THE UNIVERSITY OF SHEFFIELD

Implicit Surfaces for Modelling and Character Animation

by

Agata Opalach

THESIS SUBMITTED TO THE FACULTY OF PURE SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

JULY 1996

Implicit Surfaces for Modelling and Character Animation

Agata Opalach

Abstract

This thesis presents the results of research into the use of implicit surfaces for computer generated character animation. The ABC approach is developed in order to maintain Appearance, to prevent unwanted Blending and to preserve Coherence for a character. In particular, the approach addresses the issues of surface deformation and formulates and tests a novel solution to precise contact modelling during collisions and self-collisions. Clay-like capabilities are introduced into the technique through coherence preserving distance constraints.

Inspired by the choreographic nature of the process of animation, a dance notation is proposed as a motion control method for a layered model. It is combined with a procedural motion generation to provide hybrid control over animation and to introduce automatic tunable traditional animation effects to computer generated animation sequences. This new control method incorporates the need for a balance and interaction between high and low levels of abstraction in character motion control.

Several animation examples are presented in support of the thesis. They show that this research has been successful in adapting the natural, visually pleasing, "organic" look of implicit surfaces to the task of *animating* or breathing life into simple expressive characters. The limitations of interactive display and taxturing of implicit surfaces are considered to be the main disadvantage of the technique that stops them from being applied in character animation systems. Some suggestions for improving the rendering process are given. Several extensions and applications of the techniques developed in this thesis are proposed and the role of implicit surfaces in character animation is discussed.

Acknowledgements

It is a pleasure to thank the people whose influence and assistance have been invaluable for the development of this research. The computer graphics community is extremely welcoming and supportive. I have enjoyed enlightening discussions with many individuals whose experience, which they have generously shared, has been an unlimited inspiration for the final shape of this work. Many thanks go to Jules Bloomenthal for his many suggestions and for providing me with more insight into the area of implicit surfaces. I express my gratitude to Brian Wyvill for his constant enthusiasm and help with seeing things in a broader context. I am indebted to Marie-Paule Gascuel whom I met at the beginning of my encounter with implicit surfaces. Our productive discussions and fruitful collaborations have been invaluable.

While at the University of Sheffield, I very much enjoyed my time within the welcoming academic community. My special gratefulness goes to all in the Department of Computer Science. They have provided me with perfect working conditions and friendly atmosphere without which my work would not have been as satisfying. Sharing the ups and downs of computer graphics research with members of the Graphics Group has been an exceptionally excellent experience. I sincerely compliment Alan Watt and Steve Maddock for proficiently leading the group and for making it such a pleasant working environment. The interdisciplinary discussions and exchange of stimulating ideas with members of other research groups have been rewarding. I owe the support team a very special acknowledgment for their rescues with just about anything to do with software or hardware, in particular Simon Addy for his infinite patience and help.

I thank Claude Puech and everyone at *i*MAGIS in Grenoble for their continuous support and understanding. Without their reassurance and encouragement the final stages of the write-up would have been hellish.

Very special thanks are due to my supervisor, Steve Maddock, who suggested the initial concept and supported it with constant help and inspiring comments throughout its development into this thesis. He has generously shared his time and his ideas.

Finally, I thank all the people who have contributed to the creation of the homely feeling which has accompanied me during all this time. I would like to thank my family for preparing solid foundations for all the homes that I will ever be able to sculpt for myself. I thank everybody at Sorby Hall in Sheffield for helping me to find my place in a foreign country. I am grateful to the dancers of Running Amok for the experience of expressing my feelings through dance. I would also like to mention all my friends in many countries around the world who have given me their continuous support.

Rodzicom (For my parents)

Why character animation? Why any visual means of expression? For me, a very important part of life takes place at the non-verbal level, in the world of feelings and emotions. Some people can translate these into words. For others, the only way to express them is through the visual channel. The visual expression has no language constraints, it is universal, personal and fulfilling. A dance, a sculpture or a little wire tree can sometime say more with the visual metaphore given them by their creator than the most detailed of verbal descriptions. Then again, it is my very personal theory of life.

Contents

List of Figures

Chapter 1

Introduction

1.1 The thesis

During the process of modelling, a computer representation of an object is created. This representation should be chosen to fit the requirements of a particular application. The choice of a suitable modelling method is usually made by taking into consideration the desired static and dynamic properties of the model. For a static situation, the sought qualities include the accuracy of the model, its similarity to the modelled object in the case of real objects or its appeal in the case of artistic creations (*e.g.* virtual sculptures or characters in animation). For a dynamic situation, the questions to be answered are concerned with actions that are likely to be performed on a model: will it deform during its life, does it have to respond to physical laws and simulate the real world or does the animator need to have total control over its motion to achieve special effects.

The area of interest in this work is computer generated character animation. This thesis argues that implicit surfaces can offer useful contributions in this task. The strengths and weaknesses of the technique are discussed and its robustness in the task of modelling 3D "clay"-like characters is improved. Aspects of animation control is also considered with the goal of achieving automatic traditional animation effects in animation sequences. A balance between manual motion control and automatic motion generation for a layered character model is provided.

One of the most popular modelling primitives in computer graphics is a polygon. A polygonal model is created by approximating the surface of an object with a number of polygons; it is therefore called a *boundary representation*. If a softly blended shape is to be modelled, a polygonal model may suffer from the jagged silhouette problem that appears when the size of the polygons is too large. To solve the silhouette problem, a polygon can be replaced by a parametric patch which is a smooth curved surface that may be used to model "rounded" objects. However, a polygon or a parametric patch are low level entities; thus, manually specifying them is a tedious and time consuming task. Typically an intermediate modelling technique is used to generate a boundary representation for an object. In the case of modelling organic life forms one of the techniques available to produce a surface representation is skeletal design [Bloo95b]. A skeleton of an object is defined and a surface is fitted around it. For instance, for a 3D curve the surface can be derived by sweeping a volume cross section along the curve (see *e.g.* [Watt93]) or by using generalised cylinders [Agin72].

Problems also occur when trying to model and animate deformable objects using the boundary representation. During deformations, the surface has to be modified. Thus, the continuity of the mesh has to be ensured and sometimes its recreation may be required. Also, detecting interpenetration between objects, a process essential in animation, is an expensive operation since polygon-polygon or patch-patch intersections have to be calculated for all surface elements. Nevertheless, the polygonal approach remains very attractive in computer graphics since it allows for fast hardware supported rendering of the model.

An alternative approach is to model objects using a *volume representation*, in which the notion of inside/outside the object is known. An example of such a technique is to use a combination of mathematical equations to describe an object. For instance, a sphere with radius r, centred at the origin can be accurately modelled using its equation $f(x, y, z) = x^2 + y^2 + z^2 - r^2$. A point P lies inside the object if the value

of this function at P is negative, outside the object if it is positive and on the surface if it is zero. Since the surface of the object is specified indirectly as all points with the given inside/outside function equal to zero, rendering of a model is more costly as it requires solving an implicit equation $f(x, y, z) = 0$. However, in this method it is easier to detect interpenetration between objects.

A surface defined in this indirect way is called an *implicit surface*. Implicit surfaces have been used to create smooth blends between solids for many years [Wood87] and have been successfully utilised in constructive solid geometry (CSG) applications. With his method for visualising electron density fields using implicit equations, Blinn [Blin82] opened a new path for implicit surfaces - as a self-contained modelling technique. The visualisation in his work was performed using a cloud of points in space with the exponential that defined a scalar field around each point as a function of the distance from that point. Extracting an isosurface of equal values from the sum of all fields resulted in a smooth "*blobby*" surface blended around the cloud of points. More work was soon undertaken in this direction. In Japan, *metaballs* were developed by Nishimura *et al.* [Nish85]. At the same time, in Canada Wyvill *et al.* [Wyvi86b] introduced *soft objects*. To built their models, both latter techniques used implicitly defined spheres: isosurfaces of equal value in a scalar field defined by a polynomial function around a point. All three pioneer works [Blin82, Nish85, Wyvi86b] also proposed variations and extensions to the basic method (*e.g.* negative or ellipsoidal primitives). A further development of the technique introduced geometric skeletons (*e.g.* a line segment, a curve, a polygon or a solid) as primitives that generate the scalar field [Bloo90b, Bloo91]. Since a point is a special case of any of these skeletons, this was in fact a generalisation of the method.

The remainder of this thesis focusses on *skeleton-based implicit surfaces*, calling them *implicit surfaces* for simplicity. Spherical primitives, described by the following scalar field function, will be used:

$$f(r) = c\left(1 - \frac{r^2}{R^2}\right)^2$$

where c is the strength of a primitive, r is the distance from the given point to the centre of the primitive and R is the radius of influence of the primitive. It is the field function implemented in POV-ray [POV 93], a raytracing environment used and extended for visualisation of implicit surfaces (Appendix A). The algorithms to be proposed will only be tested for spherical primitives. It will be ensured that they are readily generalised to more complex skeletons by making as few restrictive assumptions about the scalar field calculation as possible.

The main goal for traditional animators is to *animate* the characters that they create. The final animation sequence should give an impression that the characters are independent from their author, can make their own decisions and live their own lives. Throughout the years artists have been developing techniques to achieve these goals. One of the main centres of development was the Disney Studio [Thom81]. There, a team of artists studied all aspects of animated life before attempting to create it. They were redrawing their sketches until the desired emotion was apparent to the audience. One of the techniques stressed most often was "squash and stretch". Characters showed their reactions and feelings by stretching and squashing their faces and their entire bodies. All movements were fluid, all physical laws exaggerated and each action was preceded by its anticipation to prepare the audience for it. These are the main principles of Disney animation. They make us forget that this is only the illusion of life.

In computer animation, the most popular approach to character animation is using a layered model [Chad89]. A character is modelled as an articulated rigid skeleton and covered with a layer of deformable skin. An animation sequence is specified for the skeleton and the corresponding deformation of the skin is calculated by an animation system. One problem with such a layered approach is that only local deformations of the character's skin are modelled. Therefore, the traditional animation principles, in particular squash and stretch, cannot be easily achieved because of the rigidity of the underlying skeleton. There is a need for deformable fluid models in computer generated character animation. Considering the natural clay-like properties of implicit surfaces and the automatically fitted "skin" (isosurface) around primitives, developing an alternative, more flexible layered model using them is appealing in this context.

A model created using implicit surfaces is built of a set of unstructured primitives. Thus, when it is animated, it may undergo deformations that destroy the look intended for a character. For instance, some primitives may get close to each other and blend some characteristic parts of a character (*e.g.* a leg and an arm or fingers in a hand) which is an undesired effect. Moreover, the primitives may move away from each other and the body which was sculpted for the character will break into pieces losing its

integrity. Therefore, the first problem this thesis will look into in order to use implicit surfaces for character animation, is creating a way of preserving the personality and appeal of a character throughout its life, particularly during deformations.

Another problem arises when controlling an animation sequence that uses characters built with implicit surfaces. On one hand, care has to be taken not to introduce undesired blending or coherence loss. On the other hand however, tedious manual motion specification in order to avoid any undesired effects is unacceptable. A balance between high level motion specification and low level motion adjustment is required. In addition, for complex models, generating parts of motion automatically should be considered. For a layered model, a natural approach is to specify the motion for the inner layer and automatically generate the corresponding motion for the outer layer. Applying this principle to implicit surfaces with the aim of adapting their properties to the task of creating automatic traditional animation effects will be explored in this thesis.

1.2 Overview

This thesis is organised into five chapters. After presenting the principles of implicit surfaces, the aspects of character modelling and animation are presented, followed by a more general discussion which puts the achievements of this work in a broader context and proposes some future research directions.

1.2.1 Principles

Since terminologies used by the researchers of implicit surfaces differ, Chapter 2 presents a uniform terminology to be used throughout this thesis and classifies existing approaches in these terms. At the end of this chapter, the reader should have a clear image of the development of implicit surfaces as a modelling technique and their potential applications.

1.2.2 Modelling

For deformable models, their dynamic behaviour needs to be considered at the time of modelling to make sure that the required surface changes are achievable with the technique. This is reflected in the contents of Chapter 3 which discusses the aspects of modelling deformable characters using implicit surfaces and considers their possible motion. The three main requirements identified for this task are: the need for defining and maintaining the appearance for a character, avoiding unwanted blending between the character's parts and preserving the coherence of the character. Solutions developed to them form the parts of an extension to the implicit surfaces modelling technique, the ABC approach, proposed to support character modelling and animation using them. The ABC approach was initially proposed in [Opal93b].

The extended implicit model builds characters using an appearance graph which defines the general look of the character as an articulated skeleton. This skeleton is not rigid, its parts may undergo deformation, *e.g.* squash and stretch. Based on the appearance graph for a model, the blending properties between the character's parts are obtained. Finally, to preserve the coherence of a character, distance constraints are imposed on the primitives.

Chapter 3 will detail these algorithms and present examples of animation sequences created using this extended model. The details of enhancing a standard raytracer, POVray [POV 93], to deal with the ABC approach can be found in Appendix A. At the end of Chapter 3 the reader will understand the problems encountered during modelling and animating deformable characters using implicit surfaces and appreciate the need for a further development of the technique in order to *animate* the characters in the traditional animation sense. Chapter 5 will discuss some of these aspects in more detail providing suggestions for extensions and possible future directions for the use of the ABC approach.

1.2.3 Animation

Chapter 4 will use and extend the findings of the ABC approach to create traditional animation effects in computer generated animations. A layer of automatically generated flesh will be added to the layered model. The issues of high and low level control will then be discussed for such a model. An animator will

be provided with a hybrid control method, in which the inner skeleton and the outer layer in the model will be animated using separate animation techniques. A balance between manual and automatic motion control will be established.

For the animation of the inner skeleton, the inspiration will be taken from the world of dance. Choreographing a dance piece and creating an animation sequence have a lot in common. They start from an abstract concept to be conveyed through the visual channel. Then each is planned as a "storyboard" and progressively refined until all required expressions are included in the motion of performers, dancers or characters. Appreciating this link between the two art forms, a dance notation will be used to specify the motion of the inner skeleton.

Such motion control has a relatively low level of abstraction since it requires the specification of the angles between the links in the joints of the articulation structure. Although the dance notation gives an intuitive support in this task, it is required to make sure that the motion of the outer layer does not need to be given manually. Therefore, an algorithm will proposed that will generate the motion of the outer layer based on the movement of the inner skeleton. The animator will be given control over such automatic generation in terms of a set of adjustable parameters. High level of control over these parameters will be offered in terms of heuristics translating them into traditional animation effects. Thus, an animator will be able to increase or decrease the amount of each of the chosen "Disney" effects. Some initial results of this part of this research were published in [Opal94, Opal95b]. Some issues related to faster display of implicit surface during interactive design or animation will be given in Appendix B.

At the end of this chapter the reader will appreciate the complexity of creating expressive animations and understand the advantages of using a layered model based on implicit surfaces for character animation. More advantages and disadvantages of the layered approach will be discussed in Chapter 5.

1.2.4 Contributions

This thesis proposes a new presentation and analysis of the implicit surfaces modelling technique for character modelling and animation using a layered model. It formulates and tests a new solution to the problem of unwanted blending, thus addressing the issues of surface deformation and contact modelling during collisions and self-collisions. A new approach to coherence preservation is detailed which offers clay-like modelling capabilities. Inspired by the similarities between the processes of dance choreography and animation, a dance notation is used for parts of motion specification for the layered model. A new hybrid animation control for automatic generation of traditional animation effects is developed. Finally, some work on the display techniques for implicit surfaces is presented: an extension to a standard raytracer to incorporate the ABC approach is detailed, an the possibility of using a secondary data structure for faster implicit surface polygonisation is considered.

Chapter 2

Principles of Implicit Surfaces

2.1 Introduction

The main focus of this thesis is on implicit surfaces as a modelling technique. Its evolution started from Blinn's method of modelling electron density fields which used a set of implicitly defined primitives [Blin82]. This chapter will present the area attempting to classify existing approaches. Since the terminology used by researchers in the field often differs, a set of definitions will be established and used throughout this thesis.

The main concept behind the technique is combining the functions used to define the primitives to create a final implicit equation that represents an object. A very general description of an implicit surface is $\{P \in \mathcal{R}^3 : \mathcal{F}(\mathcal{P}) = 1\}$, where F is a real function, $F : \mathcal{R}^3 \rightarrow \mathcal{R}$. A wide variety of objects, presented in this chapter, can be described in this way.

An analogy often used to describe implicit surfaces is a comparison to heat. Let us imagine a source of heat in space. All points with temperature equal to a certain value create a surface in space which is called an isosurface. Multiple sources will create a temperature distribution that is higher in the space contained between them. Thus, for any value of temperature, an isosurface will merge the two respective temperature surfaces if the sources are close enough (Figure 2.1).

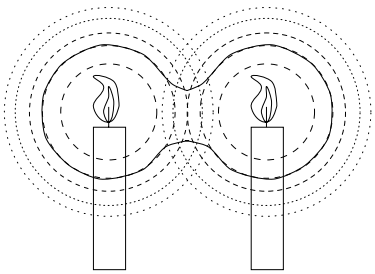


Figure 2.1: *An isosurface of equal temperature around two heat sources (solid line)*

Definitions of elements of an implicit surface modelling environment will be introduced in Section 2.2. A variety of modelling primitives will be presented in Section 2.3. Section 2.4 will discuss ways of combining such primitives. A review of a wide variety of applications to which implicit surfaces have been beneficial will be given in Section 2.5. The chapter will conclude with some general reflections on the potentials of implicit surfaces and with assumptions made for the remainder of this thesis (Section 2.6).

2.2 Terminology of implicit surfaces

This work focusses on *skeleton-based implicit surfaces*, calling them *implicit surfaces* for simplicity. The term *general implicit surfaces* will be used when referring to a superset of implicit surfaces defined generally as $F(P) = 0$, where $F(P)$ is a function $F : \mathcal{R}^3 \rightarrow \mathcal{R}$.

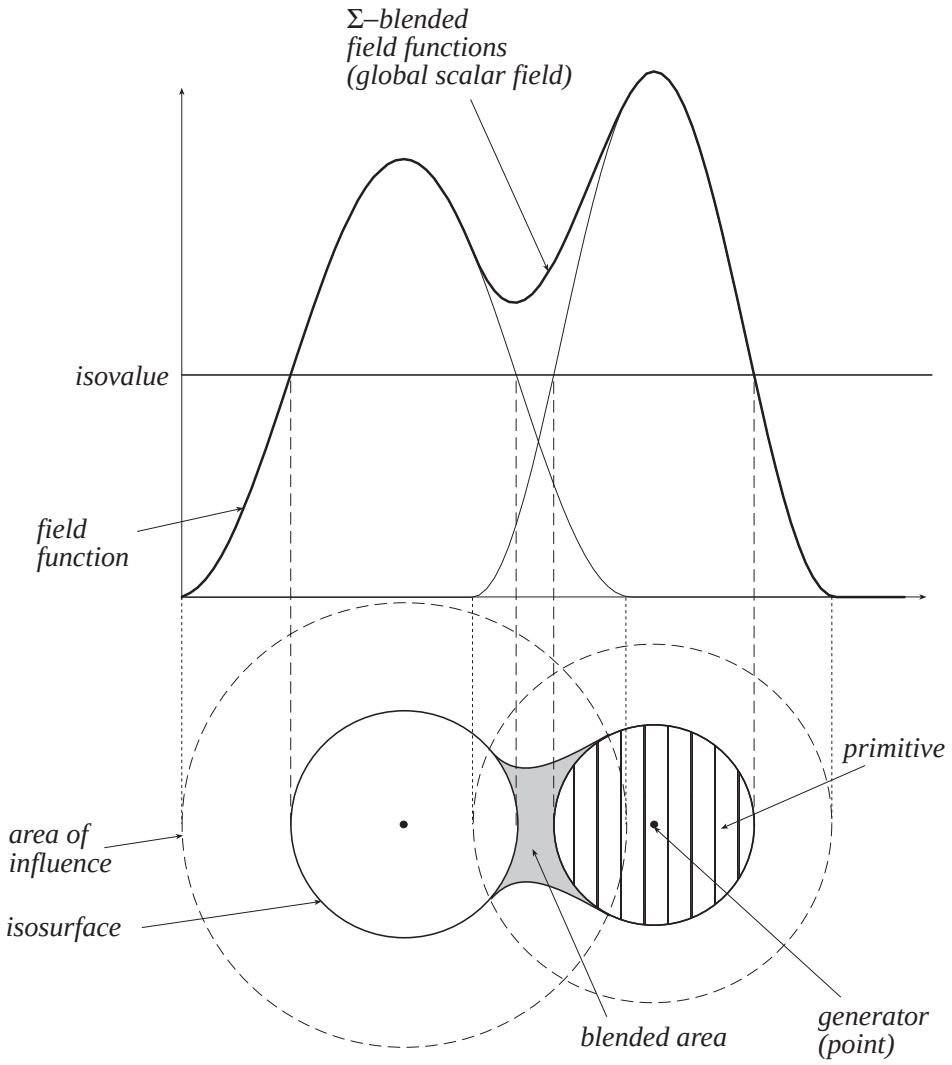


Figure 2.2: *Basic elements of implicit surfaces*

Figure 2.2 illustrates the elementary concepts of modelling with implicit surfaces. A model consists of *primitives* combined with each other using a *blending method*. Each primitive is specified by a *generator* (skeletal element) which gives the geometric skeleton of a primitive, an *area of influence* which limits the area affected by the primitive and a *field function* which defines a local scalar field around the generator. For any given point in space the global *scalar field* value is calculated by considering local scalar field values at this point contributed from all primitives influencing it and combining these values using the blending method. The surface of a model (*isosurface* or implicit surface) is defined as all points with the global scalar field value equal to a given *isovalue*. The isosurface divides the space into the inside and the outside of an *isovolume* or implicit volume. More detailed definitions for each of these key terms will now be given.

Generator (skeletal element) is a geometric object G for which a distance function $d(G, P)$ can be defined for any point $P \in \mathcal{R}^\ni$. Examples include a point C with the Euclidean distance function $d(C, P) = \|\overrightarrow{CP}\|$ or a line segment $\overline{AB}$ with the distance $d(\overline{AB}, P) = \|\overrightarrow{PP_0}\|$ where P_0 is the point of $\overline{AB}$ closest to P, more formally:

$$\forall_{P' \in \overline{AB}} \|\overrightarrow{P'P}\| \text{ ¿ } \|\overrightarrow{PP_0}\|$$

Area of influence can be given by any bounding volume, *e.g.* a bounding sphere, a bounding box, a bounding cylinder etc. For point-generators bounded by a sphere, the term *radius of influence* is often used.

The influence of a primitive outside its area of influence is set to zero. It reduces the cost of scalar field value calculation since primitives distant from a given point do not have to be considered.

Field function describes how the local scalar field is shaped around a generator. For a given point P it returns a scalar value which depends on the distance from P to the generator. The field function can therefore be decomposed into two functions, a *potential function* $f : \mathcal{R} \rightarrow \mathcal{R}$ which describes the way the scalar field changes depending on the distance from the generator and a *distance function* $d : \mathcal{R}^\ni \rightarrow \mathcal{R}$ which calculates the distance. The scalar field value $F(P)$ is then calculated as $f(d(P))$. In the following sections we will normally only describe the potential function f assuming its argument is the Euclidean distance:

$$d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

where $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$. An example potential function is:

$$f(r) = \left(1 - \frac{r^2}{R^2}\right)^2$$

where r is the distance and R is the radius of influence. For simplicity of notation it will also be written:

$$f(P) = \left(1 - \frac{r^2}{R^2}\right)^2$$

assuming that r is the Euclidean distance from the point P to the generator. The use of an alternative distance function will be noted explicitly.

Primitive is described by three elements: a generator, an area of influence and a field function.

Blending method specifies a way of combining multiple primitives to obtain a global scalar field. The most common and simplest blending method is the summation, $\sum_i F_i$, of values of all field functions F_i.

Scalar field (global scalar field) is a result of a function which determines the combined influence of all primitives in space. This function is derived from the given set of primitives and the blending method. It has to be defined for all points $P \in \mathcal{R}^\ni$. If summation is used as the blending method, the scalar field is calculated as:

$$F(P) = \sum_{i \in Influencing(P)} F_i(P)$$

where $F_i(P)$ is the field function of i-th primitive.

Isosurface consists of all points with the scalar field value equal to the given isovalue $Iso \in \mathcal{R}$, more formally it is $\{P \in \mathcal{R}^\ni : \mathcal{F}(\mathcal{P}) = \mathcal{I}\int\wr\}$

Isovolume is a set of all points with the scalar field value greater than or equal to the given isovalue $Iso \in \mathcal{R}$, more formally $\{P \in \mathcal{R}^\ni : \mathcal{F}(\mathcal{P}) \text{ ¿ } \mathcal{I}\int\wr\}$.

Inside/outside function: A point with the scalar field value greater than Iso lies inside the isovolume, a point with the scalar field value equal to Iso lies on the isosurface and a point with the scalar field value less than Iso lies outside the isovolume.

2.3 Primitives

This section will classify various types of primitives, used in the area of implicit surfaces, into three groups: skeletal primitives, algebraic primitives and empirical primitives. The first group describes the foundations of skeleton-based implicit surfaces. The second group shows how general algebraic equations can be used for implicit modelling. The last group includes models directly given as a discrete scalar field, *e.g.* medical scans. The implicit formulation allows for all three groups to coexist and interact with each other in one modelling system.

2.3.1 Skeletal primitives

Distance primitives, described in this section, offer this approach to modelling in a natural manner: a set of generators forms a skeleton of an object which is then smoothly, intuitively and automatically clothed with the isosurface. The skeletal primitives are divided into three groups based on the distance function used: Euclidean distance, anisotropic distance and convolutions.

