THE UNIVERSITY OF CALGARY

Efficient Rendering of Animated *BlobTrees*

by

Mark A. Fox

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

August, 2001

© Mark A. Fox 2001

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-65101-0

Canada

Abstract

Implicit surfaces are a powerful modelling primitive for computer graphics. The *BlobTree* is a convenient data structure for representing implicit fields and surfaces. As features have been added to the *BlobTree*, it has become more useful for modelling, and therefore the models it has been used to express have grown larger and more complex. The growth of *BlobTree* models has made it obvious that the time needed to visualize large models is directly proportional to the size of the *BlobTree* used to describe a model. The *BlobTree* also suffers from an inability to include time-dependent information, thus making animation difficult. This thesis is a presentation of two techniques, reduction and pruning, that can substantially reduce the cost of rendering large and complex models. Additionally, the extension of both techniques to animated *BlobTrees* is explored.

Preface

The research summarized in this thesis builds on the work of past contributors to *The Graphics Jungle*. The implementation of the *BlobTree* system, which is described in Chapter 3, is derived from the work of Dr. Brian Wyvill, Andy Guy, and Mark Tigges. Additionally, the task of reimplementing the *BlobTree* system was shared between myself and Callum Galbraith, with some help from Robson Lemos, Xikun Liang, Mai Nur, and Dr. Wyvill.

Much of the research presented in this thesis has been published by myself, Callum Galbraith, and Dr. Wyvill in both The Western Computer Graphics Symposium [16] and Shape Modelling International [17].

The idea of applying tree-normalization, transformation compaction, and spatial pruning, to the *BlobTree* stemmed from discussions with Dr. Geoff Wyvill at the University of Otago, New Zealand.

The implementation of *pruning* and *reduction* in the *BlobTree* system was done by myself, but the result gathering and analysis was shared between myself and Callum Galbraith.

Acknowledgements

Thanks to my supervisor, Brian Wyvill, for his endless enthusiasm. His fascination with implicits has been contagious. Thanks to Geoff Wyvill, for unselfishly sharing ideas on research, all while encouraging me to enjoy New Zealand. Many of the past and present members of the Jungle Research Group (Callum Galbraith, Cathy Jirasek, Pavol Federl, Brendan Lane, Lars Muendermann, Mai Nur, Radek Karwowski, and Mark Tigges) deserve credit for providing both academic, and social support, in addition to wonderful lunch-time conversation. Financial support from the National Science and Engineering Research Council helped keep me focused on my research rather than on the funding of my research. My parents and family deserve a special acknowledgement for infusing me with the will to undertake higher education. Lastly and most of all, for standing by me through the two toughest times of my life, Carmen.

Contents

List of Figures

CHAPTER 1

Introduction

Implicit surfaces are useful for modeling interesting and practical shapes. Data-structures like the *BlobTree* provide the foundation of an integrated system for modeling with implicit surfaces. Unfortunately, rendering large and complex models constructed using the *BlobTree* is slow due to the expense of *BlobTree* field evaluations. This research examines methods of exploiting spatial and non-spatial information in a *BlobTree* model to increase the speed with which it can be evaluated and therefore rendered.

1.1 Explanation and Motivation

Computer graphics software can be used in a variety of areas including but not limited to education, mathematics, engineering, and entertainment. The field has developed to the point where a motivated individual can begin to make simple, useful, and perhaps even interesting images and animations of three-dimensional objects with only a few thousand dollars worth of computer hardware and software, and only a few days of training.

The rapid increase in computational power available to computer scientists has driven advances in the field. The increase in computational power has made possible novel algorithms for expressing, generating, and manipulating graphical information.

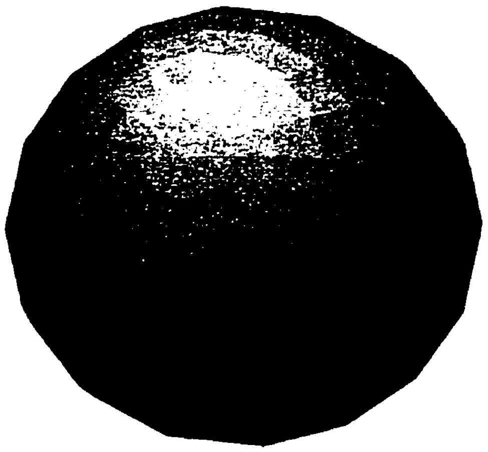


Figure 1.1: **A sphere approximated by nearly 500 triangles.**

The integration of these algorithms into software tools has greatly expanded the flexibility of the computer as a tool for computer graphics.

The increase in selection and power of modeling primitives is one example of this flexibility. A *modeling primitive*, or just *primitive*, is the smallest unit used to model a three-dimensional object. For example, by using a deck of playing cards, one could model a house. In this case, the model is a house, and the modeling primitive used to represent it is the playing card. In computer graphics, the most pervasive modeling primitive is the *triangle*.

Triangles are well suited to modeling objects that are made up of flat faces, such as boxes and simple cabinets. However, objects with curved surfaces, spheres for example, which are modeled with triangle primitives suffer due to the fact that curved surfaces can only be approximated with triangles [15]. Figure 1.1 shows a model of a sphere constructed from nearly 500 triangles. The fact that the model is only an approximation of a sphere is obvious. To generate an acceptable approximation of a curved surface, a large number of triangles must be used. A consequence of using a large number of triangles is that the data representing the object becomes cumbersome for the user to manipulate, and for the computer to process.

One strategy for solving this problem is through the use of a more powerful prim-

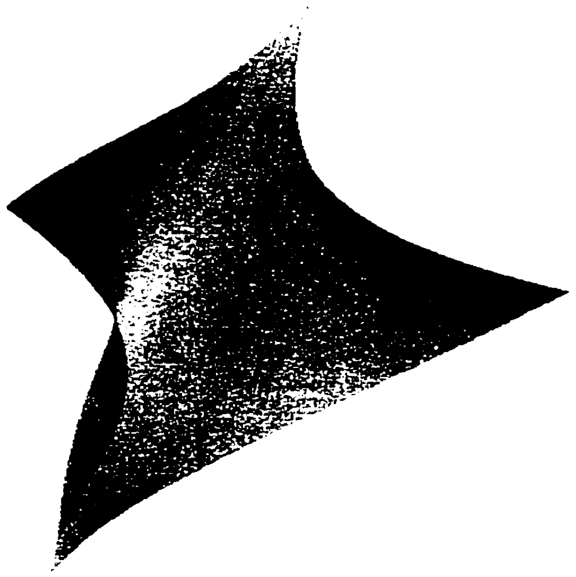


Figure 1.2: **A single Bézier primitive.**

itive than the triangle. In this context, more powerful could be taken to mean; able to represent curved surfaces accurately. The set of primitives based on *spline patches* fulfills this requirement.

For the purposes of illustration, a reasonable physical analog to the spline patch is a rectangular piece of thin and pliable sheet metal. By applying pressure to the metal at points across its surface, it can be bent into a smoothly curving surface. In this respect a spline patch is similar, by moving a set of control points, a smoothly curving surface can be manipulated.

The Bézier spline patch in Figure 1.2 is described by only sixteen points. An approximation of the same surface with triangle primitives would require several hundred points dependinging on the accuracy required. This reduction in data through the use of more powerful primitives results in a substantial increase in the expressive power of both the artist and the computer.

Unfortunately, even with spline patches, there is still a point where the amount of data necessary to represent a model can become too much both for the artist to easily understand and the computer to easily process. If it is considered that a convincing representation of the human form could easily require several hundred spline patches, it becomes obvious that a more powerful primitive is necessary.

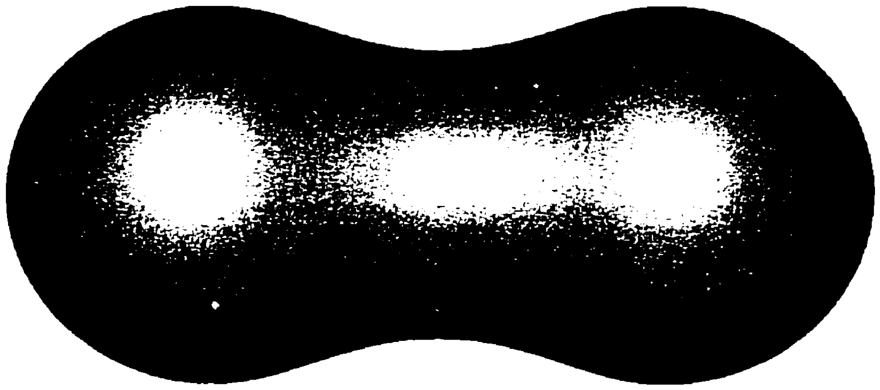


Figure 1.3: **A ray traced Peanut.**

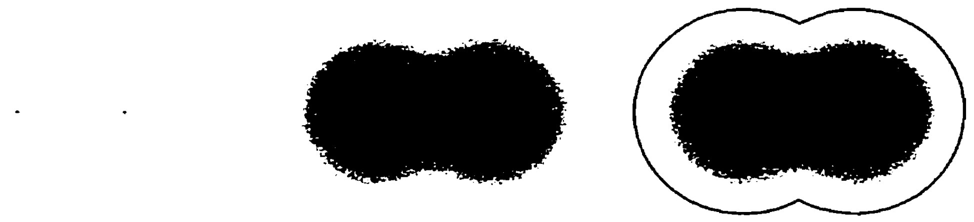
Using a type of modeling primitive called an *implicit surface*, the curved surface in Figure 1.3 is entirely represented by only two points. The same surface would require substantially more data if built with spline-patches or approximated with triangles.

An implicit surface is dependent on the notion of a *field*. A field simply assigns a value to points in space. For example, a field could be defined that assigned all points in space with their distance, in kilometers, from the tip of the Eiffel Tower. This field would assign all of the points that are 1 kilometer from the Eiffel tower with the field value 1.

Given a field, an implicit surface can be defined by stating that all the points with a specific field value are on the surface. The field value that defines the surface is called the *surface value*. For example, using our Eiffel Tower field, if we define the surface value to be 1, then we would have a description of a sphere, centered on the tip of the Eiffel Tower, with a radius of 1 kilometer.

The link between fields and implicit surfaces is illustrated in Figure 1.4. Figure

1.4(a) displays two points which are used to define the field shown in Figure 1.4(b). The inner contour in Figure 1.4(c) corresponds to the part of the field that has a value of 0.5, while the outer contour bounds the area where the field is non-zero. Figure 1.4 is intimately linked to the image in Figure 1.3, since both figures are generated from the same simple field.



(a) The points which define a field

(b) The field defined by the points

(c) The $c = \frac{1}{2}$ contour (inner curve) and the bound of the field's non-zero area (outer curve).

Figure 1.4: **The link between a field and an implicit surface.**

A computer and a software tool can be used to define and visualize fields and implicit surfaces which depend on those fields. Those implicit surfaces can then be used to model real world or imaginary objects through the computer. The *BlobTree* is one such software tool for modeling objects with implicit surfaces. The *BlobTree* allows a user to flexibly build complicated fields from simple fields. These complicated fields can then be used to describe complicated surfaces which can be used to build models of objects. The *BlobTree* used to express the field for Figures 1.4 and 1.3 is depicted in Figure 1.5.

Unfortunately, complicated models based on implicit surfaces, although they require relatively little data, require a great deal of computation on the part of the computer to produce an image. For example, using well known algorithms for rendering implicit surfaces, the image in Figure 1.6, which depicts a model of a sea shell built with the *BlobTree*, took approximately 28 hours of computation to generate.

For production purposes, this is almost certainly unacceptable.

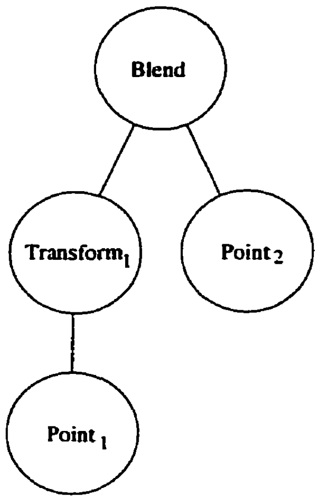


Figure 1.5: **The *BlobTree* used to define the peanut model for 1.3 and 1.4.**

For animation purposes, it is necessary that the *BlobTree* also incorporate information on how the model changes over time. The additional computation required to calculate the status of a model at a particular instant of time can add considerably to the computation time required to render an image. Since animations are made up of many individual images, the computation time to generate an animation is many times the that needed to generate a single image. Still worse, in order to simulate the look of film, images in an animation may require the generation of motion blur. In terms of computer time, the computation required to generate motion blurred images is expensive. The desire to create high-quality animations of complex *BlobTree* models demands that the software used to render images of *BlobTrees*, and the *BlobTree* software itself, be as efficient as possible.

For users of the *BlobTree* software, once they have expressed their models, how their models move and change over time, and described the position and motion of any light sources and the camera, they must let the computer render the images. The rendering time is dependent on many factors, but the most important is the complexity of the model. A model of an object demands a certain amount of complexity.

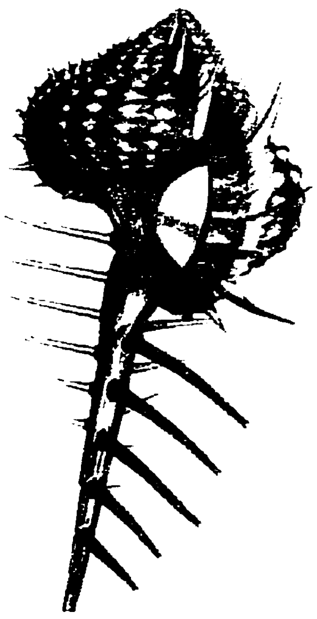


Figure 1.6: **A complex model of the sea-shell *Murex cabritii* built using the *BlobTree*. (Image courtesy of Callum Galbraith.)**

For *BlobTree* models the complexity is proportional to the number of nodes in the model. For example, the *BlobTree* used to express the peanut model used for Figures 1.3 and 1.5 has four nodes. A more complex model is depicted in Figure 1.6.

It is instructive to note that, using conventional *BlobTree* rendering algorithms, the more complex model takes approximately four-thousand times as long to render as the simple model. In fact, rendering time, T, can be expressed by the mathematical equation $T = cn$, where c is a constant, and n is the number of nodes in the *BlobTree* model. This formula implies that doubling the complexity of a *BlobTree* model will double the rendering time. This is acceptable, but could certainly be improved upon.

A *BlobTree* rendering algorithm that yields an expression for T that is less dependent on n could yield an image much more quickly. For example, if the algorithm had $T = c$, then no matter what the complexity of the *BlobTree* model, the rendering time would remain the same. This is an impossibility for a program running on a serial computer, but it is certainly possible to reduce the size of c. For the purposes of this thesis, a reduction of T by a factor of $\frac{1}{4}$ would be significant. It would also be considered significant if rendering time could be reduced to something like $T = c\log(n)$.

The work presented in this thesis is focused on methods for modifying the *BlobTree*, and extending algorithms that make use of the *BlobTree*, to significantly reduce the computational expense of rendering images and animations of *BlobTree* models.

1.2 Historical Survey

Although implicit surfaces have been used in other areas for some time, in 1973 Ricci [35] is the first author to use them in the context of computer graphics, and provides an extensive introduction to the theory of implicit surfaces along with some simple algorithms for their visualization. Ricci's work is revisited and somewhat expanded by Blinn in 1982 [4]. Despite the attention of Blinn, a very respected computer graphics researcher, implicit surfaces remained unpopular, largely due to the expense of their visualization.

In 1985, Nishimura [33] used quadratic equations to express implicit surfaces. This allowed for high-quality rendering of the implicit surfaces using the ray tracing techniques developed in [25], which were widely known at the time. Still, the computer graphics community remained largely ignorant of implicit surfaces.

An algorithm for converting an implicit surface to triangles, and data structures for representing and efficiently computing the implicit field value, is detailed in [58]. Two short animations, [48, 49], which made use of implicit surfaces for several purposes, were released in 1986 and 1988, respectively. These two animations made the case for implicit surfaces in computer animation.

General and accurate algorithms for rendering implicit surfaces are detailed by Kalra in 1989 [29], and by Mitchell in 1990 [30]. Both techniques allow a great deal of flexibility in the choice of functions, while still guaranteeing that fine details in the surface will be captured.

With the publishing of techniques for synthesis of photo-realistic images of implicit surfaces, techniques for real-time image synthesis still remained undiscovered. The use of oriented particle systems by Witkin in 1994 [46] made real-time visualization of, and interaction with, implicit surfaces possible.

Although the use of tree data structures is prevalent in computer science and for computer graphics, the application of trees to the definition of fields was not published until 1999 [51]. The data structure outlined, called the *BlobTree*, incorporates affine transformations, non-linear spatial warps as in [3], blending, and boolean operations into a single unified data structure. That data structure is an ancestor of the one detailed in this thesis.

The *BlobTree* has many similarities to scene-graphs which are used to represent traditional surface models [36]. Many techniques exist for more efficient rendering, via ray tracing, of scene-graphs [14, 15, 45]. Many of these techniques are dependent on the idea of spatial partitioning [20]. Spatial partitioning has also been applied to the simplification of Constructive Solid Geometry (CSG) trees [47, 57, 22]. Spatial partitioning has been applied to implicit surfaces [58, 55, 19], but the underlying representations of implicit surfaces were very simple, and not as general as the *BlobTree*.

Another long standing problem with implicit surfaces was the inability to apply two-dimensional texture maps. An algorithm for applying texture maps to implicit surfaces is detailed in [41]. This method relies on repeated evaluations of the function and is therefore expensive. An alternative approach, which is largely decoupled from field evaluation, and therefore much more efficient, is outlined in [42].

