IMPS: Implicit Surfaces for Interactive Animated Characters

by
Kenneth Bradley Russell

B.S., Computer Science and Electrical Engineering
Massachusetts Institute of Technology, Cambridge, MA
June 1997

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE IN MEDIA ARTS AND SCIENCES
at the
Massachusetts Institute of Technology
June 1999

Signature of Author _____

Program in Media Arts and Sciences
May 7, 1999

Certified by _____

Bruce M. Blumberg
Asahi Broadcasting Corporation Assistant Professor of Media Arts and Sciences
MIT Media Laboratory
Thesis Supervisor

Accepted by _____

Stephen A. Benton
Chairperson
Departmental Committee on Graduate Students, Program in Media Arts and Sciences
MIT Media Laboratory

IMPS: Implicit Surfaces for Interactive Animated Characters

by
Kenneth Bradley Russell

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
on May 7, 1999
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

Abstract

Implicit surface modeling in computer graphics is a powerful technique for representing smooth and organic shapes. Skeletal elements of an implicit surface blend to create a smooth, seamless skin which exhibits desired properties for animation such as squash and stretch. Because of their high computational cost to render, implicit surfaces have not been used extensively in the real-time graphics domain. This thesis discusses the problems and some solutions in the application of implicit surfaces to the domain of interactive character animation. A design process for an implicit surface-based character is proposed, from the modeling and texturing stages to animation and rendering.

Thesis Supervisor: Bruce M. Blumberg
Title: Asahi Broadcasting Corporation Assistant Professor of Media Arts and Sciences

IMPS: Implicit Surfaces for Interactive Animated Characters

by

Kenneth Bradley Russell

The following people served as readers for this thesis:

Reader: _____

Alex P. Pentland
Toshiba Professor of Media Arts and Sciences
MIT Media Laboratory

Reader: _____

Luiz Velho
Associate Professor
Instituto de Matematica Pura e Aplicada

Acknowledgments

I thank my parents for their constant love and support.

I give thanks to my advisor, Bruce Blumberg, for being a mentor, advisor, and friend for the past several years; to my undergraduate work advisor, Sandy Pentland, for his support, advice, and friendship during my undergraduate and graduate years; and to Luiz Velho, for his many helpful suggestions and for being an excellent and objective reader.

Thanks to Hans Pedersen for many helpful discussions and suggestions; my lifting partner, Tom Minka, for listening to all of the problems and offering helpful advice; Sumit Basu, for several helpful discussions; and my officemate and groupmate, Mike Hlavac, for the fun weekend projects like Melanie and Marco Pollo which kept both of us going.

Thanks to the rest of the Synthetic Characters, Vision and Modeling, and Software Agents groups for an educational and fun few years.

Contents

List of Figures

Chapter 1

Introduction, Definitions, and Motivation

This thesis proposes a real-time, interactive system for modeling animated characters using implicit surfaces.

Most current systems for creating 3D interactive animated characters [17, 18, 19, 20] use the same polygonal representation for a character's geometry. A character is typically organized into a *skeleton* of component pieces of static geometry, or limbs, and animation is performed by modifying the transforms, or joints, connecting them.

The fundamental requirement for any modeling technique for an interactive animated character is that the character be able to express its personality. This broad statement implies many specific criteria. First, the system as a whole must be fast. A character must not be limited to six frames per second if thirty are required to show its motion in adequate detail. In addition, in order for the character to appear responsive to the user, its latency must be low, which implies a high frame rate. Second, the technique must support the desired visual appearance of the character. In our case we are not striving for a realistic appearance, but a cartoon-like one. Third, and most important, the modeling technique must give the artist who is designing the character enough control over its body to create the compelling motion which will convey the character's personality. In particular, many of the elements from classical animation such as squash, stretch and follow-through [15] should be supported.

This thesis discusses the problems and some solutions in the application of implicit

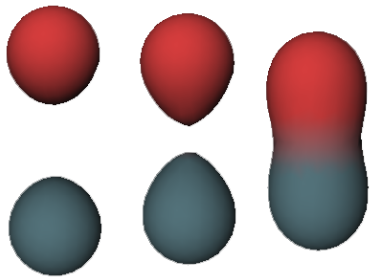

Figure 1-1: Blending of two implicit spheres.

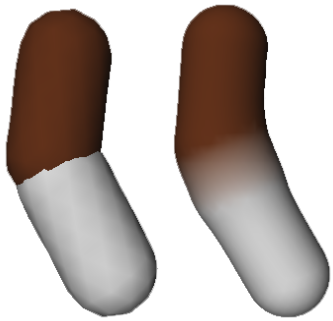

Figure 1-2: Creasing vs. Blending.

surfaces to the domain of interactive character animation. An *implicit surface*, in three dimensions, is a constant-value surface, or *isosurface*, of a function of three variables $f(x, y, z)$. For example, rendering the function $x^2 + y^2 + z^2 = r^2$ for a particular value of r produces a sphere. The surface is *implicitly* defined by the function $f(x, y, z)$ and the constraint $f = r^2$, rather than, for example, B-spline patches, which have an *explicit* parameterization of (u, v) coordinates which range over $([0..1], [0..1])$ to traverse the surface.

Implicit surfaces were introduced into the field of computer graphics by Blinn [1], who proposed the idea of blending. In his formulation, an implicit sphere is represented as a scaled three-dimensional Gaussian distribution, and the implicit function is the sum of the component Gaussians. Two implicit spheres, or *metaballs*, which approach each other appear to blend because of the overlap of the two distributions (Figure 1-1.) The function representing the sum of the two spheres' contributions is called the *potential function* or *field function*, because of its similarity to the potential field induced by the presence of two charged particles such as electrons.

Implicit surfaces appear to provide many advantages over the standard polygonal representation for animated characters. This thesis was originally inspired by the ease with

which implicit surfaces model smooth and organic shapes, which appear to be ideal for modeling cartoon-like characters; in fact, many papers on implicit surfaces are motivated by this property. Their underlying mathematical representation also provides benefits such as squash and stretch as elements approach and separate. In addition, an implicit surface can be tesselated into a single seamless mesh using a polygonization algorithm, whereas standard polygonal objects exhibit creases where component parts overlap (for example, at the elbows and knees of a character: see Figure 1-2.) Unfortunately, implicit surfaces are much more computationally expensive to render than rigid polygonal objects, which has prevented their widespread use in interactive systems.

This thesis is not the first work to study this application domain for implicit surfaces; specifically, Opalach and Maddock [8] created a "Disney-style" animation system, which simulated effects such as skin and hair using dozens of metaballs attached to the surface of an underlying implicit surface using springs. This thesis differs from their work in two fundamental ways. First, it does not attempt to model the physics of skin and hair with metaballs, but instead emphasizes a low primitive count to achieve a "cartoony" look. Second, it aims for real-time rendering and interactive character design.

1.1 Related Work

The field of implicit surfaces has evolved a great deal over the past decade, with much work in many areas. To make organization of the background work easier the following sections will be devoted, in turn, to techniques for blending, rendering, and texture mapping and design. The contributions of IMPS will then be discussed.

1.1.1 Field Functions and Blending Control

Much research has been done to find blending functions which are inexpensive to compute and create useful shapes; this section highlights only a few of these results. Blinn's original paper [1] used a quadric in the exponent of the Gaussian distribution, and suggested the use of higher-order exponents to create hyperellipsoidal shapes which are more square and less blobby. Wyvill and Wyvill [21] illustrated the use of implicit hyperellipsoids and proposed another family of functions with compact support to limit the effects of a particular skeletal element. Sclaroff and Pentland [22] described a framework for adding modal deformations

and displacement maps to skeletal elements.

The problem of *unwanted blending* is as old as the field of implicit surfaces and can be summed up by the desire to have an arm blend with the shoulder and the shoulder with the torso, but not the arm with the torso. Most implicit surface systems contain a solution to this problem; Guy and Wyvill [23], for example, used a graph to specify which skeletal elements should blend and which should not, and described a blending function which takes the graph into account during its computation.

Blanc and Schlick [24] derived a computationally efficient set of blending functions which have finite support, unlike the exponential function. Their focus is on anisotropic distance functions, which allow solids of revolution to be more easily specified using implicit surfaces.

1.1.2 Sampling and Tesselation

Wyvill and McPheeters [25] and Lorensen and Cline [26] independently developed an isosurface rendering technique commonly called Marching Cubes, which is still the standard method for rendering polygonal approximations in most molecular visualization packages. The Marching Cubes algorithm samples the potential function at the vertices of a regular 3D grid and determines the intersection of each voxel with the desired isosurface, rendering each intersecting voxel with some number of triangles. The primary advantage of this algorithm is its simple implementation; Watt and Watt's [29] is especially elegant. The disadvantage is that the algorithm is global rather than incremental, and therefore any change in the surface requires the entire voxel grid to be recomputed and traversed. Opalach and Maddock [34] attempted to localize the effects of surface motion to varying degrees of success. Problems in the original MC algorithm were noted by Düurst [27] and corrected by Nielson and Hamann [28]. Variations on the algorithm include a physically-based polygonizer which adapts the voxel shape to the surface being rendered, developed by de Figueiredo et. al [30], and Bloomenthal's non-manifold polygonizer [32].

Most non-raytracing implicit surface rendering techniques were variants of the Marching Cubes algorithm until Witkin and Heckbert [31] devised a physically-based sampling technique which simulates a set of particles drifting on an implicit surface. Two advantages of their algorithm over previous particle systems are its ability to rapidly fill in poorly-sampled regions of the surface via particle growth and fusion, and the regularity of the resulting sampling. This paper sparked new work in many areas of implicit surface research

such as polygonization and texture mapping.

Rodrian and Moock [36] devised a variant of Witkin and Heckbert's sampling algorithm which was designed to produce closed surfaces composed of triangles rather than disconnected samples. They simulate springs rather than particle repulsion, and add heuristics for handling topology changes and "folding" of the mesh which can easily occur with spring-based simulations. They also discuss modifications for curvature adaptivity and accelerated convergence by applying the skeletal motion directly to the mesh vertices. This thesis will present a much simpler algorithm for tesselation based on the Witkin-Heckbert algorithm which preserves the attractive dynamics of their simulation and requires fewer iterations to converge.

Stander and Hart [40] focused on guaranteeing the topology of an implicit surface polygonization by tracking the critical points of the potential function. Their system reaches real-time rates by combining the Witkin-Heckbert sampling technique with an incremental update method for the polygonization, avoiding the need to triangulate the entire surface each simulation step.

Rösch, Ruhl, and Saupe [37] applied Witkin-Heckbert sampling to non-manifold and infinite-extent implicit surfaces by adjusting the particle radius according to estimates of surface curvature and proximity to a critical point, as well as constraining the simulation to a bounding volume.

Desbrun, Tsingos, and Gascuel [33] proposed a new sampling and tesselation technique for implicit surfaces which samples each skeletal element by performing root-finding along fixed rays in its local coordinate system. This algorithm has many advantages including the avoidance of physical simulation and use of temporal information. However, it does not produce as regular a sampling as Witkin and Heckbert's technique.

1.1.3 Texture Mapping and Design

Implicit surfaces, in general, model smooth shapes well but finely detailed ones poorly. Adding textures increases visual fidelity and allows the placement of local details which do not directly correspond to the locations of skeletal elements. Here we are concerned with the placement of 2D textures on the implicit surface, rather than 3D, or solid, textures [44] [45], which are better suited for representing shapes hewn out of wood or marble than animated surfaces.

Pedersen's work on texturing both implicit surfaces [47] and general curved surfaces [48] is arguably the optimal method for texturing such shapes. The user specifies a set of bicubic patches which provide a parameterization of the surface with certain desirable properties at boundary conditions between the patches. This underlying parameterization can then be used transparently by the artist to interactively place texture patches, or *patchinos*, onto the surface. It also allows, for example, flood fills to be performed within user-specified curves on the surface.

Suggestions are made at the end of Pedersen [47] as to the applicability of his technique to animated surfaces. In particular, "pinning" the corners of the bicubic patches to the underlying implicit skeleton would allow the "skin" to "stretch" as the surface moved. While this method would not allow topology changes, this is not a severe limitation in the domain of character animation. Stretching is arguably the most intuitive response of the surface to animation. Unfortunately, the distortion minimization process required to keep the patches on the surface as it moves is not yet a real-time operation.

Zonenschein, Gomes, Velho and de Figueiredo [49] generated texture coordinates for an implicit surface by surrounding it with a support surface with a well-defined parameterization (such as a cylinder) and following gradient lines from the support surface down to the implicit surface. Their later work with Tigges and Wyvill [51] combines their technique with the BlobTree [50] to provide a local support surface per implicit element. This reduces the region over which a support surface has influence to make it easier to reason about which portion of the texture will show up on which portion of the surface.

1.2 Implicit Surfaces for Interactive Animated Characters

1.2.1 Domain-Specific Challenges and Opportunities

The domain of interactive characters has a set of problems which overlap with much of the current implicit surface research. One of the primary requirements is that the system run in real time, typically 30 frames per second (FPS), and support multiple characters which are always in motion. Another is the support of hierarchical structure. A skeletal implicit surface bears many structural similarities to a standard hierarchical, polygonal object. The latter has many component shapes related to each other by the use of transforms: for example, one kinematic chain contains the torso, upper arm, lower arm, and hand, connected by

the shoulder, elbow, and wrist. Animations can be created by keyframing these transforms in a modeling and animation package such as 3D Studio Max. The component shapes in a polygonal object correspond to skeletal elements in a hierarchically structured implicit surface.

A desirable property of an implicit surface rendering technique is that it support zeroth-order control of joints. In a typical run-time character animation engine, when an animation is being played, each joint's orientation is interpolated from the two nearest keyframes in the output of the animation package. This new orientation is then set immediately in the joint. First-order control, for example, would only allow the specification of angular and translational velocity per joint, while second-order control would allow only specification of torque and translational acceleration. Zeroth-order control allows standard commercial modeling tools to be used for animating characters, and makes easier the incorporation of run-time motion generation techniques such as inverse kinematics.

The structure of an animated character provides certain domain-specific opportunities as well. At least part of a character is always in motion, so slight visual artifacts on the surface may not be as noticeable as in a still frame. Topology changes are infrequent or do not occur at all. Hierarchically structured characters have additional structure over an unconnected collection of skeletal elements which can be used to infer blending properties, improve rendering performance, and generate texture coordinates.

1.2.2 Contributions of IMPS

Many of the rendering algorithms described in Section 1.1.2 either run at interactive rates or were designed with animation in mind. Witkin and Heckbert's sampling algorithm was created for interactive control as well as rendering of implicit surfaces. Stander and Hart's variant runs in real time for chains of interacting blobs. Desbrun and Gascuel's algorithm handles fairly complex scenes at interactive rates. What contributions does IMPS make?

One of the goals of IMPS is to attain real-time, or 30 FPS, rendering rates for reasonably complex characters. The library contains a new, parallelized, Witkin-Heckbert variant which polygonizes simple scenes in real time and more complex ones in interactive time (10-15 FPS.) Instead of using Delaunay triangulation [35] to obtain an optimal triangulation of a set of points, a linear-time *approximate triangulation* is used; this is discussed further in Section 3.7.

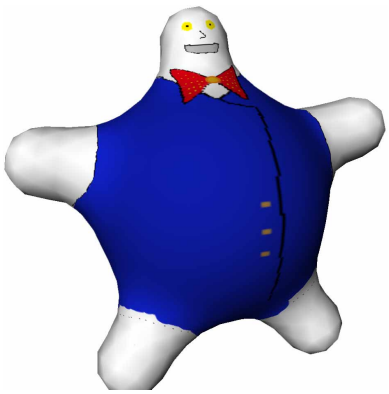

Figure 1-3: A simple texture-mapped character.

Witkin and Heckbert's sampling technique, as specified in their paper, only allows first-order control of surfaces, though it is possible to perform zeroth-order control by trading off the stability of the solution. Modifications to their algorithm which accelerate its convergence for hierarchical implicit skeletons and which provide a new solution to the unwanted blending problem are discussed in Section 3.7.

IMPS focuses not only on the real-time aspects of character animation, but on the entire process of implicit surface-based character design. The system supports interactive modeling and texturing of characters using a 3D painting approach [46]. It contains a novel real-time texture coordinate generation technique for implicit surfaces. The system currently supports static 2D texture maps which reside on animated surfaces, though the generalization to animated texture maps should be straightforward.

In addition to these specific contributions, this thesis discusses lessons learned during the implementation of these algorithms and others which did not work as well, and concludes with an analysis of the suitability of implicit surfaces to the chosen domain.

1.3 Organization of the Thesis

Chapter 2 describes the high-level requirements for IMPS and the effects they had on the system's design. Chapter 3 discusses the rendering algorithms IMPS implements, the experience gained from several experimental implementations, and the extensions made to existing algorithms. Chapter 4 describes how IMPS was integrated with SCOOT, the Synthetic Characters' Object-Oriented Toolkit, allowing polygonal and implicit objects to be combined and animated. Chapter 5 presents the character design and animation pipeline

IMPS prescribes and gives examples of characters created with the system. Chapter 6 discusses the applicability of implicit surfaces to the domain of character construction from a design standpoint, and offers suggestions to future work. Appendix A describes how to obtain source code for IMPS, contains documentation for `glImplViewer` and `ivImplViewer`, shows some example VRML files, lists all of the classes defined by the IMPS library and provides information on their use. Appendix B describes the programming conventions and structures used to develop IMPS.