Euclidean distance

The Euclidean distance between two points $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$ in the Euclidean space $\mathcal{R}^3$ is given by:

$$d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

An Euclidean distance surface is defined as an offset surface from a geometric generator G calculated using the Euclidean distance d:

$$\{P : f(d(P, G)) = Iso\}, \quad Iso \in \mathcal{R}$$

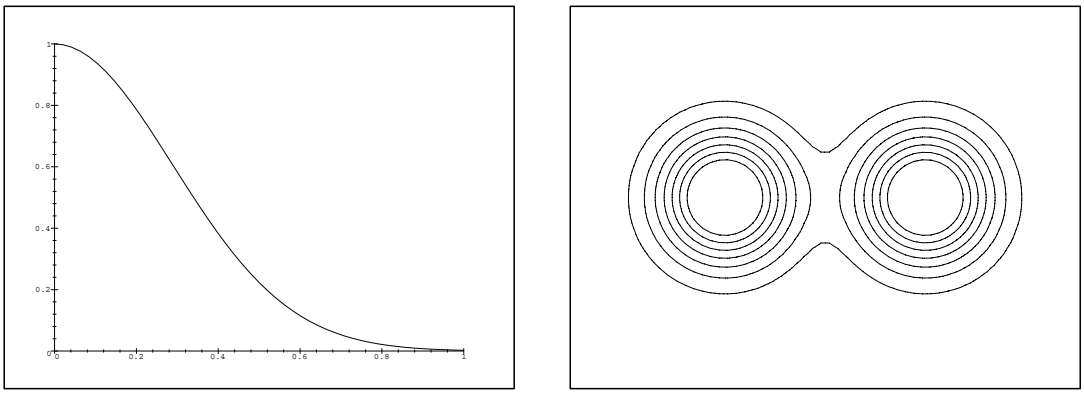


Figure 2.3: *Left: Blinn's potential function,* $b = 1$, $a = 12$*; Right: isosurfaces created by two point-generators at isovalues (from the outside inwards):* $.2, .3, .4, .5, .6, .7, .8$

Blinn [Blin82] used a Gaussian as the potential function around point-generators:

$$f(P) = be^{-ar^2}$$

based on the behaviour of hydrogen atoms (Figure 2.3). The parameters a and b control the shape of the Gaussian bump that this function describes. They are used to adjust the "blobbiness" of the model. One problem with this definition is that the assumed radius of influence is infinity, therefore all primitives have to be considered to evaluate the scalar field value at any given point. To reduce the computation cost, Blinn truncated his potential function by neglecting the influence of a primitive outside a certain range. This introduces a discontinuity into the potential function what may cause continuity problems when blending the primitives.

Nishimura *et al.* [Nish85] and Wyvill *et al.* [Wyvi86b] directly introduced the radius of influence and ensured the potential function continuity. A further reduction in computation was achieved by using piecewise polynomials, which are computationally cheaper than the exponential.

The *metaball technology* [Nish85] used the following potential function:

$$f(P) = \begin{cases} s(1 - 3(\frac{r}{R})^2) & 0 \leq r \leq \frac{R}{3} \\ \frac{3s}{2}(1 - \frac{r}{R})^2 & \frac{R}{3} \leq r \leq R \\ 0 & R \leq r \end{cases}$$

where R is the radius of influence and s is the weight of the primitive (Figure 2.4).

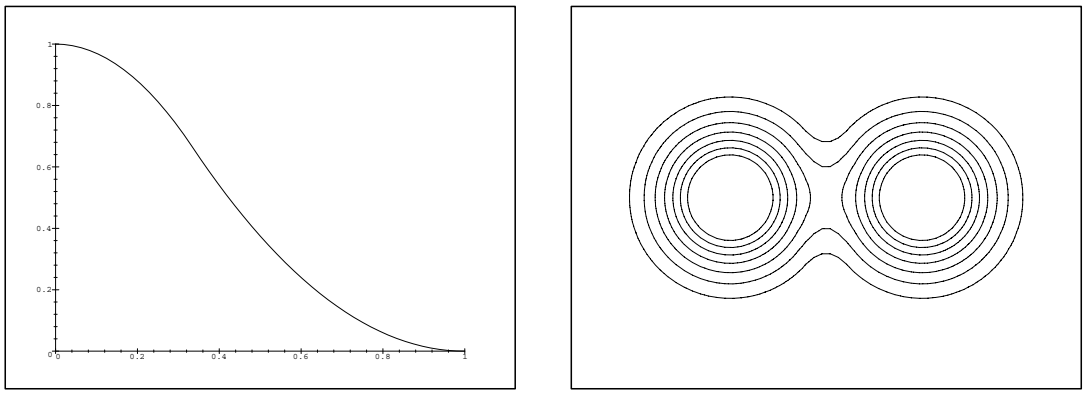


Figure 2.4: *Left: Nishimura* et al.*'s potential function,* $s = 1$*,* $R = 1$*; Right: isosurfaces created by two point-generators at isovalues (from the outside inwards):* $.2, .3, .4, .5, .6, .7, .8$

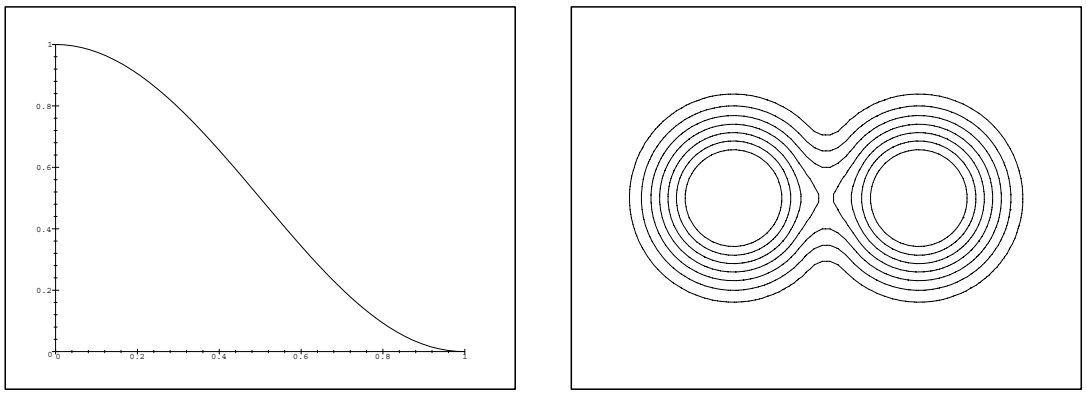


Figure 2.5: *Left: Wyvill* et al.*'s potential function,* $R = 1$*; Right: isosurfaces created by two point-generators at isovalues (from the outside inwards):* $.2, .3, .4, .5, .6, .7, .8$

Soft objects [Wyvi86b] used a different polynomial:

$$f(P) = \begin{cases} -\frac{4}{9}\frac{r^6}{R^6} + \frac{17}{19}\frac{r^4}{R^4} - \frac{22}{9}\frac{r^2}{R^2} & r \ 6 \ R \\ 0 & r \ ¿ \ R \end{cases}$$

where R is the radius of influence (Figure 2.5). This function further speeds up the calculation of the scalar field value by only using the squared distance r^2 and therefore avoiding the expensive operation of square root.

Figure 2.6 compares the potential functions in these three pioneer works. They are similar in shape, thus the visual results achievable with these methods are comparable. However, the Blinn's potential is the most expensive to calculate and changes the model globally. Therefore, the Nishimura *et al.*'s and Wyvill *et al.*'s potentials are more attractive since they are cheaper to compute due to a polynomial approximation of the Gaussian. Moreover, they allow for local modification of a model, which is essential for interactive design purposes.

More potential functions were introduced by Kaći-Alesić and Brian Wyvill [Kaci91]. They argued that the shape of the potential function should be used to control the blending between primitive. For instance, a novel potential functions proposed was the Bernstein-Bézier spline:

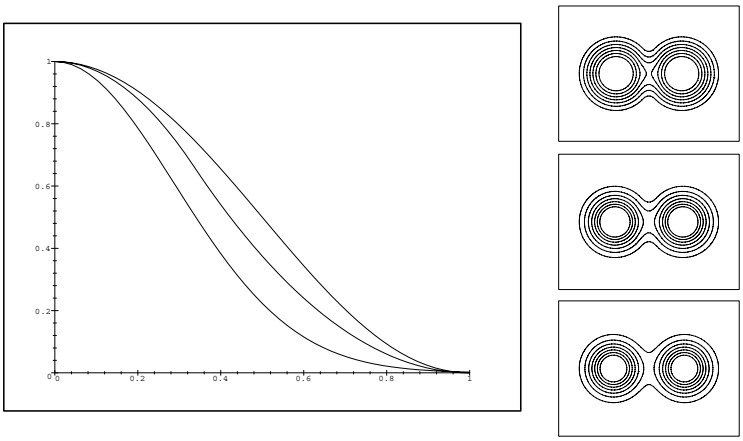


Figure 2.6: *Potential functions: Blinn (bottom), Nishimura* et al. *(middle) and Wyvill* et al. *(top)*

$$F(P) = \sum_{i=0}^{n} b_i B_i^n(\tfrac{r}{R})$$

$$B_i^n(\tfrac{r}{R}) = \binom{n}{i}(\tfrac{r}{R})^i(1-\tfrac{r}{R})^{n-i}$$

It offered a wide variety of blends depending on the chosen spline control points $P_j = (\frac{jR}{n}, b_j), j = 0, \ldots, n$ (Figure 2.7).

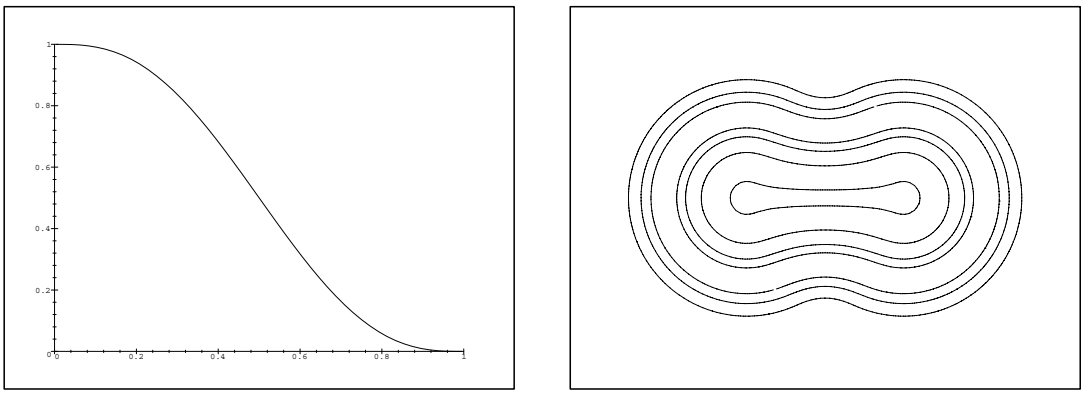


Figure 2.7: *Left: Bézier potential function,* $n = 5$, $R = 1$, $b_1 = b_2 = b_3 = 1$, $b_4 = b_5 = b_6 = 0$*; Right: isosurfaces created by two point generators at isovalues (from the outside inwards):* $.2, .3, .4, .5, .6, .7, .8$

Another potential defined was the arctangent:

$$F(P) = 0.5 - \frac{1}{\pi}\arctan\left(c_1\left(\frac{r}{R} - c_2\right)\right)$$

Blanc and Schlick proposed a potential function based on piecewise rational polynomial:
It blended primitives in a very "hard" way (Figure 2.8)

$$f(P) = \begin{cases} 1 - \frac{(3r^2)^2}{p+(4.5-4p)r^2} & r^2 < \frac{1}{4} \\ \frac{(1-r^2)^2}{0.75-p+(1.5+4p)r^2} & \frac{1}{4}\ 6\ r^2 < 1 \\ 0 & r^2\ ¿\ 1 \end{cases}$$

where r is the distance and p controls the slope of the function $f(P)$ (Figure 2.9). It offers inexpensive calculation of the potential, local influence of primitives and unbounded control of the "hardness" of blending ($p \in \mathcal{R}_+$).

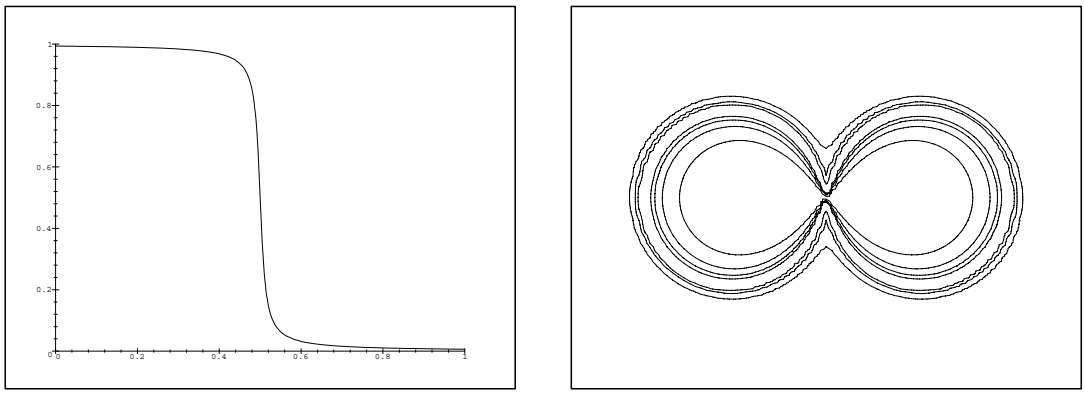


Figure 2.8: *Left: Arctangent potential function,* $R = 1$, $c_1 = 100$, $c_2 = \frac{1}{2}$*; Right: isosurfaces created by two point generators at isovalues (from the outside inwards):* $.2, .3, .5, .7, .9, .95, .99$ *(Note: the jagged effect on the curves is due to Maple rounding errors)*

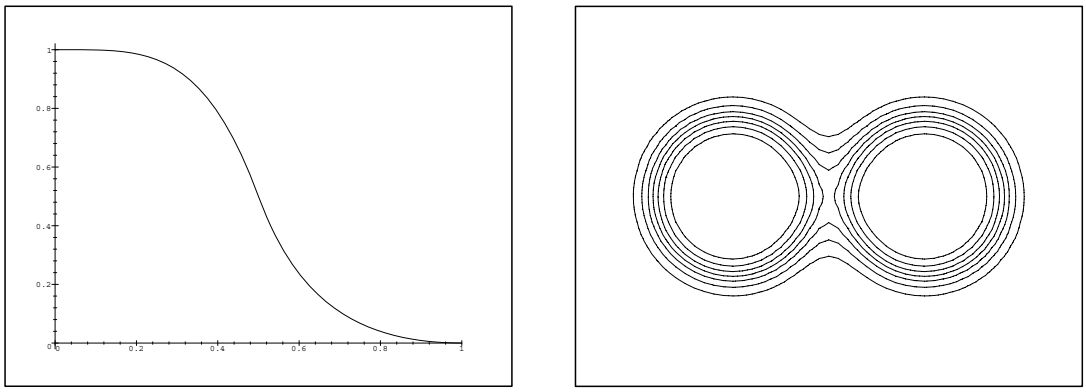


Figure 2.9: *Left: Blanc and Schlick's potential function,* $p = 1$*; Right: isosurfaces created by two point generators at isovalues (from the outside inwards):* $.2, .3, .4, .5, .6, .7, .8$

So far only point-generators were presented. More complex generators were introduced by Bloomenthal to adapt implicit surfaces to more general aspects of skeletal design of natural forms [Bloo90a]. He used the distance from a generator directly as the field function. The potential function is therefore constant:

$$f(P) = r$$

Note that this definition changes the notion of isovolume by having the values greater than isovalue outside it. This does not change the generality of this classification. Bloomenthal described points, line segments, polygons and general curves as generators and provided methods for calculating Euclidean distance for each of them. He used maximum as the blending method and also described a procedural way of combining primitives (see section 2.4.4).

Bloomenthal and Wyvill unified their approaches [Bloo90b] and used the *soft object* potential function to define the offset function around geometric generators, such as points, line segments, curves and polygons. Any other potential function can be used to describe the scalar field around such generators.

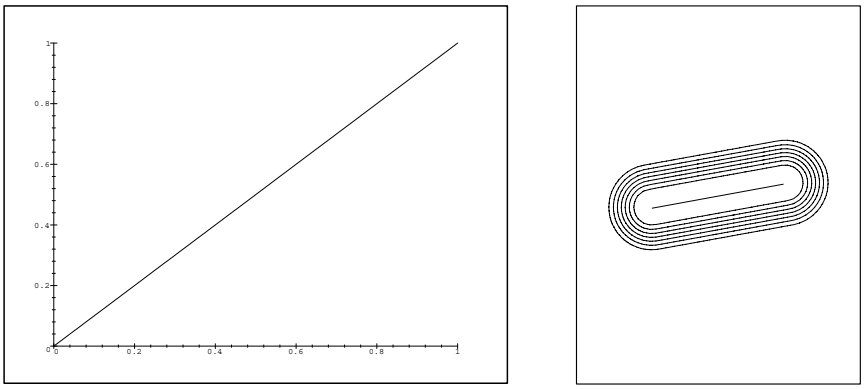


Figure 2.10: *Left: Distance as field function; Right: isosurfaces created by one line segment generator at isovalues (from the outside inwards):* $.2, .3, .4, .5, .6, .7, .8$

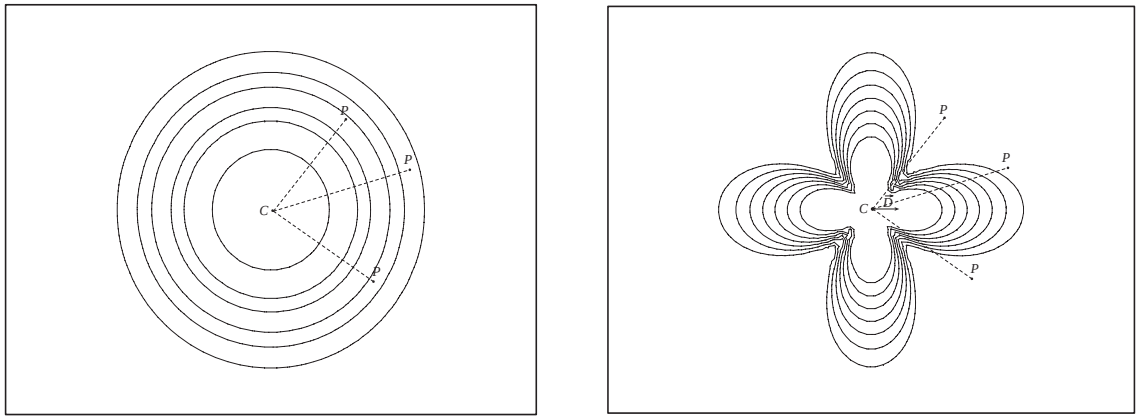


Figure 2.11: *Euclidean (left) and clover (right) distances combined with Blanc and Schlick's potential function*

Anisotropic distance

The field functions described so far were isotropic, *i.e.* they changed in the same way along any line drawn from any point P to the corresponding point P_0 closest on the generator (Figure 2.11). Anisotropic distance functions offer more complex shapes of the scalar field while using simple generators.

Karić-Alesić and Brian Wyvill [Kaci91] described a way of incorporating additional parameters, *e.g.* the angle between the position vector and a given reference vector, to the distance calculation. In this way they modelled a general cylinder with varying radius along a curve or a shell shape around a disc.

Blanc and Schlick [Blan95] proposed an interesting range of field and distance functions defined around point-generators based on piecewise rational polynomials which exhibited both isotropic and anisotropic behaviours. The anisotropic distance is defined by specifying a direction vector $\overrightarrow{D}(a, b, c)$ for a point-generator positioned at $C(x_c, y_c, z_c)$ with the radius of influence R and varying the distance at a point $P(x, y, z)$ depending on the angle between $\overrightarrow{D}$ and $\overrightarrow{CP}$. The following function can be used to define a visually attractive *clover distance* (Figure 2.11):

$$d^2(s,t) = \frac{t^2}{m + (1-m)(2cos^2\theta - 1)^2}$$

where $m \in [0, 1]$, $t^2 = u^2 + v^2 + w^2$, $u = \frac{x-x_c}{R}$, $v = \frac{y-y_c}{R}$, $w = \frac{z-z_c}{R}$, $s = ua + vb + wc$, $cos\theta = \frac{s^2}{t^2}$. This distance can then be used in any potential function. In Figure 2.11 it was applied to Blanc and Schlick's

rational polynomial potential.

Convolutions

In the third group of skeletal primitives, described in the work of Bloomenthal and Shoemake [Bloo91], the offset surface is calculated by considering not only the closest point but all the points on the generator G, using integration:

$$f(P) = \int_{S \in G} e^{\frac{-\|S-P\|^2}{2}} dS$$

The created surface, *convolution*, does not produce "bulges" during primitive blending. For a more complete discussion and a solution to this problem refer to [Bloo95a, Bloo95b].

2.3.2 Algebraic primitives

A wide range of surfaces can be described implicitly using algebraic equations. A simple example is a sphere centred at (x_c, y_c, z_c) with the radius r_s which can be specified as:

$$\{P \in \mathcal{R}^3 : (\S - \S_\rfloor)^\in + (\dagger - \dagger_\rfloor)^\in + (\ddagger - \ddagger_\rfloor)^\in - \nabla_f^\in = \prime\}$$

If a function $f(P) = (x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2$ is defined for a point $P(x, y, z)$, r_s^2 taken as the isovalue and the equation rewritten as:

$$\{P \in \mathcal{R}^3 : \{(\mathcal{P}) = \nabla_f^\in\}$$

the formulation agrees with the implicit paradigm.

A sphere is an example of a skeletal primitive described in Section 2.3.1. However, more general algebraic equations can be used, *e.g.*

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} - 1 = 0$$

which represents a hyperboloid and could also be a primitive in an implicit modelling system. Below, three groups of algebraic primitives are described: (i) quadrics and superquadrics, (ii) generalised implicit functions and (iii) function representation of objects.

Quadrics and superquadrics

A quadric surface is given by the following implicit equation:

$$c_1x^2 + c_2y^2 + c_3z^2 + c_4xy + c_5yz + c_6xz + c_7x + c_8y + c_9z + c_{10} = 0$$

A sphere, a plane, an ellipsoid, a paraboloid or a hyperboloid are sample instances of this equation. Blinn [Blin82] proposed replacing the spherical term $-ar^2$ in his field formulation by a general quadric and thus using any quadric as a primitive. Wyvills [Wyvi89] also replaced r^2 in their cubical field function and this way introduced quadrics into their *soft object* modelling environment. To maintain finite area of influence they focussed mainly on ellipsoids given by the following quadric:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} - 1 = 0$$

A further extension to the variety of primitives is to use superquadrics [Barr81] by setting the exponent in the quadric equation to any $w \in R$. For instance, a superellipsoid is described by:

$$\frac{x^w}{a^w} + \frac{y^w}{b^w} + \frac{z^w}{c^w} - 1 = 0$$

The 2D case of a superellipsoid, a superellipse, was popularised by Piet Hein, a Danish scientist and poet [Gard65]. Figure 2.12 shows several superellipses.

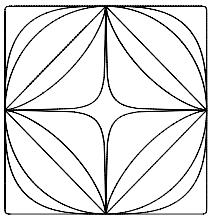


Figure 2.12: *Piet Hein's superellipse: from the centre* $w = 0.3, 0.7, 1, 2, \pi, 100$

The use of superquadrics in his Gaussian potential function was another of Blinn's suggestions [Blin82]. Wyvill and Wyvill [Wyvi89] observed that substituting a superellipsoid formula into their cubic potential function resulted in only local blending between primitives which produced less pleasing results. Thus, they proposed an alternative way of introducing superellipsoids as primitives. They used the following potential function:

$$\frac{R^w cos^w \alpha}{a^w} + \frac{R^w cos^w \beta}{b^w} + \frac{R^w cos^w \gamma}{c^w} - 1 = 0$$

where α, β and γ are angles made by the axes of the superellipsoid and the vector $\overrightarrow{OP}$ between the centre of the ellipsoid O and the given point P and $R = \|\overrightarrow{OQ}\|$ is the distance along $\overrightarrow{OP}$ at which the field drops to zero (Figure 2.13).

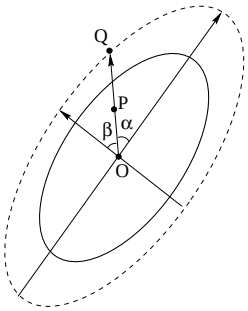


Figure 2.13: *The superellipse of Wyvill and Wyvill*

Generalised implicit functions

Hanson [Hans88] proposed a generalisation of superquadrics, *hyperquadrics*, that defined a wide class of shapes including arbitrary convex polyhedra. A primitive was created by intersecting a hyperquadric with a hyperplane. The resulting lower dimension shape could not be achieved with the use of superquadrics in that dimension. For instance, a triangle could not be constructed using the superellipse equation in 2D:

$$\left|\frac{x}{a}\right|^{w_1} + \left|\frac{y}{b}\right|^{w_2} = 1$$

The family of possible solutions is shown in Figure 2.12. However, intersecting a plane with a 3D superellipsoid given by:

$$\left|\tfrac{x}{a}\right|^{w_1} + \left|\tfrac{y}{b}\right|^{w_2} + \left|\tfrac{z}{c}\right|^{w_3} = 1$$

where $w_1 = w_2 = w_3 = 1$, can produce a triangle (Figure 2.14). This process can be taken into higher dimension and thus introduce a wide variety of primitives in 3D.

Sclaroff and Pentland [Scla91] discussed another class of generalised implicit functions based on superellipsoids. Their goal was to use the inside/outside test to perform collision detection between implicitly defined shapes. Given an inside/outside function for a superellipsoid at the point $P(x, y, z)$:

$$f(P) = ((x^{\frac{2}{w_2}} + y^{\frac{2}{w_2}})^{\frac{w_2}{w_1}} + z^{\frac{2}{w_1}})^{\frac{w_1}{2}} - 1$$

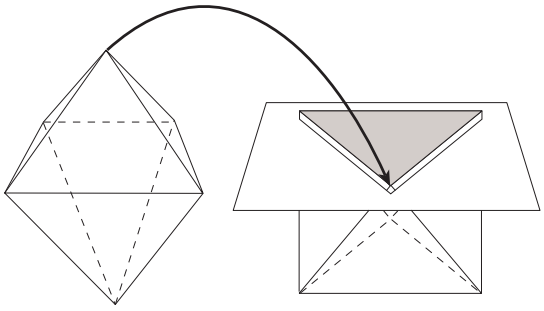


Figure 2.14: *Creating a triangle using 3D hyperquadrics*

where $w_1, w_2 \in \mathcal{R}$, the rotation matrix M, the translation vector $\overrightarrow{T}$, the deformation matrix D and the displacement function d that displaced the surface at a point P along the normal at P, $\overrightarrow{N}(P)$, the inside/outside function of an oriented, positioned, deformed superellipsoid with the displaced surface at a given point P was described by:

$$f(D^{-1}M^{-1}(P - \overrightarrow{T}) - d\overrightarrow{N})$$

Sclaroff and Pentland modelled seashell-like shapes using their generalised implicit formulation and animated them with collision detection in a physically based system.