The *BlobTree* was integrated with a visual modeling tool and later integrated with the *Python* programming language for procedural modeling [43]. The *BlobTree* and its suite of libraries have much in common with *HyperFun*, an extensive system which makes use of F-reps [34], a representation of implicit surfaces which is analagous to the *BlobTree*. The *HyperFun* tools are described in [1].

1.3 Outline of Contributions

A major part of this research is the design and implementation of a complete system for modeling and rendering of animated implicit surfaces. This system includes, *JungleGL*, a foundation graphics library, *JungleBT*, the *BlobTree* implementation, *JungleWyvillPG*, a polygonizer for *BlobTrees* based on [58], and *JungleRT*, an implicit surfaces ray tracer based on a simplification of the techniques in [30]. The system includes approximately fifty-thousand lines of C++ code, and represents a major effort by the graduate students in *The Graphics Jungle*.

The analysis and presenation of techniques for making *BlobTree* implicit field evaluations more efficient is presented in this thesis. By making *BlobTree* field evaluations more efficient, the use of these techniques will in turn make any algorithm that depends on these evaluations more efficient. The integration of these techniques with ray tracing and polygonization is detailed. Preliminary work for the integration of these techniques with animated *BlobTrees* is also presented.

Spatial pruning and its application to the *BlobTree* for spatial partitioning is introduced in this thesis. All of the features of the *BlobTree* are discussed as well. Special attention is given to the *BlobTree* features, such as CSG and warps, that complicate pruning.

Another technique for increasing the efficiency of scene-graphs, called *reduction*, is adapted to the *BlobTree*. By applying reduction to an instance of a *BlobTree* any redundant information stored in affine transformations is removed, resulting in a *BlobTree* that is more efficient to evaluate. A complication to reduction, caused by the use of warps in *BlobTrees*, along with its solution is presented.

The concepts of pruning and reduction were integrated into *JungleBT*. Both the ray tracer and polygonizer have been extended to take advantage of the two techniques. The fact that the *BlobTree* has recently been extended to support animation has caused significant complications to the integration of pruning and reduction to the *BlobTree* system. Preliminary solutions to the problem of pruning and reduction in the context of animated *BlobTree* were devised and are described in this thesis.

1.4 Thesis Overview

The focus of this research is methods for making evaluations of the *BlobTree*, and therefore rendering, more efficient. After the initial introduction and background to implicit surfaces, except where noted, the implicit surfaces system discussed will be the *BlobTree*.

The body of this document is organized as follows:

- Chapter 2 is an introduction to the theory of implicit surfaces and the algorithms that can be used for their visualization. The techniques for building interesting surfaces from simple primitive fields are explained, and several different algorithms for both ray tracing and polygonization of implicit surfaces are discussed.

- Chapter 3 details the *BlobTree*, the implicit surfaces system to which the techniques presented in this thesis are applied. All the features and operations of the *BlobTree*, including skeletal primitives, blends, controlled blends, affine transformations, spatial warps, constructive solid geometry, and attributes are examined in detail.

- Chapter 4 presents reduction and pruning, the techniques used to make the *BlobTree* more efficient. Explanations of how these techniques are applied to ray tracing and polygonization are also given.

- Chapter 5 presents the results of both ray tracing and polygonization of several test *BlobTree* models with pruning and reduction applied. Different parameter values for the application of pruning are explored in an attempt to determine appropriate values for the different test models. Trends in the results are pointed out and a discussion why these trends occur is also included.

- Chapter 6 summarizes and concludes the findings and contributions of this research. A discussion of ideas for future work is also given.

CHAPTER 2

Fundamentals of Implicit Surfaces

Implicit surfaces are dependent on the notion of a field function $F : \mathbb{R}^3 \rightarrow \mathbb{R}$. An implicit surface is defined to exist at the set of points where $F(\vec{p}) = c$, where c is some constant. Given a function, F, and a value for, c, the definition of the surface is complete, and it can be visualized directly by rendering algorithms such as ray tracing. Alternatively, it can first be polygonized and the resulting triangles visualized by traditional methods for polygons.

2.1 Fields for Implicit Surfaces

F, the field function simply assigns values to points in space. There is no constraint on how F is defined, with the exception that for the purposes of rendering an implicit surface, it is desirable that it be both continuous and computationally inexpensive to evaluate. The rendering algorithms assume that F is continuous. If that is not the case, then they will fail. The rendering algorithms also evaluate F a large number of times. If F is not computationally inexpensive, then the rendering algorithms will be unacceptably slow.

There is a great deal of freedom in defining F. For example, by using the function $F(\vec{p}) = p_x^2 + p_y^2 + p_z^2 - r^2$ the surface of a sphere of radius r can be defined to exist where $F(\vec{p}) = 0$. As stated in [6], a torus of inner radius r and outer radius R can be defined with the implicit equation $F(\vec{p}) = (p_x^2 + p_y^2 + p_z^2)^2 - 4R^2(p_x^2 + p_y^2) - r^2$. The surface of the torus exists where $F(\vec{p}) = 0$. Similar functions can be defined for many of the standard shapes used in computer graphics.

2.1.1 Convenient Construction of Fields

Although modeling a shape by deriving a formula may be attractive to mathematicians, it is probably abhorrent to all but the most dedicated of artists. Blinn presented an approach that allows much more intuitive modeling in [4]. Given a set of n points, Q, a simplified version of Blinn's field function is $F(\vec{p}) = \sum_i^n e^{-|\vec{p}-Q_i|^2}$. This field function allowed Blinn to create visualizations of approximations to the electron fields surrounding molecules, where the nuclei of the atoms were the set of points in Q. By breaking Blinn's function down into its constituent parts the notions of *blending* and *skeletal elements* can be derived.

Blinn's field function, F, is the sum of several exponential functions. The exponentials are functions of the Euclidean distance between the query point, $\vec{p}$, and another point Q_i. Let us define any geometric element, whose minimum distance to a point in space can be found, as a skeletal element. Then, in Blinn's field function, the set of points, Q, is a set of skeletal elements. Since Blinn uses points, the minimum distance between a skeletal point element, Q_i, and the query point, $\vec{p}$, is simply $|\vec{p} - Q_i|$.

Skeletal elements allows the use of a variety of geometric entities as the foundation of skeletal elements. For example, we could use a line segment as a skeletal element. In this case, the minimum distance between $\vec{p}$ and a segment of a line would be the minimum of the distances between $\vec{p}$ and the closest point on the line, and between $\vec{p}$ and the two ends of the line segment. This is illustrated in Figure 2.1.

The minimum distance, d, between the query point, $\vec{p}$, and the skeletal element is then used as a parameter to an exponential function $f(d) = e^{-d^2}$. This yields a three-dimensional field function that surrounds the skeletal element. The shape of

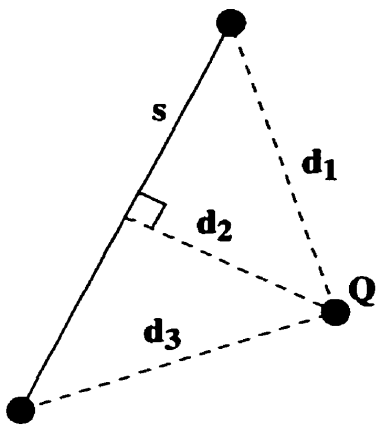


Figure 2.1: **The minimum distance between a point, $\vec{p}$ and a line segment, is the minimum of the distances, d_1, d_2, and d_3. In this case $dist(s, Q) = d_2$.**

f determines the effect that the summations of the fields surrounding two skeletal elements will have on the implicit surface. To illustrate this, a plot of the function and the resulting implicit surface, at $c = \frac{1}{2}$, for Blinn's exponential function and a linear function is given in Figures 2.2 and 2.3, respectively.

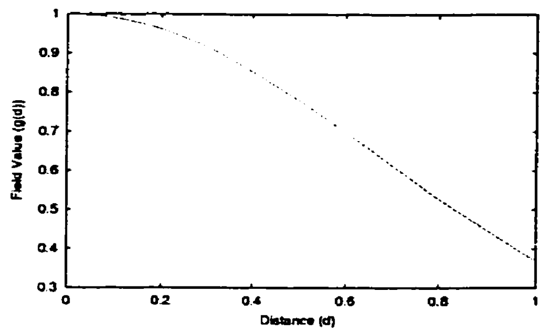
Most algorithms for rendering implicit surfaces assume that the field function, F, is continuous as well. These algorithms will fail if F is not continuous. For this reason, definitions of f should be continuous. Many different definitions of f have been used in the past, and several are mentioned in [53, 7].

Since we desire the flexibility to use different skeletal elements and functions, we can write a more generalized form of our implicit function. Given a set of functions, G, a set of skeletal elements, S, and a function, $dist(s, \vec{p})$, which will return the minimum distance between $\vec{p}$ and a skeletal element, s, we can define F to be:

$$F(\vec{p}) = \sum_i g_i(dist(s_i, \vec{p})) \tag{2.1}$$

This equation allows a great deal of freedom in specifying the field function since a variety of both skeletal elements and field functions can be used.

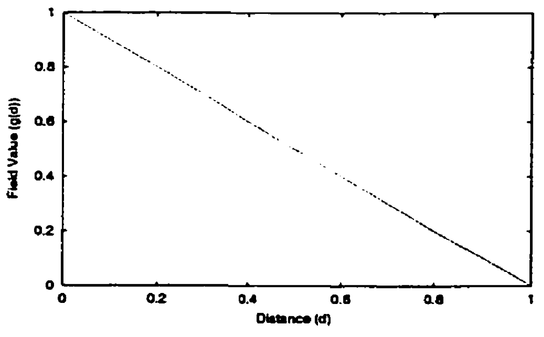
Let us call the combination of a skeletal element and its associated field function a



(a) Blinn's exponential field function.

(b) The $c = \frac{1}{2}$ surface resulting from 2.2(a).

Figure 2.2: **The Blinn function and the resulting implicit surface.**



(a) A linear field function.

(b) The $c = \frac{1}{2}$ surface resulting from 2.3(a).

Figure 2.3: **A linear function and the resulting implicit surface.**

primitive field. The summation of these primitive fields can then be defined as simple additive *blending*. This allows a complex field to be created by the blending of simple primitive fields.

Given an appropriately shaped field function, the one depicted in Figure 2.2(a) for example, blending is an intuitively accurate term, since the surface that results from the adding of several overlapping fields does indeed look like the smooth blend of volumes.

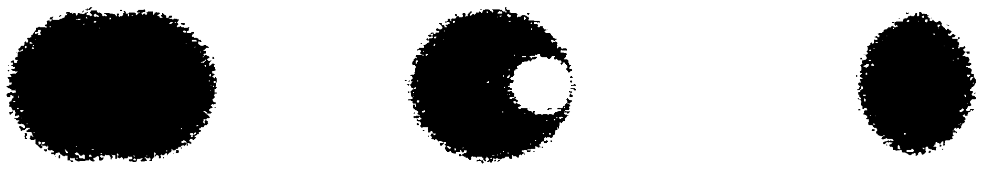
An alternative to the simple additive blending operation is the super-elliptic blend as defined by Ricci [35]. This operation is more flexible in that it allows a parameter to smoothly vary the type of blend from a simple additive blend to a union operation.

2.1.2 CSG in Implicit Surfaces

Blending simple primitive field functions is an intuitive way of building and describing organic looking models with soft flowing curves. Unfortunately, most man-made objects cannot be easily described in this way. This is because most man-made objects are built by using one object, a drill bit for example, to cut a piece out of another object, or by fastening two objects together. Put differently, man-made objects are built by subtracting one object from another or adding one object to another.

Constructive Solid Geometry (CSG) provides a way of expressing man-made models in the same way that their physical analogs would be engineered. CSG provides three operations for building models: *union*, *difference* and *intersection*. The *union* of two circles is the set of points contained by at least one of the circles. The *difference* of two circles is the set of points inside the first circle, but not the second. The *intersection* of two circles is the set of points that are inside both circles. The *union*, *difference*, and *intersection* of two circles is depicted in Figure 2.4.

As pointed out by Ricci, CSG operations can be easily expressed in the context of fields and surfaces [35]. Given two field functions $f_1(\vec{p})$ and $f_2(\vec{p})$, a function that will yield the union of the two implicit surfaces can be defined as: $U(\vec{p}) = max(f_1(\vec{p}), f_2(\vec{p}))$,



(a) The union of two fields. (b) The difference of two fields. (c) The intersection of two fields.

Figure 2.4: **The CSG operations applied to two implicit circles and their defining fields.**

where $max(n_1, n_2)$ yields the maximum of n_1 and n_2. Similarly given that $min(n_1, n_2)$ yields the minimum of n_1 and n_2, difference and intersection can be defined as $D(\vec{p}) = min(f_1(\vec{p}), 2c - f_2(\vec{p}))$ and $I(\vec{p}) = min(f_1(\vec{p}), f_2(\vec{p}))$, respectively. Each of Figures 2.4(a), 2.4(b), and 2.4(c) display the $c = \frac{1}{2}$ implicit surface resulting from the above definitions, in addition to the underlying field.

The CSG operations can also be expressed as R-functions [34]. R-functions have the advantage that they can exhibit higher degrees of continuity than simple CSG operations based on *min* and *max* functions, which are first-degree discontinuous by definition. The implementation of CSG via these R-functions does have the disadvantage of having a higher computational cost.

2.2 Visualization of Implicit Surfaces

The combination of ***skeletal primitives***, ***blending***, and CSG, allows many useful models to be simply and intuitively expressed using a tree structure. Of course, the field of computer graphics demands the existence of a way of visualizing the resulting models. Two standard rendering methods for visualizing implicit surfaces are polygonization, which approximates the implicit surface with polygons and renders the result using standard techniques for rendering polygons [7], and direct ray tracing of implicit surfaces [25, 27, 29, 30].

Polygonization has the advantage of having a much smaller computational expense relative to direct visualization methods. Additionally, a polygonization is independent of view direction, making it appropriate for interactive visualization.

Polygonization has several disadvantages. It can potentially create a massive amount of data. Even though that data may be entirely unnecessary due to the orientation of the camera. Since polygonization generates a piecewise planar approximation to a potentially curved surface, unacceptable visual artifacts may result.

Although much more expensive computationally, direct methods for visualizing implicit surfaces do not suffer from the inherent artifacts of a polygonal approximation. Direct visualization has the advantage that it requires much less data than polygonization to create an accurate image [29]. These facts combine to make direct visualization the method of choice for the generation of high-quality images that accurately depict implicit surfaces, and indirect visualization methods the choice for interactive applications.

2.2.1 Indirect Visualization of Implicit Surfaces

As far as traditional visualization methods are concerned, an implicit surface cannot be visualized directly. If traditional methods are to be used an implicit surface must first be converted to a form that can be directly visualized. Polygonizing of an implicit surface approximates it by a set of planar polygons, which can then be directly visualized.

Most implicit surface polygonization algorithms are based on the fact that $F(\vec{p}) > c$, if $\vec{p}$ is inside of the surface and $F(\vec{p}) < c$, if $\vec{p}$ is outside of the surface. Therefore if a point, $\vec{a}$, is inside, and another point $\vec{b}$ is outside, the implicit surface must intersect the line-segment between $\vec{a}$ and $\vec{b}$ an odd number of times.

Given a cube in the same space as the implicit surface, the individual vertices of the cube can be labelled as inside or outside the implicit surface. If the vertices are not all inside or all outside the surface, then the surface must intersect the cube. Any edge of the cube with one end inside and the other outside the surface intersects the implicit surface an odd number of times between its end-points. To simplify the problem, most algorithms assume that "an odd number of times" means "exactly one time". For cubes of sufficiently small size this assumption is true. A detailed discussion of this issue is given in [44].

So, by examining the inside-outside value of the vertices of a cube, it can be determined which edges, if any, intersect the implicit surface. Depending on the accuracy needed, the point on an edge which intersects the surface can be accurately determined via a numerical root-finding approach, or when accuracy is not an issue, can simply be approximated by the edge's midpoint. These intersection points can then be used to generate polygons that approximate the part, or parts, of the implicit surface that passes through the cube. In fact, a 256-entry table can be generated with each entry corresponding to a unique set of inside-outside values for the cube. This table can be used to quickly determine which edges intersect the cube, and what polygons will be generated as a result of these intersections [7].

If a grid of cubes is oriented over the space that an implicit surface occupies, the above facts can be used to generate a polygonization of an implicit surface. This is what is presented in [58] and, as pointed out in [32], is a type of cubical cell polygonization.

Cubical cell polygonizations of implicit surfaces have several shortcomings [58, 5, 32]. The surface may intersect the cube without intersecting the vertices. This is a sampling problem inherent in the use of an uniform grid. As shown in Figure 2.5, an additional problem is the fact that of the 256 different inside-outside cases for a cube, several are ambiguous and must be carefully handled or else topological ambiguities can result in holes in the polygonization.

Problems due to topological ambiguity can be reduced or alleviated by several

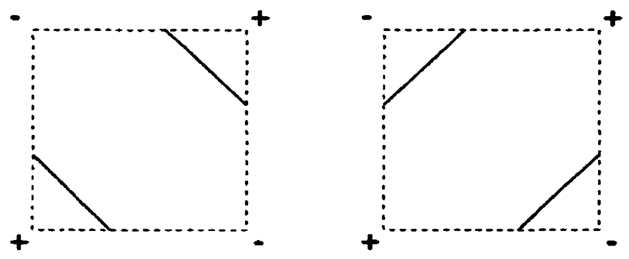


Figure 2.5: **A two-dimensional example of a case where a grid based implicit surface polygonizer can produce a topologically incorrect polygonization. Which of these polygonizations should be chosen?**

different strategies: cell decomposition, preferred polarity, and topological inference.