Chapter 2

System Design

This thesis was initially motivated by the desire to eliminate creases at the joints of characters composed of hierarchically organized polygonal objects, in which the overlaps, and resultant creases, between adjacent objects suggest a continuous skin for the character. Early on it became clear that implicit surfaces were better suited for modeling cartoon characters than more photorealistic ones, since the resulting surfaces are qualitatively smooth and blobby-looking; squash and stretch were therefore desirable properties that appeared to be automatic in the implicit surface representation.

The approach replaces the standard geometric, polygonal primitives, like ellipsoids, with implicit ones, like metaballs, meta-line segments and metaellipsoids, while retaining the hierarchical structure of the character. Joints are animated exactly as with geometrical models. The implicit representation provides automatic blending, or smoothing, at joints, and squash and stretch where adjacent primitives move closer together or further apart.

IMPS was designed to plug into the Synthetic Characters' Object-Oriented Toolkit (SCOOT). SCOOT uses VRML as its input file format for all geometric characters, so it was desirable to maintain this for implicit surface-based characters. IMPS therefore is a VRML-based implicit surface toolkit; a hierarchy of implicit primitives such as metaballs and meta-lines is maintained via the use of VRML-like transforms. The library is built on a framework for defining VRML node classes which was inspired by the design of Silicon Graphics' Open Inventor [16]. Documentation for this VRML library is in Appendix B.

IMPS' geometric primitives include metaballs, meta-line segments with differing end-point radii, and meta-ellipsoids. Each of these primitives is a node type in the library and

can be read from and written to VRML files. For example, the VRML specification of a metaball looks like

```
DEF MY_SPHERE Metaball {
  center 1.0 0.0 0.0
  radius 2.0
}
```

The library includes multiple blending functions: Blinn's original exponential (implemented using a fast approximation to the exponential function [12], Blanc and Schlick's finite-support function, and an elliptical blend from Rockwood's paper [38], which is only suitable for pairwise blends. The specification of these functions in a VRML file is similar to the specification of primitives and is illustrated in the examples in Appendix A.

IMPS is integrated into the graphics system of the SCOOT system, which is a Java-based framework for the development of interactive animated characters. Combining SCOOT with IMPS allows implicit-surface based characters to be animated from a Java application. Since both geometric and implicit surface-based characters are animated by specifying joint angles over time, the Java-side interfaces look exactly the same for these two substantially different geometrical representations. The Java framework also allows rigid and deformable geometry to be combined; polygons can be used for regions of a character which require higher detail, such as its face, while implicit surfaces can be used for its body. This will be described further in Chapter 4.

2.1 Implementation Platform

IMPS was developed on a Silicon Graphics Onyx2 with 8 195 MHz R10000 processors, 1 GB of main memory, and infiniteReality graphics. A goal of this work was to make IMPS graphics system-independent. The system has been ported to Windows NT and has been run on a PC with two 400 MHz Pentium II processors and 256 MB of RAM. Most of the classes comprising IMPS, as well as the underlying VRML library, are platform-independent; specific exceptions are listed in Appendix B.

2.2 Field Function Conventions

In order to be able to specify implicit surfaces in an object-oriented manner, the forms of the primitives' field and blending functions must follow some conventions. Specifically, the

desired isovalue of each primitive, as well as the domain and range of the blending functions applied to the primitives' field functions, must be specified.

Relatively few of the referenced papers are very precise about the forms of the primitives' field functions. Notable exceptions are Blinn's initial paper [1] and Blanc and Schlick's paper on anisotropic field functions [24].

Witkin and Heckbert, for example, [31] suggest the use of the intuitive implicit definition of a sphere centered about the origin:

$$f(x, y, z) = x^2 + y^2 + z^2 - r^2$$

where the desired surface lies at $f(x, y, z) = 0$. Assume a scaled Gaussian distribution is used as the blending function. Using vector notation, where $\mathbf{x}$ is the point at which the function is being evaluated and $\mathbf{c}$ is the center of the metaball, the resulting field function for the metaball is

$$\exp(f(\mathbf{x})) = \exp\left(\frac{\|\mathbf{x} - \mathbf{c}\|^2}{-\sigma^2} - \frac{r^2}{-\sigma^2}\right) \tag{2.1}$$

σ is proportional to the standard deviation of a Gaussian.

The above formulation, characterized by subtraction of the radius parameter, visualization at $f(\mathbf{x}) = 0$, and a global specification of σ, was originally used for IMPS' metaball and meta-line segment implicit primitives. The resulting primitives blended well when their radii were similar, but when a small metaball was brought close to a larger one, the larger would quickly envelop the smaller, rather than allowing the smaller to add detail to the larger surface.

Blinn's formulation [1] provides insight to this problem. Consider representing the field function as a weighted sum of Gaussians with differing variances,

$$\begin{aligned} f(\mathbf{x}) &= \sum_{i=1}^{N} b_i \cdot \exp(-a_i \|\mathbf{x} - \mathbf{c}_i\|^2) \\ &= \sum_{i=1}^{N} \exp(-a_i \|\mathbf{x} - \mathbf{c}_i\|^2 + \ln\ b_i) \end{aligned}$$

As Blinn points out, it is advantageous to reparameterize this equation into roughly orthogonal radius and "blobbiness" parameters for each Gaussian. The a_i coefficient can be solved for in terms of b_i by choosing an isovalue $f(\mathbf{x}) = T$ and plugging in the desired radius R of an individual metaball:

$$T = b_i \cdot \exp(-a_i R_i^2)$$
$$a_i = -\frac{\ln \frac{T}{b_i}}{R_i^2}$$

Blinn now defines a "blobbiness" parameter $B_i = \ln(\frac{T}{b_i})$, so $b_i = \frac{T}{\exp(B_i)}$. The field function now becomes

$$\begin{aligned} f(\mathbf{x}) &= \sum_{i=1}^{N} \frac{T_i}{\exp(B_i)} \cdot \exp\left(\frac{B_i}{R_i^2}\|\mathbf{x} - \mathbf{c}_i\|^2\right) \\ &= \sum_{i=1}^{N} T_i \cdot \exp\left(\frac{B_i}{R_i^2}\|\mathbf{x} - \mathbf{c}_i\|^2 - B_i\right) \end{aligned} \tag{2.2}$$

with T_i, a redundant per-metaball threshold, chosen to be 1. The global isovalue for which this function will be visualized has yet to be chosen, but is also typically chosen to be 1. Note that B_i must be negative to cause the function to decay to zero as the distance from the center point becomes large. Note also that a blobbiness closer to zero actually indicates a more blobby metaball, since the blobbiness term indicates how quickly the exponential falls off; blobbiness might be better termed "hardness", as Blanc and Schlick [24] call it.

Matching coefficients between Equations 2.1 and 2.2, it is clear that Blinn's formulation is functionally equivalent to that originally derived for IMPS from Witkin and Heckbert's suggestions, with the mapping

$$\frac{-1}{\sigma^2} = \frac{B_i}{R_i^2}$$
$$\frac{r^2}{-\sigma^2} = B_i$$

or, rearranging,

$$R_i = \sqrt{-B_i \cdot \sigma^2} = \sqrt{-\sigma^2 \frac{r^2}{-\sigma^2}} = r \tag{2.3}$$

As Blanc and Schlick [24] show, it is straightforward to separate the exponential blending function from the *distance function* of the implicit primitive to allow the blend function to be replaced. Blinn's definition of the field function implies a different metaball distance function than that suggested by Witkin and Heckbert: he uses the function $f(x, y, z) = \frac{x^2+y^2+z^2}{r^2}$ visualized for $f = 1$ rather than $f(x, y, z) = x^2 + y^2 + z^2 - r^2$ for $f = 0$. Note that Witkin and Heckbert's formulation couples Blinn's "blobbiness" parameter and the radius of the primitive. This coupling, combined with the lack of per-primitive σ parameters in IMPS' original exponential blend formulation, caused smaller primitives to more easily blend with neighbors and therefore resulted in the undesirable behavior of smaller primitives being rapidly enveloped by their larger neighbors, rather than maintaining their shape and adding detail to the larger.

For this reason, IMPS' distance functions are formulated in terms of normalized radii, as seen in the sections below. They are all expressed in terms of squared distance; this is an invariant which must be maintained in any new subclasses in order for the blend functions to operate properly.

2.3 Field Functions

2.3.1 Metaballs

IMPS' metaball distance function, as suggested above, is

$$f(\mathbf{x}) = \frac{\|\mathbf{x} - \mathbf{c}\|^2}{r^2}$$

where $\mathbf{x}$ is the three-dimensional point at which the distance function is being evaluated, $\mathbf{c}$ is the center point of the sphere, and r is the radius of the sphere.

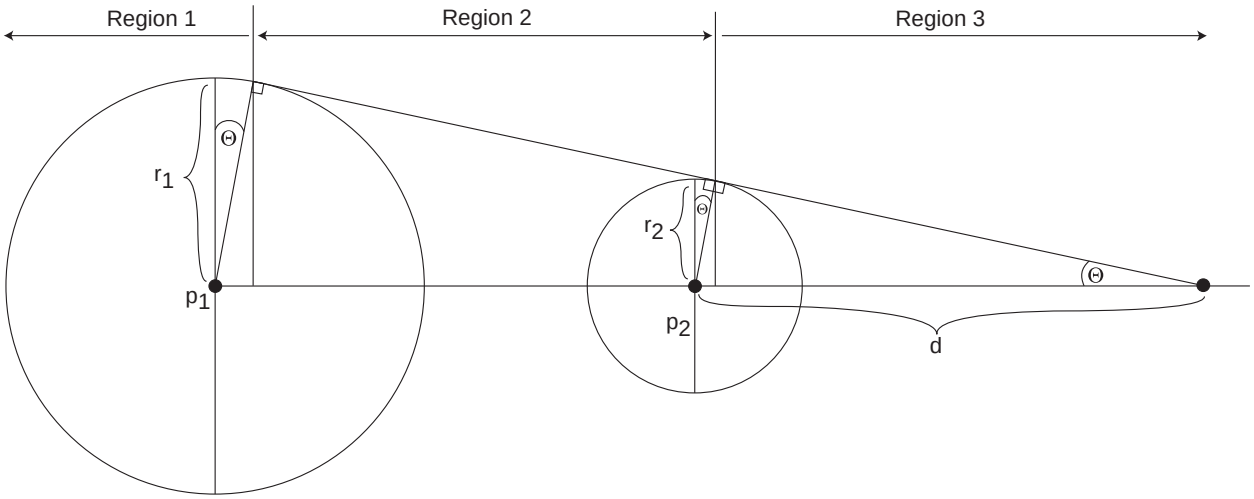

Figure 2-1: Schematic of a meta-line segment.

2.3.2 Ellipsoids

IMPS' ellipsoid is parameterized in terms of the lengths of its three axes, a, b, and c. In the ellipsoid's local coordinate system, the distance function is

$$f(x, y, z) = \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2}$$

Since IMPS' primitives must be able to be expressed in world coordinates (see Section 3.2), the actual implementation additionally contains the three normalized axes and center point of the ellipsoid. The center point is subtracted from the incoming point and dot products taken with the three axes to obtain the x, y, and z values in the above equation.

2.3.3 Line Segments

IMPS contains a meta-line segment primitive allowing specification of two endpoints and radii at these endpoints. Figure 2-1 illustrates the geometry of the primitive. Points $\mathbf{p}_1$ and $\mathbf{p}_2$, as well as radii r_1 and r_2, are given.

Clearly there are three regions of the line segment, corresponding to whether the sample point is closest to endpoint 1 (region 1), the intervening line segment (region 2), or endpoint 2 (region 3). Note that the boundaries between these regions are not coincident with the endpoints; this is important in order to avoid discontinuities in the gradient. In order to determine the boundaries' locations it is necessary to solve for the unknown angle Θ.

Comparing similar triangles,

$$d \sin \Theta = r_2$$
$$(\|\mathbf{p}_2 - \mathbf{p}_1\| + d) \sin \Theta = r_1$$

Subtracting and canceling,

$$\frac{r_1 - r_2}{\|\mathbf{p}_2 - \mathbf{p}_1\|} = \sin \Theta$$

It is now possible to compute the distance between $\mathbf{p}_1$ and the boundary between regions 1 and 2, $r_1 \sin \Theta$, and the distance between $\mathbf{p}_2$ and the boundary between regions 2 and 3, $r_2 \sin \Theta$.

Using the identity $\sin^2 \Theta + \cos^2 \Theta = 1$, $\cos \Theta$ can be found and used to compute the radius of the line segment at the region $1/2$ boundary, $r_1 \cos \Theta$, and the radius at the region $2/3$ boundary, $r_2 \cos \Theta$.

It is now possible to compute the distance function of the line segment for any sample point. In similar fashion to the other primitives, the squared distance to the line segment is divided by the squared radius at this point. For points in regions 1 and 3, the radius and distance computations reduce to the metaball's case. For points in region 2, the orthogonal distance is computed using the Pythagorean theorem, and the radii at the region boundaries are linearly interpolated based on the sample point's relative position to those boundaries.

Chapter 3

Development of the Polygonization Algorithms

3.1 Introduction

Once an implicit surface is specified by a hierarchical combination of implicit primitives, it must be translated into some number of triangles, which are the basic geometric objects renderable with today's graphics hardware. This translation process is known as polygonization, and is performed each time the surface needs to be rendered. When the surface is in motion, polygonization is necessary to render each frame of the animation.

During the development of IMPS, several polygonization algorithms were implemented and tested, the goal being a robust algorithm capable of running in real time (that is, thirty frames per second.) This chapter describes in chronological order the development and implementation of the algorithms and the problems which were encountered and solved.

3.2 Sampling in Local Vs. World Coordinates

All polygonization algorithms for implicit surfaces are necessarily based on computing some sampling of the surface. There is a fundamental difference in how implicit surfaces are sampled compared to other, more widely used, curved surface representations such as B-spline patches or NURBS. B-spline patches are organized into rows and columns of cubic splines, and are controlled by a set of three-dimensional control points organized into a grid. The patch has an *explicit* parameterization; it has two defining parameters u and v (each in

the range [0..1]) over which iteration can be performed to render the surface. The surface's three-dimensional position can be computed for any (u, v) pair by evaluating the adjacent B-splines and performing the appropriate cubic interpolation for the desired point. The fact that the surface has an explicit (u, v) parameterization makes polygonization simple; the surface needs only to be sampled on a regular (u, v) grid and the resulting samples organized into pairs of triangles. Texture mapping such surfaces is relatively straightforward because the (u, v) coordinates can be re-used as texture coordinates.

Implicit surfaces have no explicit parameterization. They are defined by an implicit equation in x, y, and z, and a constraint thereon. The field function is defined everywhere in 3-space, but the constraint is only satisfied for points on the surface. It is not possible to directly sample the surface of an implicit surface; it is only possible to sample the three-dimensional potential field defined by its primitives. For this reason, implicit surface sampling is based on evaluating the field function and its gradient at a three-dimensional point p.

The field function is defined by the sum of the field functions of its primitives (ignoring the "unwanted blending" problem for the moment), and there are therefore two fundamental methods for evaluating the function at a point p. One method is to iterate through the primitives, transforming the world-coordinate point p into the local coordinate system of each primitive, evaluating the function of the primitive in local coordinates, and summing the result with that from other primitives. The second method maintains the primitives in world coordinates, so that evaluating the field function per primitive at p is done merely by passing p into the primitive's `evaluate` method.

Any polygonization algorithm for implicit surfaces will necessarily evaluate the field function many times per frame. It is therefore obvious that if the primitives can be maintained in world coordinates, many world-to-local matrix transformations can be saved during polygonization. Since IMPS was intended to be a real-time implicit surface animation system, an early design decision was to allow primitives to be flattened into world coordinates for more efficient sampling. The disadvantage to this approach is added complexity per primitive; each must be able to transform itself into world coordinates. In the case of a metaball (sphere), for example, it is not sufficient merely to specify the radius; both its radius and center point must be specifiable to allow the value of the current transform to be flattened into the primitive.

The above formulation for a metaball does not support non-uniform scales; in order to do so it would need to have many more parameters, including the principal and scale axes, to handle the VRML Transform node's "scale" and "scaleOrientation" fields [7]. IMPS' ellipsoid class comes closer to handling non-uniform scales properly, but is more computationally expensive to evaluate. Since most animation of standard, polygonal characters is constrained to modification of joint angles, IMPS defines its own "rigid transformation" class, allowing rotation and translation, but not scale, to be modified. Making the simplifying assumption that per-primitive scales are uniform and not time-varying simplifies the implementation of primitives when they are to be sampled in world coordinates, but has the disadvantage of not being able to model per-primitive deformations like squashing. The ramifications of this design decision are discussed further in Chapter 6.