Function representation

Another generalised way of dealing with algebraic primitives was proposed by Pasko *et al.* [Pask93, Pask95a] and Shapiro [Shap94]. In this approach, algebraic primitives were blended using extended set theoretic operations defined by R functions [Rvac87] which preserve the continuity of primitives. R function blending is discussed in Section 2.4. Thanks to the preserved continuity, any algebraic surface and any procedural surface (*e.g.* CSG solids and skeletal implicit primitives) could be blended using this approach (Figure 2.15).

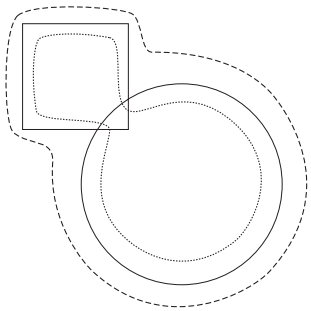


Figure 2.15: *Function representation: a square (defined as intersection of four half-planes) blended with a circle (an implicit surface); the blend is global and can be inside (dotted surface) or outside (dashed surface) the primitives*

2.3.3 Empirical primitives

The last group includes primitives given by a discrete scalar field in a 3D grid. This is a surface reconstruction situation and implicit surfaces have been successfully used in this context (see Section 2.5.4). McPheeters [McPh90] identified this group as a separate type of primitives to show the generality of the implicit formulation. In order to use empirical primitives, it is necessary to provide an interpolation function that will evaluate the field function at any point P, given only the field values at the discrete grid points.

2.4 Blending

Having defined the primitives for a model, a smooth manner of combining them is required. This process, called blending, has been applied for a long time in the context of solid modelling. This section will first present implicitly defined blends in solid modelling to prepare a background for a following discussion of blending methods used to combine primitives in the implicit surfaces model. Some questions arising for solid modelling and also valid for implicit surfaces include: global *vs* local blends, tangent continuity of the blend with the blended surfaces, intuitiveness of the shape of the blend surface or blend interference (preventing the blended surface from intersecting the surfaces being blended).

2.4.1 Foundations

Solid modelling and constructive solid geometry (CSG) find their main application in computer-aided design (CAD) and computer-aided modelling (CAM) systems that are used to model mechanical components, which are later machine-manufactured. For technical reasons, solids that result from set-theoretic operations, *e.g.* a cube with a hole, created by subtraction of a cylinder from the cube, should have their sharp edges and corners rounded (blended). The elementary objects in CSG are simple solids, *e.g.* a sphere, a plane, a cube, a cylinder, that can be described algebraically. Implicitly defined blends are created by combining algebraic surfaces using a blending function and taking the zero-set of the result as the blended surface between the two surfaces. For instance, given two algebraic surfaces, $f(x) = 0$ and $g(x) = 0$, the blend equivalent to the union operation $\cup$ can be achieved by using the function $\max(f(x), g(x))$. For a very good overview of blending techniques in solid modelling refer to [Wood87].

The set-theoretic union operation $\cup$ does not round the edges and corners of the solid. Ricci [Ricc73] was the first to introduce generalised set-theoretic operations of summation and intersection which offered a C^1-continuous blended surface. The smoothed union and intersection of two primitives defined by field functions F_1 and F_2 can be defined by:

$$\begin{aligned} U_p(F_1, F_2) &= (F_1^p + F_2^p)^{\frac{1}{p}} \\ I_p(F_1, F_2) &= (F_1^{-p} + F_2^{-p})^{\frac{-1}{p}} \end{aligned}$$

where $p \in \mathcal{R}_+$. When $p = \infty$, these functions represent exact set-theoretic union, $U_\infty = \max(F_1, F_2)$ and intersection, $I_\infty = \min(F_1, F_2)$. Ricci's blend is global, *i.e.* it affects the entire surface of primitives being blended. Figure 2.16 shows two primitives approaching each other and blended using this method.

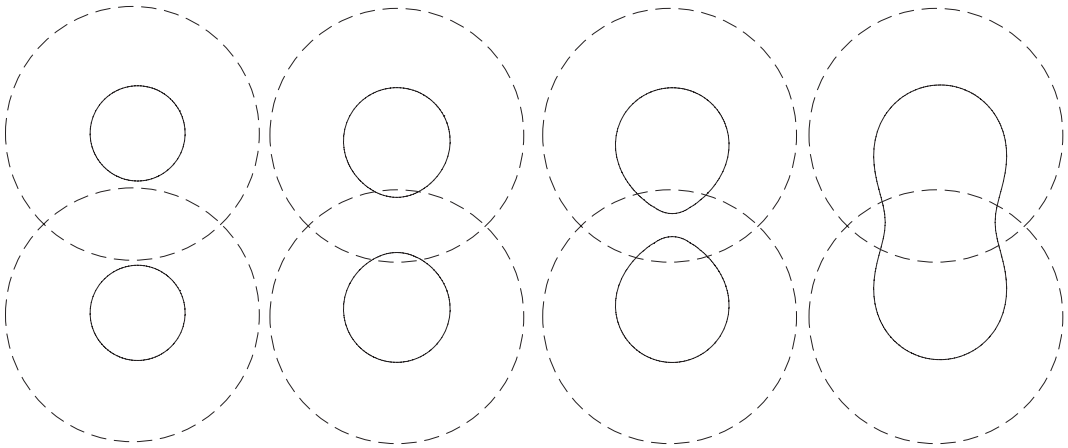


Figure 2.16: *Two primitives, defined by Wyvill* et al.*'s field function, approaching each other: Ricci's union blend,* $p = 1.5$, $Iso = 0.5$

Middleditch and Sears [Midd85] proposed a range controlled blended union between two algebraic surfaces described by F_1 and F_2, defined as follows:

$$(1 - u)F_1F_2 + u(F_1 + F_2 - r)^2 = 0$$

where r is the range of the blend and $u \in [0, 1]$. Figure 2.17 shows two primitives approaching each other and blended using this method.

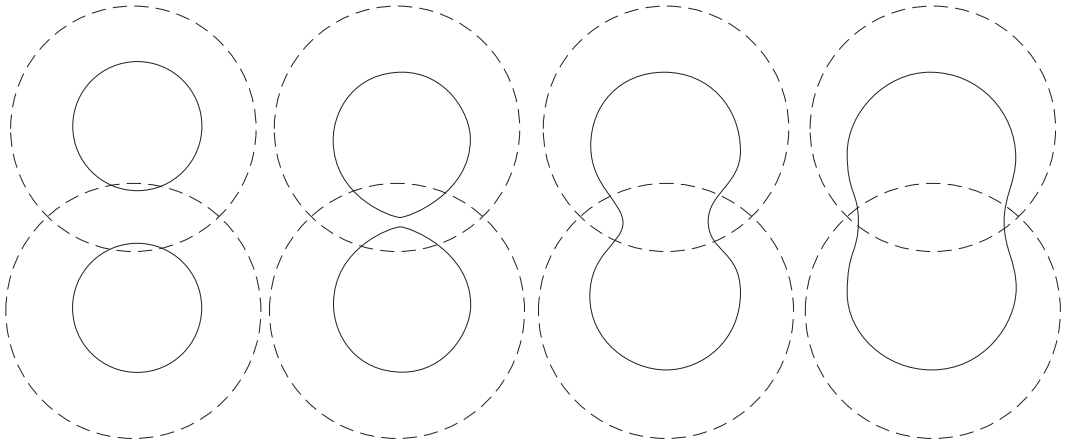


Figure 2.17: *Two primitives, defined by Wyvill* et el.*'s field function, approaching each other: Middleditch and Sears' union blend,* $u = \frac{1}{2}$, $r = 1$, $Iso = 0.5$

Another range controlled blend was proposed by Rockwood and Owen [Owen89, Rock90a, Rock90b]. The authors defined a separate range for both surfaces being blended: r_1 and r_2. The blended union surface is given by:

$$1 - \left(1 - \frac{1 - F_1}{r_1}\right)^w - \left(1 - \frac{1 - F_2}{r_2}\right)^w = 0$$

where $w \in \mathcal{R}$. The exponent w is used to control the "nearness" of the blend to the surfaces and the shape of the blend, for $w > 2$ the blend becomes superelliptic. Figure 2.18 shows two primitives approaching each other and blended using this method.

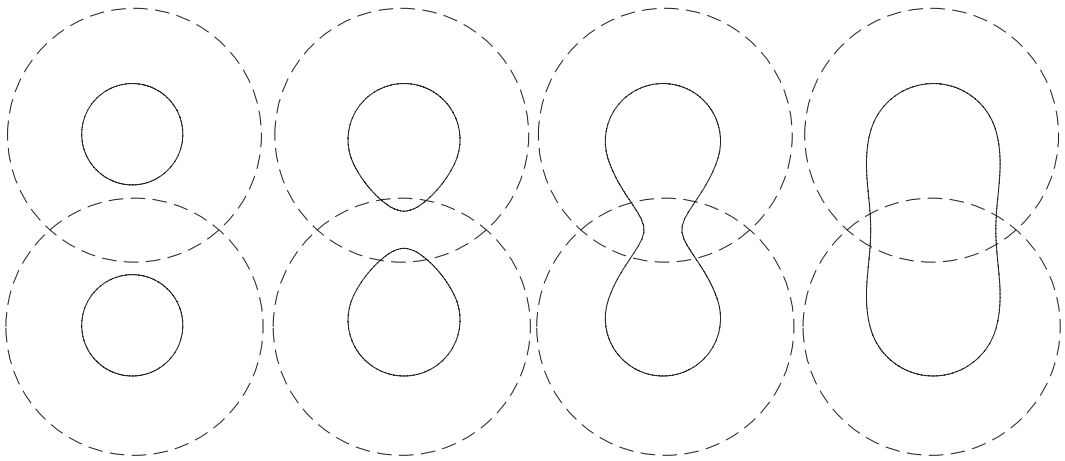


Figure 2.18: *Two primitives, defined by Wyvill* et al.*'s field function, approaching each other: Rockwood and Owen's union blend,* $w = 2$, $r_1 = r_2 = 1$, $Iso = 0.5$

The approach of Hoffman and Hopcroft [Hoff86] offered a superset of the two latter methods by using the following formulation to blend two surfaces:

$$r_2^2 F_1^2 + r_1^2 F_2^2 + 2v F_1 F_2 - 2r_1 r_2^2 F_1 - 2r_1^2 r_2 F_2 + r_1^2 r_2^2 = 0$$

where r_1, r_2 controlled the locality of the blend and $v \in [0, 1]$. Rockwood and Owen's blend can be obtained when $v = 0$ and Middleditch and Sears' blend by setting $v = \frac{1+u}{2u}$. Figure 2.19 shows two primitives approaching each other and blended using this method.

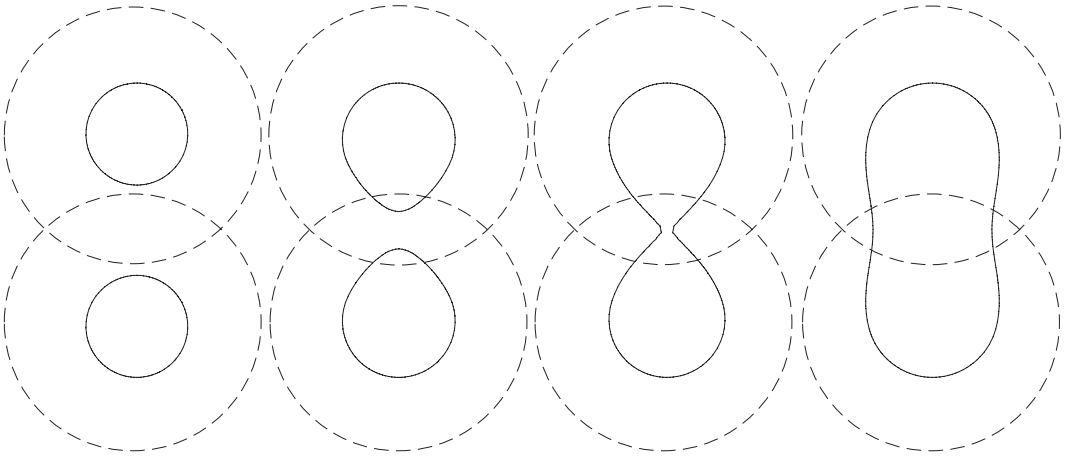


Figure 2.19: *Two primitives, defined by Wyvill* et al.*'s field function, approaching each other: Hoffman and Hopcroft's union blend,* $v = \frac{1}{2}$, $r_1 = r_2 = 1$, $Iso = 0.5$

2.4.2 R-functions

Pasko *et al.* [Pask88, Pask93, Pask95a] and Shapiro [Shap94] used another way of blending primitives, based on the theory of R-functions [Rvac87]. In this method, the set-theoretic operations were generalised as follows:

$$F_1 \cup F_2 = \frac{1}{1+\alpha}(F_1 + F_2 + \sqrt{F_1^2 + F_2^2 - 2\alpha F_1 F_2})$$

$$F_1 \cap F_2 = \frac{1}{1+\alpha}(F_1 + F_2 - \sqrt{F_1^2 + F_2^2 - 2\alpha F_1 F_2})$$

where $\alpha = \alpha(F_1, F_2)$ - an arbitrary function satisfying the following conditions:

$$-1 < \alpha(F_1, F_2) \leq 1$$
$$\alpha(F_1, F_2) = \alpha(F_2, F_1) = \alpha(-F_1, F_2) = \alpha(F_1, -F_2)$$

R-function blends can be applied to any implicit primitives. They offer C^m continuity and can exactly represent standard set-theoretic operations ($\alpha = 1$ in the above definition). They produce global blends, changing the entire surface of primitives. Figure 2.20 shows two primitives approaching each other and blended using this method.

2.4.3 Summation ($\sum$)

Summation of field functions is the simplest way of combining primitives. It is fast to evaluate and does not require any additional processing to create the blended surface. Its locality and the shape of the resulting blended surface strongly depend on the choice of the field function. For functions with infinite area of influence the blend is global, *e.g.* in [Blin82]. When the areas of influence are finite, the blend only occurs in the overlapping of the two areas and is therefore local. The tangent continuity of the resulting surface also depends on the choice of the field function. When a smooth potential function is chosen, the summation results in a smooth blend. For arbitrary field functions, *e.g.* a sphere implicitly defined using its algebraic equation, summation does not result in a C^1-continuous blended surface.

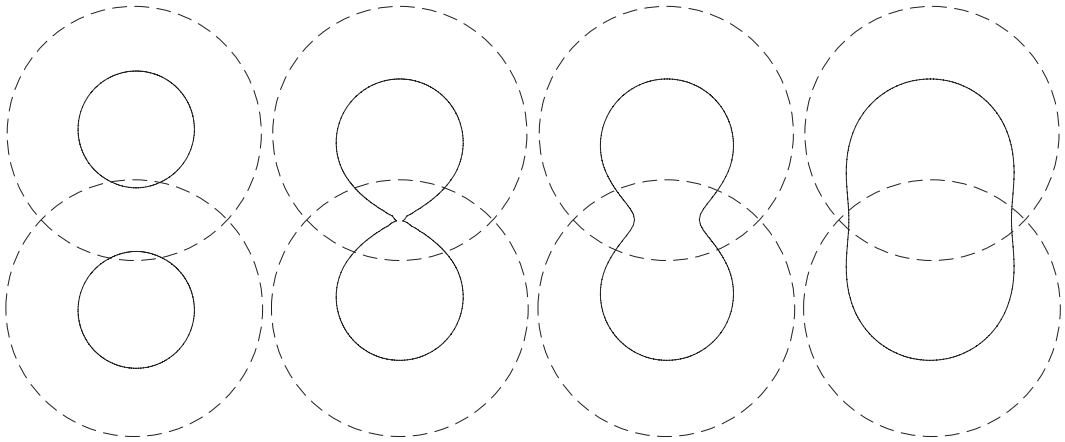


Figure 2.20: *Two primitives, defined by Wyvill* et al.*'s field function, approaching each other: R-function union blend,* $\alpha = 0$, $Iso = 0.5$

2.4.4 Procedural definition

To give the implicit designer more control over the way primitives blend, procedural blending was proposed [Bloo90a, Bloo90b]. In this method, a procedure that combines primitives is given explicitly. It attempts to deal deal, for instance, with branching structures and define the implicit surface at a branching as a complex combination of the branch field functions. In Figure 2.21 the field at the given point P is calculated considering its distance to the line connecting two closest points of the two branches. It results in a smooth blend if the angle between the two branches is greater than $\frac{\pi}{2}$. However, for angles smaller than $\frac{\pi}{2}$, discontinuities occur. Moreover, the method only offers global blending and extending it for the branchings of order higher than two is not trivial.

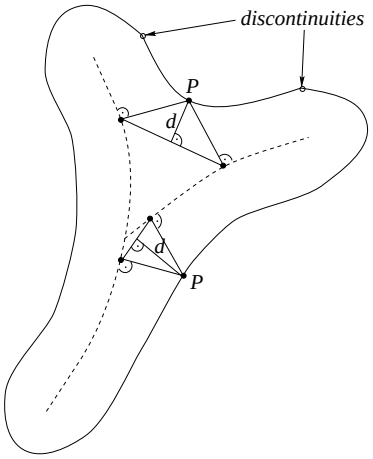


Figure 2.21: *Procedural blending at a branching: distance to the base of a triangle created between two branches is taken as the distance from* P *to the skeleton*

McPheeters [McPh90] proposed the use of a smoothing function on the result of a potential function before applying a blending method. The proposed smoothing function was:

$$f_{smooth}(v) = \frac{7}{4}v^3 - \frac{3}{4}v^4$$

The field value at a given point P due to a primitive is therefore calculated as $f_{smooth}(f(P))$, where $f(P)$ is the value of the potential function at P. Figure 2.22 illustrated the results. When two implicitly defined

spheres are blended by summation (Figure 2.22a) tangent disconituities occur. The use of a smoothing function before summation gives a tangent continuous surface (Figure 2.22b).

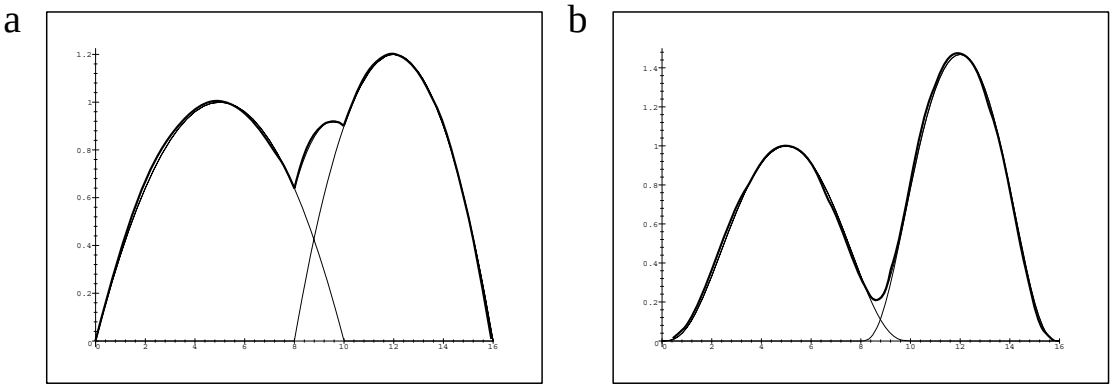


Figure 2.22: *Smoothing before blending: before applying f_{smooth} to the potential functions (left) and after, blended using $\sum$ (right)*

2.5 Applications

This section will present a variety of areas where implicit surfaces have been successfully applied. The tasks described include natural phenomena modelling, character and human animation, deformable substance simulation and surface reconstruction from three dimensional data sets.

2.5.1 Character and human modelling

The approach of Wyvill *et al.* [Wyvi86b] was used to create an animation called “The Great Train Rubbery” [Wyvi88]. The character in the film was a train, modelled using implicit surfaces, travelling through a soft landscape and emitting implicit clouds of smoke. The animation, not at all photo-realistic, was successful because of the pleasing, smooth, cartoon-like appearance of implicit surfaces. Another example of using implicit surfaces to model characters are the following three papers that presented attempts to implicitly model a very popular character, a dinosaur [Grav93, Espo95, Opal95a].

Beier has popularised implicit surfaces among the animators at the Pacific Data Images [Beie90]. After the success of their commercial for a children's some toothpaste featuring an implicit character that emerges from toothpaste to play a guitar and convince the kids to use this particular brand, the PDI team started to use implicit surfaces in much of their animation work appreciating their smooth appearance.

Opalach and Maddock [Opal94] investigated achieving Disney effects in character animation using implicit surfaces and incorporating dance notation into such animation [Opal95b]. These approaches will be detailed in the following chapters.

Implicit surfaces have also found applications in human figure modelling. Bloomenthal [Bloo92] used geometric generators to represent the bones (line segments), muscles (polygons) and veins (branching curves) of a human hand. Pelachaud *et al.* [Pela94] modelled the human tongue during speech using implicit surfaces. The model was built by taking as generators the triangles that represented the shape of the tongue. In Japan, a database of human anatomy parts modelled using implicit surfaces was created [Grav93].

Implicit surfaces can be used together with boundary representations, *e.g.* parametric patches or polygonal models. Shen and Thalmann [Shen95] created a model for the limbs of an articulated human figure using implicit surfaces and used it to find evenly spaced control points for a B-spline surface chosen for the final representation of the skin. Singh and Parent [Sing95] used implicit surfaces to control deformation of a polygonal model of a human figure. They first manually embedded the mesh in an implicit surface model,

ensuring that the polygon vertices lay on the implicit surface. During animation, the vertices of the polygonal mesh were kept on the implicit surface thus the deformation of the mesh was derived from the behaviour of the implicit surface. This approach benefited from the precise collision modelling algorithm [Gasc93] discussed in section 2.5.2 below.

2.5.2 Deformable substances

The appearance of implicit surface models suggests a clay like substance that can separate into pieces, cluster back together and undergo stretching or compression. This intuitive application of the technique stimulated its use in simulating deformable objects, including elastic and plastic behaviour.

Elastic substances

Marie-Paule Gascuel [Gasc93] developed a method for detecting and processing collisions between implicitly defined elastic objects in a physically based environment. During such a collision a precise contact surface between the objects was extracted and the reaction forces acting on the colliding objects were calculated. These forces were later integrated into the equations of motion.

The collision detection in the implicit model is efficiently performed for any given point due to the simple inside/outside test. Sampling points on a surface of an object can be tested against another object's field function. If some of them are inside the second object, a collision occurs and a deforming negative function is applied to an object in the collision zone. The deformed scalar field $F_d(P)$ is thus calculated as:

$$F_d(P) = F(P) + D(P)$$

where $F(P)$ is the scalar field function for the object being deformed and

$$D(P) \; = \; Iso \; - \; F(P)$$

is the negative deforming factor due to the objects colliding with it. This procedure was applied to all objects in the collision. For two objects in a collision the respective deformed fields are:

$$\begin{aligned} F_{1d}(P) &= F_1(P) + Iso - F_2(P) \\ F_{2d}(P) &= F_2(P) + Iso - F_1(P) \end{aligned}$$

The two isosurfaces extracted for the deformed objects are therefore in contact precisely along the surface where $F_1(P) = F_2(P)$.

The elasticity in this method was coded directly in the potential function. The local stiffness k at point P was calculated as the opposite of the slope of the potential function $k(P) = -F'(P)$. Linear and nonlinear elasticities can be modelled depending on the potential function chosen.

Plastic substances

The behaviour of a plastic substance can be modelled as an interaction between macro-particles that follow the physical laws of attraction and repulsion. Such was the principle of the method developed by Terzopoulos et al. [Terz89] in which the macro-particles were represented by point masses or of the methods of Miller and Pearce [Mill89] and Tonnesen [Tonn91] in which a particle system was used. In all three approaches implicit surfaces served as a convenient method for representing the surface of the substance by using the point masses or the particles as generators in the implicit model. However, the surface represented in this way was not used during simulations, *e.g.* for collision detection. It was utilised for a purely visual effect.

Desbrun and Marie-Paule Gascuel [Desb94, Desb95a] developed a method for modelling plastic materials, *e.g.* mud, clay or dough, taking the implicit model as the starting point. A highly deformable material was simulated in a generalised particle system with Lennard-Jones attraction/repulsion interaction forces between primitives. Such material could undergo fractures and its chunks could join back together. Collision detection and response between unblendable materials was modelled as in [Gasc93] and the progressive fusion between chunks occurred under influence of a sufficient force. Due to the use of field functions that resulted in a very softly blended surface and because the surface of the substance

was used for collision detection, the simulation required fewer particles than in the previously described methods [Terz89, Mill89, Tonn91]. An important issue of volume conservation of deformable material was also addressed. During animation, the implicit material could undergo considerable volume variations. By locally controlling the volume, this was decreased to under 3%, hence ensuring realistic simulation.

2.5.3 Natural phenomena

Fujita *et al.* [Fuji90] developed a method for simulating the splashing water effect using implicit surfaces. They modelled the behaviour of the water crown after a droplet falls into it by generating quadruplets of primitives around the fallen droplet. The primitives in each quadruplet started in a cluster and expanded during induced motion to form splashes of water.

Max and Geoff Wyvill [Max91] used implicit surfaces to model coral. The smoothly blended appearance of this species makes them particularly suitable for the implicit formulation.

Payne and Toga [Payn92] used implicit surfaces built as offset surfaces around polygonal models to model the brain of a rat. They used linear interpolation between models obtained for different stages of the brain development to reconstruct the development cycle.

Reed and Brian Wyvill [Reed94] simulated lightning using implicit surfaces. A trajectory of a lightning strike was modelled using a particle system and the glow was extracted from the *soft object* definition of the scalar field around a skeleton defined by such a trajectory.

Hart [Hart95] proposed an implicit formulation for fractals which have been successfully used to model organic shapes [Mand82]. This way fractals became another primitive in his implicit modelling environment. Hart used them to model rough surfaces, *e.g.* a blend between a jagged edge of a leaf and its stem or the rough bark of a tree.

