


Figure 4.4: *Quad-tree decomposition of the original box.*

position, assume that no curve passes exactly through the corner of a quad-tree node and define connectedness analogous to four-connectedness in image processing). The labeled regions are transformed into a graph using the following general approach (some special cases omitted):

- each region containing zero curves is omitted (it is part of the dual graph),
- each region labeled with exactly one curve is transformed into an edge,
- each region labeled with more than one curve is transformed into a vertex and associated with all its adjacent regions representing curves.

Pseudo-code of the CAPS algorithm is illustrated in Figure 4.5.

Optionally, we could additionally compute, for each region containing exactly one curve, a polygonal or spline approximation of the curve passing through that region, as shown in [SF92].

In general, the graph computed by this method will not be the arrangement graph as commonly defined in computational geometry. Let us consider some of the differences.

Most importantly, if the input curves permit self-intersections, these self-intersections may or may not not be present in the computed graph. In practice, self-intersections are often *a priori* impossible because of the nature of the input curves. Furthermore, in many applications, computation of self-intersections are not required, and we can view the graph computed by the CAPS algorithm as a well-defined transformation of the arrangement graph. When self-intersections are both possible and desired, the CAPS algorithm can be modified to compute them by introducing an additional subdivision scheme (not described here).

For regions containing exactly two curves, there are many different ways in which these curves could meet inside the region. Contacts and even numbers of intersections are distinguished from odd numbers of intersections based on the topology of the neighboring regions. The considerations for distinguishing contacts and single intersections on the one hand, and multiple intersections on the other, are analogous to self-intersections: they can usually be excluded based on the class of curves under consideration, they may not be of interest, and/or they could be calculated explicitly using additional subdivision techniques.

Subject to these considerations, we can claim a number of different, exact results like the following: *If the CAPS algorithm is applied to curves that do not self-intersect and have no multiple intersections, and if it yields a graph in which all nodes correspond to regions containing no more than two curves and in which no contacts occur, then the graph is an arrangement graph for the curves.* Note that we can control whether the CAPS algorithm yields such graphs through when we terminate the search, and we can find analogous statements about CAPS-like algorithms modified to cope with self- and multiple intersection.

For many practical applications, however, approximations of the following form are of greater interest: *If the CAPS algorithm expands nodes to a terminal size of $\frac{\delta}{2}$, it yields an arrangement graph for an arrangement of curves obtained by continuously distorting the original arrangement by no more than δ at each point.* (This claim, of course, would require a proof; not shown). We can view this as a δ-*weak* solution [GLS88] for the arrangement problem, meaning that it represents a solution that is obtainable from the true solution through a small geometric perturbation (note that this is not necessarily the same as a small perturbation of the curves themselves). In practical applications (like geometric matching) weak solutions for well-defined accuracies have turned out to be

```
function classify(box, curves)
  Input : box, implicit curves; Output : quadtree
  create a node associated to a box
  if the box is empty
    mark node as 0 and return node
  if the box contains at most one curve
    if it contains exactly one curve(index)
      mark node as 1, set the index and return node
    else
      mark node as 0 and return node
  if the box is below the threshold
    return node
  call compute_arrangement on each child box
  record the result
  return node
```

Figure 4.5: *Pseudo-code for the CAPS algorithm.*

sufficient, and this is how these methods have been used in practice [Bre03a].

4.5.2 Complexity analysis

In this Section, we consider some informal analyses of the average-case and worst-case combinatorial complexity of the CAPS algorithm in terms of the output complexity of different classes. As a measure of complexity, we use the tree size. The threshold where we stop splitting is called ε and we denote by L the size of the initial box's side, and d, the depth of the quadtree, given by $d = \log_2(L\varepsilon^{-1})$.

4.5.2.1 The linear case

Let us say that lines l_1 and l_2 are ε-close if one of the two l_1 and l_2 are parallel and $distance(l_1, l_2) \leq \varepsilon$ or if l_1 and l_2 intersect and $angle(l_1, l_2) \leq \varepsilon$.

Close lines trigger the worst-case behavior of the CAPS algorithm, since it requires exploration of large number of boxes. If two lines are ε-close, we need $O(\varepsilon^{-1})$ boxes to cover the gap, and $O(n\varepsilon^{-1})$ for n lines, and this dominates the tree.

Of course, in the linear case, we can easily use exact methods for determining whether the lines actually intersect. Alternatively, we can use recursive interval arithmetic methods, but searching, say, for an intersection of the two lines directly rather than by covering

the gap. Nevertheless, the analysis of the linear case provides a good introduction for the analysis of the more general case.

In order to get some idea of the average case, let us classify the relationships in which two lines l_1 and l_2 can appear: there are no intersection inside the bounding box explored by the search (C_1), there is one intersection with $angle(l_1,l_2) > \varepsilon$ (C_2), and there exist ε-close lines (C_3). We assume that the expected value E of the lines' configuration C gives us the average-case complexity. If we denote by c_i the previously listed configurations' complexity and by p_i their associated probabilities, we have $E(C) = \sum_i p_i c_i$. If we are in case C_1 (meaning no intersection) the tree will be very small and the algorithm will terminate very quickly, in constant time; so, c_1 is $O(1)$. For C_2, the complexity is the size of the tree, given by $d = \log_2(L\varepsilon^{-1})$, i.e. $O(\log(\varepsilon^{-1}))$. Finally, c_3 is in $O(\varepsilon^{-1})$ as previously shown in the worst-case scenario.

Now that we were able to express the complexity of each situation, we would like to know how likely it is to happen and therefore compute its probability. We first examine the one-dimensional case: if parameters are within the interval $[-K,K]$, equally likely, we get, for the two random variables k_1 and k_2 : $p(k_1) = p(k_2) = \frac{1}{2K} = O(1)$ and we can then prove that $p(|k_1 - k_2| < \varepsilon) = \frac{\varepsilon}{2K}$. Considering the case of lines in $2D$, we need to define a distribution. Within the bounding box, we represent a line by two parameters: a point $M = (x,y)$ and an angle α between 0 and 2π. The same way as in the one-dimensional case, if we have random variables k_1 and k_2 in the box $[-K,K] \times [-K,K]$ (equally likely) we obtain $p(k_1) = p(k_2) = \frac{1}{4K^2} = O(1)$. This leads us to $p_1 = p_2 = O(1)$.

We are now interested in the probability that the ε-close case occurs. For two lines (M,α_1) and (N,α_2) we have $p(\mid \alpha_1 - \alpha_2 \mid \leq \frac{\varepsilon}{2L})\frac{\varepsilon}{2L} = \frac{\varepsilon}{4\pi L}\frac{\varepsilon}{2L} = O(\varepsilon^2)$. We finally get that probabilities p_1 and p_2 are $O(1)$ and p_3 is $O(\varepsilon^2)$ and thus $E(C) = \sum_i p_i c_i = O(1)O(\log(\frac{1}{\varepsilon})) + O(\varepsilon^2)O(\frac{1}{\varepsilon}) = O(\log(\frac{1}{\varepsilon}))$. This analysis has been done for only two lines. For n lines we have an upper bound of $\frac{n(n-1)}{2} = O(n^2)$ intersections and though conclude that the average-case combinatorial complexity for n lines is $O(n^2 \log(\frac{1}{\varepsilon}))$. Notice that despite the term n^2 in the average-case complexity, the worst-case complexity is much worse than the average one because of the term $\frac{1}{\varepsilon}$.

4.5.2.2 Algebraic curves and other families

A planar algebraic curve can be represented as $P(x,y) := \sum_{i,j} p_{ij} x^i x^j = 0$, where $p_{ij} \in \mathbb{R}$ and its degree d is defined by $d = max_{i,j}(i+j)$. Bézout's theorem tells us that for two algebraic curves of degrees p and q respectively, the number of intersection points of those two curves is bounded by the product of the degrees, i.e. $N = pq$.

If there are no ε-close curves and suppose the problem locally linearized, the combina-

torial complexity of two algebraic curves of degrees respectively p and q would be in $O(pq\log(\frac{1}{\varepsilon}))$. Thus, for n algebraic curves the complexity is $O(K_n\log(\frac{1}{\varepsilon}))$ where K_n is the result of summing up two by two all possible Bézout bounds of the input algebraic curves, which is both quadratic in n and the degrees of the curves: $K_n = \sum_{i\neq j} d_i d_j$ where d_i is the degree of the ith input algebraic curve and (i, j) are all the possible pairs within $1, ..., n$.

We can now ask what the probability of the ε-close configuration is on average. Therefore, consider a mapping from the product of the parameter spaces of all input curves to a given bounding box, sub-space of $\mathbb{R}^2$. We wish to get some idea of how frequent close position and general position cases are, since the non-intersecting close position cases are particularly hard for our approach. There are three cases we need to distinguish: an intersection with an angle larger than a fixed epsilon, contacts, or close, non-intersecting curves.

Contacts exist only for a set of parameter values with measure zero. Intersections at large angles, and ε-close approaches, on the other hand, exist for parameter regions with finite measure (i.e., greater than zero). However, if the mappings from parameter space to curves is sufficiently smooth, these parameter regions will still have a small measure for small ε, so that for sufficiently smooth distributions of input problems, the probability of computationally costly cases of arrangements of curves is low.

The above arguments are merely a sketch of a possible average case analysis of the complexity of CAPS-style algorithms for classes of curves or surfaces; a detailed analysis remains to be carried out.

4.5.3 Experimental results

Class of curves	Quadtree size	Runtime (s)
Lines (5)	12097	0.8
Algebraic curves (5)	32875	1.3
Degenerate (7 lines)	19546	1.9
Degenerate (5 polynomials)	251440	2.8
Trigonometric functions (3)	12271	0.2
Inverse Sine (2)	20800	12.6
Arbitrary curves (10)	78736	7.5

Table 4.1: *Computation time for the quad-tree structure (C++, Xeon 3.6Ghz) and different families of curves.* Parameters used were $\varepsilon = 10^{-6}$ and an initial bounding box of $[-10, 10] \times [-10, 10]$.

Running times for the algorithm are shown in Table 4.1, corresponding to curves of Figure 4.6. Moreover, Figure 4.7 shows the same curves by emphasizing the subdivision process. We can see that the algorithm performs well for straight lines and polynomials, and can handle difficult curves such as inverse sine.

4.6 Discussion

The original motivation for developing CAPS-like methods was the solution of problems in computer vision for which sweep methods were not practical: subdivisions of $\mathbb{R}^4$ or $\mathbb{R}^6$ based on implicitly defined curves involving polynomials and trigonometric functions, and involving often thousands of surfaces. Section 4.5 has described an application of those ideas to the problem of computing arrangements of implicitly defined curves, permitting us to compare CAPS-style approaches with commonly-used approaches from computational geometry (including subdivision and approximate methods developed in the context of computational geometry).

First, while CAPS can be used for exact computations for some problem types and instances, CAPS can also be used to obtain well-defined approximate (δ-weak) solutions through early termination. Such solutions are often sufficient for practical applications and yield well-characterized approximations for curves for which no exact methods are known or feasible.

CAPS is designed to use floating point arithmetic instead of exact computations, even when exact computations are possible (say, for algebraic curves), but can still return well-defined solutions through the use of interval arithmetic. Either approach can be used for computing exact or approximate arrangement graphs; which approach is preferable in practice depends on the domain and remains to be determined.

For practical applications, the combination of the CAPS-approach of a strict hierarchical exploration with faster intersection tests for pairs of curves may be desirable. Initial experiments (not shown) suggest that the use of such exact tests results in speedups for simple cases (e.g., linear), but may be more costly (and is often simply impossible) for more difficult cases (e.g., algebraic or trigonometric).

Another possible application of the CAPS could be for determining flex points of mixtures of Gaussian curves, i.e. finding the zeroes of the second derivative. Indeed, our algorithm easily handles such input. Our experience with CAPS-like algorithms and design choices suggests that they are more efficient and practical than traditional algorithms for computations involving arrangements in important cases. We would like to remind the reader that the original motivation for CAPS-style algorithms was that algorithms from computational geometry for computations involving arrangements were either too slow,

or simply not applicable to the kinds of arrangements encountered in computer vision.

We hope that this research will be only the first step in a more careful exploration of the similarity and differences between CAPS-like algorithms and algorithms developed in the traditional framework of computational geometry, and that it may lead to reconsideration of some design choices and assumptions often made in geometric algorithms research.

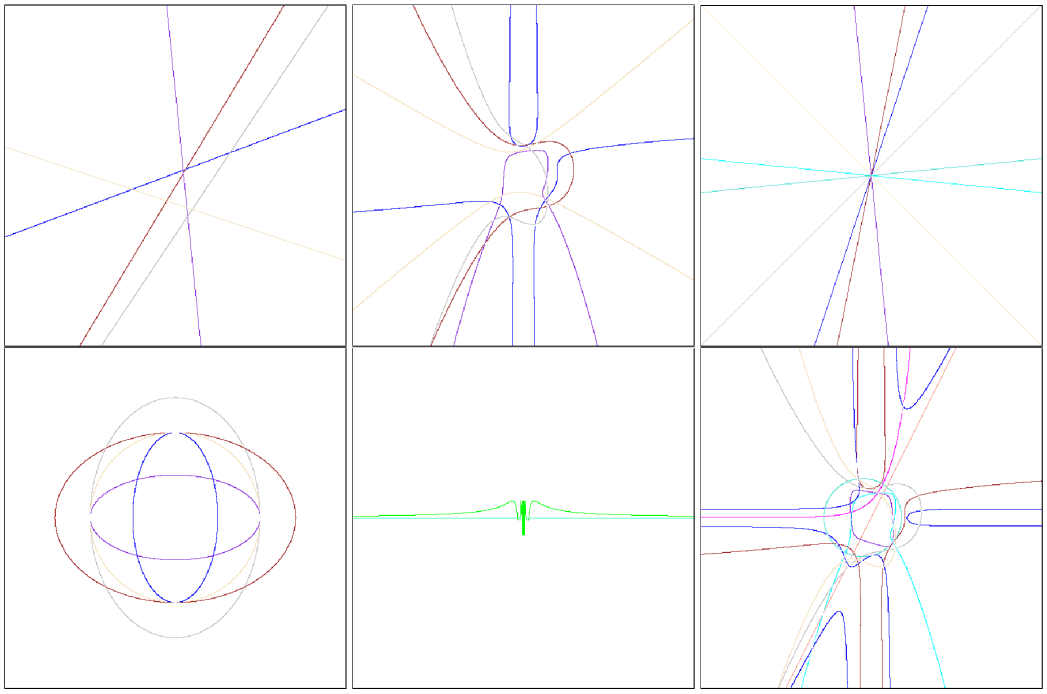


Figure 4.6: *Output of the CAPS algorithm.* From top left to bottom right: Arrangement of lines, algebraic curves, degenerate lines, degenerate polynomials, inverse sine, and arbitrary curves.

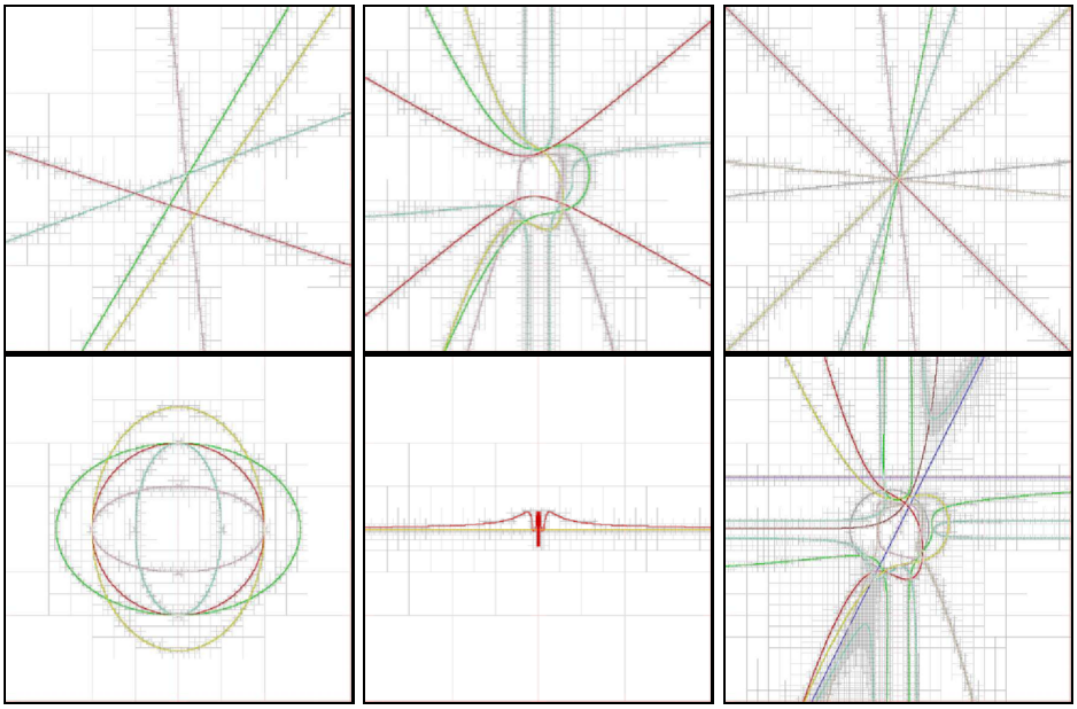


Figure 4.7: *Output of the CAPS algorithm showing the subdivisions* with same curves ordering as in Fig. 4.6.

Chapter 5

Rendering implicit functions: a brief review

In this chapter we review some background notions and present a brief state of the art of methods for rendering implicit functions, especially for ray casting.

5.1 Background

5.1.1 Implicit functions

An *implicit surface S* in $3D$ is defined as the set of solutions of an equation

$$f(x,y,z) = 0 \tag{5.1}$$

where $f : \Omega \subseteq \mathbb{R}^3 \to \mathbb{R}$. For our purposes, assume this function is defined by any analytical expression. In ray casting, we seek the intersection of a ray

$$\vec{P}(t) = \vec{O} + t\vec{D} \tag{5.2}$$

with this surface S. By simple substitution of these position coordinates, we derive a unidimensional expression

$$f_t(t) = f(O_x + tD_x, O_y + tD_y, O_z + tD_z) \tag{5.3}$$

and solve where $f_t(t) = 0$ for the smallest $t > 0$.

In this sense, ray casting is a root-finding problem. For simple implicits such as a plane or sphere, $f_t = 0$ can be solved for t trivially. More complicated expressions, such as non-algebraics and polynomials of degree 5 or higher, cannot be solved analytically. Global

iterative root-finding methods such as *regula falsi* can solve over an interval on which a root is *known* to exist, but fail otherwise. Recursive examination of sign changes, in conjunction with evaluation, work only when a function is monotonic over an interval. Such "point-sampling" methods (e.g. Kalra & Barr [KB89]) succeed when monotonicity assumptions can be made; otherwise they may fail to robustly determine zeros of the implicit, as illustrated in Figure 1.1(a) of Part I. Fortunately, interval arithmetic provides us with a robust mechanism for testing whether or not a zero of a function exists over a sub-domain of the implicit.

5.1.2 Interval arithmetic and ray casting

We shall recall that Moore's fundamental theorem of interval arithmetic [Moo66] states that for any function $f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$ (where Ω is an open subset of $\mathbb{R}^3$) and a domain box $B = X \times Y \times Z \subseteq \Omega$ the corresponding interval extension $F : B \rightarrow F(B)$ is an *inclusion function* of f, in that

$$F(B) \supseteq f(B) = \{f(x,y,z) \mid (x,y,z) \in B\} \tag{5.4}$$

Thus, by using interval arithmetic to evaluate F, we have a very simple and reliable rejection test for the box B not intersecting S,

$$0 \notin F(B) \Rightarrow 0 \notin f(B) \tag{5.5}$$

This property can be used in ray casting for identifying and skipping empty regions of space. Note, however, that although $0 \notin F(B)$ guarantees the absence of a root on an interval B, that the converse does not necessarily hold: one can have $0 \in F(B)$ without B intersecting S. When $F(B)$ loosely bounds the convex hull, as in Figure 1.1(b) of Part I, IA makes for a poor (though still reliable) rejection test. This overestimation problem is a well-known disadvantage, and is fatal to algorithms relying on iterative evaluation of non-diminishing intervals.

Fortunately, overestimation error is proportional to domain interval width; therefore IA guarantees convergence to the correct solution when interval domains diminish. This is the case in ray casting algorithms involving recursive interval bisection [Mit90, CHMS00, KHH*07]. Though the overestimation problem affects the efficiency of these algorithms, recursive IA methods robustly detect the zeros of an implicit, given an adequate termination criterion such as a sufficiently small precision ε over the domain, or tolerance δ over the range.

As explained by Mitchell [Mit91], any function can be expressed as an interval extension by considering its disjoint composition of piecewise-monotonic intervals. This includes non-algebraic piecewise or periodic functions such as modulus, and transcendentals such as exponential, logarithm and trigonometric functions [Duf92]. While rigorous definition of the class of IA-expressible functions falls outside the scope of this Section, intuitively one can derive an IA extension for any computable function. Once defined, IA operators are composable, allowing for trivial representation of arbitrary functions by their component real-operators. Ill-defined operations (e.g. division by zero, in Chapter 6 Section 6.3.4), may require special-case handling, but are typically consistent with existing numerical solutions for real numbers.

5.2 Related work

5.2.1 Proxy geometry methods

Due to the popularity of GPU rasterization, the most common approach to rendering implicits has been extraction of a mesh or proxy geometry. Application of marching cubes [LC87, WMW86] or Bloomenthal polygonization [Blo94] can generate meshes interactively, but will entirely omit features smaller than the static cell width. More sophisticated methods deliver better results, at the cost of interactivity. Paiva et al. [PLLdF06] detail a robust algorithm based on dual marching cubes, using interval arithmetic in conjunction with geometric oracles. Varadhan et al. [VKZM06] employ dual contouring and IA to decompose the implicit into patches, and compute a homeomorphic triangulation for each patch. These methods exploit inclusion arithmetic to generate desirable meshes that preserve topology within geometric constraints. However, they generally compute offline, and do not scale trivially. Moreover, each mesh is a view-independent reconstruction. Schreiner et al. [SSS06] used a moving least-square guidance field to adaptively triangulate implicits. Though they generate nice meshes that preserve topology within geometric constraints, these methods are restricted to continuous or compact manifold implicits, and compute offline in the order of seconds or minutes.

Splatting uses view-dependent point sampling of an implicit reconstruction of point cloud data [RL00]. Dynamic particle sampling methods for implicits have been demonstrated by Witkin & Heckbert [WH94] and extended by Meyer et al. [MGW05]. Slice-based GPU volume rendering, often in conjunction with ray casting, is a practical method of visualizing implicits [HSS*05].

5.2.2 Ray casting implicits

The blobby surfaces of Blinn [Bli82] provided modeling interest in an efficient method of rendering implicits. Hanrahan [Han83] proposed a general but non-robust point-sampling algorithm using Descartes' rule of signs to isolate roots. Van Wijk [vW85] implemented a recursive root bracketing algorithm using Sturm sequences, suitable for differentiable algebraics. Kalra & Barr [KB89] devised a method of rendering a subclass of algebraic surfaces (L-G surfaces) with known Lipschitz bounds. Stolte & Caubet [SC95] applied discrete ray casting to voxelized representations of implicits. Hart [Har96] proposed a robust method for ray tracing algebraics by defining signed distance functions from an arbitrary point to the surface. More recently, Loop & Blinn [LB06] implemented an extremely fast GPU ray caster approximating implicits with piecewise Bernstein polynomials. Romeiro et al. [RVdF06] proposed a hybrid GPU/CPU technique for casting rays through constructive solid geometry (CSG) trees of implicits. De Toledo et al. [dTLP07] demonstrated interactive ray casting of cubics and quartics using standard iterative numerical methods on the GPU.