3.3 Marching Cubes

One of the earliest methods for rendering isosurfaces, Marching Cubes is still the standard technique for rendering polygonal approximations in most molecular visualization packages. During the initial development of IMPS, Marching Cubes was implemented as a baseline rendering algorithm. The implementation from Watt and Watt [29] was used as a starting point. There was still a fair amount of work to make this implementation complete; specifically, the data tables for vertex and edge information were left to the reader, so a program was written to enumerate the inside/outside possibilities for each vertex in the cube as well as the orientations of the cube. The result of this program was a set of data tables compatible with Watt and Watt's Marching Cubes implementation, which is one of the most elegant and concise versions available.

The advantage of Marching Cubes is its robustness, which is why it is the standard isosurface visualization algorithm. Its primary disadvantage is the fact that the entire polygonization solution must be recomputed from scratch each frame; the solution is not incremental. This is most easily recognized by the fact that the voxels associated with the solution are aligned with world-coordinate axes. If a character waves its arm, it should be possible to identify that the limbs have not deformed much, and merely transform the "skin" vertices into their new positions, saving many evaluations of the field function. The fact that the mesh generated by Marching Cubes is aligned to world-coordinate axes rather than

the local coordinate systems of the primitives is a good indication that such incremental solutions are not efficiently applicable; in this example, the body's mesh would be aligned with the world-coordinate axes while the arm's would be aligned with the shoulder's local coordinate system, and some sort of resampling and stitching would be necessary near the shoulder. Opalach and Maddock [34] attempted to speed up the MC algorithm by keeping track of which voxels were "dirty" and resampling only at those voxels' corners each frame. Unfortunately their data sets did not have any hierarchical structure, but were merely a random set of metaballs. It is unlikely that such methods would provide any performance increase in the case of hierarchically structured implicit surfaces.

Initial tests with the MC algorithm indicated that with a moderately complex animated three-link arm, a parallel MC implementation (providing a seven-times speedup on the available hardware in the ideal case) would still not attain a real-time frame rate, so the Marching Cubes algorithm was abandoned for further development.

3.4 Witkin-Heckbert Particle Simulation

Witkin and Heckbert [31] developed a physically-based particle simulation on implicit surfaces. In their formulation, particles repel each other, grow when they are in unpopulated regions, and stochastically fission and collapse when they grow too large or too small. The dynamics of their simulation were designed to stabilize around a hexagonal packing of particles, and in fact diagrams in [31] show that this is indeed the preferred configuration. As it is obvious how a hexagonal packing should be triangulated, this algorithm was chosen as the basis for further polygonization work in IMPS.

The algorithm described in [31] was implemented. The first problem encountered was a sign error in the equation for the desired velocity; the version of the paper available from `http://www.cs.cmu.edu/~ph` contains this fix. The next problem encountered was that particles appeared to be splitting incorrectly; when sampling a sphere, for example, the first particle would fission, but the second would travel to the other side of the sphere before fissioning, and the resulting particles failed to spread to evenly cover the surface of the sphere. The problem was eventually traced down to the fact that particles' radii were shrinking to zero (and in fact becoming negative), but it was unclear which portion of the dynamical simulation was causing this condition to occur. A web search turned up

a publically available Witkin-Heckbert simulator written by Hans Pedersen, available from `http://implicit.eecs.wsu.edu/course14/code/imp.tar`. This implementation used a simplification of the radius update equation, later confirmed by Pedersen to be a fix for precisely this problem.

Witkin and Heckbert derive the update to each particle's radius as follows; see [31] for the full details and information on the notation. Each particle's energy is defined to have two components; that which it induces on its neighbors, and that which its neighbors induce on it. The latter component is related to this particle's radius, and is defined as

$$D^i = \sum_{j=1}^{n} E^{ij}$$

The hexagonal packing constraint implies a global desired energy level $\hat{E}$. To keep D^i near this value linear feedback is used:

$$\dot{D}^i = -\rho(D^i - \hat{E})$$

ρ is the feedback constant. Using the chain rule, the rate of change of the particle's radius which will cause this amount of feedback is

$$\dot{\sigma}^i = \frac{\dot{D}^i}{\beta + \frac{1}{(\sigma^i)^3} D^i_{\sigma^i}}$$

where β is a constant to avoid dividing by zero when a particle has no neighbors, $\mathbf{r}^{ij}$ is the vector from particle i to j, and where the partial derivative of the induced energy on the particle with respect to its radius is

$$D^i_{\sigma^i} = \sum_{j=1}^{n} \left|\mathbf{r}^{ij}\right|^2 E^{ij}$$

Experimentally, this radius update rule caused particles' radii to become very small and occasionally even negative. The update rule in Pedersen's implementation, which was adopted for use in IMPS, is equivalent to replacing $D^i_{\sigma^i}$ with D^i, removing the scaling by the square of the distance between the particles:

$$\dot{\sigma^i} = \frac{\dot{D^i}}{\beta + \frac{1}{(\sigma^i)^3} D^i}$$

Implementing this modification to the particle radius update equation generated the expected results, similar to the behavior described in [31].

Tests with the particle simulator were encouraging. It attained a significantly higher frame rate than the Marching Cubes algorithm. It appeared to be quite robust, successfully sampling several collections of primitives, although surface motion caused particles to drift and occasionally fly off the surface at high velocities; this problem is discussed further in Section 3.7. Because the algorithm is incremental (using the previous frame's solution as a basis for the next frame's) it appeared that it could be modified to take advantage of knowledge of the hierarchical structure of a character's geometry. Modifications made are discussed in Section 3.7.

3.5 Experimental Polygonizer 1: Witkin-Heckbert with Springs

The first experimental polygonization algorithm implemented for IMPS (`ImTesselator`) was a modification of the Witkin-Heckbert simulation designed to add triangle generation by maintaining a set of springs and dampers between vertices; these springs defined the edges of triangles. Vertex fissioning added springs between the new vertex and neighbors of the parent. The springs' rest length was determined by the desired inter-particle distance from the Witkin-Heckbert simulation. A map of the vertices on the edge of the currently polygonized patch, the *edge map*, was maintained and used to determine whether a vertex was in the interior of the patch, the goal being to avoid creating overlapping triangles when determining a new vertex's neighbors. The combination of the Witkin-Heckbert simulation with springs is similar to that described by Rodrian [36].

There were several problems with this polygonizer. Geometrical arguments used to determine neighbors for a new vertex during fission assumed relatively flat local surface patches, a condition which did not hold for more complex and high-curvature shapes; this caused incorrect polygonization of certain shapes. Although the edge map of the currently polygonized surface patch was maintained, there was still no notion of when that patch became self-intersecting or when triangles folded back over themselves, and in real-world tests

this happened more often than not. Finally, the dynamics of the Witkin-Heckbert simulation were designed to be self-contained; adding spring dynamics to the equations caused the overall performance of the system (ignoring the previous two problems) to decrease significantly compared to the unmodified particle simulation. More discussion of these problems and others is at the end of the next section.

3.6 Experimental Polygonizer 2: Surface Walker with Springs

The second experimental polygonization algorithm implemented for IMPS (`ImTesselator2`) was designed to lay triangular tiles on the surface in an expanding patch. One point was used to initialize the system; when it reached the surface (determined by the magnitude of the gradient falling beneath a threshold) its state was marked as inactive and its position in space fixed. A second point was then created, attached to the first by a rigid link of user-defined length, and allowed to drift to the surface. A third point was then attached rigidly to the other two and allowed to drift toward the surface; when it reached the surface, the first triangle was created. Successive triangles were added to the system by choosing an edge on the edge map which had two inactive (i.e., immobile) vertices, attaching a new vertex to these two by rigid links, and allowing the resulting triangle to drift outwards towards the surface. Geometrical arguments were used to determine when a portion of the edge map was significantly concave, in which case a triangle would be created between three vertices on the edge map rather than through "fissioning" of an existing edge. Fissioning of an edge was prohibited when the edge was too close to another; this was intended to allow the algorithm to halt. The intended result of the expansion of the polygonized patch was a single triangulated patch with a thin gap which could be "zippered" to close the surface. Once the surface was closed, per-vertex gradient following and spring and damper dynamics between adjacent vertices were used to track surface motion. A similar surface marching method was developed independently by Hartmann [42].

As with the first experimental algorithm, several problems were encountered. This algorithm was primarily intended to counter poor performance of the first one during the expansion of the mesh, such as the surface folding in on itself; for this reason vertices' positions in space were fixed once they reached the surface, and new vertices added only between edges whose endpoints had been fixed. The first modification necessary was the

concavity test to connect vertices on the edge map that were suitably close to one another; the result of this modification was large regions of poorly shaped (long and thin) triangles. Further, since the algorithm was deterministic once the initial point had been placed, such large regions were pervasive and inevitable. Witkin and Heckbert's simulation used stochastic fissioning and merging of particles to obtain a visually more even sampling.

Despite careful bookkeeping of vertices on the edge map and tests to ensure that triangles were not created which would cross the edge map (thereby creating a self-overlapping surface), in some cases the algorithm managed to wrap a surface twice. In other cases the algorithm failed because the randomly positioned initial point had drifted to a poorly-conditioned portion of the surface.

Once the surface's triangulation had been closed and the dynamics simulation had begun, it became obvious that spring dynamics were not sufficient to track surface motion. A long period of time was required for local changes in one portion of the mesh to propagate outward and stabilize. In contrast, the dynamics of the Witkin-Heckbert simulation manage to rapidly track surface motion and deal well with topology changes of the implicit surface, which spring-based simulations [36] can not handle without special cases.

The overall lessons from these experiments were twofold. First, geometrical arguments are not sufficient during the development of an implicit surface polygonization algorithm. If the algorithm requires rules as to when to connect adjacent vertices on the edge of a surface patch, for example, then if these rules are derived with an assumed local surface geometry (i.e., flat) then they will surely fail in the case of more complex and high-curvature surfaces. Second, spring dynamics are unattractive for the purpose of maintaining an implicit surface polygonization. High damping constants are required to prevent oscillations in the surface, which lead to instability and the necessity of complex integration algorithms. In addition, local effects require a great deal of time to propagate throughout a spring-simulated mesh. Some of these problems could be solved using a better integration technique like implicit integration; see, for example, Desbrun, Schröder, and Barr [6]. Overall, combining springs with Witkin-Heckbert dynamics was found to be a poor match; these results and those in the next section can be compared to those described in [36], which describes a similar combination.

3.7 Experimental Polygonizer 3: Witkin-Heckbert with Approximate Triangulation

The third polygonizer implemented for IMPS (`ImTesselator3`) again used the Witkin-Heckbert particle system as its basis. A spatial partitioning scheme described by Heckbert [39] was incorporated to decrease the complexity of the algorithm from $O(N^2)$ to approach $O(N)$. The observation made by Heckbert is that most of the time in the algorithm is consumed by the per-particle energy computation, which requires contributions from each particle's nearest neighbors to be summed. His spatial partitioning scheme hashes three-dimensional space into a set of buckets; many volumes in space map to the same bucket, so it is advantageous to tune the size and number of the buckets to surround the model to be visualized as closely as possible.

Once this spatial partitioning scheme was in place, each particle contained a list of its nearest neighbors. It was therefore feasible to implement the intuitive algorithm mentioned in Section 3.4 of connecting local hexagonal patches into sets of triangles. The specific algorithm is as follows:

1. Define n as the number of nearest neighbors within a user-defined radius of the current particle. This is efficiently computable since the result of the spatial subdivision query is a list of some number of the particle's nearest neighbors sorted by increasing distance.

2. Generate an arbitrary set of two unit vectors perpendicular to the gradient of the field function at the current particle's position and to each other. These form a temporary two-dimensional basis; call these vectors b_0 and b_1. These are ordered so the cross product of b_0 and b_1 is the normal to the surface at the particle's position.

3. For the first n nearest neighbors, find the vector from the current particle to the neighbor, project it onto b_0 and b_1 (call these dot products x and y), and compute the arctangent of y/x (that is, `atan2(y, x)`.) Call the resulting angle θ. Sort the first n nearest neighbors by increasing θ. This computes a counterclockwise ordering of the nearest n neighbors to the current particle.

4. Iterate through the n sorted neighbors, attempting to add triangles between the current particle, the ith neighbor, and the $(i+1)\%n$th neighbor. Each particle keeps

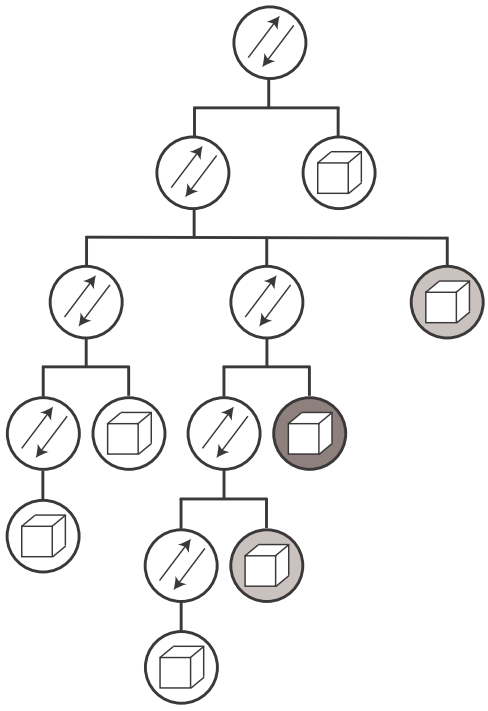

Figure 3-1: Blending rules IMPS imposes upon a hierarchy of Transforms and Shapes. Points belonging to the dark gray Shape node add in contributions from the light gray nodes, which correspond to its parents and children.

track of which triangles it is a member, so duplicate triangles are avoided.

The standard triangulation technique for unorganized point sets, Delaunay triangulation, is *optimal* according to some set of criteria. Su and Drysdale [35] compare several algorithms for computing the Delaunay triangulation of a 2D point set. The algorithm described here is *approximate*; it does not guarantee that no poorly formed triangles will be added to the triangulation, nor does it guarantee that no overlapping triangles will be created nor that no holes will appear in the triangulation. The advantage of this algorithm is its speed. When combined with the Witkin-Heckbert sampling method (with which it was designed to work), this algorithm requires only three to four milliseconds of work for a mesh of a few hundred triangles. The bottleneck in the overall algorithm is still the evaluation of the particles' energies; see Table 3.1, below.

This triangulation algorithm is similar to that described by Crossno and Angel [41], although IMPS' makes no attempts to control the aspect ratio of its generated triangles nor the appearance of overlapping triangles in the interest of speed.

With a Witkin-Heckbert based polygonization algorithm in place, modifications were

made to take advantage of the hierarchical structure of animated characters. First and most necessary was a solution to the unwanted blending problem. The rule chosen for IMPS was that a particular implicit primitive would blend with its siblings, parents and children in the hierarchy. Grandparents would not, however, blend with grandchildren. Figure 3-1 illustrates these rules. Each particle keeps track of the primitive which currently owns it, defined as the primitive whose field function contribution is closest to the value 1.0. From frame to frame, particle ownerships can only be transferred to blend neighbors; therefore a particle may move from one primitive to its sibling or parent, but not to its grandparent. Frame-to-frame coherence of the surface prevents unwanted blending. However, when particle ownership changes, the patch of which it is a member changes, and since it was previously constrained to the surface of the old patch but not the new, it can be far enough away from the surface that it "blasts off", an instability apparently inherent in the Witkin-Heckbert simulation. A significant problem in the construction of bipedal structures with IMPS was "fighting" over particle ownership at non-blending regions (for example, between a torso and an upper arm in the region of the shoulder), a consequence of repeated particle blast-offs, deaths, and fissions. These stability issues are discussed in Chapter 6.

Another significant modification to the algorithm was made to take advantage of kinematic motion of characters. For example, if a character bends its arm at the elbow, particles sampling the surface farther down the arm, for example at the hand, should be automatically transformed according to the elbow's rotation to keep up with the surface motion.

To implement this functionality two elements are necessary: first, the differential transformation from frame to frame must be computed for each joint and applied to all points belonging to primitives below this joint in the hierarchy. Second, the implicit primitives below this joint must be transformed to their correct positions for the current frame.

A typical run-time animation system contains information about how to set all of the character's joints at each frame to produce a desired motion. That is, the joint angles are specified absolutely each frame, not in terms of a differential from the previous frame. Requiring differentials or joint angle velocities to be specified by the animation system causes many problems including the potential for roundoff error and more difficult control. For this reason IMPS was designed to allow joints' orientations to be specified completely each frame; this is a *zeroth-order* control mechanism. All of the differential information

required for incremental polygonization is computed internally by the renderer.

The fundamental data structure in IMPS which assists in the computation of differential transforms is the *skeleton cache*. As its name implies, it represents a cache of the current pose of the character's geometry, or skeleton. The skeleton cache is initialized from the VRML file specifying the character's geometry when it is first loaded. At this time copies of the character's implicit primitives are made and stored internally to the skeleton cache. At run time, the application specifies the transforms as desired; the skeleton cache then traverses the scene graph, applies these transforms to the internal copies of the primitives, thereby transforming them into world coordinates, and computes the differential transform to be applied for the current frame to particles owned by a particular implicit primitive. The skeleton cache also assists in the computation of the blending function by sorting primitives into groups with which they blend, but this is conceptually less important than its other functionality.