2.5.4 Surface reconstruction

Implicit surfaces represent the surface of an object by describing a set of generators and a way of producing a scalar field around them. They are therefore an attractive tool for surface reconstruction from volume data since they can replace large data sets by a compact model.

Tatsumi *et al.* [Tats90] used implicit surfaces to model Ito cells found in the liver of a cod. The three dimensional primitives were manually fitted to two dimensional cross sections of the given medical data in an interactive editor. The resulting model was then rendered as a transparent surface and combined with a representation of blood capillary. This way the authors could observe the relationship between Ito cells and blood capillary.

Muraki [Mura91] developed an automatic method for fitting an implicit surface to volume data. The reconstruction started with approximating the data by one primitive positioned at the centre of mass of the data points. The iterative refinement was achieved by replacing a suitable primitive by two smaller primitives. The candidate primitive was chosen to minimise an error function that measured the difference between the approximated surface and the reconstructed surface. The error function was expressed in terms of an average scalar field value and an average normal vector at a chosen m points of the data set. These values were compared with the corresponding field values for the reconstructed surface. The approach was unsatisfactory for practical purposes particularly due to its expensive algorithm for choosing the splitting candidate that examined all primitives in each iteration step and the global character of the reconstruction process.

Tsingos *et al.* [Tsin95] improved Muraki's technique by proposing an efficient heuristic for the refinement step based on local field functions and “reconstruction windows” offering a further localisation of the reconstruction process. Since the field functions were local, the reconstruction could be preserved between the iteration steps in the areas where the current approximation was sufficiently accurate. The primitive to be split was chosen as the primitive for which the local fitting was the worst, *i.e.* for which the sum of field values at these points from the given data set that are inside the primitive's radius of influence divided by the number of the points was the greatest. A further improvement was to introduce “reconstruction windows” that served as bounding boxes for areas separately reconstructed considering only the primitives and data points inside each window during the refinement step. The method produced good results for

objects of arbitrary topology and geometry and only required the user specification of the position for the first approximating primitive.

Savchenko *et al.* [Savc95] defined a scalar field at the data points by introducing an implicitly defined carrier solid, *e.g.* a sphere. The inside/outside function that described the carrier solid was called the carrier function. The values of the carrier function at the given set of points were then interpolated by a spline. The function F that defined the reconstructed solid as $\{P \in \mathcal{R}^3 : \mathcal{F}(\mathcal{P}) \; 6 \; \mathit{l}\}$ was given by the algebraic difference between the spline and the carrier function.

2.6 Summary

This chapter has presented a review and a historical sketch of the development of implicit surfaces. During the past few years skeleton based implicit surfaces have become a very popular modelling technique. They offer an intuitive approach to modelling: a set of generators approximates the shape of an object, forming its skeleton, and the implicit "skin" automatically and smoothly clothes such a skeleton. Generating similar objects using parametric techniques, *e.g.* by sweeping a shape along a skeleton, is more difficult to control and may produce artifacts like interpenetration or surface discontinuities.

Implicit surfaces offer several advantages. Deformation, including topological changes, is relatively simple and is achievable by simple alterations of the descriptions of primitives in a model. A smooth, continuous surface around generators is always guaranteed. This property makes implicit surfaces particularly suitable for modelling objects that change their shape, *e.g.* for "plasticine" character modelling which is the chosen area for this thesis.

Moreover, applying implicit surfaces to animation can benefit from a simple method of collision detection between implicitly represented objects, a process essential for animation or simulation. During collision processing, a precise contact surface can be modelled, as opposed to approximated methods with possible interpenetration and deformation modelled before contact in the case of parametric surfaces.

The implicit representation is very compact, only requiring storage of the description of primitives (generators and field functions) and a blending method. This makes it attractive for three dimensional data set representation and also lowers the storage cost for any application that needs a large number of scenes to be represented, *e.g.* storing frames of an animation sequence.

Chapter 3

The ABC of Implicit Surfaces

3.1 Introduction

This chapter will present aspects of character modelling using implicit surfaces. Some limitations of the technique become apparent when attempting to animate the created models. Character animation deals with *characters*, *i.e.* objects that are given a "personality". The definition of this notion strongly relies on the subjective perception of a viewer who is guided by the look and the behaviour of a character. A character should remain distinct and retain its general shape during animation. In the realm of implicit surfaces it is easy to loose such characteristics. For instance, Figure 3.3a shows a model inspired by the work of Salvador Dali. There is no structure imposed on the primitives. When attempting to arbitrarily animate the primitives, the intended look can easily be destroyed. This is the problem of *appearance loss*, covered in Section 3.2.

One way of loosing appearance is to allow the characteristic features of a character to blend together. For instance, Figure 3.3b shows how moving the primitives too close to each other results in the face and the tail of the character progressively disappearing. Another example is shown in Figure 3.1 where an arm and a leg of a dancing character blend. Although smooth blending of primitives is a valuable asset of implicit surfaces, in such a situation it is undesired. Therefore, *unwanted blending* in the implicit model should be controlled to prevent merging between parts of a character. Section 3.3 describes the proposed solution.

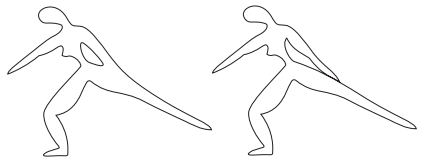


Figure 3.1: *An example of unwanted blending, Left: undesired situation, Right: corrected*

The problem opposite to unwanted blending is *coherence loss*. It occurs when two parts of a model of a character are placed too far from each other and as a result disconnect, causing the loss of topological integrity of the model. For instance, Figure 3.3c the primitives in the model were moved too far from each other and the object starts looking like a random collection of primitives. An example of coherence loss in dancing character is shown in Figure 3.2. Coherence is defined here as a property of an object which causes it to maintain its topological integrity during animation. To enforce it in a model, the primitives have to be prevented from moving too far from each other. At the same time purely manual specification of motion for all primitives in the model should be avoided. Therefore, when one primitive moves, the others should follow its motion. The proposed solution is presented in Section 3.4.

Solutions to these three problems will be referred to as the ABC (Appearance, Blending, Coherence) of implicit surfaces for character animation. It describes a method for modelling a coherent block of clay-like substance which can retain its general shape during animation according to its appearance defined using a graph of links and joints. The ABC approach was originally proposed in [Opal93b]. In this chapter more details will be included along with a working case study of a simple but expressive character animated using keyframing (Section 3.5). Because of the extended implicit model the character presents a fluidity of shape

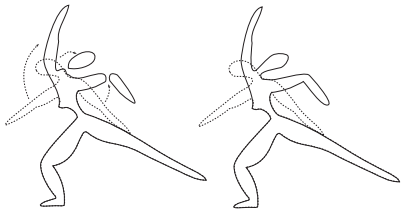


Figure 3.2: *An example of coherence loss, Left: undesired situation, Right: corrected*

and motion while staying recognisable during animation.

3.2 Appearance

Providing a means to define appearance is essential for character animation to provide structural foundations that are necessary to make characters recognisable throughout their lives. To make such recognition possible, characteristic parts of an object have to be defined and maintained during an animation sequence. The method proposed in this section defines parts of a character as elements (links and joints) of a graph. An animator manually designs all such parts and specifies their place in the character's graph. The general concept of this approach will be presented first, followed by a description of model scripting and positioning in world coordinates.

3.2.1 Appearance graph

An *appearance graph* defines a skeleton of an object in terms of joints and links. Each link and joint has a group of primitives assigned to it. Such a group of primitives, forming a characteristic part of an object, will be referred to as an *object component* or a *component*. Figure 3.4 shows a simple appearance graph composed of three components: $link_1$, $link_2$ and $joint$.

An appearance graph is the conceptual skeleton of an object. It imposes a structure within primitives and provides a division of the object into components. It does not define the number of primitives or their spacial relationship. It is possible to use the same appearance graph for deformed versions of the same model. Hence the skeleton, defined using an appearance graph, is deformable not rigid. Local deformation can be achieved by moving some of the primitives. If new primitives are added, extreme deformation, *e.g.* over-stretching of a character, can be modelled. For example in Figure 3.4 the link $link_1$ and joint $joint$ are in both cases assigned one primitive. The link $link_2$ is assigned one primitive in the left object and three primitives in the right one. As a result $link_2$ increases its length, *e.g.* representing the action of stretching.

3.2.2 Definition of a model

A script file is used to define an appearance graph and primitives that form components in an object. Figure 3.5 gives a script file for a simplified human arm. Primitives for each component (*upper arm, forearm, elbow, wrist* and *hand*) are specified in the component's local coordinates. A component's initial rotation around its local X,Y and Z axes is given as a vector of three rotation angles $(\alpha_x, \alpha_y, \alpha_z)$. In Figure 3.5 the rotation vectors are marked with $R_{upperarm}$, R_{elbow}, $R_{forearm}$, R_{wrist} and R_{hand}. The components are then linked together to form an appearance graph. For each joint, the possible cases are: (i) it is a single joint, (ii) it is attached to a link or (iii) it is between two links. Cases (ii) and (iii) may occur for the same joint if more than two links meet at that joint. For cases (ii) and (iii), displacements of the origin of the local coordinate system of a joint in relation to each of its links is given as a translation vector (t_x, t_y, t_z). There are two joints in Figure 3.5, *elbow* and *wrist*, each between two links. The displacement of *elbow* in relation to *upper_arm* is given by vector $T_{elbow\ upper\_arm}$ and in relation to *forearm* by vector $T_{elbow\ forearm}$. Similarly, $T_{wrist\ forearm}$ and $T_{wrist\ hand}$ give displacements of *wrist* in relation to *forearm* and *hand* respectively. Finally, one of the components is chosen as the *anchor* of the model and positioned in the world coordinate system by a translation of its local origin T_{anchor}. In Figure 3.5 *elbow* is the anchor. A model

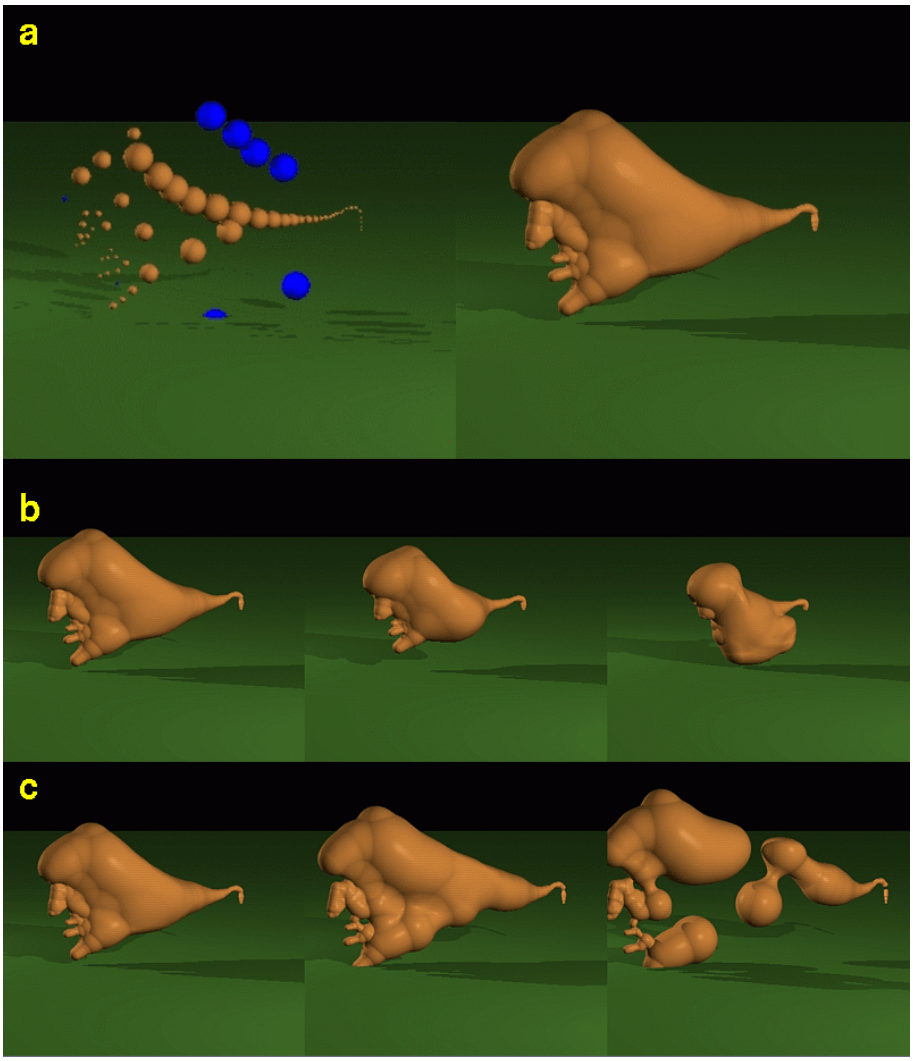


Figure 3.3: *a) A model inspired by the work of Salvador Dali: Lighter spheres represent positive primitives, darker spheres represent negative primitives, the radius of each sphere is proportional to the radius of influence of the corresponding primitive. b) Loosing appearance by unwanted blending c) Loosing appearance by coherence loss*

is positioned in the world coordinate system during parsing of its script file. The next section describes the calculation of the required transformation matrices.

3.2.3 Positioning a model in the world

A point in space is represented as a row-vector, $[\ x \quad y \quad z \quad 1\ ]$, using homogeneous coordinates, and its transformed point is $[\ x' \quad y' \quad z' \quad 1\ ]$:

$$[\ x' \quad y' \quad z' \quad 1\ ] = [\ x \quad y \quad z \quad 1\ ] \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix}$$

All components in a model are specified in their respective local coordinate systems. To position the model in the world, a transformation matrix needs to be calculated for each of them. After the scripting stage, a local rotation matrix M_R is known for each component. It is a multiplication of three rotation

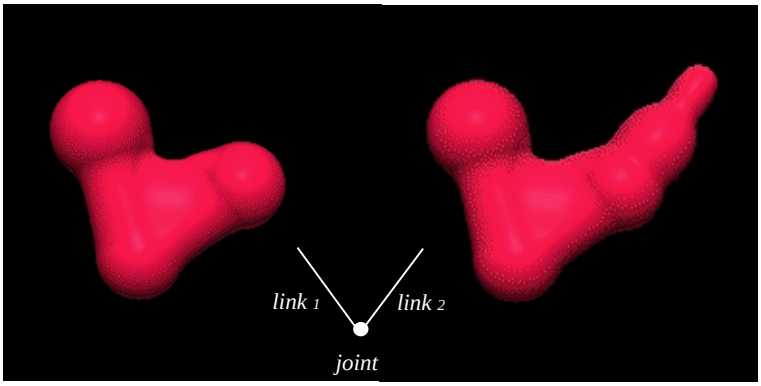


Figure 3.4: *The flexibility of appearance specification: the number of primitives or their spacial relationship are not fixed, which is useful for "squash and stretch" effects*

matrices, each of them describing clockwise rotation around an axis specified by a local rotation vector $R = (\alpha_x, \alpha_y, \alpha_z)$:

$$M_{R_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_x & \sin\alpha_x & 0 \\ 0 & -\sin\alpha_x & \cos\alpha_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{R_y} = \begin{bmatrix} \cos\alpha_y & 0 & -\sin\alpha_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha_y & 0 & \cos\alpha_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{R_z} = \begin{bmatrix} \cos\alpha_z & \sin\alpha_z & 0 & 0 \\ -\sin\alpha_z & \cos\alpha_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

After multiplying M_{R_x}, M_{R_y} and M_{R_z}, the combined local rotation around the X, Y and Z axes, M_R, becomes:

$$\begin{bmatrix} \cos\alpha_y\cos\alpha_z & \cos\alpha_y\sin\alpha_z & -\sin\alpha_y & 0 \\ \sin\alpha_x\sin\alpha_y\cos\alpha_z - \cos\alpha_x\sin\alpha_z & \sin\alpha_x\sin\alpha_y\sin\alpha_z + \cos\alpha_x\cos\alpha_z & \sin\alpha_x\cos\alpha_y & 0 \\ \cos\alpha_x\sin\alpha_y\cos\alpha_z + \sin\alpha_x\sin\alpha_z & \cos\alpha_x\sin\alpha_y\sin\alpha_z - \sin\alpha_x\cos\alpha_z & \cos\alpha_x\cos\alpha_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For the model's anchor, the translation of the origin of its local coordinate system, $T_a = (x_a, y_a, z_a)$, is also given. Its matrix, M_{Ta}, is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_a & y_a & z_a & 1 \end{bmatrix}$$

The matrix that gives the world position of the anchor, M_{wa}, is the multiplication of the anchor's local rotation matrix M_{Ra} and its translation matrix M_{Ta}:

$$M_{wa} = M_{Ra}M_{Ta}$$

To calculate the matrix that describes any other component's position in the world, the appearance graph of the model is traversed starting from the anchor. The traversal is performed breadthwise to ensure that all required matrices for a parent node are known when computing the matrices for each of its children.

```
iso 0.8
object arm colour Flesh
    component upper_arm rotation (0,0,-30) <-- R upper_arm
        point (0,0,0) POVfunction 35 1
        point (0,-25,0) POVfunction 50 1
        point (0,-45,0) POVfunction 30 1
        blend SUM
    component forearm rotation (0,0,-150) <-- R forearm
        point (0,0,0) POVfunction 30 1
        point (0,-35,0) POVfunction 25 1
        blend SUM
    component hand rotation (0,0,10) <-- R hand
        point (0,0,0) POVfunction 15 1
        point (5,-10.5,0) POVfunction 20 1
        point (5,-21.5,0) POVfunction 10 1
        point (0,-27,0) POVfunction 7 1
        blend SUM
    component elbow rotation (0,0,0) <-- R elbow
        point (0,0,0) POVfunction 30 1
        blend SUM
    component wrist rotation (0,0,0) <-- R wrist
        point (0,0,0) POVfunction 10 1
        blend SUM
        joint elbow between upper_arm (-45,-60,0) <-- T elbow upper_arm
                            forearm (2,10,0) <-- T elbow forearm
    joint wrist between hand (0,10,0) <-- T wrist hand
                         forearm (4,-40,0) <-- T wrist forearm
    anchor elbow (0,0,0) <-- T anchor
end
```

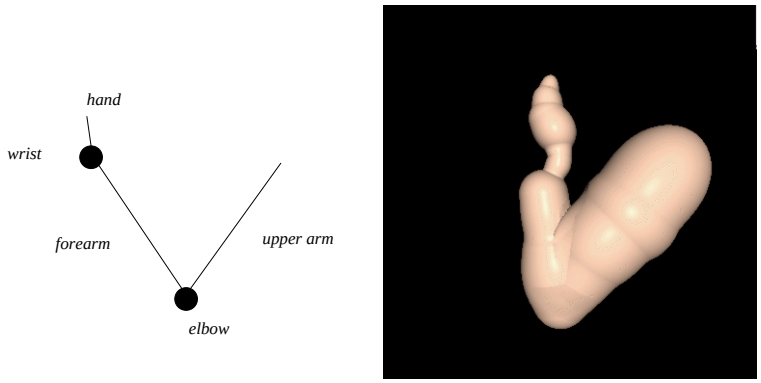


Figure 3.5: *Specifying a model for a simplified arm: script, appearance graph and model*

During the traversal, a displacement matrix M_D is calculated for each component. It is a translation matrix that describes a scripted displacement between a joint and a link ($T_{elbow\ upper\_arm}$, $T_{elbow\ forearm}$, $T_{wrist\ forearm}$ and $T_{wrist\ hand}$ in Figure 3.5). If the parent of the considered component is a joint, the displacement matrix is the translation by the specified displacement vector $T_{parent\ child}$, otherwise the reversed translation $-T_{parent\ child}$ is taken. For example, in the pair $elbow\ upper\_arm$, $elbow$'s displacement is given by the vector $T_{elbow\ upper\_arm}$ and $upper\_arm$'s displacement is described by $-T_{elbow\ upper\_arm}$.

The matrix that gives the world position for a component, M_{wc}, is the multiplication of three matrices: (i) the local rotation matrix for the component M_{Rc}, (ii) the displacement matrix describing the displacement from the parent to the child, M_{Tpc}, and (iii) the matrix inherited from the parent, M_{wp}:

$$M_{wc} = M_{Rc} M_{Tpc} M_{wp}$$

3.2.4 Flexibility of the approach

An example of the usefulness of deformable skeletons for character animation is shown in Figure 3.6. Recall the fairy-tale of Pinocchio, a wooden boy whose nose grew when he lied. With a deformable skeleton such an effect is easy to achieve. Pinocchio's nose can grow and shrink during animation, either by changing the spacial relationship between primitives or their number, without the need to alter the appearance graph.

```
iso 0.35
    object Pinocchio colour Wood
    component head rotation (0,0,0)
        point (.8,2.5,-.6) POVfunction 1 .8
        point (1.2,2.7,-1.6) POVfunction 1 .8
        blend SUM
    component nose rotation (0,0,0)
        point (1.8,2.5,-.6) POVfunction 1 .3
        point (2.0,2.5,-.6) POVfunction 1 .2
        blend SUM
    joint nose attachedto head (0,0,0)
    anchor head (0,0,0)
end
```

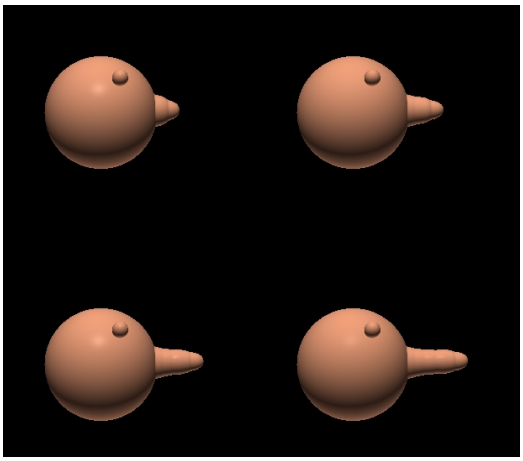


Figure 3.6: *A model of Pinocchio's head: Pinocchio's nose grows when he lies - extra primitives were added to the nose model to achieve the growing effect. The script gives the top left version of the model.*

The use of an appearance graph provides a basis for the problems of unwanted blending (Section 3.3) and coherence loss (Section 3.4). It distinguishes components within an object. The remainder of this chapter will present ways of ensuring that these components are recognisable during an animation sequence.

3.3 Blending

The appearance of a character, once defined, should be preserved during animation. However, the problem of unwanted blending may destroy the appearance by merging components of a character. It has been a long known disadvantage of implicit surfaces yet there have not been many proposed solutions. Two general approaches can be identified: overlapping unblendable objects [Wyvi89, Beie90] and modifying the scalar field to model the deformation [Opal93b, Desb95a, Guy95]. The former method is simpler to calculate yet does not produce intuitive results. In particular, since no contact surface is calculated, colliding objects overlap which can easily be revealed in a 2D cross section. The latter method offers a representation for a precise contact surface between objects but introduces C^1-discontinuities to the model. The two methods are illustrated in Figure 3.7. Although no difference can be seen in 3D (top row), the 2D cross section in the bottom left shows that in the former method objects are simply superimposed - the undeformed lighter object covers the darker one. The approach described in this chapter falls into the second category.

3.3.1 Related work

The first partial solution to the unwanted blending problem was proposed by Brian and Geoff Wyvill [Wyvi89]. The idea in their method was to divide the primitives in a scene into groups which are unblendable with each other and then render each group separately. The result was a union of unblendable groups in which objects overlap to model a collision. This method does not solve the problem of three groups, a joint and two links, in which the joint is blendable with the two links which are unblendable with each other.

Beier [Beie90] solved this problem by dividing primitives into groups and introducing a *root group* which is blendable with all other groups. In the case of a joint with two links, the joint would belong to the root group while each of the links would be a part of a different group. The unblendable groups were still handled as a union, while a more natural and expected result would be to detect and process collisions between them. Moreover, this method still leaves unsolved the problem of unwanted blending between two joints, *e.g.* two elbows of a character. Beier suggested an extension to his method in which each group would have a list of its blendable groups.

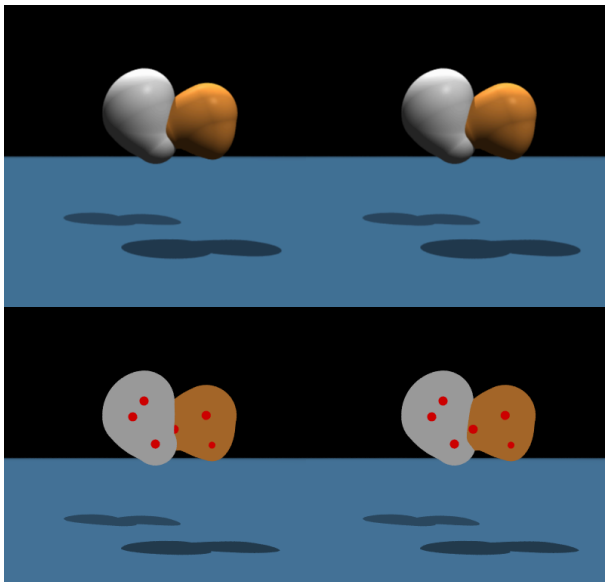


Figure 3.7: *Two approaches to unwanted blending (3D and 2D cross sections): overlapping objects (left column) and precise contact surface between objects (right column)*

The method to be described in this chapter was initially proposed in [Opal93b]. It suggested an algorithm that handled collisions between objects as well as self-collisions between different parts of one object. It was a graph based approach in which each part of an articulated figure represented a joint or a link in a graph structure. The blending properties of the model were derived from this graph: each joint was blendable with all its links and there were no other blendable groups in the model. Each object was divided into blendable components and for each of these a deformed scalar field was calculated to model precise contact surface in both collisions and self-collisions. The final scene was composed as a union of deformed blendable components. Since the blendable components fitted exactly to each other, there was no overlapping in the scene.

Recently, more graph based methods have been introduced [Desb95a, Guy95]. In these, all components of a model were represented as nodes in a graph and connections between them marked the desired blendability. Desbrun and Gascuel [Desb95a] simulated soft substance using implicit surfaces, hence they used a time-varying blending graph to record changes of blending properties due to merging or separation of chunks of substance. To render the substance according to current blending properties, they divided the scene into groups of blendable primitives, considering more than once primitives that belong to more than one such group. They composed the final scene as a union of such components. Since the precise contact modelling algorithm [Gasc93] was used, the objects did not overlap.

