CONVOLUTION SURFACES IN COMPUTER GRAPHICS

Thesis by
Andrei Sherstyuk

In Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

School Of Computer Science and Software Engineering
Monash University
Australia

Submitted January 17, 1999

©Andrei Sherstyuk 1998

Contents

Title Page i

Table of Contents ii

List of Figures v

List of Tables vii

Summary viii

Statement ix

Acknowledgments x

1 Introduction 1
- 1.1 A brief history of implicit surfaces and a literature review . . . 1
- 1.2 Generalization of classic implicit models . . . 2
- 1.3 Open questions . . . 3
  - 1.3.1 Formulation . . . 3
  - 1.3.2 Modeling . . . 4
  - 1.3.3 Rendering . . . 4
- 1.4 Contributions . . . 5

2 Formulation of Convolution Surfaces 7
- 2.1 Introduction . . . 7
- 2.2 Definitions . . . 8
- 2.3 Kernels and primitives . . . 9
  - 2.3.1 A new kernel function . . . 9
  - 2.3.2 The seven kernels . . . 10
  - 2.3.3 Primitives/kernels compatibility chart . . . 11
  - 2.3.4 Computational costs . . . 11
  - 2.3.5 Problems with polynomial kernels . . . 13
  - 2.3.6 A short summary . . . 14
- 2.4 Development of implicit primitives . . . 14
  - 2.4.1 Point sources . . . 15
  - 2.4.2 Line segments . . . 15
  - 2.4.3 Arcs . . . 15
  - 2.4.4 Triangles . . . 16
  - 2.4.5 Planes . . . 17
- 2.5 Examples . . . 17
- 2.6 Performance . . . 18
- 2.7 Conclusions . . . 19

3 Modeling with Convolution Surfaces 21
- 3.1 Introduction . . . 21
- 3.2 Definitions . . . 21
  - 3.2.1 Implicit surfaces . . . 21
  - 3.2.2 Convolution surfaces . . . 21
  - 3.2.3 Implicit primitives . . . 22
  - 3.2.4 Skeletons . . . 22

3.3 The design system . . . 22
3.3.1 New modeling primitives as skeletal elements . . . 22
3.3.2 Local properties of implicit primitives . . . 23
3.3.3 Materials and elements . . . 24
3.3.4 Profiling as a means for achieving variety . . . 25
3.3.5 Offset surfaces as visual aid and the data structures . . . 26
3.3.6 Variables . . . 26
3.4 The main modeling loop . . . 26
3.4.1 Datasets . . . 26
3.4.2 The main modeling strategy . . . 26
3.5 Practical examples of implicit design . . . 27
3.5.1 Global skeleton manipulations . . . 27
3.5.2 Local modeling techniques . . . 30
3.5.3 Volumetric detail . . . 30
3.6 Implementation details and timing results . . . 32
3.7 Conclusion . . . 33

4 Rendering Convolution Surfaces **35**
4.1 Introduction . . . 35
4.1.1 The problem . . . 35
4.1.2 Previous work . . . 35
4.1.3 Algorithm preconditions . . . 36
4.1.4 Algorithm postconditions . . . 36
4.2 The algorithm . . . 37
4.2.1 Ray-tracing algorithm for metaballs . . . 37
4.2.2 Generalization of the algorithm for metaballs . . . 37
4.2.3 An example . . . 38
4.2.4 Bounding volumes . . . 39
4.2.5 Shading and texturing . . . 39
4.3 Error analysis . . . 39
4.4 Optimizations . . . 40
4.4.1 Polynomization on demand . . . 40
4.4.2 Fast ray/surface rejection test . . . 40
4.4.3 Volatile and permanent clusters . . . 41
4.4.4 Reusing interpolants . . . 41
4.5 Examples . . . 41
4.6 Conclusions . . . 43

5 Epilogue **47**

A Field functions for point primitives **49**
A.1 Cauchy function . . . 49
A.2 Gaussian function . . . 49
A.3 Inverse function . . . 50
A.4 Inverse squared function . . . 50
A.5 Metaballs . . . 50
A.6 Soft objects . . . 50
A.7 W-quartic polynomial . . . 50

B Field functions for line primitives **51**
B.1 Cauchy line segment . . . 51
B.2 Gaussian line segment . . . 51
B.3 Inverse potential line segment . . . 52
B.4 Inverse squared line segment . . . 52
B.5 Polynomial line segment . . . 52

C RATS Overview and Command Language **53**
C.1 Overview . . . 53
C.2 Command Language . . . 54
C.3 Two examples of internal man pages . . . 59
C.3.1 Materials . . . 59
C.3.2 Textures . . . 60
C.4 Selected Datasets . . . 61

Bibliography **67**

List of Figures

1.1 Blinn's Blob. 1

2.1 An implicit surface generated by two point sources. 7
2.2 One-dimensional convolution. 8
2.3 A convolution surface produced by an implicit line segment. 9
2.4 A Gaussian function. 9
2.5 A new kernel function. 10
2.6 Both kernels. 10
2.7 Convolving a line segment with various kernels. 13
2.8 Finding integration domain for line segments. 13
2.9 Finding integration domain for triangles. 14
2.10 Point modeling primitive. 15
2.11 Line modeling primitive. 15
2.12 Arc modeling primitive. 15
2.13 Development of the triangular primitive. 16
2.14 Triangle modeling primitive. 16
2.15 Triangle modeling primitive, revisited. 16
2.16 Family of convolved starfish. 17
2.17 Bounding volumes of the family of convolved starfish. 17
2.18 Implicit sphereflakes. 18
2.19 Blends between primitives of different types. 18
2.20 'Explicit' and 'implicit' coral trees. 18

3.1 Components of a convolution surface. 22
3.2 Implicit primitives. 23
3.3 A skeleton of a coral crab. 23
3.4 A T-shaped convolution surface. 24
3.5 Elements on an implicit icicle. 24
3.6 Individual assignment of local parameters. 25
3.7 Profiling functions. 25
3.8 The main modeling loop. 27
3.9 Shell I. Point-based spherical shell. 28
3.10 Shell II. The offset surface and the convolved surface of an unidentified mollusk. 28
3.11 Shell III. Superposition of two skeletons. 28
3.12 Spindle Cowrie. 29
3.13 Seaweed, modeled with linear profiling functions. 29
3.14 Coral tree, based upon a seaweed model. 29
3.15 Seahorse: A hand drawing, an offset surface and the convolved shape. 29
3.16 Coral crab. 29
3.17 How to make volumetric wrinkles. 30
3.18 Nefertiti, before and after convolution. 31
3.19 Aging Yoda. 31

4.1 Polynomial approximation. 37
4.2 Rendering an implicit icicle. 38
4.3 Implicit cross, front view with uniform error distribution 40
4.4 Implicit cross, side view with visible distortions. 40
4.5 Implicit cross, side view revisited. 40
4.6 Creating permanent clusters. 41
4.7 Implicit sphere-flakes of various complexity. 42

4.8 Time-profiling charts. 42
4.9 Rendering of thin objects and texture mapping. 43
4.10 Coral tree, convolved with a Gaussian kernel. 43
4.11 Hermite crab 43
4.12 “Spinal Starecase”. 44

A.1 Cauchy function. 49
A.2 Gaussian function. 49
A.3 Inverse function. 50
A.4 Inverse squared function. 50
A.5 Metaballs. 50
A.6 Soft objects. 50
A.7 W-quartic polynomial. 50

B.1 Cauchy line segment. 51
B.2 Gaussian line segment. 51
B.3 Inverse potential line segment. 52
B.4 Inverse squared potential line segment. 52
B.5 Polynomial line segment. 52

List of Tables

2.1 Primitives/kernels compatibility chart. 11
2.2 Computational costs for point and line primitives. 12
2.3 Timing test results and speed ratings for points and line segments. 12
2.4 Summary of the timing tests. 12
2.5 Rendering convolved starfish 17
2.6 Timing tests for new modeling functions. 19

3.1 Modeling of spiral seashells. 30
3.2 Summary of modeling techniques. 32
3.3 Rendering time for offset surfaces and convolution surfaces. 32

4.1 Ray-surface equations along the ray's path. 38
4.2 Bounding volumes for modeling primitives. 39
4.3 Optimization methods and rendering times. 41
4.4 Rendering times for the Sphereflake model. 42
4.5 Rendering times for the Spinal Starecase model. 44
4.6 Implemented implicit primitives. 44

Summary

Implicit surfaces obtained by the convolution of mixed dimensional primitives with some potential function, are a generalization of popular implicit surface models: blobs, metaballs and soft objects. These models differ in their choice of potential function but agree upon the use of underlying modeling primitives, namely, points, which puts severe limitations on modeling process and restricts the application base of such models.

In this dissertation a method is described for creating and rendering convolution surfaces built upon an expanded set of skeletal primitives: points, line segments, polygons, curves and planes. Analytical solutions to the convolution integral are presented for a number of implicit primitives and potential functions. A comparative analysis for a number of convolution kernels is given.

In addition to conventional techniques, commonly used in implicit modeling, this dissertation describes a set of new modeling methods, which offer a better flexibility for modeling with implicit surfaces.

Finally, an efficient ray-tracing algorithm is presented, which is capable of producing high-quality images of objects modeled with convolution surfaces. The algorithm outperforms all ray-tracing algorithms in its class known to date.

Developing, modeling and rendering issues, discussed in this dissertation are illustrated by original images developed and rendered by the author.

Statement

Most parts of this dissertation appeared as technical reports published by the Department of Computer Science [42, 62, 63] and the School of Computer Science and Software Engineering [65] at Monash University, Australia. Additionally, the following papers, based on the material presented in the dissertation, have appeared or are to appear:

- "Kernel functions in convolution surfaces: a comparative analysis", accepted for publication in *The Visual Computer* in 1999. Based on Chapter 2.
- "Creating and Rendering Convolution Surfaces" (co-authored with Jon McCormack), *Computer Graphics Forum*, 17(2), 1998. Based on Chapter 2.
- "Interactive shape design with convolution surfaces", accepted for presentation at *Shape Modeling International '99* conference in The University of Aizu, Japan, March 1999. Proceedings of the conference are to be published by IEEE Computer Society Press. Based on Chapter 3.
- "Fast Ray Tracing of Implicit Surfaces", presented at the Third International Workshop on Implicit Surfaces *Implicit Surfaces '98* in Seattle, USA, and published in the conference proceedings. This paper was voted as the best paper of the conference and was recommended for publication in *Computer Graphics Forum.* Based on Chapter 4.

All software used to produce the pictorial material presented in this dissertation was written by the author. This amounts to nearly 60,000 lines of code.

To the best of my knowledge, this dissertation contains no material which has been accepted for the award of any other degree or diploma in any university or other institution.

Acknowledgments

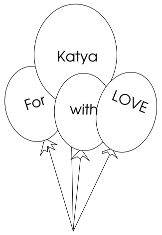
I am very grateful to Jon McCormack for his advice, guidance and support during the work on this dissertation. His patience and skills in pruning the search space of my research were invaluable.

I am also indebted to Peter Tischer for his advice and discussions and to Jules Bloomenthal for finding time and will to read and correct the early versions of my papers. I am also grateful to John Crossley for his help with proof-reading of my research reports.

My very special thanks to Jim Kajiya for introducing me to the wonderful world of computer graphics in general and ray-tracing in particular.

I want to thank David Albrecht and Alexander Kolesnikov for helping me to sort out things with finite-support functions. Many thanks to Ken Shoemake for his help and discussion on variable arc length in convolution of cubic primitives.

Most of all, I want to thank Katya for her love and encouragement which were the best resources that I enjoyed during these long years of my study.



Chapter 1

Introduction

1.1. A BRIEF HISTORY OF IMPLICIT SURFACES AND A LITERATURE REVIEW

It has been almost two decades since James F. Blinn put two Gaussian functions together and produced one dumbbell-shaped object that he called a *blobby molecule*. The first implicit blend of surfaces in computer graphics was born [7].

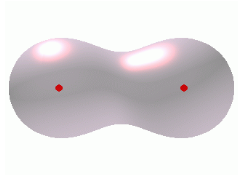


Figure 1.1: Blinn's Blob.

Almost immediately, the blobby molecule grew to an impressive size of over 4,000 atoms to present the DNA molecule in Carl Sagan's COSMOS TV Series.

At approximately the same time, Japanese researchers at Osaka University and Matsushita Electronic Industrial Co. developed a system for modeling objects with density distributions approximated by piecewise quadratic polynomials, better known as *metaballs* [48, 49, 50]. Unlike Blinn's blobby molecules, which were invented to represent blending shapes, the development of metaballs was largely motivated by the need of making smooth metamorphoses in animations. Computer graphics artist Yoichiro Kawaguchi used the metaballs system, running on a parallel computer LINKS-1 to produce the animated film *Growth: Mysterious Galaxy* [39]. *Growth* became a big success at the SIGGRAPH film show in 1983 and in years to follow.

> ...It was completely different from previous computer imagery in that the objects were comprised not of inorganic lines, but of living, curved surfaces. When the reaction to *Growth* was one of raised goose-pimples, as though reacting to a nightmare, I knew that *Growth* was a success!
> *Yoichiro Kawaguchi* [41] [1] .

What makes implicit modeling so successful? Why do objects, produced with this method look like '...living, curved surfaces'? Why are they called implicit in the first place?

Implicit surfaces and implicit modeling received their name after the method the surfaces are defined. A surface S may be defined as a set of points $\mathbf{p} = (x, y, z)$

$$\begin{aligned} \text{parametrically:} \quad & S = \mathbf{p} \mid x = x(t), y = y(t), z = z(t)\}, \\ \text{explicitly:} \quad & S = \mathbf{p} \mid z = F(x, y)\}, \\ \text{implicitly:} \quad & S = \mathbf{p} \mid F(x, y, z) = 0\}. \end{aligned}$$

It is convenient to think of an implicit surface as an isosurface, formed in a scalar field $F(x, y, z)$ at certain threshold T:

$$S = \{(x, y, z) \mid F(x, y, z) = T\}$$

The field function F is often presented as

$$F(x, y, z) = \sum_{i=1}^{N} f_i(x, y, z) \tag{1.1}$$

The summation is performed over all constituent functions $f_i(x, y, z)$, each of which defines a density distribution in 3D-space. For example, the object pictured in Figure 1.1 is modeled by a sum of two Gaussian distributions

$$e^{-a_1 r_1^2(x,y,z)} + e^{-a_2 r_2^2(x,y,z)} = T,$$

where r_1 and r_2 are distances from the centers of the first and the second component and a_i are blending factors. If the modeling functions $f_i(x, y, z)$ are continuous and smooth, the resulting surface will exhibit seamless blends between its constituent parts. Changes in number, position or properties of the modeling components will cause changes in the resulting shape.

By adjusting these parameters, a designer controls all regions of the surface.

Density distribution functions $f_i(x, y, z)$, used in the main modeling equation (1.1), may be defined in a variety of ways. Blinn used exponentially decaying

potentials [7]. The metaballs system employed piecewise quadratic polynomials [50]. In the *soft objects* model, introduced by Wyvill et al. [73], the density function is given by a polynomial of degree 6. Another popular function is a quartic polynomial, used in many public domain ray-tracing programs [56, 58]. The exact formulae for all these functions are given in Appendix A. In their basic implementations, all of these modeling functions are spherically symmetric and sigmoidal with respect to the center of the density distribution.

It has been acknowledged that spherically symmetric modeling functions are not adequate for all modeling tasks. For example, flat or cylindrical surfaces cannot be represented by a collection of spherically symmetric components. Similarly, elongated objects like limbs, branches and tentacles also require a better representation. The search for a wider range of implicit shapes took two conceptually different directions [2].

- *Point primitives with anisotropic distance functions.*

  This approach was proposed by Blinn in his seminal paper [7]. He suggested using general quadrics in order to compute the distance to the center of the primitive. Essentially, this method changes the space metric around spherical field functions and makes them look like quadrics: ellipsoids, cones, cylinders, etc.

  Superquadric implicit primitives can also be created this way, as described by Wyvill and Wyvill [78] and Kalra and Barr [38]. Blanc and Schlick [6] presented ratioquadrics, as a computational improvement of the superquadric model. Ratioquadrics and superquadrics are also based on the notion of anisotropic distance function. Many other useful field functions are reviewed in [5] and [78].

  Perhaps the most advanced modeling function in this class is an *implicit sweep object*, described by Crespin et al. [20] [3]. Swept objects are also described by Sourin and Pasko [68] in the context of more general modeling concept, the Function Representation [1, 43, 52].

- *Non-point primitives with isotropic distance functions.*

  With this approach, the modeling implicit functions are built around some non-point geometric objects, typically, a collection of lines, splines and polygons. Each point on such an object is assumed to carry a certain potential charge which decays symmetrically in all directions. To compute the value of a modeling function f_i at some point of interest (x, y, z), the potential may be measured either from the nearest point on the underlying object, or it may be summed from all points that belong to the object. Surfaces, modeled with the former approach are known as *distance surfaces*. They are surveyed in [11], [15] and [16]. Surfaces, modeled with the latter approach are called *convolution surfaces* [12, 13, 19, 42, 60].

Of all the multitude of modeling methods that utilize implicit functions, we have mentioned only those that are often referred to as 'classic implicits'. Essentially, these methods provided the environment in which convolution surfaces evolved, which will be the topic of this dissertation.

1.2. GENERALIZATION OF CLASSIC IMPLICIT MODELS

Convolution surfaces were introduced into computer graphics by Bloomenthal and Shoemake [12] as a generalization of point-based implicit models [7, 50, 73], extended to multi-dimensional modeling primitives. In short, a convolution surface is the implicit surface, based upon distribution functions f_i, each of which is obtained via a 3D-integration

$$f(\mathbf{r}) = \int_{\mathbf{V}} g(\mathbf{p}) h(\mathbf{r} - \mathbf{p})\, d\mathbf{p}, \qquad (1.2)$$

where h and g are *potential* and *geometrical* functions of the modeling primitive, respectively. The integration is performed over the volume $\mathbf{V}$ of the primitive. Equation (1.2) is commonly referred to as the convolution integral; the function f($\mathbf{r}$) is called a convolution field function. In Chapter 2, we will describe their properties in more detail.

[2] It must be noted, that the brief description given below by no means should be taken as a complete classification of modeling functions. To date, implicit modeling have gathered under its roof an immense variety of tools and concepts. For a broader view, the reader is invited to consult [52], where a unified representation of many methods is suggested, called *function representation* or *F-rep*. To demonstrate the width of this representation, consider a modeling task of creating a hair-style for a virtual actor. The F-rep solution for this task involves "...procedurally defined real functions with the use of solid noise, sweep-like technique, offsetting and set-theoretic operations, and non-linear deformations. More details on hair modeling can be found in the paper [67]" (quoted from [61]). The resulting hair looks like real hair (not shown) and nothing like a 'characteristic' implicit object pictured in Figure 1.1. More on function representation may also be found in [1, 43].

[3] An implicit sweep primitive may also be classified as a distance surface (see [11], [15] and [16]), based on some curve in 3D-space. However, the authors [20] presented it as a point source with an anisotropic field function. The modeling complexity is achieved by specifying a 3D-trajectory the point follows and the transformations that its field function undergoes.

Convolution surfaces received a thorough study in a doctoral dissertation by Jules Bloomenthal, “Skeletal Design of Natural Forms” [13] and other works [8, 10, 12, 16] with special attention to modeling and rendering issues. Using convolution as a mathematical apparatus, Bloomenthal developed a technique for creating the smooth, continuous forms that are ubiquitously present in nature.

An alternative formulation and motivation for convolution surfaces is described by Sealy and Wyvill [60]. They examined how convolution technique may be applied for smoothing models that are common in engineering applications.

1.3. OPEN QUESTIONS

There are many aspects of the convolution model that may be significantly improved. These problems fall into a framework, that is characteristic for most modeling techniques used in computer graphics: Formulation, Modeling, Rendering.

- *Formulation.*
  Questions in this category relate to how the method is defined mathematically and applied algorithmically. Topics of interest are: how rigorously the model is developed, is it consistent within to its own parts, does it allow extensions, are there special cases when the model may break down, etc.
- *Modeling.*
  These questions are of a more applied nature. Once a modeling method is developed mathematically, one has to find out how this method can be used for practical tasks. How wide is the application base of the method? How flexible and how intuitive is the process of creating new shapes?
- *Rendering.*
  What algorithms may be used for visualizing the model? What are the trade-offs between rendering speed and accuracy? Is an intermediate representation (e.g., polygonization) required? Are there any additional requirements in storage, etc.

For the convolution surfaces as a modeling concept, all these questions translate into the following concerns.

1.3.1. Formulation

The field functions f_i used in the convolution surface model are defined via spatial integration (1.2). For most modeling primitives (with the exception of points and line segments), the convolution integral (1.2) does not yield closed form solutions [4] easily and requires special methods for its evaluation. These methods normally involve approximation and storage of intermediate results. We will give two examples how the problem of evaluating (1.2) was approached by previous researchers.

Bloomenthal and Shoemake [12] described how to evaluate the convolution integral (1.2) for polygonal primitives. For a potential function h, they used a separable Gaussian function

$$h(x, y, z) = e^{-(x^2+y^2+z^2)/2}$$

which allowed them to compute the convolution integral (1.2) as a product of two convolutions, planar $e^{-(x^2+y^2)/2}$ and perpendicular $e^{-z^2/2}$. However, the intermediate planar convolution (in the polygon's plane $z = 0$) appeared to be too difficult to perform analytically and required a raster approximation, which is performed as follows. First, a polygon is scan-converted into its digital image in the $z = 0$ plane and then convolved with a Gaussian two-dimensional filter, using well-established algorithms from digital signal processing [23]. Then, the point of interest (x, y, z) is projected onto this plane and a planar component of the convolution is obtained from the filtered image. Finally, this value is multiplied by a perpendicular component of the Gaussian function, $e^{-z^2/2}$. The results of the intermediate planar convolution were originally stored as eight-bit integers [12] and were later replaced by real-valued tables to reduce quantization effects [13]. The demands on memory are $O(n^2)$ with respect to image resolution.

Sealy and Wyvill [60] suggested computing the convolution integral (1.2) using volume sampling techniques and tri-linear interpolation between the nodes. They deliberately replaced the actual integration (1.2) by a discrete summation over the model space, aiming for a more general approach. The trade-off for this generality is the usual 'accuracy-vs-storage' problem. In a brute-force uniform voxel representation memory demands are $O(n^3)$, therefore, Sealy and Wyvill used an adaptive octree representation.

> *We would like to emphasize, that at present, the lack of closed form methods of evaluating the convolution integral (1.2) is the major drawback of the convolution surface model. In general, the model requires a raster representation for two dimensional modeling primitives and volume*

[4] When saying 'closed form solution', we mean a function that can be expressed via elementary functions and algebraic operations and its evaluation does not require procedural techniques.

representation for three dimensional primitives. Both representations use point sampling evaluation techniques that may cause undersampling artifacts. Also, they require large volumes of memory for storage, which may become prohibitive for complex scenes.

The necessity of point sampling and storage makes the convolution surface model conceptually discrete, even though its mathematical foundation requires and implies continuity.

1.3.2. Modeling

Due to difficulties in formulation and actual implementation of convolution surfaces, the practical applications of this method did not receive full attention and development. Yet, modeling capabilities of this method are immense.

The convolution surface model provides a continuous description of a volume of space around the object. This information may be used for various modeling tasks, including those related to the objects' appearance and tasks of controlling the objects' behavior as well. A short excursion in school physics will help to illustrate this point.

According to its definition, the value of a convolution integral (1.2) is in fact a total potential $f(\mathbf{r})$, generated by a 3D object at point $\mathbf{r}$. Sometimes, it is convenient to think of $f(\mathbf{r})$ as of an electrostatic potential, because then one can say that the object that generates this potential is "charged" electrically. Alternatively, $f(\mathbf{r})$ may be associated with a gravitational potential field produced by a massive body, or some other force-generating potential field for that matter [5]. In any case, in order to render the object, represented as an isosurface in $f(\mathbf{r})$, the values of the normal vector

$$\mathbf{N}(\mathbf{r}) = -\nabla f(\mathbf{r})$$

must be computed at every point on the surface of the object. Notably, the gradient vector, as shown above (i.e., without normalizing), gives the vector value of the physical force that is associated with the potential. Thus, if one is creating a physically-based animation of interacting solids that are represented by convolution surfaces, one gets the forces at each point of each object for free! That may help greatly in computing trajectories, collisions and even mutual deformations of the objects.

To illustrate, consider the following example. With the help of the convolution surface model, it is easy to simulate, visualize and animate how things may evolve in a gravity field of a cigar-shaped, a cross-shaped or a donut-shaped planet. To compute the gravitational forces of such unusual configurations, it suffices to evaluate the convolution integral and its gradient for the corresponding geometric primitives that represent the basic shape of such planets (which would be a line, a cross and a circle, respectively). Observe, that in order to simulate conventional gravity, a Newtonian gravity potential $1/r$ must be used as the convolution kernel h. Nothing prevents the designer from creating and exploring worlds where gravity obeys a different potential instead, say, a Gaussian potential, or any other distribution that can be used as a kernel with the convolution integral (1.2) [6]. Appendix B shows the possible gravitational potentials of a cigar-shaped object, including Newtonian $1/r$. Antigravity in such imaginary worlds can be achieved by negating the resulting force vector [7].

Even in the domain of more traditional modeling tasks, i.e., for purely geometric shape modeling, convolution surfaces may be employed with more versatility. To date, convolution surfaces are still mostly used to represent smooth forms, either of natural [8, 13] or artificial [19, 60] origin, and are not commonly thought of as tools for creating shapes that exhibit high frequencies.

The application base of convolution surfaces is perceived as being limited to blending and branching structures which is an obvious underestimation of their modeling capabilities.

1.3.3. Rendering

In order to render a convolution surface even for a TV resolution image, the main modeling equation $F(x, y, z) = 0$ must be solved hundreds of thousands times. Solving this equation numerically requires, in turn, millions of evaluations of the convolution integral (1.2) per image. To avoid such computational costs, polygonization is usually employed to convert implicit models into their polygonal representations that can be rendered by any polygonal renderer. In

[5] Temperature is another good example of a scalar field. However, it is not related to a force field that can be imagined as easily as gravitational or electrostatic forces.

[6] As already mentioned, the vector field of a gravitational force may be obtained by computing a gradient of the potential distribution. If this distribution is "flat" at the center of mass, which is the case of the Gaussian function, the gravity forces diminish as the observer gets closer to the center of mass! This describes the situation that can be experienced during a journey towards the center of a planet, when the traveler eventually finds himself in a free-fall when he reaches the center.

[7] We remind that Newtonian gravity is a vector force field $\mathbf{F}$, with its magnitude proportional to $1/r^2$. Gravity potential is a scalar field P and $\mathbf{F} = -\nabla P(\mathbf{r})$, so the gravity potential is proportional to $1/r$.

fact, polygonal representation has become a standard in the computer graphics industry, including implicit modeling. For example, details about modeling and rendering practices in computer graphics production company Pacific Data Images are given in [3].

Two sets of problems are related to polygonal representation of implicit surfaces:

- traditional problems of polygonal surface representation;
- problems specific to implicit surfaces.

Problems of the first kind are common for all models that employ polygons for representing three dimensional objects. These problems are: geometric aliasing [8], high storage demands, the need for interpolative shading techniques, which, in turn, spawns a new subset of shading-related problems for ray-tracing algorithms [66]. Most of these common problems are well understood and can be solved with well-established methods developed for polygonal modeling and rendering. Detailed discussions on advantages and disadvantages of polygonal representation of 3D objects may be found in any text on computer graphics, for example, in [25] and [71].

Problems of the second kind are specific for implicit surfaces only and are less explored.

First, the polygonal representation of implicit surfaces is not something that can be taken for granted — this is a large area of research. For a comprehensive review and classification of polygonization methods, see [15], [47] and [80]. In the most general case of implicit surfaces, when nothing is assumed about the modeling function $F(x, y, z)$, there are no guaranteed methods to solve the isosurface equation $F(x, y, z) = 0$ reliably, and, therefore, to polygonize the surface correctly. Since the polygonization algorithms effectively flatten the isosurface between the sample points, small features may be missed if they slip between the samples.

Second, the surface has to be repolygonized each time as the modeling implicit equation $F(x, y, z) = 0$ changes, for instance, when components of the composite object, modeled with an implicit equation (1.1), move with respect to each other. In animated metamorphoses, this routinely happens with every new frame. A dramatic example of how serious the problem of repolygonization may get, is described by Hart et. al. [37]. "The Hollywood film industry issued a challenge to computer graphics community, which was iconified by the character *La Magra* in the vampire movie *Blade*. La Magra was described as a tornadic monster of blood and was expected to consist of as many as tens of thousands of metaballs [9] " (reiterated from [37]). La Magra was also expected to swirl across the scene at very high rates, so that motion blur has to be used to make her movements look realistic. To render motion blur efficiently, a polygon-based representation was required. Thus, the real challenge was how to polygonize the surface of such complexity for every single frame. To avoid the brute-force repolygonization, the authors proposed to maintain the polygon mesh dynamically, basing on the method developed in [69]. The idea was to watch for critical changes in the topology of the mesh and make the necessary adjustments locally. Although the preliminary results were promising, a number of unsolved technical problems prevented the successful completion of the project. La Magra was removed from the script.