The skeleton cache allows the previous frame's polygonization to be used as a better starting point to the current frame's by taking account, in an approximate fashion, of the rigid motion of the character. Witkin and Heckbert's original paper attempts to solve this problem in a different manner; they allow only specification of velocities of the component primitives in a compound shape, which is a *first-order* control mechanism. An additional term, shown in Equation 3.1, is added to each particle's velocity update equation, which contains the partial derivative of the particle's position with respect to the surface parameters. It seems the surface motion is therefore taken into account when the particle's position is updated each frame; in addition to the repulsion forces and surface gradient which affect the particle's position, changes to the surface itself cause the particle to move.

$$\dot{\mathbf{p}}^i = -\frac{F_{\mathbf{q}} \cdot \dot{\mathbf{q}}}{F^i_{\mathbf{x}} \cdot F^i_{\mathbf{x}}} F^i_{\mathbf{x}} + \dots \tag{3.1}$$

Unfortunately this formulation is not quite correct. First, it does not allow the specification of rotations which affect the implicit primitives. Second, even in the case of pure translation there is an error. Consider a sphere sampled by particles which translates to the right, as in Figure 3-2. The partial derivative of the field function with respect to the surface parameters ($F_{\mathbf{q}}$) points in the rightward direction, as expected. However, from examination of the particle's velocity computation (Equation 3.1), the contribution from this term is only

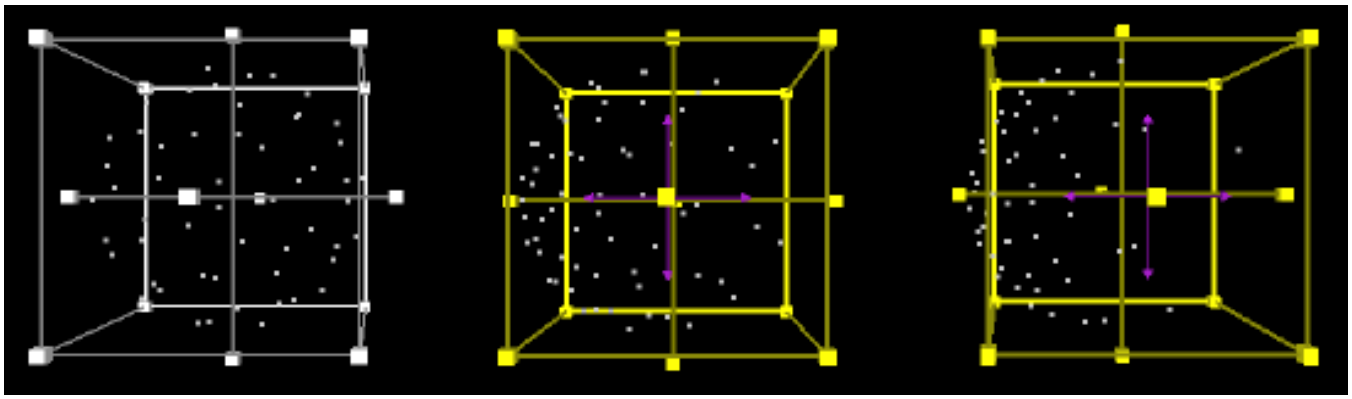

Figure 3-2: Illustration of "sliding" of points as surface translates to the right.

in the direction of the gradient of the field function at the particle's position ($F^i_{\mathbf{x}}$); since the gradient points outward from the center of the sphere, this will cause particles to drift towards the back of the sphere even if Witkin and Heckbert's surface motion compensation term is added to the solution. For these reasons this term was not implemented in IMPS' version of their particle system.

The differential transform modification to the Witkin-Heckbert particle system was independently discovered by Rodrian [36] and is also discussed in the context of texturing implicit surfaces by Zonenschein et al. [51]. However, rather than computing hard "ownership" of a particle by an implicit primitive, both papers discuss taking a weighted average of the differential transforms of surrounding primitives, weighted by the field function (approximately a Euclidean distance metric.) It is difficult to see precisely what this weighted average does to the transform. For inspiration we can look to Shoemake's paper on polar decomposition of transforms [13]. This paper shows the skew that occurs when linear interpolation is performed between two transform matrices; the intuitively proper interpolation between these matrices reqires a factorization of the matrix and spherical linear interpolation of the rotation component. For more than two transform matrices the interpolation or weighted average problem is even more difficult. It is clear, however, that a simple weighted average of transformation matrices is unlikely to produce an intuitively recognizable or formally analyzable result.

IMPS' differential transform modification, on the other hand, essentially computes a Voronoi diagram of the implicit surface, dividing it into multiple *implicit patches*. Particles are assigned to patches via the ownership mechanism and kinematic motion is applied

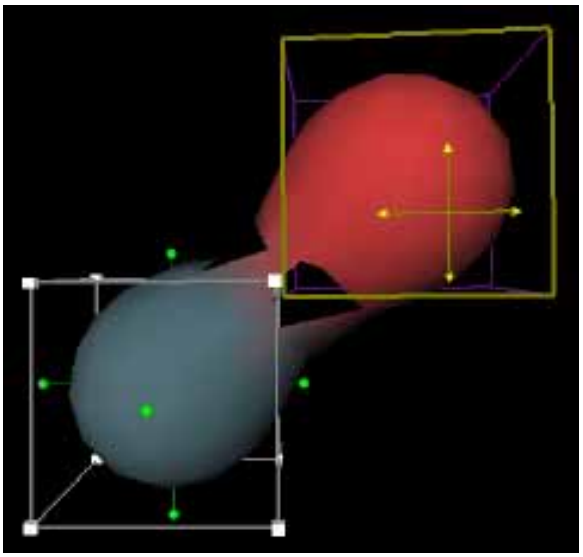

Figure 3-3: Using the differential transform formulation, the surface can tear if the components separate at too high a velocity.

independently to each implicit patch before running the particle simulation. Intuitively it seems that at patch boundaries, for example at the joints of a character, gaps will be most prominent after the differential transforms are applied. If the angular or translational velocity is too great at the joint, gaps can appear if the particle system is unable to sufficiently resample the area, as shown in Figure 3-3. These gaps are a consequence of IMPS' triangulation algorithm being both localized and approximate; it leaves holes in the surface when particles do not sufficiently sample it. On the other hand, the differential transform modification is in most cases a good approximation to the surface motion and greatly accelerates the convergence of the Witkin-Heckbert simulation when the surface is in motion, especially when rotational motion is applied to the root of a chain of implicit primitives, as is shown in Section 5.3.

3.8 Parallelism

As the initial target platform for IMPS was a multiprocessor Silicon Graphics Onyx2, parallelizing the polygonization algorithm was a high priority. As soon as `ImTesselator3` proved to be reasonably robust, a multiprocessor version was written (`ImTesselator3MP`) and used as the basis for further development.

The polygonization algorithm can be broken down into a series of steps, many of which

	TransformedBlobs[1]	TexturedBlobs[2]	Flubber8[3]	SimpleBip01[4]
Particle Sim	8.86	5.17	45.32	50.64
Add/Del	2.8	1.36	11.78	14.41
Mesh Update	3.0	2.65	25.46	17.29
Total Update	15.63	9.71	86.27	88.06
Graphics Render	17.96	7.21	14.0	12.19
Total Time	33.61	16.93	100.27	100.26
Frame Rate (Hz)	29.75	59.03	9.97	9.97
Num Particles	184	98	820	907
Num Tris	390	200	1931	2357

[1] 3 metaballs

[2] 2 metaballs

[3] 1 metaball, 5 line segments

[4] 18 line segments

Table 3.1: Timing results for `Tesselator3MP`. Particle simulation is parallelized (7 CPUs) repulsion force computation; add/del is sequential addition and deletion of particles from simulation; mesh update is sequential regeneration of triangle mesh. All times are in milliseconds per frame.

	TransformedBlobs	TexturedBlobs	Flubber8	SimpleBip01
Mesh Update	88.96	22.65	500.48	2642.91
Graphics Render	27.01	23.55	33.76	25.03
Total Time	115.97	46.2	534.24	2667.95
Frame Rate (Hz)	8.62	21.65	1.87	0.37
Num Vertices	412	214	974	870
Num Triangles	820	408	1942	1740
X Resolution	12	10	20	20
Y Resolution	12	10	20	40
Z Resolution	12	10	20	20

Table 3.2: Timing results for IMPS' non-parallelized `MarchingCubes` implementation. Resolutions were chosen to match the visual fidelity of `Tesselator3MP` as closely as possible. Note that the marching cubes renderer does not render textures. All times are in milliseconds per frame.

are made explicit in Witkin and Heckbert's original paper [31]. First, each particle's cache of its nearest neighbors is updated and used for further calculations. Second, the particles' velocities are computed; these are used later in the integration step when the positions are updated. Third, the rate of change of the particles' radii are updated. Fourth, the particles' positions and radii are updated by adding on their respective velocities times the time step. Fifth, particles undergo the fission/fusion process described in [31] to split too-large particles and remove too-small ones. Finally, the triangle mesh is updated; per-vertex colors and texture coordinates (see Section 3.9) are computed, and the triangle list for the current frame is recomputed from scratch. Note that since each particle already knows its nearest neighbors, this is a linear-time operation and is not the bottleneck in the algorithm; this has been verified experimentally (Table 3.1.)

The algorithm as expressed can clearly be broken down into two phases; the compute phase, during which particles' velocities and radius derivatives are computed, and the update phase, during which these deltas are added on to the current particle position and radius. The update phase is necessarily a sequential operation since the spatial subdivision data structure is not thread-safe when moving particles from one bin to another. However, the compute phase, including the update of the particles' nearest neighbor caches (which is a query operation on the spatial subdivision), can be parallelized.

At the beginning of time the `ImTesselator3MP` renderer starts up a user-specified number of threads (defaulting to a single thread) and, on Irix, attempts to exclusively schedule one thread per CPU using Irix-specific system calls. A work queue consisting of an array of particles is allocated for each thread. At run time, the "master" thread distributes particles to threads' work queues during the sequential or "synchronous" phase by adding each newly fissioned particle to the thread with the least number of particles; this maintains an even distribution of work.

The system starts in the sequential phase. The master thread tells all worker threads to run. In parallel, these threads query the spatial subdivision, update their particles' nearest neighbor caches, and compute velocities, normals, and radius deltas. Note that the computation of all of these quantities is independent for each particle, since each relies only on others' positions and radii, which are only changed during the sequential phase. Preliminary information for the generation of the triangle mesh is also computed in parallel. Specifically, each particle's nearest neighbors are sorted in counterclockwise order according

to an arbitrary set of basis vectors as discussed in Section 3.7. Once the worker threads have finished, the master thread enters the sequential phase. Particle additions and deletions required by worker threads are performed; it is important for the correctness of the algorithm that these be deferred until the sequential phase. Particles' positions, along with the spatial subdivision data structure, are updated. Particle radii, per-vertex colors, and texture coordinates are updated; the latter two should be parallelized but currently are not. Finally, the triangle list for the current frame is generated, and the polygonizer's update routine returns to the caller in preparation for the rendering of the triangles, which is a graphics-library specific operation.

An optimization mostly orthogonal to the issues described above was to allow multiple updates in between actual renders of the mesh, noting that rendering the mesh may take many milliseconds while the particle simulation may be substantially faster, especially for small numbers of particles. The system runs the parallel update phase, and the portion of the sequential phase which updates the particles' positions and radii, several times. Only on the last iteration is the triangle mesh generated and rendered. For simple scenes, three updates per render increases the performance of the polygonizer to real-time rates. For more complex scenes, the polygonizer is compute-bound by the parallel update phase, but performing multiple updates per render increases the perceived responsiveness of the system.

Tables 3.1 and 3.2 compare the `Tesselator3MP` algorithm to the standard Marching Cubes algorithm. For complex models such as Flubber and the biped the algorithm is significantly faster than Marching Cubes; the differences go beyond the factor of seven afforded by parallelism in the optimal case.

3.9 Decoration: Color, Texture Mapping and 3D Painting

A color may be specified for each implicit primitive in a compound surface in IMPS. Since each primitive has an list of blend neighbors inferred from the shape's hierarchical structure, its color may be blended with its blend neighbors' if desired. IMPS' current experimental renderer allows the width of the color blend regions to be specified, but only on a global basis.

IMPS supports per-primitive texture mapping via projective texturing. Each implicit primitive may be conceptually surrounded by a texture coordinate generator which has a

certain well-defined geometrical shape (currently spheres and planes are supported) and associated texture coordinate mapping. Typically the center of the generator's coordinate system is aligned with the center of the primitive. Points owned by this primitive are assigned the texture coordinates of the closest point on the generator. As the primitive is animated by the transforms above it in the hierarchy, the generator follows by applying the same transform from the skeleton cache. The intended effect was to make textures follow the primitives to which they were assigned.

Texture coordinate generation is only one portion of the problem of texturing an implicit surface. Painting a 2D texture in a program like Photoshop and then hoping to apply it to a shape requires skill at best and luck at worst; it is unlikely the generated texture coordinates will match the contours of the shape to the appropriate regions of the texture.

For this reason a 3D painting subsystem was added to IMPS. Once the surface has a complete texture coordinate parameterization, it is a relatively simple operation to cast a ray from the eye point into the model under the mouse pointer, determine the intersected triangle, associated implicit primitive and texture, look up the appropriate texel or set of texels in the texture, and modify or modulate them by an antialiased brush.

IMPS' 3D painting system allows textures to be drawn directly on the surface. This simplifies the construction of a set of texture maps because the underlying texture coordinate parameterization is not as visible to the artist. Rather than attempting to align a set of pre-drawn texture maps by modifying the local transforms of the texture coordinate generators, these texture maps are created interactively. The set of textures generated for a particular character with IMPS' 3D painting system can be loaded into Photoshop for retouching with more powerful painting tools; therefore the 3D paint system can be used either to draw the characters' textures in their entirety or to sketch a rough draft, using a commercial painting program to finish the job.

There are two significant problems with IMPS' approach. First, new texture coordinates are exposed at blend regions when the surface is in motion. An example is shown in Figure 3-4. The problem occurs because the support surface follows the primitive, but more of the surface is visible when the primitives separate, increasing the total amount of surface area each texture covers. The recommended solution is to design the texture so such borders are painted with solid colors, so that stretching of the surface does not show up as new areas of pattern appearing along seams. A part of IMPS' prescribed design process for creating

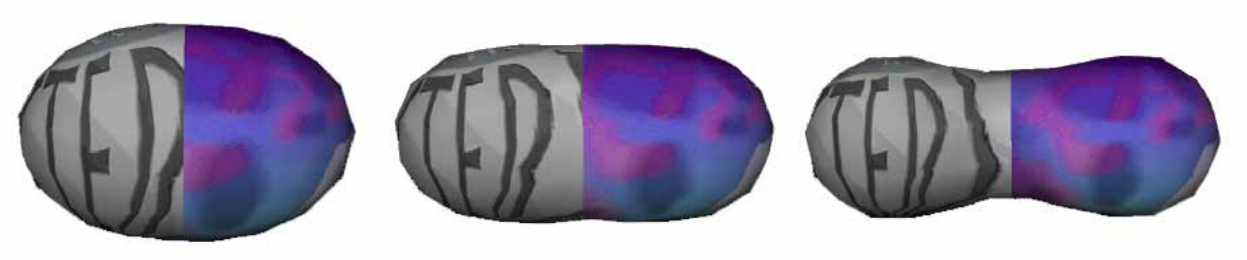

Figure 3-4: Two metaballs with very different textures illustrating texture seams and exposure of new texture coordinates as they separate.

characters is to paint the initial texture, animate the character, and watch for previously unseen portions of the surface appearing as untextured regions, cleaning up such regions in the interactive modeler. This process is discussed further in Chapter 5.

The second problem is that in concave regions of the surface, which typically occur in blend regions, it is possible that the straight-line projection used from the texture coordinate generator to the implicit primitive will actually intersect the surface twice. This problem exhibits itself as two regions of the surface sharing the same texture coordinates. When the surface is interactively painted, the brush strokes may occasionally intersect such a problem region, causing the color to be applied to an entirely different region of the surface than what was intended. This makes interactive design of textures harder. Such problem regions typically correspond to only a small portion of the surface area or do not show up at all, but in a commercial product such unintuitive response, however rare, would not be acceptable.

3.10 Comparison of IMPS' Texturing Algorithm to Existing Approaches

IMPS' approach to texture mapping implicit surfaces appears to be new, but bears many similarities to existing approaches. Discussion of the method's effectiveness and more detailed comparison to others is important to understand the context in which it comes about, as well as the contributions it makes.