Guy and Wyvill [Guy95] calculated the deformed scalar field based on a graph of blending properties in which a connection between nodes represented desired blendability. Their method was a variation of the algorithm proposed in [Opal93b]. At any given point P, the deformed scalar field value due to i-th primitive, $F_{idef}(P)$, was calculated as follows:

$$F_{idef}(P) = F_i(P) + F_{mb}(P) + \sum_{j=0}^{n} \{G_j(P) : j \neq i, j \neq mb\}$$

where $F(P)$ is the primitive's undeformed scalar field, $F_{mb}(P)$ is the maximal value at P due to all primitives blendable with the considered one and $G_j(P)$ is the deformation function implemented after [Gasc93]. The final deformed scalar field at point P is the maximum of all deformed scalar fields $F_{idef}(P)$.

The following sections give details of the algorithm initially proposed in [Opal93b]. Collisions between objects are handled in a similar way to Gascuel's precise contact modelling algorithm [Gasc93]. However, the self-collision algorithm is a novel development in the area of implicit surfaces.

3.3.2 Implicit surfaces with blending properties

To avoid unwanted blending between the components of a character, the *blending properties* of a model, *i.e.* the information about which primitives of a model blend with each other and which do not, have to be uniquely determined. In the appearance graph the default blending properties assume that each joint blends only with its links. The appearance graph is therefore also a way of specifying the blending properties of a model; there is no need to generate a supplementary data structure. Moreover, the blending graph represented by the appearance graph is simpler (fewer nodes and connections) than a graph in which all object parts are represented in nodes and connections denote blending. Figure 3.8 shows a model of a simplified arm when all its parts blend together (unwanted blending situation) and with the default blending properties (unwanted blending prevented).

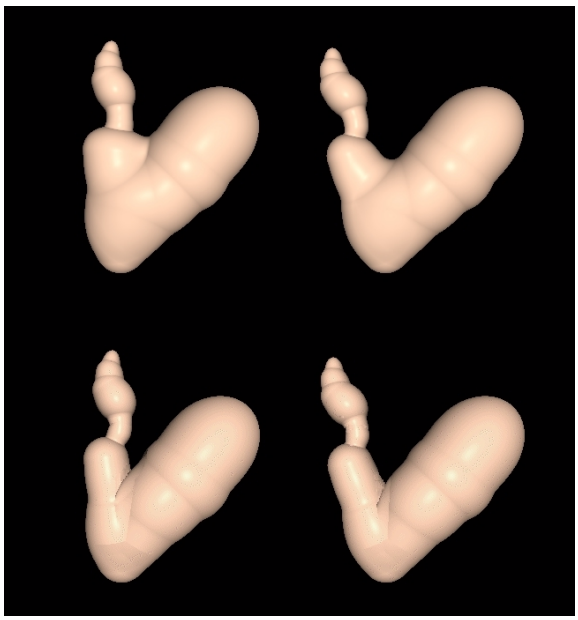


Figure 3.8: *Top: an arm with all its parts blending together (unwanted blending situation) Bottom: the same model with default blending properties (unwanted blending prevented)*

Figure 3.9 shows the principle of the proposed algorithm. A scene is specified as a set of appearance graphs defining all objects and a set of components representing links and joints in all appearance graphs. This stage is represented in Figure 3.9a. The next stage (Figure 3.9b) is to decompose each object into components (links and joints) restricted to an area where a component's influence is highest. In order to represent deformation of each such component due to blending or collisions, they are considered in turn and their scalar fields are modified to model blending between blendable components and a precise contact surface between unblendable components (Figure 3.9c). A union of deformed components for all objects gives the final scene (Figure 3.9d).

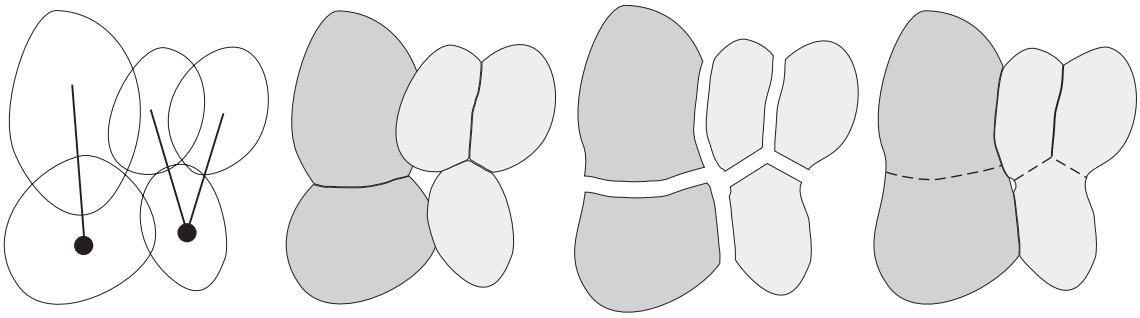
An object component

An *object component* is a group of primitives that stay blended together during animation. It represents a part of a character. Given a component C composed of n primitives with field functions $f_i, i = 1..n$, the undeformed scalar field due to C at a point $P \in \mathcal{R}^3$ is calculated as:

$$F_C(P) = \sum_{i=1}^{n} f_i(P)$$

which is the basic implicit surfaces algorithm as described in Chapter 2.

Each object component influences an area of space equal to the union of areas of influence of all primitives in the component. A *maximal influence area* for a component C, $\mathcal{M}(\mathcal{C})$, will denote the part of its



a) object components and appearance graphs | b) components in their maximum influence areas | c) components deformed by blending and collisions | d) final scene

Figure 3.9: *Implicit surfaces with blending properties: stages of the algorithm*

area of influence, in which its scalar field value is greater than the scalar field value from any other object component. More formally:

$$\mathcal{M}(\mathcal{C}) = \{\mathcal{P} \in \mathcal{R}^{\exists} : \mathcal{F}_{\mathcal{C}}(\mathcal{P}) \text{ ¿ } \prime \wedge \mathcal{F}_{\mathcal{C}}(\mathcal{P}) = \max_{\rangle}\{\mathcal{F}_{\mathcal{C}_{\rangle}}(\mathcal{P})\}\}$$

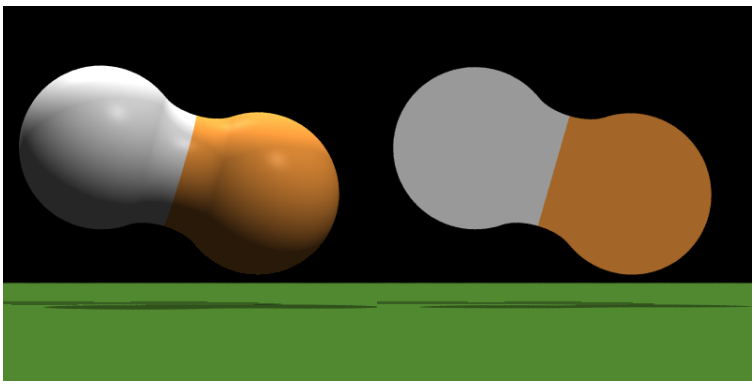


Figure 3.10: *An object built of two blendable components: 3D and a 2D cross section*

Figure 3.10 shows a simple object composed of two components that blend with each other, *i.e.* its appearance graph consists of one joint and one link. The two components, deformed due to blending, are shown in different colours.

Deformation of an object component

In order to calculate the field value at a point P due to a deformed object component C, $F_{C_{def}}(P)$, scalar field values from all components that influence P need to be considered. There are two groups of such components: a set of components blendable with C, $\mathcal{B}(\mathcal{C})$, and a set of components unblendable with it, $\mathcal{U}(\mathcal{C})$.

If the point P does not lie within the maximal influence area of the component, $\mathcal{M}(\mathcal{C})$, zero is assigned to the deformed field value. Otherwise three values at P are computed: (i) the undeformed field value for C, $F_C(P)$, (ii) the collision term, *i.e.* the sum of all field values due to components in $\mathcal{U}(\mathcal{C})$:

$$F_{\mathcal{U}(\mathcal{C})}(P) = \sum_{C_u \in \mathcal{U}(\mathcal{C})} F_{C_u}(P)$$

and (iii) the blending term, *i.e.* the sum of field values due to those components in $\mathcal{B}(\mathcal{C})$ that are greater than

all components unblendable with them:

$$F_{\mathcal{B}(\mathcal{C})}(P) = \sum_{\substack{C_b \in \mathcal{B}(\mathcal{C}) \wedge \\ \forall_{\mathcal{C}_n \in \mathcal{U}(\mathcal{C}_l)} \mathcal{F}_{\mathcal{C}_n}(P) \leq \mathcal{F}_{\mathcal{C}_l}(P)}} F_{C_b}(P)$$

$F_C(P)$ and $F_{\mathcal{B}(\mathcal{C})}(P)$ are added and field values at P due to unblendable components from $\mathcal{U}(\mathcal{C})$ are examined. If any of them is greater or equal to the isovalue and if $F_C(P)$ is greater or equal to the isovalue, the collision term $F_{\mathcal{U}(\mathcal{C})}$ needs to be considered. It is subtracted from $F_C(P) + F_{\mathcal{B}(\mathcal{C})}(P)$ and the isovalue is added to the final field value. This way the deformed surface of the component is pushed towards its inside and a precise contact surface is created at all points where $F_C(P) + F_{\mathcal{B}(\mathcal{C})} = F_{\mathcal{U}(\mathcal{C})}$.

Here is a more formal algorithmic expression for $F_{C_{def}}(P)$:

$$\begin{array}{lll} \text{if} & P \notin \mathcal{M}(\mathcal{C}) & \text{then } F_{C_{def}}(P) = 0 \\ \text{elsif} & \exists_{C_u \in \mathcal{U}(\mathcal{C})} F_{C_u}(P) \text{ ¿ } Iso \wedge F_C(P) \text{ ¿ } Iso & \text{then } F_{C_{def}}(P) = F_C(P) + F_{\mathcal{B}(\mathcal{C})} - F_{\mathcal{U}(\mathcal{C})} + Iso \\ \text{else} & & F_{C_{def}}(P) = F_C(P) + F_{\mathcal{B}(\mathcal{C})} \end{array}$$

This algorithm will now be presented in a few examples to clarify its details. Two cases of blending properties will be considered: (i) a collision between objects composed of one component, to illustrate the creation of a precise contact surface between objects and (ii) a self-collision in an object composed of two links and a joint between them, to show the precise contact surface in this case.

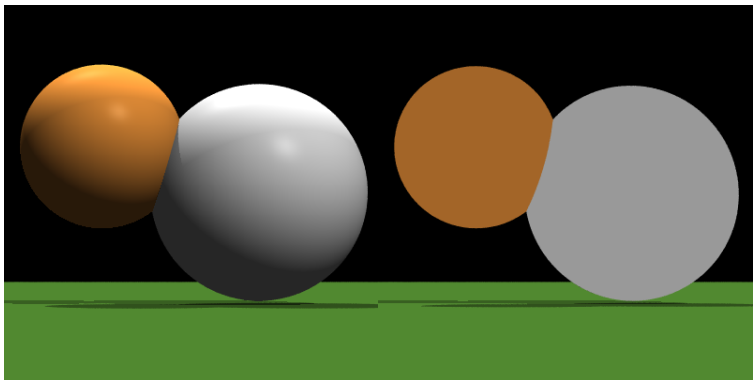


Figure 3.11: *A collision between two objects: 3D and a 2D cross section*

3.3.3 A collision between objects

Each object chosen to illustrate this section has an appearance graph composed of one joint, *i.e.* there is one component in each object. For simplicity of figures, each component contains only one primitive. This does not limit the presented considerations. More primitives can be used in each component since the details of a component's scalar field calculation are irrelevant to the calculation between components. The calculation of component deformation is order-independent. The component marked C_1 in the following diagrams is arbitrarily chosen in presenting each of the examples. Therefore it will be presented each time for only one sample component.

Two objects

Figure 3.11 shows a collision between two objects. Figure 3.12 illustrates the following reasoning which is valid for both objects. For component C_1, the set of its blendable components is an empty set, *i.e.* there are no other components in the scene that it is blendable with. Therefore, only the influence from the set of its unblendable components, $\mathcal{U}(\mathcal{C}_\infty)$, need to be considered. In Figure 3.12, $\mathcal{U}(\mathcal{C}_\infty) = \{\mathcal{C}_\epsilon\}$. The dashed line in Figure 3.12 shows the maximal influence area for the component C_1, $\mathcal{M}(\mathcal{C}_\infty)$. Outside $\mathcal{M}(\mathcal{C}_\infty)$, the deformed scalar field value is set to zero. The cross-hatched region in Figure 3.12 indicates the area where both $C_1(P)$ ¿ Iso and $C_2(P)$ ¿ Iso. Outside this region and within the maximal influence area of C_1,

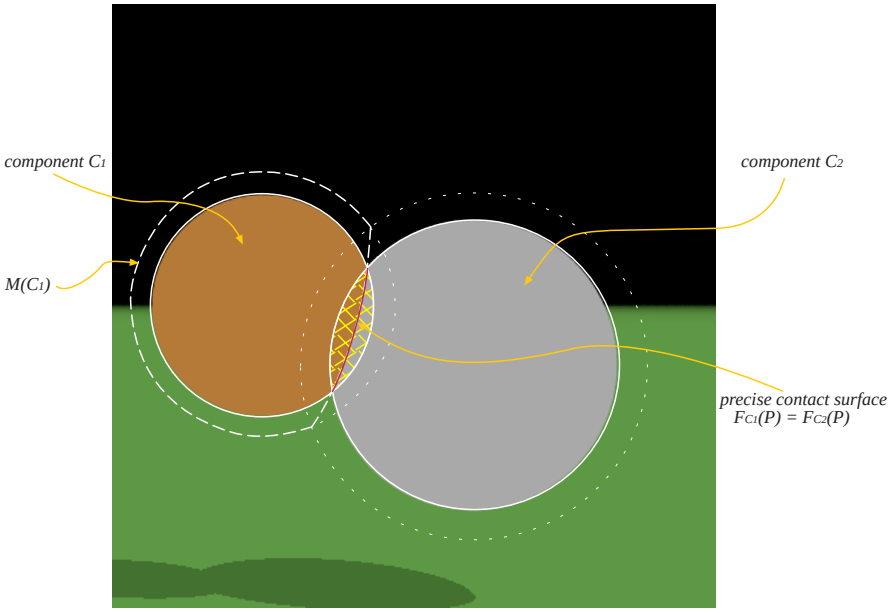


Figure 3.12: *A collision between two objects*

the field value is equal to the undeformed field value for C_1, $F_{C_1}(P)$. Within the cross-hatched region and inside $\mathcal{M}(\mathcal{C}_\infty)$, according to the proposed algorithm, the collision term, $F_{C_2}(P)$ needs to be considered. Therefore, the deformed scalar field for C_1 is:

$$F_{C_{1def}}(P) = F_{C_1}(P) - F_{C_2}(P) + Iso$$

Similarly, the deformed scalar field due to C_2 within the cross-hatched region and inside $\mathcal{M}(\mathcal{C}_\in)$ is equal to:

$$F_{C_{2def}}(P) = F_{C_2}(P) - F_{C_1}(P) + Iso$$

The surface of each deformed component is a set of points for which the scalar field value is equal to the isovalue, $\{P \in \mathcal{R}^\ni : \mathcal{F}(\mathcal{P}) = \mathcal{I}\int\wr\}$. For each component, $F(P)$ in this equation is replaced by its deformed field function. In the cross-hatched area, in which the two components collide, the deformed implicit surface for C_1 is:

$$\{P \in \mathcal{R}^\ni : \mathcal{F}_{\mathcal{C}_\infty}(\mathcal{P}) = \mathcal{F}_{\mathcal{C}_\in}(\mathcal{P})\}$$

and similarly the deformed implicit surface for C_2 is:

$$\{P \in \mathcal{R}^\ni : \mathcal{F}_{\mathcal{C}_\in}(\mathcal{P}) = \mathcal{F}_{\mathcal{C}_\infty}(\mathcal{P})\}$$

Thus, the precise contact surface between the two components is created at:

$$\{P \in \mathcal{R}^\ni : \mathcal{F}_{\mathcal{C}_\infty}(\mathcal{P}) = \mathcal{F}_{\mathcal{C}_\in}(\mathcal{P}) \wedge \mathcal{F}_{\mathcal{C}_\infty} \text{¿} \mathcal{I}\int\wr \wedge \mathcal{F}_{\mathcal{C}_\in} \text{¿} \mathcal{I}\int\wr\}$$

Three objects

A similar reasoning can be performed for more than two objects. Figure 3.13 shows a collision between three objects. The calculation of the deformed scalar field function for component C_1 is illustrated in Figure 3.14. Again, component C_1's set of blendable components is empty, *i.e.* no other components in the scene are blendable with it. Within its maximal influence area $\mathcal{M}(\mathcal{C}_\infty)$ (dashed line in Figure 3.14) two regions in which collision occurs can be identified: areas where there are two or three components influencing the space. The case of two components (cross-hatched regions in Figure 3.14) is calculated as for two objects. In the region where three components influence the space, the area where the surface deforms is where $F_{C_1}(P)$ ¿ Iso. Let us look more closely at this region. If both $F_{C_2}(P) < Iso$ and

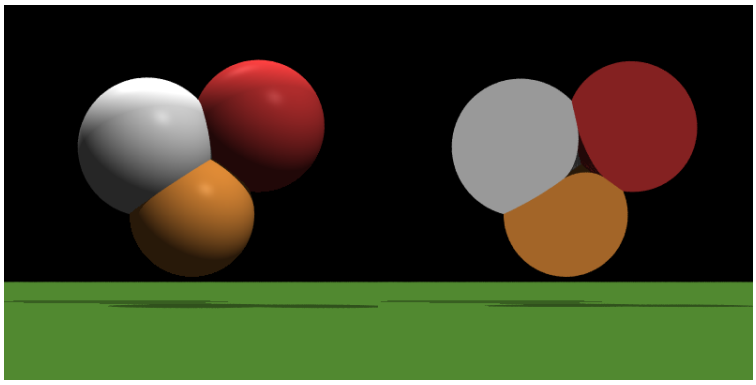


Figure 3.13: *A collision between three objects: 3D and a 2D cross section*

$F_{C_3}(P) < Iso$ (marked in Figure 3.14 with circles), the undeformed field function for C_1, $F_{C_1}(P)$, is taken as the deformed field. If $F_{C_1}(P)$ ¿ Iso or $F_{C_2}(P)$ ¿ Iso (marked in Figure 3.14 with vertical lines), the collision term $F_{C_2}(P) + F_{C_3}(P)$ is considered and the deformed field value becomes:

$$F_{C_1 def}(P) = F_{C_1}(P) - F_{C_2}(P) - F_{C_3}(P) + Iso$$

The deformed surface of C_1 in this region will therefore be the points:

$$\{P \in \mathcal{R}^\ni : \mathcal{F}_{\mathcal{C}_\infty}(\mathcal{P}) = \mathcal{F}_{\mathcal{C}_\in}(\mathcal{P}) + \mathcal{F}_{\mathcal{C}_\ni}(\mathcal{P})$$

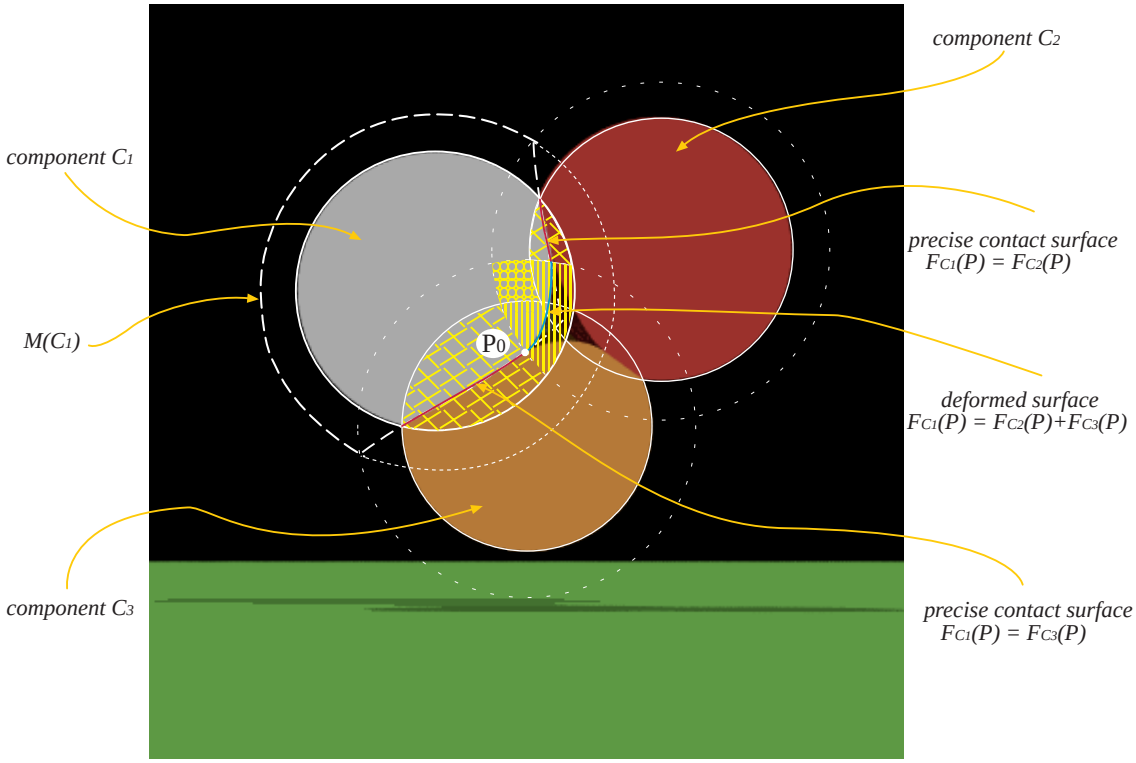


Figure 3.14: *A collision between three objects: diagram*

Surface continuity considerations

Considering the example in Figure 3.14 again, note that the deformed surface is C^1-continuous when it passes from the region with three influencing components to the regions with only two influencing components, *i.e.* in the interior of the collision. C^0-continuity, *i.e.* the positional continuity of the surface, is guaranteed for any field function. C^1-continuity, *i.e.* C^0-continuity and tangent (or first derivative) continuity of the surface, is provided due to the use of a specific field function.

To prove C^0-continuity, let us consider a point P_0 that lies on the contact surface between components C_1 and C_3 and on the radius of influence of C_2 (Figure 3.14). The scalar field value at P_0 due to component C_1 deformed by component C_3, is:

$$F_{C_{1def}}(P_0) = F_{C_1}(P_0) - F_{C_3}(P_0) + Iso$$

After considering component C_2, it becomes:

$$F_{C_{1def}}(P_0) = F_{C_1}(P_0) - F_{C_2}(P_0) - F_{C_3}(P_0) + Iso$$

Since P_0 lies on the radius of influence of C_2, the term $F_{C_2}(P_0) = 0$, thus the deformed surface is C^0-continuous at P_0.

A normal vector $\overrightarrow{N}$ to an implicit surface defined by a function F, is aligned with the gradient, ∇F:

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right)$$

To prove C^1-continuity at P_0, a reasoning similar to the one proving C^0-continuity, can be performed for gradients, yielding continuity of normals. The gradient at P_0 due to C_1 deformed by C_3, is:

$$\nabla F_{C_{1def}}(P_0) = \nabla F_{C_1}(P_0) - \nabla F_{C_3}(P_0)$$

After considering C_2, it becomes:

$$\nabla F_{C_{1def}}(P_0) = \nabla F_{C_1}(P_0) - \nabla F_{C_2}(P_0) - \nabla F_{C_3}(P_0)$$

Thus, to provide C^1-continuity of the deformed surface at P_0, the condition:

$$\nabla F_{C_2}(P_0) = \overrightarrow{0}$$

i.e. $\frac{\partial F_{C_2}}{\partial x}(P_0) = \frac{\partial F_{C_2}}{\partial y}(P_0) = \frac{\partial F_{C_2}}{\partial z}(P_0) = 0$ must hold. Since a field function F is decomposed into a potential function f and distance function r, $F = f \circ r$, each of its partial derivatives can be calculated as a combination of $\frac{\partial f}{\partial r}$ and $\frac{\partial r}{\partial x}$, $\frac{\partial r}{\partial y}$, or $\frac{\partial r}{\partial z}$, *e.g.* for x it is:

$$\frac{\partial F}{\partial x} = \frac{\partial f}{\partial r}\frac{\partial r}{\partial x}$$

The distance function r at the point P_0 is equal to the radius of influence R, $r(P_0) = R$. To obtain the desired equation $\nabla F_{C_2}(P_0) = \overrightarrow{0}$, it is thus sufficient for the partial derivative $\frac{\partial f}{\partial r}$ to be equal to zero at the radius of influence R:

$$\frac{\partial f}{\partial r}(R) = 0$$

This is a property of many potential functions (see Section 2.3). For example, the potential function used here is:

$$f(P) = \left(1 - \frac{r^2}{R^2}\right)^2$$

where $r = \|\overrightarrow{PG}\|$ is the distance to the considered point P and the primitive's generator G and R is a radius of influence. Since

$$\frac{\partial f}{\partial r} = \frac{-4r}{R^2}\left(1 - \frac{r^2}{R^2}\right)$$

and $r(P_0) = R$, it yields:

$$\frac{\partial f}{\partial r}(P_0) = \frac{-4R}{R^2}\left(1 - \frac{R^2}{R^2}\right) = 0$$

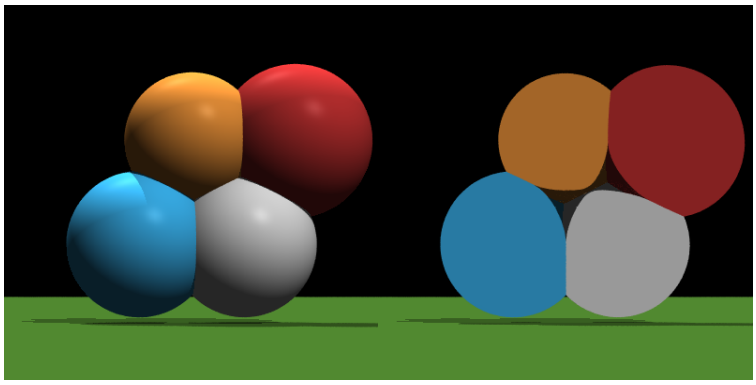


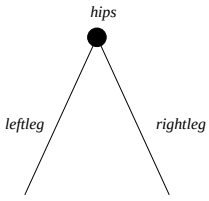
Figure 3.15: *A collision between four objects: 3D and a 2D cross section*

Four or more objects

The proposed algorithm works for any number of objects. For instance, Figure 3.15 presents a collision between four objects. The cross section shows the created contact surfaces. In all collisions C^1-discontinuities occur only on the exterior of a cluster of objects. The deformed surface inside the cluster is C^1-continuous. A method of C^1-smoothing the exterior discontinuities is reported by Marie-Paule Gascuel [Gasc93].