> *It seems apparent now, that the lack of efficient algorithms for visualizing implicit surfaces restrict their use in the computer graphics industry, especially, in the film production industry, where requirements on image resolution, modeling flexibility and complexity are very high.*

One of the core operations of visualizing implicit surfaces, with or without polygonizing, is solving the implicit equation $F(x, y, z) = 0$. Thus, it is highly desirable to have a method for solving $F(x, y, z) = 0$ rapidly, reliably and with a high degree of accuracy for a wide class of modeling implicit functions $F(x, y, z)$. If this task is achieved, implicit surfaces in general and convolution surfaces in particular could be rendered efficiently, ideally without intermediate polygonization step, i.e. directly from their model representation. The polygonization algorithms could also benefit from such method.

1.4. CONTRIBUTIONS

In this dissertation, we provide our solutions for all three sets of problems, described above.

In Chapter 2, a new mathematical formulation of convolution surfaces is developed. We present a new potential function $h(r)$ that yields closed-form solutions of the convolution integral (1.2) for a wide set of modeling primitives, including points, lines, curves, planes and polygons. These primitives form the basic

[8] 'Geometric aliasing' (also known as 'scaling problem') is an unwanted artifact in computer-generated images, which results in piecewise representation of curved surfaces by meshes of polygons. Geometric aliasing manifests itself during enlargement of an image, when straight edges of the polygons become obvious. This happens, for instance, when a camera zooms in on an object.

[9] 70,000 metaballs, as reported during the paper presentation at the 3rd International Workshop on Implicit Surfaces in Seattle, 1998.

building set of zero-, one- and two-dimensional skeletal elements for implicit modeling. Closed-form solutions eliminate the need for intermediate volumetric computation and storage, which were typical attributes for most implementations of the convolution surface models before [12, 13, 60]. Also, in this Chapter we provide a comparative analysis of alternative potential functions with respect to their compatibility with various modeling primitives and computational costs.

In Chapter 3, we explore the modeling capabilities of convolution surfaces built with the newly developed primitives. We introduce techniques that go beyond simple blends. Using implicit surfaces, we sculpt objects, manipulating their shape at all levels of detail, including fine textures. Methods for creating spikes, horns, wrinkles and other features that are common for many organic objects, are discussed.

Chapter 4 is devoted entirely to rendering. We present a ray-tracing algorithm, specifically designed for rendering convolution surfaces. As of this writing, this algorithm outperforms all other ray-tracing algorithms in its class. The algorithm is based upon a piecewise interpolation scheme, that allows the surface to be approximated as closely as required. Within this approximation, the algorithm is guaranteed to locate the surfaces to a machine-size floating-point precision. The algorithm does not employ numerical root-finding methods, therefore, it does not require iterations and multiple evaluations of the modeling functions. This permits the rapid and accurate rendering of scenes with very high complexity, and with constant computational cost with respect to the silhouette edges. Although the algorithm is implemented as part of a ray-tracing program, its core interpolating and surface-locating modules may be readily used for polygon-generating techniques. However, the high speed of our algorithm allows direct rendering with comfortable interactive rates.

"In 10 years, all rendering will be volume rendering", predicted Jim Kajiya at SIGGRAPH'91. Ergo, all modeling has to become implicit modeling!

While this statement may be a slight exaggeration, we strongly believe that the time for implicit modeling methods to take their place in the Computer Graphics arsenal is long overdue.

We believe that material presented in this dissertation, reveals the true potential of convolution surfaces, a superset of 'classic implicits'. Consistent mathematical formulation, a wide application base and the efficient rendering algorithm make convolution surfaces more practical and powerful than ever.

Each chapter of the thesis is preceded with its own short introduction that supplies the reader with the context of the problems being addressed and provides a review of related work. Also, each chapter is concluded with a brief summary of the results presented and a discussion of their possible application and further improvements. In principle, all parts of this thesis may be read separately. For better readability, we allowed some repetitions of the key notions and definitions.

Chapter 2

Formulation of Convolution Surfaces

2.1. INTRODUCTION

An *implicit surface* S is the isosurface in some scalar field F, defined in 3D space as

$$S = \{\mathbf{p} \in R^3 \mid F(\mathbf{p}) - T = 0\}, \tag{2.1}$$

where parameter T is the isopotential value. Although the scalar field function $F(\mathbf{p})$ may take any form, one of the most commonly used functions is a summation of point potentials:

$$F(\mathbf{p}) = \sum_{i=1}^{N} f_i(\mathbf{p}), \tag{2.2}$$

where $f_i(\mathbf{p})$ are usually Gaussian-like bumps, decreasing with the distance from their centers. More formally, any offset T may be contained within the definition of scalar field F. An example of a typical implicit surface and its constituent parts is shown in Figure 2.1. In this form, implicit surfaces were introduced into computer graphics by Blinn [7] as *blobby molecules*, and by Nishimura et al. [49] as *metaballs*. Later, Wyvill et. al. described their *soft objects* model [73].

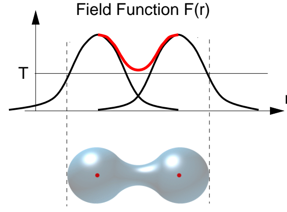


Figure 2.1: An implicit surface generated by two point sources at isopotential value T.

The constituent field functions f_i of equation (2.2) are usually defined to be monotonically decreasing, with a negligible contribution beyond a certain distance from the source. With appropriate field function selection, the resulting isosurfaces provide smooth blending between sources as they are brought together. In animation, surfaces undergo fluid variations, following the changes in position of their constituents. These features, combined with the ability to construct complex shapes that are difficult for other modeling primitives to describe, have made implicit surfaces a popular tool for many modeling tasks, particularly where the shapes to be modeled are from the natural world [7, 12, 13], or exhibit 'soft' properties [73].

The capabilities of any modeling system based on equations (2.1) and (2.2) depend on the choices of modeling primitives, denoted as field functions f_i, and this choice is the essence of the model. As with many modeling techniques in computer graphics, the following dilemma exists: whether an object should be represented by a large number of simple primitives, or by a smaller number of complex ones. With respect to the isosurface equation (2.1), many different approaches have been described. The earliest methods used only simple primitives, such as spherical or ellipsoidal bumps, either Gaussian [7] or polynomial [50, 73]. More complex primitives used in the context of implicit modeling have included superquadrics [38, 78], generalized implicit cylinders [20] and convolution surfaces [12].

These approaches have their advantages and disadvantages with respect to designing and rendering. For designing purposes, it is desirable to have a wide range of field functions f_i representing commonly used shapes. For rendering purposes, these functions should be easy to evaluate in order to locate the isosurfaces efficiently. Naturally, these constraints contradict each other.

Models using simple primitives are well adapted for direct rendering methods such as ray-tracing or ray-casting, because they yield a relatively simple implicit equation (2.1) that can be solved at runtime. However, modeling with point field sources has a number of serious disadvantages. Firstly, using point sources makes the design of complex shapes tedious. In addition, the use of point sources during an interactive modeling session provides little indication of how the final isosurface will appear when rendered. Finally, point-based field functions have difficulty describing circular structures well and can only approximate flat regions.

The use of more complex primitives can solve many of the short-comings of point sources. Convolution surfaces, introduced by Bloomenthal and Shoemake [12] operate with modeling primitives of higher dimensions. These primitives, represented by their field functions f_i, are obtained by spatial convolution between skeletal primitives (such as polygons or curves) and a Gaussian distribution function. Nice properties of convolution, such as superposition and compactness, make modeling with such functions very intuitive.

> Convolution surfaces incorporate the smooth blending power and easy manipulability of potential surfaces while expanding the skeletons from points to lines, polygons, planar curves and regions, and in principle, any geometric primitive.
>
> *Bloomenthal and Shoemake* [12].

While the convolution surface technique is an appealing method for modeling, it is difficult to determine an analytical solution of the field functions f_i, for all but the simplest of primitives. In the original work[12] this problem was circumvented by using precomputed tables. Potentially, this may result in undersampling artifacts and may cause storage problems when the number of primitives used in the model is large.

In this chapter a new method of creating convolution surfaces is presented, based upon analytical solutions for an extended set of potential source primitives. A new potential function is described and a set of field functions is derived for the following widely used primitives: points, line segments, polygons, arcs and planes. Analytical solutions offer the advantage that they provide an exact representation of the surface. Such accuracy can be important, for example, when dealing with primitives of widely varying scales. The resulting functions may be used both for polygonizing or direct rendering of the implicit surfaces formed by the combination of primitives.

2.2. DEFINITIONS

Definition 1 *A convolution surface is the implicit surface based upon a field function $f(\mathbf{p})$, obtained as a spatial convolution of two scalar tri-variate functions $g(\mathbf{p})$ and $h(\mathbf{p})$:*

$$f(\mathbf{p}) = g(\mathbf{p}) \star h(\mathbf{p}) = \int_{R^3} g(\mathbf{r}) h(\mathbf{p}-\mathbf{r})\, d\mathbf{r} \quad (2.3)$$

The geometry function $g(\mathbf{p})$ gives the spatial distribution of potential sources. The potential function $h(\mathbf{p})$ describes the decay of a potential produced by a single point source. Convolved together, they produce the potential distribution generated by all sources in 3D space. Although convolution is a commutative operation, for our purposes we say that the field function $f(\mathbf{r})$ is obtained by convolving a geometry function $g(\mathbf{r})$ with a potential function $h(\mathbf{r})$, also called a *convolution kernel*.

Definition 2 *A convolution kernel $h(\mathbf{p})$ is a tri-variate function*

$$h(\mathbf{p}) : R^3 \rightarrow R \quad (2.4)$$

that describes the scalar field generated by a single point source.

Figure 2.2 gives an a example of one-dimensional convolution with a typical bell-shaped kernel function.

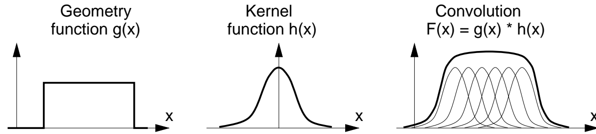


Figure 2.2: One-dimensional convolution.

The general convolution equation (2.3) makes no assumptions about the properties of the geometry function $g(\mathbf{p})$. To evaluate the integral in practice, we restrict our attention to continuous bounded functions $g(\mathbf{p})$ that can be associated with some solid primitive P:

$$g(\mathbf{p}) = \left[\begin{array}{ll} 1 & \mathbf{p} \in P; \\ 0 & everywhere\ else; \end{array} \right. \quad (2.5)$$

For a point modeling primitive, $g(\mathbf{p})$ is a delta-function and the integration (2.3) yields the convolution kernel itself, that is, $f(\mathbf{p}) = h(\mathbf{p})$. For non-point primitives, the convolution integral (2.3) must be evaluated over the volume $\mathbf{V}$ of the primitive:

$$f(\mathbf{p}) = \int_{\mathbf{V}} h(\mathbf{p}-\mathbf{r})\, d\mathbf{r} \quad (2.6)$$

Formula (2.6) describes a generic field function which is a component of a modeling equation for a convolution surface. For practical reasons, we will also refer to it as *an implicit primitive.*

Definition 3 *An implicit primitive $f(\mathbf{p})$ is completely defined by its geometry function (2.5), kernel function (2.4) and the convolution integral (2.3)*

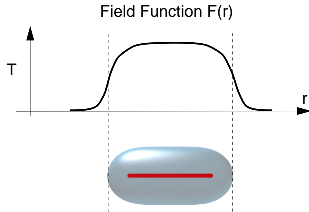


Figure 2.3: A convolution surface, produced by an implicit line segment, rendered in isolation.

When used with the modeling equation (2.1), an implicit primitive $f(\mathbf{p})$ generates an isosurface, as shown in Figure 2.3.

One can easily envisage an implicit primitive as if its potential distribution function $h(\mathbf{p})$ has been 'spread' along the volume of the primitive (see Figures 2.2 and 2.3). Each type of modeling primitive has its own unique geometry (point, line, triangle etc.), and, therefore, yields its own unique field function f_i that can be used in isosurface equations (2.1) and (2.2).

In what follows, the terms 'implicit primitives', 'field functions' and 'modeling functions' are used interchangeably. They all refer to an object, given by definition (3). Similarly, the words 'potential function', 'kernel function' or just 'kernel' are used to denote the convolution kernel as given by definition (2).

2.3. KERNELS AND PRIMITIVES

The formulation of a convolution surface, as given in definition (1), depends on the primitive's geometry $g(\mathbf{r})$ and the convolution kernel $h(\mathbf{r})$. The selection of both components is very important and may influence the effectiveness of the resulting formulation dramatically.

A usable kernel should have the properties that it is continuous, monotonic, diminishes to a negligible contribution beyond a certain distance from the center, and exhibits zero or near zero gradient at this distance. In other words, the kernel should be represented by a bell-shaped function, similar to a Gaussian distribution

$$h(r) = b\ \exp(-a\ r^2)$$

which is shown in Figure 2.4. Parameters b and a control the height and the width of the distribution and are set to 1 in this example; r denotes a Euclidean distance to a point of interest (x, y, z), i.e., $r^2 = x^2 + y^2 + z^2$. This notation will be used further in this section, unless specified otherwise.

Kernels that resemble the Gaussian distribution are in abundance (see references [5, 78] and Appendix A).

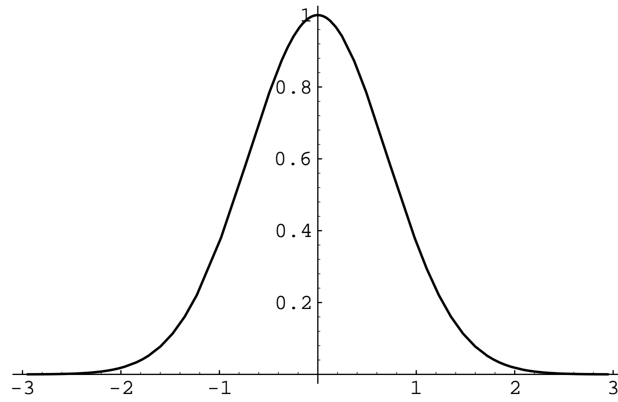


Figure 2.4: A Gaussian function.

However, few of them may be convolved analytically with anything more complex that a point modeling primitive. In fact, at the time this work was undertaken, no convolution kernel has been described that yields a closed-form solution of the integral (2.6) for any modeling primitive but a line segment.

2.3.1. A new kernel function

We propose a new convolution kernel that allows direct analytical solutions of convolution integral (2.6) for a wide family of modeling primitives. The kernel exhibits all the desirable properties for creating smooth surfaces. With this kernel we are able to calculate the *exact* amount of field generated by primitives at an arbitrary point without sacrificing accuracy nor speed. The kernel is:

$$h(r) = \frac{1}{(1 + s^2 r^2)^2}, \tag{2.7}$$

where r is the distance from the point and coefficient s controls the width of the kernel. The kernel is plotted in Figure 2.5.

This function has not been named properly yet. It is reminiscent of a function used in an example by Runge, which is of the form $1/(1 + r^2)$. Runge used this function to demonstrate an oscillatory behavior of the Lagrangian polynomial interpolants, as described in many texts on numerical methods (see, for example [17]). Alternatively, the new kernel (2.7) may also be called a quasi-Cauchy function, because it resembles a Cauchy or Lorentzian distribution, which is $\frac{1}{\pi}\frac{1}{1+r^2}$. For better readability, we will address kernel (2.7) as Cauchy function, keeping in mind that it

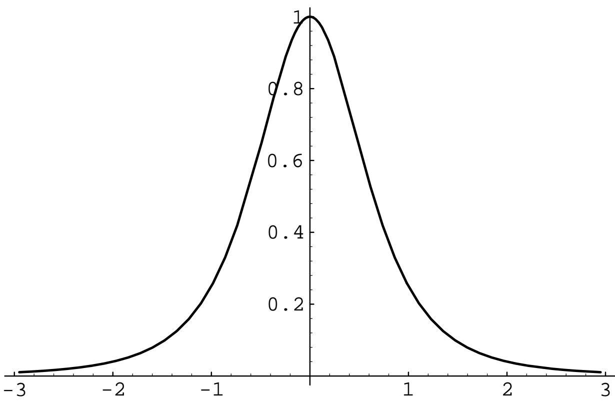


Figure 2.5: A new kernel function.

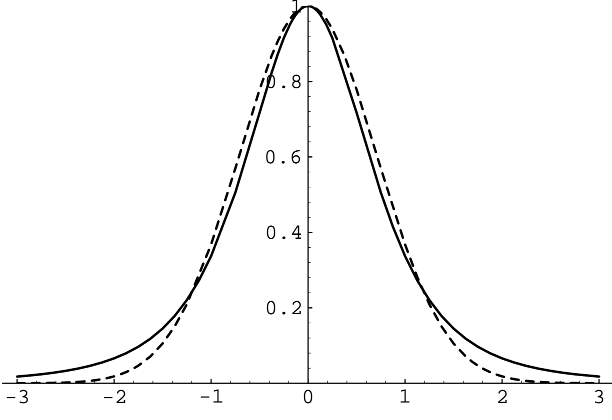


Figure 2.6: A Gaussian (dotted) and the new kernel (solid). Here $s = 0.85$.

is in fact a squared Cauchy function [1].

The question to ask is: what makes the kernel (2.7) better than the multitude of other similar functions, which may look simpler and seemingly more suitable for modeling purposes? In the following sections, we will provide answers to these questions. We will introduce a pool of alternative kernels that have gained wide recognition in the field of implicit modeling and compare them with the newly introduced kernel (2.7).

2.3.2. The seven kernels

The following seven potential functions meet all the requirements for implicit modeling: they are smooth, monotonic and bounded. Their plots are given in Appendix A.

- **Cauchy function**

$$h(r) = 1/(1 + s^2r^2)^2, \qquad r > 0 \tag{2.8}$$

The newly introduced Cauchy function heads the list of possible convolution kernels for alphabetical reasons.

- **Gaussian function**

$$h(r) = \exp(-a^2r^2), \qquad r > 0 \tag{2.9}$$

This function was first used for the purposes of implicit modeling by Blinn in his blobby model [7]. Also, it was used by Bloomenthal and Shoemake as a convolution kernel in the original paper on convolution surfaces [12] and later in [13].

- **Inverse function**

$$h(r) = 1/r, \qquad r > 0 \tag{2.10}$$

Although this function has a singular point at $r = 0$, it still can be used as a potential function. To avoid the division error, the function must be clipped by $1/\epsilon$ for all arguments $0 < r < \epsilon$. The value of ϵ must be chosen small enough so that $1/\epsilon \gg T$ (see equation (2.1)). This will ensure that creases will not occur on the isosurface formed at threshold T. This modeling function was reported to be successfully used in implicit modeling by Wyvill and van Overveld in [80].

- **Inverse squared function**

$$h(r) = 1/r^2, \qquad r > 0 \tag{2.11}$$

The same as above, squared. This function also needs clipping as prescribed for the inverse kernel function $1/r$ to ensure that a division overflow does not occur.

[1] A reader might be wondering why a squared Cauchy function is better than Cauchy function. The reason is that although a Cauchy distribution has a suitable shape for a low-pass filter, it does not yield closed form solutions of the convolution integral. The squared Cauchy function does.

- **Metaballs**

$$h(r) = \left[\begin{array}{ll} 1 - 3r^2 & 0 \leq r \leq \frac{1}{3}; \\ \frac{3}{2}(1-r)^2 & \frac{1}{3} < r \leq 1; \\ 0 & r > 1; \end{array} \right. \quad (2.12)$$

Metaballs are composed of three pieces of quadratic polynomials. They were introduced by Nishimura et. al. in [50].

- **Soft objects**

$$h(r) = \left[\begin{array}{ll} 1 - (\frac{4}{9})r^6 + (\frac{17}{9})r^4 - (\frac{22}{9})r^2, & r < 1; \\ 0 \quad r > 1; \end{array} \right. \quad (2.13)$$

This functions is derived from a piecewise Hermite cubic interpolation over the region of influence of a point potential. Introduced by Wyvill et. al. in [73].

- **W-shaped quartic polynomial**

$$h(r) = (1 - r^2)^2, \qquad r < 1 \quad (2.14)$$

This function is used for modeling blobby objects in many public domain ray-tracing programs, e.g. [56, 58]. The infinite wings of this quartic polynomial are clipped at the unit distance from the center.

As Appendix A indicates, all polynomial kernels (metaballs, soft objects, quartics) have very similar shapes. Since the quartic function is the simplest among them, we have chosen it for further analysis, assuming that the conclusions will be valid for other polynomial kernels as well.

2.3.3. Primitives/kernels compatibility chart

Applied to the main convolution integral (2.6) the kernels h listed above, performed with different degrees of success for different types of geometry functions g. Closed-form solutions were obtained for five types of primitives: point (trivial), line segment, plane, arc and triangle. There have been several attempts made to perform analytical convolution with a cubic curve, which is a very attractive primitive for purposes of implicit modeling. Unfortunately, none of the kernels produced closed-form solutions for this primitive. The difficulties with cubic curves arise from the necessity of using variable arc lengths under the convolution integral. This additional factor ensures that each point on a curve contributes a correct amount of field into the resulting integral. Line segments and arcs may be naturally parameterized by their length, thus reducing this scaling factor to 1. For cubic curves, variable arc length can not be eliminated easily which makes the integration very difficult, if at all possible [2]. The results of the integration are summarized in Table 2.1.

Kernel	**Modeling primitives**				
	point	line	plane	arc	triangle
Cauchy	•	•	•	•	•
Gaussian	•	•	•	·	·
Inverse	•	•	∞	ellip	•
Squared	•	•	∞	•	·
Polynomial	•	•	•	•	·

Table 2.1: Primitives/kernels compatibility chart: (•) closed-form solution found; (·) no closed-form solution found; (∞) integral yields infinite value; (ellip) a solution is expressed via elliptic integrals;

At a glance, the Gaussian kernel produced three implicit primitives and stopped after planes. This is the smallest number of closed-form solutions.

Inverse and inverse squared kernels yielded solutions for various primitives. Solutions for planes, however, did not converge and the solution for arcs is expressed via elliptical integrals of the first kind. For all practical purposes, this result may be dismissed as prohibitively expensive to compute, as it is not expressed via elementary functions.

The polynomial kernel produced solutions for 4 modeling primitives; unfortunately, a strategically important triangular primitive is not one of them.

Finally, the remaining Cauchy kernel covered the whole set of geometric primitives, demonstrating the best modeling flexibility among all the kernels compared.

2.3.4. Computational costs

Besides the issues of compatibility between kernels and primitives, there is another important characteristic that may in some cases help choose the right kernel. This characteristic is computational cost.

Often, a modeling system that employs implicit surfaces, is not designed with the intention of going beyond linear primitives. (For example, the computer graphics production company Pacific Data Images has chosen to use only two implicit primitives, corresponding to a sphere and a tapered cylinder. More on practices of modeling with implicit surfaces

[2] In most cases, the actual integration was carried out with the aid of the symbolic-computation package *Mathematica 3.0* [72], which appeared to be very useful for that purpose. It should be mentioned that the previous versions of *Mathematica* (2.2 and 2.0) are not capable of performing the integration for all cases — version 3.0 or higher must be used.

at PDI may be found in [3].) In order to make implicit points and line segments, as Table 2.1 demonstrates, all five kernels may be employed. Computational cost analysis is needed to choose the most effective implementation.

Table 2.2 presents the computational complexity of all five kernels. The upper table describes the kernels themselves: the numbers of floating point operations and special function calls are taken directly from the kernel's definitions, plus additional expenses to compute the squared distance r^2 from an arbitrary point. The lower table describes five implicit line primitives, obtained via convolution integral (2.6). The resulting expressions are rather bulky. An interested reader may find them in Appendix B. All of these functions define density distributions in 3D-space that are somewhat similar to the example shown in Figure 2.3.

Point primitives

Kernel	Floating point operations					Special
	*	/	+	-	Total	functions
Cauchy	3	1	3	3	10	
Gaussian	3		2	3	8	1 exp
Inverse	3		2	3	8	1 sqrt
Squared	3		2	3	8	
Polynomial	4		2	4	10	

Line primitives

Kernel	Floating point operations					Special
	*	/	+	-	Total	functions
Cauchy	29	6	11	13	53	2 atan 1 sqrt
Gaussian	11		5	11	27	1 exp 1 erf
Inverse	9		9	13	31	2 log 2 sqrt
Squared	7	3	9	13	32	2 atan 3 sqrt
Polynomial	33	3	14	22	72	1 sqrt

Table 2.2: Computational costs for point and line primitives.

As Table 2.2 demonstrates, it is not an easy task to choose the fastest kernel function, judging solely on the number of floating point operations. The use of special functions obscures the analysis and calls for a more practical method of comparison.

In most cases, the computational costs of evaluating non-algebraic functions are argument-dependent. The reason is the following: non-algebraic functions internally are computed via iterations. These iterations converge at different rates, depending on the value of the argument passed. To simulate the 'real-life' situation, a series of timing tests have been conducted for the point and line segment primitive functions. These functions have been evaluated over the bounding boxes of corresponding primitives, of dimensions (6,6,6) for points and (6,16,6) for line segments (see Appendix B for the layout of the bounding boxes and other relevant parameters for line segments). Each volume was organized into a uniform grid of 150^3 nodes, yielding over three million function evaluations. The tests have been performed on a Pentium processor running at 90 MHz and on a 150 MHz Silicon Graphics machine. The running times and relative speed ratings are given in Tables 2.3 and 2.4.

Point primitives

Kernel	Pentium Time:Rating	SGI Time:Rating	Combined Rating
Gaussian	13.93 sec : 5	4.54 sec : 4	**5**
Cauchy	7.19 sec : 3	4.51 sec : 3	**3**
Inverse	10.32 sec : 4	5.40 sec : 5	**4**
Squared	6.54 sec : 2	4.24 sec : 2	**2**
Polynomial	5.32 sec : 1	3.45 sec : 1	**1**

Line segment primitives

Kernel	Pentium Time:Rating	SGI Time:Rating	Combined Rating
Gaussian	48.22 sec : 5	20.29 sec : 4	**5**
Cauchy	40.79 sec : 4	23.35 sec : 5	**4**
Inverse	29.54 sec : 3	14.52 sec : 2	**2**
Squared	28.62 sec : 2	17.48 sec : 3	**3**
Polynomial	15.14 sec : 1	8.80 sec : 1	**1**

Table 2.3: Timing test results and speed ratings for points and line segments.

Points and line segments

Kernel	Prim	Time Pentium	Time SGI	Overall Rating
Gaussian	point segment			**5**
Cauchy	point segment			**4**
Inverse	point segment			**3**
Squared	point segment			**2**
Polynomial	point segment			**1**

Table 2.4: Summary of the timing tests and speed ratings for all kernel functions.

As Tables 2.3 and 2.4 show, the polynomial kernel consistently produces the implicit point and line modeling primitives that are the fastest to evaluate on both processors. This result confirms that polynomial formulations of point-based modeling primitives in implicit models, such as metaballs in Nishimura et al. (1985) and soft objects in Wyvill et al. (1986), are generally considered as an improvement over the

original blobby model, introduced by Blinn (1982), which employed the Gaussian function.

2.3.5. Problems with polynomial kernels

As Table 2.4 shows, the polynomial kernel has the best speed rating among other kernels for point and line primitives. In addition, it has a finite support (i.e., it is defined over a limited range of distances) which yields effective bounding volumes for the resulting modeling primitive. Finally, the polynomial kernel demonstrated a reasonably good compatibility with a a variety of modeling primitives. Why not use polynomial kernels, then?

This is an important questions and we'll spend some time on it. First, consider a simple example of finding the field function of a line segment using the main convolution integral (2.6) [3] and some generic kernel h($\mathbf{p}$):

$$f_{line}(\mathbf{p}) = \int_{V_{line}} h(\mathbf{p} - \mathbf{v})\, d\mathbf{v} \qquad (2.15)$$

Here V_{line} is the volume of the primitive, in this case, the length of the line segment L. Introducing x as a distance along the line segment and keeping in mind that most of the kernels (2.8 – 2.14) are defined via the squared distance to the point, re-write (2.15) as

$$f_{line}(\mathbf{p}) = \int_0^L h(r^2(\mathbf{p}, x))\, dx. \qquad (2.16)$$

As usual, $r(x)$ is the distance from a point of interest $\mathbf{p}$ to a point on the line segment x. Without the loss of generality, let us restrict our attention to points that belong to the line segment, which allows us to re-write the convolution integral as a function of a one-dimensional argument in the local coordinate system of the line segment [4]:

$$f_{line}(x) = \int_0^L h((x-t)^2)\, dt, \qquad 0 < x < L, \quad (2.17)$$

where x denotes a point on a line segment (see Figure 2.7). Now, we can try to plug various kernels into integral (2.17).