As implemented in [6], cell decomposition triangulates tetrahedrons rather than cubes. Tetrahedrons have the desirable property of having no ambiguous cases for triangulation. This approach completely solves the ambiguity problem but does generate many more triangles compared to the cubical algorithm.

As pointed out by Ning and Bloomenthal in [32], preferred polarity solves the ambiguity problem by producing polygons that always separate outside vertices, or always separate inside vertices. In this way, topological ambiguities can be straightforwardly avoided, although topological inaccuracy is not guaranteed.

Both cell decomposition and preferred polarity address the ambiguity problem but since both techniques make use of information at just the vertices of the cube, neither can guarantee that the resulting polygons are topologically correct when compared to the actual surface. Topological inference uses additional samples to gain more information about the surface and produce a topologically more accurate polygonization. In [58] the additional sample is at the center of certain faces of the cube. Although this reduces the chances of a topological ambiguity and increases the chances the chances of topological accuracy, it guarantees nothing. Topological ambiguities and inaccuracies may still occur using topological inference.

There are other algorithms, such as the shrink wrap algorithm [44] and point distribution methods [46, 12] that do not fall under the category of cubical cell polygonization, but can be used to generate a polygonal approximation to an implicit surface.

The shrink wrap algorithm [44] is interesting in that, like [5], it produces an adaptive polygonization. Areas with relatively low curvature are approximated with fewer triangles than areas with higher curvature. The algorithm starts with a mesh of triangles, forming a tetrahedron, that fully contains the implicit surface. The algorithm proceeds by repeatedly moving the vertices towards the implicit surface and then examining the triangles to determine if they need to be subdivided. Shrink wrap is quite fast, and can give good accuracy provided that the implicit surface being approximated is homeomorphic to a sphere. It was extended to remove this constraint in [8].

Oriented particle systems form the basis of the algorithm described in [46], a point distribution method. In their algorithm, a particle system uses four rules to approximate the shape of an implicit surface: a particle must stay on the implicit surface; a particle must align itself with the field's gradient; a particle is repulsed by other particles; and if there are no particles within some distance, ϵ, a particle will divide into two particles. In this way, particles will multiply and spread across the implicit surface, until the particle system reaches an equilibrium state. The particles can be used to define a set of discs whose normals are aligned with the normal of the implicit surface being approximated and that are nicely distributed across the surface. If polygons are necessary, the particles can be used as input points into a three-dimensional polygonization algorithm such as Delauney triangulation.

In [12], an oriented particle system is again used to quickly generate a polygonal approximation to an implicit surface. However, in this method each primitive field is associated with a set of particles and the polygonal mesh that they form. If two primitive fields interact their corresponding meshes are merged in a way that does not necessarily produce accurate results, but is certainly efficient and acceptable for some applications.

2.2.2 Ray Tracing of Implicit Surfaces

Just as there exist many methods to indirectly visualize an implicit surface via polygonization, many methods exist to visualize an implicit surface directly. Most direct algorithms are used in the context of ray tracing, although Blinn uses a direct algorithm that is more closely related to scan-line algorithms [4]. In order to make any claims of accuracy and efficiency, most direct algorithms need some sort of auxiliary information in addition to the implicit function, $F(\vec{p})$. This fact negatively impacts the generality of the algorithms in the sense that it constrains the set of functions that can be used. This discussion will focus only on robust direct and general algorithms applicable to ray tracing, therefore only Kalra and Bar's use of the Lipschitz condition [29], Mitchell's use of interval analysis [30], and Hart's sphere-tracing [27] will be examined in any detail. Algorithms that are constrained to polynomial field functions, such as [55], will be ignored.

In ray tracing, the crucial problem is to find the intersection of a ray with a surface. A ray can be defined by $r(t) = \vec{p_0} + \vec{d}t$, where $\vec{p_0}$ and $\vec{d}$ are three-dimensional vectors and t varies over the interval $[0, \infty)$. By substituting r into the field function F we can find the intersections of the implicit surface with r by solving $f(t) = (F \circ r)(t) - c = 0$. The least positive root corresponds to the first intersection with the ray. This is the most useful intersection in standard ray tracing. Since f is a black box function, a general root finding method must be used. Unfortunately, as pointed out by [27], methods such as Newton's or regular falsi have the problem that they do not necessarily converge to the least positive root.

Both the Lipschitz method and the interval analysis method have the property that they are guaranteed to not only find the least positive root, but they can easily be extended to find all roots, making them particularly appropriate when integrating implicit surfaces into a traditional ray tracer incorporating CSG [30] or transparency,

```
intersect(interval)
1  If there is only one root in interval
2     Use Newton's method or regula falsi to
         refine and return the root.
3  Else if there are no roots in interval
4     Return no roots.
5  Else
6     firsthalf = the first half of interval
7     If intersect(firsthalf) returns no roots
8        secondhalf = the second half of interval
9        Return the result of intersect(secondhalf)
10    Else
11       Return the result of intersect(firsthalf)
```

Figure 2.6: **The general algorithm used in [29] and [30] for intersecting a ray with an implicit surface.**

where all ray-surface intersections are required.

A function, f, is Lipschitz over a region, R, only if a positive constant, $\mathcal{L}$, exists such that $|f(x_0) - f(x_1)| < \mathcal{L}|x_0 - x_1|$. All three root-finding algorithms depend on the restriction that the function be Lipschitz.

The algorithms make use of the Lipschitz condition to allow for simple and reasonably efficient methods for isolating roots before refining them using a method such as Newton's or *regular falsi*. Both algorithms are essentially as outlined in Figure 2.6. The difference between the two methods is the way in which they exploit Lipschitz conditions in lines 1 and 3.

Sphere tracing as introduced in [27] uses the Lipschitz condition in a novel way to construct an algorithm that is of similar efficiency, but potentially more general.

Like the other ray-intersection algorithms sphere tracing can be easily extended to find all intersections of a ray with an implicit surface.

All three algorithms can use some sort of spatial partitioning to eliminate regions of empty space that cannot contain the implicit surface, allowing the ray tracer to avoid unnecessary queries to the field. This fact is due to each algorithm's use of the Lipschitz conditions. As pointed out in [27], spatial partitioning has the additional and greater benefit of allowing local Lipschitz constants to be computed for each sub-space.

Ray Tracing of LG-Implicit Surfaces

An implicit surface, where the auxiliary functions $L(R)$ and $G(l,T)$ can be defined, is called an LG-implicit surface, which forms the basis for the guaranteed ray intersection algorithm for implicit surfaces in [29]. $L(R)$ corresponds to a bound of the maximum rate of change of F over the region R. $G(l,T)$ is a function for the bound of the maximum rate of change for the directional gradient on the line l over the closed interval $T = [t_0, t_1]$. With definitions of L and G that correspond to a particular field function F, a recursive algorithm which finds the least positive root can be implemented as in [29]. In their algorithm the L and G functions are used to determine whether a single root is contained in T, in which case the root can be found by Newton's method or *regula falsi*, or whether no roots are contained in T, in which case the algorithm can terminate. In any other case, the algorithm subdivides T and calls itself recursively on the two sub-intervals.

The intersection algorithm for LG-implicit surfaces has the notable attribute that it is guaranteed not to miss any intersections. The implementation of the algorithm is simple and straight-forward. The only problems with the algorithm is its use of L and G which are dependent on the definition of F. It is desirable that L and G not only be close bounds, but also that they be efficient to compute. The fact that L and G are only bounds, and not necessarily exact bounds, yields great freedom in choosing an efficient implementation. As pointed out in [29], even global constants will suffice, but the algorithm is more efficient when more accurate bounds are computed. Hart [27] points out that a thorough understanding of the function, F, is required to find a tight, efficient, or both tight and efficient, Lipschitz bound.

Interval Analysis for Ray Tracing Implicit Surfaces

Interval analysis [31] is the basis of an algorithm presented in [30] with the same goal as, and a similar implementation to, the ray intersection algorithm for LG-implicit surfaces. An interval is a set of real numbers inclusively bounded by a minimum and maximum, $i = [a, b]$ where $a \leq b$. The set of real intervals contains the real numbers, since $[a, a] = a$. Given the definition of intervals, we can define functions that operate on, or yield, intervals. A pertinent example is the extension of the field function, F, to return its interval over a line segment. In this case the interval version of F would return the interval $[a, b]$, which bound the field's values over the set of points defined by the line segment.

The algorithm presented in [30] makes use of the interval version of F along with the interval version of its first derivative F', both of which compute an interval over a line segment. The interval versions of F computes the minimum and maximum field values that can occur over the points of the line segement defined by r. Likewise, F' yields the minimum and maximum rate of change that occurs over the line segment. In a fashion very similar to [29], the algorithm uses the interval versions of F and F' over an interval of a ray.

Employing interval analysis in this way results in a ray intersection algorithm that is simple and robust. Unfortunately, as is the case with LG-implicit surfaces, the auxiliary functions required reduce the size of the set of functions that can be easily and efficiently implemented. It is not clear which technique allows for a larger set of implicit surfaces to be efficiently rendered, but interval analysis allows the generation of a reasonable interval version of F automatically [30], making the rendering of practically all implicit surfaces, that satisfy the Lipschitz condition, possible.

It is not generally recognized that, rather than just using an interval version of F and accepting the very loose bounds that will result, a thorough understanding of the field function can be used to efficiently compute accurate bounds of F over a segment of a ray. For example, if we know that f is always decreasing and that our skeletal primitive, p, is a point, then we know that F will be maximized at the point on the ray segment which is closest to p. This point can be straight-forwardly computed by simple geometry.

Sphere Tracing Implicit Surfaces

Sphere tracing is a ray intersection method originally devised for deterministic fractals [26] and later generalized to many other types of surfaces [27]. The algorithm uses the Lipschitz conditions to devise a bound of the distance, $d(\vec{p})$, from an arbitrary point, $\vec{p}$, to the implicit surface. This estimate is used as a basis for a simple algorithm that steps along a ray by an amount guaranteed not to penetrate the surface. If the distance bound becomes smaller than some ϵ, where ϵ is near machine precision or display limits, it is assumed that an intersection has been found.

Sphere tracing offers a number of benefits over past methods of performing ray-intersections with implicit surfaces. It does not require a method to compute the derivative of F. A wider range of implicit surfaces, including creased, rough, and fractal surfaces are possible to render via sphere tracing. Finally, sphere tracing provides an integrated method of anti-aliasing since it approximates cone tracing [27].

Interval analysis and LG-implicit methods for ray intersection converge linearly during the root isolation stage, and then converge quadratically during the root refinement stage. In contrast, sphere tracing always converges linearly.

Like the interval analysis and LG-implicit methods for ray intersection, sphere tracing has some requirements for efficient computation. In interval analysis, the constraint is that the interval be accurately computed. LG-implicit surfaces demand that accurate bounds for L and G be found. With sphere tracing, an accurate distance bound is required. In all three algorithms a loose bound may be used, but this will not allow the algorithm to execute as efficiently as if an accurate bound is used.

CHAPTER 3

The *BlobTree* Implicit Modeling System

Developments in systems for modeling and visualizing implicit surfaces have been numerous and varied. The foundation for most systems is the basic method for combining the effect of several primitive fields specified by skeletal primitives. As proposed in [35], CSG can be easily integrated with such a system [34, 52]. Additional features include spatial warping as defined by [3], whose integration into implicit surfaces systems is documented in [50, 11], and blending graphs, which allow selective blending of surfaces [23]. The *BlobTree* [51] has been introduced as a method of unifying these features into a single data structure and is the subject of this chapter.

The *BlobTree* has undergone several redesigns and reimplementations since its inception. The first complete implementation of the *BlobTree* is due to Andy Guy and is detailed in [24]. A second implementation, that refined some aspects of Andy's implementation, introduced texture mapping, and integrated the Python programming language, is due to Mark Tigges and is covered in [40].

A major part of the research for this thesis was a complete redesign and implementation of the *BlobTree* system. The design of this new *BlobTree* system is the focus of this chapter.

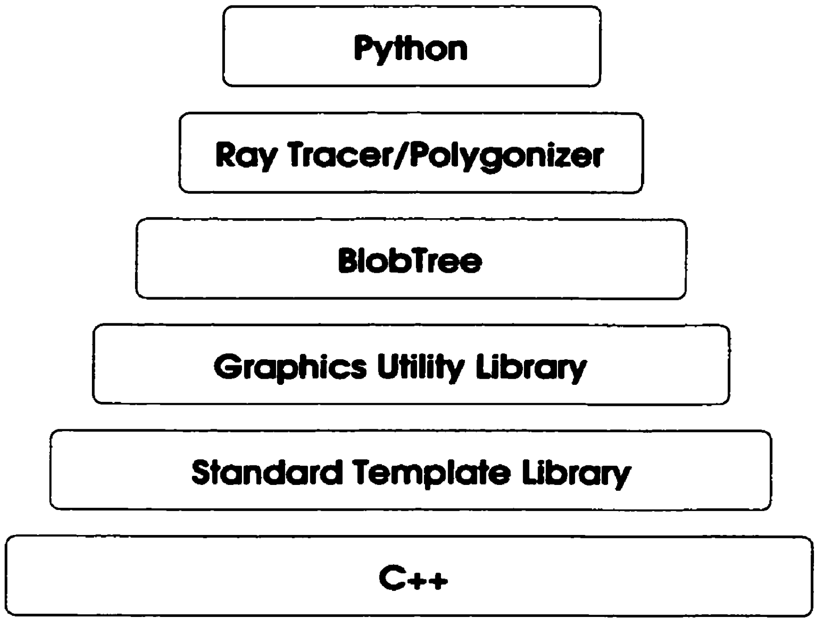


Figure 3.1: **The layout of the *BlobTree* system. The upper components depend on lower components.**

3.1 Introduction to the *BlobTree* System

The layout of the *BlobTree* system is illustrated in Figure 3.1. The *BlobTree* system is a set of layered libraries that, taken together, provide a system for building, storing, and rendering implicit surfaces models.

The *BlobTree* is a data-structure for expressing fields and surfaces. Like many data-structures in computer science, the *BlobTree* is based on a tree structure. In the *BlobTree*, each leaf node corresponds to a primitive field function, and each non-leaf node corresponds to an operation on one or more descendant field functions.

The *BlobTree* is built upon a graphics utility library. The graphics utility library includes support for the reading, writing, and manipulation of images. Linear algebra routines that are particularly useful to computer graphics are also included in the graphics utility library. Lastly, the graphics utility library also includes the foundation necessary for the expression of values that change over time.

High quality visualization of *BlobTree* models is accomplished through a ray tracer. Interactive visualization of *BlobTree* models is accomplished through the use of a polygonizer, to generate a polygonal approximation of the model, and the use of OpenGL to visualize the resulting polygons.

Although it is not required to make use of the *BlobTree* system, Python, a general purpose programming language, is used in most of the tools which make use of the *BlobTree*. The fact that Python is both embeddable and extendible makes it applicable to the *BlobTree*. Python is embeddable in the sense that it can be easily accessed from a C/C++ program. *BlobTree* nodes can call the Python interpreter to evaluate some Python code. Python is extendible in the sense that new procedures, datatypes, and classes, all implemented in C/C++ can be added to the Python language. This allows the nodes of the *BlobTree* to be instantiated, manipulated, and rendered from within a Python script.

Python provides an easy way to interactively access the functionality of the *BlobTree*, the ray tracer, and the polygonizer. Python programs are used to create, and then to store, *BlobTree* models. In a sense, Python is a glue language that helps the tools for visualizing *BlobTree* models, the tools for constructing *BlobTree* models, and the *BlobTree* models themselves, interact in a useful fashion.

3.2 Object Orientation of the *BlobTree*

The *BlobTree* is implemented in the C++ programming language [39]. This allows the use of object-oriented language features to simplify its implementation and extension. In order of importance, the major object-oriented features which the *BlobTree* uses are user-defined types (classes), polymorphism, abstract inheritance (pure virtual classes), and type instantiation (templates).

User defined types are used to define the *BlobTree* nodes themselves. Many user defined types are also used internally and externally in the implementation of each node. Prevalent examples of this include the vector and matrix classes of which the *BlobTree* makes heavy use.

The use of polymorphism and abstract inheritance in the *BlobTree* are related to one another. Inheritance simply allows for a class to redefine and extend the functionality of a class from which it inherits. Abstract inheritance allows a base class to define functionality that must be implemented in any class that inherits from that base. For example, all classes that inherit from the *field function* class must define the *value* function. Polymorphism allows a class to be used without full knowledge of its type. For example, a *blend* node has children, all of which must be *BlobTree* nodes, but beyond that, the type of each child node is of no consequence to the *blend* node.

Type instantiation is used in the *BlobTree* through its use of the C++ Standard Template Library (STL). The STL provides a set of containers, such as *list*, *vector*, and *queue*. Through type instantiation, a container for integers, matrices, or *BlobTrees*, can be instantiated and put to use with very little effort.

3.3 The Tree Nature of the *BlobTree*

The *BlobTree* is a tree data-structure. Tree structures have been used in many areas of computer science and computer graphics to represent information. The *BlobTree* has much in common with scene-graphs, which are used throughout the field of computer graphics. In particuar, the *BlobTree* incorporates hierarchical grouping, transformations, and CSG information.

It is useful to group the different types of nodes of the *BlobTree* into three broad categories based on the number of children they can have: *terminal* nodes have no children; *unary* nodes have one child; *binary* nodes have two children; and *n-ary* nodes have multiple children. *Terminal* nodes will always be at the leaves of a *BlobTree.* While *unary*, *binary*, and *n-ary* nodes will always be internal.

In the *BlobTree* implementation, a class corresponding to each of these categories of node exists. Each of these classes implements functionality common to all nodes of that category. Since each of these classes is related to the number of children a node has, the functionality they implement is limited to providing accessor and mutator methods to set and retrieve their children.

