

# 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  $F : R^3 \rightarrow R$ , 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

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

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

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

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

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

or simply

$$f(t) = 0 \quad (4.4)$$

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  $\lambda$  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  $\nabla f$  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:

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

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  $[t_1, t_2]$ ,  $i = 1, \dots, k$ .

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:

$$f(\mathbf{r}) = \begin{cases} 1 - 3(\frac{r}{R})^2 & 0 \leq r \leq \frac{R}{3}; \\ \frac{3}{2}(1 - \frac{r}{R})^2 & \frac{R}{3} < r \leq R; \\ 0 & r > R; \end{cases} \quad (4.6)$$

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 gen-

erated 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

$$\max |f(t) - p(t)| < \epsilon, \quad t_1 \leq t \leq t_2 \quad (4.7)$$

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

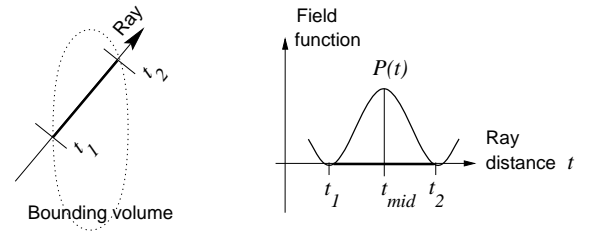


Figure 4.1: Polynomial approximation over  $[t_1, t_2]$ .

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

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

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

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

$$\begin{aligned} f_1 &= 0, \\ f'_1 &= 0, \\ f_{mid} &= f(t_{mid}), \\ f'_{mid} &= f'(t_{mid}) \end{aligned} \quad (4.9)$$

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 non-zero. 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

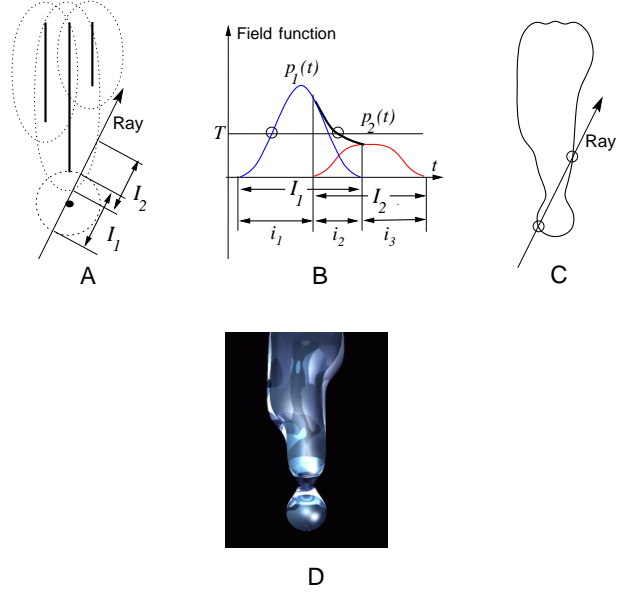


Figure 4.2: Rendering an implicit icicle: (A) intersection with bounding volumes; (B) interpolation of contributing components by polynomials  $p_i(t)$ ; (C) isosurface at level  $T$ ; (D) shaded image.

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

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

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):

$$\sum_{i=1}^N w_i = T \quad (4.11)$$

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

$$C = \sum_{i=1}^N w_i C_i \quad (4.12)$$

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<sup>1</sup>.

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  $x_1, x_2, \dots, x_n$  the error is

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

where  $L_n(t) = \prod_{i=1}^n (t - x_i)$  and point  $\xi$  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-

<sup>1</sup>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).

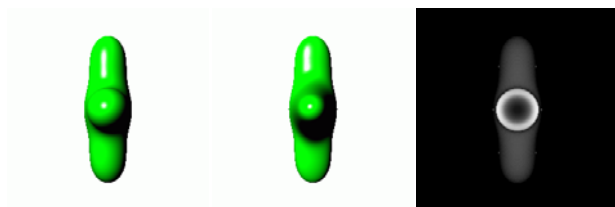


Figure 4.4: The side view. The error is substantially higher in the central areas, where rays travel along the horizontal line segment, yielding longer extents  $[t_1, t_2]$ . The surface becomes deformed. Rendering time: 20 sec (numerical) and 13 sec (analytical).