5.2.3 Ray casting with interval and affine arithmetic

Toth [Tot85] first applied interval arithmetic to ray casting parametric surfaces, in determining an initial convex bound before solving a nonlinear system. Mitchell [Mit90] was the first to employ interval arithmetic for implicit ray casting. He devised a hybrid algorithm that employed bisection to segment the ray into intervals on which the function is monotonic, followed by root refinement via a standard numerical root-finding method. De Cusatis Junior et al. [dCJdFG99] used standard affine arithmetic in conjunction with recursive bisection. Capriani et al. [CHMS00] combined interval bisection with various other iterative schemes, including the Interval Newton method. Sanjuan-Estrada et al. [SECG03] compared performance of two hybrid interval methods with implementations of the Interval Newton and a recursive point-sampling subdivision method in the POV-Ray framework. Florez et al. [FSSV06] proposed a ray tracer that antialiases surfaces by adaptive sampling during interval subdivision. Even when accounting generously for Moore's Law, none of these methods would perform interactively on a modern PC if implemented naïvely. Gamito and Maddock [GM07] proposed reduced affine arithmetic for ray casting specific implicit displacement surfaces formulated with blended noise functions, but their AA implementation fails to preserve inclusion in the general case. Knoll et al. [KHH*07] implemented a generally interactive interval bisection algorithm for arbitrary implicits on the CPU. Performance was achieved though SSE instruction-level optimization and coherent traversal methods; and exploiting the fact that numerically precise roots are not required for visual accuracy. The same authors also developed a new ray casting algorithm optimized for modern graphics hardware using reduced affine

arithmetic yielding to real-time rendering.

5.2.4 Ray coherence

The notion of a group of rays marching in a single direction is simple yet critical to the performance of coherent ray casting systems. Coherent methods have delivered real-time performance for polygonal scenes [WSBW01, RSH05], and SIMD has been used in optimized intersection algorithms for trilinear voxel interpolant surfaces [MFK*04]. The algorithm that will be introduced in the next Chapter was heavily inspired by optimized traversals for coherent SIMD ray casting, particularly the frustum grid traversal proposed by Wald et al. [WIK*06], and the hierarchical extension of that algorithm to large octree volume data by Knoll et al. [KHW07].

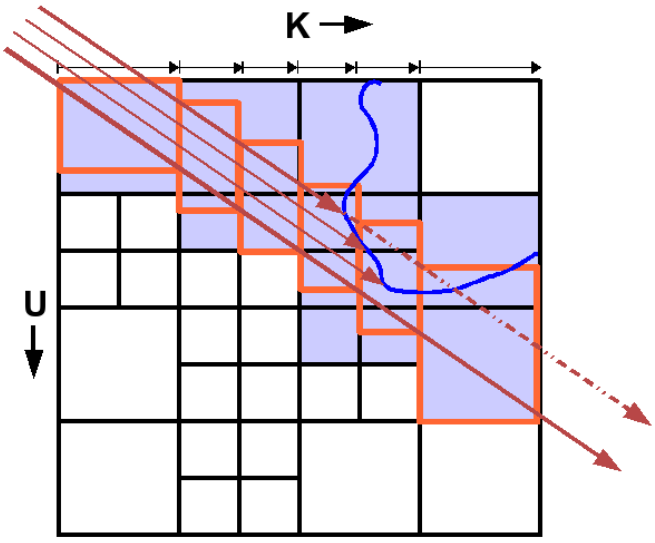


Figure 5.1: *An example of coherent traversal* using an octree as in [KHW07], effectively the hierarchical extension of [WIK*06]. The packet is defined by a bounding frustum; nodes of an acceleration structure are queried when they contain the U (and V in $3D$) extents of that frustum along an interval on K. Marching from one slice to the next simply entails addition. Unlike acceleration structures, however, we do not explicitly store *any* data; we instead evaluate the IA expression of the implicit function.

5.2.5 Coherent ray casting

The principal idea of coherent ray casting is to perform traversal and intersection on groups, or *packets*, of rays. In this way, the costs associated with ray casting are amor-

tized over that group. Aggressive coherent methods often compute traversal steps over a bounding frustum of the packet as opposed to individual rays themselves, e.g. [WIK*06, RSH05]. More conservative methods (e.g. [WSBW01]) exploit coherence on a smaller scale, specifically when encouraged by hardware. SIMD instruction sets such as SSE perform four floating point operations in parallel, encouraging operations on packets of four rays. While potential gains are more modest, rays with divergent behaviors may still benefit from instruction-level parallelism. The concept of coherent traversal is illustrated on Figure 5.1.

Coherent ray casting performs best when rays in a packet behave similarly. Ideally, neighboring rays march in lockstep, requiring the fewest total traversal steps to examine a region of space. In the Wald et al. [WIK*06] coherent grid traversal (CGT) algorithm, coherent traversal of rectilinear space is accomplished by choosing a major march axis K corresponding to the dominant ray direction, and examining slices of the other dimensions along fixed K intervals. A hierarchical octree extension of CGT was proposed by Knoll et al. [KHW07], and is the major algorithmic inspiration for this work.

5.3 Ray casting arbitrary implicits

In ray casting, all geometric primitives are at some level defined implicitly, and the problem is essentially one of solving for roots. Simple implicits such as a plane or a sphere have closed-form solutions that can be solved trivially. General implicits without a closed-form solution require iterative numerical methods. However, easy methods such as Newton-Raphson, and even "globally-convergent" methods such as *regula falsi*, only work on ray intervals where f is monotonic. As shown in Fig. 1.1 of Part I, "point sampling" using the rule of signs (e.g. [Han83]) fails as a robust rejection test on non-monotonic intervals. While many methods exist for isolating monotonic regions or approximating the solution, inclusion methods using interval or affine arithmetic are among the most robust and general. Historically, they have also been among the slowest, due to inefficient implementation and impractical numerical assumptions.

As already mentioned the inclusion property extends to multivariate implicits as well, making it suitable for a spatial rejection test in ray casting. Moreover, by substituting the inclusion extension of the ray equation (Equation 5.2) into the implicit extension $F(x,y,z)$, we have a univariate extension $F_t(X,Y,Z)$. To check whether any given ray interval $\bar{t} = [\underline{t},\bar{t}]$ *possibly* contains our surface, we simply check if $0 \in F_t(\bar{t})$. As a result, once the inclusion library is implemented, any function composed of its operators can be rendered robustly. To select domain intervals on which to evaluate the extension, one has a wide choice of interval numerical methods [Mit90, CHMS00, SECG03]. Empirical

results [KHH*07, GM07, dCJdFG99] suggest that simple bisection works best, particularly at coarser precision ε. In practice, evaluating a gradient extension is expensive, and higher-order convergent methods resort to bisection on non-monotonic regions. Moreover, high numerical precision is seldom required for accurate visualization [KHH*07].

Chapter 6

Interactive ray casting of arbitrary implicits on the CPU

In this chapter we present one of our contributions to ray casting arbitrary implicit functions, that is a practical and efficient algorithm for interactively ray casting arbitrary implicit surfaces. This work has been done in collaboration with Aaron Knoll (SCI Institute, University of Utah). We use interval arithmetic (IA) both for robust root computation and guaranteed detection of topological features. In conjunction with ray casting, this allows for rendering literally any programmable implicit function simply from its definition. Our method requires neither special hardware, nor pre-processing or storage of any data structure. Efficiency is achieved through SIMD optimization of both the interval arithmetic computation and coherent ray traversal algorithm, delivering interactive results even for complex implicit functions. To our knowledge, this is the first time interactivity has been achieved for ray casting arbitrary implicits.

6.1 Introduction

In graphics, geometry is most often modeled explicitly as a piecewise-linear mesh. An alternative is a higher-order analytical representation in implicit or parametric form. This option presents advantages, such as compact storage and view-independent local smoothness. While implicits have not experienced as widespread adoption as parametric surfaces in $3D$ modeling, they are common in other fields, such as mathematics, physics and biology. Moreover, they serve as geometric primitives for iso-surface visualization of point sets and volume data.

To render implicits in $3D$, one is principally given a choice of extracting and rasterizing a

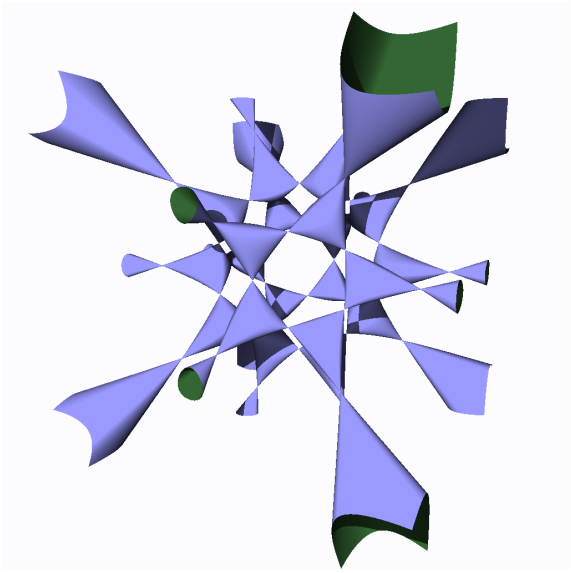


Figure 6.1: *The Barth-sextic Implicit* rendered roughly interactively at 9.0 fps (6.1 fps with shadows) with a 512^2 frame buffer on an Intel Core Duo 2.16 GHz, purely on the CPU.

mesh, or ray casting the surface directly via root-solving. Mesh extraction methods that adaptively reconstruct geometric or topological features exist; however they remain limited in the features they can reproduce, and are not sufficiently fast for dynamic extraction alongside real-time rasterization. While ray casting low-order implicits is often trivial, arbitrary implicits pose a difficult problem. In the past two decades, several techniques have been developed to ray trace general implicits robustly. Overall, these methods either are slow, restrict the class of functions they handle, or resort to piecewise approximations. Methods involving interval arithmetic (IA) are the most general in that they can accommodate any programmable function. As implemented, however, they are among the least efficient.

Recently, coherent traversal techniques, SIMD vector instructions and multi-core CPUs have enabled interactive ray casting. Applications have largely sought to compete with rasterization in rendering explicit geometries – principally offering scalability to large data, and more powerful, flexible and intuitive shading and lighting models. As geometries that *cannot* be trivially rasterized, arbitrary implicits make a particularly intriguing application for ray casting. Coherent ray casting has not been applied to this problem before, and conventional ray casting methods are slow largely due to the high computational cost of interval evaluation. By optimizing interval arithmetic with SSE, and pairing this with a fast coherent traversal algorithm, we find that interactive performance is possible on common laptop hardware, with a system that accurately visualizes any implicit surface composable by interval algebra.

The contribution of our work is the combination of a SIMD interval arithmetic library with a novel coherent ray casting algorithm for implicits that performs coherent spatial bisection without the need for an explicit acceleration structure. We require no special hardware, other than SIMD vector instructions prevalent on all modern CPUs. To render, we require only the implicit function itself, a desired graphing domain, and an appropriate precision criterion or tolerance. We demonstrate our method on various implicits, including difficult cases for extraction-based methods, such as functions with singularities and time-variant $4D$ hyper-surfaces.

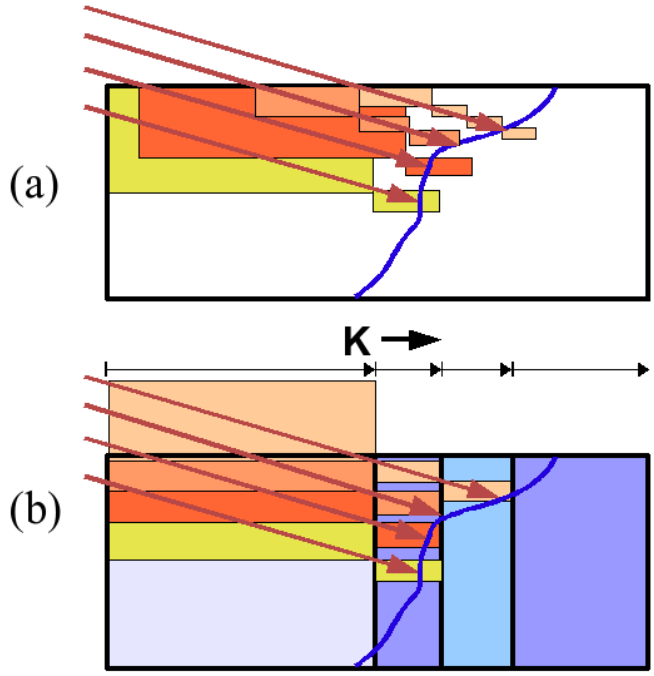


Figure 6.2: *Interval bisection methods.* The conventional method (a) recursively bisects each ray along its parameter t until a surface is located to the satisfaction of a termination criterion. Our $K-$*marching* technique (b) marches rays along a common axis in lockstep. Evaluating along $3D$ interval boxes B requires slightly less computation per iteration than evaluating the projected function $f_t(t)$. More importantly, traversing along a common spatial axis induces more coherent behavior between rays in a packet.

6.2 Coherent ray casting of implicits with IA

Our algorithm simplifies the interval bisection method first proposed by Mitchell [Mit90], and employs a variant of coherent octree traversal [KHW07] as opposed to direct bisection of t intervals along the ray. Together, these decisions allow us to perform bisection in

a non-recursive manner, evaluate intervals quickly using SIMD vector instructions, and avoid unnecessary per-step interval multiplication. The simplicity and efficiency of this algorithm allow it to interactively visualize most implicit functions.

The conventional Mitchell algorithm [Mit90] employs interval bisection to reject empty (rootless) intervals. For each nonempty interval, it then computes the *gradient interval*, and determines whether $0 \notin F_t'(T)$, i.e. if the function is monotonic over an interval T. When this occurrs, Mitchell resorts to a robust numerical "refinement" method, such as non-IA bisection or regula falsi. Interval Newton methods (e.g. [CHMS00, SECG03]) also compute $F_t'(T)$ per-iteration. Gradient interval computation proves expensive. Although previous works suggest these techniques offer improved convergence and efficiency compared to pure bisection, that supposition has been weakly scrutinized. In the context of coherent traversal, we find that interval bisection yields unequivocally *better* performance, and achieves equivalent visual results efficiently at coarser sampling rates.

To leverage SIMD vector operations, we perform interval bisection on four rays at a time. Rather than bisecting t along the ray direction as in Figure 6.2(a), we bisect space along a major directional axis K, similar to the coherent octree volume traversal proposed in [KHW07], and illustrated in Figure 6.2(b). Particularly when the space between rays exceeds the domain sampling width ε, this ensures more regular sampling of the function across neighboring rays, and preserves the spatial lockstep of coherent traversal (see Section 6.4.5).

The process of evaluating intervals is then simple. Given an interval box $B = X \times Y \times Z$, our function f and its corresponding IA evaluation F, we evaluate whether $0 \in F(B)$ for any ray in the packet. If so, we bisect that interval along the major march axis, or register a hit if a maximum depth threshold is reached. Rather than evaluating the IA extension of the implicit $F_t(T)$ projected along the ray, as preferred by previous works, our K-bisection method evaluates the $3D$ implicit $F(X,Y,Z)$ directly. This is convenient as both the IA extension and evaluation functions are natively given as $f(x,y,z)$ expressions. Moreover, our traversal algorithm computes domain intervals B incrementally, requiring only three SSE additions per iteration. Conversely, evaluating $F_t(T)$ requires IA evaluation of Equation 5.3: three IA multiplications and IA additions, or six SSE multiply, min, max and add operations in total.

6.3 Implementation

As the contribution of this work is largely algorithmic in nature, we include pseudocode for critical components of our implementation. We abbreviate the 4-vector packet floating point datatype as "simd". Implicits then call the associated SSE and IA library function calls, for example "mul4" for a SIMD multiplication, and "mul_i4" for a SIMD interval

multiplication.

Our application takes as inputs a domain $\Omega \subseteq \mathbb{R}^3$, and an implicit function expression. For simplicity, we chose to hard-code most functions as IA expressions; however the function can also be received from the user as a string and then parsed and compiled into IA code in a dynamic library on-the-fly.

Algorithm 1 SIMD Interval Arithmetic

```
struct interval4 {
  simd lo, hi;
};
interval4 add_i4(interval4 a, interval4 b) {
  return interval4( add4(a.lo, b.lo), add4(a.hi, b.hi) );
}
interval4 mul_i4(interval4 a, interval4 b) {
  simd lolo = mul4(a.lo, b.lo);
  simd lohi = mul4(a.lo, b.hi);
  simd hilo = mul4(a.hi, b.lo);
  simd hihi = mul4(a.hi, b.hi);
  return interval4( min4(lolo, min4(lohi, min4(hilo, hihi))),
                    max4(lolo, max4(lohi, max4(hilo, hihi))) );
}
interval4 abs_i4(interval4 a) {
  return interval4( max4(a.lo, max4(-a.hi), 0)),
                    max4(-a.lo, a.hi) );
}
interval4 sqr_i4(interval4 a) {
  interval4 aa = abs_i4(a);
  return interval4( mul4(a.lo, a.lo), mul4(a.hi, a.hi) );
}
interval4 circle(interval4 x, interval4 y, interval4 z,
                                                 float radius)
{
  return sub_i4(add_i4(sqr_i4(x), add_i4(sqr_i4(y), sqr_i4(z))),
                radius*radius);
}
```

6.3.1 SSE interval arithmetic

The foundation of our implicit ray casting system is our own SSE IA library, which allows us to quickly evaluate intervals in SIMD. Implementation is straightforward; interval multiplication is particularly efficient as SSE itself is relatively fast for both multiplication and minimum/maximum operation. The only non-trivial operators are periodic functions such as modulus and sine; and division which requires special-case handling during traversal (see Section 6.3.4). Examples of SSE IA pseudocode are given in Algorithm 1. We deliberately ignore IA rounding rules for numerical conditioning. For our visualization application, IEEE float rounding errors are insignificant compared to the termination tolerance of our bisection algorithm. One could likely devise numerically ill-conditioned functions that would require IA rounding, but for our purposes it is not a major issue.

6.3.2 Ray packet structure

We chose conservative 2x2 packets for our implementation. Above all, we wish to evaluate baseline performance with SIMD ray casting using 4-wide SSE vectors; thus behavior of our system should be consistent on wider SIMD hardware, such as a GPU or FGPA. Though larger packets coupled with multi-level algorithms could be significantly faster (e.g. [RSH05]), 2x2 packet traversal is better-suited for general-purpose ray casting, and easily allows our implicits to be integrated into a ray tracer as geometric intersection primitives. The actual packet architecture should generalize to any coherent ray tracer; our packet implementation consists of origin and direction stored for each X, Y, Z axis in SSE packed floats. Packets also store the ray hit parameters t, and a mask indicating which rays have hit.

Algorithm 2 Ray Packet Structure

```
struct RayPacket {
  simd org[3];
  simd dir[3];
  simd inv_dir[3];
  simd t_hit;
  simd p_hit[3];
  simd normal[3];
  simd hitmask;
};
```

6.3.3 Traversal

Once the user has supplied a function, a domain box $\Omega \subseteq \mathbb{R}^3$, and a maximum depth d_{stop}, we are ready to perform traversal. As in coherent grid traversal [WIK*06], we first find K, the dominant axis of the first ray in the packet, and denote the remaining two axes U and V. We then perform a standard ray bounding-box test on our domain. We store the actual t_{enter} and t_{exit} parameters as well as the intersections with the K entry and exit planes, t_{Kenter} and t_{Kexit}. Now, we consider the total increment along K, $t_{Kexit} - t_{Kenter}$, and compute the total U and V increments over the entire domain. As our implementation is iterative, not recursive, we store an array containing a traversal "stack" for each depth $\{0..d_{stop} - 1\}$, containing the t, K, U and V increments bisected at each level.

The algorithm then simply marches from one K slice to the next, incrementing the t, K, U and V positions once per step and keeping track of current and next values, orthogonally for each ray using SSE. It constructs intervals from the K, U and V current and next values. This enables us to iteratively increment domain intervals simply with three SSE additions, as opposed to three SIMD IA multiplications and additions using the conventional t-marching method. Branching is only used to omit intervals when $t < t_{enter}$, and exit when all rays hit successfully or have $t > t_{exit}$. We store and check a flag for each depth, which indicates when both sides of a K-subtree have been traversed. When this happens, we decrement the depth, and exit traversal when $depth = -1$.

At each march iteration, we evaluate the IA function expression on this domain interval $B = X \times Y \times Z$. If $0 \in F(X,Y,Z)$, we "recurse" by incrementing d and using the bisected increments one level deeper. We register a hit on the surface when $d == d_{stop} - 1$ (or another hit criterion is met, such as $||F(B)|| < \delta$, as in Section 6.3.5). Finally, we mask rays that successfully hit or terminate traversal when all rays hit. Traversal is illustrated in Figure 6.2(b), and pseudocode is given in Appendix A.

6.3.4 Division

IA division requires a slight modification to the above algorithm. In theory, IA division by intervals containing zero is ill-defined, similar to division of real numbers by zero. Fortunately, we can easily detect and handle these cases. For two intervals A and B, when $0 \in B$, we define $A/B = [-\infty, \infty]$. When rays traverse these intervals, they will *always* find a surface within and recurse to maximum depth. Thus, without modification to the traversal, asymptotes will be rendered. To avoid rendering asymptotes, we simply neglect to register a hit when $F_{hi} - F_{lo} = \infty$. This principle is illustrated in Figure 6.3. With division correctly handled, our traverser will work for literally any function composed of IA operators.

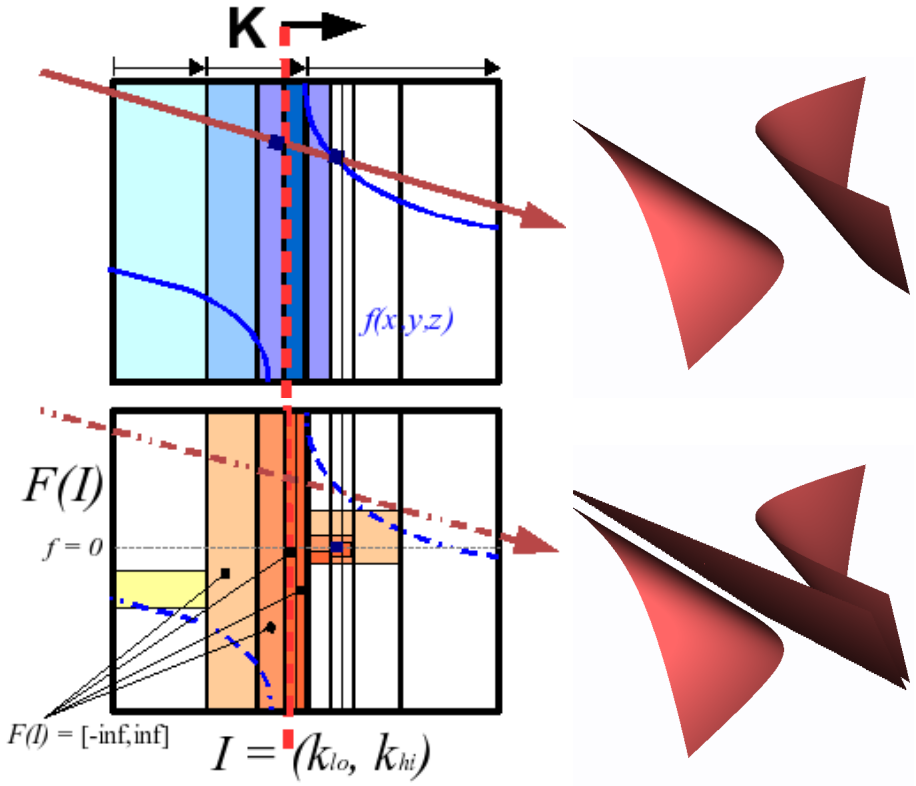


Figure 6.3: *Handling Division.* For functions with division, and intervals containing zero near an asymptote, our IA implementation returns "infinite" $F(I)$ intervals (bottom left). As a result, these regions are always subdivided until termination (top left). Fortunately, we may detect this infinite case within the traverser before registering a hit, and thus choose whether or not to visualize asymptotes.