3.10.1 Zonenschein et al.

Zonenschein et al.'s approach [51] is most similar to IMPS'. They surround each implicit primitive or group of primitives with a support surface, a concept analogous to IMPS'

texture coordinate generator; both the primitive and the support surface are affected by the same transform. The implicit surface is polygonized into a set of triangles. At each vertex, the gradient of the implicit function is followed until the support surface is intersected; this intersection point determines the texture coordinates for the vertex from the support surface's well-known mapping.

Areas where differently textured implicit primitives blend together are handled differently by the two algorithms. IMPS assigns hard ownership of regions of the surface to individual primitives and consequently to their texture coordinate generators. At regions where two or more primitives' influences overlap, the surface is essentially cut into Voronoi cells. There are hard boundaries ("seams") between adjacent texture coordinate domains. This would be much more of a limitation without the presence of the 3D painting system, which allows the implicit surface to be thought of, in a rough sense, as a surface which can be spraypainted without regard to the underlying texture coordinate parameterization.

The approach recommended by Zonenschein et al. is to blend the transforms positioning the implicit primitives to position a single support surface in the blend region. This is done by computing the contribution which each primitive makes to the overall surface at a given point on the implicit surface and normalizing these contributions to compute an alpha value per surface. A weighted average of the per-primitive transforms is computed. The support surface is transformed by this average and the gradient followed from the implicit surface to the transformed support surface to compute texture coordinates for the point. The intended effect is to slide the support surface across blend regions to achieve texture consistency over the entire implicit surface.

This approach shares many of the same problems that IMPS' faces. Creation of new texture coordinates at blending boundaries and duplication of texture coordinates are common to the two approaches; Figure 11 in [51] illustrates the problem, as the logo splits into two rather than stretching, even though the surface is continuous. The interpolation of arbitrary transform matrices is not guaranteed to keep the implicit surface surrounded by the support surface. Finally, it is not clear how well this approach will handle bifurcations of the surface.

3.10.2 Pedersen

As mentioned in Section 1.1.3, Pedersen's approach to texturing implicit and general curved surfaces [47, 48] is arguably the most intuitive of the available algorithms. The artist first divides the surface into bicubic spline patches which gives the surface a consistent underlying texture coordinate mapping with well-aligned boundaries between patches. This parameterization is then used to place patches of texture ("patchinos") directly on the surface. Higher-level algorithms like flood filling surface regions are also implementable using this framework. The method described in Section 1.1.3 for applying Pedersen's algorithm to moving surfaces would likely provide the most intuitive response of the surface to motion, since the spline patches would stretch in response to motion of the pinned corners, making the object appear as though it had a coherent skin surrounding it.

IMPS' approach is similar to Pedersen's in that it divides the surface into several disparate texture coordinate regions merely to achieve a complete parameterization for the surface, and relies on higher-level algorithms, like 3D painting, for actually decorating it. Compared to Pedersen's approach, however, IMPS' has two primary deficiencies: boundaries between texture coordinate regions are not well-aligned, and motion of the surface causes new texture coordinates to be exposed rather than existing coordinates to be stretched to fit the new surface. In practice, the former limitation does not cause significant visual artifacts; the latter is a fundamental limitation of the implicit surface representation and the fact that there is no inherent parameterization of the surface. Pedersen's approach shows that combining representations like spline patches and implicit surfaces can overcome problems inherent in the implicit surface representation. Unfortunately this particular combination is not currently implementable in real time.

Chapter 4

Integration with SCOOT

An early design goal for IMPS was to achieve integration with SCOOT, the Synthetic Characters' Object-Oriented Toolkit, a Java-based software library enabling the creation of interactive animated characters.

SCOOT, compared to earlier implementations of character toolkits [17], was designed to more easily allow animations created by artists to be loaded into the system, in contrast to requiring all animations to be specified procedurally. Early in the development of SCOOT a design process for characters' geometry was formalized; characters were to be modeled and animated in 3D Studio Max. The character's geometry was exported as a single VRML file, called the *base geometry*. Each animation was exported as a separate VRML file containing, for example, orientation and position interpolators for each joint of the model. These animations became the *motor skills* of the character, the atomic actions which the character knew how to perform, and merely modified the base geometry loaded into the system earlier.

The most fundamental piece of functionality required to implement the above animation system is the ability to modify a transform's (joint's) parameters, such as orientation and translation, at run time. Here we describe how SCOOT's graphics system was developed from a high level to support this functionality and how early design decisions made IMPS' integration much simpler.

A requirement of the SCOOT system was that it support cross-platform development. Specifically, the underlying graphics system needed to run on both Windows NT and Silicon Graphics machines, the former for debugging, and the latter for speed. This sug-

gested the use of the Abstract Factory design pattern [2]. A set of Java-side, graphics system-independent interfaces was created, such as a `Transform`, containing methods like `setOrientation` and `setTranslation`. In addition, a factory class, the `GraphicsInterface`, was defined, containing methods like `loadFile` (load the base geometry VRML file for a character and return handles to the transforms) and `loadMotorFile` (load a VRML file specifying animation for a particular base geometry file and return the animation data.) The factory is the sole means by which clients may instantiate geometry. For this reason it is possible to swap graphics systems by instantiating a different implementation of the factory at the beginning of the program; this is a one-line change. The currently supported implementations are `CsGraphicsInterface` (Cosmo3D implementation, supported on both NT and SGI's Irix) and `PfGraphicsInterface` (Performer implementation, supported only on SGI hardware.) Each of these implementations contains a set of implementing classes for the graphics system interfaces, which contain native code which calls the underlying C++ graphics library when calls like `setOrientation` are made on a Java-side `Transform` object; see [11] for more details.

Since the `Transform` abstraction was already in place and being used properly by clients, integrating IMPS was made much simpler. The Performer implementation of the graphics system was modified to support loading of implicit surfaces as base geometry files (a change at the C++ level), as well as IMPS' specialized transform class (requiring both C++ and Java-side modifications), which supports the same interface as other transforms.

Implicit surface-based characters, therefore, are loaded in exactly the same fashion as geometrical ones. Loading the base geometry file returns a set of transforms, which happen to be specialized to the implicit surface library, but appear to clients to be the same as those returned from a standard geometry file. Animations can be loaded for the implicit surface using the same code as for geometrical characters. These animations are applied using the same code in the run time animation engine, which ultimately makes `setOrientation` and `setTranslation` calls on `Transform` objects.

4.1 Combining Geometrical and Implicit Objects

SCOOT transparently combines characters comprised of geometrical and implicit primitives in the same scene. However, they can not be mixed in the same VRML file; a character's

base geometry file may contain either polygons or implicit primitives, but not both. SCOOT provides primitives to make composition of these separate objects easier.

The World object, in which all of the characters exist, supports a constraint mechanism allowing one creature to grab another [11]; the grabbed creature is attached to the grabber's end effector, with an optional offset rotation and translation. There are at least two ways to implement this functionality. The first is to reparent the grabbed creature under the grabber's end effector in the underlying scene graph; in this way the underlying graphics library causes the appropriate motion of the grabbee, and no further computation is required by the application. The second is to compute the world-to-local transform for the end effector at the application level and orient the grabbed creature appropriately, without reparenting its geometry in the underlying scene graph. The latter approach, which is used by SCOOT, requires the end effector's world-to-local transform to be recomputed each frame, but has the advantage that it does not rely on the underlying scene graph structure.

Implicit surfaces in IMPS have hierarchical structure, but the generated geometry does not map to a scene graph in the underlying graphics library, since only one "skin" is created for multiple, hierarchically organized, implicit primitives. Because SCOOT implements grabbing at the application rather than the scene graph level, it is possible to attach implicit objects to polygonal ones. For example, polygons may be used to model a character's head and neck while implicit surfaces model its body. This is a powerful addition to the system because implicit surfaces are, generally speaking, better for modeling smooth objects but not finely detailed ones such as a face. IMPS' integration with SCOOT allows polygons to be used for fine detail in the regions of the character where they are more appropriate. An example of this is discussed in Section 5.1.

Chapter 5

Character Design and Animation

IMPS supports interactive character design most directly through its combined viewer, modeler, and 3D paint system. Running the OpenGL-based viewer, `glImplViewer`, or the Open Inventor-based version, `ivImplViewer`, on a VRML file containing implicit primitives brings up a window with the scene rendered using the appropriate polygonizer (also specified in the VRML file.) By default all implicit primitives are surrounded by 3D manipulators which can be translated, rotated, and scaled using the mouse, allowing interactive modeling of implicit surfaces in 3D. Pressing the "t" key toggles the 3D paint system; when it is active, all manipulators disappear and visible surfaces which have textures assigned to them in the VRML file can be interactively edited.

IMPS was designed as a research testbed for designing implicit surface-based characters suitable for real time, and therefore does not implement many features of more general commercial modeling packages. Specifically, editing of the hierarchy at run time is not supported; the VRML file must be edited to add all of the desired implicit primitives. The manipulator editors do not always expose all of the parameters of the underlying implicit primitives for interactive editing, so additional hand editing of the VRML file is necessary when, for example, changing the length (rather than the orientation) of a limb (line segment.)

Most importantly, there are no facilities for animation scripting built into IMPS. Designing a character animation package is a great deal of work, and an early design decision was to leverage existing commercial packages, like 3D Studio Max, for this purpose.

Building a character with IMPS begins with the modeling of the character's geometry.

A text editor is used to design the hierarchy of the character and thereby the animation knobs, or joints, to be animated later. The interactive viewer is used to place the implicit primitives relative to each other. Once the geometry is in place, the character's textures can be painted, again using the interactive viewer.

After the static model of the character's implicit surface-based geometry is created, 3D Studio Max is used to create animations for the character. This is begun by creating a hierarchy in 3DS of the same structure as the implicit surface-based character, and giving the transforms in 3DS the same names as the joints in the implicit surface VRML file. The character's geometry in 3DS is designed to match the implicit primitives as closely as possible; for example, a metaball would be replaced by a sphere, and a meta-line segment would be replaced with a cylinder. This geometry is only used as a placeholder, to give the artist visual feedback of the character's motion. Note that construction of the character's "mirror" in 3DS is currently an operation done by hand; this will be discussed further in Chapter 6.

Once the 3DS-based version of the character is in place, animations are designed using 3DS's built-in animation editor. A full description of the design principles of animation libraries for interactive characters is beyond the scope of this thesis, but the high-level goal is to create short animation "snippets", like a single step of a walk cycle, which can be attached together at run time to generate the character's motion. These animations are exported as VRML files read into SCOOT, the Synthetic Characters' Object-Oriented Toolkit, and entered as motor skills into the character's motor system. The implicit primitives in the VRML file designed with IMPS are animated by the keyframed transforms from 3D Studio Max, causing the implicit surface-based character to move.

5.1 Design of Beanito, the Mexican Jumping Bean

It is hard to imagine a more simple character than one made of two metaballs; this was the first full test of the character design process using IMPS. Beanito is a Mexican Jumping Bean whose sole animation is a jump cycle; he squashes in anticipation, stretches as he launches himself, glides through the air, and bounces upon landing.

What little modeling was necessary for Beanito's body was done with IMPS' interactive viewer. Two spheres were modeled in 3D Studio Max and their hierarchy set up to match

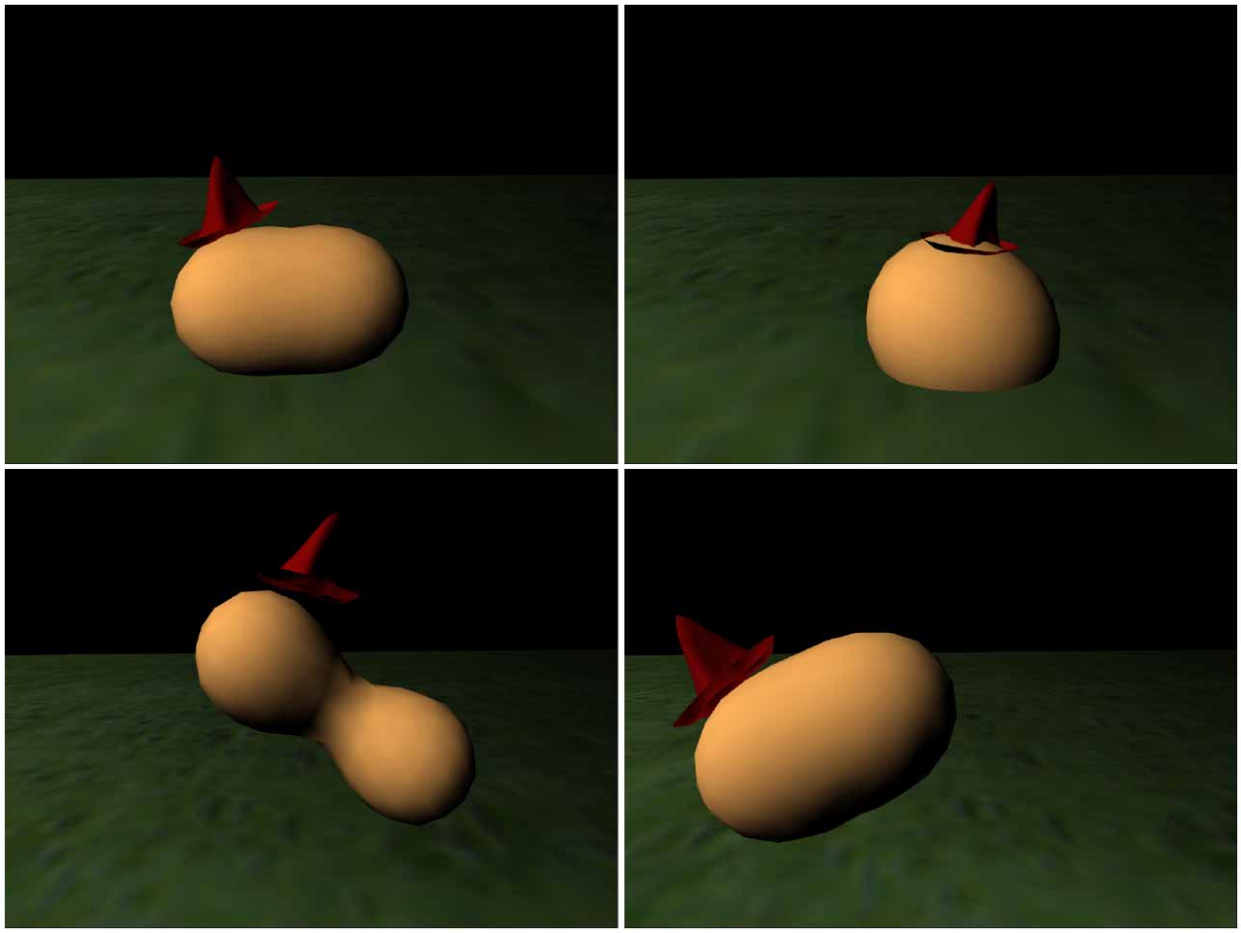

Figure 5-1: Beanito, the Mexican Jumping Bean.

the implicit surface VRML file. They were animated to generate the squash and stretch that would be present in the final implicit surface; bringing the spheres closer together in 3D Studio caused the implicit surface to squash, while separating them caused it to stretch. The view in 3D Studio did not exhibit these properties, however; it was necessary to export and play the animation in SCOOT to see the final result.

Beanito wears a hat on his head. The hat is a polygonal object modeled with 3D Studio and saved as a separate geometry file from the body of the character. At run time the hat is grabbed by the head's transform using the process described in Section 4.1, causing the hat to be moved and oriented each frame to lie at a fixed offset from the origin in the local coordinate system of the character's head. The fact that such grabbing of one character by another (in this case, the hat by the head) is implemented at the application level rather than down in the scene graph library (for example, by reparenting the hat under the head's

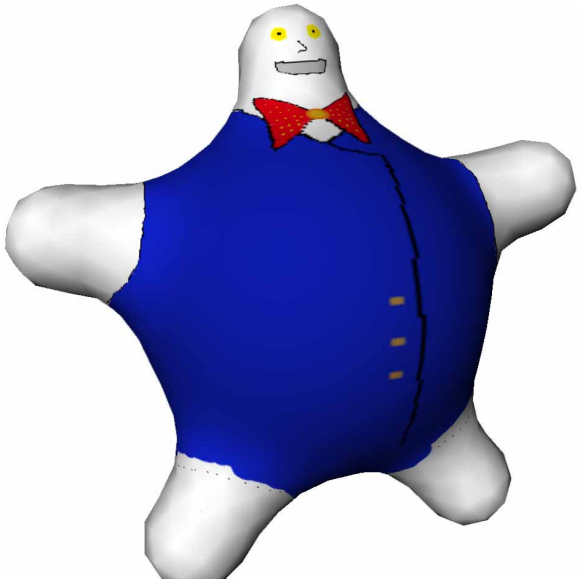

Figure 5-2: Flubber dressed up.

transform) allows implicit surface-based objects to be easily combined with non-deformable geometry, since implicit surfaces do not actually have a set of associated transforms in the underlying scene graph, although they can be controlled as if they did.