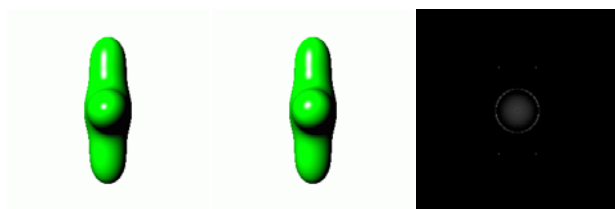


Figure 4.5: The side view, revisited. The error is reduced by using four sub-intervals and four Hermite interpolants per line segment. The left and middle images are practically identical. Rendering time: 20 sec and 17 sec.

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

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

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  $i_1, \dots, i_n$  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  $i_1, \dots, i_n$ . 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.

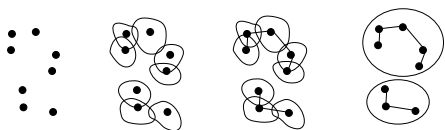


Figure 4.6: Creating permanent clusters: originally chaotic implicit primitives (left) are organized into a graph using their bounding volumes (middle left) which is then searched for connected components (middle right). The connected components form two permanent clusters (right). Black dots show the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

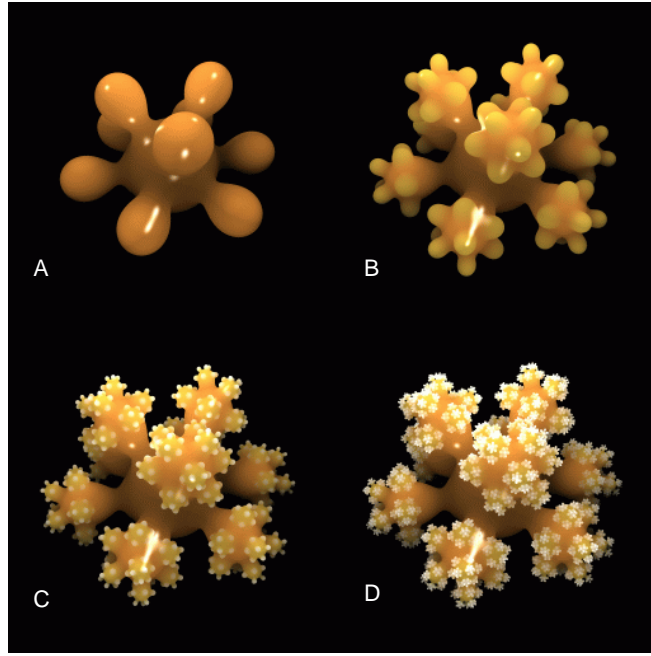


Figure 4.7: Implicit sphere-flakes made with 10, 91, 820 and 7381 Gaussian density functions. The smaller objects have lighter surface colors for better contrast.

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 Sphreflake model.

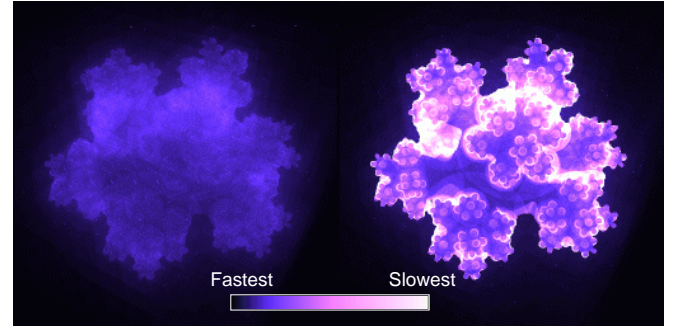


Figure 4.8: Time-profiling charts for our algorithm (left) and the *LG*-algorithm (right). The left image shows almost uniform time complexity over the scene. The right image shows that the rendering was much slower at the silhouette edges, which agrees with the results reported by Kalra and Barr [38]. 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  $\epsilon$  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]<sup>2</sup>. There are 31 ‘branches’ and 512 ‘needles’ in the dataset. The left image is rendered with conventional primitives in 9 min 36 sec;

<sup>2</sup>This image is created with a Gaussian kernel. Compare with a coral tree in Fig. 2.20, which was made with a Cauchy kernel.