6.3.5 Precision criterion

In our implementation, d_{stop} determines the default precision for rendering the implicit. Roughly, this corresponds to a domain precision of $2^{-d_{stop}}$, though indeed this varies by ray. However, for a more view-independent domain-space metric, the user may optionally specify an ε, such that $||B||_2 < \varepsilon$ serves as hit criterion, where B is an interval box $X \times Y \times Z$. In this case, the stopping depth is determined adaptively per-packet as

$$d_{stop} = log_2(\Delta_{packet}/\varepsilon) \tag{6.1}$$

where for world-space ray entry and exits $\vec{P}_r$ with the domain box Ω, and their corresponding K-coordinates K_r,

$$\Delta_{packet} = max_{r \in packet} \frac{(||\vec{P}_r^{exit} - \vec{P}_r^{enter}||_2)^2}{|K_r^{exit} - K_r^{enter}|} \tag{6.2}$$

Alternately, the user may specify a range tolerance δ, in which case our algorithm registers a hit when $||F(B)|| < \delta$. Empirically, the performance differences between these metrics proved minor, and at low precision the d_{stop} method yields more continuous results for neighboring rays. Thus, we use d_{stop} as the default metric for evaluating performance at varying sampling quality.

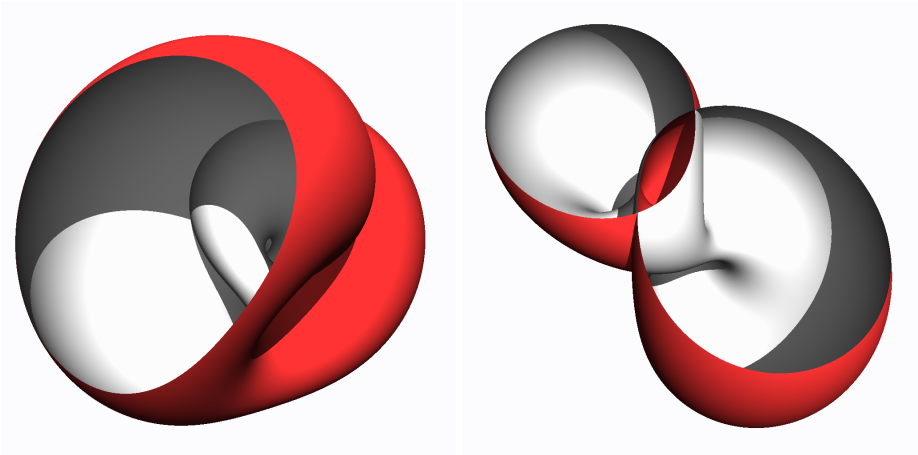


Figure 6.4: *Dynamic shadows* aid greatly in visualizing the Klein Bottle. Images rendered at 4.0 fps and 2.9 fps, respectively at $d_{stop} = 12$.

6.3.6 Shadows

In ray casting, hard shadows are fairly trivial, requiring a shadow ray cast for every primary camera ray that hits a surface. This typically entails a 20% to 50% decrease in frame rate, depending on the coherent behavior of shadow rays. Fortunately, useful shadow rays require less accuracy than primary rays; it frequently suffices to cast shadows to a coarser termination depth, such as $d_{stop} - 2$, while employing a higher depth for primary rays. As shadows are primarily useful as depth cues, this is generally acceptable. The performance penalty is reduced, and loss of shadow detail is seldom perceptible (Figures 6.1 and 6.4).

6.3.7 Gradient computation

For Lambertian shading, we require the surface normal at the ray hit position, given by the $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$ partial derivatives at that point. While analytical gradients can be manually defined, they are not strictly necessary. If the user fails to define partials, we employ central differences by evaluating our function (using SSE, not SSE IA evaluation) six times to create a central differences stencil. The results look excellent in most cases,

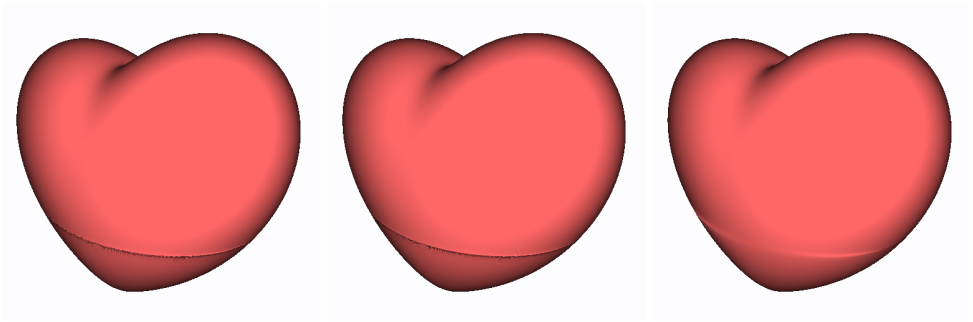


Figure 6.5: *Gradient normal computation,* on the Heart function $f(x,y,z) = (2x^2 + y^2 + z^2 - 1)^3 - (.1x^2 + y^2)z^3$. Left: using analytical partial derivatives as gradient, we see shading artifacts where the gradient magnitude approaches zero. Center: with a central differences stencil of width $\Delta_S = 0.001$, the results are visually indistinguishable. Right: smoother normals with $\Delta_S = 0.01$. All images render at 6.7 fps.

and have no appreciable impact on performance. We allow the user to specify stencil width; this is frequently beneficial for surface regions with near-zero gradient magnitude (Figure 6.5).

6.4 Results

6.4.1 General performance

All benchmarks were performed on an Intel Core Duo 2.16 GHz laptop with a 512^2 frame buffer. Figures B.1 and B.2 in Appendix B shows various implicit surfaces with their associated equations and performance. Our system achieves well over 20 frames per second for simple objects such as the torus, sphere and conic sections. For more complex objects, performance can fall below interactive speeds on our hardware, but generally exploration at 1-5 fps is possible even for the worst cases. Complicated implicits such as the Barth-sextic exhibit similar performance. Most importantly, we are not restricted to any particular class of surfaces. Non-differentiable, non-continuous, non-manifold, self-intersecting and linked implicits are all robustly rendered.

6.4.2 Precision and quality

We use a common bisection depth d_{stop} for benchmarking, which corresponds to a domain precision of $2^{-d_{stop}}$ along the K axis of a given packet. The minimum depth required

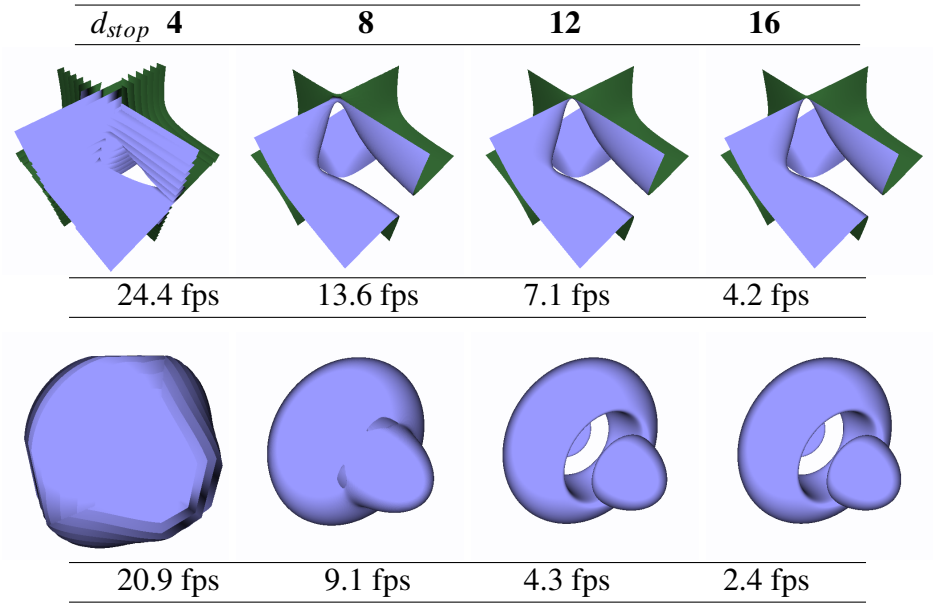


Figure 6.6: *Quality at various d_{stop} bisection depths.* Performance is inversely proportional to depth. Top: the 1st-order Lagrangian trilinear interpolant patch, a cubic implicit, yields tight intervals and converges quickly to the correct contour. Bottom: the Mitchell function causes relatively high IA bound overestimation, and requires greater depth for correct visualization. Even here, a coarse precision criterion $\varepsilon < 10^{-3}$ is sufficient to capture the correct topology.

for accurate visualization depends largely on the bound overestimation of the composed IA rules for that function (Figure 6.6). As seen in Figures B.1 and B.2 of Appendix B, $d_{stop} = 10$ is in practice a good balance of performance and feature reproduction for the vast majority of functions we test. This finding is surprising: a domain precision of $\varepsilon = 2^{-10} \approx 10^{-3}$ suffices to accurately visualize most implicits.

6.4.3 Feature reproduction

The tear drop implicit (Figure 6.7) demonstrates how our algorithm can reproduce fine details that extraction-based approaches often omit. View-independent mesh extraction methods, e.g. [PLLdF06], frequently fail to capture such regions of a surface, leading to misclassification of details such as asymptotes, singularities or infinitely thin connected surfaces. However, when thin regions or singularities lie between two rays and the interval bounds are sufficiently small, both discrete rays will (correctly) miss the surface, even though that surface would be encountered by an interval beam. To accurately reproduce such sub-pixel features would be expensive, requiring both supersampling and beam trac-

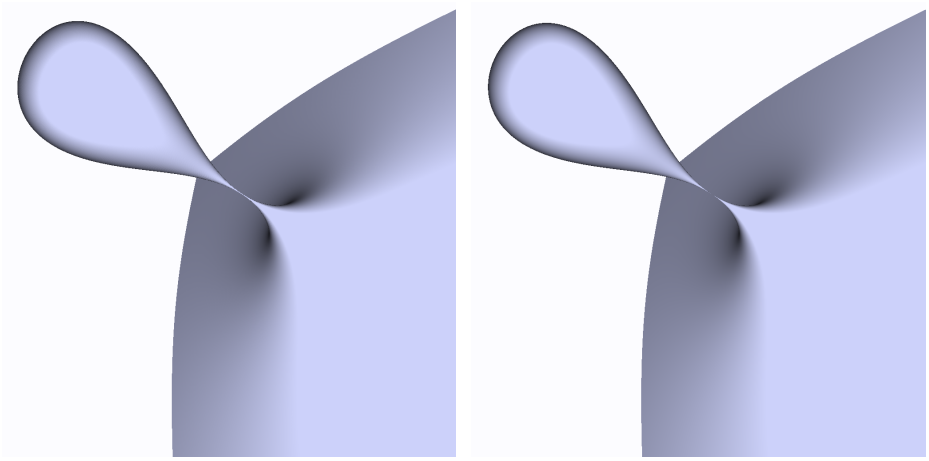


Figure 6.7: *Reproduction of fine features.* Though robust for each individual ray, ray casting (as opposed to beam tracing) may fail to capture infinitely thin features. Coarser-contour visualization at lower precision actually aids in understanding these functions. Left: $d_{stop} = 10$ at 11 fps. Right: $d_{stop} = 14$ at 6.5 fps.

ing of ray intervals, as detailed by Gavriliu [Gav05]. Rendering at lower precision can actually aid in visualizing these features, as the IA inclusion property guarantees that our rendered surface will always form a convex contour of the actual zero-set (Figure 6.6). In this way, the user can iteratively modify d_{stop} until the true surface topology is understood.

6.4.4 Dynamic scenes

Because we neither precompute an explicit representation of the object, nor a physical acceleration structure in memory, we have great flexibility in rendering dynamically changing N-dimensional implicits. For example, we can render $4D$ implicits as $3D$ over time, using a $f(x,y,z,w)$ expression. An example of a two-sheeted hyperboloid morphing into a torus is shown in Figure 6.8. Though dynamic implicits would be difficult to achieve with mesh extraction techniques, they are trivial in our ray casting system.

6.4.5 Algorithm performance analysis

Perhaps our most striking finding is that practical IA-based implicit rendering is not inherently slow, even though previous techniques yielded generally poor performance. Implementations such as Mitchell [Mit90] and Capriani et al. [CHMS00] sought to render implicits at up to machine precision (up to $\varepsilon = 10^{-7}$) with superlinearly convergent numerical methods. Despite its slower theoretical convergence, we find that pure interval

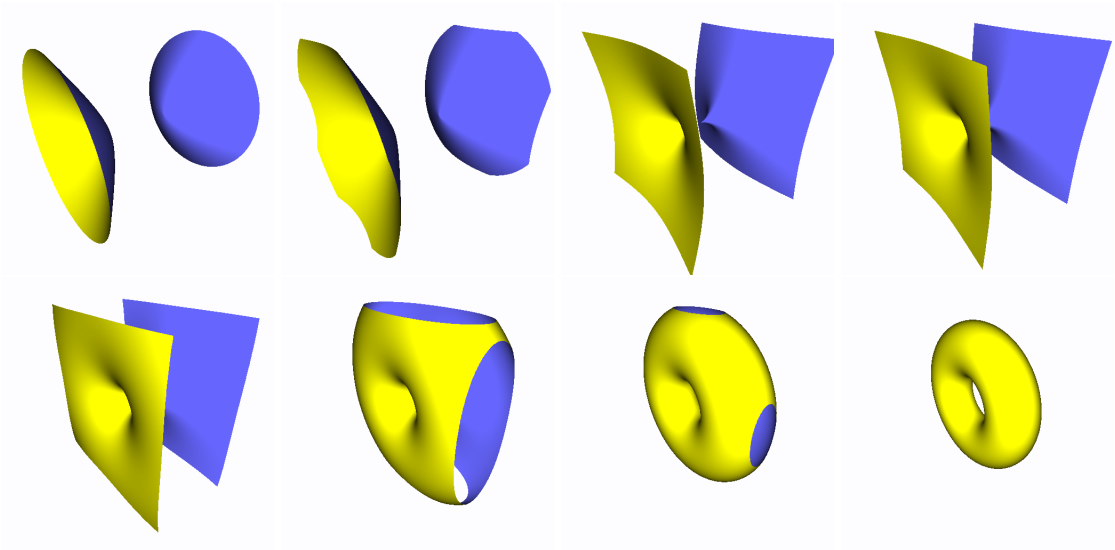


Figure 6.8: *Animated 4D implicits*. As our algorithm does not compute or store any acceleration structure, we can make arbitrary changes to the implicit function on the fly. In this example, we interactively morph a hyperboloid into a torus at 9-20 fps.

bisection is more efficient than these methods, particularly at lower precision which is more than adequate for correct visualization (see Section 6.4.2). To verify this, we implement an SSE variation of the Mitchell [Mit90] algorithm, which performs interval bisection until *all* rays in the packet have $0 \notin F'(B)$, followed by non-interval bisection for root refinement. Implemented in SSE, this method proves far slower than pure bisection, even with small ε. In addition, we compare our K-marching algorithm with a standard t-bisection. For large, practical ε, standard t-marching only performs $5\% - 20\%$ slower, depending on scene and computational demand of implicit evaluation. However, at smaller ε, where the actual domain intervals of neighboring rays diverge spatially (Figure 6.2(a)), coherence suffers and K-marching is significantly more efficient, potentially by an order of magnitude. These findings are summarized in Table 6.1, and overall encourage implementation of our K-marching method.

6.4.6 Comparison to existing techniques

It is difficult to assess the performance of comparable works in implicit IA ray casting. Fortunately, many papers evaluate performance with a sphere. [dCJdFG99] reported around 1.3 fps at 64x64 on a Pentium 166. Accounting generously for Moore's Law (doubling performance every 18 months), we still achieve between two and three orders of magnitude better performance. Similarly, the hybrid and Interval Newton methods benchmarked in [SECG03] perform at two to three orders of magnitude slower than our

Algorithm	K-bisect		t-bisect		Mitchell	
Domain ε	1e-3	2e-7	1e-3	2e-7	1e-3	2e-7
FUNCTION	FPS					
trilerp	10.6	2.8	9.9	0.31	1.20	0.75
mitchell	5.9	1.3	5.7	1.0	0.61	0.24

Table 6.1: *Algorithm performance comparison* between our K-bisection method, an SSE 2x2 packet implementation of the Mitchell [Mit90] algorithm, and a pure t-marching interval bisection. For the K-bisection method, these ε correspond to $d_{stop} = 10$ and $d_{stop} = 22$. Refer to Figure 6.6 for images of the trilinear interpolant (trilerp) and Mitchell functions.

method. Florez et al. [FSSV06] rendered a sphere in 40 seconds at 300x300 resolution on a P4 2.4 GHz, albeit with adaptive antialiasing; again our method delivers over two orders of magnitude better frame rate (see Appendix Figures B.1 and B.2).

6.5 Conclusion

We have detailed a coherent ray casting technique for rendering arbitrary implicit functions. By combining a coherent traversal algorithm with an efficient SSE interval arithmetic library, we are able to visualize implicits robustly, accurately, and interactively at rates over two orders of magnitude faster than previous implementations.

Possibilities for extending our system abound. Performance could be further improved by using larger packets and multilevel coherent ray casting techniques. Adaptive methods (e.g. [FSSV06]) might be desirable for better image quality at lower cost, particularly in conjunction with beam tracing (e.g. [Gav05]), which could robustly antialias thin features and singularities. Performance with computationally difficult implicits, and particularly those with high bound overestimation, would improve with a higher-order inclusion rule set such as affine arithmetic [dCJdFG99] or midpoint-Taylor arithmetic [Gav05]. Though it would entail some sacrifice in generality and portability, a similar interval bisection algorithm would be simple to implement, and likely fast, as a fragment program on the GPU.

While powerful, our method has some limitations. It is not an interval beam tracer; aliasing may occur when rendering functions with sub-pixel features at small tolerance. Though interactive for most implicits we tested, it is still computationally demanding and may not be as fast as special-case intersections, particularly for lower order implicits. More generally, implicits have not experienced widespread adoption in graphics compared to explicit modeling methods for smooth surfaces such as subdivision surfaces, though this has perhaps been partly due to their difficult rendering.

An immediate application for this work is a general-purpose 3*D* graphing application, for use in conjunction with a mathematical software package. CPU ray casting is particularly attractive for this task as it requires no specialized graphics hardware. Ultimately, the ability to efficiently render general implicits could have interesting implications in graphics. Point-set rendering methods such as MPU [OBA*03] relying on rational implicits could easily be ray-traced using this technique. Procedural noise implicits could be employed for surfaces, as in [GM07]. In visualization, isosurfaces of higher-order finite elements [NK06] could be more efficiently rendered. Also of interest would be using a similar IA technique to ray-trace arbitrary parametric surfaces, as suggested by Mitchell [Mit91].

Chapter 7

Real-time ray casting of arbitrary implicits on the GPU

Existing methods for rendering arbitrary implicit functions are limited, either in performance, correctness or flexibility. Ray casting methods in conjunction with an inclusion algebra such as interval arithmetic (IA) or affine arithmetic (AA) have historically proven robust and flexible, but slow. In this chapter, we present a new stackless ray traversal algorithm optimized for modern graphics hardware, and a correct inclusion-preserving reduced affine arithmetic (RAA) suitable for fragment shader languages. Shader metaprogramming allows for immediate and automatic generation of functions and their interval or affine extensions, enhancing user interaction. Ray casting lends itself to multi-bounce effects, such as shadows and depth peeling, which are useful modalities for visualizing complicated implicit functions. With this system, we are able to render even complex implicits correctly, in real-time at high resolution. This work has been done in collaboration with Aaron Knoll (SCI Institute, University of Utah).

7.1 Introduction

In computer graphics, geometry is most frequently rendered as a piecewise-linear mesh, which is both intuitive to model with and trivial to rasterize on z-buffer graphics hardware. As visual complexity and realism increase, it becomes difficult to model fine-level geometry directly. Ultimately, procedural geometry is a necessary supplement for multi-level anisotropic representations of created content. While the most common applications of procedural geometry have entailed smooth refinement of a coarse base, as in free-form subdivision surfaces, one can equally employ procedures to augment complexity and add features. Implicits such as procedural noise have been employed to great effect as $3D$

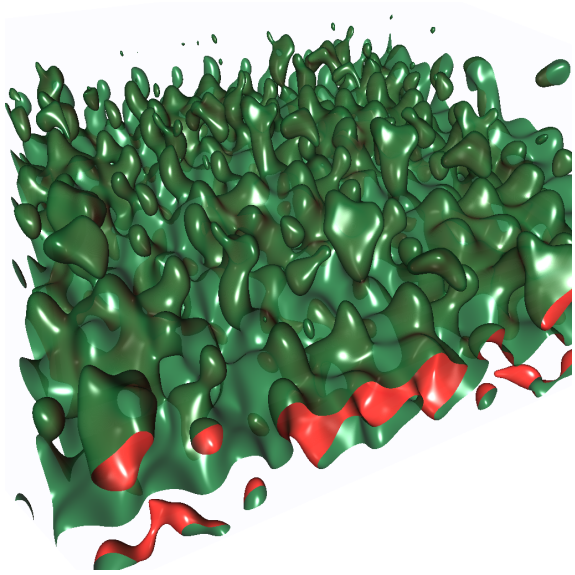


Figure 7.1: *An animated sinusoid-kernel surface.* Ray-traced directly on fragment units, no new geometry is introduced into the rasterization pipeline. IA/AA methods ensure robust rendering of any inclusion-computable implicit.

textures, but not as hyper-texture surfaces for displaced geometry. Largely, this has been due to the slow performance in rendering general implicit surfaces; and perhaps partly as a result, poor understanding of their behavior in modeling and animating physical phenomena.

In scientific simulation, visualization and data analysis, implicits play a central role, albeit often behind the curtain. Reconstructions of point and volume data invariably take implicit form, regardless of the smoothing or interpolation metric. For improved filtering and better reconstruction of data, there is also a need for flexibility in rendering any creatable implicit. In computation and visualization as well as graphics, the limitations of existing rendering techniques have adversely impacted understanding and adoption of arbitrary higher-order implicit forms. In mathematics, visualizing implicits is a goal in and of itself, particularly in the fields of topology, differential geometry and abstract algebra.

To render implicits, one is principally given two choices: sampling the implicit and extracting proxy geometry such as a mesh, volume or point cloud; or ray casting the implicit directly. Though the former methods are often preferred due to the speed of rasterizing proxy geometries, extraction methods yield isotropic geometry and often scale poorly. Though computationally expensive, ray casting methods parallelize efficiently and trivially. Modern graphics hardware offers enormous parallel computational power, at the cost of poor efficiency under algorithms with branching and irregular memory access. GPU-based ray casting [PBMH02] is increasingly common, but often algorithmically in-

efficient.