There were two primary problems discovered during the implementation of Beanito. First, squash and stretch were very difficult to gauge during animation in 3D Studio. The first few animations of the bean's jump cycle actually tore the two metaballs apart during the launch; several iterations were necessary to achieve the desired animation of the implicit surface. This could be improved by integrating IMPS more tightly with 3D Studio, for example as a plug-in. Second, and more significantly, the original animation was far too fast. IMPS' run time system requires that surfaces not move too quickly, because the application of differential transforms which causes holes at joints requires several iterations of the particle simulation to patch such holes, as shown in Figure 3-3. The original animation tore the surface apart at its center during the launch; it needed to be slowed down by a factor of four to solve this problem. More general solutions to this problem will be discussed in Chapter 6.

5.2 Well-Dressed Flubber

For the October 1998 sponsor meeting at the Media Lab, a character inspired by Disney's Flubber was created. The geometry is relatively simple: one metaball at the center surrounded by five meta-line segments. Untextured, the static geometry looks very similar to the characters in Disney's movie. IMPS' 3D painting functionality was used to paint a suit and tie onto the character's body, along with a very rudimentary face. The goal was a cartoon-like character with a high degree of dynamism, for example squash and stretch.

Despite the simplicity of the geometry, several problems were encountered during the construction of this character. Foremost among these was the lack of robustness of IMPS' renderer. The character requires several hundred triangles to provide a reasonable approximation to its geometry; despite the presence of the spatial subdivision mechanism, the particle simulation does not maintain a high enough frame rate (only about ten frames per second) for this number of particles to allow reasonable animation of this character. When the frame rate of the renderer drops off, numerical instability of the Witkin-Heckbert particle system becomes much more apparent. Particles not already close to the implicit surface tend to fly off at extremely high velocities. It was necessary to insert code to test for this condition and remove such particles from the simulation and the mesh.

Although the frame rate was low for this geometry, conclusions can be drawn from its construction and experiments with the interactive viewer. First, the squash and stretch expected to be afforded by the implicit surface representation were not apparent. This was an unexpected result given Beanito's successful use of these effects. The reason is that squashing and stretching occur only in blending regions of implicit surfaces, and the bulk of Beanito's body is this region. The blending region can be enlarged by increasing the blobbiness parameter of the implicit primitives; however, as discussed in Section 2.2, this decreases local control of the surface. In the case of this character, moving the arms closer to the body, rather than making the body bulge larger in that region, causes them to disappear into the stomach. Second, the visual appearance of the character creates the expectation that the character would exhibit a very "bouncy" animation look: for example, that the stomach would jiggle during a walk cycle. Unfortunately, and very significantly, IMPS does not provide the set of control knobs necessary to generate such motion. IMPS was designed to mirror the animation controls used for polygonal characters, which are typically only rotations and occasionally translations of joints. In the absence of squashing and stretching expected to be afforded by radial translations of the limbs, the character appears very static

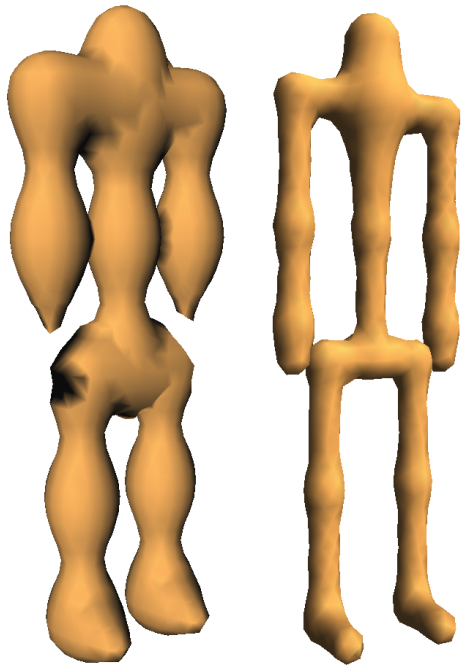

Figure 5-3: The biped before and after IMPS' implicit function reformulation. The figure on the left is rendered using IMPS' Marching Cubes implementation, while the figure on the right is triangulated using `Tesselator3MP`.

and rigid, since it does not deform significantly as the arms are moved. While it would be possible to make the stomach translate up and down during a walk cycle, it is not possible to make it, for example, sag around the hips. Third, IMPS' texturing approach causes a texture seam to occur across the bow tie of the character, which essentially means the head can not be animated relative to the body. These problems and potential solutions will be discussed further in Chapter 6.

5.3 Simple Biped

To illustrate IMPS' use of the hierarchical structure of characters, a simple biped was built (`tess3SimpleBip01.wrl`). The biped is composed of eighteen line segments arranged into a hierarchy using transforms.

Early versions of this biped did not work well; the joints ballooned out and the middle regions of the limbs vanished. Reducing the radii of the line segments only exacerbated the problem, as illustrated in Figure 5-3. The problem was eventually discovered to be the coupling of the implicit primitives' radii with their blending parameters, as discussed

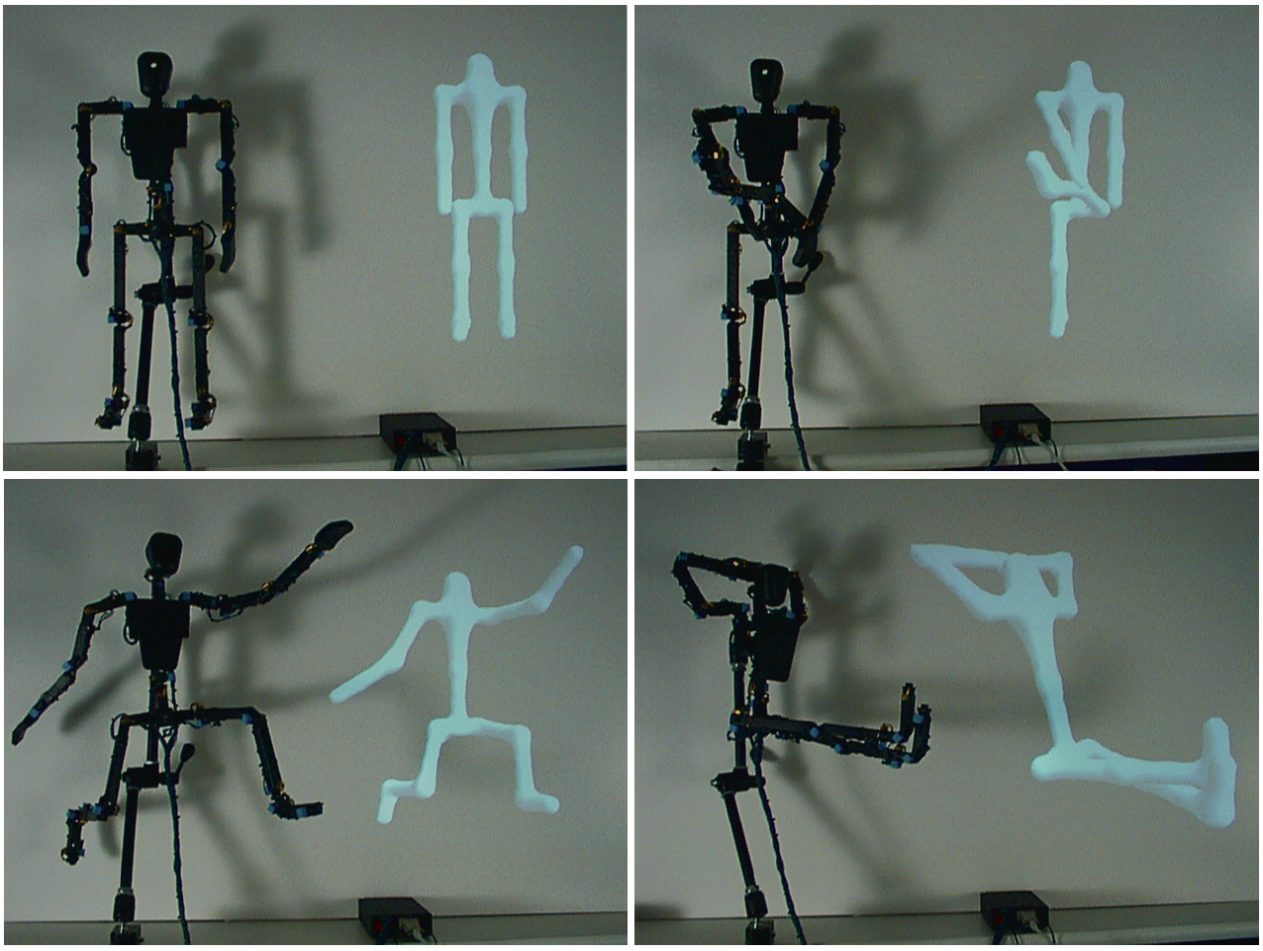

Figure 5-4: Left to right, top to bottom: calibration pose; gymnastics, illustrating no unwanted blending between hands, knees and hips; Saturday Night Fever; relaxing after a hard day's work.

in Section 2.2. Reformulating the primitives' field functions allowed much finer control of the biped's shape. In addition, the singularities in the middle of the figure's limbs which had been present vanished; this allowed IMPS' parallelized gradient-based polygonizer, `Tesselator3MP`, to be used to render the figure instead of the Marching Cubes algorithm, yielding a speedup of roughly a factor of 25 (Tables 3.1 and 3.2), and allowing the skeleton cache to assist in the figure's animation (Section 3.7).

To demonstrate the figure's interactivity, a small stand-alone program was written to animate the biped from data acquired using Digital Image Design's Monkey 2 (`http://www.didi.com/`). The Monkey provides absolute orientation information for each joint when it is queried. Since IMPS was designed to handle zeroth-order control of joints, it was straightforward to fill the biped's transforms with the Monkey's data. The Monkey can

be used to pose the character interactively, as illustrated in Figure 5-4. The Monkey's data acquisition call, which is made every frame, takes roughly forty milliseconds and reduces the frame rate from ten frames per second to seven. This call could be made asynchronously to the rest of the application to achieve the higher frame rate.

This character illustrates the benefits of IMPS' differential transform modification to Witkin and Heckbert's algorithm (Section 3.7). When differential transforms are disabled, reverting the particle simulation to Witkin and Heckbert's original algorithm modulo their surface motion parameters, the figure breaks apart with the slightest rotation about its joints, as particles further down the hierarchy suddenly find themselves stranded because of the large motion of the underlying surface due to the figure's long limbs. This demonstrates the necessity of keeping track of the hierarchical structure of the implicit surface. This particular character does not exhibit the tearing illustrated in Figure 3-3 because the joints comprise a relatively small portion of the surface area and because the physical device imposes a constraint on how quickly the joints can move.

Of the three examples shown here, the biped best illustrates the types of characters IMPS was originally designed to build: long-limbed, skeletal creatures. Compared to the Flubber character, the biped is more easily animatable because it does not rely as heavily on the blending properties of its implicit primitives to achieve its shape, and because it is more highly articulated via the use of transforms.

As with the other two characters, construction of the biped made clear a set of problems. First, texturing the character would be very difficult, because while the geometry appears to be smooth, the texture coordinate parameterization which IMPS provides has discontinuities at each joint, which would create visible seams as the character animates. Because of the number of animated joints, it is doubtful that effective textures which hide these seams could be created. Second, the blend graph inferred from the hierarchy does not work properly in the neck region. When the neck is articulated, it is prevented from blending with the characters' shoulders; however, this causes the shoulders and neck to "fight" for ownership of the particles, leading to bubbling of the surface. Both a mechanism for specifying the blend graph independently of the hierarchy and a better solution to the unwanted blending problem which does not cause instabilities in the particle system are needed. Third, the frame rate is still too low for interactive character animation. The Monkey hides this problem to some degree because the physical device enforces some constraints

on how quickly joints can rotate; however, experience has shown that ten frames per second peak performance is not sufficient to depict a fast-moving or highly animated character. These issues are discussed further in the next chapter.

Chapter 6

Discussion and Future Work

6.1 Discussion

IMPS was originally inspired by the desire to eliminate visible seams in the characters designed by our group. It appeared that using an implicit representation would provide a simple solution; replacing limbs constructed from rigid, non-deformable geometry with lines of charge and rendering the resulting potential field would instantly provide a seamless mesh while still maintaining the joint-level control with which our animators were familiar. The biped in the previous chapter illustrates this goal.

Many papers in the field of implicit surfaces have as their sole motivation the fact that implicit surfaces are good for modeling smooth, organic shapes. This thesis is no exception and this was, as discussed, the original reason for researching the applicability of implicit surfaces to interactive animated characters. Experience with implicit surfaces has indicated that more motivation than this is needed, as well as more critical discussion of the advantages and disadvantages of the representation.

6.1.1 Problems and Lessons Learned Related to IMPS' Implementation

IMPS' renderer is not fast enough to animate interactive characters. This problem might be soluble using a different polygonization algorithm. Instability of the Witkin-Heckbert algorithm makes working with the particle system difficult; particles far from the surface "blast off" at high velocities, making implementation of an unwanted blending solution difficult (Section 3.7). An even more significant problem is the fact that the simulation does not converge; all of the particles are being simulated all of the time, even if the

surface is not changing in a certain region. IMPS' differential transform modification would theoretically allow many of the particles to be removed from the simulation and animated solely by rigid-body motion once the surface has been polygonized. However, identifying regions of the surface not modeled well by this approximation would likely be difficult, as the implicit function needs to be evaluated in order to do so, and as has been discussed, this is currently the bottleneck in the algorithm. Section 6.2 offers suggestions on directions which could be taken to address this problem.

IMPS does not provide suitable animation controls for animating implicit surface-based characters. The initial assumption was made that the animation controls for polygonal characters, primarily rotations and occasionally translations, would transfer well to hierarchically structured, implicit surface-based characters. This assumption was incorrect because implicit surfaces look blobby and therefore need to be able to act blobby. Non-uniform scales are much more important than with polygonal characters, and effects such as bending are desirable, as was seen in the construction of the Flubber character. The design decision to sample primitives in the world coordinate system (Section 3.2) precluded the use of non-uniform scales in IMPS, which is a serious limitation. Transforming sample points into primitives' local coordinate systems requires increased computation time, but is necessary to be able to animate such deformations. Pentland and Williams' application of modal deformations to non-blending implicit surfaces [9] illustrates how higher-order deformations such as bending could be applied on a per-primitive basis. Section 6.2 discusses this issue further.

Squash and stretch is not free. The canonical first test of implicit surfaces, animation of two metaballs, gives the misleading impression that implicit surfaces are "soft" objects. In fact, soft effects like squash and stretch occur only in the filleting, or blending, regions of implicit surfaces. If the blobbiness of the surface is increased to cause these effects to dominate, local control of the shape of the surface is lost. As discussed above, if blobby effects of the surface are desired, it is necessary to model the deformations of the primitives. The cartoon-like appearance of implicit surfaces belies the fact that they do not inherently provide animation controls for such soft effects. This is a conclusion reached through the experience of having attempted to model characters taking advantage of the representation.

6.1.2 Problems with the Implicit Surface Representation

The lack of a well-behaved parameterization for implicit surfaces is fundamental and precludes many additions which would make the representation more useful. Texturing implicit surfaces is a problem for which there is not yet a good real-time solution. The expected behavior of the surface is for the texture to stretch, rather than break, as the surface deforms. The real-time or interactive-time algorithms currently proposed do not provide this behavior. Pedersen's solution, which could, achieves a well-behaved parameterization by replacing the implicit surface with a set of bicubic patches which are constrained to lie on it. Unfortunately, as mentioned earlier, this solution does not yet approach real time and has not yet been applied to animated surfaces.

Implicit surfaces do not provide good control over the shape of the surface. Especially in an interactive or real-time system, the choice of primitives is limited to those whose field functions can be evaluated quickly, precluding the use of general polyhedra as primitives. Bulging at overlapping regions makes it difficult to obtain fine modeling control. Solutions to this problem such as convolution surfaces (see, for example, Bloomenthal's article in [38]) do not appear to be close to real time. Sclaroff and Pentland [22] propose the use of displacement maps for independent (non-blending) implicit surfaces to increase the modeling power of each primitive. However, for blended implicit surfaces, it is not possible to use displacement maps to model local detail, because evaluation of the displacement map assumes a well-behaved parameterization of the underlying surface, which, as has been discussed, is not present for blended surfaces.

6.2 Future Work

There are several extensions which could be made to IMPS which would improve its usefulness in the application of implicit surfaces to hierarchical animated characters.

The extension which would arguably provide the most benefit would be to incorporate per-primitive deformations to allow more complex animation control. Modal deformations, as described by Pentland, Williams and Sclaroff [9, 22] would provide the best starting point. Their implicit surface work focused on the use of independent (i.e., non-blending) superquadrics with modal deformation matrices to achieve physical simulation and more precise modeling. Incorporating such a system into IMPS would likely be straightforward;

each primitive would need to store a set of modal deformation parameters. As the modal deformations must be computed in the object's local coordinate system, IMPS' primitives would need to be modified to allow sampling in local, rather than world, coordinates (Section 3.2). This modification would have the added benefits of vastly simplifying the representations of the implicit primitives and allowing primitives to be animated using non-uniform scales, though the question remains whether the increased computation time will significantly reduce performance.