It is easy to see that for convolution with kernels of infinite support (2.8 – 2.11), we can integrate along

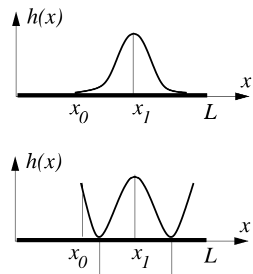


Figure 2.7: Convolving a line segment with Cauchy kernel (top) and polynomial kernel (bottom).

the whole length of the segment L, for any point x_0 (see Figure 2.7, top). In this case, all points x_1, lying on the segment, will contribute the correct amount of their potential at the point of interest x_0. Thus, integral (2.17) may be calculated from 0 to L for all points x_0.

For polynomial kernels (2.12 – 2.14), the situation is very different. Consider the same point x_0 (Figure 2.7, bottom). The contribution from point x_0, positioned farther than the half of the kernel's width $|l_2 - l_1|$, is grossly incorrect. To cut off the infinite wings of polynomial kernels, these kernels must be windowed, i.e., the appropriate interval of integration must be obtained, as pictured in Figure 2.8. The size of this interval $I = [l_1, l_2]$ depends on the mutual position of the line segment and point of interest x_0. Once the values for l_1 and l_2 are obtained, integral (2.17) can be evaluated:

$$f_{line}(x) = \int_{l_1}^{l_2} h((x-t)^2)\, dt, \qquad 0 < x < L, \quad (2.18)$$

The result of this integration may be found in Appendix B.

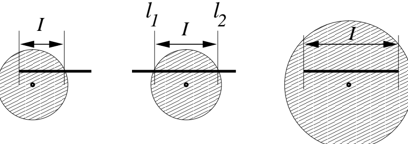


Figure 2.8: Finding integration domain for line segments: three cases of line/sphere intersections.

Thus, finding the proper integration boundaries introduces additional costs for using polynomial kernels. For line segments, these costs involve intersecting a line segment with a sphere — a region where

[3] It should be clearly stated that integration here is analytical, not numerical. Numerical integration can easily be performed by scattering sample points along a primitive.

[4] This simplified integration is used for illustrative purposes only. The real field functions for line segments are defined over 3D space in the world coordinates. They are presented in Section "Development of Implicit Primitives" and also in Appendix B).

the kernel is defined. This operation requires solving one quadratic equation, which is easy to do.

For triangles, this task is much more difficult. Clipping a triangle against a sphere yields a variety of possible configurations. They are pictured in Figure 2.9.

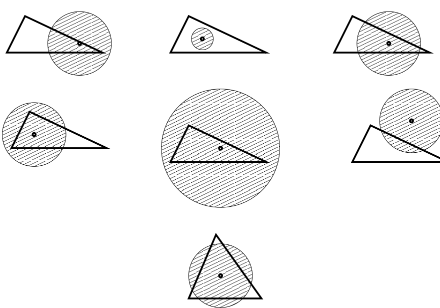


Figure 2.9: Finding integration domain for triangles.

In each particular case, the area of intersection between the sphere and the triangle defines the planar integration domain S:

$$f_{triangle}(\mathbf{p}) = \int_{S(p,x,y)} h(r^2(\mathbf{p}, x, y))\, dx\, dy. \quad (2.19)$$

integrating in the triangle's plane inside region S. In the general case, this task is not suitable for analytical solution. The only easy way to compute the convolution between a triangle and a polynomial kernel (or any other kernel with a finite support) is to ensure that the kernel is always defined over the area of the whole primitive (Figure 2.9, center). This may be achieved by using small triangles or widening the kernel. In both cases, however, triangles would appear more like point primitives and will lose their geometric identity and modeling value, as their size decreases with respect to the kernel's width.

> *To conclude: because of the finite domain size, polynomial kernels are inherently ill-suited for finding analytical solutions of the convolution integral (2.6), unless the characteristic size of the primitive is much smaller than the width of the kernel.*

2.3.6. A short summary

In this section, we have compared a number of kernel functions. We examined their suitability for using with the convolution surface model, paying special attention to

- existence of an analytical solutions to a convolution integral (Table 2.1);
- computational complexity (Table 2.4).

As we have demonstrated, choosing the right convolution kernel is a delicate task that should be approached carefully. Before making that choice, one must decide which modeling primitives have to be 'implicitized' with the convolution integral (2.6).

Sometimes, only the simplest modeling primitives are required, e.g. points and line segments. In such cases, a wide family of kernels may be considered for practical implementation. The speed ratings, given in Tables (2.3, 2.4) may serve as a good criterion for choosing the best kernel.

Another possible selection criterion is the esthetic appeal of the resulting shapes. Each kernel has its unique 'signature' which shows in the way the surface components blend together. For instance, the image of a coral tree shown later in this chapter, is produced with a Cauchy kernel. The same coral tree appears in Chapter 4, this time convolved with a Gaussian kernel function. Although the skeletons of both coral trees are identical, different convolution kernels result in slightly different looking shapes. The difference may be especially noticeable between kernels with finite and infinite support. To illustrate, consider images of blobs, produced with a Gaussian kernel (Figure 1.1) and a polynomial kernel (Figure 2.1). Some people may argue that exponential blends look more 'organic' and polynomial blends are more 'mechanical'. This is a matter of personal choice.

> *For a wider variety of modeling primitives, the first priority should be given to kernels which can be convolved with these primitives analytically. Thus, we have chosen the Cauchy kernel that demonstrated the best computability among other kernels and primitives.*

2.4. DEVELOPMENT OF IMPLICIT PRIMITIVES

Using the new convolution kernel (2.7) shown in Figure (2.5), the field functions for a number of primitives can be derived. These primitives are

- point sources
- line segments
- arcs
- triangles
- planes

Other primitives, e.g. circles and polygons, may be built by combining, respectively, arcs and triangles. In the following section, development of field functions for each primitive is given in detail.

2.4.1. Point sources

A point is a 0-dimensional object, so its geometry function is a simple delta-function at point $\mathbf{p}$ (see Figure 2.10). Integration (2.6) yields the kernel function, projected into 3D-space as:

$$F_{point}(\mathbf{r}) = \frac{1}{(1 + s^2|\mathbf{p} - \mathbf{r}|^2)^2} \tag{2.20}$$

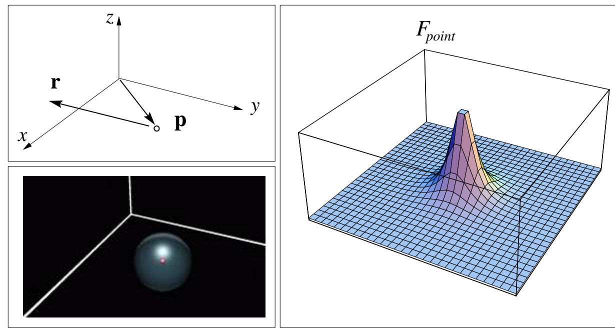


Figure 2.10: Point modeling primitive: geometry (top-left); an intensity plot of F_{point} over the $Z = 0$ plane (right); a rendered version of an isolated primitive (in red) and it's corresponding isosurface (transparent surface).

2.4.2. Line segments

A line segment of length l is defined as:

$$\mathbf{p}(t) = \mathbf{b} + t\mathbf{a}, \quad 0 \leq t \leq l,$$

where $\mathbf{b}$ is the base vector, $\mathbf{a}$ is the normalized axis. The squared distance between an arbitrary point $\mathbf{r}$ and a point on the line segment is

$$r^2(t) = d^2 + t^2 - 2t\mathbf{da},$$

where $d = ||\mathbf{d}||$ is the magnitude of a vector from the segment base to $\mathbf{r}$: $\mathbf{d} = \mathbf{r} - \mathbf{b}$. To obtain the field function, r^2 is substituted into the general formula (2.6) and integrate:

$$\begin{aligned} F_{line}(\mathbf{r}) &= \\ &= \int_0^l \frac{dt}{(1 + s^2 r^2(t))^2} = \\ &= \frac{x}{2p^2(p^2 + s^2x^2)} + \frac{l - x}{2p^2q^2} + \\ &+ \frac{1}{2sp^3}(\mathrm{atan}[\frac{sx}{p}] + \mathrm{atan}[\frac{s(l - x)}{p}]), \end{aligned} \tag{2.21}$$

where $x = \mathbf{da}$ and p and q are distance terms:

$$\begin{aligned} p^2 &= 1 + s^2(d^2 - x^2), \\ q^2 &= 1 + s^2(d^2 + l^2 - 2lx) \end{aligned}$$

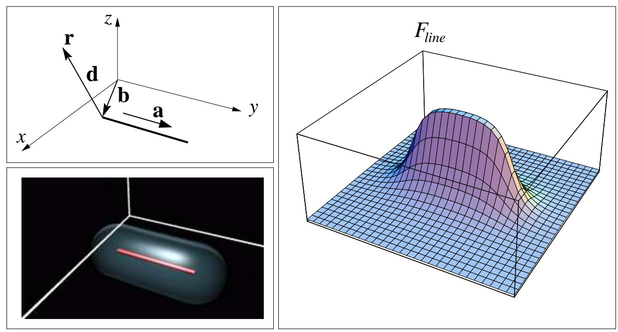


Figure 2.11: Line modeling primitive.

2.4.3. Arcs

An arc and its distance function r^2 are conveniently defined in the arc's local z-aligned coordinate system as:

$$\begin{aligned} \mathbf{p}(t) &= (r\cos(t), r\sin(t), 0), \quad 0 \leq t \leq \theta, \\ r^2(t) &= (x - r\cos(t))^2 + (y - r\sin(t))^2 + z^2, \end{aligned}$$

where r is the radius of the circle the arc lies on, θ is the arc angle (Figure 2.12, top-left). Integrating along the angle, the field function in local coordinates is obtained:

$$\begin{aligned} F_{arc}(x, y, z) &= \\ &= \int_0^\theta \frac{dt}{(1 + s^2 r^2(t))^2} = \\ &= \frac{by}{xp^2(kx - b)} + \frac{k(x^2 + y^2)\sin(\theta) - by}{xp^2(k(x\cos(\theta) + y\sin(\theta)) - b)} + \\ &+ \frac{2b}{p^3}(\mathrm{atanh}[\frac{ky}{p}] + \mathrm{atanh}[\frac{(kx + b)\tan(\frac{\theta}{2}) - ky}{p}]), \end{aligned} \tag{2.22}$$

where $k = 2rs^2$ and distance terms d, b and p are:

$$\begin{aligned} d^2 &= x^2 + y^2 + z^2, \\ b &= 1 + r^2s^2 + s^2d^2, \\ p^2 &= -r^4s^4 + 2r^2s^2(s^2(d^2 - 2z^2) - 1) - (1 + s^2d^2)^2. \end{aligned}$$

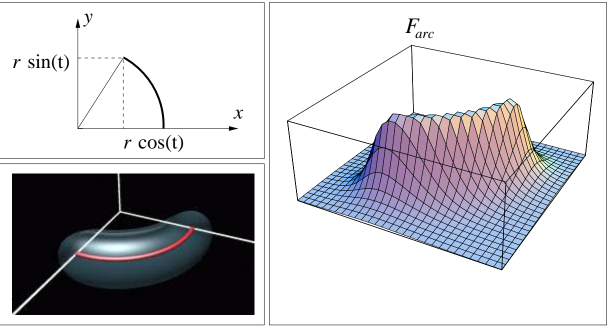


Figure 2.12: Arc modeling primitive.

2.4.4. Triangles

A triangular primitive requires integration in two dimensions, which is slightly more complicated than for one-dimensional cases. First, a triangle is split along the longest edge into two right-angled triangles (Figure 2.13).

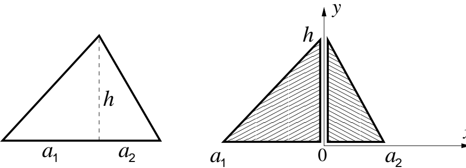


Figure 2.13: Integration parameters.

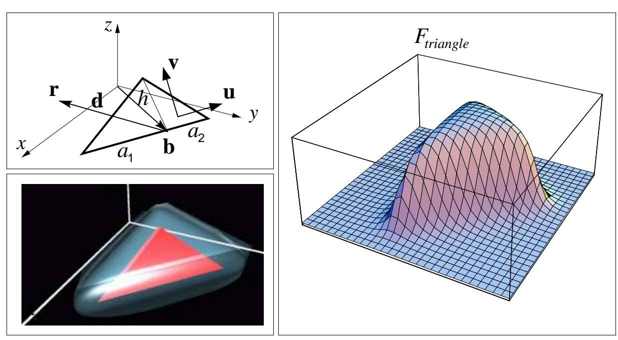


Figure 2.14: Triangle modeling primitive.

Next, the field function for the right half is obtained:

$$F_{right}(x,y) = \int_{y=0}^{h-\frac{hx}{a_2}} \int_{x=0}^{a_2} \frac{dxdy}{(1+s^2r^2(x,y))^2}$$

and F_{left} is derived from F_{right} by replacing x by $-x$ and a_2 by a_1. Finally, F_{left} and F_{right} are added together.

For an arbitrarily positioned triangle (Figure 2.14, top-left), the following parameters are defined: a point **b**, the projection onto the longest edge of the opposite vertex; vectors **u** and **v** that form the local surface coordinate system, with **b** as its origin and **u** aligned in the direction of the longest edge. h is the distance from **b** to the apex of the triangle. Introducing the vector $\mathbf{d} = \mathbf{r} - \mathbf{b}$ and scalars $u = \mathbf{du}$ and $v = \mathbf{dv}$, the final field function is:

$$\begin{aligned} F_{triangle}(\mathbf{r}) &= \\ &= \frac{1}{2qs}(\frac{n}{A}(\mathrm{atan}[\frac{vh+a_1(a_1+u)}{A}] + \mathrm{atan}[\frac{gh+a_1u}{-A}]) + \\ &+ \frac{m}{B}(\mathrm{atan}[\frac{vh+a_2(a_2-u)}{-B}] + \mathrm{atan}[\frac{gh-a_2u}{B}]) + \\ &+ \frac{v}{C}(\mathrm{atan}[\frac{a_1+u}{C}] + \mathrm{atan}[\frac{a_2-u}{C}])), \end{aligned} \tag{2.23}$$

$$\begin{aligned} A^2 &= a_1^2w + h^2(q+u^2) - 2a_1hug, \\ B^2 &= a_2^2w + h^2(q+u^2) + 2a_2hug, \\ C^2 &= 1/s^2 + d^2 - u^2, \\ g &= v - h, \\ q &= C^2 - v^2, \\ w &= C^2 - 2vh + h^2, \\ m &= a_2g + uh, \\ n &= uh - a_1g \end{aligned}$$

Geometrically, the field function (2.23) is a sum of three 3D step-functions, one for each edge of the underlying triangle. They are pictured in Figure 2.15. These step-functions cancel each other at infinity and form a positive triangular bump between the edges.

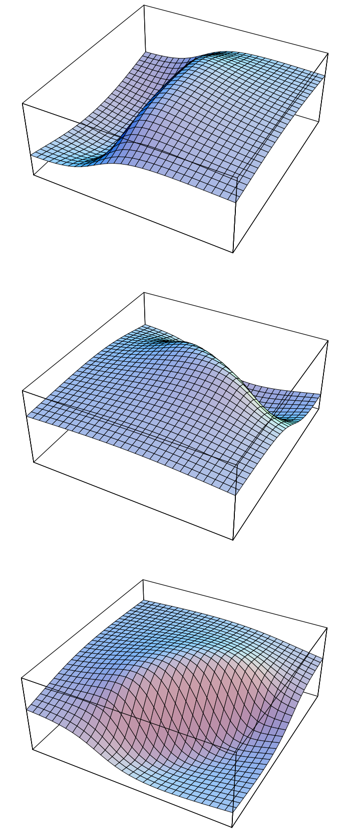


Figure 2.15: Field of a triangle decomposed into three step-functions.

2.4.5. Planes

The field function of an unbounded plane primitive is computed as follows:

$$F_{plane}(\mathbf{r}) = = \int_S \frac{dS}{(1+s^2r^2(t))^2} = = \int_0^\infty \frac{2\pi t\, dt}{(1+s^2r^2(t))^2} = \frac{\pi}{s^2(1+s^2d^2)} \quad (2.24)$$

where d^2 is the distance between point $\mathbf{r}$ and the plane and dS is the area of an integration ring.

In this section, field functions for five modeling primitives are developed: points (2.20), lines (2.21), arcs (2.22), triangles (2.23) and planes (2.24). Some practical examples of using these functions are given next.

2.5. EXAMPLES

Several images have been computed to demonstrate the field functions developed in the previous section. All implicit primitives pictured in Figures 2.16 – 2.20 form blends with each other; all conventional primitives form unions. These images are rendered with a ray-tracing algorithm that will be described in Chapter 4.

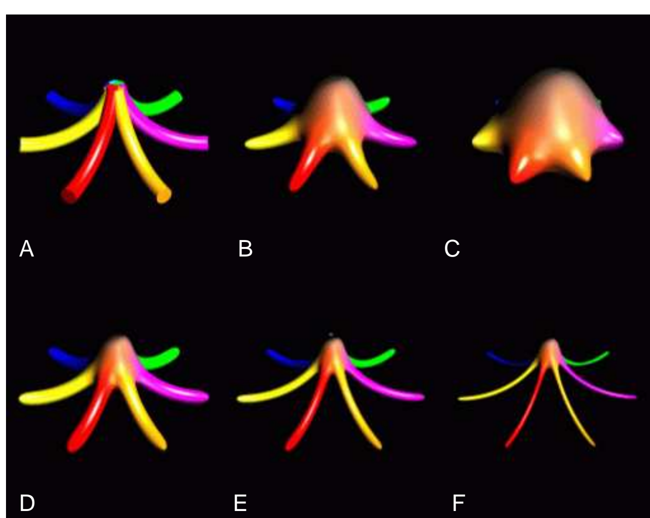


Figure 2.16: Family of convolved starfish. Notice how the colors of the tentacles are blended along the surface.

The starfish-like object in Figure 2.16 is modeled with seven arcs. The top-left image (A) shows the underlying skeletal representation (union of seven arcs). Images (B – F) show the arcs convolved with kernels of various widths and heights. Notice how the color of the tentacle is blended along the surface. Rendering times are given in Table 2.5 (image size 426 x 512, antialiasing).

Image	Rendering time	Ratio to A
A	5 min 59 sec	1.00
B	18 min 47 sec	3.14
C	24 min 01 sec	4.01
D	12 min 02 sec	2.00
E	9 min 18 sec	1.55
F	7 min 51 sec	1.31

Table 2.5: Rendering convolved starfish.

As Table 2.5 indicates, image C in Figure 2.16 (a very "fat" starfish) takes the longest time to render. The reason is as follows: this particular starfish is modeled with implicit arcs that have very large regions of influence (3D areas where their field is significant). Consider Figure 2.17, which presents time-profiling diagrams for all images A - F. Regions of influence are clearly visible as light silhouettes. The rendering algorithm spent most time in these areas evaluating the field functions. The size of the regions of influence is directly proportional to the width of the kernel and the maximal height of the kernel. For example, "skinny" starfishes E and F have relatively small regions, so they were rendered faster.

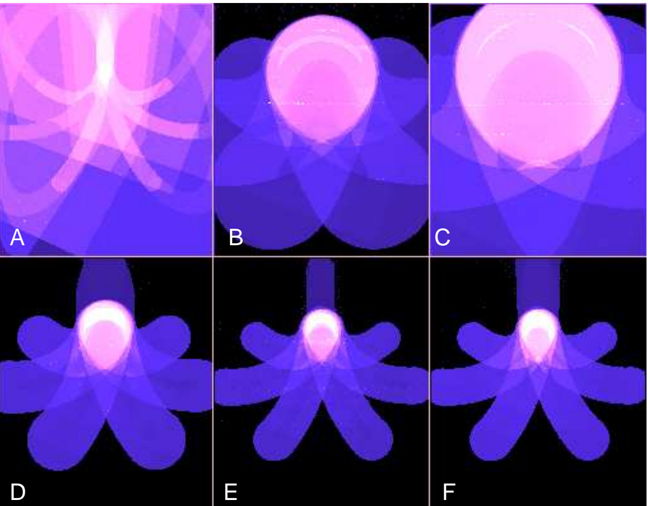


Figure 2.17: Rendering the family of convolved starfish Lighter areas correspond to longer rendering time, revealing regions of influence of all arcs (pieces of tori capped with spheres). Each time-map image is self-normalized from white to black and gamma-enhanced.

Another example of using implicit arcs is shown in Figure 2.18. The well-known sphere-flake model of Eric Haines [31], has been enhanced by adding stems

to each sphere[5]. The stems are modeled with implicit arcs, the spheres – with implicit Gaussian points. Despite the fact that the convolution kernels are different, the resulting surface is perfectly smooth, both geometrically and photometrically.

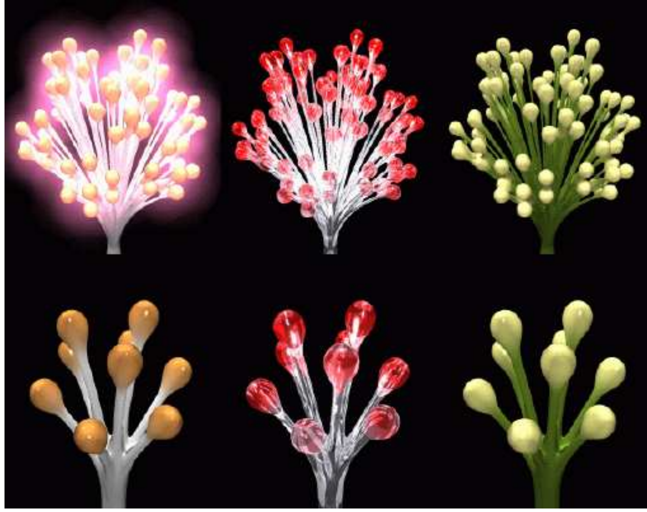


Figure 2.18: Modified implicit sphereflakes: Gaussian point sources (tops) blend with Cauchy arcs (stems).

Figure 2.19 shows some well-known objects, that use arcs, line segments, triangles and planes as their building elements. Primitives of different types blend together smoothly.

Figure 2.19: Blends between primitives of different types: arcs + lines (left); triangle + triangle (right); arcs + lines + planes (bottom).

The images of coral trees in Figure 2.20 illustrate the modeling power of convolution surfaces built with

[5]The sphere-flake model, grown to a full size of 7,381 components, will appear again in Chapter 4, where rendering issues will be discussed.

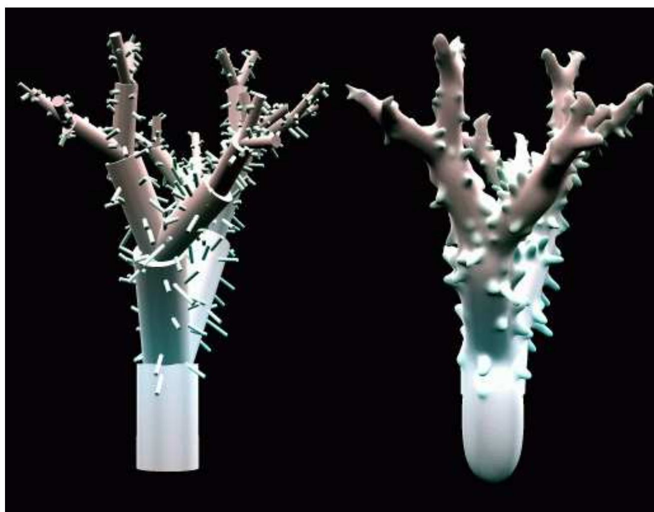


Figure 2.20: Coral tree made of line segments, 'explicit' and 'implicit' versions.

field functions developed in this chapter. The underlying model for both images is based upon a tree model generator by Eric Haines [31]. Both trees consist of 31 branches, modeled with cylinders of various radii, and 512 randomly positioned spikes, also cylindrical.

2.6. PERFORMANCE

The analytical solutions of the convolution integral (2.3), presented by field functions (2.20 – 2.24) are computationally more expensive than alternative procedural approaches [12, 20, 60]. To compare, evaluation of the analytical field function (2.23) of a triangular primitive requires 43 multiplications/divisions, 35 additions/subtractions, 3 square roots and 6 arctans. The procedural method discussed in the original convolution paper [12] uses a precomputed raster representation of a 2D convolution of a triangle, which allows to evaluate its field function using 14 multiplications and 9 additions only.

Some efficiency gains can be made by using tabulated versions of arctan and square root functions. Timing tests, similar to those performed for the comparative analysis of implicit points and lines, indicate the analytical method is approximately 5.4 times slower than the equivalent procedural solution. However, it is reasonable to expect that in a real rendering or polygonizing environment the timing results should be more favorable. The reason is for that is that analytical functions (2.20 – 2.24) are defined in the world coordinate system (except implicit arcs). Thus, there is no need to perform costly world-to-local coordinate system transformations that are typical for evaluation of field functions of complex implicit modeling primitives.

Between themselves, the newly developed implicit primitives demonstrated computational complexity of wide range, as shown in Table 2.6.

Prim	Time t_i	t_i/t_{point}	Bar chart
Point	3.18 sec	1.00	
Line	14.25 sec	4.48	
Arc	25.97 sec	8.17	
Triangle	45.66 sec	14.36	
Plane	3.26 sec	1.02	

Table 2.6: Timing tests for new modeling functions. The tests were conducted on a 90MHz Pentium, 125^3 evaluations for each primitive.

These data give a good indication when the convolution surface model becomes more effective computationally than a brute-force point-sampling technique. For instance, in a general situation an implicit line primitive that is "longer-that-five-points", benefits from convolution representation, both geometrically (no wave pattern) and computationally (it can be evaluated faster).

2.7. CONCLUSIONS

We have presented a consistent and mathematically sound method of creating convolution surfaces based upon exact evaluation of the field functions for a wide set of geometric primitives. Functions (2.20–2.24) provide means to calculate values of their field and the gradient at an arbitrary point with an arbitrary precision.

Thus, we suggested a new formulation of the convolution surface model, that contains no granularity in its definition. The new formulation does not require volumetric computation and storage. We believe it to be a most important contribution to the convolution surface model.

We argue that the set of primitives developed in this chapter can be considered as a necessary minimal set of tools for modeling with convolution surfaces. Indeed, points, lines and triangles are the essential building elements of dimensionality 0, 1 and 2, respectively. In general case, they cannot be approximated by any other data types without introducing undersampling artifacts. On the other hand, due to the additive property of convolution, these primitives may be successfully used in creating more complex data types for implicit modeling. For example, arcs may be combined into circles, spirals and other 3D curves, triangles into polygons, segments into polylines etc.

Convolution surfaces allow modeling possibilities that would be difficult to achieve using other geometric modeling techniques. The analytical solutions for the convolution of higher order primitives as presented above, extends the options for the modeling with implicit surfaces, which will be discussed in the next chapter.

> We are all dreaming of a speech without words that utters the inexpressible and gives form to the formless.
>
> *–Hermann Hesse (1877-1962)*
> *"Steppenwolf"*

Chapter 3

Modeling with Convolution Surfaces

3.1. INTRODUCTION

Convolution surfaces were introduced to computer graphics by Bloomenthal and Shoemake [12] as a logical generalization of the classic models of implicit surfaces: blobby objects [7], metaballs [50], soft objects [73]. Being a superset of these models, convolution surfaces inherit their valuable properties, such as an ability to form smooth blends and free-form shapes. At the same time, convolution surfaces demonstrate much greater modeling flexibility, allowing a designer to create objects using skeletal-based techniques with skeletal elements of various shapes and sizes.

Such qualities make convolution surfaces particularly attractive for design of articulated objects, especially of organic origin. However, the modeling practices, developed over nearly two decades for 'classic implicits' [7, 50, 73] cannot be carried over to convolution surfaces 'as is', without doing an injustice to the convolution surface model. As a more versatile and more powerful tool, convolution surfaces need a better 'manual of operation' to fully reveal their potential.

To date, practical examples of modeling with implicit surfaces in general, and convolution surfaces in particular, mainly exercise their blending abilities. Consequently, the majority of objects, designed with implicit surfaces, demonstrate various branching structures, such as trees [8, 35], hands [13], paws [60], chromosomes [24]; or exhibit certain softness or fluidity, either in animation or as perceived from static images. Examples are: human lips [30] and faces [45], molecular shapes [7], boiling liquids [74, 75] and rubbery-looking objects [77]. Few attempts have been made to employ implicit surfaces for modeling rough objects. One of them is presented in [33].

Starting from the previous work [42, 65], we show that the modeling capabilities of convolution surfaces extend beyond traditional blends. We introduce a set of primitives and a number of techniques that allow us to sculpture objects, manipulating their shape at all levels of detail, including fine textures. We demonstrate that convolution surfaces can be successfully used to represent not only soft and pliable substances, but also objects that are hard and fragile. For that reason, most examples show marine life forms – there are plenty of creatures of all types. We also outline a system architecture for interactive design with convolution surfaces. The system allows objects to be modified at interactive rates, which helps the design to converge quickly to the final shape.

3.2. DEFINITIONS

The following are the basic concepts and equations used in modeling with implicit surfaces.

3.2.1. Implicit surfaces

An *implicit surface* S is defined as an isosurface at level T in some scalar field $F(\mathbf{p})$:

$$S = \{\mathbf{p} \in R^3 \mid f(\mathbf{p}) - T = 0\} \qquad (3.1)$$

An example of a typical early implicit surface is shown in Figure 2.1.