Ray casting methods for implicits have historically sacrificed either speed, correctness or flexibility. Piecewise algebraic implicits have been rendered in real-time on the GPU using Bezier decompositions [LB06], but approximating methods do not render arbitrary functions directly, nor always robustly. Inclusion methods, such as interval arithmetic (IA) or affine arithmetic (AA), are considered the most general and robust, but traditionally the slowest. Recently, arbitrary implicits were rendered interactively on the CPU by optimizing IA ray casting with SIMD vector instructions, and by making practical assumptions about the numerical precision needed for correct visualization [KHH*07]. Though that system is over two orders of magnitude faster than its predecessors, it is still only roughly interactive on current CPU hardware. A GPU implementation is desirable for its superior computational throughput, and use in conjunction with the conventional rasterization pipeline.

Our major contributions are a new iterative spatial traversal algorithm for implicit intersection; and an efficient implementation of a correct reduced affine arithmetic (RAA) suitable for shader languages. Together, these allow real-time rendering of complex implicit functions. Shader meta-programming allows users to design implicits and procedural hyper-textures flexibly, with immediate results and full support for dynamic $4D$ surfaces. The ray casting algorithm enables multi-bounce effects to be computed interactively without image-space approximations, enabling effects such as translucent depth peeling and shadows which further assist visualization.

7.2 Ray casting implicits with IA and AA on the GPU

In many ways, modern shader languages such as Cg or GLSL allow for a far more graceful implementation than the optimized SSE C++ counterpart on the CPU. Thanks to this language flexibility, it is possible to design a full ray tracer within a fragment program. On-the-fly shader compilation, in conjunction with metaprogramming, can easily and dynamically generate IA/AA extension routines from an input expression.

Nonetheless, implementing a robust interval-bisection ray tracer on the GPU poses challenges. Principally, the CPU algorithm relies on an efficient iterative algorithm for bisection: employing a read/write array for the recursion stack. Storing such an array per-fragment occupies numerous infrequently-used registers, which slows processing on the GPU. Similar problems have clearly hampered performance of hierarchical acceleration structure traversal for mesh ray tracing [PGSS07]. Our most significant contribution is a traversal algorithm that overcomes this problem. By employing simple floating-point modulus arithmetic in conjunction with a DDA-like incremental algorithm operating on specially constructed intervals, we are able to perform traversal without any stack.

Though this algorithm would be prohibitively expensive on a CPU, it is well-suited for the GPU architecture due to efficient division operations.

In implementing affine arithmetic to mitigate IA bound overestimation, it was immediately clear that a full array-based implementation of conventional AA would be impractical on the GPU. Though efficient, the reduced affine arithmetic method proposed by Gamito & Maddock [GM07] only preserves inclusion under specific circumstances. Fortunately, with modifications ensuring that the last error term is positive-definite, a formulation similar to that of Messine et al. [Mes02] implements a correct inclusion for all compositions of AA operations. In adopting such an arithmetic, we implement a robust reduced AA suitable for ray casting on the GPU. Particularly for complex implicits requiring cross-multiplication between interval entities, this yields more correct results at lower required precision than standard IA, and superior frame rates for most functions.

7.2.1 Application pipeline

As input, the user must simply specify an implicit function, a domain $\Omega \subset \mathbb{R}^3$, and a termination precision ε. User-specified variables are stored on the CPU and passed dynamically to Cg as uniform parameters.

7.2.1.1 Meta-programming with Cg

Some runtime options, such as the implicit function, choice of inclusion algebra, or shading modality, are compiled directly into the Cg shader through meta-programming. In simple cases, the CPU merely searches for a stub substring within a base shader file, and replaces it with Cg code corresponding to the selected option. The most complicated meta-programming involves creating routines for function evaluation. Given an implicit function, we generally require two routines to be created within the shader: one evaluating the implicit f, and another evaluating an inclusion function, the interval or affine extension F. We use a simple recursive-descent parser to generate these routines in the output Cg shader. Alternately, we allow the user to provide "inline" Cg code, which can be useful in optimizing performance of implicits with repeated identical blocks of terms, and expressing special-case CSG models.

7.2.1.2 Rasterization

Though our system is built on top of OpenGL, we use the fixed-function rasterization pipeline very little. Given a domain $\Omega \subset \mathbb{R}^3$ specified by the user, we simply rasterize

that bounding box once per frame. We specify the world-space box vertex coordinates as texture coordinates as well. These are passed straight through a minimal vertex program, and the fragment program merely looks up the automatically interpolated world-space entry point of the ray and the bounding box. By subtracting that point from the origin, we generate a primary camera ray for each fragment.

7.2.2 Interval arithmetic library

Implementing an IA library is straightforward in Cg. Most operations employed in interval arithmetic (such as min and max) are highly efficient on the GPU, and swizzling allows for graceful SIMD computation (Algorithm 3). Transcendental functions are particularly efficient for both their floating-point and interval computations. Moderate integer powers are yet more efficient, thanks to unrolling multiplication chains via meta-programming and the Russian peasants algorithm; and a bound-efficient IA rule for even powers.

Algorithm 3 Interval Arithmetic examples.

```
typedef float2 interval;

interval iadd(interval a, interval b) {
  return interval( add(a.x, b.x), add(a.y, b.y) );
}
interval imul(interval a, interval b) {
  float4 lh = a.xxyy * b.xyxy;
  return interval(min(lh.x, min(lh.y, min(lh.z, lh.w))),
                  max(lh.x, max4(lh.y, max(lh.z, lh.w))));
}
interval ircp(const float inf, interval i) {
  const bool ic0 = (i.x <= 0 && i.y >= 0);
  return ( (i.x <= 0 && i.y >= 0) ?
           interval(-inf, inf) : 1/i.yx );
}
```

7.2.3 Reduced affine arithmetic library

In implementing our RAA library on the GPU, we adopt a formulation similar to AF1 in Messine et al. [Mes02], with changes to the absolute value bracketing that are mathematically equivalent but slightly faster to compute. In AF1, for some constant n a reduced affine form is given as:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i e_i + x_{n+1} e_{n+1}$$

We shall recall here the arithmetic operations:

$$\mathbf{c} \times \hat{x} = (\mathbf{c}x_0) + \sum_{i=1}^{n} \mathbf{c}x_i e_i + |cx_{n+1}| e_{n+1}$$

$$\mathbf{c} \pm \hat{x} = (\mathbf{c} \pm x_0) + \sum_{i=1}^{n} x_i e_i + |x_{n+1}| e_{n+1}$$

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i) e_i + (x_{n+1} + y_{n+1}) e_{n+1}$$

$$\hat{x} \times \hat{y} = (x_0 y_0) + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i) e_i + (|x_0 y_{n+1}| + |y_0 x_{n+1}| + \mathrm{rad}(\hat{x})\mathrm{rad}(\hat{y})) e_{n+1}$$

We implemented this formulation with $n = 1$ using a float3 to represent the reduced affine form. We also experimented with $n = 2$ (float4), and $n = 6$ (a double-float4 structure). For all the functions in our collection, the float3 version delivered the fastest results by far. We also found that the computational overhead of the bound-improved AF2 formulation [Mes02] was too high to be efficient. Examples of the float3 version are given in Algorithm 4.

The float3 implementation of AF1 makes for a versatile and fast reduced affine arithmetic. Particularly for functions with significant multiplication between non-correlated affine variables, such as the Mitchell function or the Barth implicits involving cross-multiplication of Chebyshev polynomials, significant speedup can be achieved over standard IA.

7.2.4 Numerical considerations

A technical difficulty arises in the expression of infinite intervals, which may occur in division; and empty intervals that are necessary in omitting non-real results from a fractional power or logarithm. While these are natively expressed by nan on the CPU, GPU's are not always IEEE compliant. The NVIDIA G80 architecture correctly detects and propagates infinity and nan, but the values themselves (inf $= 1/0$ and nan $= 0/0$) must be generated on the CPU and passed into the fragment program and subsequent IA/AA calls.

Algorithm 4 Reduced Affine Arithmetic examples.

```
typedef float3 raf;

raf interval_to_raf(interval i){
  raf r;
  r.x = (i.y + i.x);
  r.y = (i.y - i.x);
  r.xy *= .5; r.z = 0; return r;
}
float raf_radius(raf a){
  return abs(a.y) + a.z;
}
interval raf_to_interval(raf a){
  const float rad = raf_radius(a);
  return interval(a.x - rad, a.x + rad);
}
raf_add(raf a, raf b){
  return a + b;
}
raf_mul{raf a, raf b){
  raf r;
  r.x = a.x * b.x;
  r.y = a.x*b.y + b.x*a.y;
  r.z = abs(a.x)*b.z + abs(b.x)*a.z +
        raf_radius(a)*raf_radius(b);
  return r;
}
```

Conventionally, IA and AA employ a rounding step after every operation, padding the result to the previous or next expressible floating point number. We deliberately omit rounding – in practice the typical precision ε is sufficiently large that rounding has negligible impact on the correct computation of the extension F. However, numerical issues can be problematic in certain affine operations: RAA implementations of square root, transcendentals and division itself all rely on accurate floating point division for computing the regression lines approximating affine forms. Though inclusion-preserving in theory, these methods are ill-suited for inaccurate GPU floating point arithmetic; and a robust strategy to overcome these issues has not yet been developed for RAA. We therefore resort to interval arithmetic for functions that require regression-approximation AA operators.

7.2.5 Traversal

With the IA/RAA extension and a primary ray generated on the fragment unit, we can perform ray traversal of the domain $\Omega \subset \mathbb{R}^3$. Though not as trivial as standard numerical bisection for root finding, the ray traversal algorithm is nonetheless elegantly simple (see Algorithm 5).

7.2.5.1 Initialization

We begin by computing the exit point p_{exit} of the generated ray and the bounding box Ω. We reparameterize the ray as $\vec{r}(t) := \vec{p}_{enter} + t(\vec{p}_{exit} - \vec{p}_{enter})$. The interval $\underline{t}$ along the ray intersecting Ω is now $[0,1]$. We now perform a first rejection test outside the main loop.

7.2.5.2 Rejection test

In the rejection test, we evaluate the IA/AA extensions of the ray equation to find X, Y and Z over $\underline{t}$, and use these (as well as scalars w, r_i for time and other animation variables) to evaluate the extension of our implicit function. The result gives us an interval or affine approximation of the range F. If $0 \in F$, then we must continue to bisect and search for roots. Otherwise, we may safely ignore this interval and proceed to the next, or terminate if it is the last.

7.2.5.3 Main loop

If the outer rejection test succeeds, we compute the effective bisection depth required for the user-specified precision ε. This is given by

$$d_{max} := \log_2\left(\frac{||\vec{p}_{exit} - \vec{p}_{enter}||}{\varepsilon}\right) \tag{7.1}$$

We initialize our depth $d = 0$, and distance increment, $t_{incr} = 0.5$. Now, recalling the bisection interval $\underline{t}$, we set $\bar{t} := \underline{t} + t_{incr}$. We then perform the rejection test on this new $\underline{t}$. If the test succeeds, we either hit the surface if we have reached $d = d_{max}$, or recurse to the next level by setting $t_{incr} := t_{incr}/2$, and incrementing d.

If the rejection test fails, we proceed to the next interval segment at the current depth level by setting $\bar{t} := \underline{t}$. Within the main loop, we now perform another loop to back-recurse to the appropriate depth level.

7.2.5.4 Back-recursion loop

In back-recursion, we basically decrement the depth (and update t_{incr}) as long as we have visited both "sides" of the bisection tree at the current depth. Conventionally, this algorithm is performed by caching an array in place of a recursion stack. As this is ineffective on the GPU, we note that we can perform a similar query by a floating-point modulus: checking if $(\underline{t} \ \% \ 2t_{incr} == 0)$. Currently on the G80, the fastest method proves to be performing division and examining the remainder. Back-recursion proceeds iteratively until either one side of the bisection has not yet been visited, or $d = -1$.

7.2.6 Traversal meta-programming

The traversal algorithm largely remains static, but some functions and visualization modalities require special handling. To render functions containing division operations, we must check whether intervals are infinitely wide before successfully hitting, as detailed in Knoll et al. [KHH*07]. Multiple iso-values and transparency require modifications to the rejection test and hit registration, respectively, as discussed in Section 7.3.2. More generally, modifications to the traversal algorithm are simple to implement via "inline" implicit files (Section 7.2.1). We allow the user to directly program behavior of the rejection test, hit registration and shading. This is particularly useful in rendering special-case constructive solid geometry objects (Fig. 7.8).

7.2.7 Shading

Phong shading requires a surface normal, specifically the gradient of the implicit at the found intersection position. We find central differencing to be more than adequate, as it requires no effort on the part of the user in specifying analytical derivatives, nor special meta-programming in computing separable partials via automatic differentiation. By default we use a stencil width proportional to the traversal precision ε; variable width is often also desirable [KHH*07].

7.3 Results

All benchmarks are measured in frames per second on an NVIDIA 8800 GTX, at 1024x1024 frame buffer resolution.

		CPU	GPU		
		$\varepsilon = 2^{-11}$			correct
function (fig)	**degree**	IA		RAA	IA/RAA
sphere	2	15	75	147	165
steiner (7.3)	4	7.5	34	40	38
mitchell (7.2)	8	5.2	16	58	60
teardrop (7.4a)	4	5.5	102	115	121
4-bretzel (7.5a)	12	13	78	48	90
klein b. (7.4b)	6	11	30	110	101
tangle (7.5b)	4	3.2	15	68	71
decocube (7.7)	4	5.5	28	27	28
barth sex. (7.6l)	6	7.4	31	76	88
barth dec. (7.6r)	10	0.9	4.9	15.6	15.6
superquadric	200	18	119	8.3	108
icos.csg (7.8l)	na	-	13.3	-	13.3
sq.csg (7.8r)	na	-	8.9	-	7.2
sin.blob (7.1)	na	-	6.0	-	6.0
cloth (7.9l)	na	-	38	-	44
water (7.9r)	na	-	37	-	44

Table 7.1: *Single-ray casting performance in fps.* We indicate the figure illustrating each function where available. We compare the SSE IA implementation of Knoll et al. [KHH*07] on four 2.33 GHz cores; and our IA and RAA implementations on the G80 GPU, using a common $\varepsilon = 2^{-11}$. The last column shows frame rate at the lowest ε yielding visually correct results, using either IA or RAA.

7.3.1 Performance

Table 7.1 shows base frame rates of a variety of implicits using single ray-casting and basic Phong shading. Performance on the NVIDIA 8800 GTX is up to 22× faster than the SSE method of Knoll et al. [KHH*07] on the 4-core Xeon 2.33 GHz CPU workstation. Frame rate is determined both by the bound tightness of the chosen inclusion extension, and the computational cost of evaluating it. In practice, the order of the implicit has little impact on performance. Equations for most functions can be found in [KHH*07] and [Res].

7.3.1.1 IA vs RAA

For typical functions with fairly low-order coefficients and moderate cross-multiplication of terms, reduced affine arithmetic is generally $1.5-2\times$ faster than interval arithmetic.

For functions with high bound overestimation, such as those involving multiplication of large polynomial terms (the Barth implicits) or Horner-like forms, RAA is frequently 3 to 4 times faster. Conversely, thanks to an efficient inclusion rule for integer powers, IA remains far more efficient for superquadrics, as evident in Table 7.1. As explained in Section 7.2.4, IA is currently required for extensions of division, transcendentals, and fractional powers.

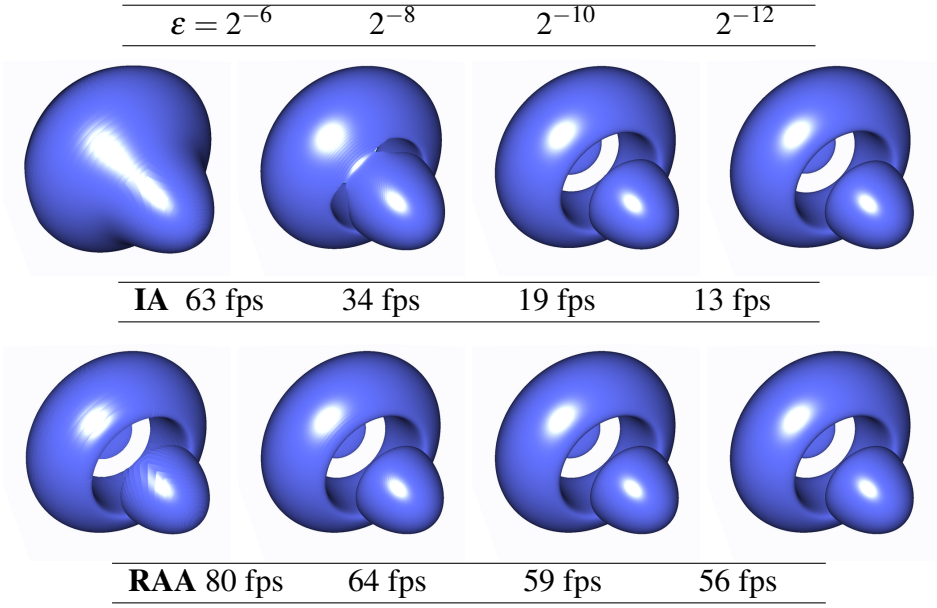


Figure 7.2: *IA (top) and RAA (bottom) at various* ε.

7.3.1.2 Precision and quality

Concerning visual quality and robustness, our findings for IA are generally in line with those of the CPU implementation of Knoll et al. [KHH*07]. For the analytic functions it supports, and particularly pathological cases for IA, RAA usually converges far more quickly to the correct solution, given lesser bound overestimation at low precision ε. In addition, refining ε has little impact on frame rate once RAA has effectively converged (Fig. 7.2).

7.3.1.3 Correctness and robustness

As it entails more floating-point computation than IA, RAA has worse numerical conditioning. This is particularly noticeable with more precise ε. Fig. 7.3 illustrates the challenge in robustly ray casting the Steiner surface with IA and AA. Both inclusion

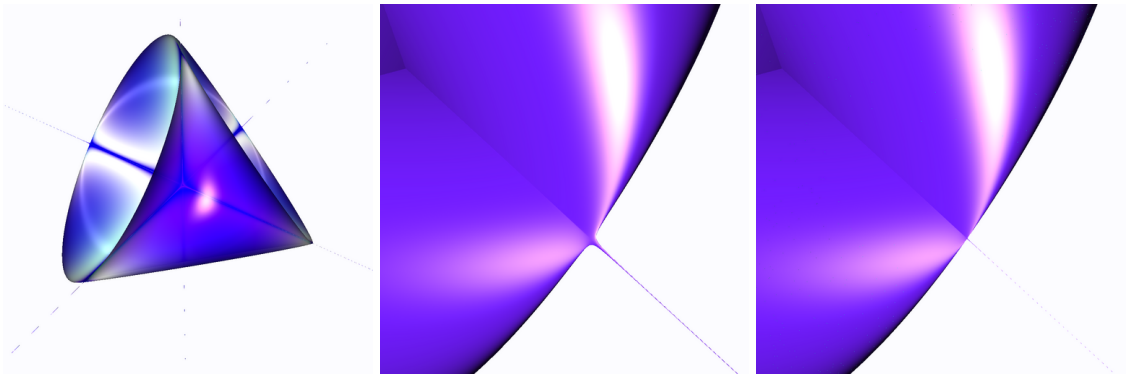


Figure 7.3: *Fine feature visualization in the Steiner surface.* Left to right: shading with depth peeling and gradient magnitude coloration; close-up on a singularity with IA at $\varepsilon = 2^{-18}$; and with RAA at the same depth.

methods identify the infinitely thin surface regions at the axes, but fairly precise $\varepsilon < 2^{-18}$ is required for correct close-up visualization of these features. Affine arithmetic yields a tighter contour of the true zero-set than IA, but with some speckling artifacts. Nonetheless, both IA and RAA yield more robust results than non-inclusion ray casting methods [LB06], or inclusion-based extraction [PLLdF06] on the teardop (Fig. 7.4a).

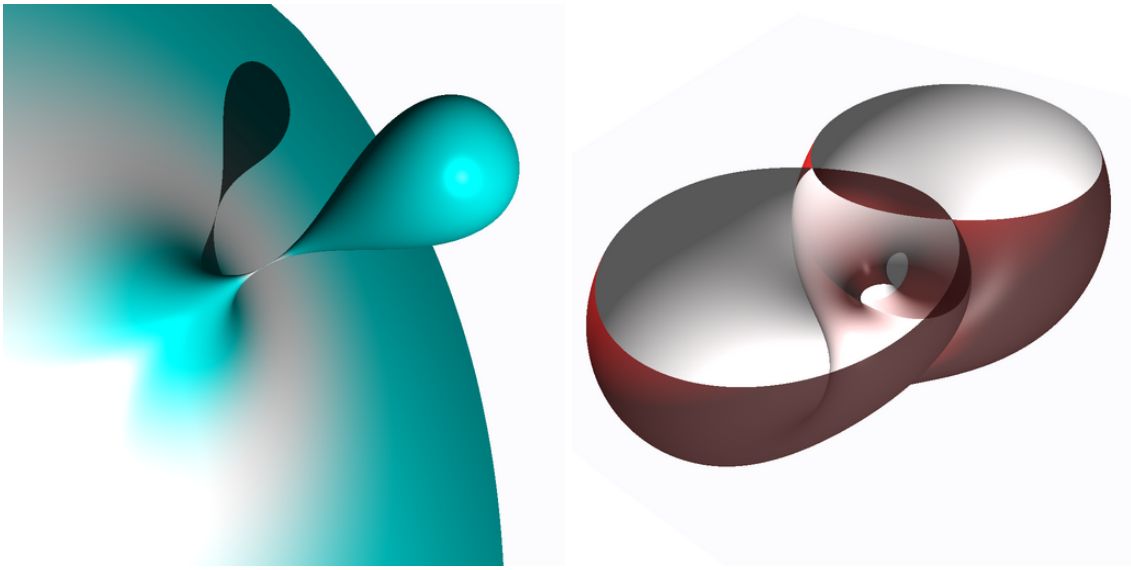


Figure 7.4: *Shading effects: shadows and transparency.* Left: (a) shadows on the teardop (40 fps); Right: (b) transparency on the klein bottle (41 fps).