3.4 The *BlobTree*

The obligations of all *BlobTree* nodes are encapsulated in the declaration of the *BlobTree* base class, the class from which all *BlobTree* nodes must inherit. The functionality declared in the *BlobTree* base class must be implemented by all *BlobTree* nodes. This functionality includes the ability to compute the value, gradient, interval along a line, and interval within a box, of the field function. Additionally, all *BlobTree* nodes must be able to compute field dependent *attributes*, such as colour.

Since the *BlobTree* is used to express fields, the most important feature of all *BlobTree* nodes is that they return the implicit field value given a point in space. This is the *value* function, which corresponds to $F(\vec{p})$.

The gradient of a field is important in determining the normal of the implicit surface. Therefore all nodes of the *BlobTree* must be able to compute $\nabla F(\vec{p})$, the gradient of $F(\vec{p})$. This functionality is encompassed in the *gradient* function. Given a small value for δ, where $\delta \in \mathbb{R}$, the gradient can be numerically approximated with $(\frac{(F(\vec{p_x})-F(\vec{p})}{\delta}, \frac{F(\vec{p_y})-F(\vec{p})}{\delta}, \frac{F(\vec{p_z})-F(\vec{p})}{\delta})$, given that $\vec{p_x} = \vec{p} + (\delta, 0, 0)$, $\vec{p_y} = \vec{p} + (0, \delta, 0)$, and $\vec{p_z} = \vec{p} + (0, 0, \delta)$. Due to the problem of choosing δ and the problems with machine precision when δ is too small, it is desirable to use an exact value of $\nabla F(\vec{p})$ rather than a numerical approximation.

In order to accurately ray trace an implicit surface, a guaranteed method of intersecting a ray with the implicit surface is necessary. The *line interval* and *box interval* functions respectively yield the interval of possible field values for the set of points on a line and within a box. Given these two functions, a guaranteed method for ray tracing, based on a simplification of [30], can be devised.

For some applications, the ability to compute the colour of an implicit surface at a particular point in space is needed. This allows the colour of an implicit surface to be dependent on the field value, yielding the ability to blend colour as well as geometry. This functionality is accessed through the *attribute query* method, which allows arbitrary field dependent values, called *attributes* to be computed for any point in space. *Attribute query* takes a position, a list of strings that identify the *attributes* needing to be computed, and a list of *attributes*. The list of *attributes* represents default values for each *attribute*. The *attribute query* method returns a list of *attributes*. This list contains the computed *attributes* corresponding to the method's parameters and the node it is called upon. *Attributes* need only have the capability to be multiplied by a scalar number and summed with *attributes* of like type. Therefore an *attribute* representing *colour* can be easily defined.

Lastly, in order to make texture-mapping of implicit surface possible, a two-dimensional uv-mapping of a point on the surface is required. This functionality is accessed through the uv method, which maps a query point to uv-coordinates. The implementation and application of the uv method will largely be ignored, since it is beyond the scope of this thesis. For a detailed discussion of issues surrounding the determination of uv, the reader is referred to [40].

3.5 Skeletal Primitive Nodes

Although it is possible to directly specify a function within the *BlobTree*, most instances of *BlobTrees* are based on ***skeletal primitive*** nodes, which are terminal nodes corresponding to the combination of a skeletal element and a ***primitive field function*** both of which are defined in Section 2.1.1. Given a distance, d, the ***primitive field function***, f_p, must be able to compute its value, $f_p(d)$, and its first derivative, $f_p'(d)$.

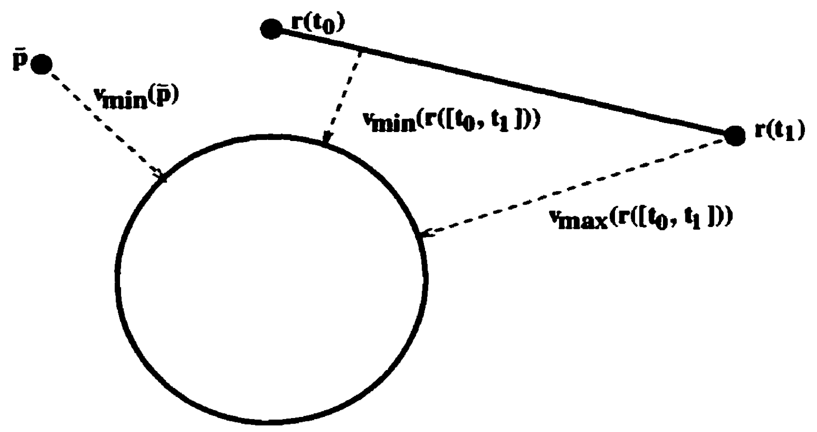


Figure 3.2: **An illustration of the crucial vectors that all *skeletal primitives* must be able to compute. In this case, the *skeletal primitive* is a circle.**

Each *skeletal primitive* must be able to compute five crucial vectors. The implementations of *value*, *gradient*, *line interval*, *box interval*, and *attribute query* are made in terms of these five vectors and the stored *primitive field function.*

The most fundamental vector for a *skeletal primitive* node to compute is the shortest vector from a point in space to a point on the skeleton, $v_{min}(\vec{p})$. A *skeletal primitive* must also implement $v_{min}(r([t_0, t_1]))$ and $v_{max}(r([t_0, t_1]))$, which respectively correspond to the shortest and longest values of $v_{min}(\vec{p})$ for the set of points on an interval of a ray. Similarly, implementations must be provided for $v_{min}([x_0, x_1], [y_0, y_1], [z_0, z_1])$ and $v_{max}([x_0, x_1], [y_0, y_1], [z_0, z_1])$, which correspond respectively to the shortest and longest vectors resulting from $v_{min}(\vec{p})$ for the set of points in an axially aligned box. These vectors are illustrated for the hypothetical *circle skeletal primitive* in Figure 3.2. For a discussion of other sorts of primitives, see [24].

If a natural local coordinate system exists, it should be used to compute the five required vectors. This can significantly simplify their computation. For example, using the center of the circle as the origin of the local coordinate system for the implementation of the *circle skeletal primitive.* In this case, and in most others, using a natural coordinate system for the *skeletal primitive* does not impact the flexibility of the *BlobTree* system since an *affine transformation node* (defined in Section 3.6) can be used to perform any affine transformation on the *skeletal primitive*. However, it should be noted that in the *BlobTree* system, the effect of applying an *affine transformation* is that the field is transformed, not that the underlying *skeletal primitives* are transformed.

The methods for *value*, *gradient*, *line interval*, and *box interval* can be implemented in terms of $v_{min}(\vec{p})$, $v_{min}(r([t_0, t_1]))$, $v_{max}(r([t_0, t_1]))$, $v_{min}([x_0, x_1], [y_0, y_1], [z_0, z_1])$ and $v_{max}([x_0, x_1], [y_0, y_1], [z_0, z_1])$: the *value* method is implemented as $f_p(||v_{min}(\vec{p})||)$; the *gradient* method is implemented as $(v_{min}(\vec{p}) \cdot f'_p(||v_{min}(\vec{p})||)/||v_{min}(\vec{p})||$; the *line interval* method is implemented as $[f(||v_{max}(r([t_0, t_1]))||), f(||v_{min}(r([t_0, t_1]))||)]$; and the *box interval* method is implemented as:

$$[f(\ ||v_{max}([x_0, x_1], [y_0, y_1], [z_0, z_1])||\ ), f(\ ||v_{min}([x_0, x_1], [y_0, y_1], [z_0, z_1])||\ )] \tag{3.1}$$

The *skeletal primitive* node implements the *attribute query* method by computing its field value at the query point, creating a copy of the list of *attributes* passed into the method, multiplying each *attribute* in the copy by the field value, and returning the new list of *attributes*.

3.6 Affine Transformation Nodes

Affine transformation nodes are unary nodes that apply affine transformations to their child nodes. In order to do this, the inverse of a matrix, T, which encodes the desired affine transformationan is applied to the appropriate argument of each method, and the transformed argument is passed to the corresponding query in the child node. The result of this is that calls to *value*, *gradient*, *line interval*, and *box interval*, respectively become $F(T^{-1}\vec{p})$, $\nabla F(T^{-1}\vec{p})T^t$, $F(T^{-1}r, [t_0, t_1])$, and $F(T^{-1}b)$. Notice that for the *gradient* method the result of a call to the child's *gradient* method,

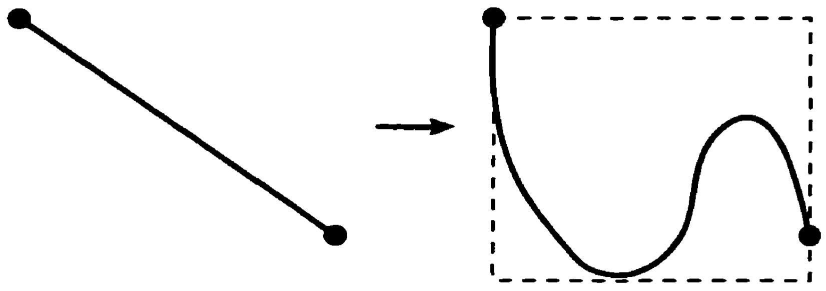


Figure 3.3: **A line warped to a curve and the resulting bound of the curve.**

$\vec{g}$, is treated as a row-vector and transformed by the transpose of T, ie. $\vec{g}T^t$. This special transformation of $\vec{g}$ is necessary since $\vec{g}$ is a gradient rather than an ordinary vector.

With the exception of the query point, an *affine transformation*'s *attribute query* method passes its unaltered arguments to its child. The result of transforming the query point by T^{-1}, is passed to the child node's *attribute query* method.

3.7 Warps

A *warp* node applies a non-affine transformation to its child node. To simplify their implementation, *warp* nodes incorporate the fact that the effect of a non-affine transformation on a particular point in space can be expressed as an affine-transformation. So, given a point in space, an affine transformation, M, is computed and applied to the point in the same way that *affine transformation* nodes apply T. The exception to this is the *line interval* and *box interval* methods.

Since the non-affine transformation of a line can be a curve, calls to a *warp* node's *line interval* method must result in calls to the child node's *box interval* method. A box which bounds the non-affine transformation of the line segment is computed and this is used as an argument to the child node's *box interval* method. This case is illustrated in Figure 3.3. Similarly, calls to a *warp* node's *box interval* method result in the computation of a new box, that bounds the non-affine transformation of the original box, and is then used as an argument to the child node's *box interval* method.

3.8 *BlobTree* Nodes for CSG

Although the nodes implementing CSG operations in the *BlobTree* are n-ary nodes, for simplicity the following discussion treats them as binary nodes.

The union, intersection, and difference of two fields can be easily expressed in an implicit surfaces system [35]. The union of two fields, F_1 and F_2, is simply the maximum of the two field values at a point, $max(F_1(\vec{p}), F_2(\vec{p}))$. Similarly, the intersection is expressed as the minimum of the two field values at the query point, $min(F_1(\vec{p}), F_2(\vec{p}))$. The difference of F_1 and F_2 is the intersection between F_1 and the negation F_2, where negation is taken to mean $2c - F(\vec{p})$. This yields $min(F_1(\vec{p}), 2c - F_2(\vec{p}))$. Each of these expressions can be viewed as being conditional on the relative values of the fields at the query point. The *union*, *intersection*, and *difference* nodes use this fact in each of their implementations.

The implementation of the *gradient* method is similar to the *value* method. For the *union* node the gradient corresponds to whichever child has the maximum field value at the query point. The *intersection* node is similar but the gradient of the child with the minimum field value at the query point is returned. With the *difference* node, $\nabla F_1(\vec{p})$ is returned if $F_1(\vec{p})$ is less than $2c - F_2(\vec{p})$, otherwise $-\nabla F_2(\vec{p})$ is returned.

A similar situation occurs in the implementation of the *attribute query* methods for CSG nodes. The result of calling *attribute query* on a CSG node will be the result of calling *attribute query*, with the same arguments, on one of the CSG node's children. Unlike the traditional methods used for computing the surface attributes in ray tracing (which is simply wrong for difference and intersection) [54], the *BlobTree* produces an intuitively correct result for CSG nodes. This implies that the child whose *attribute query* will be returned, as a result of calling *attribute query* on a *union* or *difference* node, will be whichever child has the maximum field value at the query point. In *difference* nodes, the child whose *attribute query* is returned is always the one whose field is not negated.

The implementations of the *line interval* and *box interval* methods are nearly identical for each of the CSG nodes. In both cases, two intervals, $[s, t]$ and $[u, v]$, corresponding to either the *line interval* or *box interval*, are computed by the two children of the CSG node. The computation of the interval returned is identical for both the *line interval* and *box interval* methods, $[max(a, c), max(b, d)]$ for *union* nodes, $[min(s, u), min(t, v)]$ for *intersection* nodes, and $[min(s, 2c - v), min(t, 2c - u)]$ for *difference* nodes.

3.9 Blend Nodes

A *blend* node is an n-ary node that simply sums the field values of its children. This fact makes its implementation simple and straightforward. The *value* and *gradient* methods for a *blend* node sum the result of respectively calling *value* or *gradient* for each of its children. Both the *line interval* and *box interval* methods simply sum the result of calling the corresponding interval method for each child, using the interval definition of addition $([s, t] + [u, v] = [s + u, t + v])$.

The *attribute query* method for a *blend* node computes the weighted average of the result of calling each child's *attribute query* method. For example, if diffuse colour were one of the *attributes* being computed, the list of *attributes* returned would contain an entry corresponding to the diffuse colour. That entry would contain the weighted average of each diffuse colour *attribute* returned by the children. In computing this average, each child's diffuse colour *attribute* would be weighted by the child's field value at the query point. The result of using this approach is that, like the surface geometry, each child's contribution to the blended *attributes* values is proportional to its field value at the query point.

3.10 Controlled Blend Nodes

Controlled blend nodes are used whenever it is required that certain nodes blend with one another while others do not. For example, in a *BlobTree* model of a human hand which is clenched into a fist, the tips of the fingers should definitely not blend with the palm. This functionality can be achieved through the use of ordinary *blend* and *union* nodes, but the *controlled blend* node is more convenient and much more efficient. Controlled blending was introduced in [23].

A *controlled blend* node contains a list of children, and a list of groups of these children. To compute the *value* of a *controlled blend* node, the *value* of each child is computed and stored. Then, for each group, the pre-computed values, corresponding to each child in the group, are summed. The maximum of these sums is returned. In this way, each child is only queried once for its *value*, rather than being queried once for each group of which it is a member.

Just as is done for the *value* method, the *gradient* method computes the sums for each group. The group with the largest sum of values has all of its *gradients* computed and summed. This sum of *gradients* is then returned.

The *attribute query* method works in a similar fashion to the gradient method, the group with the largest sum of field values determines which group will be used to compute the *attributes* returned. Using the same method as in ordinary *blend*, the group's *attributes* are determined by computing the weighted average of the result of calling each group member's *attribute query* method.

The *line interval* and *box interval* methods work in a very similar fashion to the *gradient* method. The group with the largest sum of values is computed, and that group is used to compute the sum of *line intervals* or *box intervals*, just as in a *blend* node.

3.11 Attribute Nodes

Attribute nodes are unary nodes that contain a string and a value for an *attribute*. The string represents the name of the stored *attribute*. A common example would be an *attribute* node which contains a *colour attribute* and the string "diffuse colour".

Attribute nodes are indifferent to the *value*, *gradient*, *line interval* and *box interval* methods. Therefore, calls to these methods result in corresponding calls to the *attribute* node's child. The arguments passed to the *attribute* node's method are passed unaltered to the child's corresponding method.

Attribute nodes have a non-trivial implementation of the *attribute query* method. A call to an *attribute* node's *attribute query* method passes arguments for a position, a list of strings that identify the *attributes* needing to be computed, and a list of *attributes*, specifying default values. If the name of the *attribute* node's stored *attribute* is in the list of strings, the corresponding entry in the list of *attributes* is over-written with the value of the *attribute* node's stored *attribute*. In this way the value stored in an *attribute* node is applied to its children, unless it is over-ridden by an *attribute* node further down in the tree. This is illustrated in Figure 3.4.

The application of two-dimensional texture maps to implicit surfaces defined by the *BlobTree* is accomplished through *texturing* nodes which are a type of *attribute* node that use a two-dimensional image to assign *uv*-dependent *attribute* values to the query point. There are two methods for computing the *uv* value used. The simplest is to call the child's *uv* method. The other method is to simulate a particle moving along the field's gradient until it strikes a surface for which a simple *uv* mapping exists. This method uses repeated calls to the child's *gradient* method and is therefore very expensive. Further advantages and disadvantages of these techniques are detailed in [42].

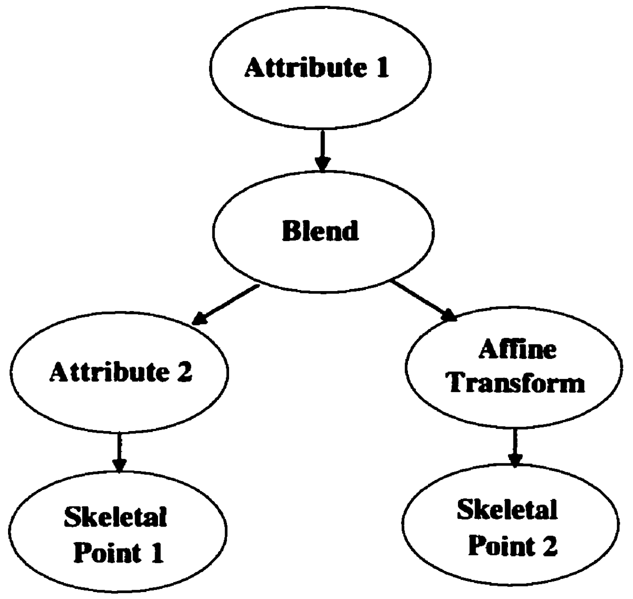


Figure 3.4: **An illustration of how *attribute* nodes are applied in a *BlobTree*. *Skeletal Point* 1 will be affected by *Attribute Node* 2, while *Skeletal Point* 2 will be affected by *Attribute Node* 1.**

3.12 Polygonizing the *BlobTree*

There are many algorithms for the polygonization of implicit surfaces [32]. Although these algorithms are applicable to the *BlobTree*, the method outlined in [58], augmented with the technique in [52] for accurately representing first order discontinuities due to CSG, is used. This method is used due to its robustness, generality, and speed.