3.2.2. Convolution surfaces

A *convolution surface* is the implicit surface based on a field function $f(\mathbf{p})$, obtained via convolution of some kernel function h and a geometry function g:

$$f(\mathbf{p}) = g(\mathbf{p}) \star h(\mathbf{p}) = \int_{R^3} g(\mathbf{r})h(\mathbf{p}-\mathbf{r})\, d\mathbf{r} \qquad (3.2)$$

The geometry function $g(\mathbf{p})$ defines the shape of an object and its position in 3D space. The kernel function $h(\mathbf{p})$ defines the distribution of some potential that is produced by each point on the object. Convolved together, these two functions produce a tri-variate scalar function $f(\mathbf{p})$ that defines the convolution surface.

Figure 3.1 gives an example of convolution with a Gaussian-like kernel. Here the geometry function $g(\mathbf{p})$ describes a line segment aligned along the x-axis.

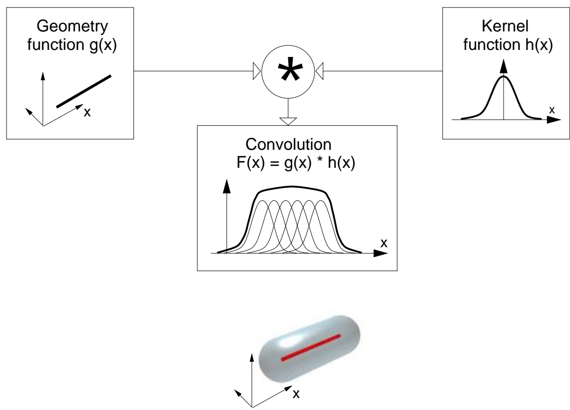


Figure 3.1: Components of a convolution surface: geometry function g (top left), kernel function h (top right), convolution field function f (center), convolution surface (bottom).

3.2.3. Implicit primitives

In the metaballs model of implicit surfaces [50], the point potential sources were introduced as "meta-primitives defined by their distribution function, that together form a meta-surface". With respect to convolution surfaces, we say that the distribution function (3.2) defines an *implicit primitive* with geometry g.

Metaballs, as defined by Nishimura et. al. [50], are essentially meta-spheres (see Figure 2.1). Implicit primitives, produced via the convolution integral (3.2), yield a variety of modeling shapes. For example, a line segment, convolved with a kernel function, produces an elongated shape as shown in Figure 3.1.

Since most modeling primitives form closed compact sets (points, line segments, triangles, etc), the integration (3.2) over 3D space may be conveniently replaced with integration over the volume $\mathbf{V}$ of the modeling primitive:

$$f(\mathbf{p}) = \int_{\mathbf{V}} h(\mathbf{p} - \mathbf{r})\, d\mathbf{r} \qquad (3.3)$$

3.2.4. Skeletons

For the purpose of this thesis, we use the following definition of a skeleton. A *skeleton* is a collection of geometric primitives that outline the inner structure of an object being modeled. With respect to the convolution surface model, a skeleton is a sum of geometry functions $\sum_{i=1}^{N} g_i(\mathbf{p})$. Visually, such a skeleton is represented by a union of corresponding primitives. Convolved with a kernel function, the skeleton yields a field function $f(\mathbf{p})$ and a convolution surface S.

Other researches used slightly different definitions of skeletons [13, 24, 80]. In general, these definitions are similar and imply that an object being modeled may be regarded as composed of distinct components, as opposed to being totally formless. However, the precedence of the skeleton and the shape of the object may be different. In most data-fitting problems [24, 45], skeletons are derived from data associated with the object that has to be visualized. In contrary, for skeletal-based modeling environments [12, 13, 30], the shape of the object follows the skeleton.

The concept of a skeletal design with convolution surfaces was introduced to computer graphics by Bloomenthal and Shoemake in [12]. They observed that the additive property of convolution allows us to build complex skeletons out of simple primitives and, most important, allows us to evaluate them individually. That makes convolution surfaces computationally practical.

Bloomenthal [13] applied convolution surfaces to generate smooth shapes, resembling various organic objects. In what follows, we will use implicit primitives obtained via convolution technique, to produce a wide variety of surfaces, including rough, prickly and wrinkled surfaces.

3.3. THE DESIGN SYSTEM

In this section, we present a set of tools and techniques for modeling with convolution surfaces.

3.3.1. New modeling primitives as skeletal elements

In principle, any geometric primitive may be used as a skeletal element for the convolution surface model by means of the generic integral (3.2). In practice, the choice of such primitives is often limited by technical difficulties of evaluating the convolution integral.

As most implementations of the convolution surface model demonstrate [12, 13, 60], such computations require point-sampling of the field in the model space and storage of intermediate results for rendering[1]. As for techniques that employ point-sampling, special care must be taken to ensure that small features of the object being modeled are not missed. This task may be especially difficult for models that contain primitives of widely varying characteristic sizes.

Closed-form solution for the convolution integral (3.2) provides such a care-free modeling environment. Based

[1]Bloomenthal and Shoemake [12] described how to evaluate the convolution integral (3.2) for polygons. They used a pre-computed raster representation of a 2D convolution of a polygon and then stored results as 2D images. Sealy and Wyvill [60] developed a method for computing the convolution integral (3.2) for 3D objects. Their method involves replacing the actual integration with discrete sums over the model space. To reduce the demands on memory, Sealy and Wyvill used octrees for storage of their 3D data.

on our prior results [42], also given in Chapter 2, we suggest the following implicit primitives as skeletal elements for convolution surfaces:

- points
- line segments
- arcs
- triangles
- planes

All these modeling primitives are presented as closed-form functions, that return the amount of field generated by the primitive at an arbitrary point, calculated to a machine-size float precision. This makes it possible to visualize convolution surfaces using direct rendering algorithms, such as ray-tracing. Thus, neither preprocessing, nor intermediate storage are required to render the surface, which is particularly convenient for interactive design. Figure 3.2 shows the implicit primitives, rendered as semi-transparent surfaces with the underlying 'bare' geometric primitives inside. The actual field functions are presented in Chapter 2.

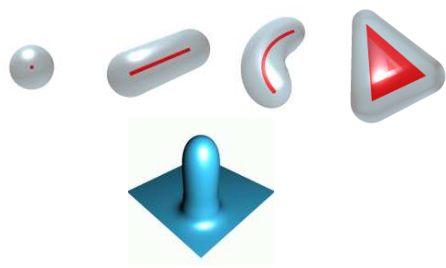


Figure 3.2: A set of implicit primitives: point, line segment, arc, triangle and plane. The plane primitive is clipped and blended with a line segment.

Although most of these primitives have been used for implicit modeling before (points in [7, 45, 50, 73], line segments in [3, 8, 60], triangles in [13, 60]), the closed form formulation of their field functions [42] is still unexplored. These functions constitute the core of our design system.

With a multitude of modeling primitives available, the concept of a skeleton becomes especially life-like, because now a designer may think of skeletons as if they consist of solid 'bones', each of unique shape and size: point, line segments, curves, triangular pieces. In addition, arcs may be combined into circles and spirals, triangles – into polygons, line segments – into polylines, etc. This allows a designer to work with the object, creating and using those elements that fit best for each particular part of the object. An example of such a multi-primitive skeleton is shown in Figure 3.3, depicting a model of a crab. Note that even this simple sketch gives a good idea of what the resulting shape of the crab is going to look like. The completed model of the crab will appear later in section 3.5.

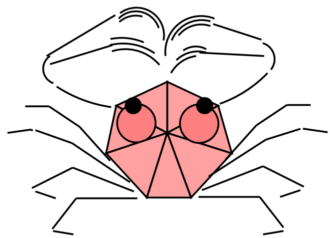


Figure 3.3: The skeleton of this coral crab is composed of 7 triangles, 24 arcs and 24 line segments.

3.3.2. Local properties of implicit primitives

As defined by equation (3.3), a convolution surface is the product of a geometry function g and a kernel function h. A geometry function $g = \sum_{i=1}^{N} g_i$ forms a skeleton, which provides a global description of the object: its general layout, location and the basic shapes of its parts. To complete the description of the object, the local properties of each element g_i of the skeleton must be specified.

Such properties may be conveniently described in terms of the radius in isolation R and blobbiness B, as was suggested by Blinn [7]. A *radius in isolation* is a characteristic distance between a geometric primitive and the implicit surface that it generates. For non-point primitives, the actual distance may vary along the surface. A *blobbiness* parameter controls blending of the object's parts. Very blobby objects, when brought together, tend to form sphere-like shapes with very little or no distinguishable detail preserved. Objects with low blobbiness look more like their skeletons. Figure 3.4 shows a series of convolution surfaces created with various values of blobbiness and radius in isolation. Blinn used these two parameters to define appearance of his point-based modeling primitives. We extend them to all the modeling primitives pictured in Figure 3.2.

The skeleton, radius in isolation and blobbiness of all skeletal elements completely define an implicit surface. The threshold T, used in the implicit surfaces equation (3.1), may be conveniently set to some canonical value, as described in [7].

As an alternative to radius in isolation, an implicit primitive may be characterized by its *region of influence*. A region of influence is the region in 3D space, where the field function f of the primitive has

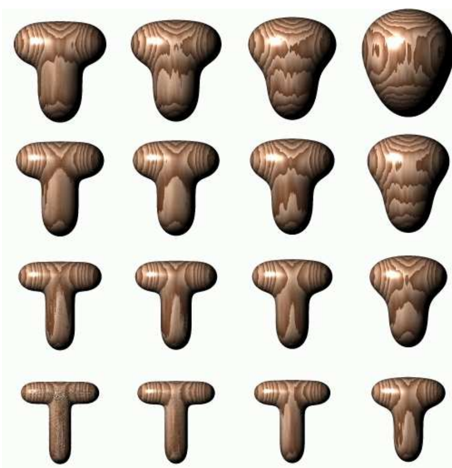


Figure 3.4: A T-shaped convolution surface, formed by two line segments. Increasing blobbiness (left to right) and radius in isolation (from bottom to top) allows us to achieve a variety of forms.

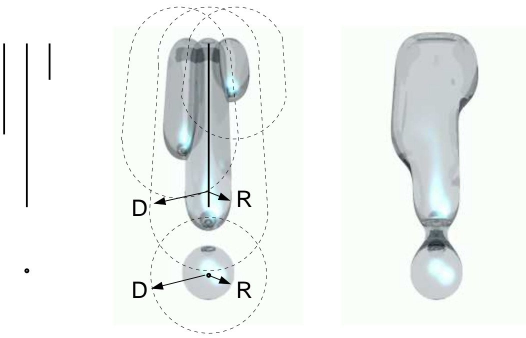


Figure 3.5: Elements on an implicit icicle, from left to right: skeleton made of 'bare' geometric primitives (three segments and one point); their implicit surfaces, rendered as stand-alone objects; the final convolution surface. In the middle image, blending between implicit surfaces is disabled to show the radius in isolation R and depth of influence D better.

non-zero values. A region of influence depends on the geometry of the primitive and the properties of the kernel function. For example, with a spherically symmetric kernel, the region of influence for a point primitive is a sphere; for a line segment — a cylinder capped by two hemispheres, etc. Regions of influence are normally used during rendering to prune out non-significant contributions from distant primitives.

An important property of regions of influence is that implicit surfaces are always located inside the boundaries of these regions. Regardless of the values of blobbiness and radius in isolation, the convolution surface of the icicle in Figure 3.5 (middle and right), always covers its 'bare' skeleton (solid lines) and is always covered by the boundaries of the regions of influence of its components (dotted lines). This property is very valuable for estimating extents of convolution surfaces.

Thus, the depth of the region of influence (or simply depth of influence) may serve as a modeling parameter, which is an alternative to radius in isolation. Figure 3.5 demonstrates the basic elements and parameters of modeling with implicit surfaces.

3.3.3. Materials and elements

Having defined the properties of skeletal elements (which are radius in isolation R and blobbiness B), we must provide a way to associate them with the skeletal elements. This may be done by

- sharing these parameters among a large number of elements;
- assigning them to each skeletal element individually.

Practice shows that both ways have their uses, so we incorporated them both into our modeling system.

3.3.3.1. Materials

The first approach is more convenient for modeling objects that have large areas with the same properties, because it allows employing the notion of a material. Normally, a description of a material includes conventional photometric characteristics, dependent on a lighting model: diffuse color, specular reflectivity, transparency, etc. Such descriptions may be easily enhanced by adding values of blobbiness and radius in isolation, which resemble softness and thickness of the material, respectively. This fits well with the idea of skeletal design: now each 'bone' in the skeleton is covered with some 'soft tissue', which is shared between skeletal elements. As an example, consider the material of an implicit icicle in Figure 3.5:

```
material Ice {
    body           SummerSky,
    ambient        1%,
    diffuse        1%,
    specular       SummerSky, shine 30,
    transparent    40%, index 1.33,
    reflective     50%,
    radius 0.25,   blobbiness -0.30
}
```

In this example, material `Ice` is defined as an almost colorless, reflective and transparent substance. Values of `radius` and `blobbiness` dictate the geometric aspects of the convolution surface, that is based upon the skeleton, defined as

```
object ICICLE
    line Ice, <0 0 0>, <0 0 4.5>
    line Ice, <0 0.5 3.0>, <0 0.5 4.5>
    line Ice, <0 -0.5 1.5>, <0 -0.5 4.5>
    dot  Ice, <0 0 -0.85>
close
```

The material-based description of convolution surfaces has two advantages.

Firstly, for design purposes, it is more convenient to think of skeletons and 'soft tissues' as of separate entities, that can be modified independently of each other. Keeping the material descriptions separate from the skeleton often allows materials of the objects to be changed interactively. This helps many surfaces parameters, both photometric and geometric, to be adjusted without reloading the model into memory for rendering.

Secondly, the local properties of the skeleton are not duplicated for each skeletal element, but are shared via description of a material. (Note the simplicity of syntax for `dot` and `line` modeling primitives in the example above.) This arrangement is memory-friendly, which may be important when the number of skeletal elements in the model is large.

3.3.3.2. Individual elements

The individual assignment of parameters R and B is needed when the model requires each primitive to be strictly different from its neighbors. Such models often come from procedural methods, especially in simulation of growth [31, 55]. For example, consider a sequence of spheres, pictured in Figure 3.6, that is characteristic for solid-based models of seashells [55]. In the domain of implicit modeling, each sphere depicts a region of influence of a point primitive, so the whole object must be described as

```
object SHELL
    sphere Plaster, r1, x1 y1 z1
    sphere Plaster, r2, x2 y2 z2
    ...
close
```

Here $r_1, r_2, ..., r_i$ denote the depth of influence of each point, triples (x, y, z) give locations of their centers. The value of blobbiness is stored in the specifications for the material `Plaster`, defined elsewhere, similarly as it was done for the material `Ice`. All seashell models that will be discussed further are modeled by setting individual depths of influence for each element in their models.

3.3.4. Profiling as a means for achieving variety

Figure 3.2 shows the basic primitives in their canonical, undeformed shapes, which may not be the best for a particular modeling task. Rather, these shapes provide a platform for variations and experiments.

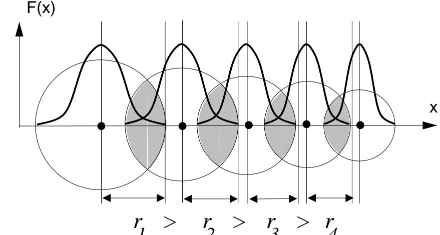


Figure 3.6: Strictly decreasing regions of influence of these implicit point primitives call for individual assignment of their values.

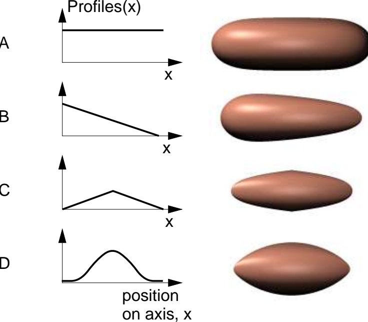


Figure 3.7: Line segments with modified field functions: constant (A), linear (B), hat (C), $sin^2(x\pi)$ (D).

As shown in Figure 3.7, an undeformed line segment (A) easily assumes various shapes (B, C, D), following the *profiling functions*. Profiling functions scale the values of the field, causing changes in the resulting implicit surface. Since line segments are axially symmetric, each profiling function, pictured in Figure 3.7, defines a surface of revolution around the axis. Profiling technique scales the value of a field function, not the final iso-surface. In terms of the modeling equation (2.2) profiling functions modify values of each constituent f_i. For example, the field function of a line segment shown in Figure 3.7 is

$$F_{modified}(x, y, z) = F_{line}(x, y, z)\, sin^2(x)$$

where $F_{line}(x, y, z)$ may be any of the functions given in Appendix B.

To continue a parallel with skeletons, profiling functions provide a mechanism for simulating changes in thickness and/or softness of the tissues along the bones, e.g., line segments. There are infinitely many ways to modify the field function of an implicit primitive. Profiling functions are used with nearly all examples of modeling with convolution surfaces.

3.3.5. Offset surfaces as visual aid and the data structures

As Figure 3.2 shows, the stand-alone implicit primitives based upon points, line segments, arcs, triangles and planes may be sufficiently approximated by an offset surface [12, 15], based upon the same primitives. They are, respectively: spheres, cylinders, arc tubes, prisms and infinite slabs. The extrusion distance may be set to radius in isolation R or depth of influence D (see Figure 3.5, middle), according to the current modeling situation.

Such similarity suggests a convenient modeling strategy: first an object is approximated by an offset surface and then refined as a convolution surface. The important feature of this approach is that most of the hard modeling work, related to building the skeleton, may be done with easy-to-render offset surfaces. To support this similarity, we use the same syntax and internal data structures both for implicit primitives and their offset counterparts. For instance, the command line

```
sphere Plaster, r0, 0 0 0
```

creates an object in memory which may appear as an ordinary sphere, with radius `r0`, made of `Plaster` and positioned at the origin. Also, this object may appear as an implicit point with a radius in isolation `r0`. As described above, its blobbiness is stored in the description of the material `Plaster`.

Cylinders, arcs, triangles and planes also have reusable syntax and data structures. When the set of primitives is loaded into memory, it is ready for rendering in either mode: as an offset surface and as a convolution surface. In the former case, the implicit nature of all the modeling primitives is ignored and they are rendered as a union of offset surfaces. In the latter case, the primitives are treated as field functions $f_i(\mathbf{p})$ that form the composite scalar field of a convolution surface. To switch between two rendering modes, a simple on/off flag is used; the model does not have to be reloaded into memory.

Such duality of data structures has proven itself very useful. Computationally inexpensive offset drafts allow fine tuning of models at interactive rates. Since ray-tracing is used as a rendering method, all camera settings, lights and photometric properties of the objects are also adjusted in the draft mode as well.

3.3.6. Variables

For better interactivity in our design environment, the following data types are allowed to be defined as variables:

- scalars
- vectors
- colors
- textures
- materials

Variables may be created and modified 'on the fly' during design session; they also may be assigned to each other, observing type conversion. With the use of variables, it is possible to modify practically all elements of the model, while it is still in computer memory. They include: positions and shapes of all skeletal elements, blobbiness and thickness, all photometric characteristics. In addition, variables may be declared as auto-incremental and/or auto-multipliable, which helps specifying objects' motion in animations.

3.4. THE MAIN MODELING LOOP

3.4.1. Datasets

In the previous section, we mentioned elements of what we call *a dataset*, which is a complete collection of parameters that defines a particular geometric model. Every dataset contains descriptions of all materials and skeletal elements that together form a convolution surface. In addition, a dataset describes all modifications of local properties of skeletal elements that are needed in order to add a desired shape to the final surface. Thus, a dataset may be considered as a program in some highly specialized language for describing convolution surfaces. The complete vocabulary of this language and several complete datasets are given in Appendix C.

With respect to design and rendering, a dataset may be regarded as an output of a design process and an input for a rendering program.

3.4.2. The main modeling strategy

We use a method of progressive refinement of the dataset as the modeling strategy. An initial dataset is acquired first, using some foreign geometric model, a program generator or a user's inspiration with a paper-and-pencil method. Then, the dataset undergoes a series of iterations, during which a designer modifies existing elements of the dataset, adds new ones and deletes those that fail to fit the model.

The global dataflow chart of a typical design session is depicted in Figure 3.8. For this particular example, the skeletal model was produced by a tree-growing program, based upon Eric Haines' implementation [31] of Aono and Kunii's tree-generation method [2]. The rest of the dataset, full of question marks, has to be designed interactively.

By applying various values of thickness to the skeletal elements, several drafts have been made, pictured as a stack of images on the right-hand side of

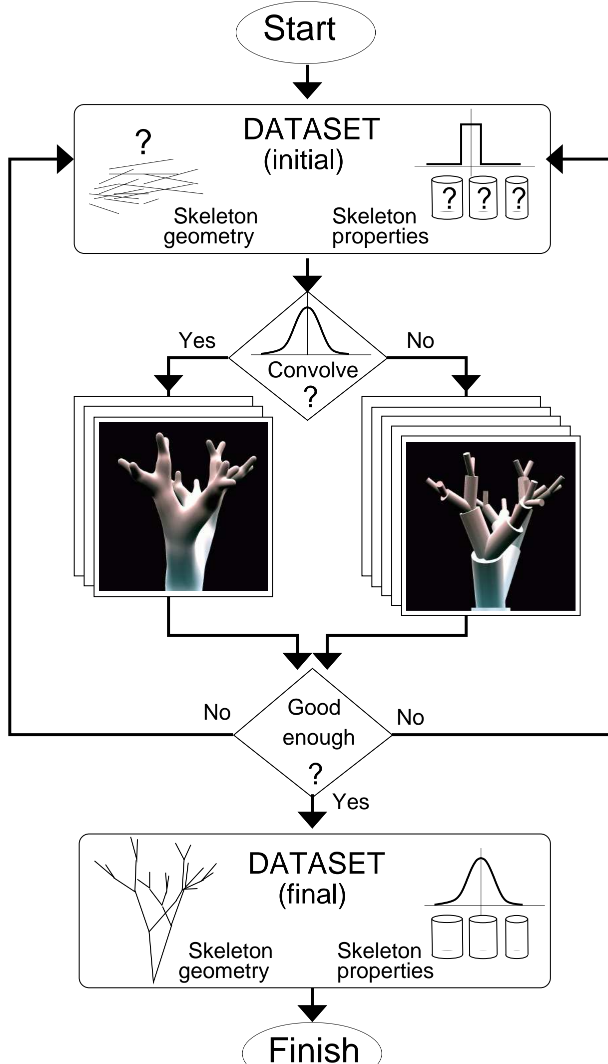


Figure 3.8: The main modeling loop.

the chart. Since offset surfaces, used for the drafts, are easy to render, the process may be easily re-iterated as many times as needed to find the perfect values for thickness of all branches.

At this point, the designer may switch to using convolution surfaces and try different values of blobbiness. After several attempts, the desired shape is found – the dataset is now complete and the process terminates [2].

Note that this modeling strategy is a winning strategy by construction: the process terminates when the designer is satisfied with the result. Visual control over the quality of the current state of the model guarantees that the design process moves in the right direction. Backup copies of the dataset help to fix irreparable changes.

The modeling process bears certain resemblance to the hill-climbing programming archetype: the process enters the loop at some point and then moves towards the solution, iterating on the current state of the task.

3.5. PRACTICAL EXAMPLES OF IMPLICIT DESIGN

In this section, a number of practical modeling examples are discussed. They are grouped by methods that have been used for their design. With respect to the main modeling loop, pictured in Figure 3.8, these methods are roughly divided into those that deal with the skeletons (right-hand side of the chart) and methods that refine the local properties of the convolved surfaces (left-hand side).

3.5.1. Global skeleton manipulations

The modeling system allows, in principle, us to build a new skeleton from scratch interactively. However, it is more convenient to prepare an initial skeleton, using one of the methods listed below.

3.5.1.1. Procedural methods

Two skeletons have been produced using purely procedural methods: the flat spiral shell in Figure 3.9 and the seaweed in Figure 3.13. Both are generated using modified utilities `shell` and `tree`, respectively, from Standard Procedural Database by Eric Haines [31]. Their skeletons are fairly simple. The `tree`-skeleton was also used in the model of a coral tree, shown in Figure 3.14. Additional 512 spikes were added to the model procedurally, using random placement.

[2]Or, the designer may also try to enhance the resulting shape by applying different profiling functions. The result of such experiments are shown in Figure 3.13.

3.5.1.2. Hand-crafting

A skeleton of a seahorse in Figure 3.15 is 100% hand-crafted. The initial paper-and-pencil drawing was created first. Then it was duplicated with the *xfig* drawing tool (Figure 3.15, left), which produced the coordinates of the skeletal elements, in this case, arcs. In most cases, the joints between the arcs in the line drawing are C^1 and C^2 continuous; the cracks in the offset surface (Figure 3.15, middle) are introduced intentionally to show the skeletal elements better. The final convolved surface (Figure 3.15, right) seals these cracks. Note, that the thickness of each skeletal element is set individually. The central arc in the body of the seahorse is made especially thick. The resulting inflation fills the empty space inside the body and, perhaps, indicates the presence of internal organs.

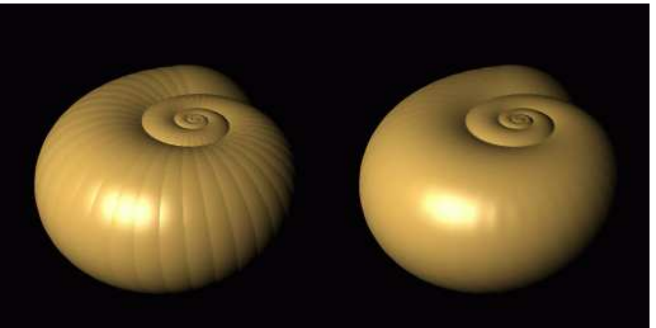


Figure 3.9: Shell I. The point-based offset surface (left) and the convolved surface (right) are very similar. This dataset served as a base model for Shell II and Shell III. 361 elements.

3.5.1.3. Mixed methods

Procedurally generated models may be significantly improved by manual editing. For instance, the oval seashell, pictured in Figure 3.10, is derived from the previous flat sphere-based model (Figure 3.9) by a simple addition of one point, which was then connected with all points in the initial dataset. Thus, all spheres become cylinders with a common base. By moving this common base, the shape of a new shell may be controlled easily. This is similar to rubber-band techniques with the wire-frame representation of solid object. Using a vector variable for holding coordinates of the common base point makes such manipulations especially convenient.

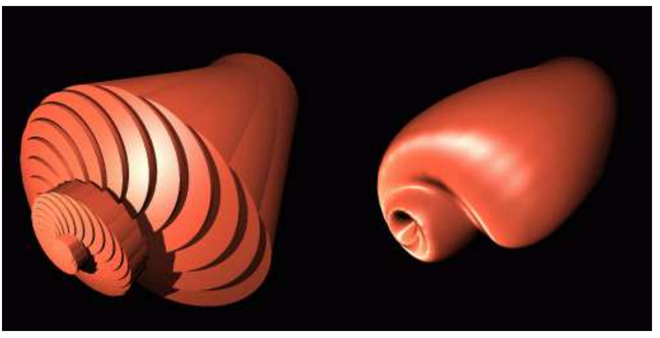


Figure 3.10: Shell II. The offset surface of this mollusk is pictured as a union of simple cylinders (instead of sphere-capped cylinders) to emphasize the spiral structure better. 42 elements.

The skeleton of the next shell was also created procedurally and then updated manually. This time, two skeletons were created as described above. Then they were superimposed to produce a combined skeleton for a spiked shell, shown in Figure 3.11.

While manual intervention often improves procedural skeletons, the reciprocal is also true: hand-crafted skeletons can be made much more interesting and visually appealing with the help of special purpose program generators. Examples of such skeletons are: a spindle cowrie shell (Figure 3.12) and coral crab (Figure 3.16). Both skeletons were hand-copied from the actual photographs [46], using *xfig* drawing tool. Then, small details were added procedurally.

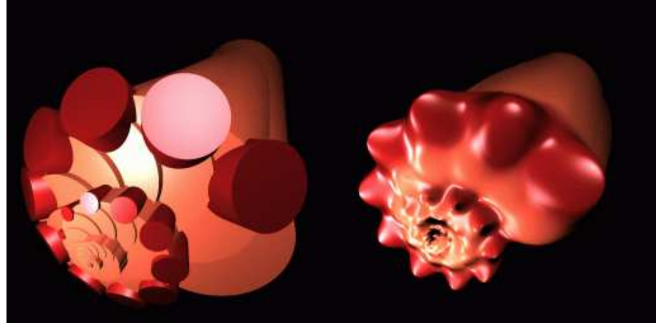


Figure 3.11: Shell III. Superposition of two skeletons: the orange base (55 segments) blends with the spiral row of red horns (15 segments) yielding a smooth convolved shape of a new seashell. Both skeletons are made procedurally.

Usually, skeletons of mixed origin yield the most interesting shapes.