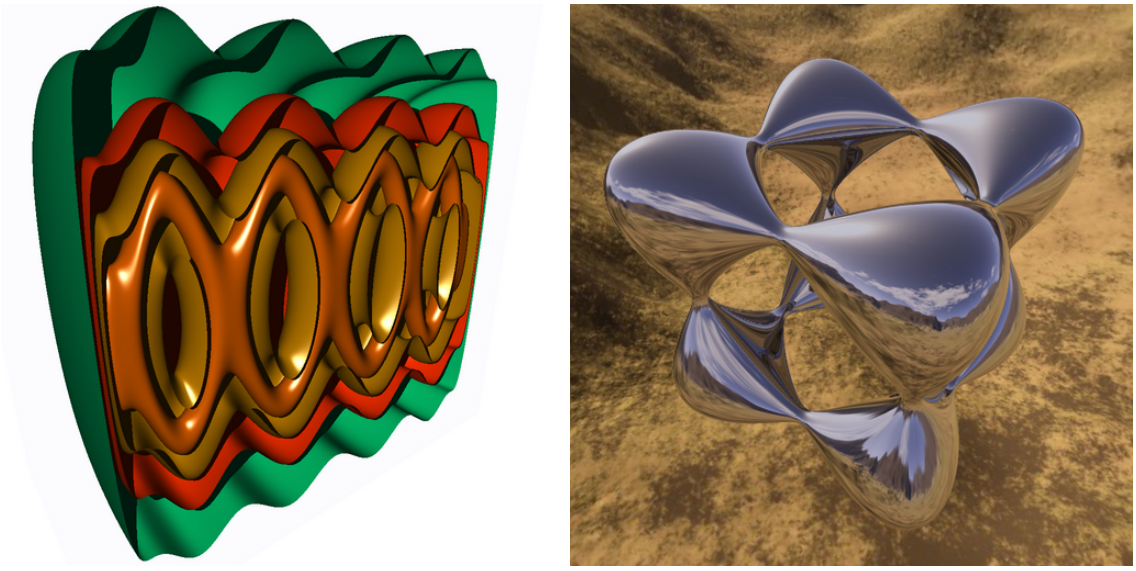


Figure 7.5: *Shading effects: multiple iso-values and reflections.* Left: (a) shadows and multiple isovalues of the 4-Bretzel (18 fps); Right (b) the tangle with up to six reflection rays (44 fps).

7.3.2 Shading modalities

As our algorithm relies purely on ray-tracing, we can easily support per-pixel lighting models and multi-bounce effects, many of which would be difficult with rasterization (Figs. 7.4 and 7.5). We briefly describe the implementation of these modalities, and their impact on performance.

7.3.2.1 Shadows

Non-recursive secondary rays such as shadows are straightforward to implement. Within the main fragment program, after a successfully hit traversal, we check whether $\vec{N} \cdot \vec{L} > 0$, and if so, perform traversal with a shadow ray. To ensure we do not hit the same surface, we cast the shadow from the light to the hit position, and use their difference to reparameterize the ray so that $\bar{t} = [0,1]$, as for primary rays. Shadows often entail around $20-50\%$ performance penalty. One can equally use a coarser precision for casting shadow rays than primary rays. With RAA, contour overestimation is seldom a problem even at $\varepsilon > .01$; this can decrease the performance overhead to $10-30\%$.

7.3.2.2 Transparency

Transparency is also useful in visualizing implicits, particularly functions with odd connectivity or disjoint features. With ray casting, it is simple to implement front-to-back, order-independent transparency, in which rays are only counted as transparent if a surface behind them exists. Our implementation lets the user specify the blending opacity, and casts up to four transparent rays. This costs around $3\times$ as much as one primary ray per pixel.

7.3.2.3 Multiple isosurfaces

One may equally use multiple iso-values to render the surface. This is significantly less expensive than evaluating the CSG object of multiple surfaces, as the implicit extension need only be evaluated once for the surface. The rejection test then requires that *any* of those iso-values hit. At hit registration, we simply determine which of those iso-values hit, and flag the shader accordingly to use different surface colors. With no other effects, multiple iso-values typically entail a cost of anywhere from $10-40\%$.

7.3.2.4 Reflections

Reflections are a good example of how built-in features of rasterization hardware can be seamlessly combined with the implicit ray casting system. Looking up up a single reflected value from a cubic environment map invokes no performance penalty. Tracing multiple reflection rays in an iterative loop is not significantly more expensive ($20-30\%$), and yields clearly superior results (Fig. 7.5b).

7.3.3 Application

7.3.3.1 Mathematical visualization

The immediate application of this system is a graphing tool for mathematically interesting implicits in $3D$ and $4D$. Ray casting ensures view-dependent visualization of infinitely thin features, as in the teardrop and Steiner surfaces. It is similarly useful in rendering singularities – Fig. 7.6 shows the Barth sextic and decic surfaces, which contain the maximum number of ordinary double points for functions of their respective degrees in $\mathbb{R}^3$.

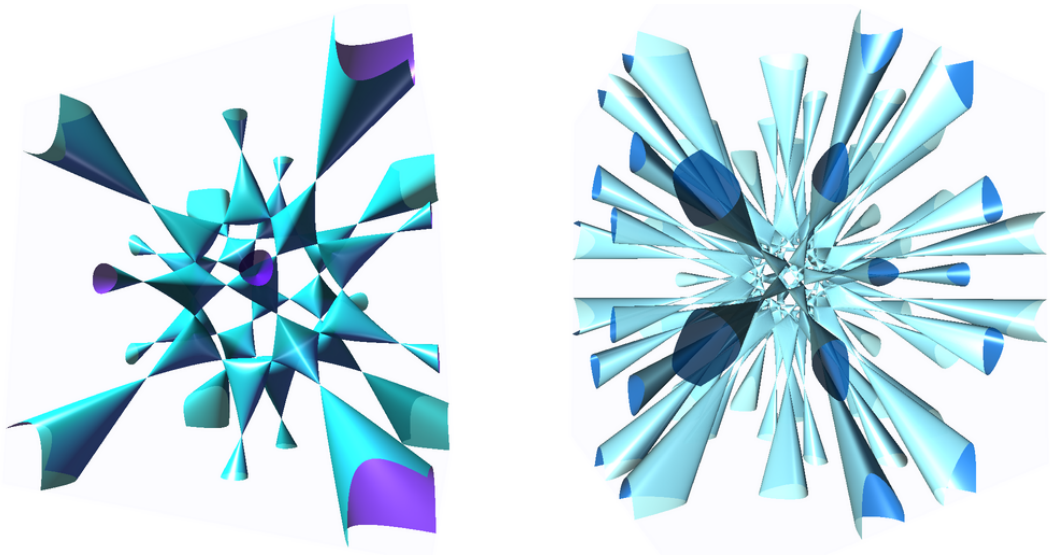


Figure 7.6: *The Barth sextic and decic surfaces.*

7.3.3.2 Interpolation, morphing and blending

Implicits inherently support blending operations between multiple basis functions. Such forms need only be expressed as an arbitrary $4D$ implicit $f(x,y,z,w)$, where w varies over time. As ray-tracing is performed purely on-the-fly with no pre-computation, we have great flexibility in dynamically rendering these functions. Useful morphing methods include product implicits, linear interpolation between surfaces; and gaussian or sigmoid blending, shown in Fig. 7.7 between the decocube and the sphere.

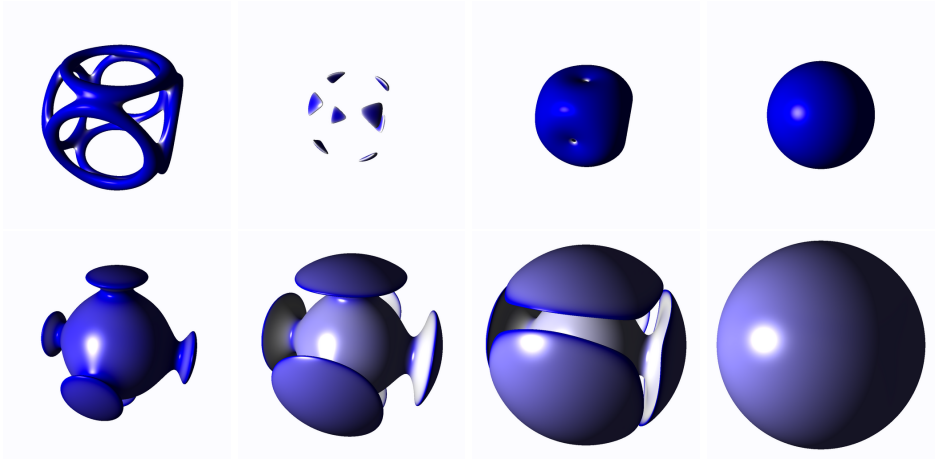


Figure 7.7: $4D$ *sigmoid blending of the decocube and a sphere*, with interpolation and extrapolation phases, running at $33-50$ fps.

7.3.3.3 Constructive Solid Geometry

Multiple-implicit CSG objects can accomplish similar effects to product surfaces and sigmoid blending, but with C^0 trimming. In particular, CSG intersection allows us to specify 3-manifold level sets as arbitrary conditions over an implicit or set of implicits. Given an implicit $f(\omega)$ and a condition $g(\omega)$, inclusion arithmetic allows us to verify $g_+ = \{\underline{g}(\omega) > 0\}$ or $g_- = \{\overline{g}(\omega) < 0\}$, given the interval form of the inclusion extension G over an interval domain $\omega \subseteq \Omega$. Then, one can render $f \cap g_+$ or $f \cap g_-$ for arbitrary level sets of g. By determining which level sets are intersected inside the traversal, we can shade components differently as desired (Fig. 7.8).

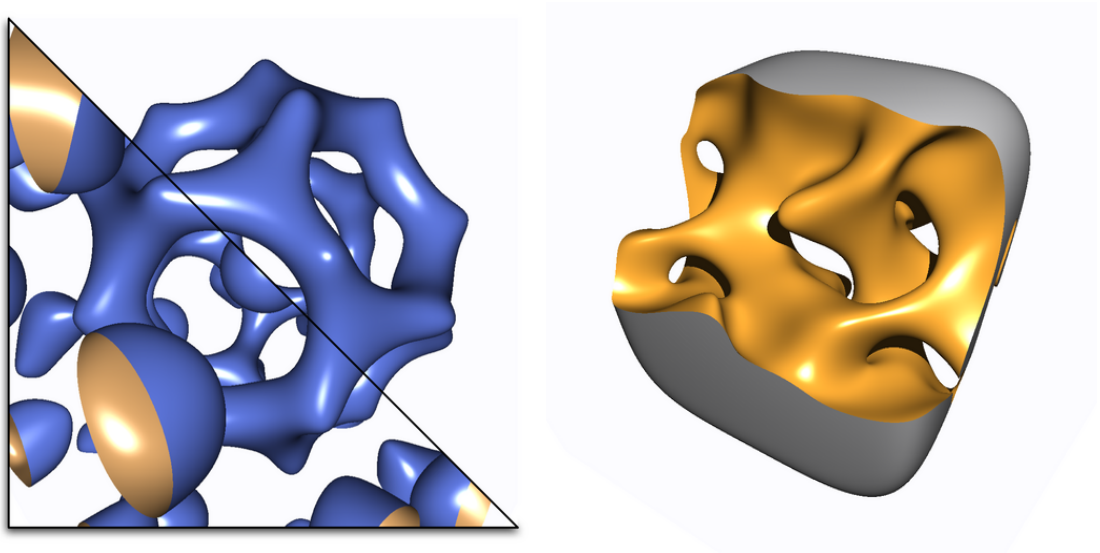


Figure 7.8: *CSG surfaces using level-set conditions.*

7.3.3.4 Procedural geometry

Implicits have historically been non-intuitive and unpopular for modeling large-scale objects. However, the ability to render dynamic surfaces and natural phenomena using combinations of known closed-form expressions could prove useful in modeling small-scale and dynamic features. Sinc expressions, for example, define closed-form solutions of simple wave equations for modeling water and cloth (Fig. 7.9). Previous applications of implicit hyper-textures focused on blended procedural noise functions [Per85, GM07]. Recently, implicits based predominately on generalized sinusoid product forms similar to that in Fig. 7.1 have been used within some modeling communities [k3d]. Arbitrary implicits are intriguing in their flexibility, and ray casting promises the ability to dynamically render entire new classes of procedural geometries, independently from any polygonal geometry budget.

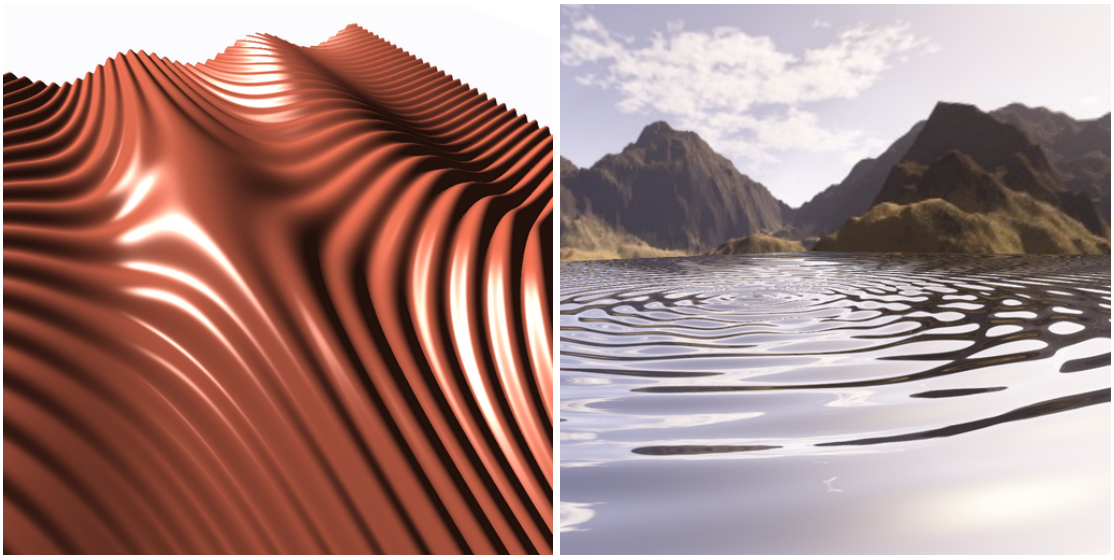


Figure 7.9: *Sinusoid procedural geometry for dynamic simulation of cloth and water.* With IA, these surfaces render at 38 and 37 fps respectively.

7.4 Conclusion

We have demonstrated a fast, robust and general algorithm for rendering implicits on the GPU. Performance was achieved by devising a stackless-recursion ray traversal algorithm; and a shader-language implementation of a generally correct reduced affine arithmetic, which improves performance for complex functions with high bound overestimation. We have shown the flexibility and potential of this approach for mathematical function visualization and rendering of procedural geometry.

Some drawbacks should be noted. While general, correct and fast, IA/AA methods still require copious computation compared to other approaches involving basis approximations, distance functions, or point sampling. A comprehensive comparison using optimized implementations of these methods would be useful. Also, while robust per-ray, our system ignores aliasing issues on boundaries and sub-pixel features. To robustly reconstruct the surface between pixels, one would require beam tracing and likely super-sampling.

Many extensions to this implementation would be useful. Further development of approximating regression operations for RAA could allow for correct and fast affine extensions of transcendental functions and their compositions. Of more general importance would be support on the application front-end for point, mesh or volume data, which could then be filtered and reconstructed by arbitrary implicits. This could be accomplished either by extending the rasterization system and restricting the application to ray casting; or by attempting a full ray-tracing system, with hierarchical acceleration structures, for the

fragment shader. Though applied here to general implicits, inclusion methods could potentially be employed in rendering arbitrary parametric or free-form surfaces. Besides visualizing implicits for their own sake and for data reconstruction, long-term potential of these methods depends on the modeling and content-creation communities. Implicits have historically been non-intuitive and unpopular in modeling large-scale geometry. However, the ability to model dynamic surfaces and natural phenomena using combinations of known closed-form expressions might gain traction for modeling small-scale and dynamic features, particularly in conjunction with efficient per-pixel ray casting methods that entail computation but no added geometry.

Algorithm 5 Traversal algorithm with RAA.

```
float traverse(float3 penter, float3 pexit, float w,
   float max_depth, float eps, float nan, float inf){
  const float3 org = penter;
  const float3 dir = pexit-penter;
  interval t(0,1);
  raf F, it, ix, iy, iz;
  //rejection test
  ix = raf_add(org.x, raf_mul(it, dir.x));
  iy = raf_add(org.y, raf_mul(it, dir.y));
  iz = raf_add(org.z, raf_mul(it, dir.z));
  F = evaluate_raf(ix, iy, iz, w, nan, inf);
  if (raf_contains(F, 0)){
    int d=0;
    float tincr = .5;
    const int dlast = log2(length(dir)/epsilon);
    //main loop
    for(;;){
      t.y = t.x + tincr;
      (compute ix, iy, iz, F again for rejection test)
      if (raf_contains(F, 0)){
        if (d==dlast){ return t.x; /*hit*/}
        else{ tincr *= .5; d++; continue; }
      }
      t.x = t.y;
      //back-recursion
      float fp = frac(.5*t.x/tincr);
      if (fp < 1e-8){
        for(int j=0; j<24; j++){
          tincr *= 2;
          d--;
          fp = frac(.5*t.x/tincr);
          if (d==-1 || fp > 1e-8) break;
        }
        if (d==-1) break;
      }
    }
  }
  return -1;  //no hit
}
```

Conclusion

In this Part we introduced three new contributions to the topic *Feature Based Visualization* within scalar fields, all based on subdivision and interval arithmetic: the CAPS algorithm to compute the arrangement of arbitrary implicit curves and two algorithms for ray casting arbitrary implicit functions, one interactively on the CPU, the other one real-time on the GPU.

Part III

Features in tensor fields

Introduction

In this study we are concerned with comparative visualization of difference tensor fields motivated by a mechanical engineering visualization need. As we will detail in Chapter 8.1 Section 8.1.1, the problem consists of analyzing the influence of the consistent stress enhancement (representing a modified time quadrature rule) based on the spatial distribution of the tensor-valued difference between the standard quadrature rule and the favored nonstandard quadrature rule.

We present two new paradigms for comparative tensor visualization. One is based on an interpretation of the symmetric tensor field as a vector field; the other one is based on the tensor's invariants.

This comparative analysis is carried out using several visualization tools tailored to set apart spatial and temporal patterns that allow to deduce the influence of both step size and material constants on the stress enhancement. The resulting visualizations indeed confirm the physical intuition by pointing out locations where interesting changes happen in the data. This work has been done in collaboration with Gerd Reis and Rouven Mohr (TU Kaiserslautern).

Chapter 8

Comparative tensor visualization

In this Chapter, we first motivate our work starting by the description of the mechanical engineering problem. We also present the different key steps towards the new visualization paradigms. One is based on an interpretation of the symmetric tensor field as a vector field; the other one is based on the tensor's invariants.

The following Section motivates our work from the engineering point of view as detailed in [MBH*07].

8.1 Motivation for tensor visualization

8.1.1 A mechanical engineering problem

Nowadays, the design of so-called consistent time-stepping schemes that basically feature a physically correct time integration, is still a state-of-the-art topic in the area of numerical mechanics. Within the proposed framework for finite elasto-plasto-dynamics, the spatial as well as the time discretization rely both on a Finite Element approach and the resulting algorithmic conservation properties have been shown to be closely related to quadrature formulas that are required for the calculation of time-integrals. Thereby, consistent integration schemes, which allow a superior numerical performance, have been developed based on the introduction of an enhanced algorithmic stress tensor, compare [MMS06b]-[MMS07].

It is well-known in literature that the performance of classical time integration schemes for structural dynamics, as for instance developed in [New59], is strongly limited when dealing with highly nonlinear systems. In a nonlinear setting, sophisticated numerical

techniques are required to satisfy the classical balance laws, as for instance balance of linear and angular momentum or the classical laws of thermodynamics. Nowadays, energy and momentum conserving time integrators for dynamical systems, like multi-body systems or elasto-dynamics, are well-established in the computational dynamics community, compare e.g. [ST92]. In contrast to the commonly used time discretization based on Finite Differences, one-step implicit integration algorithms relying on Finite Elements in space and time were developed, for instance, in Betsch and Steinmann [BS01]. Therein, conservation of energy and angular momentum have been shown to be closely related to quadrature formulas required for numerical integration in time. In this context, specific algorithmic energy conserving schemes for hyper-elastic materials can be based on the introduction of an enhanced stress tensor for time shape functions of arbitrary order, compare Gross *et al.* [GBS05]. Recently, a generalization of these Galerkin-based concepts to finite elasto-plasto-dynamics has been worked out by Mohr *et al.* [MMS06b, MMS07].

However, it has been shown by many authors that the introduction of a modified stress tensor represents an appropriate tool to design specific conserving respectively consistent time-stepping schemes, compare e.g. [Arm06, Gon00, GBS05, ML02, MMS06a, NSP06]. Nevertheless, the influence of this stress enhancement is not completely understood yet. One very interesting aspect, that has not been addressed in the literature so far, is for instance the spatial distribution of the difference tensor between the stresses of the continuum model and the enhanced stresses for the time-stepping. In this context, some basic discussions have already been encouraged in [MMS06b] based on an 'ad hoc' visualization approach that, however, only provides very limited information. In this contribution, several more sophisticated techniques have been developed to visualize the difference between both second-order tensor fields. It will be demonstrated by means of representative parameter studies that the proposed concepts indeed represent an effective tool to understand better the numerical behavior of the underlying time-stepping scheme.

8.1.2 From practice to theory

This experimental study was initiated by discussions with mechanical engineers who expressed a need of visualization tools, even simple ones. A first step was to experiment with Matlab in order to rapidly obtain a proof of concept. To state the problem with simple words: Given two $2D$ symmetric tensor fields, we are interested in regions where the most important changes happen between them.

First, for each tensor field, the orthogonal eigenvectors (normalized using the eigenvalues) were displayed, leading to a dense image. Thus, our first concern was the occlusion problem and it was clear that a scaling needed to be applied. As will be shown in Chapter 8, a linear and a logarithmic have been used, each for a different visualization scheme. Then, for each point of the tensor field, we used the eigenvectors to build plain ellipses, as being

a standard approach in tensor visualization [Kin04]. This method still wasn't very satisfying as the occlusion problem wasn't solved, especially because we were still considering both tensor fields on the same image, despite being interested only in their difference. At this point, we could have simply computed the difference tensor field component-wise and then use the ellipse-based visualization to represent only the difference tensor field. The following question arises: What does the component-wise tensor difference field mean? Indeed it has no physical meaning. Though convenient, it would be difficult to interpret.

Precisely at this stage of the study came a thought: we are concerned with $2D$ symmetric tensor fields, which means that we only have three components for each tensor, which can be represented by an object living in $3D$ space. Thus, in theory, we could represent a $2D$ symmetric tensor by a $3D$ vector. What is more natural than vectors to represent a difference (or motion)? Indeed, the $3D$ vector seemed intuitive for visualizing the difference field as between two $3D$ vectors (representing $2D$ symmetric tensors) that are completely opposed in their orientation, the resulting difference vector would be very important; and on the other hand, if there is no (or almost no) change between the two vectors, the difference vector would be very small. We first visualized this $3D$ vector field using Matlab, which confirmed the intuition (see Fig. 8.1). The results seemed to point out regions where the tensor difference was the most important.

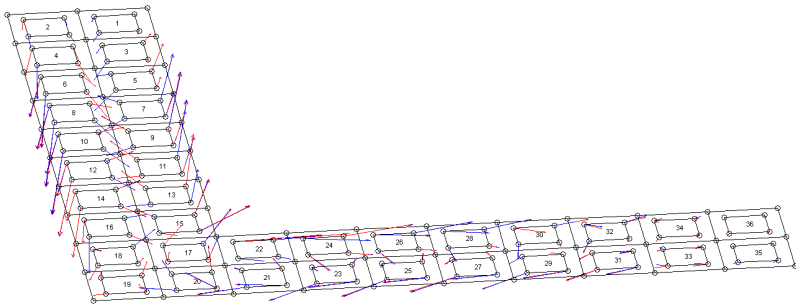


Figure 8.1: *Early result showing the two (scaled)* $3D$ *vector fields.*

We extended this approach by connecting the neighboring points of the 'L', four by four, as patches in order to see more structure. At this point, the problem was that we started with $2D$ information and ended up with $3D$ information. Notice that the direction of those $3D$ vectors doesn't help much for the understanding as it has no physical and, above all, no intuitive meaning; only the norm of the resulting $3D$ vector field was somehow inter-

pretable. Realizing that, we reconsidered the visualization scheme for $2D$: we projected the previously described patches back in $2D$ and displayed circles whose radii correspond to the norm of the $3D$ vectors. This scheme, despite its lack of rigor, happened to be the most helpful for understanding the data. It will be further detailed in Section 8.4.1.