3.3.4 Self-collision for an object

Figure 3.16 shows a simple object composed of three components: one joint and two links. Again, each component contains only one primitive for the sake of simplicity. The links are in collision while blended with the joint. The 2D cross section shows the continuity of the deformed surface. The calculation of deformed scalar field for the joint and for one of the links is presented in the following sections. Surface continuity considerations follow.



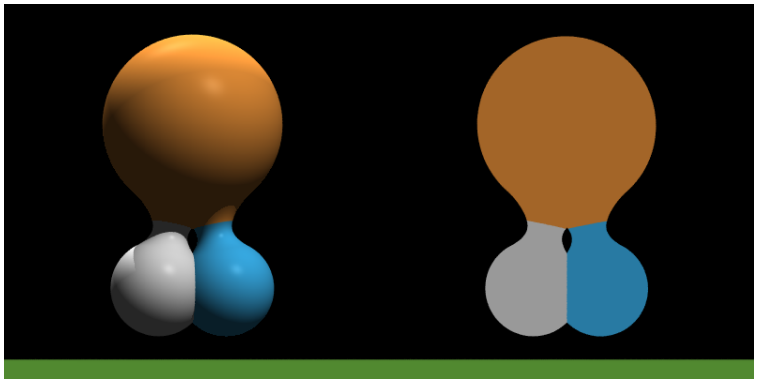


Figure 3.16: *Self collision in an object: 3D and a 2D cross section*

Calculation of a deformed field for a joint

The component that represents the joint (C_1 in Figure 3.17) has an empty set of unblendable components and two components in its blendable set:

$$\mathcal{B}(\mathcal{C}_\infty) = \{\mathcal{C}_\epsilon, \mathcal{C}_\ni\}$$

Outside its maximal influence area $\mathcal{M}(\mathcal{C}_\infty)$ (dashed line in Figure 3.17) the deformed field is set to zero. Within $\mathcal{M}(\mathcal{C}_\infty)$, the surface of C_1 will deform to create a blend between C_1 and its blendable components in

the areas where components C_2 or C_3 influence the space. In these areas the blending term is calculated. C_2 and C_3 are unblendable with each other, thus, according to the proposed algorithm, they will not contribute to the blending term at the same time. C_2 is used in the blending term in the area marked with lines in Figure 3.17 (vertical lines in the area where C_2 is the only influencing component from $\mathcal{B}(\mathcal{C})$, horizontal lines where both C_2 and C_3 influence the space). The deformed scalar field for C_1, $F_{C_{1def}}$, is equal to the sum of its undeformed field, F_{C_1} and the blending term. Thus, when C_2 contributes to the blending term, it is:

$$F_{C_{1def}}(P) = F_{C_1}(P) + F_{C_2}(P)$$

and when C_3 contributes to the blending term, it is:

$$F_{C_{1def}}(P) = F_{C_1}(P) + F_{C_3}(P)$$

Calculation of a deformed field for a link

Figure 3.18 illustrates the deformed field calculation for one of the links, C_2. The set of its blendable components $\mathcal{B}(\mathcal{C}_\in) = \{\mathcal{C}_\infty\}$ and the set of its unblendable components $\mathcal{U}(\mathcal{C}_\in) = \{\mathcal{C}_\ni\}$. Within its maximal influence area $\mathcal{M}(\mathcal{C}_\in)$ (dashed line in Figure 3.18) and outside the area of influence of its blendable component C_1, the collision between C_2 and C_3 occurs as described in Section 3.3.3. Within C_1's area of influence but outside C_3's area of influence (vertical lines in Figure 3.18), the deformed scalar field is the sum of $C_2(P)$ and the blending term which is $C_1(P)$. Inside C_3's area of influence (horizontal lines in Figure 3.18), the collision term, $C_3(P)$, has to be considered, *i.e.* at the areas where both $F_{C_2}(P)$ and $F_{C_3}(P)$ are greater than isovalue, the deformed scalar field becomes:

$$F_{C_{2def}}(P) = F_{C_2}(P) + F_{C_1}(P) - F_{C_3}(P) + Iso$$

Otherwise, only the sum of the undeformed field function for C_2 and the blending term is taken:

$$F_{C_{2def}}(P) = F_{C_2}(P) + F_{C_1}(P)$$

Surface continuity considerations

A naive way of calculating scalar field deformation during a self-collision is to first apply the relevant collision terms to all unblendable components and then add the influence from blendable components. Thus, the deformed field value for a component C would be:

$$F_{Cdef}(P) = F_C(P) - \sum_{C_u \in \mathcal{U}(\mathcal{C})} F_{C_u}(P) + \sum_{C_b \in \mathcal{B}(\mathcal{C})} F_{C_b}(P) + Iso$$

Applying this algorithm in the case illustrated in Figure 3.16 would result in the following calculation of deformed components C_1, C_2 and C_3 in the intersection of their areas of influence:

$$\begin{aligned} F_{C_1def}(P) &= F_{C_1}(P) + F_{C_2}(P) + F_{C_3}(P) \\ F_{C_2def}(P) &= F_{C_2}(P) - F_{C_3}(P) + F_{C_1}(P) + Iso \\ F_{C_3def}(P) &= F_{C_3}(P) - F_{C_2}(P) + F_{C_1}(P) + Iso \end{aligned}$$

This calculation differs from the proposed algorithm only for C_1, since its blending term is now always the sum $F_{C_2}(P) + F_{C_3}(P)$. For C_1, the field function is therefore calculated as if all three components were blendable with each other. Hence, the created implicit surface passes through the intersection of C_1's maximal influence area and the surface $F_{C_1}(P) + F_{C_2}(P) + F_{C_3}(P) = Iso$. The calculation for C_2 and C_3 has not changed, and the isosurface in their maximal influence areas goes through the union of $F_{C_1}(P) + F_{C_2}(P) = Iso$ and $F_{C_1}(P) + F_{C_3}(P) = Iso$. Figure 3.19 shows the resulting C^0-discontinuity.

In the proposed algorithm the blending term for C_1 is restricted to only field values from components that are greater than the components in their unblendable sets. In this way the resulting implicit surface for C_1 also passes through the union of $F_{C_1}(P) + F_{C_2}(P) = Iso$ and $F_{C_1}(P) + F_{C_3}(P) = Iso$ providing C^0-continuity.

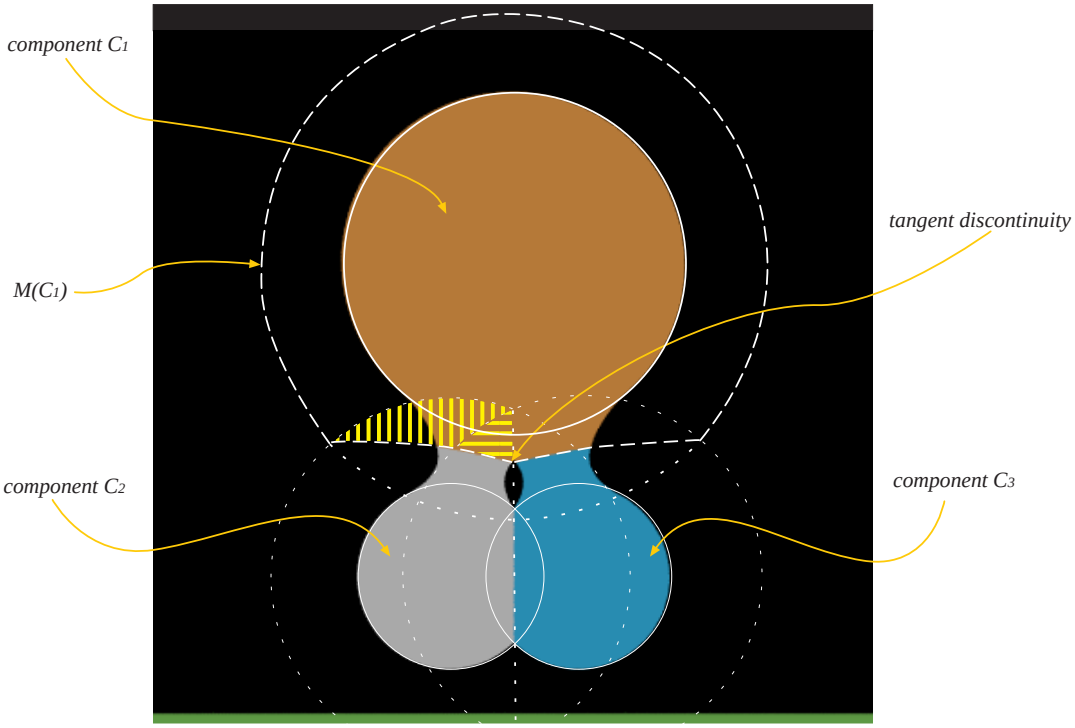


Figure 3.17: *Self collision in an object: deformation for a joint*

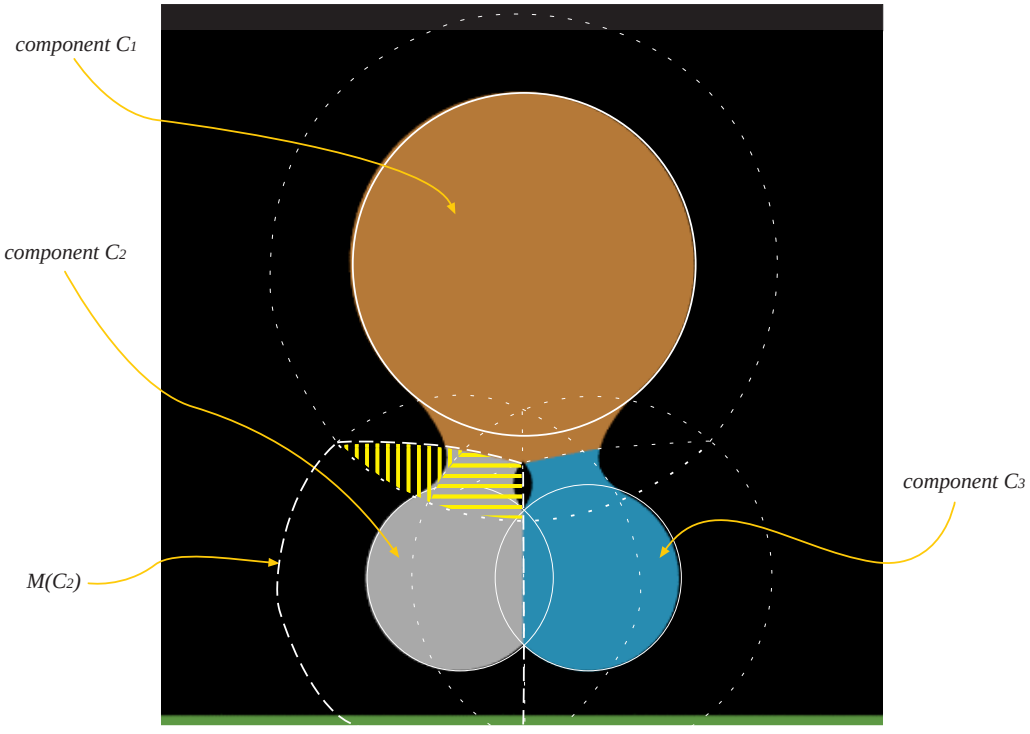


Figure 3.18: *Self collision in an object: deformation for a link*

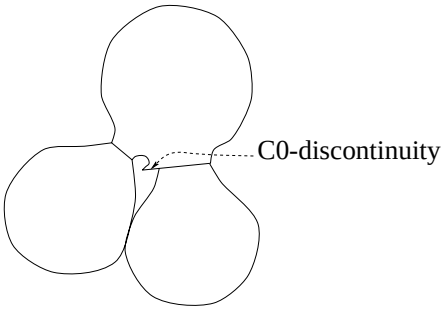


Figure 3.19: *C^0-discontinuity resulting from adding all blendable components to the blending term*

3.3.5 More complex cases of collisions and self-collisions

Figure 3.20 shows an example of a self-collision for an object with more complex components. An appearance graph for the model consists of three components: a joint *hip* and two links *legs*. The hip contains one primitive and there are six primitives in each leg. When unwanted blending is not prevented, the legs blend with each other (left). When considering the default blending properties for the appearance graph, the undesired blending between the legs is prevented (middle and right).

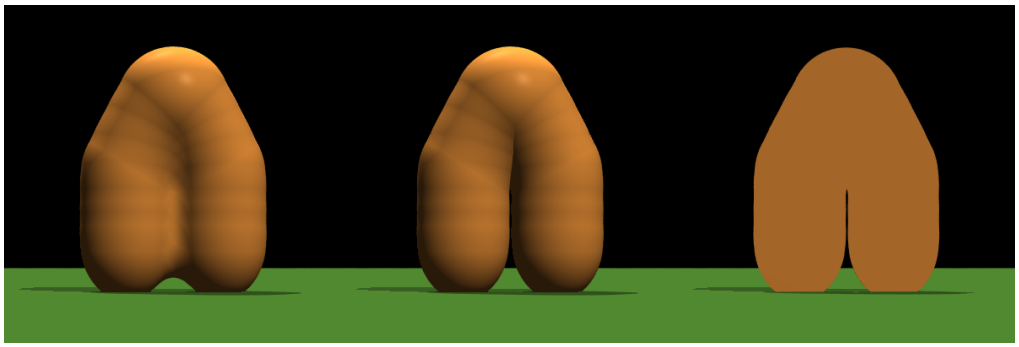


Figure 3.20: *Another example of self collision, from left to right: all primitives blend, a collision between two "legs", a 2D cross section of the collision*

Figure 3.21 shows a scene composed of two colliding objects, each modelled using a three component appearance graph: one joint and two links. In the top sequence, no self-collision was calculated, thus the links in both objects blend which is an undesired effect that destroys the intended appearance of the objects. The bottom sequence is calculated using the proposed algorithm, *i.e.* taking into consideration the collision between the links and avoiding unwanted blending.

3.4 Coherence

So far, the appearance of characters modelled was not destroyed by unwanted blending. Ways of ensuring the topological integrity of the model during animation, *i.e.* the problem of coherence loss, will now be looked at.

3.4.1 Coherent implicit surfaces

One of the features of implicit surfaces is that local surface deformations can be modelled by moving some of the primitives in a model. This property inspires the idea of simulating clay-like behaviour for implicit objects, in which the goal is to squash and stretch an object by moving the primitives and let the system take care of the object's topological integrity.

To avoid coherence loss, the distance between primitives has to be restricted to eliminate surface separation (maximum distance) and primitive overlapping (minimum distance). Since all the distance values

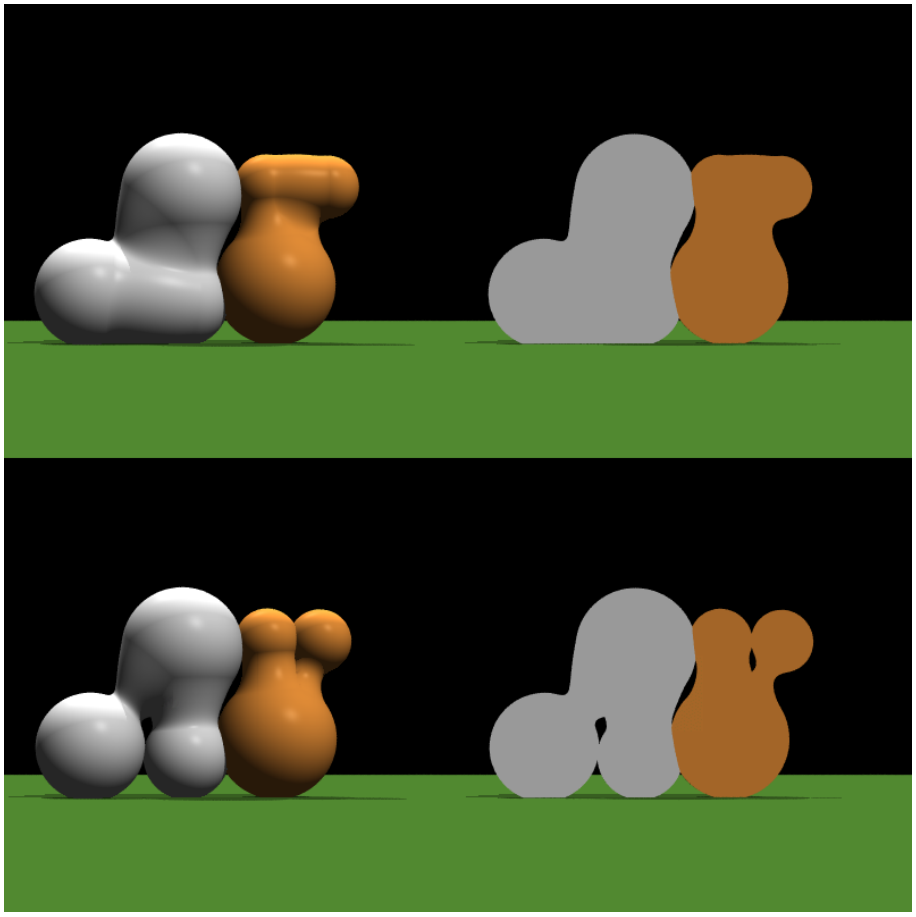


Figure 3.21: *A collision between two objects, each in a state of a self-collision: 3D and a 2D cross section. Top: no self-collision is calculated (unwanted blending destroys appearance), Bottom: the same collision with self-collisions (unwanted blending prevented)*

between minimum and maximum are allowed, a connection between two primitives is not rigid but deformable causing a local deformation of the model without coherence loss. By controlling the values of minimum and maximum distance between primitives various levels of *firmness* of an object can be modelled. A firmer object will squash and stretch less than a looser one.

Two primitives, between which distance constraints are maintained, will be called a *connection*. In a model constructed according to an appearance graph, two types of connections can be identified: internal to an object component and external to it, between primitives from two object components (Figure 3.22). The former ensures the coherence of a component, the latter provides bonds between parts of a model. When a primitive moves, it is first "followed" by remaining primitives from the same component and then by primitives from the adjacent components. Distance constraints yield topological integrity of the model, therefore coherence loss does not occur.

To explain the proposed solution, the process for defining and controlling distance constraints between two primitives, including firmness specification, will be presented first. Then, a way of imposing them within an object component and between components will be described. Finally, the results will be shown in a series of examples.

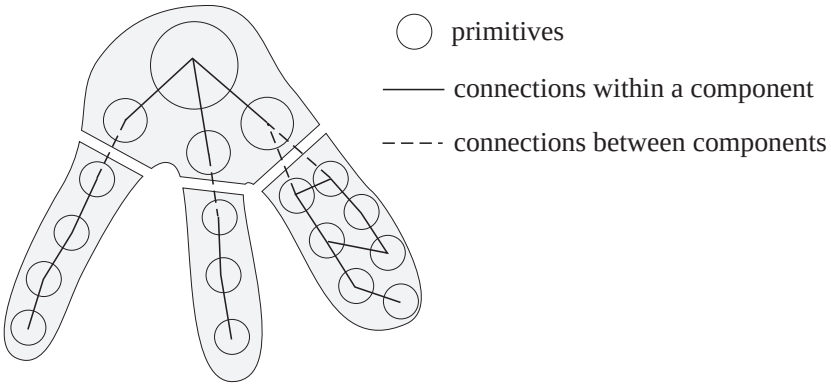


Figure 3.22: *Two types of connections in a model: within a component and between components.*

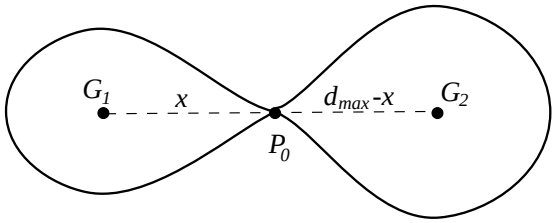


Figure 3.23: *The maximum coherence preserving distance between two primitives*

3.4.2 Distance constraints

Two blendable primitives stay blended with each other as long as for all points along the shortest line segment connecting their generators G_1 and G_2:

$$F_1(P) + F_2(P) \text{ ¿ } Iso$$

where F_1, F_2 are the field functions of the two primitives and Iso is the isovalue. The maximum distance at which the two primitives stay connected is when the minimum field value between the two generators is equal to Iso. The primitives are then connected by exactly one point P_0 (Figure 3.23). Let us mark the distance between the two generators as d_{max} and the distance between P_0 and G_1 as x. The distance between P_0 and G_2 is then $d_{max} - x$. In this situation, the potential functions f_1, f_2 of the two primitives and their first derivatives $\frac{d}{dr}f_1, \frac{d}{dr}f_2$ satisfy the following equations:

$$\begin{cases} f_1(x) + f_2(d_{max} - x) = Iso \\ \frac{d}{dr}f_1(x) + \frac{d}{dr}f_2(d_{max} - x) = 0 \end{cases}$$

The first equation ensures that the field value at P_0 is equal to the isovalue and thus the point P_0 lays on the blended surface. The second equation ensures that the first derivative of the field function is zero and thus the field function takes a minimum at P_0. The point P_0 is therefore the point with the lowest field value between the two generators and the only point of connection between the two primitives. To avoid the costly calculation of the solution to these equations, its approximation can be used. Instead of looking for the distance at which two primitives are connected by one point, a distance that guarantees that there are connected can be computed in an easy way. Consider the situation in Figure 3.24. The dotted circles represent the two primitives at their radii in isolation, $r1_{iso}, r2_{iso}$. The radius in isolation for a primitive can be calculated by solving the equation:

$$f(r_{iso}) = Iso$$

where f is the potential function. The field value due to one primitive for distances smaller than the radius in isolation is greater than the isovalue. For the two primitives in Figure 3.24 it is therefore sufficient to ensure that both field values $F_1(P)$ and $F_2(P)$ are greater than $\frac{Iso}{2}$ for any point P along the line segment

marked x. The radius at which the field value is equal to $\frac{Iso}{2}$ can be calculated for a primitive by solving the equation:

$$f(r_{\frac{Iso}{2}}) = \frac{Iso}{2}$$

where f is the potential function. To ensure that field values due to both primitives are greater than $\frac{Iso}{2}$, the minimum of the two distances, $r1_{\frac{Iso}{2}}$ and $r2_{\frac{Iso}{2}}$ has to be taken.

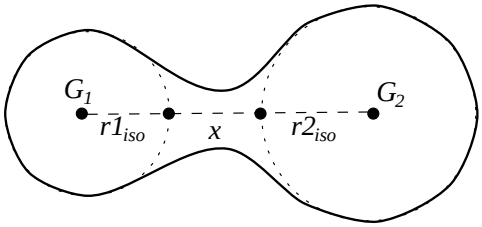


Figure 3.24: *An approximated coherence preserving distance between two primitives*

The resulting approximated distance will be called *maximum coherence distance*:

$$d_{max} = \min\left\{ f_1^{-1}\left(\frac{Iso}{2}\right), f_2^{-1}\left(\frac{Iso}{2}\right)\right\} + f_1^{-1}(Iso) + f_2^{-1}(Iso)$$

where f_1, f_2 are the potential functions of the two primitives. d_{max} gives the maximum separation distance allowed for the two primitives. When one primitive moves away, the other should follow its motion when the distance between them reaches d_{max}. Similarly, the *minimum coherence distance*, d_{min}, between two primitives needs to be defined to handle situations when a primitive moves towards the other one and to prevent generators from overlapping and thus primitives from traversing through each other. $d_{min} = 0$ is proposed. The second primitive should follow the motion when the distance between them is equal to d_{min}.

The minimum and maximum coherence distances mark the limits of overlapping and blending respectively. In order to control the manner of motion propagation restricting the distance between two primitives to a subinterval of $[d_{min}, d_{max}]$ is required. The notion of *firmness* specifies such an interval. Assuming that the initial distance between two primitives, d_{init}, is within their corresponding minimum and maximum coherence distances, two parameters specify firmness: $f_{min} \in [0,1]$ and $f_{max} \in [0,1]$. The distance to be maintained between the primitives is given by $[R_{min}, R_{max}]$, where

$$\begin{aligned} R_{min} &= d_{min} + f_{min}(d_{init} - d_{min}) \\ R_{max} &= d_{max} + f_{max}(d_{init} - d_{max}) \end{aligned}$$

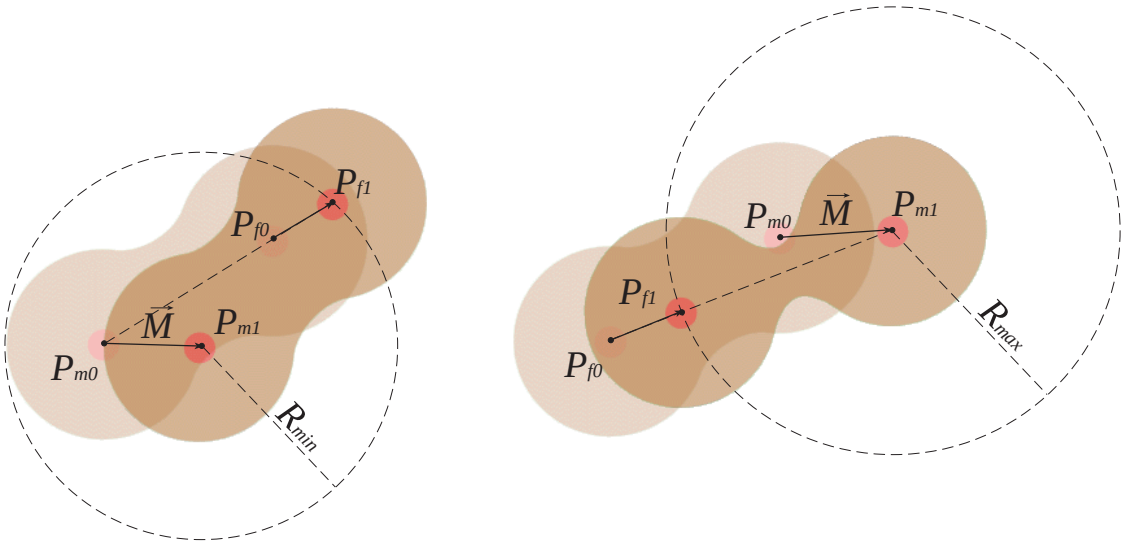
The two parameters represent the "squashability", *i.e.* how much the two primitives can approach each other (f_{min}) and "stretchability", *i.e.* how much they get get away from each other (f_{max}) of the relationship between the two primitives. The higher the value of either of the two parameters, the less deformation of the initial configuration between the primitives is allowed and therefore, the less fluid the motion of an object which contains the pair.