3.5.1.4. Imported models

Two fully imported polygonal models of Yoda and Nefertiti are shown in Figure 3.19 and Figure 3.18, respectively. These models have been used as polygonal skeletons for the convolution surfaces without

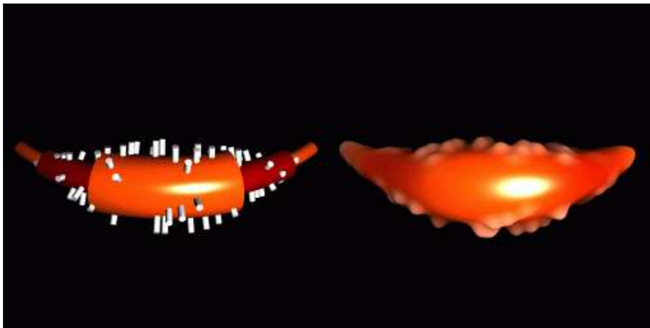


Figure 3.12: This very simple model of Spindle Cowrie is based on a croissant-like combination of 3 arcs. Additional 100 cylinders make the surface look more interesting.

Figure 3.13: Seaweed. The offset surface is made of cones and spheres. The conical shapes of the tree branches suggested applying profiling functions in the convolution surface.

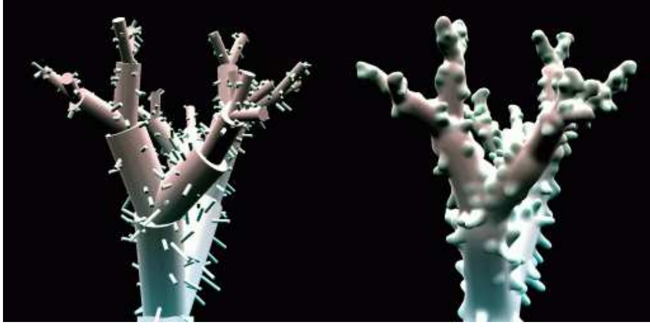


Figure 3.14: Coral trees. Note that the base skeleton (31 cylindrical branches) is identical to the seaweed model. Additional 512 cylindrical spikes of various lengths are scattered along the surface randomly.

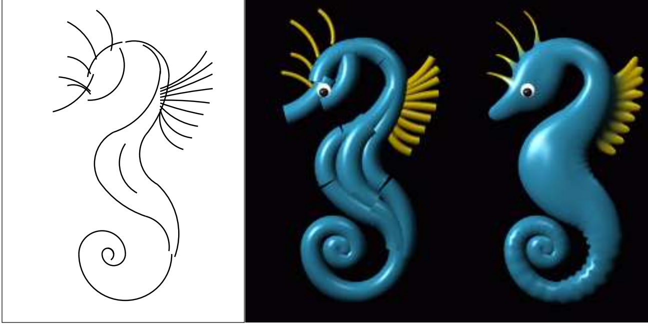


Figure 3.15: Seahorse. The hand drawing (left), the offset surface (middle) and the convolution surface (right). Note how the wrinkles in the back and the tail indicate softness of the skin. Total number of elements 45 (43 arcs and 2 spheres).

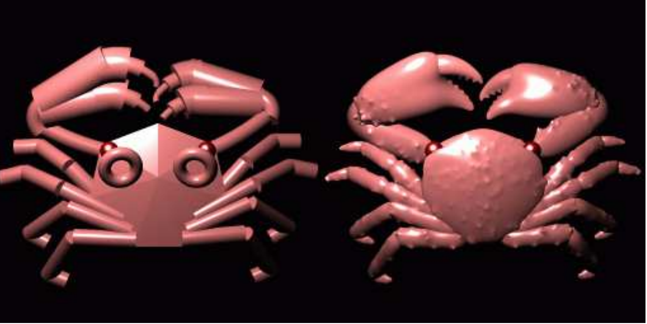


Figure 3.16: Coral crab. The simplified draft surface consists of 7 polygons (body), 26 arcs (claws, eye sockets), 28 cylinders (legs, claws) and 2 spheres (eyes). The convolved version has additional 608 spikes, scattered along the surface.

any modification done to their initial meshes. The origin of both models is unclear.

3.5.2. *Local modeling techniques*

After the global skeleton is complete, a designer may switch to more subtle modeling techniques and refine the model to their taste. With respect to the modeling loop (Figure 3.8), these operations happen on the left-hand side of the loop, because most techniques that are described below apply to convolved versions of the object.

3.5.2.1. *Shaping techniques*

Shaping is a straight-forward application of profiling functions, which creates local variations of radius in isolation of modeling primitives (or the thickness of the material). Examples of shaping are presented in Figure 3.8 and Figure 3.13, where normal tree is turned into a seaweed by applying a linear shaping function to its branches. Less obvious examples of shaping may be found in the model of a coral crab (Figure 3.16). The end segments of the crab's legs are also linearly shaped to make them look sharper.

3.5.2.2. *Carving techniques*

Carving is a combination of using depth of influence as a modeling parameter with some profiling functions. The depth of influence around modeling primitives defines a volume of 'matter' to carve from; the profiling function defines the shape of the 'cutting tool'.

Technically, profiling functions operate on each modeling primitive individually, as shown in Figure 3.7. However, when the modeling primitives interpenetrate, which is true for the spiral seashells (Figures 3.9, 3.10, 3.11), the result of carving is as if the cutting tool has been revolved in the model space following the spiral curve of the shell. In this way, profiling functions are similar to the *generating curves* used for modeling seashells in [27]. The carving technique was used with all the models of spiral shells, pictured in Figures 3.9, 3.10 and detailed in Table 3.1.

Fig.	**Primitives**	**Profiles**
3.9	361 points	constant
3.10	42 line segments	linear
3.11	70 line segments	$sin^2(x\pi)$

Table 3.1: Modeling of spiral seashells.

3.5.3. *Volumetric detail*

Defined as isosurfaces in a scalar field, convolution surfaces are sensitive to variations of that field. Large-scale variations cause global changes in the appearance of the object. Small-scale variations modify the geometry of the surface locally, adding more detail to the surface. Such detail enhances the appearance of the object and produce various texturing effects. Since these details are based upon implicit functions, which are volumetric by their nature, we call this *volumetric detail*.

Most models presented in the paper contain volumetric details, which may be grouped in three major ways: Structural, Functional and Procedural.

3.5.3.1. *Structural detail*

This method involves adding more modeling primitives, which may be done manually or by using some auxiliary utilities. Although it seems like a brute-force solution, structural detail may be very effective in modeling various irregular and rough-looking surfaces, such as the body, claws and pincer grip of a coral crab, shown in Figure 3.16 or a complex shape of a coral tree (Figure 3.13). Similarly, the shell of a spindle cowrie (Figure 3.12) is enhanced by cylindrical spikes. Other examples of using spikes with convolution surfaces are presented in [13, 15].

One convenient feature of structural details is that, unlike other methods of adding detail, most iterations on the dataset may be done with the offset representation of the surfaces. This corresponds to the right-hand side of the main modeling loop (Figure 3.8), where a designer may afford a large number of takes.

3.5.3.2. *Functional detail*

Functional detail is produced by applying profiling functions on small areas of the surfaces. As an example, consider wrinkles on the back and the tail of a seahorse (Figure 3.15).

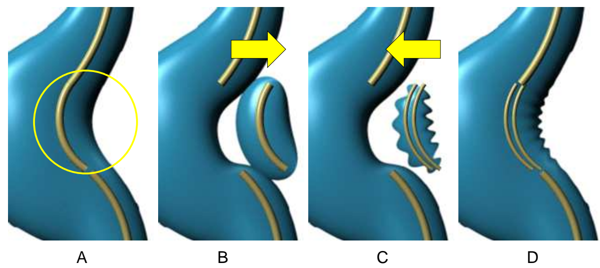


Figure 3.17: How to make volumetric wrinkles: (A) The place for wrinkles is chosen. (B) The closest arc is pulled out. (C) The arc is split into halves and one half is modulated by a sine wave. Together, they form a wrinkled volumetric implant, ready for re-insertion. (D) The wrinkled implant is put back in place.

Figure 3.17 illustrates how the wrinkles were implanted in the originally smooth convolution surface.

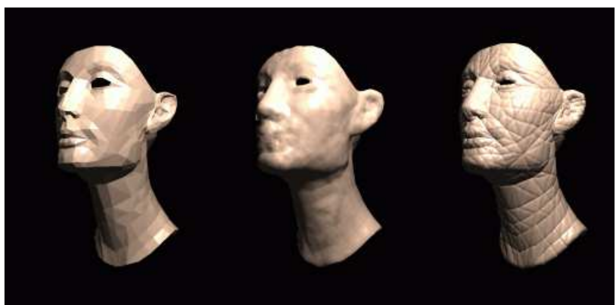


Figure 3.18: Nefertiti, before (left) and after (middle, right) convolution. Restricted blending near the edges creates the wrinkles (right). Total 1,242 implicit triangles.

This figure is an exact close-up of Figure 3.15 with more anatomical parts shown (only participating skeletal elements and close neighbors are displayed).

Note that the field function of the wrinkled volumetric implant, as described in Figure 3.15, could have been represented by a single arc, modified by an elevated and scaled sine wave $a_1(1 + a_2\sin(x))$, instead of two, as shown in the picture. In this case, it is a matter of choice whether to add another primitive into the dataset, or define one more profiling function in the modeling system.

A similar use of high-frequency functions for adding volumetric details is reported in [4]. The difference of our approach is that we use it in the skeletal-based modeling environment, which allows selective placement of such details.

3.5.3.3. Procedural detail

Perhaps, the most unusual examples of volumetric detail are given in Figure 3.18. and Figure 3.19. These images present the results of the experiments with triangular meshes used as skeletons for convolution surfaces. Triangular meshes are normally employed to approximate surfaces, as shown in Figure 3.18, left and Figure 3.19, top. Convolving triangles with a potential kernel function introduces remarkable changes into appearance of the object being modeled, as illustrated in Figure 3.18, middle. However, simple convolution makes the skin of Nefertiti look swollen. All fine features are missed, as high frequencies are removed by a low-pass convolution filter.

To reduce the swelling effect, a restricted mechanism is used, which artificially reduces the amount of field generated by triangular primitives near the edges. Thus, the surface exhibits hills over the central parts of each triangle and valleys along their edges. The maximal difference in elevation of these hills and valleys depends on the thickness of the material of the face (i.e., skin) and the value of the restriction factor. The former tends to elevate the convolution surface

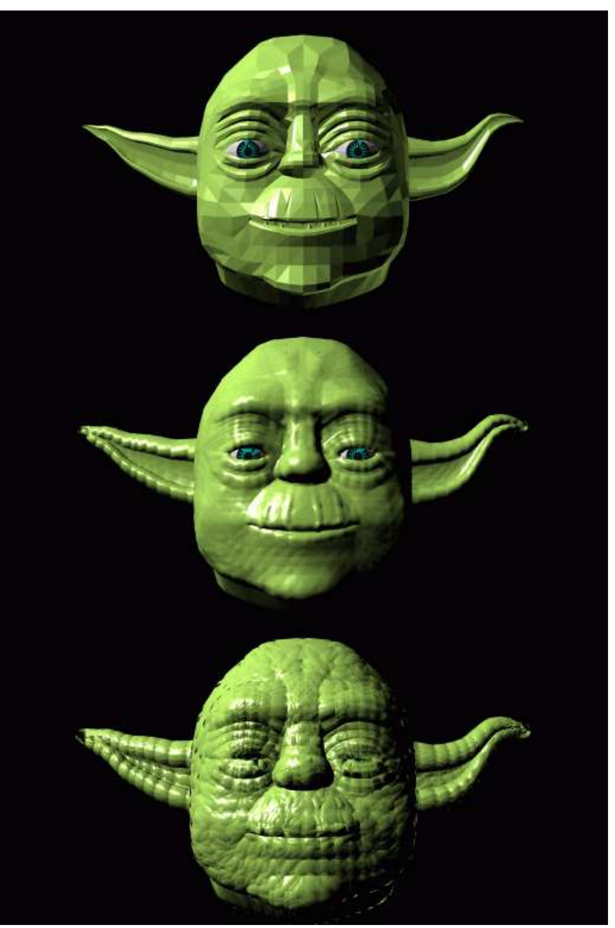


Figure 3.19: Aging Yoda: polygonal mesh (top), convolution surfaces (middle and bottom). Total 3,422 implicit triangles.

Model	**Skeleton**	**Techniques**
Shell I	Procedural	Carving
Shell II	Procedural + hand-made	Carving + + shaping
Shell III	Procedural + hand-made	Carving + + shaping
Seaweed	Procedural	Shaping
Seahorse	Hand-made	Functional wrinkles
Cowrie	Hand-made + procedural	Structural detail
Crab	Hand-made + procedural	Structural detail + + shaping
Nefertiti	Imported	Procedural wrinkles
Yoda	Imported	Procedural wrinkles

Table 3.2: Summary of modeling techniques used with all datasets.

above the 'bare' polygonal skeleton, especially in the central areas of each polygon. The latter pulls the surface closer to edges of triangles, producing wrinkles along the edges. The roughness of the resulting implicit surface may be effectively controlled by varying both parameters: thickness and restriction scaling factor.

The visible outcome of such controlled convolution is the perceived age of the character. The wrinkled and dried-up skin makes the face of Nefertiti arguably more interesting and definitely more realistic — the prototype of this model is 5,500 years old. Similarly, convolved skin gives Yoda a much more mature look.

The restriction mechanism may be combined with an additional UV-mapping to produce even more wrinkles inside each triangle. Alternatively, procedural or noisy functions may be used to modify the values of the field functions. In all cases, the visible complexity of the resulting convolution surface will be increased without increasing the number of elements in the skeleton.

Note, that all types of volumetric detail presented above, have truly geometric nature; they are not shading tricks. They participate in a full range of graphics manipulations, such as visibility calculations, shadows, transformations, etc. By modifying the local geometry of the surface, volumetric detail creates additional features that may be very informative. For instance, spikes and horns imply firm and rigid surfaces; wrinkles often suggest that the object is flexible, allows deformations, and is currently deformed from its usual state; wrinkled skin suggests old age. These details enhance the visual realism of the model and have a strong potential for complex modeling with implicit surfaces.

3.6. IMPLEMENTATION DETAILS AND TIMING RESULTS

All models discussed above were designed and ray-traced using the in-house modeling/rendering system *RATS Version 7.31* running under the Linux OS on a 90 MHz Pentium processor. The algorithm for ray-tracing convolution surfaces, used in the *RATS* system, is described in Chapter 4. Offset surfaces are also ray-traced; the relevant algorithms may be found in [29] and [71]. Table 3.3 gives details about each dataset and the rendering times. Image resolution: 512 x 512. Anti-aliasing method: shooting at most 16 primary rays per pixel (Of course, during interactive design sessions, much smaller images were used, typically 128 x 128. Shooting 1 ray per pixel allowed to render all offset surfaces under 10 seconds per iteration.)

Model	**Contains**		**Off. surf.** t_1	**Conv. surf.** t_2	t_2/t_1
Shell I	361	points	2:52	22:46	7.81
Shell II	42	lines	20:45	32:51	1.58
Shell III	70	lines	12:54	41:16	3.20
Cowrie	103	total	7:34	24:40	3.26
Seaweed	31	lines	3:34	4:46	1.34
Coral	543	lines	11:18	38:50	3.43
Seahorse	43	arcs	5:25	11:52	2.19
Crab	641	total	14:03	46:13	3.29

Table 3.3: Rendering time (min:sec) for offset surfaces and convolution surfaces.

These timing data give a somewhat distorted picture of the actual situation. For instance, some offset surfaces contain a large number of edges, that had to be anti-aliased by shooting more rays. Convolved shapes tend to be smoother, thus fewer pixels had to be supersampled. This explains an unusually low time increase of 1.58 for rendering the second seashell as a convolution surface: it has a very sharp offset-surface counterpart.

The seahorse dataset presents another 'special case' of low time ratio. Since arc tubes, used as offset surfaces, are already computationally expensive (they require solving quartic polynomials for each intersection test), switching to convolution surfaces does not introduce dramatic changes in rendering time.

The fastest implicit primitive, according to Table 3.3, is a line segment (see the seaweed model in Figure 3.8). It required only 34% more rendering time than its explicit counterpart. Such efficiency is due to a better kernel that was used for this particular dataset, that is $h(x) = b\ exp(-a^2x^2)$. The field function for line segments, produced with this kernel, turned out to be very efficient for this particular

model [3]. Exponential kernel function was first used by Blinn [7].

The average time penalty for using convolution surfaces compared to offset surfaces is about 3.2. The crab model, which contains all types of new implicit primitives, shows a similar slow-down of 3.29.

3.7. CONCLUSION

In this chapter, we have presented a variety of primitives and modeling techniques for implicit design. Points, lines, arcs and triangles, when used as elements of convolution surfaces, spawn a rich palette of unusual variations of this modeling concept. Methods of adding details are discussed, which enhance the visual realism of the surfaces and expand the application base of implicit modeling.

We would like to point out that most images presented in this chapter could not have been produced by conventional techniques developed for implicit modeling to date. These images are products of the concept of convolution surfaces and a number of additional modeling methods described above, and rendering methods discussed in the next chapter.

A certain boy once drew a beautiful, if somewhat abstract picture. He was asked: "And what did you mean to say by this?" To which he replied, "I meant exactly what I said."

A boy (perhaps the same one) was taken to a museum and shown an abstract picture. The guide explained: "This is meant to be a horse." To which the boy replied, "If it is meant to be a horse, why isn't it a horse?"

–*Two complementary stories*

[3]Note, that the field function for a line segment produced with a Gaussian kernel looses to the Cauchy line segment in the 'bare' speed test, as shown in Tables (2.3, 2.4) (see Chapter 2). However, in different modeling situations with different values of blobbiness and radius in isolation, the Gaussian line segment may be faster (for instance, due to tighter bounding volumes), as it apparently happened in this case.

Chapter 4

Rendering Convolution Surfaces

4.1. INTRODUCTION

In this chapter, a ray-tracing algorithm is described for rendering implicit surfaces formed with C^1 - continuous bounded functions $f(\mathbf{r})$. This class of functions includes such popular implicit models as blobby molecules, metaballs, soft objects and convolution surfaces. The algorithm employs analytical methods only. It is fast, robust, and numerically stable.

4.1.1. The problem

An implicit surface is defined as $S = \{\mathbf{r} \mid F(\mathbf{r}) = 0\}$ where F is a scalar function $F: R^3 \to R$, defined over all points in 3D space analytically, procedurally, or with elements of both. Such functions are often called field functions or implicit functions. An equation

$$F(\mathbf{r}) = 0 \tag{4.1}$$

is referred to as an implicit surface equation. The generic form of equation (4.1) makes implicit surfaces one of the most powerful and flexible modeling tools available.

Ray-tracing [25] allows implicit surfaces to be visualized directly from their models as defined by equation (4.1), without tessellating it into curves or polygons. The basic operation of ray-tracing is finding ray/surface intersections. Representing the ray parametrically as

$$\mathbf{r}(t) = \mathbf{a} + t\mathbf{b} \tag{4.2}$$

($\mathbf{a}$ is the ray's origin and $\mathbf{b}$ is its direction), an implicit surface equation for all points on a given ray becomes

$$F(\mathbf{r}) = f(\mathbf{a}, \mathbf{b}, t) = 0 \tag{4.3}$$

or simply

$$f(t) = 0 \tag{4.4}$$

Equations (4.3) and (4.4) must be formulated and solved for millions of rays $(\mathbf{a}, \mathbf{b})$ which poses very heavy computational demands on the rendering system. Few functions (4.1) currently used in implicit modeling yield closed-form solutions of (4.4), which necessitates the use of numerical methods and makes the problem even more difficult.

4.1.2. Previous work

There is a multitude of algorithms for ray-tracing implicit surfaces developed to date. A proper classification and evaluation of these algorithms deserves a separate work. Overviews of most general methods for ray-tracing implicit surfaces are given in [15] and [32]; a number of numerical methods are also described in [44].

With respect to the main implicit equation (4.1), all algorithms demonstrate various degrees of reliability, speed and generality. Very often, these characteristics are mutually exclusive. The following two algorithms, *ray-marching* and *LG-surfaces*, represent, perhaps, the most extreme approaches in ray-tracing implicit surfaces.

Ray-marching. This is a brute-force method that steps along the ray, evaluating the field function $f(t)$ on each step. The surface is detected when the sign of $f(t)$ first changes. The ray-marching algorithm treats field functions $f(t)$ as true 'black boxes' and makes no assumptions about their properties. Ray-marching is one of the most general algorithms which can render anything, given enough time. The robustness is achieved by setting the incremental step low, which makes the algorithm extremely time-consuming.

Ray-marching was introduced by Tuy and Tuy [70] for direct visualization of medical data. Perlin and Hoffert [54] used ray-marching to render remarkably complex and realistically looking objects, such as fur, fire and eroded metal, modeled with very noisy functions.

LG-surfaces. Kalra and Barr [38] developed an algorithm guaranteed to detect the surface, modeled by functions with computable L and G parameters, that represent the Lipschitz constants for the function f and its derivative df/dt along the ray. A Lipschitz constant λ is defined for a scalar function f over region A as

$$|f(\mathbf{x}) - f(\mathbf{y})| < \lambda ||\mathbf{x} - \mathbf{y}||, \quad \mathbf{x} \in A, \mathbf{y} \in A$$

A Lipschitz constant L, computed for a function f, provides a means to find regions of space where f is

guaranteed not to intersect the surface. A Lipschitz constant G, computed for the directional derivative df/dt along a given ray, allows the finding of the intervals of monotonicity of $f(t)$ and, therefore, the isolation of all roots of $f(t) = 0$ reliably. The roots are then refined with any well-established method such as regula falsi [57].

In order to be effective, the *LG-surface* algorithm requires run-time computations of L and G for different regions of space and intervals along each ray. To obtain the value of L over a region of space, the gradient ∇f is computed and its magnitude is maximized over the region. For G, the similar computations must be carried out with df/dt: the second derivative d^2f/dt^2 is computed and maximized over the interval along the ray. For non-algebraic modeling functions f, these computations may become prohibitively difficult, even if L and G are derived in symbolic form.

There are less demanding algorithms that guarantee to find ray/surface intersections without evaluating second derivatives. For the *sphere-tracing* algorithm, described by Hart [34], even the first derivatives need not be defined. Sphere-tracing avoids the problem of isolating roots and converges on the surface from one side, using a Lipschitz constant to compute the signed distance to the surface. Sphere-tracing can render a wide class of surfaces, including fractal, rough and creased ones.

Ray-tracing with interval analysis, introduced by Mitchell [44], requires run-time evaluation of the first derivative df/dt to isolate the roots.

It is important to note that all algorithms that bound the rate of change of the implicit functions $f(t)$, either with a Lipschitz constant ([34] and [38]) or with derivatives df/dt ([44]) work better when these bounds are as tight as possible. Therefore, they must be computed at run-time for each ray individually and for each interval along this ray, which may not be an easy task to accomplish for complex functions f. The use of global precomputed values will degrade the efficiency and ultimately will turn the root-isolating algorithms ([38] and [44]) into a simple bisection, and the sphere-tracing algorithm into ray-marching.

To summarize: the most general algorithms with the widest application base ([54] and [70]) are very slow and do not guarantee to locate the surface. On the other hand, reliable methods require auxiliary computations that may become too expensive for complex modeling functions f.

Next, we provide an overview of the algorithm for ray-tracing implicit surfaces that combines generality, reliability and efficiency.

4.1.3. Algorithm preconditions

The ray-tracing algorithm presented in this chapter has been designed to render implicit surfaces modeled with the following equation:

$$F(\mathbf{r}) = -T + \sum_{i=1}^{N} f_i(\mathbf{r}) \qquad (4.5)$$

where T is the isopotential value and each constituent function $f_i(\mathbf{r})$, when parameterized along an arbitrary ray (4.2) as $f(t)$, satisfies the following conditions:

1. $f(t)$ is C^1-continuous for all t,

2. $f(t)$ and $df(t)/dt$ only have non-zero values over a finite set of non-overlapping intervals $[t_{1_i}, t_{2_i}]$, $i = 1, ..., k$.

The second condition implies that each object represented by its function f can be enclosed by a bounding volume (of not necessarily finite size). For example, an infinite implicit plane may be enclosed in a co-planar infinite slab.

In general, the algorithm requires that both $f(t)$ and $f'(t)$ be defined. In special cases when f is symmetric about the midpoint of each interval $[t_1, t_2]$, the derivative $f'(t)$ is not required. Examples are: implicit points, lines, tori.

The class of modeling functions that meets the said conditions (and, therefore, can be rendered by our algorithm) is very wide and includes such well-known models such as blobby molecules [7], metaballs [50], soft objects [73] and convolution surfaces [12].

4.1.4. Algorithm postconditions

The implicit surfaces rendered by the algorithm will exhibit:

1. *Smoothness*
   The algorithm will render the object as a C^1-continuous surface. The surface is guaranteed not to contain pixel dropouts and shading will be smooth along the surface.

2. *Fine features*
   The algorithm will render fine features of the implicit surface with the same fidelity as for conventional, non-implicit primitives. The algorithm will miss the implicit surface if its modeling function $f(\mathbf{r})$ has a bounding volume so small that it slips between the sampling rays. This is the general problem of point-sampling methods that is well understood and may be solved using stochastic or oversampling methods [25].

3. *Limited accuracy*
The rendered surface may slightly deviate from its true location as implied by the modeling equation (4.5). The error is individual for each field function f_i and is not cumulative.

The rest of the chapter is organized as follows. The next section provides a detailed description of the algorithm. Section 4.3 provides a discussion of errors. Implementation issues and speed-up techniques are given in Section 4.4 and illustrated by practical examples in Section 4.5. Some possible improvements are suggested in Section 4.6.

4.2. THE ALGORITHM

The algorithm is based on the original work by Nishimura et al., who developed a very efficient, though highly specialized, algorithm for ray-tracing *metaballs* [50]. We first describe Nishimura's algorithm and then show how it can be extended to render implicit functions that meet our preconditions.

4.2.1. Ray-tracing algorithm for metaballs

In the *metaball* model, the constituents f_i of the isosurface equation (4.5) are point potentials represented by piecewise quadratics:

$$f(\mathbf{r}) = \left[\begin{array}{ll} 1 - 3(\frac{r}{R})^2 & 0 \le r \le \frac{R}{3}; \\ \frac{3}{2}(1 - \frac{r}{R})^2 & \frac{R}{3} < r \le R; \\ 0 & r > R; \end{array} \right. \tag{4.6}$$

where R is radius of influence of the metaball and r is the distance from its center to point $\mathbf{r}$.

To find the ray/surface intersections, the ray equation (4.2) is substituted into the potential function (4.6), yielding at most three piecewise quadratic polynomials per metaball that describe its field along the ray. When all metaballs have been processed in this manner, the whole extent of the ray inside their collective field becomes sliced into a set of intervals with corresponding polynomials derived from equation (4.6). The algorithm walks through these intervals, building and solving the isosurface equation (4.5). Since all components are represented by quadratic polynomials, the collective isosurface equation is also a quadratic, and all roots (hence intersections) can be found analytically.

A similar technique is described by Wyvill and Trotman [79]. They modeled point sources by pieces of polynomials of degree 6, and solved the implicit equations for roots using Laguerre's method.

4.2.2. Generalization of the algorithm for metaballs

The nature of the algorithm described above is that it may be applied to solve isosurface equations generated by *any field function*, provided it can be represented as a sum of polynomials with ray distance t as argument. For metaballs, this representation is straightforward, because the modeling functions (4.6) are already specified in polynomial form. For an arbitrary function f, additional processing may be required in order to provide this representation. This can be achieved using polynomial approximation.

Weierstrass's theorem states that if f is a continuous function on the interval $[t_1, t_2]$, then for any $\epsilon > 0$ there exists a polynomial p such that

$$max|f(t) - p(t)| < \epsilon, \qquad t_1 \le t \le t_2 \tag{4.7}$$

which simply means that any continuous function f may be approximated on a closed interval by a polynomial p as closely as required.

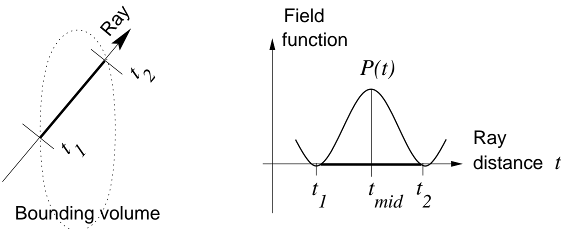


Figure 4.1: Polynomial approximation over $[t_1, t_2]$.

In our case, the interval $[t_1, t_2]$ is obtained via intersecting the ray with the bounding volume of the function f (see Figure 4.1). Finding the right approximating polynomial is a matter of balancing between the following constraints:

- The degree of the approximating polynomials $p(t)$ must be at most 4, so that analytical methods can be used to solve for roots t [59].
- For piecewise polynomials, the representation must be C^1-continuous to guarantee a smooth shading of the surface.
- The number of evaluations of the function $f(t)$ required to compute the polynomial representation, should be as low as possible to keep the overall performance high.

Interpolation using Hermite polynomials provides a feasible solution to this problem. The Hermite interpolant of a function f is the polynomial p of least degree for which

$$\begin{aligned} p(x_i) &= f_i := f(x_i), \\ p'(x_i) &= f'_i := f'(x_i), \end{aligned} \tag{4.8}$$

at given node points $x_i, \quad i = 1, ..., n$.