The polygonization algorithm uses a uniform grid to divide space into cubes. The algorithm is only concerned with cubes that intersect the implicit surface. The algorithm makes use of a continuation algorithm [2] to find these cubes. The continuation algorithm must be seeded by cubes that are guaranteed to either be intersecting, or completely inside of, the implicit surface.

The algorithm finds seed cubes by first querying the *BlobTree* for a set of seed-points that are guaranteed to be on the inside of the implicit surface. This function-

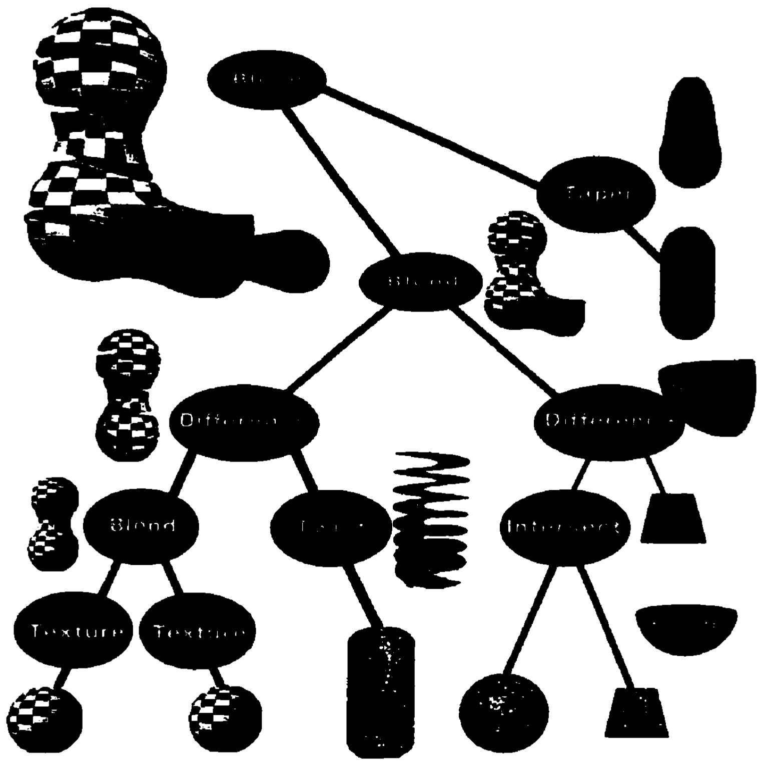


Figure 3.5: ***BlobTree* nodes used in the construction of a novel model. (Image courtesy of Brian Wyvill.)**

ality is encapsulated in the *seed points* query. For *skeletal primitive* nodes this query simply returns a set of points that lie on the skeleton. For example, a *point primitive* would return the origin. The other nodes in the *BlobTree* must implement the *seed points* query appropriately. *Affine transformation* and *warp* nodes transform and then return the seed-points of their child. *Blend* and *controlled blend* nodes simply return the union of their children's sets of seed points. CSG nodes must apply the rules of CSG to their children's seed points to determine if they should be included or removed. *Union* nodes act exactly like *blend* nodes. *Intersection* nodes can only return seed points that are inside all of their children. *Difference* nodes can only return seed points that are inside their first child, and outside of all other children.

3.13 Ray Tracing the *BlobTree*

For ray tracing *BlobTrees*, a simplified application of interval analysis as presented in [30] is used. The simplified algorithm does not require the formulation of the interval of the field's derivative along a ray. It only requires that the minimum and maximum field values, both along a ray segment and within an axially aligned box, be computable. This functionality is accessed through the *line interval* method which all *BlobTrees* must implement. The algorithm proceeds by finding the line interval for the ray segment of interest, and if the surface's iso-value is contained in the field interval, calls itself recursively on the first half and then the second half of the ray-segment. The algorithm terminates when a ray-segment is below some threshold length, ϵ, and still has a *line interval* that contains the surface value, c.

This algorithm linearly converges on an intersection between a ray and an implicit surface and is guaranteed to find all ray-surface intersections. It is important to note that this algorithm queries the *BlobTree* many times to compute a single ray-surface intersection.

3.14 Animating the *BlobTree*

The *BlobTree*'s primary purpose is for modeling. For this reason, past implementations did not allow for temporally-dependent information to be included in the *BlobTree*. Instead, a *BlobTree* would be created, rendered, and destroyed, for each frame of animation. This works, but makes time-dependent effects such as accurate motion blur impractical. It also requires that the code that drives the rendering be aware of the structure of the *BlobTree* model being animated, and how to instantiate such a model for each frame of animation. A *BlobTree* which incorporates time-dependent information, and allows for time-dependent queries, circumvents these problems. This section describes how the *BlobTree* has been extended to incorporate time-dependency.

3.14.1 Tracks for Time-Dependent Values

A *track* is simply a value that changes over time [9]. Any *track* can be queried for its value at an instant of time. How a *track's* value is computed is of little consequence, however it should be as efficient as possible. Some examples of tracks used by *The Graphics Jungle* include *constant tracks* which return the same value regardless of the scene-time they are queried at, *linear interpolated tracks*, which compute their values by linear interpolation, and *spline tracks*, which use splines to compute their values at a particular time.

A special type of track, called a *matrix track*, exists to specify time-dependent matrices. These tracks cannot only compute the time-dependent value of a matrix, but the time-dependent value of the matrix's inverse as well. This functionality exists to avoid the expense of computing a matrix's value and then being forced to invert it using a matrix inversion algorithm. *Matrix tracks* exist for computing time-dependent analogues of all the standard affine transformations: translation, rotation, scaling, and shearing.

Whenever possible, *tracks* have been implemented using C++ templates, so that *tracks* for arbitrary data-types can be instantiated by the compiler. Where appropriate, values in *BlobTree* nodes are replaced with a reference to a track representing a time-dependent version of that value.

Since time-dependent *BlobTree* queries may require computing additional information when compared to their time-independent counterparts, they are more expensive. However, this expense is reasonable (usually less than a factor of $1\frac{1}{2}$), and necessary for time-dependent effects such as motion-blur.

3.14.2 Time-Dependent *BlobTree* Queries

If *BlobTree* queries are to be time-dependent, then they must be modified to take a parameter representing the scene-time at which the query occurs. This must be done for each of the required *BlobTree* queries. Each node's implementation of *value*, *gradient*, *line interval*, *box interval*, *attribute query*, and *seed points* must be modified to take a time parameter and use it appropriately. In nearly all *BlobTree* nodes, the appropriate handling of the time parameter is simply to pass it to queries made to child nodes and to use it in querying any time-dependent values (represented by *tracks*) for their value. A particularly important example of this is the *affine transformation* node.

Affine transformation nodes that changes over time are crucial in the specification of animated *BlobTrees*. In spite of its importance, extending the standard *affine transformation* node to allow for time-dependent queries is straight-forward and uncomplicated. Firstly, the matrix which represents the affine transformation being applied must be replaced with a reference to a *matrix track*. All of the constructors, mutators, and accessors, that deal with the matrix, must be modified to deal with a *matrix track* instead. The *BlobTree* queries must also be modified to first query the *matrix track* for its value or inverse value, and apply it in place of the ordinary matrix's value or inverse value, to the query point, line, or box.

The necessary modifications to the *warp* node are very similar to the ones made to the *affine transformation* node. *Attribute* nodes must also be modified to contain tracks in place of any attributes and to replace any references to the attributes with calls to the tracks. *Blend* and *controlled blend* nodes simply pass the time parameter to their children. *Skeletal primitive* are the one node that can safely ignore their time-parameter for all standard *BlobTree* queries.

3.15 Python for Procedural Building of *BlobTrees*

Python is a full-fledged general purpose interpreted programming language. In addition to being object-oriented in nature, one of Python's major attributes is the ease with which it can be both embedded and extended.

```
def peanut():
    b = blobTreeBuilder()
    b.beginMultiple(blend())
    b.insertBlobTree(point())
    b.translate(1.0, 0.0, 0.0)
    b.insertBlobTree(point())
    b.endMultiple()
    return b
```

Figure 3.6: **The classic implicit peanut expressed as a Python script.**

Embedding Python in an application allows the Python interpreter to be called and used to execute some Python script. In this way, the Python interpreter could be used to evaluate a mathematical expression entered by the user.

Extending the Python language allows new functionality to be added to the language and accessed through a Python script. Extending Python to deal with *BlobTrees* is described in [43]. The major extension of the Python language is the addition of the *BlobTree builder* class. The *BlobTree builder* was originally a C++ class used to simplify the building of *BlobTrees*, but it is now almost exclusively used through a Python interpreter.

A simple example of a *BlobTree* model expressed in Python is shown in Figure 3.6. A slightly more complicated model which makes use of Python's flow-control structures is given in 3.7.

Largely because a procedural programming interface is familiar ground for computer science students, Python scripts have become the standard "file format" for *BlobTree* models built by *The Graphics Jungle* students.

```
def bumpyTorus():
  b = blobTreeBuilder()

  b.beginMultiple(blend())

  angle = 0.0
  delta = 2.0*3.14159 / 7.0

  while angle < 2.0*3.14159:
    b.translate(2.0*math.cos(angle), 2.0*math.sin(angle), 0.0)
    b.insertBlobTree(point())
    angle = angle + delta

  b.endMultiple()

  return b
```

Figure 3.7: **A bumpy torus expressed as a Python script.**

3.16 Comparison to Past *BlobTree* Systems

The *BlobTree* system constructed as part of this research is very similar to the past implementations by Andy Guy and Mark Tigges. During the design of the new system it was decided that a clean and understandable design and implementation would be the foremost concern, followed by efficiency.

Past implementations of the *BlobTree* system were one person efforts. The result of this was that these systems were consistent and made sense, to one person. When that person left *The Graphics Jungle*, the remaining students were left to decipher the behaviour of the system via the source code.

The design and implementation of the current *BlobTree* system has benefitted from the fact that it was a team effort. At all points of the design and implementation process at least two people had input on any design decision and nearly all of the code was examined by at least two people. The result is a reasonably consistent and understandable system. The longevity of the system has also been positively affected by these practices, as there are at least two people familiar with any part of the project.

The fact that the design and implementation was shared between a group of people also freed the developers to spend more time deciding on the best course of action for a particular problem. For this reason, it came to the attention of the development team that certain *BlobTree* queries could have high computational cost if they were not implementated with a great deal of care and caution. The best example of this is the *attribute query* in a *blend* node.

As noted in Section 3.4, the *attribute query* method computes the field-dependant *attribute* values at a particular point in space and time. The resulting *attribute* values are weighted according to the field contributions of the *primitive* nodes in the model. A straight-forward way to compute these *attribute* values, in a *blend* node, would be to first compute the field value of each child, then to compute the *attribute* value of each child, and finally to use the field values to weight the *attribute* values. The problem with this approach is that a single query results in multiple queries of nodes further down in the tree. If a particular model has several layers of *blend* nodes, then the number of queries that occur at leaf nodes rises dramatically. This problem occurs throughout the past implementations of the *BlobTree*, but in the current implementation it has been eliminated. The solution is quite simple, and is detailed in [17]. For *attribute query* the solution is to allow *leaf* nodes, rather than *attribute* nodes, to compute their contributions to the query. In this scheme, *attribute* nodes only modify the parameters to the *attribute query* and pass the information down to their sub-trees. This is very similar to the way in which most unary nodes behave.

The way that Python has been utilized in the current system is slightly modified compared to Tigges' use of the language. In the old system, a *BlobTree* tree could only be constructed from the top-down. In the new system, slight changes have been introduced that remove this restriction.

The techniques noted in this research, animation, pruning, and reduction, are additional differences between the past and current *BlobTree* implementations.

CHAPTER 4

More Efficient *BlobTree* Rendering

With only a few exceptions, methods for improving the efficiency of implicit surface rendering have focused on minimizing the number of queries made to the field function. The implicit function evaluations per-triangle (IFEPT) metric was devised in [58]. In that paper, the described polygonizer minimizes the IFEPT through the use of hash-tables. Another approach, put forth in [46], uses oriented particle systems to produce a sampling of the implicit surface suitable for real-time interaction. Their algorithm allows particles that are attracted to the iso-surface, but repelled from each other, to distribute themselves and orient themselves with the gradient of the field. The surface can then be visualized by drawing discs in place of each particle, with the discs' normals oriented along the field's gradient. Both of these algorithms for implicit surface visualization use very different approaches to visualize the implicit surface with as few field evaluations as possible although complex and computationally-expensive-to-evalutate-fields are a stumbling point.

Minimizing the number of field evaluations is perfectly sensible approach, but an alternative is to minimize the expense, in terms of computation time, of the field evaluation. In the simplest of fields, this is quite difficult and probably not worthwhile. However, in systems that build complicated field functions from primitive field functions, minimizing the expense of the field function can be very profitable in terms of computational savings.

As illustrated in Figure 4.1, the number of primitive field functions that contribute to a particular volume of space can be quite small relative to the number of primitive field functions in the model. In this case, although there are seven primitive field functions in the model, only two contribute to the highlighted volume of space. In larger models, the difference between the number of primitive field functions in the model and the number that contribute to a small volume of space can be much more profound. Many past techniques for efficient field evaluation have taken advantage of this fact.

4.1 Past Techniques For Efficient Field Evaluation

Techniques for improving the efficiency of field evaluation were developed alongside algorithms for implicit surface visualization. Blinn [4] used a scan-line method to determine which primitive fields could significantly contribute to the field function. In this way, only the significant primitive field functions are used for a particular scan-line.

In polygonizing an implicit surface, Wyvill and Wyvill [58] used a uniform grid and primitive fields with finite extent to determine exactly which primitive fields could contribute to a cubical voxel of space. In a preprocessing step to polygonization, this method associates a list ot primitive field functions with each cubical grid element in a course voxel grid. Only the primitive field functions which affect the cubical space are contained in the list. In this way, the expense of evaluating the field function is proportional to the number of primitive field functions that affect the voxel that contains the query point. This technique is applicable to both polygonization and ray tracing, and it works for flat data-structures which simply add the contributions of a list of primitive fields. It has not been generalized to more complicated data-structures, like the *BlobTree*, that incorporate concepts such as CSG.

Polynomial approximation to a field along a ray are used for ray tracing in [37].

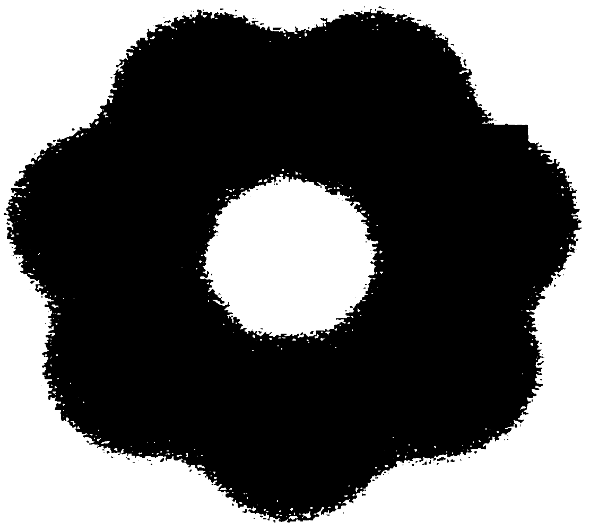


Figure 4.1: **A visualization of the field and $c = \frac{1}{2}$ surface of the *BlobTree* expressed by the Python script in Figure 3.7. Only two of the seven primitive field functions contribute to the highlighted area.**

In this way, once a polynomial approximation is made, the ray-intersection can be found analytically and very quickly compared to algorithms for general implicit functions. For many applications, the error introduced by the polynomial approximation is entirely acceptable.

Implicit *patches* are related to Voronoi diagrams and defined as the volume of space where one primitive field has more influence than any other. Implicit *patches* are used in [19] to attempt to reduce the cost of field evaluations in the context of a LG-implicit surface ray tracer. Each *patch* is associated with the field primitives that affect the space which the patch occupies, so that only those field primitives will need to be queried for a query point anywhere in the *patch*. *Patches* have a relatively complicated shape that is relatively expensive to intersect against a ray. In order to avoid this expense, a more efficient data-structure, based on voxels, is built from the *patches*. Each voxel contains a list of the *patches* that affect its volume. When evaluating the field value for a point, the voxel containing the point is found, and the maximum of the contributions of each *patch* in that voxel is used. This approach is far from optimal. It is also important to note that the approach was only applied to a simple data-structure for expressing fields, and not a general one like the *BlobTree*.

In past implementations of the *BlobTree*, each node incorporates a box that bounds the space that the node affects [51]. This bounding box can be computed in terms of a node's children. For any query of a *BlobTree* node, the query point, line-segment, or box, is first tested to see if it is inside the node's bounding box. If so, the query continues to the node's children. Otherwise, the node can safely assume that the field does not affect the query, act appropriately, and safely avoid a great deal of unnecessary computation.

Duff [13] introduces an elegant technique which makes use of interval arithmetic and the rules of CSG to minimize the size of the tree used to represent the field while applying a recursive algorithm, similar to Mitchell's [30], for ray-surface intersection. The difference is that interval analysis is used to remove parts of the subtree that cannot possibly intersect the segment. In this way, the tree being queried shrinks as the segment along the ray is subdivided.