The biggest challenge in this particular extension is implementing a system for controlling such deformations. IMPS leveraged an existing commercial package, 3D Studio Max, to some degree of success because 3D Studio already supported the animation paradigm upon which IMPS was based, namely time-varying joint angles. Unfortunately, modal deformations are not supported by any major modeling and animation package, increasing the amount of implementation required for this extension. One solution is to implement an animation editor for IMPS which would support both kinematic and per-primitive modal deformation animation. Another is to incorporate IMPS as a plugin to a package such as 3D Studio and leverage its kinematic animation tools. Either of these solutions would also solve the significant problems associated with creating a 3D Studio "shadow" of the character's implicit geometry, including duplication of work, failure to fit the geometry properly, and lack of immediate feedback during the animation process.

A more stable polygonization algorithm for implicit surfaces than IMPS' current particle system-based renderer is needed. As has been discussed, particles far from the surface tend to "blast off" at high velocities in Witkin and Heckbert's formulation. This is a problem not only for surface motion in general (hence their addition of the surface parameter term discussed in Section 3.7) but also for the chosen solution to the unwanted blending problem (implicit patches, discussed in the same section.) In addition, because of its particular solution to the unwanted blending problem, which is tied to the dynamics of the particle simulation, IMPS' renderer fails to completely polygonize surfaces where an implicit primitive in the middle of the hierarchy (i.e., the sole link between a parent and child, such as the elbow between the upper and lower arm) is small enough that it is completely enveloped by an adjacent primitive. Finally, a solution is needed for the tearing that occurs at a character's joints, caused both by the application of differential transforms and by the absence of hole-patching logic in the triangulation algorithm.

Two recent papers on implicit surface polygonization would provide good starting points for new renderers for IMPS. Desbrun, Tsingos, and Gascuel's algorithm for the interactive polygonization of implicit surfaces [33] is structured around the notion of implicit primitives and therefore might be most compatible with IMPS' data structures. It should be possible to generate an "approximate triangulation" to skin the overall surface rather than per primitive as they do in their paper. Velho, de Figueiredo, and Gomes's multiresolution implicit surface polygonizer [43] would provide the additional significant benefit of automatic view-based simplification for the generated mesh. It is possible that Desbrun et al.'s method of sampling the surface (using rays attached to the local coordinate systems of primitives) could be used to generate the base mesh for Velho et al.'s algorithm to easily support hierarchical implicit surfaces, and that differential transforms could be used to make the multiresolution algorithm more incremental.

6.3 Conclusion

IMPS represents a step towards the application of implicit surfaces to the domain of interactive animated characters. It takes advantage of the hierarchical structure of characters' geometry to accelerate existing algorithms for sampling implicit surfaces and provide solutions to common problems like unwanted blending.

More work is needed to address limitations in the system's design such as the lack of non-uniform scales. Extensions to the system such as per-primitive modal deformations could provide the necessary animation controls required to make its characters act blobby as well as look blobby.

Unfortunately, the lack of parameterization of implicit surfaces makes texturing operations at interactive rates difficult, if not impossible, to achieve in a principled manner. More research is needed to determine whether there is a good solution to this problem. In the domain of interactive animated characters, where texturing is strongly desired to add color and detail, this is an especially serious problem, and suggests that alternative representations for deformable geometry may be more applicable.

Generalized cylinders (see, for example, Bloomenthal [3]) are a well-studied technique which have reasonable control mechanisms and a well-defined texture coordinate parameterization. Protozoa (`http://www.protozoa.com/`), for example, has used generalized

cylinders extensively in the construction of interactive-rate characters driven by motion capture data. Subdivision surfaces (see, for example, Catmull and Clark [4]) are another technique which has recently received renewed attention after Pixar's use of the technique in modeling the character Geri in the animated short *Geri's Game* (DeRose et al. [5].) A recent paper by Stam [14] shows that it is possible to directly evaluate the subdivision surface for any point on the control mesh, giving the subdivision surface a well-behaved parameterization essential for texturing operations. Applying subdivision surfaces to the interactive character domain, and addressing problems such as how to control the control mesh and reach real-time rates [10], would be an interesting area for future research.

Bibliography

[1] Blinn, James F. "A Generalization of Algebraic Surface Drawing," *ACM Transactions On Graphics*, Vol. 1, No. 3, July 1982, pp. 235-256.

[2] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley: Reading, Massachusetts, 1995.

[3] Bloomenthal, Jules. "Calculation of Reference Frames Along a Space Curve," in Glassner, Andrew S., ed. *Graphics Gems.* Academic Press: San Diego, 1990, 1998.

[4] Catmull, E. and Clark, J. "Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes," *Computer Aided Design*, vol. 10, no. 6, pp. 350-355, 1978.

[5] DeRose, Tony, Michael Kass, and Tien Truong. "Subdivision Surfaces in Character Animation," *Computer Graphics*, Proceedings of SIGGRAPH 98, July 19-24, 1998, pp. 85-94.

[6] Desbrun, Mathieu, Peter Schröder, and Alan Barr. "Interactive animation of structured deformable objects," In *Graphics Interface '99* (June 99, Kingston, Canada).

[7] Hartman, Jed and Josie Wernecke. *The VRML 2.0 Handbook.* Addison-Wesley: Reading, Massachusetts, 1996.

[8] Opalach, Agata and Steve Maddock. "High Level Control of Implicit Surfaces for Character Animation," *Implicit Surfaces '95*, Proceedings of the Eurographics Workshop on Implicit Surfaces, April 18-19, 1995.

[9] Pentland, Alex and John Williams. "Good Vibrations: Modal Dynamics for Graphics and Animation," *Computer Graphics*, Proceedings of SIGGRAPH 89, pp. 215-222.

[10] Pulli, Kari and Mark Segal. "Fast Rendering of Subdivision Surfaces," *Visual Proceedings*, SIGGRAPH 96 Technical Sketches, 1996, p. 144.

[11] Russell, Kenneth B. and Bruce M. Blumberg. "Behavior friendly graphics," *CGI 99*, Proceedings of Computer Graphics International 99, Alberta, Canada, June 7-11, 1999. To appear.

[12] Schraudolph, Nicol N. "A Fast, Compact Approximation of the Exponential Function," *Neural Computation*, Volume 11, Number 4. To appear. `ftp://ftp.idsia.ch/pub/nic/exp.ps.gz`

[13] Shoemake, Ken and Tom Duff. "Matrix animation and polar decomposition," *Graphics Interface '92*, pp. 258-64.

[14] Stam, Jos. "Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values," *Computer Graphics*, Proceedings of SIGGRAPH 98, July 19-24, 1998, pp. 395-404.

[15] Thomas, Frank and Ollie Johnson. *The Illusion of Life: Disney Animation.* Hyperion: New York, 1981.

[16] Wernecke, Josie. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2.* Addison-Wesley: Reading, Massachusetts, 1994.

Interactive Animated Character Systems

[17] Blumberg, Bruce. "Old Tricks, New Dogs: Ethology and Interactive Creatures," Ph.D Thesis, Massachusetts Institute of Technology, 1996.

[18] Perlin, Ken and Athomas Goldberg. "Improv: A System for Scripting Interactive Actors in Virtual Worlds," *Computer Graphics*, Proceedings of SIGGRAPH 96, August 4-9, 1996, pp. 205-216.

[19] Blumberg, Bruce et al. "Swamped!: Using Plush Toys to Direct Autonomous Animated Characters," *SIGGRAPH 98 Conference Abstracts and Applications*, p. 109. `http://characters.www.media.mit.edu/groups/characters/swamped/`

[20] Galeyan, Tinsley et al. "The Virtual FishTank," *SIGGRAPH 98 Conference Abstracts and Applications*, p. 116.

Field Functions and Blending Control

[21] Wyvill, Brian and Geoff Wyvill. "Field functions for implicit surfaces," *The Visual Computer*, Vol. 5 (1989), pp. 75-82.

[22] Sclaroff, Stan and Alex Pentland. "Generalized Implicit Functions For Computer Graphics," *Computer Graphics*, Proceedings of SIGGRAPH 91, Volume 25, Number 4, July 1991, pp. 247-250.

[23] Guy, Andrew and Brian Wyvill. "Controlled Blending for Implicit Surfaces using a Graph," *Implicit Surfaces '95*, Proceedings of the Eurographics Workshop on Implicit Surfaces, April 18-19, 1995.

[24] Blanc, Carole and Christophe Schlick. "Extended Field Functions for Soft Objects," *Implicit Surfaces '95*, Proceedings of the Eurographics Workshop on Implicit Surfaces, April 18-19, 1995.

Polygonization

[25] Wyvill, B., McPheeters, C., and Wyvill, G.. "Data Structures for Soft Objects," *The Visual Computer*, Vol. 2, Number 4, pp. 227-34.

[26] Lorensen, William E. and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, Proceedings of SIGGRAPH 87, Volume 21, Number 4, July 1987, pp. 163-169.

[27] Düurst, Martin J. "Letters: Additional Reference to Marching Cubes," *Computer Graphics*, Volume 22, Number 2, April 1988.

[28] Nielson, Gregory M. and Bernd Hamann. "The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes," *Proceedings of Visualization 91*, 1991, pp. 83-91.

[29] Watt, Alan and Mark Watt. *Advanced Animation and Rendering Techniques.* New York, NY: ACM Press, 1992.

[30] de Figueiredo, Luiz Henrique, Jonas de Miranda Gomes, Demetri Terzopoulos, and Luiz Velho. "Physically-Based Methods for Polygonization of Implicit Surfaces," *Graphics Interface '92*, pp. 250-257.

[31] Witkin, Andrew P, and Paul S. Heckbert. "Using Particles to Sample and Control Implicit Surfaces,", *Computer Graphics*, Proceedings of SIGGRAPH 94, July 24-29, 1994.

[32] Bloomenthal, Jules and Keith Ferguson, "Polygonization of Non-Manifold Implicit Surfaces," *Computer Graphics*, Proceedings of SIGGRAPH 95, August 6-11, 1995, pp. 309-316.

[33] Desbrun, Mathieu, Nicolas Tsingos, and Marie-Paule Gascuel. "Adaptive Sampling of Implicit Surfaces for Interactive Modeling and Animation," *Implicit Surfaces '95*, Proceedings of the Eurographics Workshop on Implicit Surfaces, April 18-19, 1995.

[34] Opalach, Agata and Steve Maddock. "Speeding Up Grid-Data Generation for Polygonisation of Implicit Surfaces," *Eurographics 95*, Proc. 13th Eurographics UK Chapter Annual Conference, pp. 153-160.

[35] Su, Peter and Robert L. Scot Drysdale. "A Comparison of Sequential Delaunay Triangulation Algorithms," Proceedings of the 11th Annual Symposium on Computational Geometry, Vancouver, British Columbia, Canada, June 5-7, 1995.

[36] Rodrian, Hans-Christian and Hardy Moock. "Dynamic Triangulation of Animated Skeleton-Based Implicit Surfaces," *Implicit Surfaces '96*, Proceedings of the Eurographics/SIGGRAPH Workshop on Implicit Surfaces, October 7-8, 1996.

[37] Rösch, Angela, Matthias Ruhl, and Dietmar Saupe. "Interactive Visualization of Implicit Surfaces with Singularities," *Implicit Surfaces '96*, Proceedings of the Eurographics/SIGGRAPH Workshop on Implicit Surfaces, October 7-8, 1996.

[38] Bloomenthal, Jules, Chandrajit Bajaj, Jim Blinn, Marie-Paule Cani-Gascuel, Alyn Rockwood, Brian Wyvill, and Geoff Wyvill. *Introduction to Implicit Surfaces.* San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1997.

[39] Heckbert, Paul, "Fast Surface Particle Repulsion", *New Frontiers in Modeling and Texturing*, SIGGRAPH 97 Course Notes, August 3-8, 1997, pp. 95-114.

[40] Stander, Barton T. and John C. Hart, "Guaranteeing the Topology of an Implicit Surface Polygonization for Interactive Modeling," *Computer Graphics*, Proceedings of SIGGRAPH 97, August 3-8, 1997, pp. 279-286.

[41] Crossno, Patricia and Edward Angel, "Isosurface Extraction Using Particle Systems," *Proceedings of Visualization '97*, IEEE Computer Society Press, pp. 495-498.

[42] Hartmann, Erich. "A marching method for the triangulation of surfaces," *The Visual Computer*, Vol. 14 (1998), pp. 95-108.

[43] Velho, Luiz, Luiz Henrique de Figueiredo, and Jonas Gomes. "A Unified Approach for Hierarchical Adaptive Tesselation," Preprint, Instituto de Matematica Pura e Aplicada, Rio de Janeiro, Brazil. `http://www.visgraf.impa.br/People/lvelho/`

Texturing

[44] Peachey, Darwyn R. "Solid Texturing of Complex Surfaces," *Computer Graphics*, Proceedings of SIGGRAPH 85, Volume 19, Number 3, July 1985, pp. 279-286.

[45] Perlin, Ken. "An Image Synthesizer," *Computer Graphics*, Proceedings of SIGGRAPH 85, Volume 19, Number 3, July 1985, pp. 287-296.

[46] Hanrahan, Pat and Paul Haeberli. "Direct WYSIWYG Painting and Texturing on 3D Shapes," *Computer Graphics*, Proceedings of SIGGRAPH 90, August 1990, pp. 215-223.

[47] Pedersen, Hans Køhling, "Decorating Implicit Surfaces," *Computer Graphics*, Proceedings of SIGGRAPH 95, August 6-11, 1995, pp. 291-300.

[48] Pedersen, Hans Køhling, "A Framework for Interactive Texturing on Curved Surfaces," *Computer Graphics*, Proceedings of SIGGRAPH 96, August 4-9, 1996, pp. 295-302.

[49] Zonenschein, Ruben, Jonas Gomes, Luiz Velho, and Luiz Henrique de Figueiredo, "Texturing Implicit Surfaces with Particle Systems," *Visual Proceedings*, SIGGRAPH 97 Technical Sketches, 1997, p. 172.

[50] Tigges, Mark, and Brian Wyvill, "Texture Mapping the BlobTree," *Implicit Surfaces '98*, Proceedings of the Eurographics/SIGGRAPH Workshop on Implicit Surfaces, June 15-16, 1998.

[51] Zonenschein, Ruben, Jonas Gomes, Luiz Velho, Luiz Henrique de Figueiredo, Mark Tigges, and Brian Wyvill, "Texturing Composite Deformable Implicit Objects," *Proceedings of SIBGRAPI 98*, Sociedade Brasileira de Computacao, IEEE Press, 1998.

Appendix A

Examples and Per-Class Documentation

A.1 Obtaining the Source Code

IMPS' source code is available from `http://www.media.mit.edu/~kbrussel/imps/`. It is released under the MIT Media Lab's academic license, which allows free use for educational and personal purposes.

A.2 Usage Notes and Examples

The recommended polygonizer to use is `Tesselator3MP`, the parallelized version of the renderer described in Section 3.7. Many of the improvements made to the polygonization algorithm, including texture mapping, have not been ported back to the single-threaded version, `Tesselator3`. In addition, `Tesselator3MP` running in single-threaded mode is more efficient than `Tesselator3`.

`glImplViewer` is IMPS' OpenGL-based interactive viewer, modeler and painting system. `ivImplViewer` is an earlier implementation based on Silicon Graphics' Open Inventor. `glImplViewer` is currently the most portable, and recommended, version.

Occasionally the viewer window and manipulators may appear but the surface may not; if this situation occurs, reset the renderer by pressing the "r" key (see below).

A.2.1 Using `glImplViewer` and `ivImplViewer`

Each of the viewers takes the name of a VRML file containing IMPS' nodes as its argument. Each also takes three command line options:

`-n` turns off manipulators, preventing interactive modification of the surface.

`-p` shows the vertices of the triangles as well as the shaded triangles themselves, and is a useful tool for debugging.

`-s` *size* sets the point size of displayed vertices to *size*.

The viewers currently have only a minimal 2D user interface built with FLTK (`http://fltk.easysw.com/`). A reset button can be made visible by setting the renderer's `showResetButton` field to `TRUE`. `glImplViewer` also uses FLTK's color chooser widget for its 3D painting system. This version of the interactive viewer has a popup menu available by clicking the right mouse button in the viewer's window which allows interactive modification of some of the viewer's parameters. The analagous functionality in `ivImplViewer` is provided automatically by Open Inventor.