For our purposes, the Hermite interpolants can be computed as follows. The initial interval $[t_1, t_2]$, obtained via intersecting the ray with the bounding volume, is divided at the midpoint t_{mid} (see Figure 4.1). For endpoints of the first sub-interval $[t_1, t_{mid}]$ conditions (4.8) are written as

$$\begin{aligned} f_1 &= 0, \\ f_1' &= 0, \\ f_{mid} &= f(t_{mid}), \\ f_{mid}' &= f'(t_{mid}) \end{aligned} \tag{4.9}$$

because the function f is known to have zero values and zero derivatives at the boundaries of its region of influence. If the function f is symmetric about the midpoint t_{mid}, the second derivative condition is reduced to

$$f_{mid}' = 0$$

Equations (4.9) solved for sub-intervals $[t_1, t_{mid}]$ and $[t_{mid}, t_2]$ produce a piecewise representation of the function f over interval $[t_1, t_2]$ that satisfy the requirements listed above: it is of low-degree, C^1-continuous and computationally efficient.

4.2.3. An example

To demonstrate, consider the simple example of finding all intersections between a ray and a surface modeled with three implicit line segments and one point potential (Figure 4.2 A). We assume that the functions $f_{point}(t)$ and $f_{line}(t)$ are defined at any point t on the ray. The exact expressions for the field function $f_{point}(t)$ are given in [7], [50] and [73]; implicit functions for a line segment may be found in [42]. All these functions are also given in Chapter 2.

First, the ray is intersected with the bounding volumes of all modeling functions f_i (Figure 4.2A). The resulting intervals I_1 and I_2 define the geometric location along the ray where the field is considered nonzero. Next, for each interval I_i, the corresponding field function f_i is interpolated by polynomials p_i (in this example shown as quartics without loss of generality) (Figure 4.2B). Finally, all intervals I_i are intersected and sorted along the ray, yielding a sequence i_1, i_2, i_3 (Figure 4.2B). At this point, the algorithm is ready to proceed with the root-finding. The isosurface equations are built and solved for roots t in all intervals, as demonstrated in Table 4.1.

In general, the number of roots per interval may be as high as the highest degree of all interpolating polynomials $p_i(t)$ defined over this interval. These roots may also occur outside the interval. Therefore, for each interval, the algorithm validates all roots by checking if they belong to the interval. In our example, the only valid roots are marked by circles in Figure 4.2(B and C). The corresponding points give

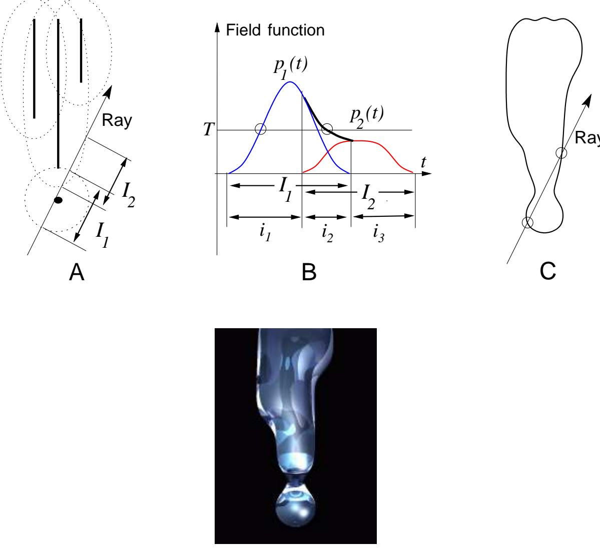


Figure 4.2: Rendering an implicit icicle: (A) intersection with bounding volumes; (B) interpolation of contributing components by polynomials $p_i(t)$; (C) isosurface at level T; (D) shaded image.

Interval	Field equation	Valid roots
i_1	$p_1(t) = T$	1
i_2	$p_1(t) + p_2(t) = T$	1
i_3	$p_1(t) = T$	0

Table 4.1: Ray-surface equations along the ray's path.

the location of the isosurface at level T (Figure 4.2C). The shaded image is shown in Figure 4.2D.

4.2.4. Bounding volumes

No one can embrace the boundless.

–Koz'ma Prutkov
'Thoughts and Aphorisms", 1854

This observation is a reminder that the algorithm presented above will work only with those primitives that can be enclosed into a bounding volume. Table 4.2 shows the bounding volumes for all implicit primitives, developed in Chapter 2.

Primitive	Bounding volume
Point	1 sphere
Line segment	1 cylinder + 2 spheres
Arc	1 piece of torus + 2 spheres
Triangle	3 cylinders + 3 spheres + 1 prism
Plane	1 infinite slab

Table 4.2: Bounding volumes for modeling primitives.

4.2.5. Shading and texturing

The normal vector $\mathbf{n}$ at the ray/surface intersection point $\mathbf{x}$ is

$$\mathbf{n} = -\sum_{i=1}^{N} w_i \frac{\nabla f_i(\mathbf{x})}{||\nabla f_i(\mathbf{x})||} \tag{4.10}$$

where the scalar weights, obtained as $w_i = f_i(\mathbf{x})$, are normalized to sum up to the threshold value T (which equals 1 by convention):

$$\sum_{i=1}^{N} w_i = T \tag{4.11}$$

Scalar weights w_i are also used to interpolate between photometric characteristics C of materials associated with the modeling primitives f_i:

$$C = \sum_{i=1}^{N} w_i C_i \tag{4.12}$$

The choice of parameters C depends on the lighting model. They usually include ambient, diffuse and specular colors, transparency, reflectivity and other data.

The normalization of w_i is necessary because of the approximate nature of the algorithm. The actual value of the field at the intersection point $\mathbf{x}$, as found by the algorithm, may slightly differ from T. Thus, the weights w_i must be scaled to ensure that all photometric parameters and normal vector components blend correctly [1].

Surface texturing, both flat and solid, can also be performed. Applying solid textures is straightforward and depends on the location of the surface only. Flat textures are more difficult, because local texture coordinates (U, V) must be computed in the local space of all constituents f_i then combined together and re-mapped onto the resulting surface. This re-mapping is not easy and is addressed in [53] and [76].

We used the simple fact that each function f_i is accompanied by its own bounding volume, usually a simple geometric object. Therefore, the UV-values for the bounding volumes are obtained first and then used to texture the surface parameters C for each participating f_i separately. The new textured values C_i are finally mixed as defined by equation (4.12).

4.3. ERROR ANALYSIS

As with any technique that involves approximation, it is important to obtain bounds for the errors. In the case of Hermite interpolation on n nodes $x_1, x_2, ..., x_n$ the error is

$$\epsilon(t) = \frac{f^{(2n)}(\xi)}{(2n)!}[L_n(t)]^2 \tag{4.13}$$

where $L_n(t) = \prod_{i=1}^{n}(t - x_i)$ and point ξ belongs to an interval containing all x_i and the point of interest t. Here x_i denotes the location of an interpolation node and is not to be confused with the endpoints of the initial interval $[t_1, t_2]$. The derivation of formula (4.13) may be found in [17].

It is apparent from formula (4.13) that the error may be reduced by increasing the number of node points n within the initial interval $[t_1, t_2]$ (see Figure 4.1). Alternatively, one may fix the number of node points n and split the initial interval $[t_1, t_2]$ into smaller sub-intervals, reducing the value of the L_n term. Both methods reduce the amount of error very efficiently. However, increasing the number of node points n yields interpolating polynomials of higher degrees $(2n - 1)$, which disables the use of analytical root-solvers for $n > 2$.

Thus, we have chosen to stay with cubic interpolants $(n = 2)$ and increase the number of sub-

[1] Obviously the normal vector may be obtained without computing the weights w_i. In the current implementation of the algorithm, however, it is more convenient to compute the magnitude and direction of the normal vector separately for all components f_i of the iso-surface.

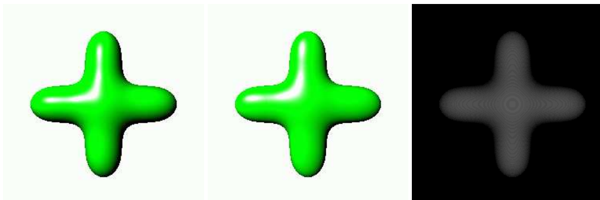


Figure 4.3: The front view. The error is uniform over the image. Two Hermite interpolants are used per line segment. The error is noticeable but insignificant. Rendering time: 24 sec (numerical, left) and 13 sec (analytical, middle).

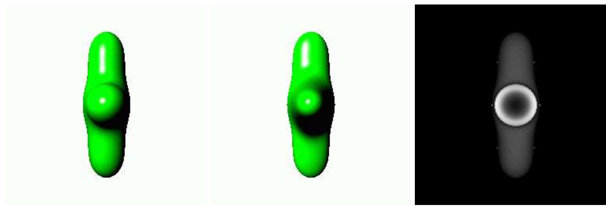


Figure 4.4: The side view. The error is substantially higher in the central areas, where rays travel along the horizontal line segment, yielding longer extents $[t_1, t_2]$. The surface becomes deformed. Rendering time: 20 sec (numerical) and 13 sec (analytical).

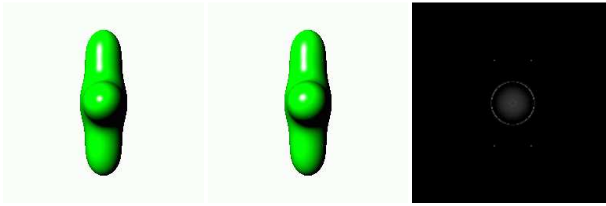


Figure 4.5: The side view, revisited. The error is reduced by using four sub-intervals and four Hermite interpolants per line segment. The left and middle images are practically identical. Rendering time: 20 sec and 17 sec.

intervals instead. On each sub-interval $[x_1, x_2]$, the error is

$$\epsilon(t) = \frac{f^{(4)}(\xi)}{24}[(t - x_1)(t - x_2)]^2 \tag{4.14}$$

Since the error is a quartic function of $|x_1 - x_2|$, doubling the number of sub-intervals over the initial extent $[t_1, t_2]$ decreases the error by the factor of 16.

To see the error-control mechanism in practice, consider a following example. A cross is modeled with two implicit line segments and rendered several times in different views. In Figures (4.3 – 4.5), the leftmost images are rendered with numerical root-finding techniques, using true function evaluations. The middle images are rendered using analytical root-solver and Hermite interpolation. The rightmost images show the rendering error, measured for each pixel as the distance between surface locations as computed by numerical and analytical methods.

4.4. OPTIMIZATIONS

The algorithm may be optimized for a particular rendering task. Here are some examples.

4.4.1. Polynomization on demand

If constructive solid geometry (CSG) is not required, the algorithm does not require computation of *all* polynomials along the ray – it suffices to prepare the intervals $i_1, ..., i_n$ only (see Figure 4.2B). The corresponding interpolants p_i will be computed on demand, as the algorithm goes through the list of intervals. The process terminates after the first intersection is found. This simple technique may reduce the number of calls to the interpolation routine significantly. In the example given in Figure 4.2B, the second field function, defined over I_2, need not be interpolated.

4.4.2. Fast ray/surface rejection test

Before attempting the intersection test, some preliminary interrogation of the modeling functions may help to speed up the intersection test. For example, if the total sum of the maximum contributions from all constituents f_i is still less than the isopotential value T, the isosurface equation $\sum_i^n f_i = T$ will have no roots and there will be no intersections in the whole sequence of intervals $i_1, ..., i_n$. This test is especially effective if the modeling functions f_i are symmetric about the midpoint of their intervals (which is true for implicit points, lines, tori and planes), where the field reaches its maximum value.

A similar root-exclusion test may be used locally for some intervals $i_j = [x_1, x_2]$, where the field function f is known to be monotonic. If T is not contained

between $f(x_1)$ and $f(x_2)$, there will be no intersection in i_j, and the algorithm may move on to the next i_{j+1} interval, skipping the interpolation and root-solving stages. To determine if the function $f = \sum_i^n f_i$ is monotonic, one must check that all its constituents f_i are consistently non-decreasing (or non-increasing) over that interval.

4.4.3. Volatile and permanent clusters

The modeling functions f_i that require blending are normally organized in linked lists, or *fusion clusters*, using the terminology of Nishimura et al. [50]. The way these clusters are created may influence the efficiency of the algorithm. The following two types of clusters have been implemented and compared: permanent and volatile.

The first method involves a preprocessing stage, during which all the field functions f_i in the database are arranged into permanent lists. To create these lists, a connectivity graph is built whose vertices represent all modeling functions f_i and whose edges are set between objects with intersecting bounding boxes. A depth-first search for fully connected components in this graph extracts all permanent clusters, including degenerate ones for isolated functions. These clusters are permanent because they remain unchanged during the ray-tracing of the whole image. This process is illustrated in Figure 4.6.

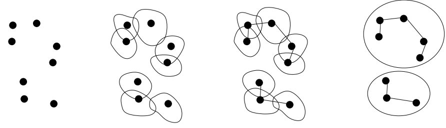


Figure 4.6: Creating permanent clusters: originally chaotic implicit primitives (left) are organized into a graph using their bounding volumes (middle left) which is then searched for connected components (middle right). The connected components form two permanent clusters (right). Black dots show the locations of the primitives, lines indicate their bounding volumes.

This method is quite suitable for rendering still pictures and animation sequences where modeling functions f_i do not change parameters that might affect their regions of influence, and hence the connectivity graph. In the general case, clusters must be decomposed into their constituents and rebuilt again for every new frame. Use of permanent clusters gives better rendering times for simple models, such as in Figure 4.2.

Alternatively, fusion clusters may be created 'on the fly' for each ray. When the ray encounters an implicit function (i.e., intersects its bounding volume), the function is linked to a temporary list. When the traversing of the whole database finishes, this temporary cluster contains all functions f_i that may contribute to the field along the ray. The cluster is tested for intersections and then destroyed.

Practice shows that volatile clusters are best for rendering complex scenes. Volatile clusters are even more efficient when dynamic memory allocations are replaced by the use of static pools. It is important to note that, regardless of the type of clusters used, the implicit equations formulated for each ray are the same. Choosing between volatile and permanent clusters only changes the way the memory is organized and addressed.

4.4.4. Reusing interpolants

When shading an intersection point, the algorithm re-evaluates all contributing functions f_i at that point to compute the weights w_i (equations (4.10) and (4.12)). The speed may be improved significantly by re-using the polynomial representations for each f_i that were obtained during the ray/surface intersection test. Hermite interpolants $p_i(t)$ of degree 3 are evaluated at a flat rate of 3 multiplications and 3 additions; true values of field functions $f_i(t)$ may cost as much as dozens of floating point operations and may contain calls to special functions too. (See Chapter 2 for examples of very complex field functions).

Table 4.3 provides the actual timing results for the model of a coral tree (Figure 4.10, right), rendered with various optimizations.

Optimization method	**Time** (min:sec)	**Speed-up** (%)
None	34 : 37	—
Interpolation on demand	33 : 10	4.2
Fast rejection test	31 : 13	9.8
Static memory pools	31 : 31	8.9
Reusing interpolants	34 : 07	1.4
All of the above	28 : 30	17.7

Table 4.3: Optimization methods and rendering times.

4.5. EXAMPLES

The ray-tracing algorithm described in this chapter has been implemented as part of the integrated environment for modeling, rendering and animating implicit surfaces *RATS* Version 7.31. The modeling techniques that can be used with the *RATS* system are described in great detail in Chapter 3. The complete command language set and the list of all features

of this system is given in Appendix C. The following images illustrate the algorithm.

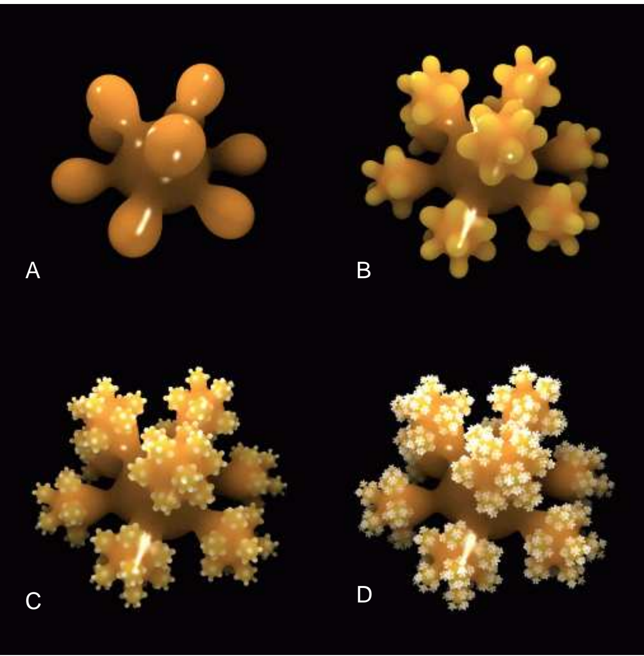


Figure 4.7: Implicit sphere-flakes made with 10, 91, 820 and 7381 Gaussian density functions. The smaller objects have lighter surface colors for better contrast.

The well-known sphere-flake model of Eric Haines [31], was re-modeled by replacing the original spheres with Blinn's blobs $f(r) = b\ e^{-ar^2}$, where the scalar parameters a and b were derived from the radius in isolation for each layer of elements and their blobbiness that controls the blending (see [7] for more detail). To check the speed of our algorithm, each image in Figure 4.7 was rendered twice: first with the algorithm presented in this chapter and then with the LG-based algorithm as described by Kalra and Barr [38]. To make a fair comparison, both algorithms were implemented and tested in the identical environment. Special care was taken to implement both methods with the same level of optimization, i.e. obvious things such as multiple function evaluations were carefully avoided. Running times are given in Table 4.4 (frame size 512 x 512, supersampling with at most 16 rays per pixel).

The pseudo-color chart shows the relative amount of time spent rendering the image in Figure 4.7C with our algorithm (left) and the LG-algorithm (right). As expected, silhouette edges do not require extra efforts, because multiple roots at the edges are resolved by analytic root-finding techniques and not by reducing the size of iterative steps as in most numerical methods. For these datasets, our algorithm appeared to be three times faster than the LG-algorithm.

It is worth mentioning that a very simple hill-

Image	Number of elements	LG-algorithm (min:sec)	Our algorithm (min:sec)
—	1	:57	:46
4.7 A	10	10:27	3:23
4.7 B	91	27:02	7:31
4.7 C	820	50:32	13:53
4.7 D	7381	95:52	31:30

Table 4.4: Rendering times for the Sphereflake model.

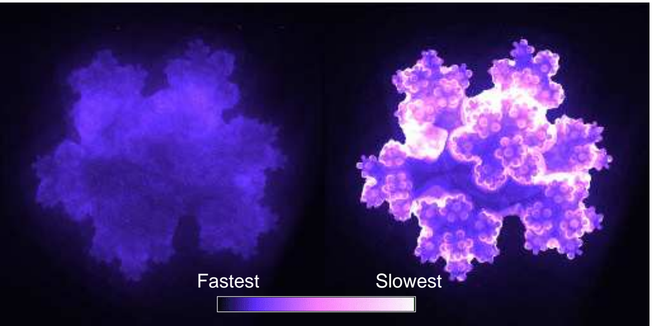


Figure 4.8: Time-profiling charts for our algorithm (left) and the LG-algorithm (right). The left image shows almost uniform time complexity over the scene. The right image shows that the rendering was much slower at the silhouette edges, which agrees with the results reported by Kalra and Barr [38]. Both images were rendered shooting 1 ray/pixel, frame size 512 x 512.

climbing approach renders images 4.7A - 4.7D only 1.75 times slower than our algorithm. The hill-climbing method steps along the ray, evaluating pairs $f(t)$ and $f(t+\epsilon)$, until a sign change is detected. Then a standard root-refinement routine (such as regula falsi) is called to close on the root. However, the optimal climbing step ϵ had to be found experimentally for each sphere-flake, which makes this method much less attractive.

The next picture demonstrates the small-objects problem. The sea urchin in Figure 4.9 is modeled by a spherical Gaussian bump and several radial implicit line segments of various lengths. The endpoints of the longest segments are located outside the region of influence of the central core, resulting in very sharp spikes that are thinner than the pixel size. The algorithm did not miss these fine features. The right image in Figure 4.9 gives an example of UV-mapping, projected back onto the surface.

The dataset for the images in Figure 4.10 is based upon the tree model from [31][2]. There are 31 'branches' and 512 'needles' in the dataset. The left image is rendered with conventional primitives in 9 min 36 sec;

[2] This image is created with a Gaussian kernel. Compare with a coral tree in Fig. 2.20, which was made with a Cauchy kernel.

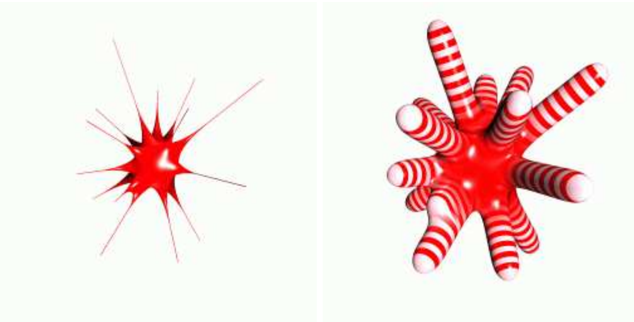


Figure 4.9: Two sea-urchins. Left: thin objects are not a problem for the rendering algorithm. Right: *UV*-mapping, cross-dissolved over blending regions.

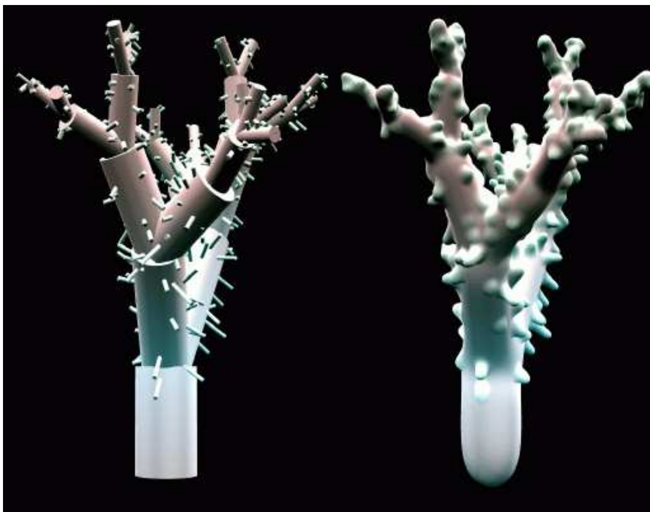


Figure 4.10: Coral tree: offset surface (left) and smoothed implicit surface (right). This model served as a test-bed for various speed optimizations discussed in Section 4.4. Number of elements 548.

the right image was re-rendered several times with different optimization techniques as discussed in the previous section, yielding a best time of 28 min 30 sec (frame size 320 x 240, supersampling adaptively).

The next image shows the generality of the algorithm. The Hermite crab in Figure 4.11 is modeled by implicit functions of 5 different types with characteristic sizes a hundred times different. The surface was successfully detected and shaded everywhere, including the fine details, such as the pincer grips.

The last example demonstrates the rendering speed of the algorithm. Figure 4.12 shows three images of the *Spinal Starecase*, a 36-legged 333-segmented creature modeled by Alan Dorin [3] as described in [22]. The upper image is rendered by *Rayshade*, a public domain ray-tracer [58], widely used in computer

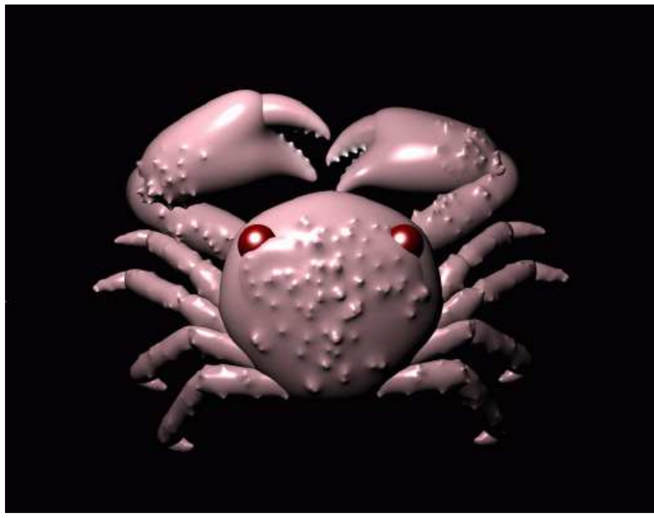


Figure 4.11: Large Hermite crab. Notice the use of controlled blending: leg segments do not blend with each other while most other body parts do. Number of elements 988.

graphics research community because of its high efficiency and extensibility. The middle and the lower images are rendered by *RATS*. To make a fair comparison, the same accelerating techniques are used in both programs (4 x 10 x 10 grids). Table 4.5 gives more details.

The upper and the middle images in Figure 4.12 are not identical – the lighting schemes are different and so are the surface colors. Still, these images look similar enough to give a good indication about the speed of our program. The *Spinal Starecase* model shows the lowest ratio of rendering times between its 'soft' and 'hard' versions, which is 1.15. Normally, this ratio is much higher: 2.97 for the model of a coral in Figure 4.10 and the average of 3.2 for the numerous examples of marine life forms presented in Chapter 3.

4.6. CONCLUSIONS

The key idea of the ray-tracing algorithm presented in this chapter is to keep the modeling implicit equation (4.5) in polynomial form, so it may be solved quickly during the ray/surface intersection test. The polynomial representation of all implicit functions is obtained via Hermite interpolation.

The algorithm has been extensively tested with a large number of implicit functions, most of which were obtained via a convolution technique. Table 4.6 lists all implicit primitives that are implemented in the *RATS* system. All of these functions work well with the algorithm.

[3] The author's original spelling is preserved.

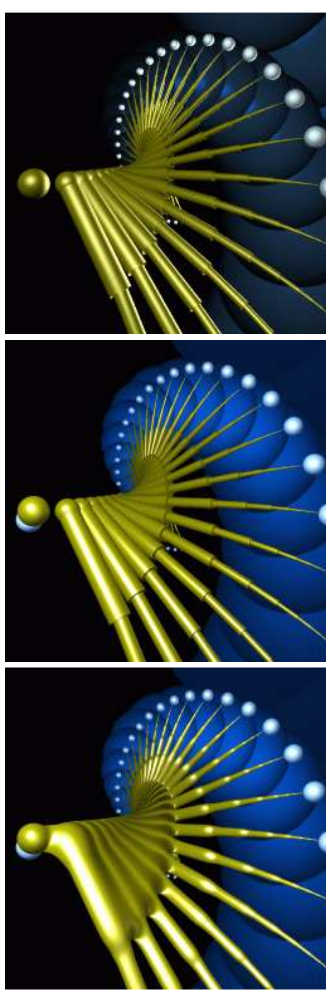


Figure 4.12: *RATS* renders the convolution surfaces of "Spinal Starecase" (bottom) faster than Rayshade does it for the conventional version of the model, represented by cylinders (top). The middle image is also rendered by *RATS*. Total number of elements: 478. Conventional model (top and middle) contains 333 cylinders and 145 spheres. Convolution surface (bottom) is based upon 333 implicit line segments, 38 implicit Gaussian points, 107 spheres.

Image	**Rendered by**	**Rendering time**
Top	Rayshade	30 min 24 sec
Middle	RATS	22 min 11 sec
Bottom	RATS	25 min 28 sec

Table 4.5: Rendering times for the Spinal Starecase model, pictured in Figure 4.12. All images are supersampled with at most 16 rays per pixel, frame size 320 x 480.

Kernel	**Modeling primitives**				
	point	line	plane	arc	triangle
Cauchy	•	•	•	•	•
Gaussian	•	•	•	·	·
Inverse	∘	∘	·	·	•
Squared	∘	∘	·	∘	·
Polynomial	•	∘	∘	∘	·

Table 4.6: Primitives/kernels implementation chart: (•) primitive implemented; (∘) primitive not implemented; (·) primitive can not be implemented (no closed-form solution).

The algorithm has a number of valuable features:

1. *Modularity*
   New modeling implicit functions f_i may easily be added, provided they meet the general assumptions given in Section 4.1.3.

2. *Generality*
   Changing parameters of the modeling functions does not require re-adjustments of the rendering parameters. This allows the designer to concentrate on the model, not on the rendering.

3. *Compatibility with other speed-up techniques*
   Models may consist of a large number of implicit functions f_i. Since most of them have finite bounds, the rendering speed benefits from space-packing methods, such as grids, in the same manner as for conventional primitives.

4. *High speed*
   Rendering speed has always been an issue of great importance, especially when interactive rates are required during designing stage. All implicit objects presented in this paper were created or modified from their original 'explicit' models interactively, using the described algorithm as a viewing tool.

The only drawback of the algorithm that may cause objections is that it works with an approximation, not with a 'true' implicit formulation of the surface. The answer to that is somewhat philosophical: any observation adds distortions to the system or phenomenon that is being observed. This principle seems to hold for any system with an external observer present. Quantum behavior of particles and human ideas are both impossible to communicate in a 'lossless' manner; distortions are inevitable as the information is being passed to an observer.