All of the above algorithms make use of spatial knowledge to determine which primitive fields need to be queried. The first two are only applicable to a flat data-structure, whereas the bounding box method and Duff's method are applicable to a more complicated tree-like data-structure such as the *BlobTree*.

4.2 Spatial Techniques for Rendering

Spatial techniques have been used to increase the efficiency of many algorithms in computer graphics. Perhaps the most prevalent application of a spatial technique is the use of spatial sub-division in ray tracing [18, 20]. In ray tracing, spacial subdivision is used to reduce the number of objects that a ray must be tested against. As a preprocessing step to the actual ray tracing, space is sub-divided into sub-volumes and each sub-volume is associated with the primitives that intersect it. In order to test a ray for intersections with primitives in the scene, the ray only has to be tested against the primitives in the sub-volumes that the ray intersects.

The two most popular spatial subdivision techniques employ cubes in a uniform grid or an oct-tree. In a uniform spatial subdivision, an axially aligned grid is used to divide the scene into cubes. This allows the ray to be very efficiently tested against the cubes that it may intersect [10]. An oct-tree employs axially aligned cubes as well, but makes use of a hierarchy of cubes, where each cube can be sub-divided into eight children [20]. Hybrid methods have also been proposed, where each cube can be subdivided into n sub-cubes, where n depends on the number of primitives present in the cube [28].

Spatial subdivision has also been used to reduce the complexity of CSG trees in scene-graphs for the purposes of ray tracing [36, 22, 47, 56, 57, 13]. All of these techniques make use of the fact that given a subset of space, a CSG tree may be simplified if one or more of its sub-trees does not affect the subset in question. In essence, for a sub-volume of space, the scene-graph is pruned of all nodes that do not effect that sub-volume.

4.3 Pruning the *BlobTree*

The fact that the number of primitives that influence a point in space is small relative to the number of field primitives in the model is used to accelerate ray tracing of implicit surfaces in [19]. The approach used in the research presented in this thesis uses the same principal and has the same goal, but uses a simplified approach.

Given an axially aligned box and a *BlobTree* all sub-trees which do not affect the volume bounded by the box are pruned away. This functionality is encapsulated in

```
prune($U$, box)
1  If $U$ is an affine transformation or warp node
2     Transform the box as appropriate
3  Set $C$ to the result of calling prune on $U$'s child with box
4  If $C$ is the Null node
5     return the Null node
6  Else If $C$ is the $U$'s child node
7     return the child node
8  Else
9     Allocate a node of $U'$, of the same type as $U$
10    Set $U'$'s child to $C$
11    return $U'$
```

Figure 4.2: **The algorithm used to prune a unary node.**

the *prune* method. Calling *prune* on a node can have one of only four results: the node itself; a new instance of the same type of node; a child of the node, or the *Null* node. The return values when pruning terminal, unary, and n-ary nodes are introduced below.

A terminal node T, tests whether the box over-laps the bound of its field. If it does, T is returned. Otherwise the *Null* node is returned.

A unary node, U, transforms the box if it is an *affine transformation* or *warp* node. The box is then used to prune U's child. If the child prunes to the *Null* node, the *Null* node is returned. If the result of pruning the child is some other node C, a unary node, U' of the same type as U is allocated, U''s child is set to C, and U' is returned as the result. This algorithm is expressed as pseudo-code in Figure 4.2.

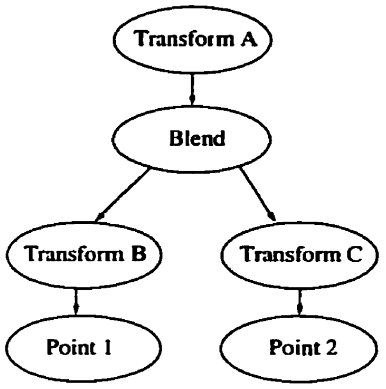
An n-ary node, N, will return itself if all of its children prune to themselves. N

will return the *Null* node if all of its children do so as well. If only one of N's children returns a non-*Null* node, C, then C is returned. The only case that remains is that at least one of the children prunes to something other than itself and at least one other child prunes to something other than the *Null* node. In this case, the set of non-*Null* pruned children is made the set of children for a newly allocated node N', of the same type as N, and N' is returned as the result.

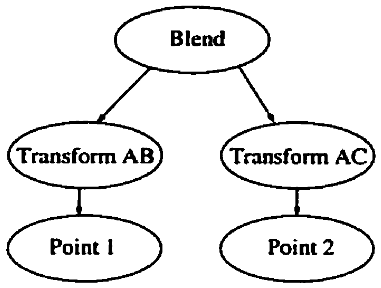
Two of the n-ary CSG nodes, *intersection* and *difference*, behave slightly differently than other n-ary nodes. An *intersection* node returns the *Null* node if *any* of its children return the *Null* node. The *difference* node returns the *Null* node if its base child (the child which all others are subtracted from) returns the *Null* node. With these exceptions, the CSG nodes behave identically to all other n-ary nodes.

One shortcoming of pruning as implemented in the *BlobTree* stems from its use of axially aligned boxes. If an ordinary box is rotated its volume does not change. If only axially aligned boxes are allowed, then this may not be the case. For example, given an axially aligned box, B_1, that is rotated and then bounded by another axially aligned box, B_2, it may be that B_2 has a larger volume than B_1. In fact, with the exception of translations and scales, all affine transformations can result in the unnecessary growth of the axially aligned box. The affect of this is that by the time the *prune* method reaches a leaf node in the *BlobTree*, its *box* parameter will be larger than necessary. This means that pruning using axially aligned boxes is unnecessarily conservative and yields *BlobTrees* that are more complex than needed.

The expense of pruning a *BlobTree* model to a sub-space is proportional to the number of nodes in the model's *BlobTree*. That is to say that if the number of nodes in a *BlobTree* model is n, then pruning is an $O(n)$ operation.



(a) Hierarchical *BlobTree* used for modeling.



(b) Reduced *BlobTree* used for rendering.

Figure 4.3: **Tree reduction creates a tree with a single transformation node above each leaf node.**

4.4 Reducing the *BlobTree*

Although spacial knowledge is useful in reducing the cost of *BlobTree* queries, it is not entirely necessary. Another strategy is to apply knowledge about the nodes of the *BlobTree* and how they can be combined with one-another to remove redundancy. This leads us to the technique called reduction which was first published in [56].

Reduction is a technique for converting a *BlobTree* into another *BlobTree* that is less costly to query. The technique exploits the fact that redundancy is usually present in the way that *affine transformation* nodes are used in a hierarchical *BlobTree* model.

Given two matrices, M_a and M_b, of the same dimensions we can combine them into a single matrix, $M_c = M_a M_b$, via matrix multiplication. Since, *affine transformation* nodes simply encapsulate a matrix, the same approach can be used to combine two *affine transformation* nodes.

Figure 4.3(a) represents an example of a *BlobTree* which may be reduced. In this case, *affine transformation A* can be combined with each of *affine transformations B* and C. The result is shown in Figure 4.3(b). Another way of thinking about the reduction results from the fact that the tree in Figure 4.3(a) can be expressed as $T_a(T_b(P_1) + T_c(P_2))$. Since *affine transformations* are distributive, the expression can be simplified to $T_a(T_b(P_1)) + T_a(T_c(P_2))$ and by recognizing that *affine transformations* can be combined, we end up with $T_{ab}(P_1) + T_{ac}(P_2)$, which is equivalent to the tree in Figure 4.3(b).

The reduction algorithm proceeds by "pushing" *affine transformations* down the *BlobTree* until they are combined into a single *affine transformation* directly above a primitive. After reduction there will be exactly one transformation between the root node and each *primitive* node.

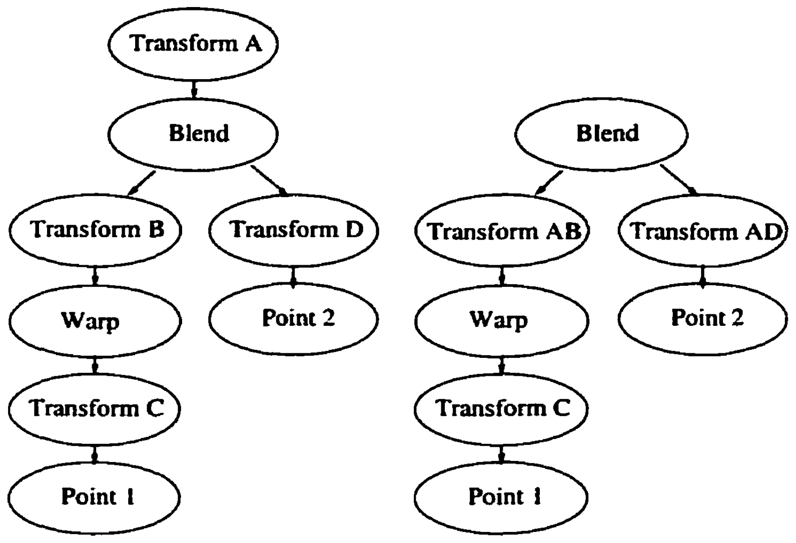
Unfortunately, there is an exceptional case that muddies an otherwise clear implementation. The exceptional case involves *warp* nodes. Since *warp* nodes are non-affine spatial transformations, they can neither be combined with *affine transformation* nodes, nor can they allow parent *affine transformation* nodes to be combined with child *affine transformation* nodes.

This case is illustrated in Figure 4.4. The *BlobTree* depicted in Figure 4.4(a) contains a *warp* node. Consistent with the above explanation, *affine transformations* A and B can be combined, but the resulting *affine transformation* cannot be pushed further down the tree to be combined with *affine transformation* C. Figure 4.4(b) illustrates the result of reduction in this case.

The expense of reducing a *BlobTree* model is proportional to the number of nodes in the model's *BlobTree*. More succinctly, if the number of nodes in a *BlobTree* model is n, then pruning is an $O(n)$ operation.

4.5 Rendering With Pruning and Reduction

The *BlobTree* allows the specification of hierarchical models. Organizational information and instancing of structures are easily expressed using the various modeling nodes of the *BlobTree*. Applying reduction or pruning to the *BlobTree* will destroy any orga-



(a) Hierarchical *BlobTree* used for modeling.

(b) Reduced *BlobTree* used for rendering.

Figure 4.4: **Transformation nodes cannot pass through warp nodes in reduction.**

nizational information. For this reason, these methods are applied as a preprocessing step before rendering. This allows model specification to take place entirely within the context of a hierarchically organized *BlobTree*, without any regard to rendering efficiency.

4.5.1 Applying Pruning to Rendering

When used in conjunction with a spatial subdivision technique, pruning can be used to build a data structure that associates a sub-space with a pruned *BlobTree*. In this research, uniform spatial subdivision is used since in most cases it is significantly faster than other methods [10, 28]. Alternative spatial data structures can also be used to associate sub-spaces with their corresponding pruned *BlobTrees*.

A naive way of building such a data structure, based on a uniform grid, could

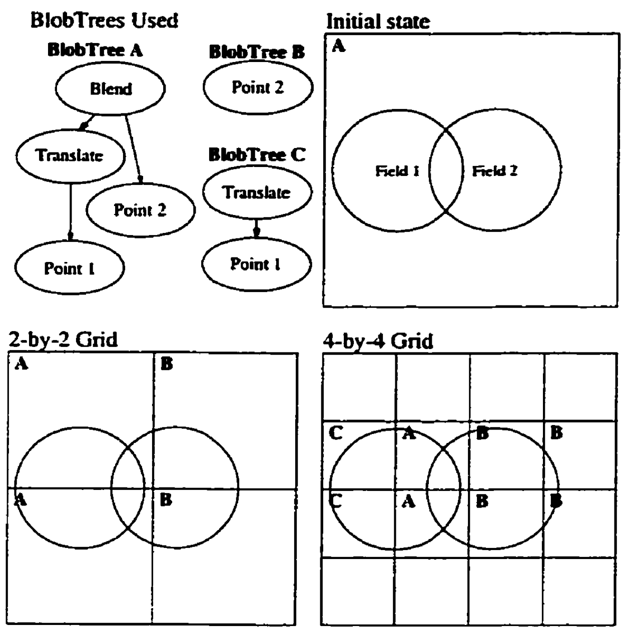


Figure 4.5: **Pruning applied to a two-by-two and four-by-four uniform grid.**

simply loop through all the voxels of the grid, using each voxel to prune *BlobTree* model and associate each voxel with the result. Since the entire model is pruned for each voxel, this approach becomes expensive as the grid's resolution, and the model's complexity, increases.

A better technique is to apply a recursive algorithm to use the result of pruning to accelerate further applications of pruning. In this research, a form of binary space partitioning was used to accomplish this task. First the spatial-volume of interest is bound with an axially aligned box. The *BlobTree* model is pruned to that box, and the box is split along one axis into two sub-boxes. Each sub-box is then used to prune its parent box's *BlobTree* and then split along another axis. The algorithm continues recursively until the boxes are no larger than the voxels in the desired uniform grid, or the pruned *BlobTree* is the *Null* node.

A two-dimensional example of the resulting pruned *BlobTrees* when using a uniform grid, is shown in Figure 4.5. Three cases are illustrated. In the first, there is one *BlobTree*, *A*, for the entire area. The extent of the field corresponding to each *point* *primitive* is indicated by a circle. In the second case, the area has been uniformly subdivided into a two-by-two grid. One additional *BlobTree*, B, has been created for the two right hand voxels which do not contain any part of the field corresponding to *Point 1*. In the third case, a four-by-four grid is used and one more *BlobTree*, C, has been created for those cells which do not contain any part of the field corresponding to *Point 2*. The result is that 50% of the area contains the *Null BlobTree*; 37.5% of the area is occupied with simpler *BlobTrees*; and the original *BlobTree* is only applied over 12.5% of the area.

4.5.2 Applying Reduction to Rendering

Since reduction requires no spatial information, its application is straight-forward. All that is required is that *reduce* be called on the *BlobTree* model, and the reduced *BlobTree* be used in place of the original *BlobTree* for all queries.

4.6 Polygonization and Pruning

As noted in Chapter 3, the method used to polygonize implicit surfaces, represented with *BlobTrees*, makes use of a uniform grid to simplify polygon generation. With a complex *BlobTree*, the bulk of the polygonizer's time is spent querying the *BlobTree* for its field value at the corners and along the edges of voxels that may intersect the implicit surface. If these queries could be accelerated, the polygonization would be accelerated as well.

To accelerate these queries, a uniform axially aligned grid, called the super grid, is created. The super grid is much coarser than the polygonization grid. Each super-voxel in the super grid stores the *BlobTree* created using the binary space partitioning scheme described in Section 4.5.1. Upon entering a voxel, the super-voxel that contains it is determined. For all the field value queries that will occur within the voxel, the pruned *BlobTree* associated with the super-voxel is used.

In addition to the initial preprocessing to build the pruned *BlobTrees* stored in the super grid, there is some small overhead to determine which super-voxel contains a particular voxel, followed by a table lookup to find the appropriate super-voxel. For a three dimensional grid, this overhead amounts to three integer divisions, two integer multiplications, and an array index operation.

4.7 Ray Tracing and Pruning

Pruning of *BlobTrees* integrates very easily with ray tracers that employ standard spacial subdivision techniques. Given that a spatial subdivision scheme is already used in the ray tracer to avoid unnecessary ray-surface intersection tests, the same spatial subdivision scheme can be used to simplify *BlobTree* models. This means that, with the exception of the initial preprocessing to prune the *BlobTrees*, no additional overhead is required.

The ray tracer used for visualization of implicit surfaces based on *BlobTrees* makes use of a uniform grid for traditional spatially based acceleration of ray tracing. In addition to traditional nodes for the camera, lights, transformations, and primitives, the scene-graph used for this ray tracer includes a *BlobTree* node. So from the perspective of the ray tracer, a *BlobTree* scene-graph node is, like a sphere or cylinder, just another primitive in the ray tracer's scene-graph.

In a manner similar to the *BlobTree*, the scene-graph is able to prune itself to an axially aligned box. The difference being that normal scene-graph primitives return a *Null* scene-graph node when the box does not contain any part of their *surface*, otherwise they return themselves. In contrast, *BlobTree* scene-graph nodes can return a new *BlobTree* scene-graph node that encapsulates a *BlobTree* that is pruned to the box. If the *BlobTree* prunes to the *Null BlobTree* node, a *Null* scene-graph node is returned.

These changes allow the ray tracer to build a uniform grid for spatial subdivision normally, with no knowledge of the *BlobTree*. When ray tracing, the uniform grid will be leveraged in two ways: to reduce the number of unnecessary ray-intersection tests with scene-graph primitives and to accelerate intersection tests with *BlobTree* scene-graph nodes by using pruned *BlobTrees*.

4.8 Reduction and Pruning Applied to Animated *BlobTrees*

Animated *BlobTrees* introduce a whole set of problems that complicate the application of reduction and pruning. These problems stem from the fact that nodes in an animated *BlobTree* can change over time. Since pruning is inherently spatially based, it must be extended to account for *BlobTrees* that change spatially over time. Reduction is not spatially based, but since it attempts to combine *affine transformations*, a way to combine animated *affine transformation* nodes must be devised. The solution to combining animated *affine transformations* for the purpose of reduction also makes a method for pruning *BlobTrees* with animated *affine transformations* possible.

4.8.1 Reducing Animated *BlobTrees*

Since animated *affine transformation* nodes represent an affine matrix that changes over time, in the form of a *matrix track*, they cannot be combined in the same way as normal *affine transformations* nodes. If two *matrix tracks*, A and B, are to be combined, one approach would be to simply multiply the resulting values of the two *matrix tracks* for the query time. This behaviour could be encapsulated in a type of *matrix track* that is defined to return the result of multiplying the values of its children for a particular query time. This approach does succeed at combining the *matrix tracks*, but yields nothing in terms of computational savings. This is due to the fact that combining *matrix tracks* in this way replaces several matrix-vector multiplications with matrix-matrix multiplications. Since it is more expensive to multiply two matrices, rather than a vector and a matrix, this technique is much slower.