Another approach consisted of visualizing the difference invariants of the two tensor fields. Indeed, each tensor field provides two scalar invariants. Once the information reduced from tensor to scalar, visualization becomes much easier. First, as we did with the $3D$ vectors, we can simply compute the differences between those invariants and we end up with two scalars for the difference field. These difference invariants could be visualized in many ways; we chose the ellipsoid scheme. Note that we don't consider at all the tensor's eigenvectors here. We mapped those ellipsoids with the previously described patches to facilitate visual interpretation. This invariant-based visualization scheme is presented in Section 8.4.2.

8.2 Comparative tensor visualization

Previously, the essential ingredients for a thermodynamically consistent time-integration have been presented. Thereby, the crucial difference between the standard Gauss quadrature rule and the more sophisticated nonstandard quadrature rule is directly related to the tensor-valued difference between the standard stresses of the continuum model S and the algorithmic stresses S^{alg}, involving the enhancement tensor (8.7). One interesting aspect, that has not been addressed in the literature before, is the spatial distribution of the corresponding difference tensor field, whereby we are quite optimistic that such a comparison between both tensor fields would provide a much deeper insight into the numerical behavior of the related time-stepping schemes. In this context important issues are for instance: the correlation between the corrections and the underlying deformation, the influence of the time-step size or the material properties, the evolution of the corrections in time, the existence of characteristic patterns within the difference tensor field, ... However, a satisfying visualization is a non-trivial task, dealing with two different tensor fields and a large number of time steps. A further difficulty is the fact that a direct physical interpretation of the enhancement term ${}^{el}S^{enh}$ and the algorithmic stress tensor S^{alg} respectively is not valid since it represents only a numerical tool to support the quadrature rule, approximating time-integrals. In the following, we will focus on the development and the comparison of various visualization approaches to better understand the influence of the correction on the time-quadrature rule.

To generate a benchmark data set, we calculated the motion of a 'Flying L' based on 36 4-node Finite Elements in space, using linear Finite Elements in time. For further set-up details we refer to Mohr *et al.* [MMS06b]. In view of the above-mentioned issues,

the calculations have been performed with stiff/non-stiff material properties, involving $[\lambda,\mu] = [10000,5000]/[1000,500]$, and with large ($h_n = 0.4$) respectively small ($h_n = 0.04$) time-step sizes. Since the considered tensor fields are both defined in the reference configuration $\mathscr{B}_0$ only the undeformed configuration is of interest and, consequently, the actual deformation of the body is not shown (Figure 8.2).

Figure 8.2 shows a visualisation obtained with Matlab by representing the tensor with its two orthogonal, normalised eigenvectors based on the spectral decompositions

$$S^{alg} = \sum_{i=1}^{2} {}^{S}\lambda_i^{alg} N_i^{alg} \otimes N_i^{alg} \qquad\qquad S = \sum_{i=1}^{2} {}^{S}\lambda_i N_i \otimes N_i \tag{8.1}$$

This figure was our initial motivation for experimenting with more advanced visualizations, especially overcoming the occlusion problem. Moreover, Figure 8.2 shows the two tensor fields and not its difference. However, one natural possibility to reduce the complexity of information is to find an appropriate representation of the difference field, since basically the corrections are of particular interest.

Our goal is to provide as many visualization tools as possible to support the understanding of both the spatial distribution of the algorithmic enhancement terms and their effect on the stress field. Therefore we examine the data in a spatial context from different points of view, one focusing on the magnitude of numerical differences in the stress tensors S and S^{alg} such as tensor invariants, another focusing on differences in extracted entities like the $3D$ vectors. We combine basic visualization techniques such as color coding, transparency effects, and scaling together in order to provide the most helpful tools, thereby applying Information Visualization [Jac99] techniques.

We analyze a particular time step from data reflecting the simulation of stiff and non-stiff material with fine and coarse time resolution, resulting in four different configurations.

8.3 Background: Finite Elasto-Plasto-Dynamics

This Section introduces the necessary physical background as detailed in [MMS06b, MBH*07].

First, the nonlinear deformation map $\varphi(X,t) : \mathscr{B}_0 \times [0,T] \rightarrow \mathscr{B}_t$ is introduced as a mapping from the material to the spatial configuration. In the context of finite plasticity, the resulting deformation gradient $F := \nabla_X \varphi(X,t)$ is assumed to be multiplicatively decomposed into an elastic and a plastic part:

$$F \doteq F_e \cdot F_p \tag{8.2}$$

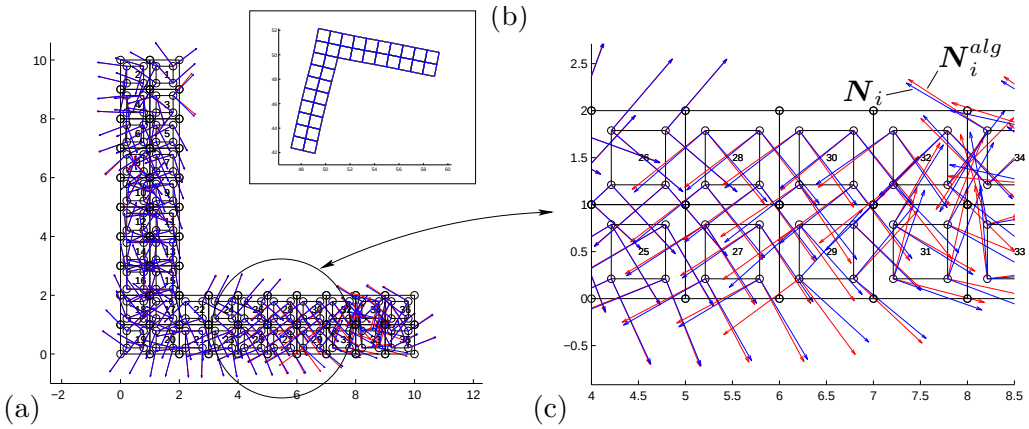


Figure 8.2: *(a) reference configuration* $\mathscr{B}_0$ with the eigenvectors $[N_i^{alg}, N_i]$ of the elastic-enhanced algorithmic stress tensor S^{alg} & the Piola Kirchhoff stress tensor S, *(b) deformed configuration* $\mathscr{B}_t$ after $10s$, *(c) zoom of the principal directions* $[N_i^{alg}, N_i]$.

In contrast to the modeling of elasticity, additional internal variables κ are included in the Helmholtz energy density $\psi(F,\kappa)$ for the plastic case to model the loading history. Moreover, it is accepted to introduce the so-called conjugated thermodynamical forces $\beta := -\nabla_\kappa \psi$ which render the dissipation inequality, namely $\mathscr{D} = \langle \beta, \dot{\kappa} \rangle \geq 0$. In view of a thermodynamically consistent modeling this dissipation inequality has to be respected not only by the continuum model, but also by the applied numerical integration scheme. In a next step, we apply a standard Finite Element discretization in space for the material configuration of a solid continuum body. Using the spatial approximations, the semi-discrete deformation map can be written by means of the spatial shape functions $N_A(X)$ in the form: $\varphi(X,t) = \sum_{A=1}^{n_{node}} q_A(t) N_A(X)$. Consequently, the approximations in space of the spatial velocity $v := \sum_{A=1}^{n_{node}} \dot{q}_A N_A$ and the right Cauchy-Green strain tensor $C := F^{\mathrm{t}} \cdot F = \sum_{A,B=1}^{n_{node}} q_A \cdot q_B \nabla N_A \otimes \nabla N_B$ can be computed straightforwardly. To obtain a semi-discrete system of equations of motion, we combine the placements of the spatial nodes $q = [q_1, ..., q_{n_{node}}]^{\mathrm{t}}$ and the nodal generalized momenta $p := \mathbb{M} \cdot \dot{q} = [p_1, ..., p_{n_{node}}]^{\mathrm{t}}$ to the vector $z := [q, p]^{\mathrm{t}}$. Furthermore, the sum of the kinetic energy $T(p) = \frac{1}{2}\, p \cdot \mathbb{M}^{-1} \cdot p$, the free energy $\Psi = \int_{\mathscr{B}_0} \psi \, \mathrm{d}V$ and possibly an external potential V^{ext} is defined as $H(q,p;\kappa) := T + \Psi + V^{ext}$. Inspired by the purely elastic case, the resulting equations of motion can still be written in a compact format of Hamilton-type

$$\dot{z}(t) = \mathbb{J} \cdot \nabla_z H(z;\kappa) \qquad \text{with} \qquad \nabla_z H = \begin{bmatrix} F^{int} - F^{ext} \\ \mathbb{M}^{-1} \cdot p \end{bmatrix}, \tag{8.3}$$

wherein we have incorporated the symplectic matrix $\mathbb{J}$ and the internal load vector $F^{int}(S)$, involving the Piola Kirchhoff stresses $S = 2\nabla_C \psi$. Next, the time discretization of the semi-discrete system of equations of motion (8.3) is considered. We start with a decom-

position of the time interval $[0,T] = \bigcup_{n=0}^{N} [t_n, t_{n+1}]$ and a map of each sub-interval to the reference time interval $[0,1]$ via the function $\alpha(t) := [t - t_n]/h_n$ based on the time-step size $h_n = t_{n+1} - t_n$. For the approximation in time a continuous Galerkin method – abbreviated by: cG(k)-method – is applied. Therefore, the time approximations of the unknown function $z^h = \sum_{j=1}^{k+1} M_j(\alpha)\, z_j$ and the test function $\delta z^h = \sum_{i=1}^{k} \widetilde{M}_i(\alpha)\, \delta z_i$ are introduced [1]. In a compact notation the resulting weak form in time is given by

$$\int_0^1 \left[\mathbb{J} \cdot \delta z^h \right] \cdot \left[\mathrm{D}_\alpha z^h - h_n \, \mathbb{J} \cdot \nabla_z H(z;\kappa) \right] \mathrm{d}\alpha = 0 \,. \tag{8.4}$$

Obviously, Equation (8.4) involves time-integrated internal load vectors, which will be referred to as $\bar{F}^{int}_{Ai}$ related to the spatial node A. As discussed for instance in Mohr *et al.* [MMS06b]-[MMS07], the crucial aspect for the conservation properties of the resulting time-stepping schemes is the approximation of these highly nonlinear time integrals. Of course, one potential option concerning the approximation is the application of a standard Gauss quadrature rule represented by

$$\bar{F}^{int}_{Ai} \approx \sum_{l=1}^{n_{gp_t}} \sum_{B=1}^{node} w_l \, \widetilde{M}_i(\zeta_l)\, q^h_B(\zeta_l) \left[\int_{\mathscr{B}_0} \nabla N_A \otimes \nabla N_B : S \, \mathrm{d}V \right] \Big|_{\zeta_l} \,, \tag{8.5}$$

using the Gauss points ζ_l and the Gauss weights w_l. The foregoing discretizations render a completely discrete system of equations, representing a time-stepping scheme with the following conservation properties. If we assume vanishing external loads, the resulting integration scheme allows the conservation of linear momentum as well as the conservation of angular momentum. Nevertheless, it can be shown that such a standard quadrature rule is not able to guarantee a conservation of the total energy for elastic deformations that is, however, an essential feature which has to be captured by the integrator regarding the claimed thermodynamical consistency. Consequently, we introduce the nonstandard quadrature rule

$$\bar{F}^{int}_{Ai} \approx \sum_{l=1}^{n_{gp_t}} \sum_{B=1}^{node} w_l \, \widetilde{M}_i(\zeta_l)\, q^h_B(\zeta_l) \left[\int_{\mathscr{B}_0} \nabla N_A \otimes \nabla N_B : S^{alg} \, \mathrm{d}V \right] \Big|_{\zeta_l} \,, \tag{8.6}$$

wherein the so-called *elastic-enhanced algorithmic* stress tensor $S^{alg} := S + {}^{el}S^{enh}$ has been applied based on the enhancement

$${}^{el}S^{enh}(S) = 2 \, \frac{\psi_{\alpha=1} - \psi_{\alpha=0} - \int_0^1 S : \frac{1}{2} \mathrm{D}_\alpha C^h \, \mathrm{d}\alpha}{\int_0^1 ||\mathrm{D}_\alpha C^h||^2 \, \mathrm{d}\alpha} \, \mathrm{D}_\alpha C^h \,. \tag{8.7}$$

This approach follows the enhanced Galerkin methods – or short: eG(k)-methods – that have been proposed originally by Gross *et al.* [GBS05] in the context of hyperelasticity.

[1] It is important to emphasize that the time shape functions $M_j \in \mathscr{P}^k$ are polynomials of degree k, whereas the reduced shape functions $\widetilde{M}_i \in \mathscr{P}^{k-1}$ are only of degree $k-1$.

Based on this specific nonstandard quadrature rule, the resulting time integrators offer additionally a conservation of the total energy $H_{\alpha=1} - H_{\alpha=0} = 0$ when the deformation is elastic. So that in combination with a strictly positive dissipation in the plastic case, a monotonic decrease of the total energy $H_{\alpha=1} - H_{\alpha=0} < 0$ is obtained and, consequently, a thermodynamically consistent time-integration is guaranteed, offering a superior performance in comparison to standard integration schemes. In this context, we want to point out once more that the key to thermodynamical consistency exclusively relies on a modified approximation of the corresponding time-integrals based on the elastic-enhanced algorithmic stress tensor.

8.4 Our contributions

In this Section, we present our two main contributions to comparative tensor visualization in the context of this particular mechanical engineering problem.

8.4.1 Interpreting the $2D$ symmetric difference tensor field as a $3D$ vector field

As already mentioned, we are basically interested in a way of representing the difference tensor field. Notice that the considered tensors are all symmetric so we basically have three independent components for each, i.e.

$$S = \begin{bmatrix} S_{11} & S_{12} \\ S_{12} & S_{22} \end{bmatrix} \tag{8.8}$$

which we could simply represent as a $3D$ vector $s = [S_{11}\ S_{22}\ S_{12}]^{\mathrm{t}}$, similar to the classical Voigt notation in the Finite Element context.

We have chosen this approach since we find it much more intuitive to compute the difference between two vectors than computing the difference between two tensors. We then connected the $3D$ vectors of each Gauss point, four by four, to create patches resulting in a quad-patch for every calculation element. Even if the resulting patches are indeed $3D$ we find it useful to simply visualize their $2D$ projection, as it shows the deformations that are, however, not related to the physical deformation of the considered body. Based on this visualization, we compensated the loss of one dimension by adding circles at each Gauss point whose radii are the Euclidean norm of the $3D$ difference vectors, namely

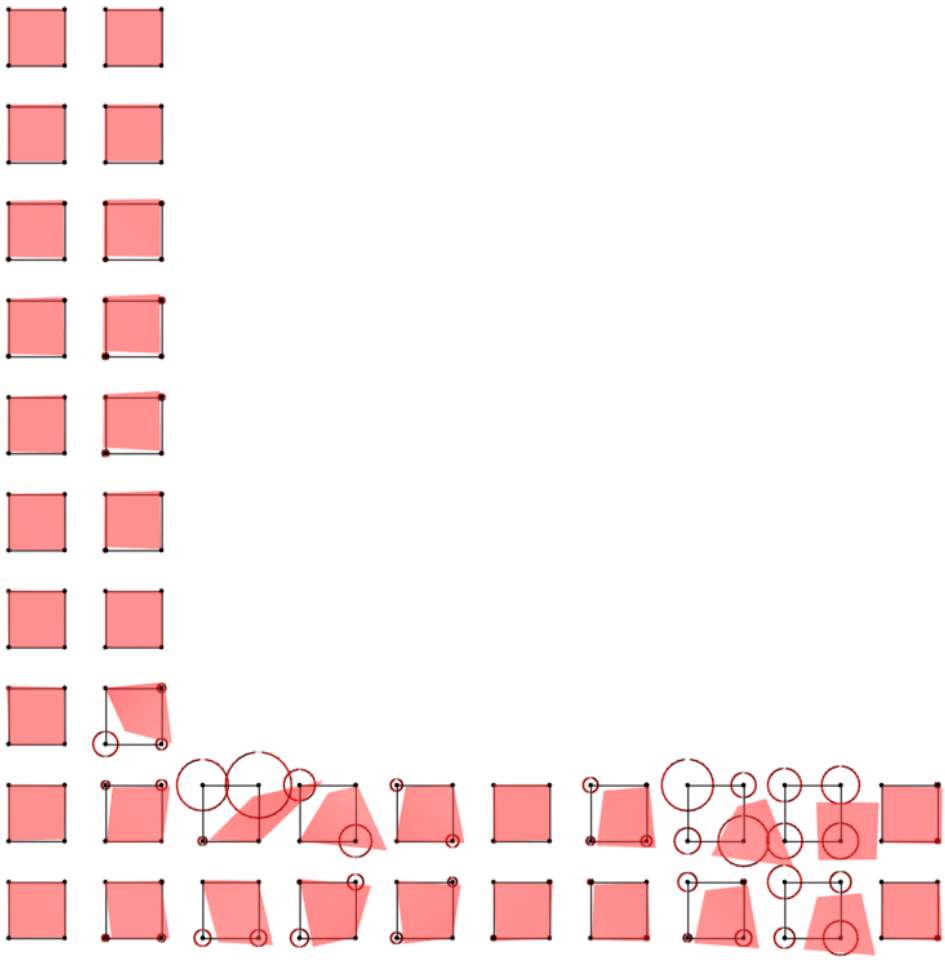


Figure 8.3: *Illustration of the* 3*D vector field scheme.*

$r = \sqrt{\Delta S_{11}{}^2 + \Delta S_{22}{}^2 + \Delta S_{12}{}^2}$. Figures 8.3, 8.4(a),(b) and 8.5(a),(b) illustrate those visualizations for stiff and non-stiff data sets. Note that we used a linear scaling to avoid occlusion. The results clearly demonstrate that the proposed approach is considerably well-suited to highlight regions of the body in which large corrections occur, compare Figure 8.4(a). Moreover, it is quite obvious that the needed correction are higher when large time-step sizes are involved, compare e.g. Figure 8.4(a) with Figure 8.4(b).

8.4.2 Visualizing the tensor's invariants through ellipsoids

The other proposed approach consists of visualizing the tensor's invariants as ellipsoids (see Fig. 8.6). Despite looking very similar to Kindlmann's tensor glyphs [Kin04], our

ellipsoids don't involve the tensor's eigenvectors at all here. Given the tensor S, its characteristic function is given by

$$\chi(S) = |S - \lambda I| = \lambda^2 - [S_{11} + S_{22}]\lambda + [S_{11}S_{22} - {S_{12}}^2] \tag{8.9}$$

and provides two invariants, namely the trace and the determinant of the tensor:

$$I_1 = \mathrm{tr}(S) = S_{11} + S_{22} \tag{8.10}$$

and

$$I_2 = \det(S) = S_{11}S_{22} - {S_{12}}^2 \tag{8.11}$$

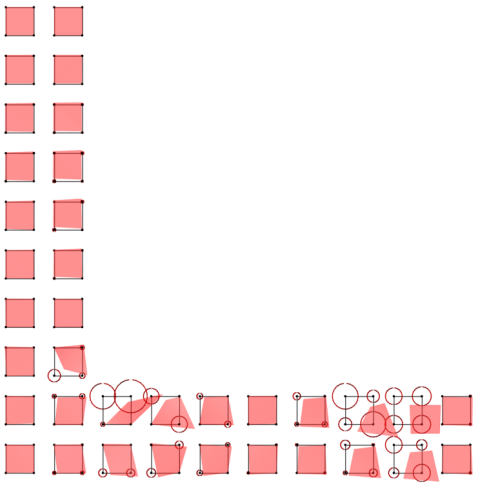
The ellipsoid is built using the components' basis

$$(x, y, z) = (\Delta I_1, \Delta I_2, \frac{\Delta I_1 + \Delta I_2}{2}), \tag{8.12}$$

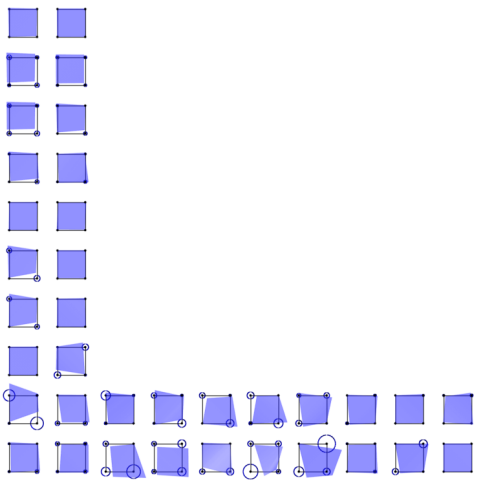
where ΔI_i is the difference between the invariants of both tensor fields.

Figures 8.4 (c)(d) and 8.5 (c)(d) respectively illustrate the ellipsoid-based visualizations for data sets that have been calculated by means of stiff and non-stiff material properties. Moreover, in Figure 8.7 we can see the evolution of the differences between the invariants over time. To investigate a potential correlation between the deformation and the corrections, the norm of the physical strain field based on the right Cauchy-Green strain tensor C has been additionally incorporated, whereby the following color-coding has been used: from blue to red for increasing strain norms. Note that we used a logarithmic scaling here, as opposed to a linear scaling, as the differences are much greater than in the circle-based visualization.

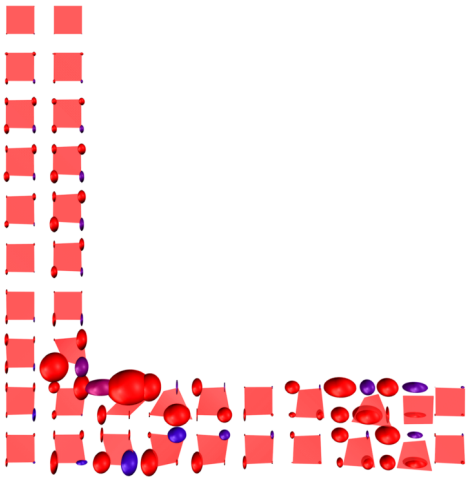
In comparison to the previous approach, the corresponding plots provide an essentially better view on the spatial distribution of the corrections, since the regions with extremely large corrections are not so dominant due to the mentioned logarithmic scaling. In this context, it becomes obvious that the locations of the corrections are, especially for the stiff data set shown in Figure 8.4 (c)(d), more homogeneously distributed when a smaller time-step size is applied. Also very interesting is the clustering of large corrections in certain regions of the 'L' particularly if the norm of the strains is high, as pictured in Figure 8.7.