Everything becomes a little different
as soon as it is spoken out loud.
Hermann Hesse (1877-1962)

During rendering, our algorithm also creates a surface that is 'a little different' from the original plan. The important fact is that we are aware of such deviations and have means to control it effectively. It could be useful to develop an automatic error-control mechanism that would be able to determine if more than two interpolants are required for a particular implicit model. A solution to this problem seems to be achievable.

To conclude, we want to emphasize the importance of analytical nature of our algorithm. In its basic version (two Hermite interpolants per ray per primitive), the algorithm finds the location of the surface at a flat rate of a **single** field function evaluation plus fixed cost of interpolation and root-solving (Figure 4.7 shows nearly constant rendering time over the whole image). That compares favorably to numeric methods that require iteration, especially if the modeling functions f_i are non-algebraic. The field functions, developed and presented in Chapter 2 and Appendix B provide convincing examples of such non-algebraic functions.

> I consider that I understand an equation when I can predict the properties of its solutions, without actually solving it.
>
> *–P.A.M. Dirac*

> See in the root!
>
> *–Koz'ma Prutkov*
> *"Thoughts and Aphorisms", 1854*

Chapter 5

Epilogue

In this dissertation, we have developed a solid platform for using convolution surfaces for computer graphics purposes. Specifically, we addressed three major aspects of the problem: how to describe convolution surfaces mathematically, how to employ them for design of complex shapes and how to render objects modeled with convolution surfaces. We hope that methods and algorithms discussed on these pages will find their users among computer graphics community, both researchers and practitioners.

There are certain areas in our *Formulation – Modeling – Rendering* framework that still can be improved. For instance, an implicit cubic curve, produced via convolution with a polynomial kernel, may be a valuable addition to the arsenal of implicit primitives (*Formulation*). Modifying an object's blobbiness along the surface of the object was mentioned but never actually used, although it could help reducing unwanted blending (*Modeling*)[1]. Volumetric wrinkles and other high-frequency details could have been explored more (*Modeling*). Finally, the mapping of flat textures onto convolution surfaces is a topic of great importance (*Rendering*), which is at present undergoing major developments [53, 81].

As far as implementation is concerned, a graphics user interface for the modeling part of the *RATS* system could facilitate the design process. However, speaking in terms of a software production cycle, we strongly believe that the core of the *RATS* system is already wide and stable enough to be passed to the development stage.

What's next? Although the mathematical and rendering aspects of using convolution surfaces merit additional research, we believe that modeling with convolution surfaces should take precedence and guide the process of their development. So far, we have considered only a limited range of modeling situations where convolution surfaces can be applied. The topics of further research, that may benefit from the results presented in this thesis, are truly inviting.

Several areas of possible application of convolution surfaces were briefly mentioned in the introduction. They concern modeling and animating situations where objects behave essentially as rigid bodies, moving in some unusual force fields (recall an example with the gravity field of a donut-shaped planet or a journey to the center of a Gaussian blob).

Another large set of interesting problems is related to modeling interactions between objects represented with implicit surfaces. This territory seems more exciting and much more challenging. Objects may collide, deform and undergo topological changes. Objects also may react to external forces as elastic and non-elastic bodies. In general, the behavior of interacting solids is very complex and often deceptive.

To illustrate, consider a bubble floating in a lava lamp. At a glance, it may seem as a perfect example of the original Blinn's blob, i.e. a subject for straightforward implicit representation. In fact, on its way *up* along the lamp, such a bubble *does* look like a blob, as it separates from the lava substance and forms an isolated spherical shape. The situation changes drastically when the bubble cools down, descends and starts to merge with the lava mass. Before the actual merger happens, both the bubble and the rest of the lava mass participate in the following processes: collision, mutual deformation (squeezing and bulging), growing tension on the contact surface (double membrane), formation of a puncture in the contact membrane. During the explosive growth of the puncture, the actual merger happens. None of this occurs during the separation phase.

Naturally, it is very desirable to be able to describe the motion and appearance of a 'simple' bubble in some unified consistent way, regardless of direction of its movements and topological changes it undergoes.

Physically-based animation of implicit surfaces is currently a focus of intensive research [18, 21, 28]. To what extent convolution surfaces may help in modeling interactions between implicit surfaces, is sub-

[1] Such blending occurs, for instance, in animating human lips that are modeled with implicit arcs, one arc per lip. The upper and lower arcs must always blend in the corners, thus blendability in these regions must be high. The middle regions should never blend with each other.

ject for further investigation. However, it seems very likely that the closed form implicit functions, developed in this dissertation, may assist in accurate simulations of such subtle phenomena as the building of tension in membrane structures, which are needed to model merger under pressure correctly. Similarly, the gridless (i.e., contiguous) volumetric representation may help to ensure that the contact between colliding bodies and the consequent merger are detected and described accurately.

It also seems clear that a more general modeling paradigm is needed to incorporate various types of interactions between implicit surfaces. Every change in the appearance and topology of the object should be governed by a single set of rules and corresponding algorithms. The Unified Implicit Shape Equation that will incorporate the appearance, motion and deformation of complex interacting systems, is still to be written.

Appendix A

Field functions for point primitives

In the following formulae r denoted a Euclidean distance to a point of interest (x, y, z), i.e., $r^2 = x^2 + y^2 + z^2$. For better clarity, all scaling coefficients that control the width and the height of all function, are set to 1.

A.1. CAUCHY FUNCTION

$$h(r) = 1/(1 + s^2 r^2)^2, \qquad r > 0$$

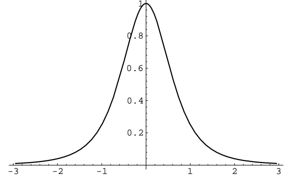


Figure A.1: Cauchy function.

Note: as explained in the text, this is in fact a squared Cauchy distribution.

A.2. GAUSSIAN FUNCTION

$$h(r) = \exp(-a^2 r^2), \qquad r > 0$$

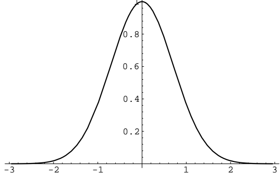


Figure A.2: Gaussian function.

A.3. INVERSE FUNCTION

$$h(r) = 1/r, \qquad r > 0$$

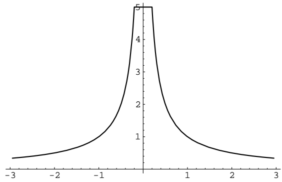


Figure A.3: Inverse function.

A.4. INVERSE SQUARED FUNCTION

$$h(r) = 1/r^2, \qquad r > 0$$

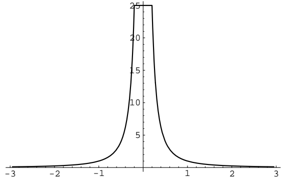


Figure A.4: Inverse squared function.

A.5. METABALLS

$$h(r) = \left[\begin{array}{ll} 1 - 3r^2 & 0 \leq r \leq \frac{1}{3}; \\ \frac{3}{2}(1-r)^2 & \frac{1}{3} < r \leq 1; \\ 0 & r > 1; \end{array} \right.$$

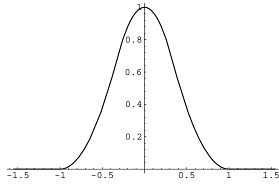


Figure A.5: Metaballs.

A.6. SOFT OBJECTS

$$h(r) = \left[\begin{array}{l} 1 - (\frac{4}{9})r^6 + (\frac{17}{9})r^4 - (\frac{22}{9})r^2, \qquad r < 1; \\ 0 \qquad r > 1; \end{array} \right.$$

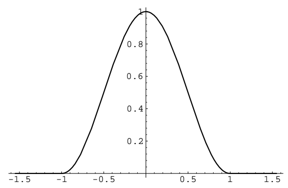


Figure A.6: Soft objects.

A.7. W-QUARTIC POLYNOMIAL

$$h(r) = (1 - r^2)^2, \qquad r < 1$$

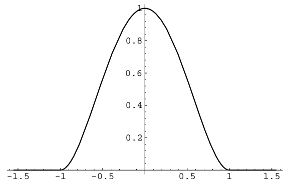


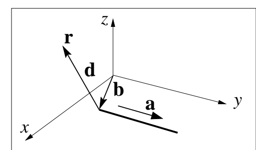
Figure A.7: W-quartic polynomial.

Appendix B

Field functions for line primitives

The following functions describe the scalar fields produced by line segments of length L, convolved with various kernels. The notation used:

$\mathbf{r}$	point of interest (x, y, z)
$\mathbf{b}$	segment base (bx, by, bz)
$\mathbf{a}$	segment axis (ax, ay, az)
$\mathbf{d}$	vector from segment base to a point $\mathbf{r}$
d	length of $\mathbf{d}$
x	dot product of $\mathbf{d}$ and $\mathbf{a}$



Three-dimensional plots show the field distributions in $z = 0$ plane.

B.1. CAUCHY LINE SEGMENT

$$
\begin{aligned}
F_{line}(\mathbf{r}) = & \\
& = \frac{x}{2p^2(p^2 + s^2x^2)} + \frac{L - x}{2p^2q^2} + \\
& + \frac{1}{2sp^3}(\text{atan}[\frac{sx}{p}] + \text{atan}[\frac{s(L-x)}{p}]),
\end{aligned}
$$

where s defines the kernel width and p and q are distance terms:

$$
\begin{aligned}
p^2 &= 1 + s^2(d^2 - x^2), \\
q^2 &= 1 + s^2(d^2 + L^2 - 2Lx)
\end{aligned}
$$

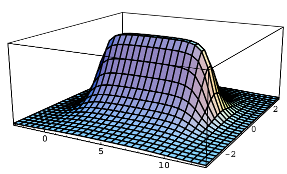


Figure B.1: Cauchy line segment.

B.2. GAUSSIAN LINE SEGMENT

$$F_{line}(\mathbf{r}) = e^{-a^2(d^2 - x^2)}(\text{erf}[a(L - x)] + \text{erf}[ax])$$

Figure B.2: Gaussian line segment.

B.3. INVERSE POTENTIAL LINE SEGMENT

$$F_{line}(\mathbf{r}) = \ln(L - x + \sqrt{d^2 - 2xL + L^2}) - \ln(d - x)$$

Figure B.3: Inverse potential line segment.

B.4. INVERSE SQUARED LINE SEGMENT

$$F_{line}(\mathbf{r}) = \frac{\operatorname{atan}\frac{x}{\sqrt{d^2-x^2}} + \operatorname{atan}\frac{L-x}{\sqrt{d^2-x^2}}}{\sqrt{d^2 - x^2}}$$

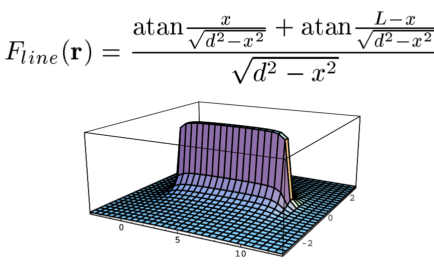


Figure B.4: Inverse squared potential line segment.

Formerly, when one invented a new function, it was to further some practical purpose; today one invents them in order to make incorrect the reasoning of our fathers, and nothing more will ever be accomplished by these inventions.

Jules Henry Poincare (1854-1912)

B.5. POLYNOMIAL LINE SEGMENT

$$\begin{aligned} F_{line}(\mathbf{r}) &= \\ &= R^4((l_2^5 - l_1^5)\frac{1}{5} - \\ &= (l_2^4 - l_1^4)x + \\ &= (l_2^3 - l_1^3)\frac{2}{3}((2x^2 + d^2 - R^2)) + \\ &= (l_2^2 - l_1^2)2x(R^2 - d^2) + \\ &= (l_2 - l_1)(R^2 - d^2)^2), \end{aligned}$$

where R is the width of the kernel and $[l_1, l_2]$ is the integration interval I as shown in Figure 2.8.

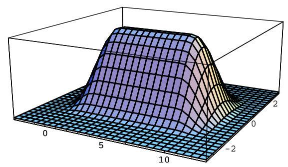


Figure B.5: Polynomial line segment.

Appendix C

RATS Overview and Command Language

C.1. OVERVIEW

NAME

RATS – Ray-Tracing and Animation Tools Software

SYNOPSIS

rats [-/+options] [filename.{art, rat, ice, dat}]

options are:

-?	print this message
-a	use automatic tiling of windows for X displays
-c	use color XPM icons
-d	display traced image line by line
-e	echo on/off
-g	graphics environment on/off
-i	print version information
-j	jitter on/off
-l filename	log all output into a file
-m	memory watcher on/off (may be slow!)
-o file.fmt	set output file name and format
-q	quit after processing input files
-s value	set sampling (number of rays per pixel)
-t	use thumbnail image icons
-v	verbose mode on/off
-w	warning messages on/off
-y	use gray-scale visual for pseudo-color X displays
-r width hight	set horizontal and vertical image resolution

Plus '+' turns the options on, minus '-' turns it off. If no input files are specified, *RATS* read commands from keyboard. Otherwise, *RATS* reads command from input files, that are stored in a LIFO queue; after the last file is processed, *RATS* continues to read commands from keyboard. Command files may be called from inside each other; infinite loop calls are disabled. By convention, input files are expected to have 'art', 'rat', 'ice', 'dat' extensions – they all are treated as batch files.

DESCRIPTION

RATS was born at Caltech in 1994 as a toy ray-tracer written by the author for the *Computer Graphics Laboratory* course CS174. At that time, the course was taught by Jim Kajiya, with a driving force 'of great pitch and moment', which propelled the project far enough to be able to live on its own. Over the years, *RATS*

[1], matured into a complete system for modeling, rendering and animation of conventional and implicit surfaces. Version 7.31 amounts to nearly 60,000 lines of code. *RATS* has been compiled and tested in various flavors of UNIX operating system, e.g. HP-UNIX (Hewlett Packard), Sun OSF1 (Alpha stations), SunOS 4.X (Sun SPARC Stations), ULTRIX (DEC workstations), IRIX (SGI) and Linux (PC).

At a glance, *RATS* has the following features.

- Conventional modeling primitives: arc, box, brick, cone, cylinder, dot, patch, polygon, plane, quadrics, sphere, torus, triangle
- Height fields
- Object hierarchies
- Object instances
- Linear transformations
- Flexible super/under/sampling
- Depth of field
- Motion blur
- Penumbrae
- Transparent shadows
- Solid and flat textures
- Automatic and nested grids
- Implicit modeling primitives: point, line, arc, triangle, plane
- Selective visibility [64]
- Clouds of light and halos
- Surface and texture assignments
- Scalar, color and vector variables
- Internal man pages
- Internal tests for most commands
- Animation tools
- Time profiling tools
- Image arithmetic (AND, XOR, SUB)
- Image viewer/tiler for X displays
- Thumbnail image icons
- Interactive previewer
- TIFF, TARGA, MTV, FLI support

C.2. COMMAND LANGUAGE

RATS command language contains over one hundred commands that are grouped according to their purpose as follows.

RUN	basic commands to start/run/stop the program
CAMERA	create virtual camera
MATERIALS	create textures and surfaces
PRIMITIVES	create primitive objects
OBJECTS	create and manipulate objects
OPTIONS	set rendering/modeling/running/viewing options
SCENE	set lights, backgrounds, atmosphere... and shoot
TOOLS	handy little things: animation, file conversion, etc.

The following naming conventions are used:

`UPPERCASE`	explicit triple x y z or user-defined vector or color variable
`lowercase`	keyword, explicit scalar or user defined scalar variable
`[anything]`	square brackets indicate optional arguments
`Color`	standard color, as LightBlue or user-defined color variable
`fn`	file name, as '../somewhere/somefile.tif'
`sn`	surface name, as 'Plastic'

Many commands have internal test suites and manual pages. Such commands are marked with letters 'T' and 'M', respectively. M-commands usually have complex syntax and some of them require more than one line of comments. If the 'argument' field of an M-command shows dots '...', consult the manual pages for detailed description of the arguments.

[1] The exact meaning of this acronym is still undecided — it evolves with the program and the process is not over. Among the most recent interpretations are Ray-tracing/Animation/Tools/Software, Ray-tracer/Animator/Testbed/System, Rocket/Assisted/Takeoff/System, and Russian/Artists/Can't/Spell.

RUN

Commands of this group control the basic execution of the program.

Opcode	Arguments	Comments	
`read`	`fn1 [fn2 ...]`	read commands from batch file(s)	
`pause`	`[message]`	[print message] and wait for ENTER	
`return`		return from a batch file	
`info`		print version information	
`help`	`[command]`	print help [for a single command]	
`man`	`command`	print the manual page for a command	
`test`	`command [print]`	run a test for a command [print only]	
`log`	`[filename]`	open/close a log file	
`var`	`name = value`	create/update a variable	M
`stopwatch`		start/stop stopwatch	
`quit`		finish the session	

The following characters have special purpose.

Character	Comments
`#`	comment sign
`@`	suppress echo of the following command
`, < >`	delimiters (also space and tab)
{	arguments continue on the next line[s]
}	list of arguments ended

CAMERA

RATS supports two types of camera settings: a conventional type that employs `eyep` and `fov` parameters and more flexible type as described in Foley et. al. [25], via `cp` and `window` command. The other camera-related parameters, such as `aperture, shutter, focus`, etc. are common for both types.

Opcode	Arguments	Comments	
`aperture`	`radius`	camera lens, default 0 (pinhole camera)	T
`focus`	`distance`	to focal plane, default distance to VRP	
`shutter`	`fraction`	of open time, min 0 (no blur) max 1.0	MT
`vrp`	`x y z`	view reference point in world coordinates	
`vup`	`x y z`	view up in world coordinates	
`cp`	`x y z w`	center of projection in VRC	
`vpn`	`x y z`	view plane normal in world coordinates	
`window`	`x X y Y`	min x max X, min y max Y in VRC	
`eyep`	`x y z`	eye position in world coordinates	
`fov`	`hor vert`	field of view in degrees	

MATERIALS

Before any objects are created, their material must be defined. In the text, the `surface` command is referred as `material`, which is more correct conceptually. For compatibility purposes, *RATS* is still using the opcode `surface`.

Opcode	Arguments	Comments	
`texture`	`...`	create/update a texture	MT
`surface`	`...`	create/update a surface (a.k.a. material)	MT

PRIMITIVES

To create a stand-alone primitive, its surface must be specified after the opcode, e.g. `sphere Plastic, 1, 0 0 0`. However, if the primitive belongs to a chain of the same primitive objects (see `chain`), the surface name is omitted.

Opcode	Arguments	Comments	
`arc`	`a b P N a1 a2`	see torus; add start angle and width (deg)	T
`arch`	`b P1 P2 P3`	tube radius + three points P1 P2 P3	T
`box`	`PMIN PMAX`	box defined by min and max points	T
`brick`	`x y z`	box of (xyz) dimensions around the origin	T
`cone`	`b BASE a APEX`	radius at base, radius at apex	T
`cylinder`	`r BASE APEX`	radius, base and apex; lids are closed	T
`dot`	`POS`	polynomial point source	T
`gauss`	`POS`	Gaussian point source	T
`line`	`BASE APEX`	line convolved with a Gaussian potential	T
`patch`	`(V1 N1) x 3`	phong shaded triangular patch	T
`pipe`	`r BASE APEX`	same as cylinder but without lids	T
`polygon`	`V1... Vn`	planar polygon with n vertices (max 128)	T
`plane`	`POS NORM`	plane with a point and normal	T
`quadric`	`...`	create a quadric in local coordinates	MT
`sphere`	`r POS`	radius and center	T
`terra`	`...`	create a terrain out of image file	MT
`torus`	`a b POS NORM`	sweep and tube radii, center, normal	T
`triad`	`V1 V2 V3`	triangle conv. with Newtonian potential	T
`triangle`	`V1 V2 V3`	flat triangle	T

OBJECTS

Primitive objects, created via commands listed above may be organized and manipulated as hierarchies of objects. Objects at every layer of the hierarchy may be accompanied by their transformations.

Opcode	Arguments	Comments	
`chain`	`sn, name`	start a new chain of primitives	T
`object`	`name`	start new compound object	T
`instance`	`...`	clone object	T
`close`		finish current compound object or chain	T
`translate`	`x y z [object]`	along (x y z), last object by default	T
`rotate`	`x y z [object]`	about (x y z)	T
`scale`	`x y z [object]`	along (x y z)	T
`explode`	`x y z [object]`	radially explode an object by (x y z)	
`transform`	`[object]`	force transforms now (to make blur)	

OPTIONS

Options control the output image size and quality, accelerations techniques, memory managements and the way *RATS* interacts with the user.

Opcode	Arguments	**OPTIONS**	**Default**	
`echo`	`on/off`	turn echo on/off	ON	
`warning`	`on/off`	allow warning messages	ON	
`verbose`	`on/off`	run in verbose mode	ON	
`bounds`	`on/off`	use bounding volumes	ON	
`soft`	`on/off`	allow soft objects	ON	
`timer`	`on/off`	do time reports after RT	ON	
`stat`	`on/off`	do statistic report after RT	off	
`map`	`on/off [N]`	create time-profile image	off	
`double`	`on/off`	use double-sided faces	off	
`penumbra`	`on/off`	enable soft shadows	off	T
`noview`	`on/off`	don't display RT image	off	
`nosave`	`on/off`	don't save RT image	off	
`dither`	`on/off`	dither images for viewing	ON	
`gamma`	`value`	gamma correction for viewing	1.0	
`framesize`	`width height`	output resolution	128 128	
`filename`	`name`	output filename	scene	
`format`	`fmt`	save as tif/tga/mtv	tif	
`compress`	`on/off`	compress or not when saving	ON	
`epsilon`	`value`	ray-surface hit precision	1e-4	
`vanish`	`value`	min color value	4e-3	
`maxdepth`	`number`	depth of shading tree	5	
`maxmol`	`number`	max molecule size	300	
`digger`	`[hermite|lagrange|`	who digs roots of		
	`brent|ridder|RF]`	isosurface equations?	hermite	
`contrast`	`r g b`	supersample threshold	.25 .2 .4	
`pack`	`...`	set packing method	grid	M
`indicator`	`[none|all|bar|text|`	set RT progress indicator		
	`pixel|line|ETA]`		bar	
`fragile`	`[none|all|arc|pipe|`	which soft prims change R		
	`cylinder|sphere]`	as set in the material?	none	
`sample`	`...`	set sampling method	1	MT
`mesh`	`...`	polygonal mesh control	robust	M

SCENE

Objects act on *scene*, which contains lights, atmospheric effects and pretty much the rest of the synthetic world which is built by other commands. For instance, arguments for `report` and `reset` commands are: `data`, `model`, `surfaces`, `textures`, `lights`, `background`, `clouds`, `fog`, `cameras`, `options`, `var`.

Opcode	Arguments	Comments	
`light`	`Color [options]`	create a new light source	MT
`fog`	`Color [options]`	add fog to the scene	MT
`background`	`Color [options]`	create a background layer	MT
`cloud`	`Color [options]`	create a cloud layer	MT
`remove`	`object name`	remove an object from the scene	T
`report`	`...`	display values of most parameters	M
`reset`	`...`	set most parameters to default values	M
`preview`	`...`	visual RT preview for X displays	M
`shoot`	`...`	start ray-tracing of the scene	M

TOOLS

Tools group is a catch-all for utilities that perform various operations on images, control the appearance of the X display, etc.

Opcode	Arguments	Comments	
`animate`	`fn1... [options]`	make animation [loop—mirror—dither]	T
`play`	`fn [options]`	playback animation	
`split`	`fn [frame.fmt]`	split animation into frameNNN.fmt files	
`collage`	`[options] fn1 fn2`	make a collage of image several files	
`fli`	`fn1... [options]`	make a FLI file of [speed N][size W H]	T
`convert`	`fn1 fn2`	convert image file fn1 -¿ fn2	
`resize`	`fn x y`	resize image file fn by x and y	
`diff`	`fn1 fn2 [save]`	SUB two image files and save the result	
`and`	`fn1 fn2 [save]`	AND two image files	
`xor`	`fn1 fn2 [save]`	XOR two image files	
`show`	`...`	display file, color, palette, etc.	M
`kill`	`[w1 w2 ...]`	kill some windows, [all of them]	
`refresh`	`[w1 w2 ...]`	refresh some windows, [all of them]	
`tile`	`[options]`	tile windows on an X display	
`colors`		list available standard colors by names	
`rats`		start a friendly smalltalk	
`table`	`v1 v2 v3 v4 ...`	plot a polyline of up to 256 points	
`shell`	`[options]`	shell dataset generator (make your own)	
`horn`	`[options]`	horn dataset generator	

BUGS and RESTRICTIONS

Certain restrictions are hard-wired into the code. For instance, maximal number of nested input files (128), maximal number of operands per command (512), maximal number of layers per texture (12), maximal depth of object hierarchies (16), etc. When an attempt is made to step over these limits, a 'sorry' message is issued, the command is aborted and the previous state of the program is restored. Several bugs have been spotted but not fixed in time.