The approach used for this research is to combine *matrix tracks* by approximating the result of their multiplication with a piecewise-linear approximation, which is stored in a *linearly interpolated matrix track*. The resulting *linearly interpolated matrix track* can then be used in a single animated *affine transformation* which approximates the combined affect of the set of *affine transformations* being applied. This approach replaces the expense of matrix-matrix multiplies with a linear interpolation for each element of the matrix. Not only does this yield a way of combining several animated matrices, but it also yields a method of easily applying pruning animated *BlobTrees*.

The piecewise-linear approximation is built by extending a recursive subdivision algorithm for drawing spline curves [15] to matrices. As illustrated in Figure 4.6, given a parametric curve, $C(t)$, an interval in the curve's parameter space, $[t_0, t_1]$, and an error tolerance, ϵ, the algorithm recursively subdivides the curve as long as $||C((t_0 + t_1)/2) - (C(t_0) + C(t_1))/2|| > \epsilon$. Essentially, the algorithm is unchanged, with the exception that a *matrix track* is used in place of a spline curve, matrices are used in place of spatial points, and all the calculations take place in sixteen-dimensional space.

There is one serious problem with the approximation of a temporally-dependent affine transformation with a piecewise-linear approximation. A singular $n \times n$ matrix, M, is one where the image of $\mathbb{R}^n$ is a proper subset of $\mathbb{R}^n$. Singular matrices have the property that they are not invertible, and are therefore not affine-transformations. From a modeller's perspective, it may be desirable to model an object getting squashed into a plane or shrunk to a point, but due to the way matrices are actually applied in both scene-graphs and the *BlobTree*, this behaviour is entirely unacceptable.

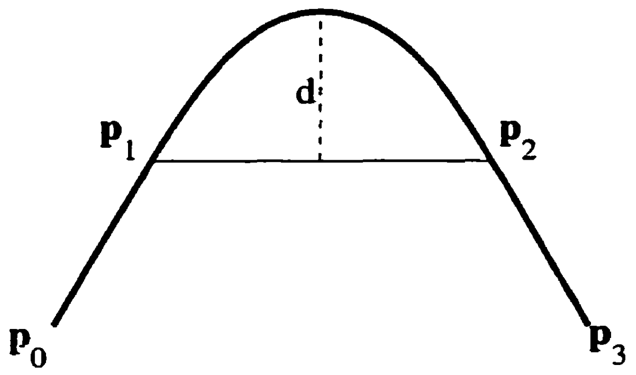


Figure 4.6: **The spline approximation algorithm recursively subdivides the curve between p_1 and p_2 as long as $d > \epsilon$.**

In most scene-graphs, and in the *BlobTree*, the inverse of an affine transformation is applied to a query point. The inverse operation of scaling an object into a plane is undefined. These problems make it desirable to disallow the use of singular matrices in *affine transformation* nodes. Unfortunately, the use of piecewise-linear approximation of temporally-dependent matrices can approximate a perfectly valid temporally-dependent matrix, that never becomes singular, with an interpolated temporally-dependent approximation that does become singular. For the purposes of this research, this problem has been ignored. For an indepth discussion of meaningful matrix interpolation without this problem, see [38].

4.8.2 Pruning Animated *BlobTrees*

With animated *BlobTrees* the problem of pruning a *BlobTree* for a particular region of space, becomes a problem of pruning for a region of space over a period of time. When looking at the problem from the local coordinate system of the leaf node, it comes down to determining whether the leaf node's field affects any of the set of points inside an axially aligned box, B, that moves over a period of time. If the moving box ever overlaps the leaf node's field, then the leaf node can not be pruned, otherwise it can be.

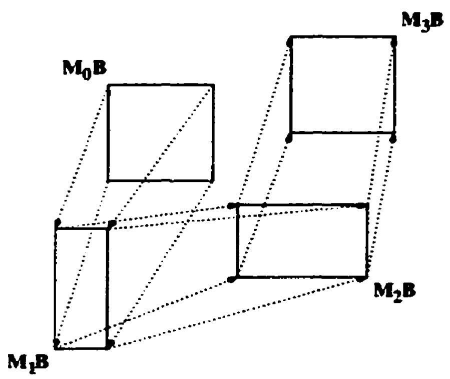
So it is necessary to determine if a leaf node's field overlaps an axially aligned box that moves over time. The leaf node's field may or may not change over time, in either case, since the leaf node's field is usually simple, it is also simple to compute an axially aligned box, B_f, that bounds the leaf node's field over the interval of time concerned. B_f can then be used in order to simplify the test against B.

If a bound for B, in the local coordinate system of the primitive and over the time interval of interest, can be generated, B and B_f can be tested against one another. If B and B_f do not overlap, it can be safely assumed that the leaf node's field does not overlap the moving axially aligned box over the time interval.

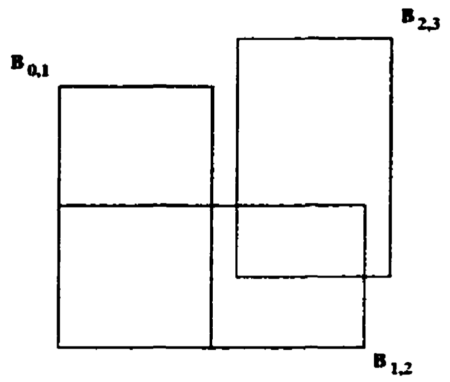
Unfortunately, it is quite difficult to generate a bound of an axially aligned bounding box that undergoes an affine transformation that can change arbitrarily over time. However, it is simple to generate a bound of an axially aligned bounding box that is transformed over time by an animated affine transformation that is specified as a piece-wise-linear interpolation of several affine matrices, $M_0, ..., M_n$, as in Figure 4.7(a). In this case, the bound is simply the union of the axially aligned bounding boxes that result from bounding B as it it transformed from M_0 to M_1, from M_1 to M_2, and so on until we reach M_{n-1} to M_n. This is illustrated in Figure 4.7(b). This bound is quite conservative for *BlobTrees* which contain primitive fields that have velocities in the direction of the world's main diagonal.

4.9 Efficient Ray Tracing of Animated *BlobTrees*

To efficiently ray trace an animated *BlobTree*, it is necessary to apply pruning and reduction. Section 4.7 contains an illustration of a data structure that associates a volume of space with a pruned and reduced *BlobTree*. With animated *BlobTrees*, a volume of *space-time* needs to be associated with a pruned and reduced *BlobTree*. For this research, a uniform axially aligned four-dimensional grid of space-time voxels is used, although hex-trees [21] could also be used and would offer the benefits of an



(a) The transformation of a bounding box by a piecewise-linear interpolated time-dependent *matrix track*.



(b) The bounds of the transformed bounding boxes between pairs of matrices.

Figure 4.7: **Given that an axially aligned box is transformed by a piecewise-linearly interpolated *matrix track*, its bound over time can be computed by taking the union of the bounds of the linear-interpolation of the box between each pair of matrices, M_i and M_{i+1}.**

adaptive spatial partitioning scheme.

The four-dimensional grid structure is built by extending the binary space partitioning technique outlined in Section 4.5.1 to four-dimensions. The data-structure is built for the whole animation rather than for each frame. This has the disadvantage that the grid must be built for the animation, rather than just a single frame. This large grid is stored in memory for the duration of rendering.

4.10 Efficient Polygonization of Animated *Blob-Trees*

Similar to ray tracing, efficient polygonization of animated *BlobTree* uses a four-dimensional axially aligned grid, called the super-grid, to reduce the cost of querying a *BlobTree* model. For simplicity, the four-dimensional super-grid is built for the entire animation, rather than individually for each frame.

A polygonization can only occur for a particular value of time, which must be specified before polygonization can begin. That time value is used, in addition to the spatial position of the voxel, to determine the super-voxel that a voxel occupies. With this exception polygonization of animated *BlobTrees* is identical to the polygonization of static *BlobTrees* explained in Section 4.6.

CHAPTER 5

Results

For testing purposes, three models were created. The first is an implicit peanut as rendered in Figure 1.3. This model is referred to as the *Peanut* model. The *Peanut* model contains only four nodes and is therefore an example of an extremely simple model. Due to its simplicity, the *Peanut* model should not be greatly affected by the application of reduction and pruning.

The second test model is the patch of grass shown in Figure 5.1. This model will be referred to as the *Grass* model. As *BlobTree* models go, the *Grass* model is quite complex, containing 4610 nodes, 2048 of which are *primitive* nodes. The *primitives* are distributed approximately evenly throughout the region of space occupied by the model. The complexity of the *Grass* model and the fairly uniform distribution of its primitives should allow reduction and pruning, as applied in the *BlobTree* system, to be used to good effect.

The third test model is four smaller patches of grass separated by a large region of empty space. This model will be referred to as the *Sparse* model, and is depicted in Figure 5.2. Just as with the *Grass* model, the *Sparse* model contains 4610 nodes, 2048 of which are *primitives*. The *primitives* are grouped in four widely separated clusters of 512 *primitives*. Although the *Sparse* model contains the same number of *primitives* as the *Grass* model, it is not expected that reduction and pruning will have as profound an affect due to the large volume of empty space separating the

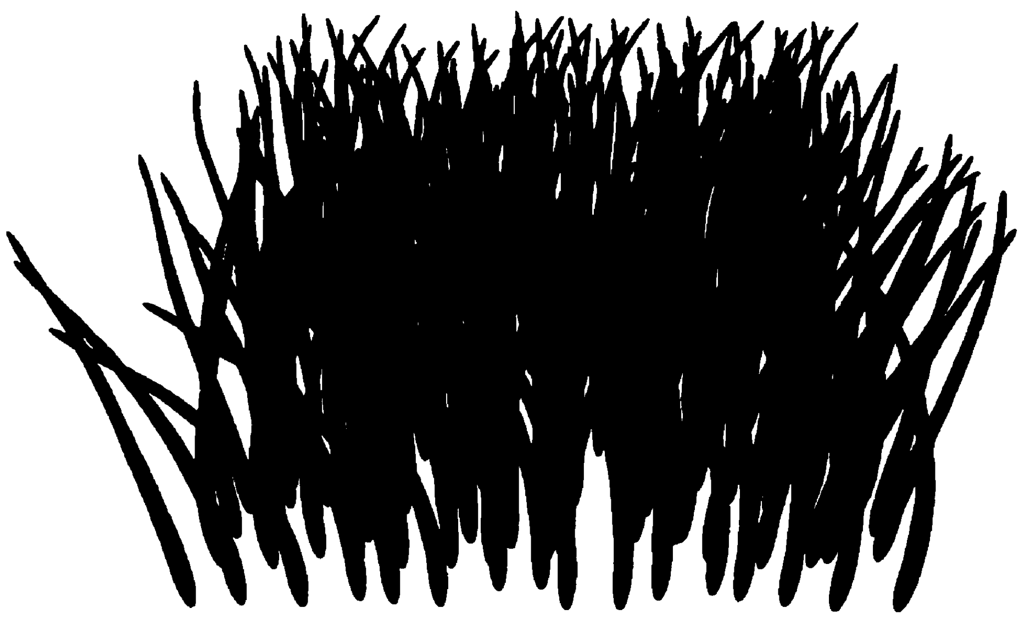


Figure 5.1: **A rendering of the *Grass* model.**

primitives.

Each model was polygonized and ray traced with a variety of grid resolutions for pruning purposes. The highest grid resolution used was $64 \times 16 \times 64$. This limit was due to the large memory overhead of having thousands of *BlobTree* models in memory at the same time. Higher grid resolutions required the use of virtual memory and ran exceptionally slow.

The preprocessing time required to generate the pruned *BlobTrees* and the average number of nodes, n of the resulting *BlobTrees*, is summarized in Table 5.1. The relative speed-ups, over non-reduced and non-pruned *BlobTrees*, for polygonization of each model at several grid-resolutions is tabulated in Table 5.2. Similar results for ray tracing are given in Table 5.3.

As shown in Table 5.1, the amount of time spent building a data-structure to associate voxels of space with their associated *BlobTree* can become significant as high grid resolutions are reached. With preprocessing times as high as 208 seconds

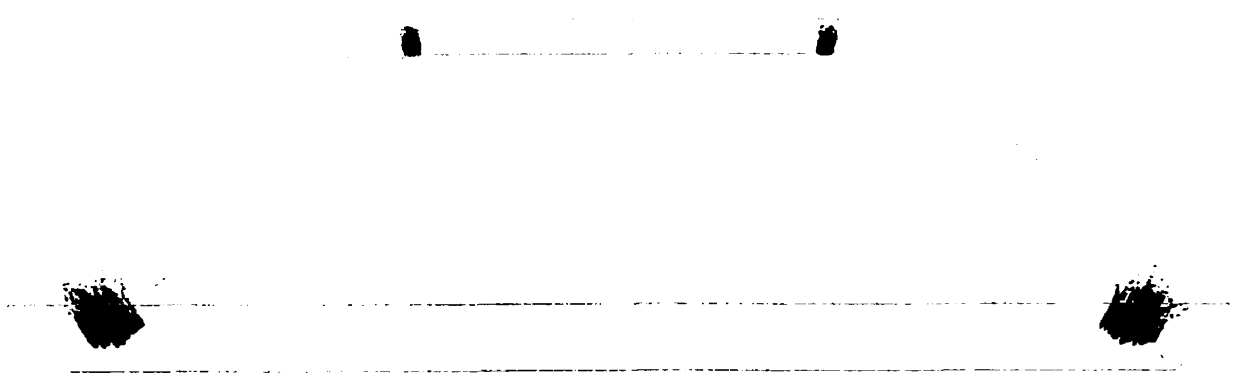


Figure 5.2: **The *Sparse* model, composed of four small patches of grass separated by a large region of empty space.**

for the *Grass* model, it is obvious that the grid resolution must be carefully weighed against the potential savings at rendering time.

The algorithm (as outlined in Section 4.5.1) for building the super-voxel grid *BlobTrees* is efficient. The maximum increase in cost for generating eight times as many voxels is only by a factor of 5.16. The minimum increase in cost is only by a factor of 1.18. This is due to the fact that, as the algorithm subdivides space, it is working with smaller and smaller *BlobTrees*, which are less expensive to prune.

Table 5.1, also shows, n, the average number of nodes in the pruned *BlobTrees* over all the voxels in the super-voxel grid. This average is not weighted by the area of the voxels, which explains the fact that occasionally it does increase slightly. As we can see, the decrease in n strongly depends on the complexity of the *BlobTree* model. In the case of both the *grass* and *sparse* models, there is still a significant decrease in n at the highest grid resolutions tested. This implies that, with these two models, there is still more efficiency that can be gained from further increases in grid resolution.

From Tables 5.2 and 5.3, it can be seen that for simple models, such as the *Peanut* model, there is little benefit, possibly even a cost, to the application of pruning and

Grid Resolution	Preprocessing Results					
	Peanut		Grass		Sparse	
	PPT	n	PPT	n	PPT	n
1×1×1	0	4	0	4610	0	4611
4×4×4	.025	3.00	22.2	258	14.5	878
8×8×8	.110	3.20	46.1	71.4	27.0	752
16×16×16	.568	2.82	91.1	27.4	51.7	351
32×16×32	1.58	2.89	128	14.0	61.2	148
64×16×64	5.24	2.80	208	9.20	75.5	58.2

Table 5.1: **preprocessing time (PPT) in seconds and average number of nodes (n) for pruned *BlobTrees* in the super-voxel grid.**

Grid Resolution	Polygonization Speed-Up		
	Peanut	Grass	Sparse
1×1×1	1	1	1
4×4×4	0.982	18.0	5.06
8×8×8	0.998	57.3	6.02
16×16×16	1.03	138	8.38
32×16×32	1.03	207	21.7
64×16×64	1.03	243	32.9

Table 5.2: **Relative speed-up results for polygonizing with both reduction and pruning applied.**

Grid Resolution	Ray Tracing Speed-Up		
	Peanut	Grass	Sparse
1×1×1	1	1	1
4×4×4	1.0	15.9	5.63
8×8×8	0.765	54.0	10.0
16×16×16	0.292	138	22.5
32×16×32	0.163	248	45.0

Table 5.3: **Relative speed-up results for ray tracing with both reduction and pruning applied.**

reduction. It should be noted that the application of reduction alone would yield some efficiency gains due to the fact that an *affine transformation* node would be removed . The collected data illustrates the potential of pruning and reduction to greatly accelerate the rendering of complex *BlobTree* models. The largest speed-up observed is nearly 250 for the *Grass* model.

These speed-ups make it possible to model complex and realistic models such as the sea-shell, in Figure 5.3. This model was rendered in $1\frac{1}{2}$ hours using the techniques presented in this paper. Without the application of reduction and pruning rendering takes over 90 hours, which makes the construction of such a model infeasible.

The running time of *BlobTree* evaluations is $O(n)$ with n nodes. Expected speed-ups can be obtained from reduction of the average number of nodes in the *BlobTrees* being evaluated, which is given in Table 5.1. The expected speed-up can be compared to the actual speed-ups recorded in Tables 5.2 and 5.3.

With low grid resolutions, the expected speed-up is close to the observed speed-up. With large grid resolutions, the actual speed-up is less than expected due to the increased over-head of traversing voxels. The exception is the ray tracing of the *Sparse* model, which is faster than expected by a factor of almost 2. This is due to the fact that no *BlobTree* traversals are necessary in the large areas of empty space. By contrast, polygonization always follows the surface, and because of this, does not gain any benefit from pruning of empty space.