Figure 3.25 presents how the distance constraints are maintained in a connection (*push-pull action*). Two primitives, one moving (P_m) and one following (P_f), are shown before (lighter shading) and after (darker shading) the motion $\overrightarrow{M}$ of P_m. Which of the actions occurs (push or pull) is determined by calculating the dot product of the vector between the primitives $\overrightarrow{P_{f0}P_{m0}}$ and the vector of motion $\overrightarrow{M}$ and examining the distance resulting between two primitives after the movement of P_m from P_{m0} to P_{m1}, $\|\overrightarrow{P_{f0}P_{m1}}\|$.

When

$$\overrightarrow{P_{f0}P_{m0}} \cdot \overrightarrow{M} < 0 \wedge \|\overrightarrow{P_{f0}P_{m1}}\| < R_{min}$$

the primitive P_f is *pushed*. It will move along the vector $\overrightarrow{P_{m0}P_{f0}}$ until the distance between the primitives becomes R_{min}. The new position P_{f1} is calculated by a ray-sphere intersection algorithm [Glas89], taking a sphere centred at P_{m1} with the radius R_{min} and intersecting it with a ray starting at P_{f0} in the direction of $\overrightarrow{P_{m0}P_{f0}}$.



"push" "pull"

Figure 3.25: *A distance constraint maintained between two primitives: light shading represents old positions of primitives, dark shading gives new positions; M is the direction of motion of one primitive which then causes a movement of the connected primitive*

When

$$\overrightarrow{P_{f0}P_{m0}} \cdot \vec{M} \text{ ¿ } 0 \wedge \|\overrightarrow{P_{f0}P_{m1}}\| > R_{max}$$

P_f is *pulled*. It will move along the vector $\overrightarrow{P_{f0}P_{m1}}$ until the distance between the primitives becomes R_{max}. Again, a ray-sphere intersection algorithm is used to calculate the new position for P_f, taking a sphere centred at P_{m1} with the radius R_{max} and intersecting it with a ray starting at P_{f0} in the direction of $\overrightarrow{P_{f0}P_{m1}}$.

3.4.3 Connections within a component

An object component consists of a collection of primitives. It is therefore necessary to define a network of connections to be maintained between them. A naive approach would be to attempt the distance constraint enforcement between each pair of primitives using a full graph of connections. However, this process is compute intensive. Here, an alternative, user-defined network of connections that fits the particular requirements of an animator is proposed. A network has to contain all the primitives of the component and there has to exist a path in the graph between each pair of primitives. At the moment the connections have to be specified manually, no scripting language is provided. Automatic creation of simple linear connections between all the primitives in each component, in the order of their specification in the script file, is implemented but no support exists for more complex networks.

Figures 3.26, 3.27 and 3.28 show examples of networks defined within a component and the behaviour of the component during motion of one of the primitives. The induced motion of the component depends on the number of children in the connection network of the primitive causing the motion. The more connections, the more immediate the motion of the component. In Figures 3.26, 3.27 and 3.28 the number of connections starting in the moved primitive increases resulting in a more and more immediate motion.

Firmness parameter is constant for all connections within a component. Figures 3.29 and 3.30 show examples of the same motion propagated in a component with the same network of connections for varying values of firmness. In Figure 3.29, firmness parameters are $f_{min} = 0.9$ and $f_{max} = 0.9$. The motion of the component after pulling one of the primitives results in the primitives staying close to each other, the component is “firmer”. When the firmness is lowered by decreasing f_{max} down to $f_{max} = 0.2$, a more fluid motion of the component is achieved in the same situation (Figure 3.30).

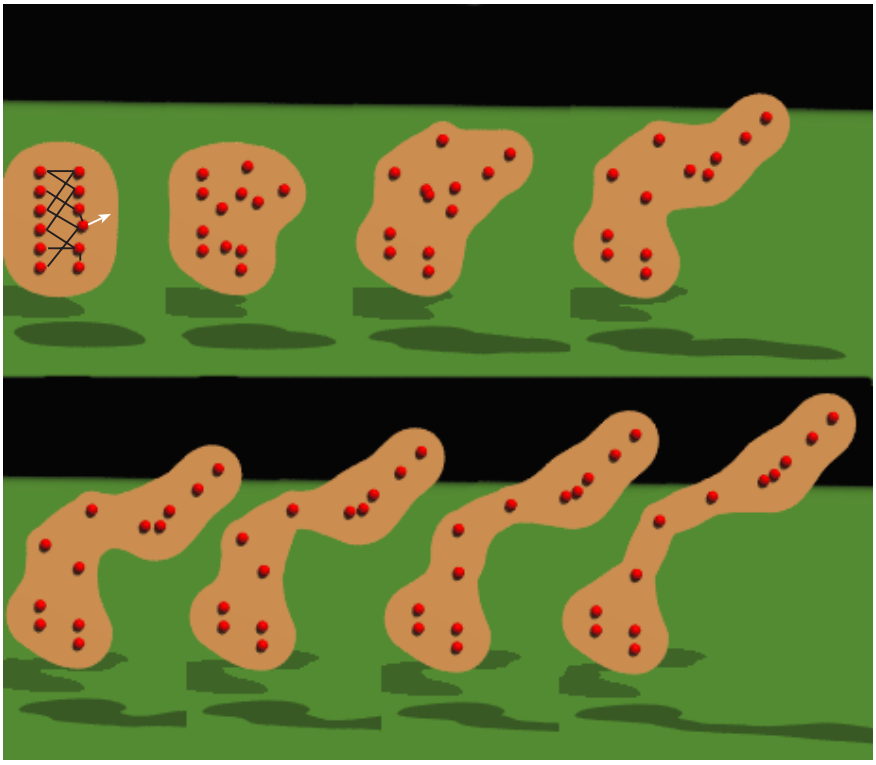


Figure 3.26: *Coherence maintained within a component for firmness given by* $f_{min} = 0.9, f_{max} = 0.5$ *for a network of connections shown in the first frame*

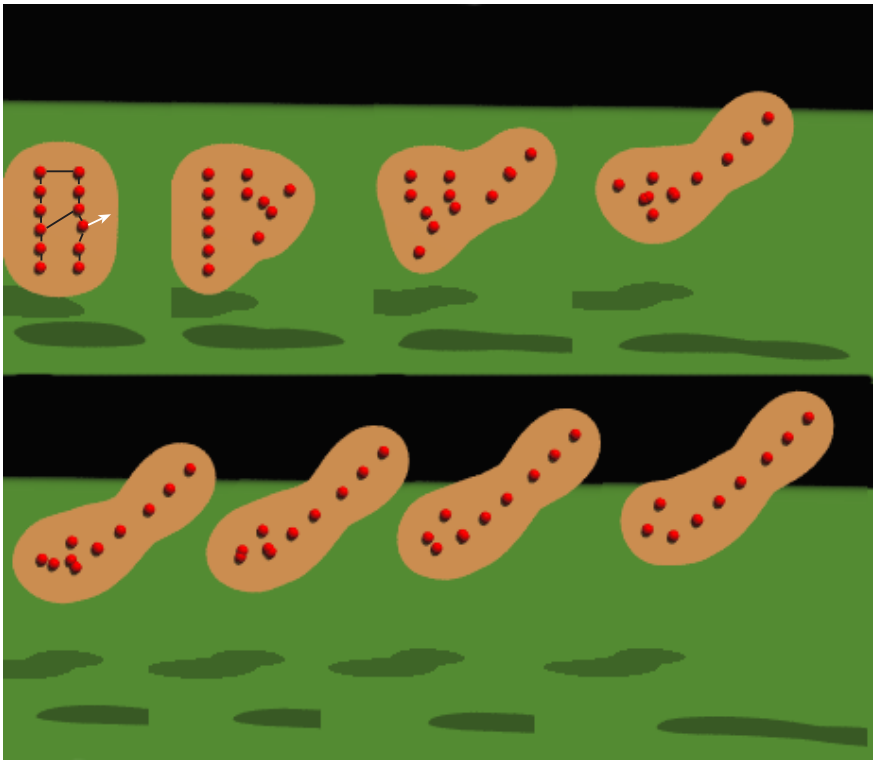


Figure 3.27: *Coherence maintained within a component for firmness given by* $f_{min} = 0.9, f_{max} = 0.5$ *for a network of connections shown in the first frame*

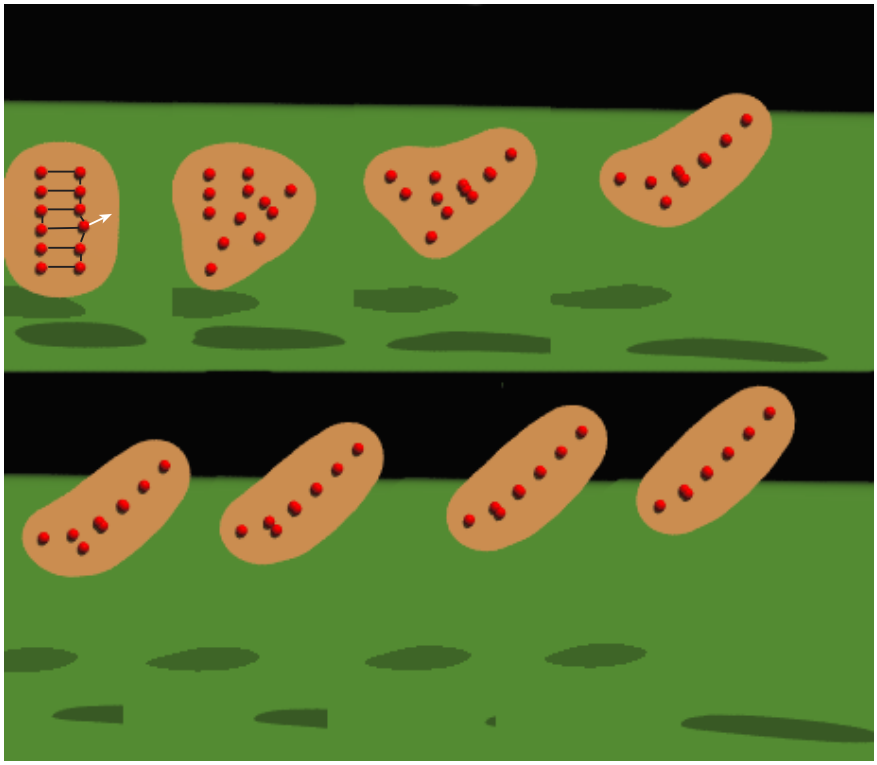


Figure 3.28: *Coherence maintained within a component for firmness given by* $f_{min} = 0.9, f_{max} = 0.5$ *for a network of connections shown in the first frame*

Figure 3.31 shows three consecutive motions applied to a component. Each row in the figure corresponds to one primitive being translated. The component behaves like a chunk of clay while being stretched and squashed in various directions without breaking into pieces.

An assumption is made that the initial network of primitives within a component preserves its coherence, *i.e.* that the initial distance in each connection is within the minimum and maximum coherence distances. This condition has to be ensured by the user during the modelling stage. A warning is issued by the system if a model does not meet this requirement, *i.e.* if there is a connection between two primitives, with generators G_i, G_j, for which $\|\overrightarrow{G_iG_j}\| > d_{max}$.

The motion starts from one moving primitive and is propagated within the connection network during its traversal. Thus, when a primitive is moved in a component, its neighbours in the network are examined and they follow its motion when the new distance between them falls outside the corresponding $[R_{min}, R_{max}]$ limits. The process of motion following is described in Section 3.4.2 (push-pull action). The induced motion of primitives finishes when the connection network has been traversed, *i.e.* when each primitive has been visited. Note that some distance constraints may thus remain unsatisfied since for a network with cycles the traversal will be completed without verifying all connections. For instance, in Figure 3.31 the bottom row of frames shows the network of connections after the first two movements. Some of the primitives have been allowed to remain further from each other than the coherence preserving distance and connections between them have been breached. However, since all primitives are in the network and there is a path between each pair of primitives in it, each primitive has at least one neighbour to which it stays blended. The coherence of the component is therefore preserved.

3.4.4 Connections between components

To provide a general method of propagating the induced motion in the appearance graph (between components), it is required to find the primitive in each adjacent component that is closest to the deformed component and start the motion propagation from that primitive. However, a simpler way is to predefine

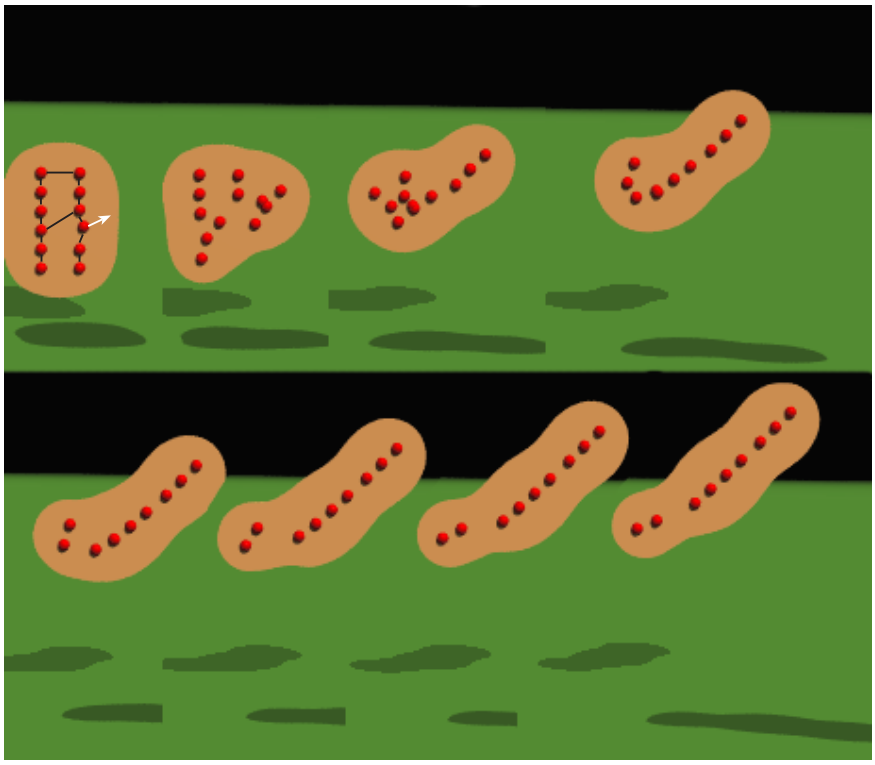


Figure 3.29: *Coherence maintained within a component for firmness given by* $f_{min} = 0.9, f_{max} = 0.9$*, a "firmer" component*

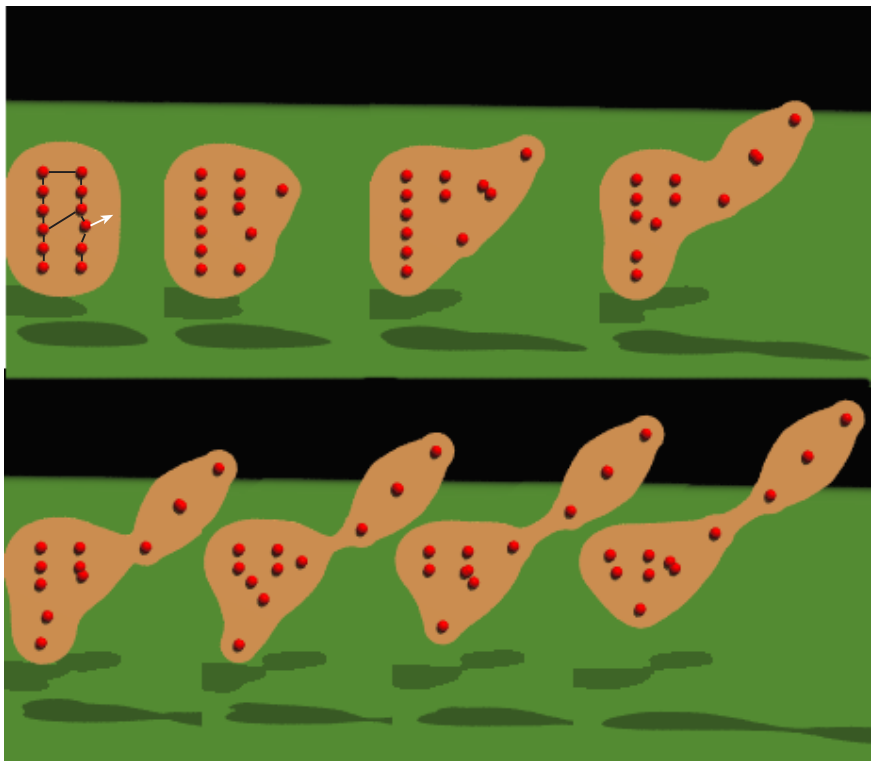


Figure 3.30: *Coherence maintained within a component for firmness given by* $f_{min} = 0.9, f_{max} = 0.2$*, a "looser" component*

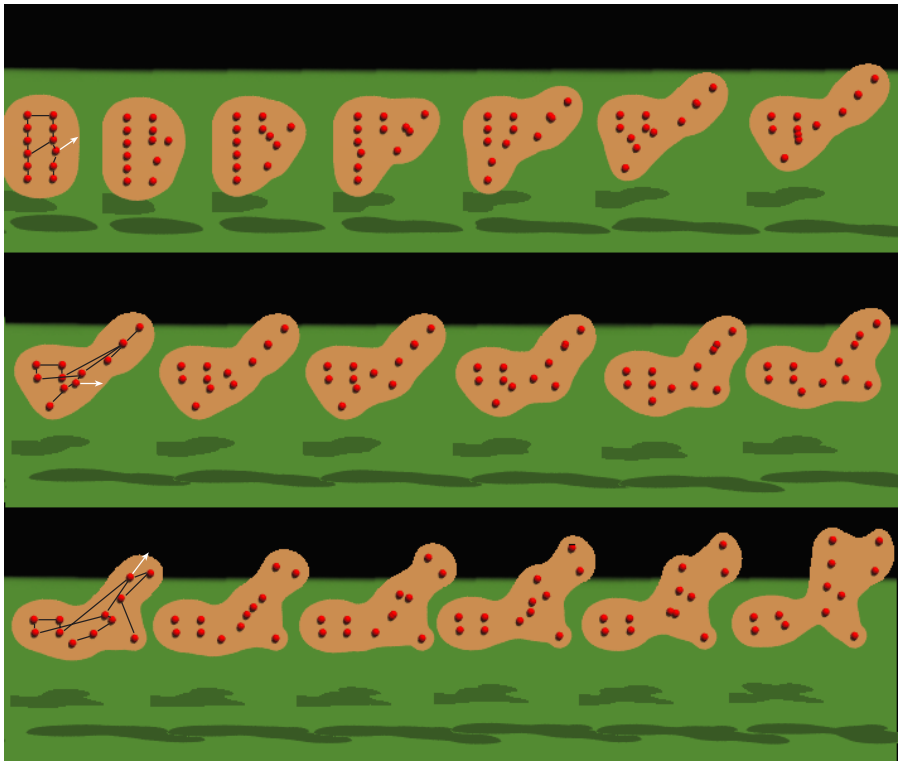


Figure 3.31: *A component being deformed by three consecutive motions applied to it*

the connections between primitives from adjacent components. Such connections are user defined and they can be fitted to the particular requirements of a model. With connections specified this way, the same propagation algorithm as for connections within a component can be applied. Figure 3.32 shows a simple model during deformation due to motion propagation. The character is composed of one joint (*body*) with three links (*legs*) attached to it. Motion starts by moving a primitive in one of the legs. The distance constraint in the connection between the body and this leg cause the body to follow this motion. Then, the remaining two legs follow since there are connections between them and the body. As a consequence the entire model is pulled in the up-right direction. To specify this sequence, only one translation had to be given which is an important advantage of the coherence preservation algorithm.

3.5 Tripod - a case study

In this example, the process of creating a short animation sequence using the ABC approach with one simple character, Tripod, is presented. Tripod has three *legs*, each consisting of two primitives, connected by distance constraints (Figure 3.33). Its *trunkhead* is composed of one primitive. The *trunkhead* is a joint and the three *legs* are links are attached to it by distance constraints. The firmness parameters are $f_{min} = 0.9$ and $f_{max} = 0.8$.

Having defined the model, an animation sequence is created by moving one primitive at a time (Figure 3.34). The aim was to animate the Tripod during a jump. The first step was to prepare the character for a jump by bending two of its *legs* (Frames 00-10) and lowering its *trunkhead* (Frames 10-15). The jump is then created by translating the *trunkhead* in the upper right direction (Frames 15-40). During the upward motion, Tripod's *legs* approach each other and collide. For instance, in Frame 40 the *legs* a self-collision is modelled. Precise contact surface between the legs is created. This effect would have been lost without preventing unwanted blending between legs. Also, the effects of squash and stretch can be seen during the animation - the *legs* lengthen during the upwards motion and shorten during the descend. When Tripod reaches its highest position (Frame 40) it is pulled down by translating the *right leg* down (Frames 40-50). It approaches the floor leaning to the right side and its balance is retrieved by pulling the *left leg* down left

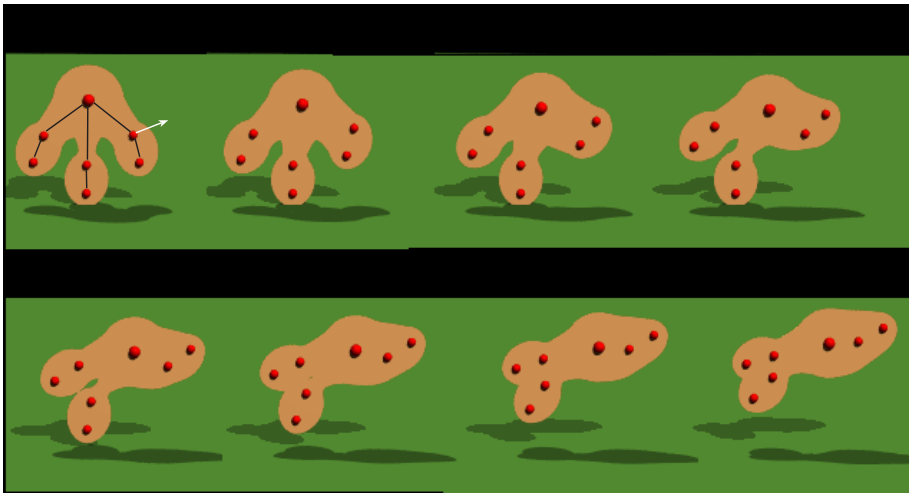


Figure 3.32: *Propagation of motion in a graph*

and forward (Frames 50-70). Finally, the Tripod returns to its upright position by pulling the *trunkhead* upwards (Frames 70-75).

The more detailed description of this animation is as follows (primitive numbers refer to Figure 3.33):

Motion	Translation
Frames 00-05 *right leg* up	$p_4 : (0.1, 0, 0.05)$
Frames 05-10 *left leg* up	$p_0 : (-0.1, 0, 0.05)$
Frames 10-15 *trunkhead* down	$p_6 : (0, 0, -0.5)$
Frames 15-40 *trunkhead* up right	$p_6 : (0.1, 0, 0.15)$
Frames 40-50 *right leg* down	$p_4 : (0, 0, -0.2)$
Frames 50-70 *left leg* down left forward	$p_0 : (-0.05, -0.05, -0.15)$
Frames 70-75 *trunkhead* up	$p_6 : (0, 0, 0.1)$

Tripod is modelled in a “plasticine” manner. The deformation due to all movements is plastic, *i.e.* it lasts in time. For instance, when Tripod lands on the floor, its body does not return to its initial position but keeps the shape resulted from the motion.

Collisions with the floor is not calculated with the implicit model. They are approximated by maintaining all generators at a distance from the floor level by at least their radius in isolation. Precise collisions could have been modelled if a large primitive was used to model the floor.

3.6 Summary

This chapter has presented implicit surfaces with blending properties that can be used to model objects capable of collisions and self-collisions. The new formulation produces contact surfaces between objects and between parts of one object. A simple general deformation algorithm has been proposed. It is valid for any configuration of blending properties in a model. The algorithm guarantees C^0-continuity of the deformed surface. Higher level continuity and volume preserving deformation are two extensions the algorithm would benefit from. For C^1-continuity, the work of Marie-Paule Gascuel [Gasc93] could be followed. Volume conserving deformation was investigated by Desbrun and Marie-Paule Gascuel [Desb95a].

The main advantage of processing collisions using implicit surfaces is the simplicity of collision detection and creation of a precise contact surface between colliding objects. Therefore, no surface interpenetration or occurrence of surface deformation before contact will take place.