AUTHOR

Andrei Sherstyuk

15 September, 1998

Version 7.31

C.3. TWO EXAMPLES OF INTERNAL MAN PAGES

C.3.1. Materials

As mentioned before, materials are created using `surface` command, to support compatibility with datasets developed for earlier versions of *RATS*. The following manual pages are copied from the screen as they were produced by the program.

```
RATS> man surface
------------------------------------------------------------------------------
create a new surface or assigns one surface to another

FORMAT 1:
   surface {                  #    create a brand new surface
       alias                  #    a single word as a surface name (required)
       body        RGB,       #  * main body color
       ambient     RGB,       #  * emitting light
       diffuse     RGB,       #  * diffuse color
       specular    RGB,       #  * highlight spot color
       reflective  RGB,       #  * reflected color
       transparent RGB,       #  * transmitted color
       index       k,         #  . index of refraction, default 1
       dispersion  d,         #  . optical density variation, default 0
       shine       p,         #  . Phong shining
       radius      r,         #  . in isolation for blobby materials, default 0
       blobbiness  b,         #  . ditto
       strength    s,         #  . scale the field, default 1
       skip        RAYTYPE    #    make the surface undetectable for some rays
       texture     t1 [+ t2 + ...], #  add up to 12 layers of textures
   }
FORMAT 2:
   surface new = old       # surface assignment


   -----------
   L E G E N D
   -----------

   RGB may be one of the following:
         1. Color     standard palette color or user-defined color var
         2. RGB       explicit color values (0.25 1 0.75)
         3. number%   percent of body color for diffuse and ambient components
                      percent of incident light for spec, refl and transparent
   RAYTYPE may be one or more of the following keywords: pixel, shadow,
                      reflected, transmitted

         *   fields may be defined via vector variables,
         .   fields may be defined via scalars variables
         all fields are optional, except 'alias'


   ---------------
   E X A M P L E S
   ---------------
      surface Mirror   { diffuse LightBlue, reflective 0.7 0.7 0.7          }
      surface Plastic  { body Red, diffuse 80%, specular 50%, shine 40      }
      surface Wood     { diff LightWood, spec White, shine 100, texture Wood }

------------------------------------------------------------------------------
Test etude available. Type "test surface" to try it out
```

C.3.2. Textures

```
RATS> man texture
-----------------------------------------------------------------------------
create a new texture or assigns one texture to another

FORMAT 1:
   texture {                # create a brand new texture
       alias,               # a single word as a name for a new texture
       Target,              # required (see below)
       Method,              # required (see below)       default:
       translate  XYZ,      # translate argument         0 0 0
       scale      XYZ,      # scale texture argument     1 1 1
       times      RGB,      # scale texture values       1 1 1
       bounds     RGB RGB,  # clamp texture values       none (-Inf,Inf)
       turbulence t,        # noise in noisy textures    1 (full noise)
       octaves    r,        # level of noisy details    5-6
       mask       1 1 0..   # masking tiles out, if "scale" option was used
                            # there must be X*Y entries, as set in scale
       replace              # keyword: texture values replace target values
                            # (normally scale)
       texture    t1 [+ t2 +...], # add up to 12 layers of other textures
   }
FORMAT 2:
   texture new = old        # create/update a texture using assignment


   -----------
   L E G E N D
   -----------

   "Target" specifies WHAT must be textured. One of the following keywords:
         1. diffuse, ambient, specular, reflective, transparent, index, shine
                    - modify usual  photometric components of the surface
         2. pigment - modify both diffuse and ambient components
         3. normal  - modify normal vector

   "Method" specifies HOW to texture. One of the following keywords:
         1. marble, agate, granite, moon, onion, wood,
            ripples, sandal, checker, paint
                    - use a  predefined solid textures
         2. Color   - apply color values as a texture function
         3. Fu Fv   - use a pair of UV-based functions. Available functions are:
                      one, x, saw, hat, step, sin, cos, sin^2, cos^2
         3. img.fmt - texture values are taken as RGB from the image file. The
                      are two formats, plain and region-dependent:
            img.fmt plain
                    - all points on the surface are textured, using UV values
            img.fmt region <Min Max> <P1 P2 P3 P4>
                    - a point is textured if it belongs to <Min Max> region and
                      its normal vector goes thru the rectangle set by vertices
                      P1 P2 P3 P4

   ---------------
   E X A M P L E S
   ----------------
      texture RedChecker { pigment Red,    scale 2 2 1, mask 1 0 1 0          }
      texture Bumps      { normal  sin cos,  scale 3 4 1                      }
      texture Wood       { diffuse wood, scale 9 8 1,  bounds <DarkWood White> }
      texture MonaTiled  { diffuse monalisa.tif plain, scale 2 2 1, replace    }
      texture MonaBumped   texture   Bumps + MonaTiled

-----------------------------------------------------------------------------
Test etude available. Type "test texture" to try it out
```

C.4. SELECTED DATASETS

```
################################################################
#   cowrie
#
#   Scene       Spindle Cowrie -- a tiny mollusk (at most 4 cm)
#               that lives usually with gorgonians and feed on
#               their polyps.
#   Date        Wed Aug 27 17:29:06 EST 1997
#   Author      Andrei Sherstyuk
#   Features    Blobby arcs and cylinders
#   Comments    Very simple model, produced by the code below
#               The croissant-like combination of arcs turned
#               out to be so good, that inspired the gorgeous
#               model of a coral crab and 'Wow' animation.
#
#   Objects     100 cylinders + 3 arcs
#   Version     7.12
#   Time        demiurge, sample 4, 640x480: 24 min 29 sec (v 7.11)
#
#   Nice and simple
################################################################
@reset    all  # clean up
@echo     off  # no text output, until it's time to trace
soft      on   # enable implicit surfaces
fragile   all  # make sure that material thickness overwrite individual
sample    4    # 4x4 supersampling grid

#
# Camera
#
eyep      3 3 3
fov       50 50
vrp       0 0 0
vup       0 0 1

#
# Output
#
framesize 640 480
filename  cowrie
format    tif

light White point, position 2 4 3, intensity 1.2, noshadow

#
# The radius of spikes-cylinders is declared as
# a variable so I can adjust it interactively
#
var r = 0.05 hot

#
# Blobby surfaces for the body (s1, s2, s3) and the spikes (s0)
#
surface s0 diff White,     spec White, shine 10,  blob -128, radius .034 strength 2
surface s1 diff OrangeRed, spec White, shine 100, blob  -16, radius .125
surface s2 diff DarkRed,   spec White, shine 100, blob   -8, radius .250
surface s3 diff OrangeRed, spec White, shine 100, blob   -2, radius .500

#
# The skeletal model
#
object SHELL
    arc s1 4, 0.125, <-2.5 -2.5 0>, <0 0 1>  5 80
    arc s2 4, 0.250, <-2.5 -2.5 0>, <0 0 1> 15 60
    arc s3 4, 0.500, <-2.5 -2.5 0>, <0 0 1> 30 30
    # read a datafile produced as "cowrie 100 0.75 > spikes.dat"
    read spikes.dat
close
rotate 1 1 0 30, SHELL

#
# Start ray-tracing
#
echo on
```

```
shoot
return

#-------------------------cut here and save as cowrie.c-----------------------
/*
##############################################################################
#  cowrie.c -- utility to add spikes to the croissant-like body of the shell
#
#  Compile: gcc -o cowrie cowrie.c -lm
#  Use:     cowrie N length > output.dat,
#           where
#           N      - number of spikes
#           length - the length of the spikes
#  Example:
#           cowrie 100 0.75 > spikes.dat
##############################################################################
*/
#include "system.h"
#include "types.h"
#include "vectors.h"

/*
 * This is the "body", an arc defined as
 * sweep radius (float),
 * tube radius (float),
 * center (3 vector),
 * normal (3 vector),
 * start angle (float degree),
 * stop angle (float degree)
 * arc s3 4, 0.500, <-2.5 -2.5 0>, <0 0 1> 30 30
 */
#define CENTERx -2.5
#define CENTERy -2.5
#define CENTERz 0

#define START 5.0   /* degrees */
#define THETA 75.0  /* degrees */
#define R     4.0   /* sweep radius */


/*
 * Prints usage and exits
 */
void usage(char *module)
{
    printf("Usage: %s N length\n", module);
    exit(-1);
}

/*
 * Rotate the point about Z axis
 */
void rotate_point(P, a)
Vector           *P;
double            a;
{
    double x, y, sinA, cosA;

    sinA = sin(a);
    cosA = cos(a);

    x = P->x;
    y = P->y;
    P->x = x*cosA - y*sinA;
    P->y = x*sinA + y*cosA;
}




void main(int argc, char *argv[])
{
    Vector   Base, Apex;
    double   len, length, chance, inc;
```

```
    int      N, i;


    if (argc < 3 ||
       ((N = atoi(argv[1])) == 0) ||
       ((length = atof(argv[2])) == 0.0))
        usage(argv[0]);

    printf("# N = %d, length %g\n", N, length);

    for (i = 0; i < N; i++) {
        inc = deg2rad(START + (double)i/(double)N*THETA);
        /*
         * Make the spike as a cylinder "base -> apex":
         * (0,0,0) -> (length, 0,0), then rotate the spike apex aroung Y
         * randomly and then rotate base and new apex around Z incrementally
         */
        chance = 2.0 * PI * drand48();

        /*
         * Apex point is rotated around Y and translated along X
         */
        len = length * sin(2.0 * inc)*sin(2.0 * inc);

        Apex.x = len * cos(chance) + R;
        Apex.y = 0;
        Apex.z = len * sin(chance);

        /*
         * Base translated along X
         */
        Base.x = R;
        Base.y = 0;
        Base.z = 0;
        rotate_point(&Base, inc);
        rotate_point(&Apex, inc);

        /*
         * Ajust the center
         */
        Apex.x += CENTERx, Apex.y += CENTERy, Apex.z += CENTERz;
        Base.x += CENTERx, Base.y += CENTERy, Base.z += CENTERz;

        /*
         * The spike is ready, dump it
         */
        printf("cylinder s0 r, <%g %g %g>, <%g %g %g>\n",
                                          Base.x, Base.y, Base.z,
                                          Apex.x, Apex.y, Apex.z);
    }
}
```

```
##############################################################
# HORSE:    Sea horse for "Modeling Marine Life".
#
# Comments  This model was conceived sitting in the restaurant
#           "Hideout" in Melbourne, where I and Katya went for
#           deserts one night. All the walls were painted with
#           incredibly grotesque seafood samples, including a
#           seahorse that I liked. So here it is.
#           Use: hermite re-used interpolants, precondensed
#           molecules, faster 30/35% than volatile molecules.
# Objects   Soft: 43 arcs
#           Hard:  2 spheres (the eyes)
# Version   7.12
# Precision SINGLE
# TIME:     resulution 320 512, demiurge, sample 4:
#           --------------------------------------
#           hard:   5 min  25 sec
#           soft:  11 min  52 sec
##############################################################
@reset all
@echo off
pack none
fragile all
sample 4

#
# Camera
#
eyep 0 0 20
fov  13 13
vup  0 1 0
vrp  0 0 0

# Output
#
framesize 160 256
filename  horse
format    tif

# Lights
#
light White point, pos -10 5 10 noshadow, inten 0.75
light White point, pos  5 10 10 noshadow, inten 0.25

# Colors
#
var Body  = SummerSky
var Flin  = Gold
var Apple = Gray15
var Ball  = White
var Spec  = White
var phong = 100

var Lo = 0.5 0.5 0.5
var Hi = 0.7 0.7 0.7

texture TT strength sin one bounds Lo Hi scale 22 22 1 times 1.15 1.15 1.15

# Flins
surface F0 diffuse Flin specular Spec, shine phong, blob -256 rad 0.025
surface F1 diffuse Flin specular Spec, shine phong, blob -128 rad 0.10
# Eyes
surface E0 diffuse Ball  specular Spec, shine phong, ambient Gray
surface E1 diffuse Apple specular Spec, shine 50
# Tail
surface T0 diffuse Body, specular Spec, shine phong, blob -64 rad 0.32
surface T1 diffuse Body, specular Spec, shine phong
surface T1 diffuse Body, specular Spec, shine phong
surface T1 diffuse Body, specular Spec, shine phong
surface T1 diffuse Body, specular Spec, shine phong
surface T1 diffuse Body, specular Spec, shine phong
surface T1 diffuse Body, specular Spec, shine phong
surface T1 diffuse Body, specular Spec, shine phong
# Body
surface B0 diffuse Body, specular Spec, shine phong, blob  -8 rad 0.25
```

```
surface B1 diffuse Body, specular Spec, shine phong, blob -16 rad 0.25
surface B2 diffuse Body, specular Spec, shine phong, blob  -8 rad 0.35
surface B3 diffuse Body, specular Spec, shine phong, blob  -8 rad 0.25 text TT
surface B5 diffuse Body, specular Spec, shine phong, blob  -8 rad 0.05 text TT
surface B4 diffuse Body, specular Spec, shine phong, blob  -8 rad 0.25 stren 0.5

object HORSE
object HEAD # nose + forehead + top + back + cheek + eyes (2 spheres) + horns
    arch B0 0.25 { -0.42857 1.8277 0, -0.87857 1.2277 0, -1.37857 0.9777 0 }
    arch B0 0.25 { -0.62857 1.7277 0, -0.37857 2.2277 0, 0.17143  2.4777 0 }
    arch B0 0.25 {  0.18393 2.4777 0,  0.55893 2.4777 0, 0.87143  2.2902 0 }
    arch B1 0.25 {  0.87143 2.2902 0,  1.05893 2.0402 0, 1.12143  1.7277 0 }
    arch B1 0.25 {  0.02143 2.3527 0, -0.1     1.6027 0, -0.86607 1.2277 0 }
    sphere E0 0.15 <-0.50 1.50 0.75 >
    sphere E1 0.10 <-0.50 1.45 0.85 >
    arch F0 0.05 { -0.766 1.615 0.25, -1.016 1.8027 0.25, -1.45357 1.9277 0.25 }
    arch F0 0.05 { -0.766 1.9902 0, -0.95357 2.2402 0, -1.26607 2.3652 0 }
    arch F0 0.05 { -0.641 2.3027 0, -0.82857 2.7402 0, -1.26607 3.1152 0 }
    arch F0 0.05 { -0.328 2.5902 0, -0.32857 2.9277 0, -0.64107 3.3652 0 }
close
object BODY
    # front
    arch B0 0.25 {  1.07143 1.5527 0, 0.72143 1.0027 0, -0.0660701 0.5402 0 }
    arch B0 0.25 { -0.0660701 0.5402 0, -0.44107 0.1027 0, -0.37857 -0.5223 0 }
    arch B0 0.25 { -0.37857 -0.5223 0, 0.12143 -1.0223 0, 0.62143 -1.2723 0 }
    arch B4 0.25 {  0.62143 -1.2723 0, 0.87143 -1.3973 0, 1.12143 -2.0223 0 }
    # back
    arch B0 0.25 { 1.12143 1.7277 0, 0.93393 0.9777 0, 0.62143 0.4777 0 }
    arch B3 0.25 { 0.62143 0.4777 0, 0.49643 -0.0222998 0, 0.87143 -0.5848 0 }
    arch B4 0.25 { 0.62143 0.4777 0, 0.49643 -0.0222998 0, 0.87143 -0.5848 0 }
    arch B0 0.25 { 0.87143 -0.5848 0, 1.24643 -1.1473 0, 1.24643 -2.1473 0 }
    # middle
    arch B0 0.25 { 0.00393 0.1902 0.0, 0.05893 -0.2100 0, 0.63393 -0.9723 0 }
    arch B2 0.33 { 0.52143 0.7777 0.0, 0.06430 -0.0223 0, 0.57143 -0.9848 0 }
close
object TAIL
    arc T0 1.000 0.25 < 0.2   -2.0125 0> <0 0 -1> 0 90
    arc T1 1.000 0.25 < 0.2   -2 0> <0 0 -1>   90 90
    arc T1 0.500 0.25 <-0.3   -2 0> <0 0 -1>  180 90
    arc T1 0.500 0.25 <-0.3   -2 0> <0 0 -1>  270 90
    arc T1 0.250 0.25 <-0.05  -2 0> <0 0 -1>    0 90
    arc T1 0.250 0.25 <-0.05  -2 0> <0 0 -1>   90 90
    arc T1 0.125 0.25 <-0.175 -2 0> <0 0 -1>  180 90
    arc T1 0.125 0.25 <-0.175 -2 0> <0 0 -1>  270 90
    sphere T1 0.25 <-0.050 -2 0>

    arc B5 1.000 0.25 < 0.20   -2 0> <0 0 -1>  290 90
    arc B3 1.000 0.25 < 0.21   -2 0> <0 0 -1>  20 90
    arc B3 1.000 0.25 < 0.20   -2 0> <0 0 -1>  340 60
    arc B3 1.000 0.25 < 0.19   -2 0> <0 0 -1>   30 60
close
object FLIN
    arch F1 0.10  0.93393 1.4777 0, 1.43393 1.7277 0, 1.93393 2.2902 0
    arch F1 0.10  0.93393 0.9777 0, 0.93393 0.7277 0, 1.43393 0.1652 0
    arch F1 0.10  0.93393 1.0402 0, 0.99643 0.8527 0, 1.62143 0.4152 0
    arch F1 0.10  0.93393 1.1027 0, 1.05893 0.9777 0, 1.74643 0.6652 0
    arch F1 0.10  0.93393 1.1652 0, 1.12143 1.1027 0, 1.87143 0.9152 0
    arch F1 0.10  0.93393 1.2277 0, 1.18393 1.2277 0, 1.99643 1.1652 0
    arch F1 0.10  0.93393 1.2902 0, 1.24643 1.3527 0, 2.05893 1.4777 0
    arch F1 0.10  0.93393 1.3527 0, 1.30893 1.4777 0, 2.05893 1.7902 0
    arch F1 0.10  0.93393 1.4152 0, 1.37143 1.6027 0, 2.05893 2.0402 0
close
close HORSE

echo on
shoot
return
```

Bibliography

[1] V. Adzhiev, A. Pasko, V. Savchenko and A. Sourin, "Shape Modeling with Real Functions", *Open Systems*, Vol.5, No. 19, 1995, pp.14–18 (in Russian). Electronic version available at http://www.osp.ru/os/1996/05/source/14.html.

[2] M. Aono and T. Kunii, "Botanical Tree Image Generation", *IEEE Computer Graphics and Applications*, Vol. 4, No. 5, May 1984, pp. 10–29, 32–34.

[3] T. Beier, "Practical Uses for Implicit Surfaces in Animation", *SIGGRAPH Course 23*, August 1990, pp. 20.1–20.11.

[4] H. Bidasaria, "Defining and Rendering of Textured Objects through the Use of Exponential Functions", *Graphical models and image processing*, Vol. 54, No. 2, March, 1992, pp.97–102.

[5] C. Blanc and C. Schlick, "Extended Field Functions for Soft Objects", *Implicit Surfaces'95*, Proceedings of the first international workshop on Implicit Surfaces, Grenoble, France, April 1995, pp. 21–32.

[6] C. Blanc and C. Schlick, "Ratioquadrics: an Alternative Model for Superquadrics", *The Visual Computer*, Vol. 12, No. 8, pp.420–428, 10/1996.

[7] J. Blinn, "A Generalization of Algebraic Surface Drawing", *ACM TOG*, Vol. 1, No. 3, July 1982, pp. 235–256.

[8] J. Bloomenthal, "Modeling the Mighty Maple", *Computer Graphics (SIGGRAPH '85 Proceedings)*, Vol. 19, No. 3, San Francisco, California, July 1985, pp.305–311.

[9] J. Bloomenthal, "Polygonization of Implicit Surfaces", *Computer Aided Geometric Design*, 5(1988), pp. 341–355.

[10] J. Bloomenthal and B. Wyvill, "Interactive Techniques for Implicit Modeling", *SIGGRAPH Course 23*, August 1990, pp. 17.1–17.8

[11] J. Bloomenthal, "Techniques for Implicit Modeling", *SIGGRAPH Course 23*, August 1990, pp. 13.1–13.8.

[12] J. Bloomenthal and K. Shoemake, "Convolution Surfaces", *Computer Graphics (SIGGRAPH '91 Proceedings)*, Vol. 25, No. 4, Las Vegas, Nevada, July 1991, pp. 251–257.

[13] J. Bloomenthal, *Skeletal Design of Natural Forms*, doctoral dissertation, University of Calgary, Dept. Computer Science, 1995.

[14] J. Bloomenthal, "Bulge elimination in implicit surface blends", *Implicit Surfaces'95*, Proceedings of the first international workshop on Implicit Surfaces, Grenoble, France, April 1995, pp. 7–20.

[15] J. Bloomenthal, editor, *Introduction to Implicit Surfaces*, Morgan Kaufmann Inc, 1997.

[16] J. Bloomenthal, "Bulge Elimination in Convolution Surfaces", *Computer Graphics Forum*, Vol. 16, No. 1, 1997, pp.31–41.

[17] J. Buchanan and P. Turner, *Numerical Methods and Analysis*, McGraw-Hill, Inc, 1992.

[18] M.-P. Cani-Gascuel, "Layered Deformable Models with Implicit Surfaces", *Proceedings of Graphics Interface '98*, Vacouver, Canada, June 1998, pp. 201–208.

[19] S. Colburn, "Solid Modeling with Global Blending for Machining Dies and Patterns", *SAE Technical Paper Series* 900878, Society of Automotive Engineers, Inc., 1990.

[20] B. Crespin, C. Blanc and C. Schlick, "Implicit Sweep Objects", *Computer Graphics Forum*, Vol.15, No.3, pp. C165–74, 1996.

[21] M. Desbrun and M.-P. Gascuel, "Animating Soft Substances with Implicit Surfaces", *Computer Graphics (SIGGRAPH '95 Proceedings)*, Los Angeles, California, August 1995, pp. 287–290.

[22] A. Dorin, "A Model of Protozoan Movement for Artificial Life", *Insight Through Computer Graphics: Proceedings of the Computer Graphics International 1994 (CGI94)*, Gigante and Kunii (eds), World Scientific, 1996, pp. 28–38.

[23] P. Embree and B. Kimble, *C Language Algorithms for Digiatal Signal Processin*, Prentice Hall, 1991.

[24] E. Ferley, M.-P. Cani-Gascuel, and D. Attali, "Skeletal Reconstruction of Branching Shapes", *Implicit Surfaces'96: 2nd International Workshop on Implicit Surfaces*, Eindhoven, The Netherlands, October 1996.

[25] Foley J.D., van Dam A., Feiner S.K. and Hughes J.F., *Computer Graphics, Principles and Practice*, second edition. Reading, Massachusetts: Addison-Wesley, 1990.

[26] A. Fournier, "The Modelling of Natural Phenomena", *SIGGRAPH Course 22*, August 1994, pp. 32–48.

[27] D. Fowler and H. Meinhardt and P. Prusinkiewicz, "Modeling Seashells", *Computer Graphics (SIGGRAPH '92 Proceedings)*, Vol. 26, July 1992, pp. 379–388.

[28] M.-P. Gascuel, "An Implicit Formulation for Precise Contact Modeling between Flexible Solids", *Computer Graphics (SIGGRAPH '93 Proceedings)*, Anaheim, California, August 1993, pp. 313–320.

[29] A. Glassner, editor, *An Introduction to Ray Tracing*, Academic Press, 1989.

[30] T. Guiard-Marigny, N. Tsingos, A. Adjoudani, C. Benoit, and M.-P. Gascuel, "3D models of the lips for realistic speech animation", *Computer Animation'96*, D. Thalmann and N. Magnenat-Thalmann, Eds, Geneva (Switzerland), pp. 80–89.

[31] E. Haines, "A Proposal for Standard Graphics Environments", *IEEE Computer Graphics and Applications*, Vol. 7, No. 11, November 1987, pp. 3–5. The SPD package is available at ftp.princeton.edu: /pub/Graphics/SPD

[32] J. Hart, "Ray Tracing Implicit Surfaces", in *Course Notes 25, SIGGRAPH 1993, Modeling, Visualizing and Animating Implicit Surfaces* pages 13.1–13.15.

[33] J. Hart, "Implicit formulations of rough surfaces", Proceedings of *Implicit Surfaces '95*, Eurographics Workshop, April 1995, pp. 33–44.

[34] J. Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces", *The Visual Computer*, 12(10), pp. 527–545, December 1996.

[35] J. Hart and B. Baker, "Implicit Modeling of Tree Surfaces", Proceedings of *Implicit Surfaces '96*, October 1996, pp. 143–152.

[36] J. Hart, "Guaranteeing the Topology of an Implicit Surface Polygonization for Interactive Modeling", *Computer Graphics (SIGGRAPH '97 Proceedings)*, August 1997, pp. 279–286.

[37] J. Hart, A. Durr and D. Harsh, "Critical Points of Polynomial Metaballs", Proceedings of *Implicit Surfaces '98*, June 1998, Seattle, pp. 69–76.

[38] D. Kalra and A. Barr, "Guaranteed Ray Intersection with Implicit Surfaces", *Computer Graphics (SIGGRAPH '89 Proceedings)*, Vol. 23, No. 3, July 1989, pp. 297–306.

[39] Y. Kawaguchi, "Growth/Mysterious Galaxy", SIGGRAPH '83 *Film & Video Show*, issue 11, 1985.

[40] Y. Kawaguchi, "Growth III: Origin", SIGGRAPH '85 *Film & Video Show*, issue 22, 1985.

[41] Y. Kawaguchi, "The Fantastic Self-Organization in Cyberspace", *Computer Graphics*, Vol. 31, No. 1, February 1997, pp.16–17.

[42] J. McCormack and A. Sherstyuk, "Creating and Rendering Convolution Surfaces", *Computer Graphics Forum*, Vol. 17, No. 2, 1998, pp.113–120.

[43] J. Menon, "An Introduction to Implicit Techniques", *SIGGRAPH Course 11*, August 1996.

[44] D. Mitchell, "Robust Ray Tracing with Interval Arithmetic", *Proceedings of Graphics Interface '90*, Canadian Information Processing Society, Toronto, 1990, pp. 68–74.

[45] S. Muraki, "Volumetric shape description of range data using "Blobby Model"", *Computer Graphics (SIGGRAPH '91 Proceedings)*, Vol. 25, No. 4, Las Vegas, Nevada, July 1991, pp. 251–257.

[46] F. Nichols and M. Stachels, editors, *Marine Life of the Indo-Pacific Region*, Periplus Editions, 1996.

[47] Ning P. and Bloomenthal J., "An Evaluation of Implicit Surface Tilers", *IEEE Computer Graphics and Applications*, November 1993.

[48] H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa and K. Omura, "LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation", *Proceedings of the Tenth International Symposium on Computer Architecture, ACM SIGARCH Newsletter*, Vol. 11, No. 3, 1983, pp. 387–394.

[49] M. Hirai, H. Nishimura, T. Kawata, I. Shirakawa and K. Omura, "Object Modeling By Distribution Function and an Efficient Method of Image Generation ", *Technical Report of TV Soc, IPD81-5*, 1983, pp.21–26, (in Japanese).

[50] H. Nishimura, M. Hirai, T. Kawai, T. Kawata, I. Shirakawa, and K. Omura, "Object Modelling by Distribution Function and a Method of Image Generation", *The Transactions of the Institute of Electronics and Communication Engineers of Japan*, 1985, Vol. J68-D, Part 4, pp. 718–725, in Japanese (English translation by Takao Fujuwara, Advanced Studies in Computer Aided Art and Design, Middlesex Polytechnic, England, 1989)

[51] A. Pasko, V. Pilyugin and V. Pokrovskiy, "Geometric Modeling in the analysis of trivariate functions", *Computers and Graphics*, Vol. 12, No. 3/4, 1988, pp. 429–446.

[52] A. Pasko, V. Adzhiev, A. Sourin, V. Savchenko, "Function representation in geometric modeling: concepts, implementation and applications", *The Visual Computer*, Vol. 11, No. 8, 1995, pp.429–446.

[53] H. Pedersen, "Decorating Implicit Surfaces", *Computer Graphics (SIGGRAPH '95 Proceedings)*, August 1995, pp. 291–300.

[54] K. Perlin and E. Hoffert, "Hypertexture", *Computer Graphics (SIGGRAPH '89 Proceedings)*, Vol. 23, July 1989, pp. 253–262.

[55] C. Pickover, "A Short Recipe for Seashell Synthesis", *IEEE Computer Graphics and Applications*, Vol. 9, No. 6, November 1989, pp. 8–11.

[56] *POV-Ray* , http://www.povray.org/

[57] W. Press, S. Teukolsky, W. Vetterling and B. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, second edition, Cambridge University Press, 1992.

[58] *Rayshade* , http://www-graphics.stanford.edu/~cek/rayshade/rayshade.html

[59] J. Schwarze, "Cubic and Quartic Roots", *Graphics Gems* (editor, Andrew S. Glassner), Academic Press, Cambridge, MA, 1990, pp. 404–407.

[60] G. Sealy and G. Wyvill, "Smoothing of three dimensional models by convolution", *Computer Graphics International '96 Proceedings*, June 1996, Pohang, Korea, pp. 184–190.

[61] *Shape Modeling and Computer Graphics with Real Functions*, http://www.u-aizu.ac.jp /public/www/labs/sw-sm/FrepWWW/F-rep.html

[62] A. Sherstyuk, *Ray tracing implicit surfaces: a generalized approach*, technical report No 96/290, Monash University, Dept. Computer Science, 1996.

[63] A. Sherstyuk, *Shells, Crabs and Seahorses: Expanding the Modeling Power of Implicit Surfaces*, technical report No 97/330, Monash University, Dept. Computer Science, 1997.

[64] A. Sherstyuk, "Ray-tracing with selective visibility", *Journal of graphics tools*, Vol. 1, No. 4, 1997.

[65] A. Sherstyuk, "Fast Ray Tracing Of Implicit Surfaces", *Implicit Surfaces'98*, Proceedings of the third international workshop on Implicit Surfaces, Seattle, USA, June 1998, pp. 145–153.

[66] J. Snyder and A. Barr, "Ray Tracing Complex Models Containing Surface Tessellations", *Computer Graphics (SIGGRAPH '87 Proceedings)*, Vol. 21, No. 4, July 1987, pp. 119–128.

[67] A. Sourin, A. Pasko, V. Savchenko, "Using real functions with application to hair modelling", *Computers and Graphics*, Vol. 20, No. 1, 1996, pp. 11–19.

[68] A. Sourin and A. Pasko, "Function Representation for Sweeping by a Moving Solid", *IEEE Transactions on Visualization and Computer Graphics"*, Vol. 2, No. 1, March 1996, pp. 11–18.

[69] B. Stander and J. Hart, "Guaranteeing the Topology of an Implicit Surface Polygonization for Interactive Modeling", *Computer Graphics (SIGGRAPH '97 Proceedings)*, August 1997, pp. 29–33.

[70] H. Tuy and L. Tuy, "Direct 2-D Display of 3-D Objects", *IEEE Computer Graphics and Applications*, Vol. 4, No. 10, October 1984, pp. 29–33.

[71] A. Watt and M. Watt, *Advanced Animation and Rendering Techniques*, the 2nd edition, ACM Press, Addison-Wesley Publishing Company, 1993.

[72] S. Wolfram, *The Mathematica Book*, Third edition, Wolfram Media, Inc. and Cambridge University Press, 1996.

[73] G. Wyvill, C. McPheeters and B. Wyvill, "Data Structure for Soft Objects", *The Visual Computer*, Vol. 2, No. 4, 1986, pp. 227–234.

[74] B. Wyvill, "SOFT", SIGGRAPH '86 *Electronic Theater and Video Review*, issue 24, 1986.

[75] G. Wyvill, C. McPheeters and B. Wyvill, "Animating Soft Objects", *The Visual Computer*, Vol. 2, No 4, Aug. 1986, pp. 235–242.

[76] G. Wyvill, B. Wyvill and C. McPheeters, "Solid Texturing of Soft Objects", *IEEE Computer Graphics and Applications*, Vol. 7, No. 12, December 1987, pp. 20–26.

[77] B. Wyvill, "The Great Train Rubbery", SIGGRAPH '88 *Electronic Theater and Video Review*, issue 26, 1988.

[78] B. Wyvill and G. Wyvill, "Field functions for implicit surfaces", *The Visual Computer*, Vol. 5, No. 1/2, 1989, pp.75–82.

[79] G. Wyvill and A. Trotman, "Ray Tracing Soft Objects", *CG International 90*, Springer Verlag, pp. 469–475, 1990.

[80] B. Wyvill and K. van Overveld, "Tiling Techniques for Implicit Skeletal Models", in *Implicit Surfaces for Geometric Modeling and Computer Graphics, SIGGRAPH 1996, Course Notes*, pp. C1.1–C1.26.

[81] R. Zonenschein, J. Gomes, L. Velho, and L. H. de Figueiredo, "Controlling Texture Mapping onto Implicit Surfaces with Particle Systems" Proceedings of *Implicit Surfaces '98*, June 1998, Seattle, pp. 131–139.

"If anybody wants to clap,"
said Eeyore when he had read this,
"now is the time to do it."
They all clapped.
"Thank you,"
said Eeyore.
"Unexpected and gratifying, if a little lacking in Smack."

"It's much better than mine,"
said Pooh admirably, and he really thought it was.
"Well,"
explained Eeyore modestly,
"it was meant to be."

– Eeyore from "The House at Pooh Corner"
by A. A. Milne