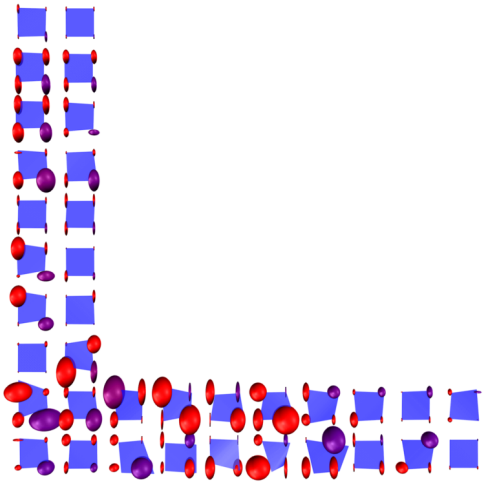
8.4(a) stiff data set, $h_n = 0.4$, time step 11



8.4(b) stiff data set, $h_n = 0.04$, time step 110

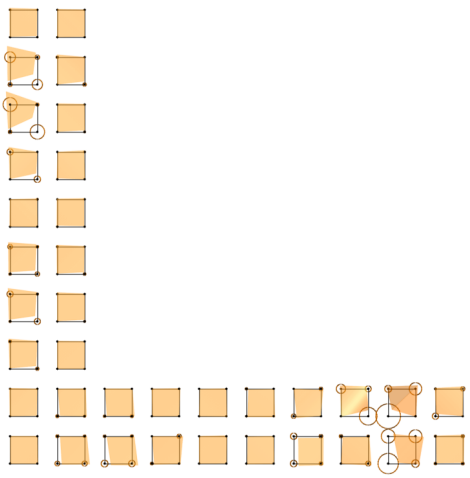


8.4(c) stiff data set, $h_n = 0.4$, time step 11

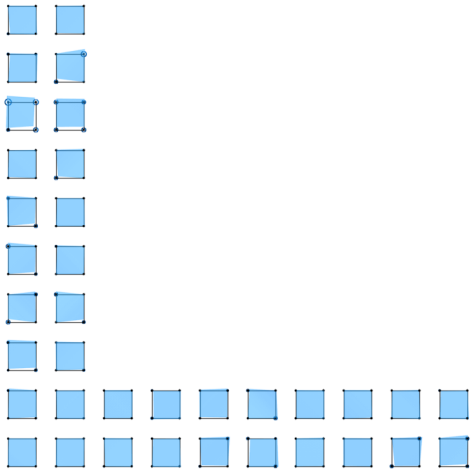


8.4(d) stiff data set, $h_n = 0.04$, time step 110

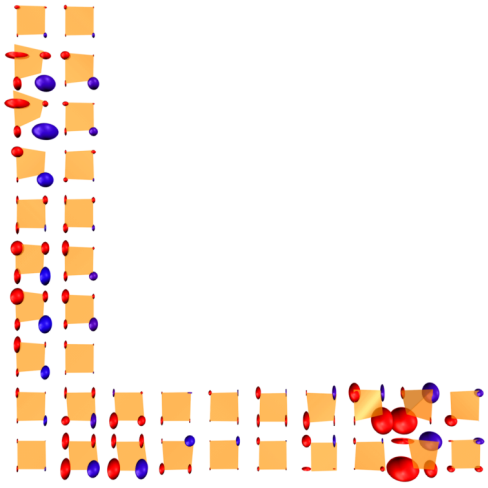
Figure 8.4: *Circle- and ellipsoid-based visualization using stiff material properties.*



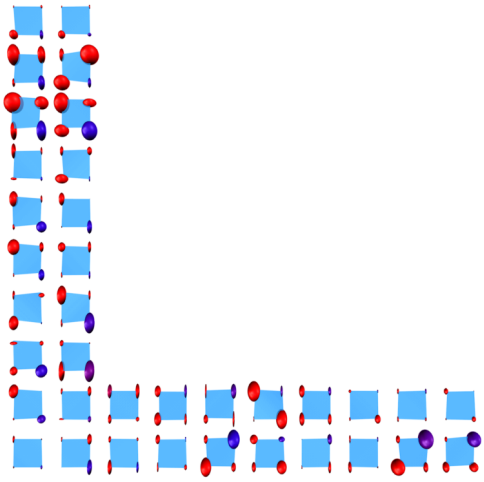
8.5(a) non-stiff data set, $h_n = 0.4$, time step 11



8.5(b) non-stiff data set, $h_n = 0.04$, time step 110



8.5(c) non-stiff data set, $h_n = 0.4$, time step 11



8.5(d) non-stiff data set, $h_n = 0.04$, time step 110

Figure 8.5: *Circle- and ellipsoid-based visualization using non-stiff material properties.*

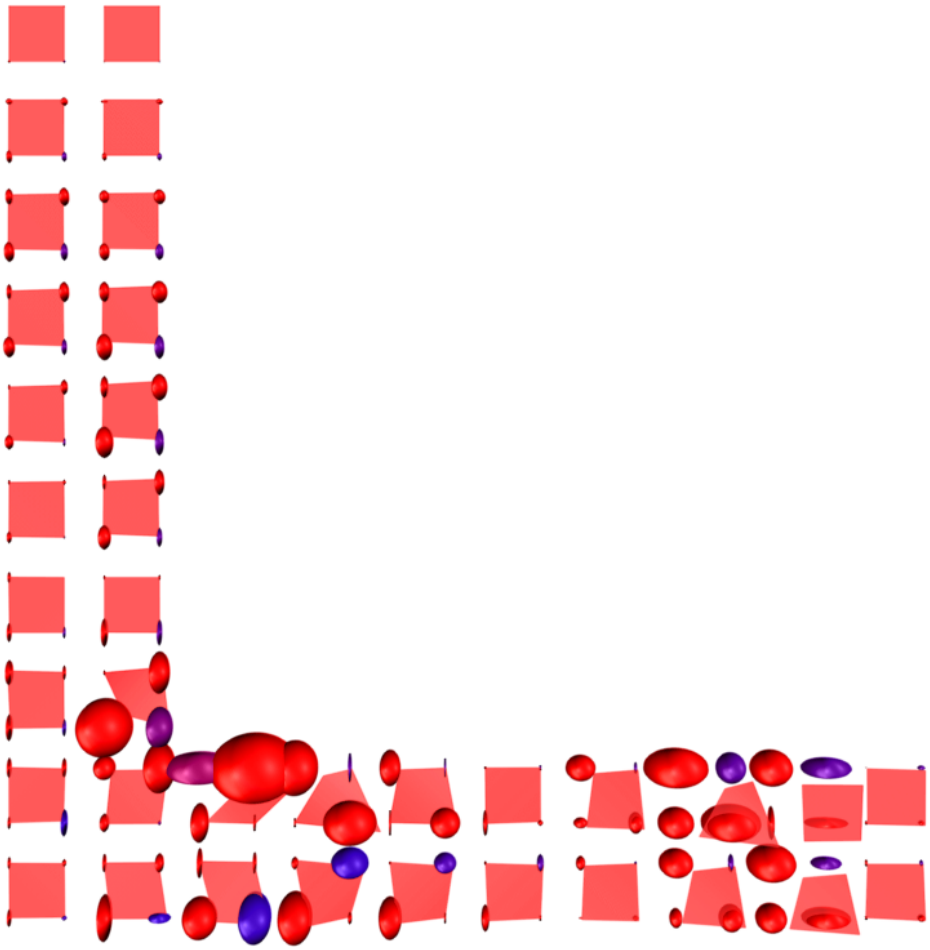


Figure 8.6: *Illustration of the invariant-based scheme.*

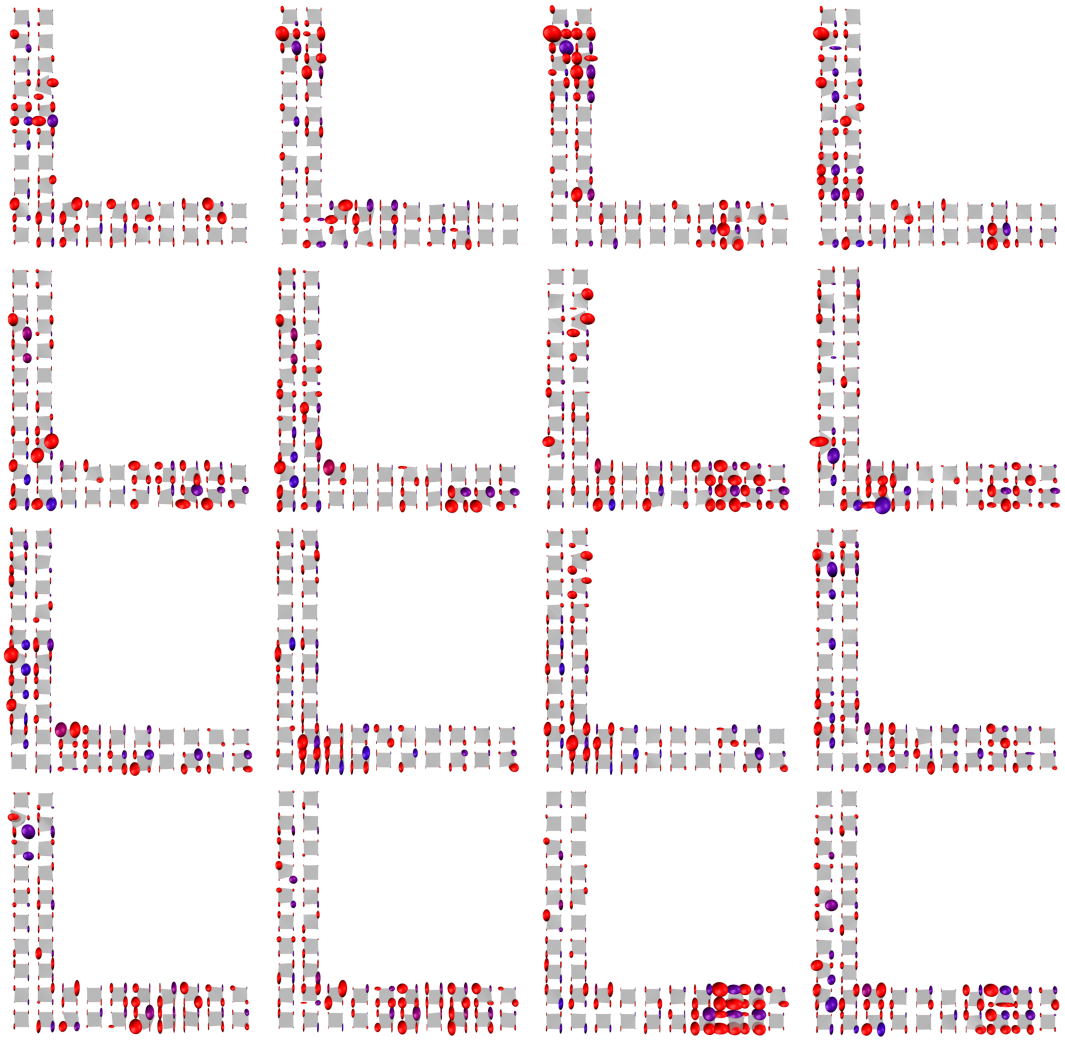


Figure 8.7: *Ellipsoid-based visualisation over time.* From top-left to bottom-right: time step 220 to 235.

Conclusion

In the first part of this study, we have presented the essential ingredients for a thermodynamically consistent time-stepping scheme for finite elasto-plasto-dynamics, whereby the conservation properties are directly related to the approximation of related time-integrals. In this context, a modified quadrature rule has been applied based on a so-called elastic-enhanced algorithmic stress tensor. In the second part, special emphasis has been placed on the investigation of the spatial distribution of the resulting difference between the stresses of the continuum model and the enhanced stresses for the time-stepping. Thereby, it has been shown in previous works that an 'ad hoc' visualization is not able to provide satisfying information. Therefore, we have devised visualizations of both abstract and physically based measures in the spatial context of the simulated domain. The results help revealing the intrinsic qualities of the data, especially by pointing out regions of interest. Indeed, our new visualization approaches, i.e. the vector interpretation and the invariant-based method provide a deeper insight in the numerical behavior of the algorithmic stress tensor and, consequently, they enable a better understanding of the discussed integration algorithms.

In future work, the discussed results, like influence of the time-step size or clustering of the corrections, should be verified for further data sets. Also, we would be interested by saliency-based visualization methods, which would emphasize - even more - regions of interest. Moreover, we plan to incorporate the time dimension, looking at the evolution of the corrections based on the here proposed visualization techniques. Thereby, especially the question of time continuity of the difference between both tensor fields seems to be essential.

Summary and Outlook

In Chapter 4 we have presented our new approach for computing the arrangement of arbitrary implicit planar curves [HB07]. Chapter 6 detailed our first contribution to rendering implicit functions robustly and efficiently [KHH*07]. In Chapter 7 we extended this approach using the latest graphics hardware and demonstrated a real-time ray tracer of implicit functions [KHK*08], our second contribution to this topic. Finally, in Part III Chapter 8 we have introduced two new visualization paradigms for comparative tensor visualization [MBH*07]; one reduces the tensor field to a vector field and the other one uses the tensor's invariants.

Future Work abounds. We would be interested in extending the arrangement of curves algorithm to higher dimensions, e.g. computing the arrangement of $3D$ curves or $3D$ surfaces. Regarding our two ray casting algorithms, we could implement beam casting and/or super-sampling to limit aliasing problems. Also of great importance is finding a reasonable trade-off between accuracy and speed for the inclusion algebra; as we mentioned, we still have some numerical issues with Reduced Affine Arithmetic and also need to implement transcendental functions with this form. We could also investigate the rendering of arbitrary parametric or free-form surfaces using the same approach. Considering the comparative tensor visualization paradigms, we would be interested by saliency-based visualization methods, which would highly emphasize regions of interest. Moreover, we could investigate the extension of those paradigms to $3D$ symmetric tensor fields by using a similar approach.

Bibliography

[AE98] AGARWAL P. K., ERICKSON J.: *Geometric Range Searching and Its Relatives, Advances in Discrete and Computational Geometry*. American Mathematical Society, Providence, 1998.

[Arm06] ARMERO F.: Energy-dissipative momentum-conserving time-stepping algorithms for finite strain multiplicative plasticity. *Computer Methods in Applied Mechanics and Engineering 195* (2006), 4862–4889.

[AS00] AGARWAL P. K., SHARIR M.: Arrangements and their applications. Handbook of Computational Geometry (J. Sack, ed.), pp. 49-119, 2000.

[BEW03] BOSE P., EVERETT H., WISMATH S.: Properties of arrangements graphs. *International Journal of Computational Geometry and Applications 13(6))* (2003), 447–462.

[BK90] BARNHILL R. E., KERSEY S. N.: A marching method for parametric surface/surface intersection. *Computer Aided Geometric Design 7* (1990), 257–280.

[Bli82] BLINN J.: A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics 1*, 3 (July 1982), 235–256.

[Blo94] BLOOMENTHAL J.: An implicit surface polygonizer. 324–349.

[BO79] BENTLEY J. L., OTTMANN T. A.: Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers* (1979).

[Bre92a] BREUEL T. M.: Fast recognition using adaptive subdivisions of transformation space. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (1992), pp. 445–451.

[Bre92b] BREUEL T. M.: *Geometric Aspects of Visual Object Recognition*. PhD thesis, Massachusetts Institute of Technology, 1992.

[Bre03a] BREUEL T. M.: Implementation techniques for geometric branch-and-bound matching methods. *Computer Vision and Image Understanding 90(3)* (2003), 258–294.

[Bre03b] BREUEL T. M.: On the use of interval arithmetic in geometric branch-and-bound algorithms. *Pattern Recognition Letters 24*, 9-10 (2003), 1375–1384.

[BS01] BETSCH P., STEINMANN P.: Conservation properties of a time fe method. part ii: Time-stepping schemes for nonlinear elastodynamics. *International Journal for Numerical Methods in Engineering 50* (2001), 1931–1955.

[Cas90] CASS T. A.: Feature matching for object localization in the presence of uncertainty. In *Proceedings of the International Conference on Computer Vision, Osaka, Japan* (1990).

[CdFC98] CARVALHO P. C., DE FIGUEIREDO L. H., CAVALCANTI P. R.: Computing arrangements of implicit curves. Extended abstract in Anais do VERMAC, pp. 19-22, 1998.

[Cha93] CHAZELLE B.: Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom. 9(2)* (1993), 145–158.

[CHMS00] CAPRIANI O., HVIDEGAARD L., MORTENSEN M., SCHNEIDER T.: Robust and efficient ray intersection of implicit surfaces. *Reliable Computing 6* (2000), 9–21.

[CMN02] CAPRANI O., MADSEN K., NIELSEN H. B.: Introduction to interval analysis, nov 2002.

[CS93] COMBA J. L. D., STOLFI J.: Affine arithmetic and its applications to computer graphics. In *Proc. VI Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'93)* (1993), pp. 9–18.

[dCJdFG99] DE CUSATIS JUNIOR A., DE FIGUEIREDO L., GATTAS M.: Interval methods for raycasting implicit surfaces with affine arithmetic. In *Proceedings of XII SIBGRAPI* (1999), pp. 1–7.

[dF96] DE FIGUEIREDO L. H.: Surface intersection using affine arithmetic. In *Graphics Interface* (May 1996), pp. 168–175.

[dTLP07] DE TOLEDO R., LEVY B., PAUL J.-C.: Iterative methods for visualization of implicit surfaces on gpu. In *ISVC, International Symposium on Visual Computing* (Lake Tahoe, Nevada/California, November 2007), Lecture Notes in Computer Science, SBC - Sociedade Brasileira de Computacao, Springer.

[Duf92] DUFF T.: Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), ACM Press, pp. 131–138.

[Ede87] EDELSBRUNNER H.: *Algorithms in Combinatorial Geometry*. Springer-Verlag, Heidelberg, 1987.

[EG89] EDELSBRUNNER H., GUIBAS L. J.: Topologically sweeping an arrangement. *J. Comput. Syst. Sci. 38* (1989), 165–194.

[EKSW04] EIGENWILLIG A., KETTNER L., SCHOEMER E., WOLPERT N.: Complete, exact, and efficient computations with cubic curves. In *20th Annual ACM Symposium on Computational Geometry* (2004), pp. 409–418.

[ESW02] EIGENWILLIG A., SCHOEMER E., WOLPERT N.: Sweeping arrangements of cubic segments exactly and efficiently. Technical Report ECG-TR-182202-01, 2002.

[FSSV06] FLOREZ J., SBERT M., SAINZ M. A., VEHI J.: Improving the interval ray tracing of implicit surfaces. In *Lecture Notes in Computer Science* (2006), vol. 4035, pp. 655–664.

[Gav05] GAVRILIU M.: *Towards More Efficient Interval Analysis: Corner Forms and a Remainder Interval Newton Method*. PhD thesis, California Institute of Technology, July 2005.

[GBS05] GROSS M., BETSCH P., STEINMANN P.: Conservation properties of a time fe method. part iv: Higher order energy and momentum conserving schemes. *International Journal for Numerical Methods in Engineering 63* (2005), 1849–1897.

[GCH00] GAY O., COEURJOLLY D., HURST N. J.: Libaffa - c++ affine arithmetic library for gnu/linux, 2000.

[GLS88] GROETSCHEL M., LOVASZ L., SCHRIJVER A.: *Geometric algorithms and combinatorial optimization*. Springer-Verlag, 1988.

[GM07] GAMITO M. N., MADDOCK S. C.: Ray casting implicit fractal surfaces with reduced affine arithmetic. *Vis. Comput. 23*, 3 (2007), 155–165.

[Gon00] GONZALEZ O.: Exact energy and momentum conserving algorithms for general models in nonlinear elasticity. *Computer Methods in Applied Mechanics and Engineering 190* (2000), 1763–1783.

[Gri85] GRIFFITHS P. A.: *Introduction to Algebraic Curves*. Kuniko Weltin, trans., American Mathematical Society, Translation of Mathematical Monographs volume 70, 1985.

[Hal97] HALPERIN D.: Arrangements. Jacob E. Goodman and Joseph O'Rourke, editors, Handbook of Discrete and Computational Geometry, chapter 21, pp. 389-412, 1997.

[Han75] HANSEN E. R.: A generalized interval arithmetic. In *Proceedings of the International Symposium on Interval Mathemantics* (London, UK, 1975), Springer-Verlag, pp. 7–18.

[Han83] HANRAHAN P.: Ray tracing algebraic surfaces. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1983), ACM Press, pp. 83–90.

[Har96] HART J. C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10 (1996), 527–545.

[HB07] HIJAZI Y. O., BREUEL T. M.: Computing arrangements using subdivision and interval arithmetic. In *P. Chenin, T. Lyche and L.L. Schumaker (Eds.), Proceedings of the Sixth International Conference on Curves and Surfaces, June 29-July 5, 2006, Avignon, France, Curve and Surface Design: Avignon 2006, Nashboro Press* (2007), pp. 173–182.

[HHHJ07] HIJAZI Y. O., HAGEN H., HANSEN C., JOY K. I.: Why interval arithmetic is so useful. Second Annual Workshop of DFG's International Research Training Group 1131 (to be published), September 2007.

[Hij06] HIJAZI Y. O.: Arrangements of planar curves. In *H. Hagen, A. Kerren, P. Dannenmann (Eds.), Visualization of Large and Unstructured Data Sets, Proceedings of the first workshop of DFG's International Research Training Group "Visualization of Large and Unstructured Data Sets - Applications in Geospatial Planning, Modeling, and Engineering", June 14-16, 2006, Dagstuhl, Germany, GI-Edition, Lecture Notes in Informatics, Seminars Series* (2006), vol. S-4, pp. 59–68.

[HS98] HEIDRICH W., SEIDEL H.-P.: Ray-tracing procedural displacement shaders. In *Graphics Interface* (1998), pp. 8–16.

[HSS*05] HADWIGER M., SIGG* C., SCHARSACH H., BUHLER K., GROSS* M.: Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum 24*, 3 (2005), 303–312.

[Jac99] JACOBSON R.: *Information Design*. The MIT Press, 1999.

[k3d] K3DSURF: The k3dsurf project. http://k3dsurf.sourceforge.net/.

[KB89] KALRA D., BARR A. H.: Guaranteed ray intersections with implicit surfaces. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1989), ACM Press, pp. 297–306.

[KHH*07] KNOLL A., HIJAZI Y. O., HANSEN C., WALD I., HAGEN H.: Interactive ray tracing of arbitrary implicits with simd interval arithmetic. In *Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 11–18.

[KHK*08] KNOLL A., HIJAZI Y. O., KENSLER A., SCHOTT M., HANSEN C., HAGEN H.: *Fast and Robust Ray Tracing of General Implicits on the GPU*. Tech. rep., University of Utah Technical Report UUSCI-2007-014 (conditionally accepted for publication in *Computer Graphics Forum*), 2008.

[KHW07] KNOLL A., HANSEN C., WALD I.: *Coherent Multiresolution Isosurface Ray Tracing*. Tech. Rep. UUSCI-2007-001, SCI Institute, University of Utah, 2007. (submitted for publication).

[Kin04] KINDLMANN G.: *Visualization and Analysis of Diffusion Tensor Fields*. PhD thesis, University of Utah, 2004.

[LB06] LOOP C., BLINN J.: Real-time GPU rendering of piecewise algebraic surfaces. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), ACM Press, pp. 664–670.

[LC87] LORENSEN W. E., CLINE H. E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (Proceedings of ACM SIGGRAPH) 21*, 4 (1987), 163–169.

[LMP06] LIANG C., MOURRAIN B., PAVONE J. P.: Subdivision methods for 2d and 3d implicit curves. to appear in Computational Methods for Algebraic Spline Surfaces, Springer-Verlag, 2006.

[LOdF01] LOPES H., OLIVEIRA J., DE FIGUEIREDO L. H.: Robust adaptive approximation of implicit curves. In *SIBGRAPI '01: Proceedings of the XIV Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'01)* (Washington, DC, USA, 2001), IEEE Computer Society.

[LTWVG*06] LERCH M., TISCHLER G., WOLFF VON GUDENBERG J., HOFSCHUSTER W., KRÄMER W.: Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw. 32*, 2 (2006), 299–324.