Since a uniform grid is employed for spatial partitioning, the results with the *Sparse* model, which contains the same number of nodes as the *Grass* model, do not show as large a speed-up as with the *Grass* model. This is expected and could be alleviated by the application of an adaptive spatial partitioning technique.

A strength of pruning is that it can generate the minimum possible collection of primitives for a given region of space. This does not actually occur in the current implementation due to the use of axially aligned boxes and the fact that *affine transformation* nodes and *warp* nodes can cause them to grow unnecessarily. The algorithm described in [19] cannot produce a minimal collection of primitives for a region of space. The algorithms described in [19, 37] both show speed-ups, however they require the calculation of Lipschitz constants which imposes a restriction on the class of functions which can be rendered with their techniques.

Pruning and reduction compares favorably with a previous optimization method described in [51], which requires intersection tests with bounding boxes to determine whether to avoid traversal of sub-trees which do not affect the current query point. Speed-ups obtained using the bounding box method are highly dependent on the structure of the *BlobTree* used to describe the model. Because entirely different *BlobTrees* can be used to describe the same implicit surface, speed-ups based on this algorithm can vary dramatically for the same implicit surface. For example, with the *Grass* model, the bounding box method produced a speed-up of 26.6. An alternate *BlobTree* for the *Grass* model produced a speed-up of only 3.5 when using the bounding box method. Contrast this with the reduction and pruning method where unneeded sub-trees are simply removed. Due to this fact, the speed-ups are similar for any *BlobTree* used to represent the model. For the two formulations of the *Grass* model mentioned above, the speed-up was 257 in both cases.

The *Sparse* model is a case where the bounding box scheme performed slightly better than pruning and reduction. This is due to the *Sparse* model containing a large amount of empty space as well as small areas of densely packed *BlobTree* primitives. Because of this, sufficiently small *BlobTrees* cannot be created without using an excessively fine grid, which is impractical. An adaptive space partitioning scheme would be better able to leverage pruning in this case.

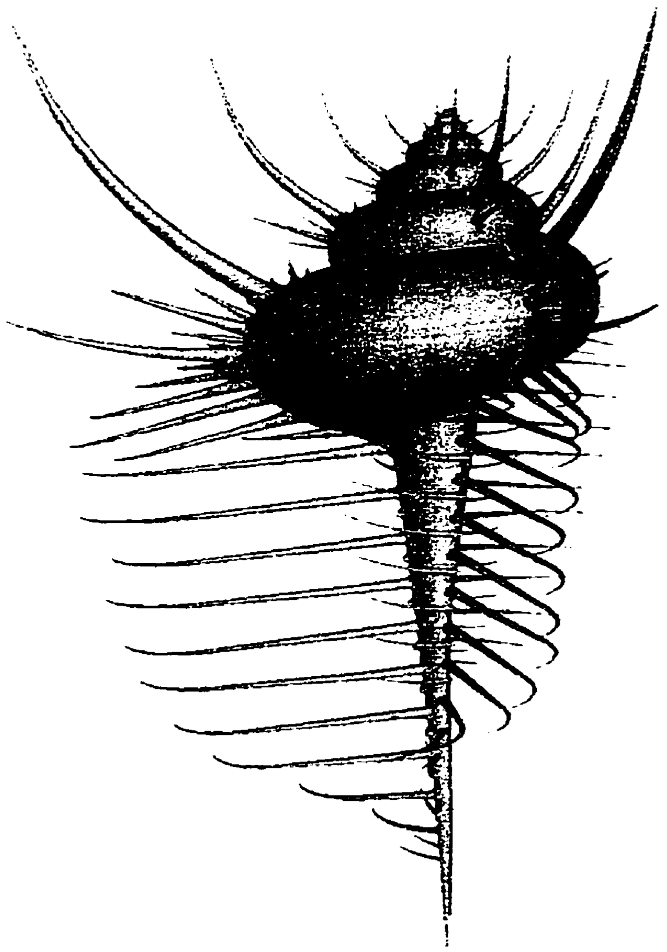


Figure 5.3: **A complex model built using the *BlobTree*. (Image courtesy of Callum Galbraith.)**

CHAPTER 6

Conclusions and Future Work

6.1 Summary

This research has found that the application of reduction, a global optimization strategy, and pruning, a spatially aware local optimization strategy, in an implicit surfaces modeling and animation system can yield significant efficiency gains. These gains are most pronounced with large models containing hundreds or even thousands of primitive field functions.

The modification of the *BlobTree* system, a system for implicit surfaces modeling and animation, to leverage both reduction and pruning has significantly increased the complexity of models that it is practical to work with. The best observed speed-up to date has been more than two orders of magnitude. Such significant speed-ups allow the users of the *BlobTree* system to attempt to model objects and scenes that were simply not possible, due to computational constraints, in the past.

An explanation of the design and implementation of the *BlobTree* system has been included. The extension of the *BlobTree* system to include reduction and pruning has also been explained. It has been shown that the incorporation of both techniques into the *BlobTree* itself is both simple and straightforward. A description of how the *BlobTree* polygonizer can be extended to support pruning has shown that it is reasonably easy to do so. A similar description of how the *BlobTree* ray tracer can be extended to support pruning has detailed many of the choices that must be made. An analysis of the performance of the *BlobTree* system with and without reduction and pruning has shown how the techniques can best be applied in both the polygonizer and the ray tracer to increase rendering efficiency.

Lastly, algorithms which attempt to reduce the number of field evaluations, such as [46], can easily be layered on top of our optimized *BlobTrees* to generate additional speed-ups.

6.2 Thesis Contributions

The contributions made in this research can be separated into four general tasks:

- A complete redesign and implementation of the *BlobTree* system. This includes the graphics utility and math libraries that the *BlobTree* is based upon, the *BlobTree* library itself, the *BlobTree* polygonizer, and the *BlobTree* ray tracer. A requirement of the new design and implementation is the integrated support for animation within the *BlobTree*. The new system also makes much better use of object-oriented features of the C++ programming language in order to make the system's implementation smaller and more maintainable.

- The modification of the *BlobTree* library to incorporate the *prune* and *reduce* operations to allow for less computationally expensive *BlobTree* evaluations.

- The redesign and implementation of both the *BlobTree* polygonizer and ray tracer to incorporate spatially aware data-structures in order to leverage pruning.

- A comparison and analysis of the performance of the unmodified and modified *BlobTree* system.

The completion of these tasks has resulted in an extendible and efficient system for modeling and animating with implicit surfaces. The performance analysis not only verified that the new system is usually much more efficient than past *BlobTree* implementations, but also exposed where the new system excels and how it can be improved.

6.3 Future Work

The *BlobTree* has recently been extended to support temporally dependent information, giving us animatable *BlobTrees*and making rendering effects, such as accurate motion blur possible. Animatable *BlobTrees* are much more complicated both to prune and to reduce. Although, pruning of animatable *BlobTrees* has been described in this thesis, a more in depth examination is required.

In animatable *BlobTrees*, *affine transformation* nodes may become much more expensive to traverse if they are representing temporally-dependent transformations. A potential solution to this is a caching scheme that will cache data that can be reused when time does not change between queries. Caching can also be used to improve the efficiency of nodes that, to complete a single traversal of themselves, require multiple traversals of their children.

Since reduction is a global optimization strategy for *BlobTrees*, the reduction algorithm explained in this paper can be improved upon by incorporating techniques from the compiler field. For a concrete example, it is possible that pushing an animated *affine transformation* node down the tree can in fact increase the expense of a field evaluation by increasing the complexity of many underlying non-animated *affine transformation* nodes. In this case our tree-reduction algorithm, which is minimizing the number of *affine transformation* nodes, may increase the expense of querying the *BlobTree*. Input from the fields of compiler and graph theory could be used to produce algorithms which could determine whether the expense of transforming child non-animated *affine transformation* nodes into animated *affine transformation* nodes would outweigh the savings of combining several *affine transformation* nodes.

It is desirable to have a faster method of creating data-structure that maps volumes of space to their corresponding pruned *BlobTrees*. One potential improvement is to use planar half-spaces rather than axially aligned boxes to prune a *BlobTree*, in combination with the recursive pruning scheme described in Section 4.5.1. This could offer a solution to the problem of the axially aligned boxes growing due to transformations.

Since ray tracers that make use of spatial subdivision must use a mail box or ray signature algorithm to avoid testing a ray against a primitive more than once [14], an important modification to pruning should be made. The modification ensures that no sub-tree of the scene-graph is instantiated more than once.

Another method to reduce the expense of pruning may be to apply a lazy algorithm to create only the needed parts of the pruned data-structure. Applying a lazy method to the polygonizing algorithm would only require that the *BlobTree* model be pruned to the super-voxel the first time the super-voxel is entered. In a ray tracer this approach would have to be applied to the whole scene-graph the first time a ray enters a voxel. In both cases, this would mean that only the spaces of interest would incur the expense of pruning.

Bibliography

[1] V. Adzhiev, R. Cartwright, E. Fausett, A. Ossipov, A. Pasko, and V. Savchenko. HyperFun Project: A Framework for Collaborative Multidimensional F-rep Modeling. *Implicit Surfaces*, pages 59–69, September 1999.

[2] E. L. Allgower and K. Georg. *Numerical Continuation Methods: An Introduction*, volume 13 of *Series in Computational Mathematics*. Springer Verlag, Berlin, Heidelberg, New York, 1990. pp 388.

[3] Alan H. Barr. Global and local deformations of solid primitives. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 21–30, July 1984.

[4] James Blinn. A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics*, 1:235, 1982.

[5] Jules Bloomenthal. Polygonisation of Implicit Surfaces. *Computer Aided Geometric Design*, 4(5):341–355, 1988.

[6] Jules Bloomenthal. An Implicit Surface Polygonizer. *Graphics Gems IV*, pages 324–349, 1994. Edited by Paul Heckbert.

[7] Jules Bloomenthal. *Introduction to Implicit Surfaces.* Morgan Kaufmann, ISBN 1-55860-233-X, 1997. Edited by Jules Bloomenthal With Chandrajit Bajaj,

Jim Blinn, Marie-Paule Cani-Gascuel, Alyn Rockwood, Brian Wyvill, and Geoff Wyvill.

[8] Andrea Bottino, Wim Nuij, and Kess van Overveld. How to Shrinkwrap a Critical Point: An Algorithm for the Adaptive Triangulation of Iso-surfaces with Arbitrary Topology. *Implicit Surfaces, Proceedings of Second Eurographics Workshop on Implicit Surfaces, Eindhoven, Holland.*, October 1996.

[9] Michael Chmilar. A Kernel for Computer Animation. Master's thesis, Department of Computer Science, University of Calgary, Calgary, Canada, 1990.

[10] John Cleary and Geoff Wyvill. Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision. *The Visual Computer*, 4(2), 1988.

[11] B. Crespin, C. Blanc, and C. Schlick. Implicit sweep objects. In *Eurographics '96*, volume 15, pages 165–174, August 1996.

[12] Mathieu Desbrun, Nicolas Tsingos, and Marie-Paule Gascuel. Adaptive sampling of implicit surfaces for interactive modeling and animation. *Implicit Surfaces*, pages 171–185, April 1995.

[13] Tom Duff. Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry. *Computer Graphics*, 26(2):109–116, July 1992.

[14] Andrew Glassner (editor). *An Introduction to Ray Tracing.* Academic Press, San Diego, CA, 1989.

[15] James D. Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics Priniciples and Practice.* Addison-Wesley, 1990.

[16] Mark Fox, Callum Galbraith, and Brian Wyvill. Efficient Implementation of the BlobTree for Rendering Purposes. *Western Computer Graphics Symposium*, pages 47–54, March 2000.

[17] Mark Fox, Callum Galbraith, and Brian Wyvill. Efficient Use of the BlobTree for Rendering Purposes. *Shape Modeling and Applications*, pages 306–314, May 2001.

[18] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications*, 1986.

[19] Jean-Dominique Gascuel. Implicit patches: An optimised and powerful ray intersection algorithm. In *Implicit Surfaces '95*, April 1995.

[20] Andrew Glassner. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[21] Andrew Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, pages 60–70, March 1988.

[22] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning. *IEEE Computer Graphics and Applications*, 9(3):20–28, May 1989.

[23] Andrew Guy and Brian Wyvil. Controlled blending for implicit surfaces. In *Implicit Surfaces '95*, April 1995.

[24] Andrew W. P. Guy. Building Blocks for Implicit Surfaces. Master's thesis, Department of Computer Science, University of Calgary, Calgary, Canada, 1998.

[25] Pat Hanrahan. Ray Tracing Algebraic Surfaces. *Computer Graphics (Proc. SIGGRAPH 83)*, 17(3):83–90, July 1983.

[26] J. C. Hart and T. A. Defanti. Efficient antialiased rendering of 3-d linear fractals. *Computer Graphics*, 25(3), 1991.

[27] John C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(9):527–545, 1996. ISSN 0178-2789.

[28] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. *Proc. Graphics Interface 1989*, pages 164–172, 1989.

[29] D. Kalra and A. Barr. Guaranteed Ray Intersections with Implicit Functions. *Computer Graphics (Proc. SIGGRAPH 89)*, 23(3):297–306, July 1989.

[30] D.P. Mitchell. Robust Ray Intersection with Interval Arithmetic. *Proceeding of Graphics Interface*, pages 68–74, May 1990.

[31] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.

[32] Paul Ning and Jules Bloomenthal. An evaluation of implicit surface tilers. *IEEE Computer Graphics and Applications*, 13(6):33–41, November 1993.

[33] H. Nishimura, A. Hirai, T. Kawai, T. Kawata, I .Shirakawa, and K. Omura. Object Modelling by Distribution Function and a Method of Image Generation. *Journal of papers given at the Electronics Communication Conference '85*, J68-D(4), 1985. In Japanese.

[34] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 2(8):429–446, 1995.

[35] A. Ricci. A constructive geometry for computer graphics. *Computer Journal*, 16(2):157—160, May 1973.

[36] S. D. Roth. Ray Casting for Modelling Solids. *Computer Graphics and Image Processing*, 18:109–144, February 1982.

[37] Andrei Sherstyuk. Fast Ray Tracing of Implicit Surfaces. *Computer Graphics Forum*, 18(2):139–148, June 1999.

[38] Ken Shoemake and Tom Duff. Matrix animation and polar decomposition. *Graphics Interface '92*, pages 258–264, May 1992.

[39] Bjarne Stroustrup. *The C++ Programming Language (Third Edition).* Addison-Wesley, Reading, MA, 1997.

[40] Mark Tigges. Two Dimensional Texture Mapping of Implicit Surfaces. Master's thesis, Department of Computer Science, University of Calgary, Calgary, Canada, 1999.

[41] Mark Tigges and Brian Wyvill. Texture Mapping the BlobTree. *Implicit Surfaces*, 3, June 1998.

[42] Mark Tigges and Brian Wyvill. A field interpolated texture mapping algorithm for skeletal implicit surfaces. *Proc. CG International 99*, pages 25–33, 1999.

[43] Mark Tigges and Brian Wyvill. Python for scene and model description for computer graphics. *Proc. IPC 2000*, January 2000.

[44] Kees van Overveld and Brian Wyvill. Potentials, Polygons and Penguins. An efficient adaptive algorithm for triangulating an equi-potential surface . In *Proc. 5th Annual Western Computer Graphics Symposium (SKIGRAPH 93)*, pages 31-62, 1993.

[45] Alan Watt. *Fundamentals of Three-Dimensional Computer Graphics.* Addison-Wesley, Reading, MA, 1989.

[46] Andrew Witkin and Paul Heckbert. Using particles to sample and control implicit surfaces. *Computer Graphics (Proc. SIGGRAPH 94)*, 28:269–277, July 1994.

[47] J. R. Woodwark and K. M. Quinlan. The derivation of graphics from volume models by recursive subdivision of the object space. *Computer Graphics 80, Proceedings of a Conference at Brighton*, pages 335–343, August 1980. Held in Northwood Hills, Middx..

[48] Brian Wyvill. SOFT. *SIGGRAPH 86 Electronic Theatre and Video Review,* Issue 24, 1986.

[49] Brian Wyvill. The Great Train Rubbery. *SIGGRAPH 88 Electronic Theatre and Video Review*, Issue 26, 1988.

[50] Brian Wyvill. Warping Implicit Surface for Animation Effects. In *Proc. Western Computer Graphics Symposium (SKIGRAPH 92)*, pages 55–63, 1992.

[51] Brian Wyvill, Eric Galin, and Andrew Guy. Extending The CSG Tree. Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *Computer Graphics Forum*, 18(2):149–158, June 1999.

[52] Brian Wyvill and Kees van Overveld. Polygonization of Implicit Surfaces with Constructive Solid Geometry. *Journal of Shape Modelling*, 2(4):257–274, 1996.

[53] Brian Wyvill and Geoff Wyvill. Field functions for iso-surfaces. *The Visual Computer*, 5(1/2):75–82, March 1989.

[54] G. Wyvill and P. Sharp. Volume and surface properties in csg. *New Trends in Computer Graphics (Proceedings of CG International '88)*, pages 257–266, 1988.

[55] G. Wyvill and A. Trotman. Ray tracing soft objects. *Proc. CG International 90*, pages 469–476, 1990.

[56] Geoff Wyvill and Tosiyasu Kunii. A Functional Model for Constructive Solid Geometry. *The Visual Computer*, 1(1):3–14, July 1985.

[57] Geoff Wyvill, Tosiyasu Kunii, and Yasuto Shirai. Space Division for Ray Tracing in CSG. *IEEE Computer Graphics and Applications*, 6(4):28–33, April 1986.

[58] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data Structure for Soft Objects. *The Visual Computer*, 2(4):227–234, February 1986.