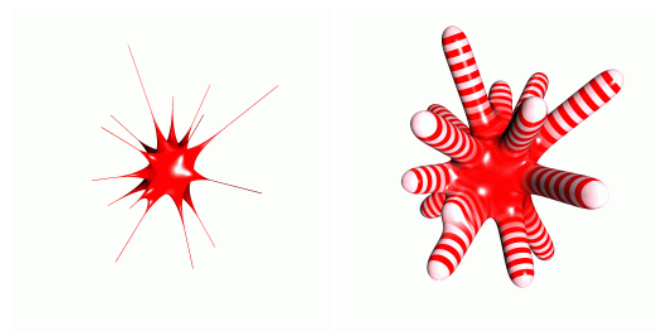


Figure 4.9: Two sea-urchins. Left: thin objects are not a problem for the rendering algorithm. Right: *UV*-mapping, cross-dissolved over blending regions.

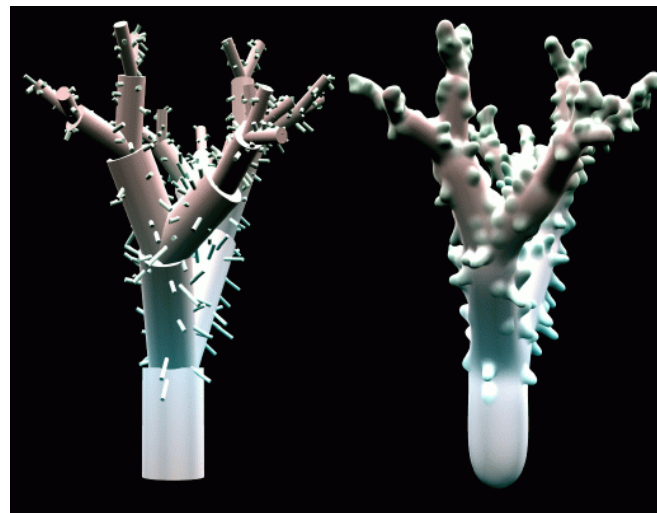


Figure 4.10: Coral tree: offset surface (left) and smoothed implicit surface (right). This model served as a test-bed for various speed optimizations discussed in Section 4.4. Number of elements 548.

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<sup>3</sup> as described in [22]. The upper image is rendered by *Rayshade*, a public domain ray-tracer [58], widely used in computer

<sup>3</sup>The author's original spelling is preserved.



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.

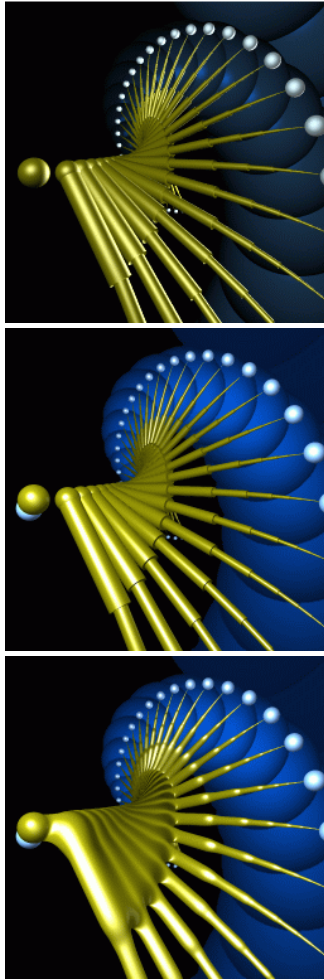


Figure 4.12: *RATS* renders the convolution surfaces of “Spinal Starecase” (bottom) faster than Rayshade does it for the conventional version of the model, represented by cylinders (top). The middle image is also rendered by *RATS*. Total number of elements: 478. Conventional model (top and middle) contains 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*