[MBH*07] MOHR R., BOBACH T., HIJAZI Y. O., REIS G., STEINMANN P., HAGEN H.: Comparative tensor visualisation within the framework of consistent time-stepping schemes. Second Annual Workshop of DFG's International Research Training Group 1131 (to be published), September 2007.

[Mes02] MESSINE F.: Extentions of affine arithmetic: Application to unconstrained global optimization. *Journal of Universal Computer Science 8*, 11 (2002), 992–1015.

[MFK*04] MARMITT G., FRIEDRICH H., KLEER A., WALD I., SLUSALLEK P.: Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)* (2004), pp. 429–435.

[MGW05] MEYER M. D., GEORGEL P., WHITAKER R. T.: Robust particle systems for curvature dependent sampling of implicit surfaces. In *SMI '05: Proceedings of the International Conference on Shape Modeling and Applications 2005 (SMI' 05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 124–133.

[Mit90] MITCHELL D. P.: Robust ray intersection with interval arithmetic. In *Proceedings on Graphics Interface 1990* (1990), pp. 68–74.

[Mit91] MITCHELL D. P.: Three applications of interval analysis in computer graphics. In Frontiers in Rendering course notes, pages 14-1–14-13. SIGGRAPH'91, July 1991.

[ML02] MENG X., LAURSEN T.: Energy consistent algorithms for dynamic finite deformation plasticity. *Computer Methods in Applied Mechanics and Engineering 191* (2002), 1639–1675.

[MMS06a] MOHR R., MENZEL A., STEINMANN P.: Consistent galerkin–based time–stepping schemes for geometrically nonlinear elasto–plastodynamics, 2006.

[MMS06b] MOHR R., MENZEL A., STEINMANN P.: Galerkin-based time integrators for geometrically nonlinear elasto-plastodynamics – challenges in modeling and visualization. In *Visualization of Large and Unstructured Data Sets, GI-Edition Lecture Notes in Informatics (LNI), S-4* (2006), pp. 185–194.

[MMS07] MOHR R., MENZEL A., STEINMANN P.: Conservation properties of galerkin-based time-stepping schemes for finite elasto-plasto-dynamics. In *COMPLAS IX – International Conference on Computational Plasticity, Barcelona* (2007).

[MN00] MEHLHORN K., NAEHER S.: *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[Moo66] MOORE R. E.: *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[MPT*07] MOURRAIN B., PAVONE J.-P., TREBUCHET P., TSIGARIDAS E., WINTZ J.: Synaps: a library for symbolic-numeric computation. IMA volume on software for algebraic geometry. Springer-Verlag. To be published., 2007.

[MS06] MILENKOVIC V., SACKS E.: An approximate arrangement algorithm for semi-algebraic curves. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry* (2006), pp. 237–246.

[Neu03] NEUMAIER A.: Taylor forms-use and limits. *Reliable Computing 9* (February 2003), 43–79(37).

[New59] NEWMARK N.: A method of computation for structural dynamics. *ASCE Journal of the Engineering Mechanics Division 85* (1959), 67–94.

[NK06] NELSON B., KIRBY R. M.: Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering. *IEEE Transactions on Visualization and Computer Graphics 12*, 1 (2006), 114–125.

[NSP06] NOELS L., STAINIER L., PONTHOT J.: An energy momentum conserving algorithm using the variational formulation of visco-plastic updates. *International Journal for Numerical Methods in Engineering 65* (2006), 904–942.

[OBA*03] OHTAKE Y., BELYAEV A., ALEXA M., TURK G., SEIDEL H.-P.: Multi-level partition of unity implicits. *ACM Trans. Graph. 22*, 3 (2003), 463–470.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 21*, 3 (2002), 703–712.

[Per85] PERLIN K.: An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1985), ACM Press, pp. 287–296.

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum 26*, 3 (sep 2007). (Proceedings of Eurographics), to appear.

[PLLdF06] PAIVA A., LOPES H., LEWINER T., DE FIGUEIREDO L. H.: Robust adaptive meshes for implicit surfaces. In *19th Brazilian Symposium on Computer Graphics and Image Processing* (2006), pp. 205–212.

[PV04] PLANTINGA S., VEGTER G.: Isotopic approximation of implicit curves and surfaces. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (2004), pp. 245–254.

[Res] RESEARCH W.: Mathworld. http://mathworld.wolfram.com.

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: A multiresolution point rendering system for large meshes. In *Proc. of ACM SIGGRAPH* (2000), pp. 343–352.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH).

[RVdF06] ROMEIRO F., VELHO L., DE FIGUEIREDO L. H.: Hardware-assisted Rendering of CSG Models. In *SIBGRAPI* (2006), pp. 139–146.

[SC95] STOLTE N., CAUBET R.: Fast high definition discrete ray tracing implicit surfaces. In *5th DGCI - Discrete Geometry for Computer Imagery, pages 61-70, Clermont-Ferrand, Universite d'Auvergne* (1995).

[SECG03] SANJUAN-ESTRADA J. F., CASADO L. G., GARCIA I.: Reliable algorithms for ray intersection in computer graphics based on interval arithmetic. In *XVI Brazilian Symposium on Computer Graphics and Image Processing, SIBGRAPI 2003* (2003), pp. 35–42.

[SF92] SEDERBERG T. W., FAROUKI R. T.: Approximation by interval bézier curves. *Computer Graphics and Applications 12* (1992), 87–95.

[SSS06] SCHREINER J., SCHEIDEGGER C., SILVA C.: High-Quality Extraction of Isosurfaces from Regular and Irregular Grids. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1205–1212. Proceedings of IEEE Visualization 2006.

[ST92] SIMO J., TARNOW N.: The discrete energy-momentum method. conserving algorithms for nonlinear elastodynamics. *Zeitschrift fuer Angewandte Mathematik und Physik (ZAMP) 43* (1992), 757–792.

[Tot85] TOTH D. L.: On ray tracing parametric surfaces. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1985), ACM Press, pp. 171–179.

[VKZM06] VARADHAN G., KRISHNAN S., ZHANG L., MANOCHA D.: Reliable implicit surface polygonization using visibility mapping. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing* (Aire-la-Ville, Switzerland, Switzerland, 2006), Eurographics Association, pp. 211–221.

[vW85] VAN WIJK J. J.: Ray tracing objects defined by sweeping a sphere. *Computers & Graphics 9* (1985), 283–290.

[Wal97] WALSTER G. W.: Introduction to interval arithmetic, 1997.

[WH94] WITKIN A. P., HECKBERT P. S.: Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), ACM Press, pp. 269–277.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics* (2006), 485–493. (Proceedings of ACM SIGGRAPH).

[WM06] WINTZ J., MOURRAIN B.: Subdivision method for computing an arrangement of implicit planar curves. In *Proceedings of the Algebraic Geometry and Geometric Modeling 2006* (2006).

[WMW86] WYVILL G., MCPHEETERS C., WYVILL B.: Data structure for soft objects. *The Visual Computer 2* (1986), 227–234.

[Wol02] WOLPERT N.: *An Exact and Efficient Approach for Computing a Cell in an Arrangement of Quadrics*. PhD thesis, Saarland University, Saarbrucken, Germany, 2002.

[Wol03] WOLPERT N.: Jacobi curves: Computing the exact topology of arrangements of non-singular algebraic curves. In *11th Europoean Symposium on Algorithms (ESA), Budapest* (2003), pp. 532–543.

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics).

[Yap06] YAP C. K.: Complete subdivision algorithms, i: intersection of bézier curves. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry* (2006), pp. 217–226.

List of Figures

List of Tables

Appendix A

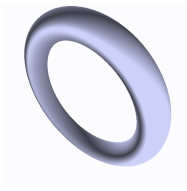
Traversal pseudo-code

Algorithm 6 Ray-Implicit Traversal.

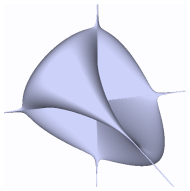
```
template<int K, int U, int V, int DK>
void traverse(RayPacket r, Box domain, Implicit implicit, int d_stop) {
  (get t_enter, t_exit, t_kenter, t_kexit)
  simd validmask = intersectBB(r, domain);
  //validmask indicates rays that are active
  float full_tk = tk_exit - tk_enter;
  float full_u = mul4(r.dir[U], full_tk);
  float full_v = mul4(r.dir[V], full_tk);
  struct Stack {
    simd t_incr;
    simd u_incr, v_incr;
    float k_incr;
    char side;
  };
  Stack stk[maxDepth];
  for(int d=0;d<maxDepth;d++){
    float width = 1.f / (float)(1<<d);
    stk[d].t_incr = mul4(full_tk, width);
    stk[d].u_incr = mul4(full_u, width);
    stk[d].v_incr = mul4(full_v, width);
    stk[d].side = -1;
  }
  int depth = 0;
  float curr_k = DK==+1 ? domain.min[K]:domain.max[k];
  simd curr_t, curr_u, curr_v;
  curr_t = t_kenter;
  curr_u = add4(r.org[U],mul4(r.dir[U],curr_t));
  curr_v = add4(r.org[V],mul4(r.dir[V],curr_t));
  simd next_t, next_u, next_v;
  for(;;) {
    stk[depth].side++;
    next_k = DK==+1 ?
             curr_k + stk[depth].k_increment :
             curr_k - stk[depth].k_increment;
    next_u = add4(curr_u, stk[depth].u_increment);
    next_v = add4(curr_v, stk[depth].v_increment);
    next_t = add4(curr_t, stk[depth].t_increment);
    hitmask = and4(validmask, cmp_ge4(next_t, tenter));
    if (any4(simd_hitmask)) {
      interval4 ibox;
      (fill ibox with curr and next k,u,v)
      interval4 F = implicit.evalute_interval4(ibox);
      if (any4(F.contains(0))) {
        if (!all4(cmp_ge4(sub4(F.hi,F.lo),INFINITY))){
          if (depth == maxDepth-1){
            //hit
            hit(r, curr_t);
            (compute normal);
            if (all4(r.hitmask))
              return;
          } else {
            //recurse
            depth++;
            continue;
          }
        }
      }
    }
    validmask = and4(validmask, cmp_le4(next_t, texit));
    if (none4(validmask))
      return;
    curr_k = next_k;
    curr_t = next_t;
    curr_u = next_u;
    curr_v = next_v;
    if (stk[depth].side & 1)
    {
      do{
        if (depth-- == -1)
          return; }
      while(stk[depth] & 1);
      continue;
    }
  }
}
```

Appendix B

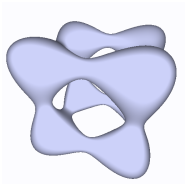
Selected implicits on the CPU



Torus	**Sphere**	**Parabolic Cylinder**	**Hyperboloid**
$(r_i - \sqrt{x^2+y^2})^2 + z^2 - r_o^2$	$x^2+y^2+z^2-r^2$	$x^2 - y - r^2$	$-\frac{x^2}{a^2} - \frac{y^2}{a^2} + \frac{z^2}{c^2} - 1$
22.9 fps	20.5 fps	34.1 fps	19.2 fps

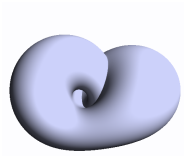


Steiner 1	**Steiner 2**	**Mitchell**
$x^2y^2 + y^2z^2 + x^2z^2 + xyz$	$(x^2y^2 + y^2z^2 + x^2z^2)^2 + xyz$	$4(x^4 + (y^2+z^2)^2) + 17x^2(y^2+z^2)$ $-20(x^2+y^2+z^2) + 17$
6.1 fps	17.2 fps	5.9 fps

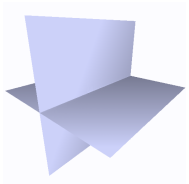


Tangle	**Blobby**	**Absolute value**	**Inverse function**
$x^4 - 5x^2 + y^4 - 5y^2 + z^4 - 5z^2 + 11.8$	$\Sigma_{i=1}^{N} \frac{r_i^2}{\|x-p_i\|^2} - 1$	$\|x\| + \|y\| - z$	$\frac{1}{x-y^2} - z$
3.8 fps	4.7 fps	30.8 fps	20.8 fps

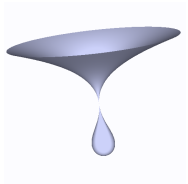
Figure B.1: *Selected implicits on the CPU*, covering a wide range of different shapes and topologies. All examples are rendered at $d_{stop} = 10$ at 512^2 frame buffer resolution, on an Intel Core Duo 2.16 GHz. Performance is largely dependent on the number of operations required to evaluate the implicit, the entailed cost of computing the associated IA expressions, and the spatial complexity (effectively, implicit surface area) of the scene.



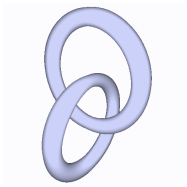
Klein Bottle
$(x^2+y^2+z^2+2y-1)((x^2+y^2+z^2-2y-1)^2-8z^2)$ $+16xz(x^2+y^2+z^2-2y-1)$
6.0 fps



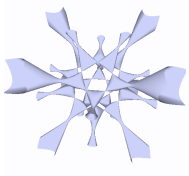
Intersecting planes
xy
33.0 fps



Teardrop
$0.5x^5+0.5x^4-y^2-z^2$
15.3 fps



Linked tori
$g(10x,10y-2,10z,13)g(10z,10y+2,10x,13)+1000$ $g(x,y,z,c)=(x^2+y^2+z^2+c)^2-53(x^2+y^2)$
4.0 fps



Barth-sextic
$4(\tau^2x^2-y^2)(\tau^2y^2-z^2)(\tau^2z^2-x^2)$ $-(1+2\tau)(x^2+y^2+z^2-1)^2$
4.9 fps

Figure B.2: *Selected implicits on the CPU*, ray traced with $d_{stop}=10$ at 512^2 frame buffer resolution, on an Intel Core Duo 2.16 GHz. Barth-sextic was rendered using $\tau=\frac{1+\sqrt{5}}{2}$.

Appendix C

Selected implicits on the GPU

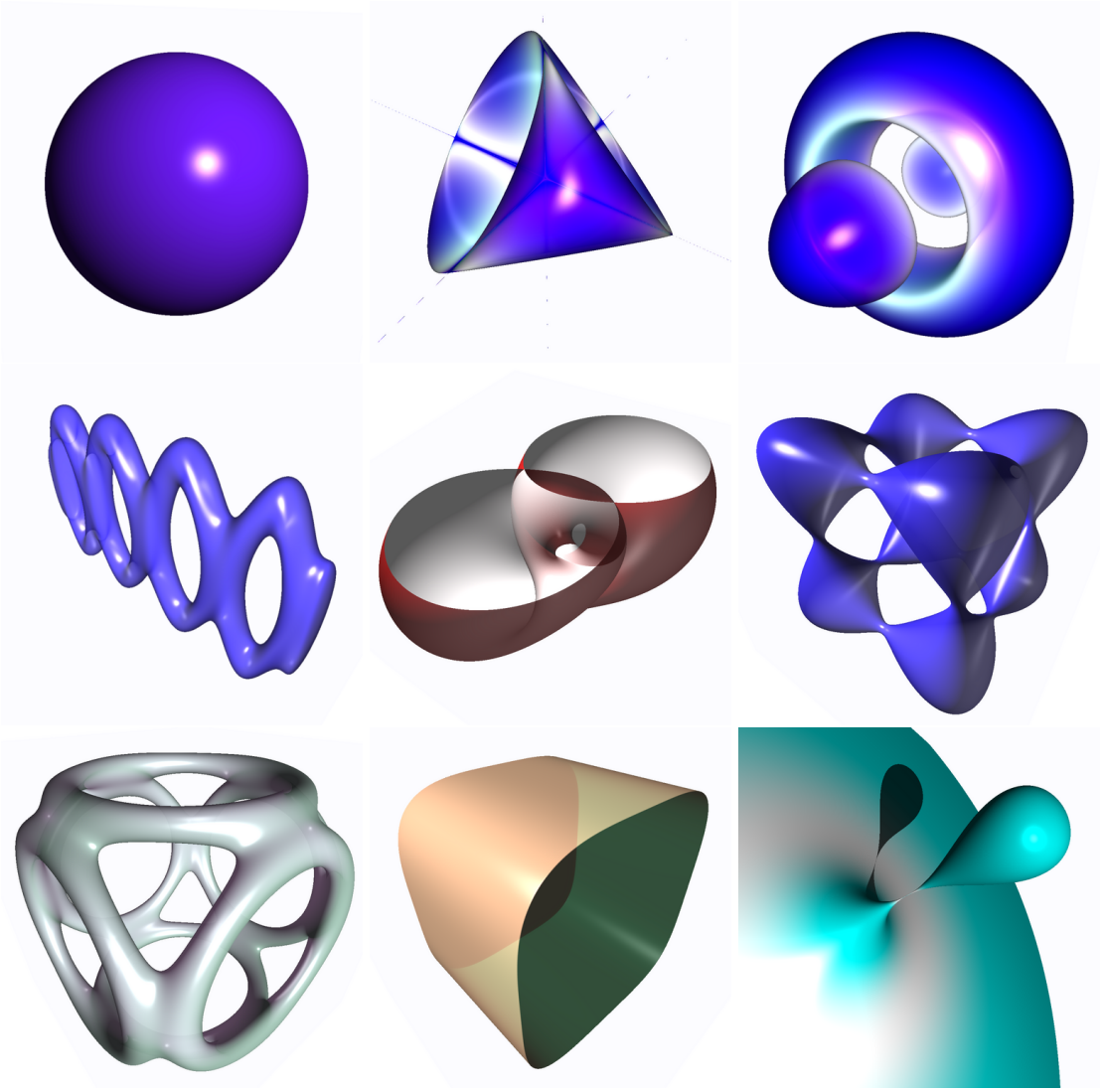


Figure C.1: *Selected implicits on the GPU*, rendered on a 4-core Xeon 2.33GHz with an NVIDIA 8800 GTX. From top-left to bottom-right: a sphere, the Steiner, the Mitchell, a 4-bretzel, the Klein bottle (cut out), the tangle, a decocube, a super-quadric and a tear drop.

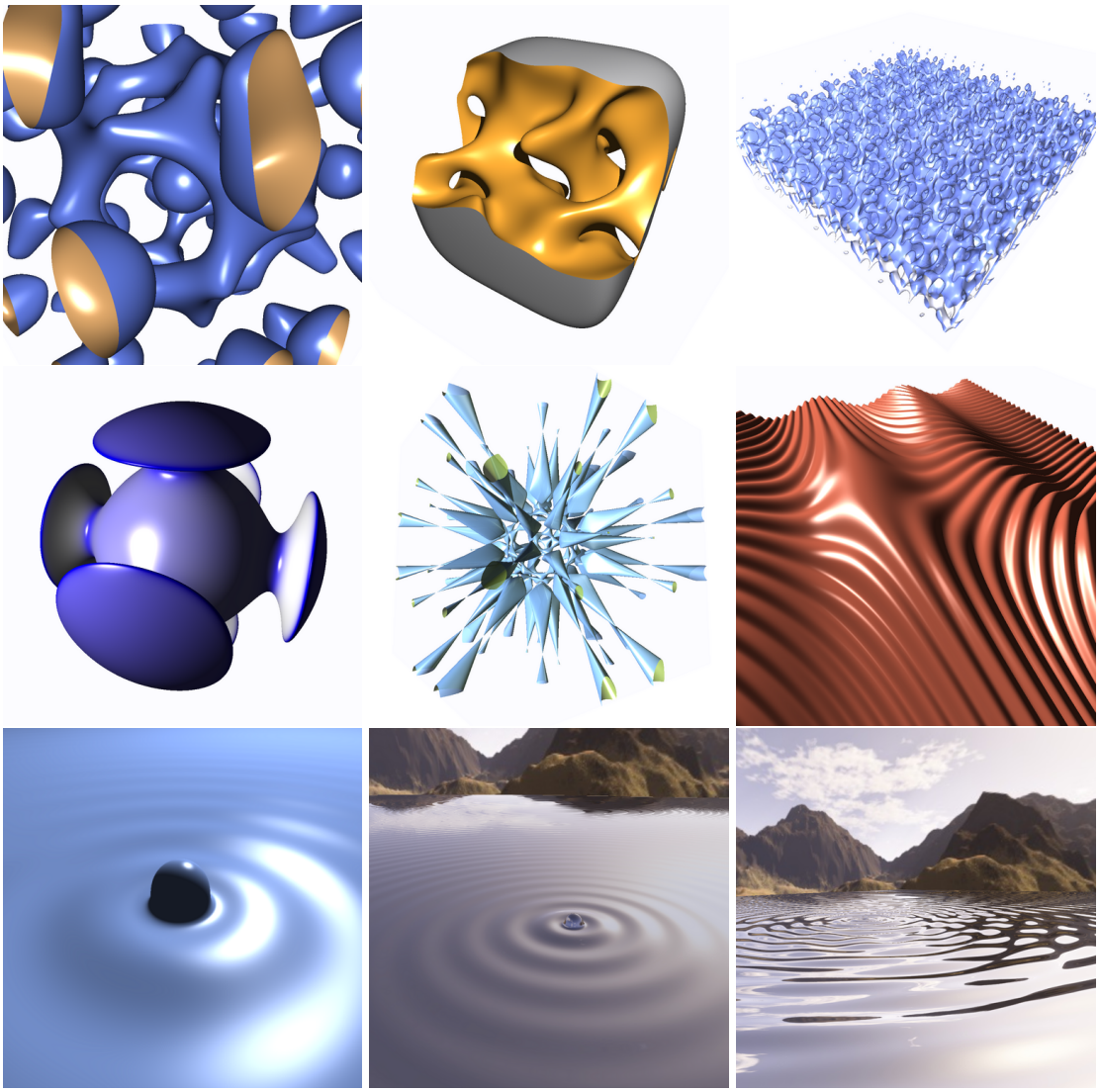


Figure C.2: *Selected implicits on the GPU*, rendered on a 4-core Xeon 2.33GHz with an NVIDIA 8800 GTX. From top-left to bottom-right: a icosahedron, a stone cube, a stone surface, a decocube to sphere morphing, a Barth-decic to sphere morphing, cloth modeling, a water drop, a "realistic" water drop and multiple water drops.

Curriculum Vitae

Curriculum Vitae

Personal data

Last name: **HIJAZI**

First name: **Younis**

Birthday: 20.09.1981

Place of birth: Paris (13ième)

Nationality: French

Education

Since 2005: **PhD in computer science**, University of Kaiserslautern.

2003 - 2004: **Master in Computer Science**, University Paris Sud, Orsay.

2001 - 2003: **Bachelor and "Maîtrise" in Mathematics**, UHP Nancy I, Nancy.

2000 - 2001: **French engineering school**: École des Mines de Nantes, Nantes.

1999 - 2000: **University-level preperation for the competitive entrance to French engineering schools**, Lycée La Malgrange, Nancy.

1996 - 1999: **High School**, Lycée Nicolas Appert, Nantes & Lycée Henri Poincaré, Nancy.

Degrees

2004: **Master in Computer Science (DEA I3)**, "mention *Bien*", University Paris 11, Orsay. Thesis: $4D$ *Animation*, supervised by Prof. Dr. Dominique Bechmann, University Louis Pasteur, Strasbourg.

2003: **"Maîtrise" in Mathematics**, "mention *Assez Bien*", UHP Nancy I, Nancy.

2002: **Bachelor in Mathematics**, UHP Nancy I, Nancy.

1999: **Baccalaureate S**, "mention *Assez Bien*", Lycée Henri Poincaré, Nancy.