Keyboard commands supported by the renderers:

Key	Description
`r`	Reset the polygonizer
`t`	Toggle 3D painting mode on/off
`s`	Save the scene currently in the viewer to a new file name
`[0-9]`	Change painter's brush size (1 = smallest (default), 0 = largest)

A.2.2 Examples

Colored Metaballs

Figure A-1 contains the text from `tess3TransformedBlobs.wrl`, containing two differently-colored metaballs. The scene graph is hierarchically structured, so that one metaball is the child of the other; this is indicated by the fact that a `RigidTransform` parents the hierarchy. When the scene is displayed, two manipulators appear, one surrounding each metaball; these correspond to the transforms containing the primitives, not the primitives' center points themselves; see Section A.2.2.

```
#VRML V2.0 utf8

Tesselator3MP {
  deltaT 0.045
  desiredRadius 0.3
  queryRadius 0.8
  energyCoeff 10
  surfaceCoeff 30
  fissioningFrac 0.5
  isoValue 1.0
  useSimpleRadii TRUE
  numThreads 3
  updatesPerRender 3
  colorWidth 3.0
  doDifferentialTransforms TRUE
  blendingFunction ExponentialBlend {
   variance 2.0
  }
  spatialSubdivision UniformHash {
   cubeSize 0.6
   bitsPerDim 3
  }
  sceneGraph RigidTransform {
   children [
    Metaball {
     material Material {
      diffuseColor 0.29 0.41 0.44
     }
    }
    RigidTransform {
     translation 2.0 2.0 0.0
     children [
      Metaball {
       material Material {
        diffuseColor 0.8 0.2 0.2
       }
      }
     ]
    }
   ]
  }
}
```

Figure A-1: Example derived from `tess3TransformedBlobs.wrl`.

This example illustrates the use of the top-level blending function (`blendingFunction ExponentialBlend`) instead of one embedded in the scene graph as in the case of flat scene graphs (see `tess3GaussBlob.wrl` for an example.) This is necessary for hierarchical scene graphs because the skeleton cache implements IMPS' solution to the unwanted blending problem as multiple independent groups of shapes, each with their own blending function, which is copied down into the skeleton cache from that specified in the polygonizer.

Flubber

The sequence of files `flubber.wrl` through `flubber8.wrl` show the assembly and painting of the well-dressed flubber character described in Section 5.2.

Flubber was created by specifying the metaball and five line segments in the VRML file and using the interactive editor to position them appropriately. Empty (pure white) textures were used as the starting point, and the 3D painting system was used to paint detail onto the character. The save function was used frequently to checkpoint the model in case the polygonizer crashed. Note that the save routines rename and save new copies of all of the textures in the model for convenience.

Simple Biped

The file `tess3SimpleBip01.wrl` contains the geometry for the biped character described in Section 5.3. It is a hierarchical scene graph (see below) rooted at the pelvis and branching to give the implicit surface the structure necessary for animation.

The biped was created by drawing line segments on graph paper and converting these into a scene graph of metalines by hand using a text editor. As the geometry was designed to follow a regular structure, no modifications in the interactive modeler were required.

Distinction Between Hierarchical and Flat Scene Graphs

IMPS supports both hierarchically organized and flat scene graph structures for implicit primitives. Hierarchical scene graphs are defined as those which have a `RigidTransform` as their root. Flat scene graphs have a blend function like `ExponentialBlend` or `EllipticalBlend` as their root, with potentially many implicit primitives as children of the blend. A `RigidTransform` may, however, be contained as a child in a flat scene graph, since it inherits from `ImShape`.

From the user's standpoint, the fundamental difference between hierarchical and flat scene graphs is the meaning of the 3D manipulators in the scene. For a hierarchical scene graph, one manipulator is created for each `RigidTransform` node and placed at its origin. For a flat scene graph, however, manipulators are created for each primitive, and some primitives, like `LineSeg`s, may have more than one manipulator (in this case, one for each endpoint).

From an implementor's standpoint, `Tesselator3MP` creates a skeleton cache only for hierarchical scene graphs. This means that IMPS' unwanted blending rule (Section 3.7) will only apply to this type of graph. The flag enabling differential transforms only makes sense if the scene graph is hierarchical. Per-vertex colors and texture coordinates also work only for hierarchical scene graphs. As mentioned above, the skeleton cache inherits its blending function from that specified in the `blendingFunction` field of `Tesselator3MP`.

Polygonizers such as `MarchingCubes` which do not understand hierarchical scene graphs can still render such scenes with two limitations. First, unwanted blending will appear since the skeleton cache and its associated blend graph is not created. Second, it is more difficult to construct the scene graph because in order to apply the blend function at the correct time and precisely once per primitive, it is necessary to surround each shape with the appropriate blending node, as in `mcSimpleBip01.wrl`.

A.3 Per-Class Documentation

A.3.1 Nodes: Shapes, Blends and Tesselators

- **ImBlancSchlickBlend** Blanc and Schlick's [24] blending function. Interprets `ImShape`'s `blendParam` field as "hardness".
- **ImBlend** Base class for all blending functions supporting multiple children. Interface was extended from original to support skeleton cache; see `ImBlend.h`.
- **ImEllipsoid** Meta-ellipsoid shape.
- **ImEllipticalBlend** Pairwise elliptical blend from Alyn Rockwood's paper in [38].
- **ImExponentialBlend** Exponential ("Gaussian") blend function similar to that proposed by Blinn.

- **ImLineSeg** Meta-line segment supporting differing endpoint radii.
- **ImMarchingCubes** Marching Cubes polygonizer based on Watt and Watt's implementation.
- **ImMetaball** Metaball shape.
- **ImNode** Base class for all of IMPS' node types.
- **ImParticleSim** Witkin-Heckbert particle simulator.
- **ImRenderer** Base class for all polygonizers: the marching cubes, particle simulator, and all tesselators are derived from this.
- **ImRigidTransform** Transform supporting only rotation and translation. This is the only type of transform node supported by IMPS.
- **ImShape** Base class for all implicit shapes.
- **ImTesselator** First experimental polygonizer: Witkin-Heckbert plus springs.
- **ImTesselator2** Second experimental polygonizer: Surface walking plus springs.
- **ImTesselator3** Third experimental polygonizer: Witkin-Heckbert plus approximate triangulation.
- **ImTesselator3MP** Multi-processor and most recent version of above polygonizer. Supports skeleton caches, differential transforms, texture mapping, and 3D painting.
- **ImTexGen** Base class for all per-primitive texture coordinate generators.
- **ImTexGenPlane** Projects a planar texture along a ray; similar to a slide projector.
- **ImTexGenSphere** Wraps a texture around a sphere, mapping (u, v) to (theta, phi) in spherical coordinates.
- **ImUniformHash** Paul Heckbert's spatial subdivision data structure [39] which hashes 3D space into buckets.

A.3.2 Auxiliary Classes

- **Im2DArray** Dynamically allocated 2D array class used in marching cubes renderer.
- **Im3DPaint** 3D painter class which attaches to a renderer, obtains textures, and receives mouse events, causing modifications to the textures.
- **ImBrush** Brush class used by the 3D painter.
- **ImEdgeMap** Used to keep track of which vertices are on the edge of the currently polygonized patch in `ImTesselator2`.
- **ImEdgeMap2** Enhanced version of original edge map allowing parts of the loop of vertices on the edge to be connected, creating sub-loops.
- **ImFastTimer** Wrapper for individual platforms' highest-resolution hardware timers. Currently ported to Irix and NT.
- **ImRayCast** Ray-triangle intersection routines.
- **ImSkeletonCache** Supports animation of hierarchically organized implicit surface scene graphs, as well as functionality like unwanted blending computations, color blending, and texture mapping.
- **ImSys** Initialization routine for IMPS; `ImSys::init` must be called before doing any work with the library.
- **ImTexture** Minimal wrapper for dealing with reading and writing of textures. Textures are resources which are reference counted and should only be referenced using texture references.
- **ImTextureRef** Reference to a texture; this is the public interface for dealing with textures.
- **ImTriangle** Notion of a triangle; used in `Tesselator3` and `Tesselator3MP`. Primarily used to determine whether a given triangle is already in the mesh.
- **ImWHParticle** Data structure wrapping all state for the Witkin-Heckbert particle simulator. Prior to `Tesselator3`, this data structure was broken out into its components and individual vectors of, for example, position and velocity information were

stored independently. This was unmaintainable for development purposes, but in the multiprocessor version it is likely that breaking the data structure back out will cause fewer cache line invalidations during the update cycle.

Appendix B

Information for the Developer

IMPS is based on a C++ library supplying an API for a subset of VRML 2, `libvrmlapi`. `libvrmlapi`'s design and implementation were inspired by that of Open Inventor [16], though no headers were copied during the construction of the library; it was reimplemented based on several years' development experience with Open Inventor. Inventor's naming conventions for common methods (i.e., `getValue`, `setValue`, `isOfType`) have been followed wherever possible. In the following discussion, a *node* is an object which can be contained in the scene graph; nodes contain *fields*, which are distinct from C++ fields in that they are objects as well, allowing, for example, field-to-field connections within the scene graph. See [16] for a more detailed discussion of Open Inventor's programming paradigms.

B.1 Notable missing functionality

It is important to recognize that `libvrmlapi` only implements an API for reading, writing, and modifying a subset of VRML 2; it does not implement, most importantly, rendering. In order to render standard VRML 2 files using `libvrmlapi` it is necessary to either implement a render action using, for example, OpenGL, or write a conversion action which traverses `libvrmlapi`'s scene graph and creates a corresponding one for an existing rendering library like Cosmo3D or Fahrenheit. `libvrmlapi` itself is self-contained and graphics system-independent, which is one reason why it was relatively easy to port IMPS from an Inventor-based renderer to one which is OpenGL-based.

Notable unimplemented features in `libvrmlapi` are field-to-field connections, paths, VRML 2's distinction between eventIns, eventOuts, and exposedFields, the event mecha-

nism, notification, and most of the node types, including TimerSensors. VRML 2 nodes which are implemented contain no logic, merely storage for data to be read in from the file. Actions are very incomplete and do not implement Inventor's ref/unref semantics upon visiting a node. However, the functionality which is implemented is sufficient to read in files from 3D Studio Max's VRML 2 exporter.

B.2 Implementation Details

There are three primary pieces of functionality required when writing a library for reading and writing VRML-style files. The first is a run-time type system to allow, for example, instantiation of a class by class name; this is used when a node's type is read from the file as a string (i.e., `IndexedFaceSet`) and an object of the appropriate type must be constructed. The second is a mechanism for obtaining a pointer to a field of a node by specifying the field's name (`getFieldByName`). The third is a reference counting mechanism to allow proper deallocation of nodes which have multiple parents pointing to them.

C++ implements "run time type information" (RTTI) via, for example, the `typeid` operator. However, it is not possible to implement the required "instantiate by class name" functionality using C++'s RTTI. At minimum a hash table is needed which maps strings to function pointers returning a new object of the appropriate type. `libvrmlapi`'s `VrmlType` class implements this as well as the tree data structures needed to model a class hierarchy. The run time type system `VrmlType` implements is similar to RTTI but does not support multiple inheritance for reasons discussed in `VrmlType.h`. This implies that **the entire `libvrmlapi` library, as well as all subclasses, may only use single inheritance.**

`libvrmlapi` implements `getFieldByName` functionality by registering each field of a given class at run time in a hash table mapping strings to offsets which are added to the base object's address to obtain pointers to member data. To avoid unnecessary duplication of this string-to-offset hash table (`VrmlFieldData`) in each node, it is made static in each node's class and filled in the first time the class is instantiated.

The `VrmlNode` base class implements Inventor-style `ref`, `unref`, and `unrefNoDelete` methods, and makes the destructors for nodes protected, so they can not be called directly by the user. As in Inventor, nodes are deleted when their reference counts are *decremented* to zero. Nodes have a zero reference count when they are created; a node's reference count

is incremented when it is added to a `VrmlSFNode` or `VrmlMFNode` field, and decremented when it is removed from the field. As discussed in Section B.1, Inventor-style ref/unref semantics are not implemented for `VrmlActions`.

B.3 Developing New Node Types

When using `libvrmlapi` to develop new node types, the above information is not, strictly speaking, necessary. All of the library's run time functionality is incorporated into new node classes via the use of macros.

B.3.1 The Header File

The first step when creating a new node type is deciding from which parent class to inherit. IMPS derives its own base class, `ImNode`, from `libvrmlapi`'s, `VrmlNode`, but strictly speaking this was not necessary (the "dirty flags" implemented by `ImNode` were not used in IMPS.)

The first item in the new class's definition should be the `VRML_NODE_HEADER` or `VRML_NODE_ABSTRACT_HEADER` macro. Each of these takes the class name as its argument. The "abstract" version indicates that this type may not be instantiated from a VRML file, and should be used for any "abstract base classes", whether they have pure virtual functions (`ImShape`) or not (`ImNode`). Regardless of whether the class is abstract, it must declare exactly one constructor, taking no arguments.

The fields of the class which may be read from and written to a VRML file are simply data members which are subclasses of `VrmlField`; see `VrmlIndexedFaceSet.h` for an example.

B.3.2 Method Definitions

In the new class's `.cpp` file, outside of any method's scope, the `VRML_NODE_SOURCE` or `VRML_NODE_ABSTRACT_SOURCE` macro must be added, which inserts the definitions of implementation-level methods declared by the `VRML_NODE_HEADER` macro. The non-abstract version of this macro takes three arguments: the name of the class, the string by which it will be called in the VRML file, and the name of its parent class. The abstract version takes only two arguments, the name of this class and the name of its parent.

The `VRML_NODE_CONSTRUCTOR` macro must be the first line of the class's constructor. After this macro, all of the VRML fields from the class's definition must be listed using either the `VRML_NODE_ADD_SFIELD` or `VRML_NODE_ADD_MFIELD` macro. These give the fields default values (multiple-valued fields default to being empty) and add them to the `VrmlFieldData` structure so they can be found by the `getFieldByName` method when a VRML file is loaded. See `VrmlIndexedFaceSet` for an example of adding many different types of fields to a node.

B.3.3 Useful Methods

`libvrmlapi` declares two virtual functions in the `VrmlNode` base class which may be overridden by subclasses. The first, `notifyFieldChanged`, is called whenever `setValue` is called on a field contained within this node; the argument is the pointer to the field which changed. It is crucial that the overriding method call the parent's version after it has done its subclass-specific tasks (as discussed in `VrmlNode.h`). This form of "notification" propagates up to the root of the scene graph, and currently can not be disabled. It also does not support loops in the scene graph structure.

The second virtual function, `notifyReadIn`, is useful for determining when this node and all of its children have just been loaded from a VRML file. For example, `ImTesselator3MP` overrides this to build necessary caches and to initialize worker threads. Strictly speaking, the overriding method should call the parent's version, but `VrmlNode`'s implementation does nothing, so it is (currently) safe to skip this step.

A new class may also specify a destructor, which will become virtual because of `VrmlNode`'s declaration of its destructor as virtual. When the node's reference count is decremented to zero then it will be deleted and the destructor will be called. To avoid accidental bypassing of the reference counting mechanism, it is strongly recommended that all subclass's destructors be made protected.

B.4 Actions

`libvrmlapi` implements a rudimentary `VrmlAction` class which is sufficient to obtain a callback for nodes of a specific type. However, it does not automatically descend into nodes as an Inventor-style action would, because in VRML 2, children nodes are contained in fields as opposed to being children of parent nodes. This is a discrepancy which should probably

be fixed, or at least the option given to descend into nodes by default.

`VrmlEchoAction` provides a simple example which prints all top-level nodes (i.e., `Groups` and `Shapes`) in a given VRML file. As mentioned above, it does not descend into more specialized node types' fields like the `material` node of an `Appearance` node.

B.5 Building an Application

In similar fashion to Open Inventor, `libvrmlapi` is initialized by a call to `VrmlDB::init` at the beginning of the program. IMPS is initialized after this by a call to `ImSys::init`.

VRML files are loaded by calling `VrmlDB::readAll`. The second argument, returned to the caller, is a list of the routes found within the file, and may not be NULL. This interim solution is in place primarily because field-to-field connections are not implemented, but also because SCOOT's animation system requires the list of routes to process animation files properly.

Since `libvrmlapi` supplies only reading and writing functionality, it does not require control of the application's main loop.

`GLImplViewer.cpp` and `IvImplViewer.cpp` contain the main loops for IMPS' OpenGL and Open Inventor-based viewers and interactive editors, respectively. Auxiliary files assist in constructing manipulators for implicit primitives and implementing 3D painting functionality.

B.6 Known Problems and Porting Notes

The structure of the file format and implementation of the texture mapping algorithms led to the consequence that the system functions best when there is one implicit primitive per RigidTransform. Siblings under a transform blend, but can not be articulated by the transform. In addition, it would be advantageous to be able to specify one texture coordinate generator and associated texture map for multiple implicit primitives, to allow more freedom in the kinds of shapes which can be textured.

Neither of the viewers has an integrated user interface. FLTK was integrated relatively late in the development process, but can be used for user interface development while maintaining cross-platform compatibility between Unix and NT.

`ImTexture.cpp` references Silicon Graphics' Image Format Library (IFL). With SGI's re-

cent port of IFL to Windows (`http://www.sgi.com/Technology/ImageVision/windows.html`), this code is now portable between Irix and NT but not to other platforms, as IFL's source code is not available.

`ImTesselator3MP.cpp` uses Irix's atomic `test_then_add` to implement its own semaphores; these are not portable across platforms. This class also uses Irix's system calls to attempt to lock down worker threads to CPUs; these are also non-portable.

The modalities of interaction (that is, one manipulator per transform in a hierarchical scene graph, but two manipulators per line segment in a flat scene graph, as discussed in Section A.2.2) are confusing and should be rethought or extended and control options given in a user interface.