Another important technique developed in this chapter is a coherence preserving algorithm that allows a user to animate a model by simple translation of individual primitives and calculating the resulting motion of remaining primitives in the object so that the topological integrity of the model is preserved. The algorithm,

```
iso 0.8

object tripod colour Gold
    component left rotation (45,0,-20)
        point (0,0.5,0) POVfunction 2 1
        point (0,-1,0) POVfunction 2 1
        blend SUM
    component middle rotation (-45,0,0)
        point (0,0.5,0) POVfunction 2 1
        point (0,-1,0) POVfunction 2 1
        blend SUM
    component right rotation (45,0,20)
        point (0,0.5,0) POVfunction 2 1
        point (0,-1,0) POVfunction 2 1
        blend SUM
    component trunkhead rotation (0,0,0)
        point (0,0,0) POVfunction 2.5 2
        blend SUM
    joint trunkhead attachedto left (1.5,2.2,1.5)
    joint trunkhead attachedto middle (0,2.2,-1.5)
    joint trunkhead attachedto right (-1.5,2.2,1.5)
    anchor trunkhead (1,1.8,0)

end
```

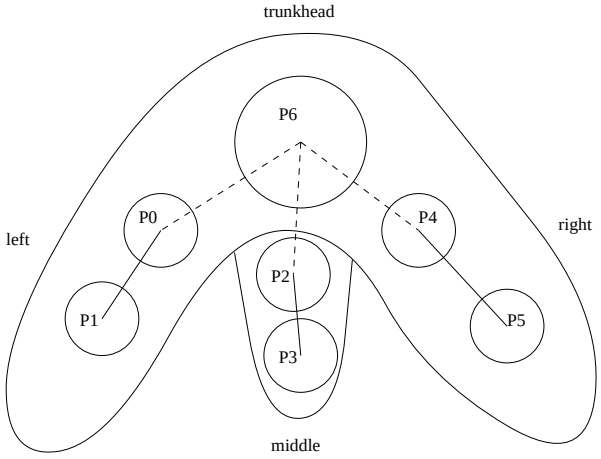


Figure 3.33: *A model of Tripod with a connection network*

based on simple geometric principles, is computationally inexpensive to implement and produces good results. The calculated motion can be adjusted using an intuitive parameter of firmness. A firmer object will deform less than a looser one. The coherence preserving algorithm would benefit from a deformation method that maintains the volume of an object at a constant level and from a more interactive manner of defining a connection network.

The methods presented offer tools to model deformable “clay” or “plasticine” characters that can easily be squashed and stretched to create expressive animations. A user is given simple manual control over the behaviour of an object. Good results can be achieved for simple characters (*e.g.* Tripod consisting of 7 primitives). However, for more complex models, specifying an animation by moving one primitive at the time becomes tedious and not intuitive. For instance, the behaviour of a model for two primitives being moved at the same time is not defined. The next chapter will investigate using a combination of manual and automatic animation control to achieve a goal of *animating* implicit surfaces in the traditional character animation sense.

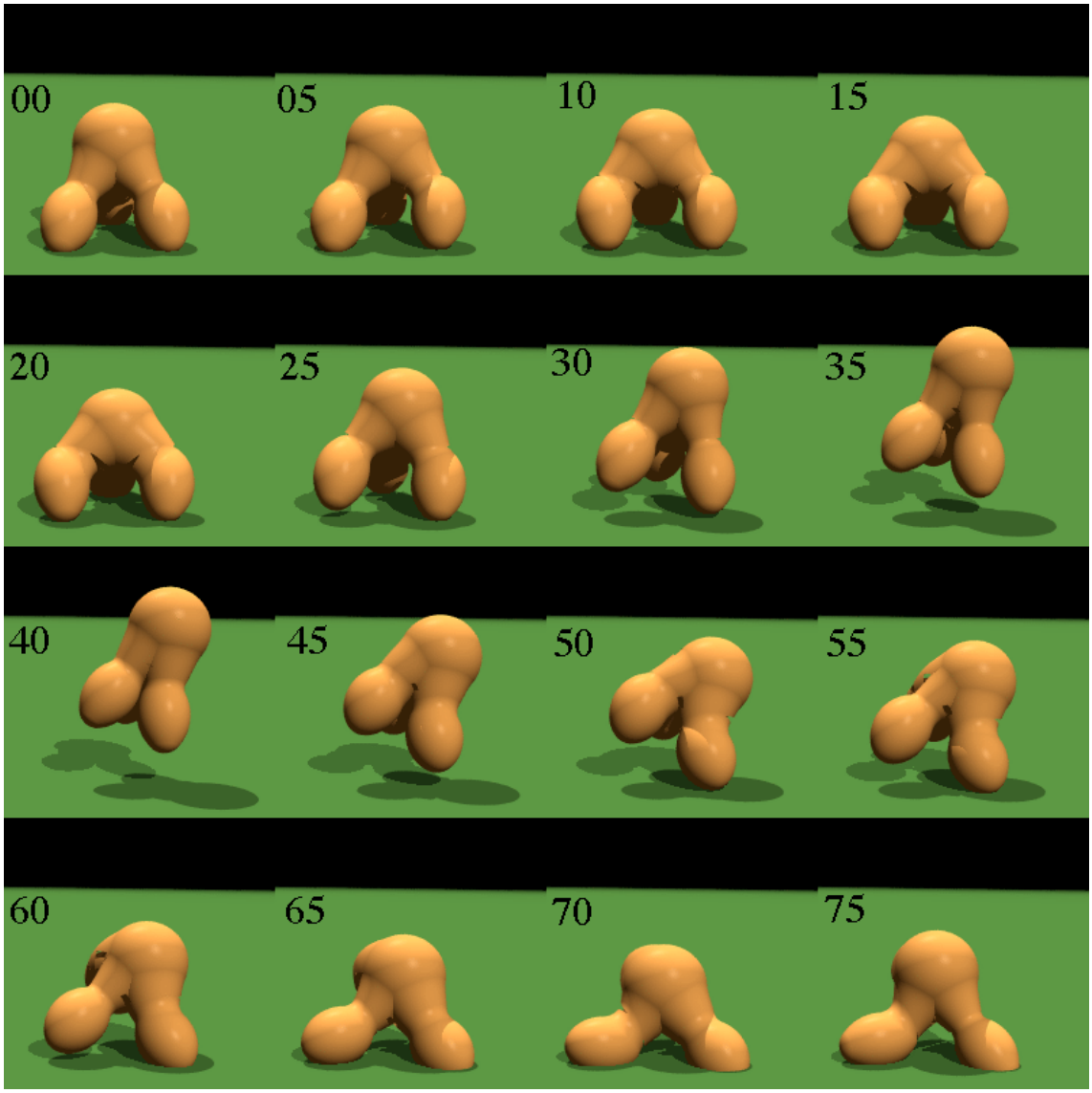


Figure 3.34: *Tripod jumping*

Chapter 4

Animating Implicit Surfaces

4.1 Introduction

Chapter 3 introduced an extended model for implicit surfaces with blending properties, in which clay-like material can be modelled and deformed in a coherent way. This model makes use of several advantageous properties of implicit surfaces: their ability to conform to the objects in the environment (precise contact modelling), their naturally layered structure (generators and an isosurface) which automatically provides a "skin" for an object, and their deformability, *i.e.* the fact that, within certain limits, primitives can be pulled away and pushed towards each other, locally changing the shape of the surface, without the loss of its continuity. Such effects are often required in character animation and more difficult to achieve using polygonal or parametric surface models. This model offers deformable implicit surfaces that behave in an "organic" way. The next goal is to "breathe life" into such material, *i.e.* to *animate* it in the traditional animation sense.

The experience of traditional character animators, collected over the years, has been condensed into a list of principles of traditional animation [Thom81]. One of the most often stressed rules is "squash and stretch". Characters show their reactions and feelings by deforming to the limits of these two extremes. The animated world is in constant motion. All movements are fluid and all physical laws exaggerated, with each action preceded by its anticipation to prepare an audience for it. These are the main principles of traditional animation. They make us forget that this is only an illusion of life. These principles are the inspiration for the work on *animating* implicit surfaces described in this chapter.

In computer generated character animation, the level of abstraction at which the animator interacts with the software, is an important issue. At the lowest level, all parts of a character have to be positioned and oriented in space and their possible deformation specified. When applied to a more complex model, the process may prove very tedious. Thus, animators would not be provided with a tool that gave the necessary level of creative spontaneity. Therefore, some parts of motion specification should be generated by the computer animation system. On the other hand however, animation control cannot be taken completely out of the animator's hands and automated, since it would not normally produce the exact required motion. Assistance with motion generation is desired but fine tuning of detail should be left to the animator. The manual motion control, used in Chapter 3, provides mechanisms to animate models by moving single primitives. The motion for all the primitives but the moving one is calculated by the coherence preserving algorithm (Section 3.4). Nevertheless, it is a low level animation method and for models built with many primitives, such an approach is difficult to apply. This chapter will show how manual and automatic motion control can be combined to provide a hybrid animation method for implicit surfaces with blending properties aiming at creating automatic traditional animation effects.

There are several similarities between choreographing a dance sequence and creating an animation sequence. Both processes start from a vague idea, sketch the general shape of the piece, choose their performers, develop phrases to express intended feelings and refine them until the composition is completed. This correspondence was the inspiration for the use of a dance notation as input to an animation system. It provides a higher level of animation control for articulated models defined by an appearance graph.

After briefly describing traditional animation principles (Section 4.2), we will look at related work in the

field of computer character animation (Section 4.3). Then, Section 4.4 will introduce a new multi-layered model, composed of a manually specified inner layer and automatically generated outer layer. Section 4.5 will present the process for its user-controlled creation of the outer layer. Section 4.6 will describe the choreographic nature of animation and present three dance notations used in the dance community, choosing one of them for implementation and justifying the choice. An overview of a hybrid animation control technique proposed for the layered model, *i.e.* methods for animating the inner and the outer layers of the model, will then be presented (Section 4.7). A way of manual animation control of the inner layer based on the Eshkol-Wachmann dance notation will be discussed in Section 4.8. Two methods of automatic motion generation for the outer layer will be proposed and compared in Section 4.9. Applying a combination of these manual and automatic motion control methods to the creation of traditional animation effects in implicit surface animation will be shown in Section 4.10. The chapter will conclude with a case study of a "cute" dinosaur (Section 4.11).

4.2 Traditional animation principles

This section briefly describes the principles of traditional animation interpreted after "Disney Animation - The Illusion of Life" by Thomas and Johnston [Thom81].

Squash and Stretch: In the real world only metal pans and concrete blocks move in a rigid way. Everything that is alive squashes and stretches to a certain extent during its motion. Therefore, any living object in Disney cartoons, *e.g.* a "rigid" broom or a "wooden" boy (Pinocchio), deforms and expresses its feelings, moods and personality by squashing and stretching its body.

Anticipation: Anticipation prepares an audience for the forthcoming action. The feeling of suspense is created and it ensures that none of the important events in an animation sequence will be missed by the audience.

Staging: An action of a story can be presented in many locations. Staging an action means putting it in a background that creates the required atmosphere through the use of appropriate symbols. For instance, to create the feeling of fear, a dark, sharp landscape may be used, with a few bats and a black cat.

Straight Ahead Action and Pose-To-Pose Action: These are two contrasting methods of developing an animation sequence. Straight Ahead Action is a more spontaneous way which uses the ideas that are coming along while the sequence is being created. Such actions are usually more wild and expressive. Pose-To-Pose Action is Disney's definition of keyframing. A sequence is carefully planned as a series of *keyframes* that mark the important events of a story. *Inbetweens*, *i.e.* frames between each pair of keyframes are then developed. The story told this way becomes clearer and stronger. However, traditional animators found it more difficult to deal with Pose-To-Pose Action. The inbetween frames often lacked the spontaneity of Straight Ahead Action. Some tricks to make the change from one key position to another more convincing are described as the next principle.

Follow Through and Overlapping Action: These are two terms describing a set of tricks used by traditional animators to add more life to their characters. Apart from the main action taking place, some additional movements are always happening to make sure that characters never stop moving and thus never look like flat drawings. Several ways of achieving such effects have been developed. If a character has any "appendages", *e.g.* long ears, tail, clothes, feathers, these will move separately following the main movement and emphasising their weight and dependence on the character. Sometimes parts of a body can behave this way, *e.g.* a big stomach or loose flesh being "dragged behind" the body. A character will not move all its parts at the same time, instead, each of them will get to their final position individually, possibly with a delay, creating a more dramatic effect. Finally, there is a Moving Hold, a way of stopping a character without using a still image. A character will stop and then slowly exaggerate its stopping position and expression, giving an audience the time for a reflection on the events that caused such a halt. All these tricks add little life-giving features to the behaviour of a character.

Slow In and Out: This is a way of creating inbetween frames for specified keyframes. By creating more inbetweens at the beginning of a transition and at its end, with fewer frames in the middle, a gradual change of attitude is achieved.

Arcs: At the beginnings of the art of traditional character animation, motion took place along straight lines resulting in a rigid behaviour. To avoid such stiffness, movement along curved lines was introduced. Characters started to "arc" over stairs or "arc" their arms reaching for objects. The result was a more realistic, less mechanical motion.

Secondary Action: The main action can be supported by a secondary action that strengthens the required emotion. For instance, a disappointed person may discretely wipe off a tear before turning round to walk away. Care should be taken not to make the secondary action too big and therefore diminish the impact of the main action.

Timing: Timing is extremely important in character animation. The same motion performed at a different speed may completely change the meaning of an action. In [Thom81] the following example is given:

Just two drawings of a head, the first showing it leaning towards the right shoulder and the second with it over on the left and its chin slightly raised, can be made to communicate a multitude of ideas, depending entirely on the Timing used. Each inbetween drawing added between these two "extremes" gives a new meaning to the action.

NO inbetweens:	*The Character has been hit by a tremendous force. His head is nearly snapped off.*
ONE inbetween:	*The Character has been hit by a brick, rolling pin, frying pan.*
TWO inbetweens:	*The Character has a nervous tic, a muscle spasm, an uncontrollable twitch.*
THREE inbetweens:	*The Character is dodging a brick, rolling pin, frying pan.*
FOUR inbetweens:	*The Character is giving a crisp order, "Get going!", "Move it!".*
FIVE inbetweens:	*The Character is more friendly, "Over here", "Come on - hurry!".*
SIX inbetweens:	*The Character sees a good looking girl (man) or the sports car always wanted.*
SEVEN inbetweens:	*The Character tries to have a better look at something.*
EIGHT inbetweens:	*The Character searches for the peanut butter on the kitchen shelf.*
NINE inbetweens:	*The Character appraises, considering carefully.*
TEN inbetweens:	*The Character stretches a sore muscle.*

Exaggeration: To make an animation more convincing, all actions were exaggerated to produce a caricature of reality. Such an animation tells a story more clearly and emphasises the key events or personality characteristics that are essential to create the "illusion of life".

Solid Drawing: This principle relates to the process of character modelling. The features to be avoided are "twins", *i.e.* same legs, hands doing the same thing or eyes looking in the same direction. The sought shapes have volume and flexibility, are strong but not rigid. Parallel lines are out and curved lines are in. A character modelled without "solid drawing" is more lively even when static.

Appeal: Appeal describes the whole set of features of a character that create the required impression on an audience. For a bad character, *e.g.* a nasty witch, this may not necessarily be pleasant or positive characteristics. By emphasising this principle, animators are made aware of the importance of the personality, reflected in both the look and the motion, that they are creating for any of their creatures.

These principles address all stages of the process of animating. Solid Drawing and Appeal guide the character design and modelling. Staging suggests a choice of an appropriate background location for the action. Two styles in which the sequence can be developed are proposed: Straight Ahead or Pose-To-Pose Action. The remaining principles deal with the generation of motion for the characters. Among them, Slow In and Out, Arcs and Timing describe how inbetweens can be created to achieve more expressive effects. The manner of performing any motion is enhanced by a preparatory stage (Anticipation), additional effects while it is happening (Secondary Action) and its progressive finishing (Follow Through and Overlapping Action). Squash and Stretch argues the essential presence of deformation in expressing any feeling and performing any action. Finally Exaggeration suggests that any output from any of the stages of the animation creation can be exaggerated to make the main points more evident to the audience.

In order to introduce automatic traditional animation effects to the animation of implicit surfaces, the principles of motion generation were considered. Among them, Squash and Stretch, Anticipation, Secondary Action, Follow Through and Overlapping Action and Exaggeration, are probably the ones that "breathe life" into traditionally animated characters and that are lacking in computer generated characters.

Implicit surfaces can be coherently squashed and stretched (see Chapter 3). The principle of Squash and Stretch is therefore automatically achievable in their context. Anticipation effect cannot be easily generated for an arbitrary motion, since a more thorough semantical analysis of the event is required to derive its potential anticipating action. Semantical analysis of motion is beyond the scope of this thesis. Similarly, Secondary Action was ruled out since it means explicitly adding new behaviour to parts of a character in order to emphasise the main expression or feeling. Follow Through and Overlapping Action can be automatically generated by procedurally animating parts of a character. Finally, for a defined animation sequence possible ways of its automatic Exaggeration will be looked into.

4.3 Related work

The experience of traditional character animation has been successfully applied to computer animation. In [Lass87] John Lasseter, who was trained at Disney, described the creation of “Luxo Jr.”, an animation sequence starring an anglepoise lamp that appealed to the audience in the same way as characters in Disney animation. This work is undoubtably a milestone in the history of computer character animation used as a creative tool for animation artists. It was the first computer animation to be nominated for an Academy Award. However, to create “Luxo Jr.” traditional animation principles were applied manually. The fact that an anglepoise lamp was chosen as the character helped the squash and stretch and other effects to be achieved more easily.

Following the success of “Luxo Jr.” in 1986, more PIXAR productions appeared: “Red's Dream” in 1987, “Tin Toy” in 1988 - the first computer animation to be awarded an Academy Award , “Knick Knack” in 1989. The first fully computer animated feature film, “Toy Story”, created by PIXAR for Disney, was released in 1996. A lot of artistic skill applied manually to computer models is required to create all of these animations which makes it very hard for an ordinary user to reproduce similar effects. More support from an animation software is thus required.

4.3.1 Layered models

A more automated approach to computer character animation is the use of a layered model. In such a model, a character is created as an articulated skeleton covered with deformable layers of muscle, fat and skin. The skeleton can be animated using any method, *e.g.* keyframing or a physically-based approach. Corresponding deformation of the skin is generated according to the motion of the underlying skeleton. The skin in layered approaches is most often modelled using polygonal or parametric surfaces. A mesh of polygons or parametric patches is deformed to fit the moving links and joints of the articulated skeleton.

Magnenat-Thalmann *et al.* [Magn88] develop a method called joint-dependent local deformation (JLD) to model deformation of a polygonal mesh. Polygon vertices are bound to points on the articulated skeleton and move following with the skeleton motion. The skin deformation around joints in a human figure is handled by geometric deformation procedures tailored for each joint type. The procedure take into consideration the angles between limbs at the joint and approximates the result expected for this particular joint in humans.

Chadwick *et al.* [Chad89] use free-form deformation (FFD) blocks [Sede86] to represent a layer of muscle between the skeleton and the skin. FFD blocks are associated with each muscle and a subset of their control points is bound to the relevant link in the skeleton by spring and masses connections. The polygonal skin is embedded in FFD blocks and is parametrised with relation to the FFD space. The flexing of the muscles is modelled by deforming the associated FFD blocks. The resulting deformed parametrisation for the skin is calculated and represents the deformed skin layer.

Forsey [Fors91] describes a hierarchical B-spline surface which can generate folds and creases around joints. The deformation is achieved by moving some control points of surface patches. The surface change is restricted to the patches that share the moved control points. For deformation that is more local than the size of a patch, the mesh of patch control points can be progressively refined until a small enough mesh of control points is achieved.

These three approaches [Magn88, Chad89, Fors91] defined the skin in a separate modelling process using 3D scanning techniques or existing databases of human figures. Skin creation can be simplified by using implicit surfaces as a layer in the model. For instance, Shen and Thalmann [Shen95] use implicit

surfaces to define a muscle layer for a character. The resulting implicit surface is sampled and the sampling points are used to control a B- spline surface of the skin. The implicit surface helps to evenly distribute the control points of the parametric surface, otherwise a non-trivial problem.

Another approach to skin modelling is to generate an elastic surface attached to a rigid body at specified points using spring connections. Gascuel *et al.* [Gasc91] proposed a layered model in which the skin is defined by a spline surface and attached to a rigid kernel by stiff springs. The skin deformation is calculated by combining the influence of external forces with the internal response of the springs. Turner [Turn95] adapted the deformable elastic polygonal mesh of point masses given by Terzopoulos *et al.* [Terz87] to the task of skin definition for a layered character. The surface is attached to the body by a series of point-to-point spring constraints and deforms in response to environmental forces such as gravity and air pressure. Turner uses a set of unblended implicit superellipsoids to define the body of a character.

4.3.2 Implicit surfaces

Implicit surfaces offer a naturally layered model (generators and an isosurface) in which the continuity of the implicit "skin" is automatically attached to the skeleton and easily deformed with its continuity preserved. This is one of the main motivations for using implicit surfaces to develop an alternative layered approach for computer character animation in which the skin for a character is associated with an isosurface, *i.e.* it does not have an explicit representation.

Implicit surfaces have already been used for character animation. Although manually animated, a guitar player in a toothpaste advert [Beie90] proves that the smooth appeal of implicit surfaces and its layered structure that promotes fast prototyping of an animation sequence are highly beneficial for computer character animation.

The principles of traditional animation have also been applied to implicit surfaces. Wyvill [Wyvi92] developed a technique for modelling squash and stretch effects. In his work, the scalar field around generators was deformed by warping different regions of space and applying the warping function to the overall scalar field value. A warping region was normally associated with an object in a scene, *e.g.* a floor. Thus, when an object traverses a warping region, its scalar field deforms to model squash, *e.g.* on contact with the floor, or stretch, *e.g.* after having bounced off the floor. Effects such as slug-like motion or waves going through a surface can be achieved by choosing a suitable warping function.

Such a layered approach to character construction is particularly attractive for fast design and motion prototyping. A character can be viewed at varying levels of detail. The the low level, a stick figure representing the articulated structure can be displayed. The subsequent layers can be added to improve the animator's perception of the character's look and the manner of motion. A fully rendered, slower to produce, version of the character may only be used to view the final result. If further refinement is required, it can be performed at a lower level of detail.

4.4 Layered model

The model proposed in Chapter 3 consists of two layers: a set of generators and an implicit surface that represents the "skin" for an object. Animation can be specified at the generator level and the implicit coating will follow the resulting motion. Deformations that occur due to collisions between objects or self-collisions within an object are modelled at the implicit surface level. Therefore, an animator can focus on the desired motion letting the software take care of deformation processing. Moreover, the software ensures coherence preservation so a character can be animated by pulling and pushing its parts without topological integrity loss.

The goal of extending such a model is to provide a higher level of animation control that will automatically generate the chosen traditional animation effects: Squash and Stretch, Follow Through and Exaggeration. The extension proposed is to add a layer of *flesh* to the model. Flesh consists of a group of primitives distributed around the components, attached to them and blendable with them.

Three layers can be identified in this extended model (Figure 4.1): user-designed *object components*, automatically generated *flesh* and an automatically created *skin*, *i.e.* the isosurface that covers both components and flesh. In this model, object components form a layer that is animated by a user. Based on their

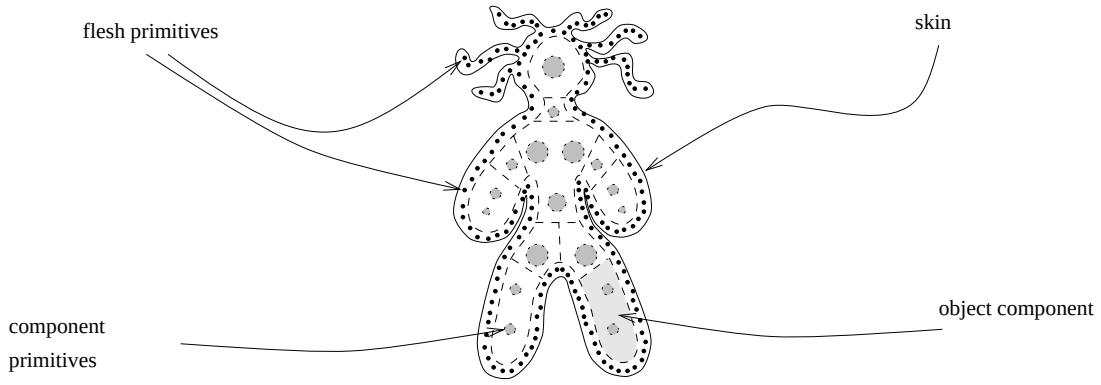


Figure 4.1: *Three layers in the new model: object components, flesh and skin*

motion, the flesh movement is computed. The skin is automatically deformed according to the motion of both components and flesh taking into consideration the blending properties for an object. The blending properties of flesh are derived from the blending properties of components. A more detailed description of the process of flesh generation for a given model, including assigning flesh blending properties will now be presented.

4.5 Flesh generation

Two kinds of flesh are proposed: a *coat of flesh*, which covers a component, and a *strand of flesh*, attached to a component by one of its ends. Both types of flesh are connected to their associated components by distance constraints of a certain, user-specified, firmness characteristic. A user also specifies the field function to be used for flesh primitives.

4.5.1 Coat of flesh

A coat of flesh around a component is generated automatically using an algorithm adapted from the scattering method described in [Bloo90b], in which particles, initially scattered in space, travel to the isosurface. First, for each primitive in a given component, flesh primitives are distributed around it at its radius of influence (Figure 4.2). The primitive around which flesh is distributed will be called the *parent primitive* for all flesh primitives created around it. A simple parametrisation of a sphere is used to distribute primitives: for a parent primitive with a generator G_i and a radius of influence R_i, a coat of flesh is given by generators $P_{\phi\psi}$:

$$P_{\phi\psi} = G_i + (R_i \sin\phi \cos\psi, R_i \sin\phi \sin\psi, R_i \cos\phi)$$

where $\phi = \Delta\phi, 2\Delta\phi \ldots 2\pi$ and $\psi = \Delta\psi, 2\Delta\psi \ldots 2\pi$. The density of flesh primitives can be controlled by varying the sampling rate, *i.e.* by adjusting the values of $\Delta\phi$ and $\Delta\psi$. These initial flesh primitives migrate towards the parent until they reach the isosurface. A linear search is used to find the intersection point. These primitives that reach the parent without intersecting the isosurface are ignored. They are marked with crosses in Figure 4.2. The remaining ones constitute a coat of flesh. The distribution of flesh primitives is not even which produces less regular but not undesirable effects. Some recent works [Witk94, Desb95b] propose methods of adaptive sampling of implicit surfaces which produce a more uniform distribution of sample points. A coat of flesh directly inherits the blending properties of its associated component.

4.5.2 Strand of flesh

A strand of flesh is modelled as a chain of primitives attached to a component. A user has to specify a parent primitive in the component, an attachment point, a direction of growth and the number of flesh primitives to be used. The system will then automatically create the strand. The first strand primitive (a *root*) is

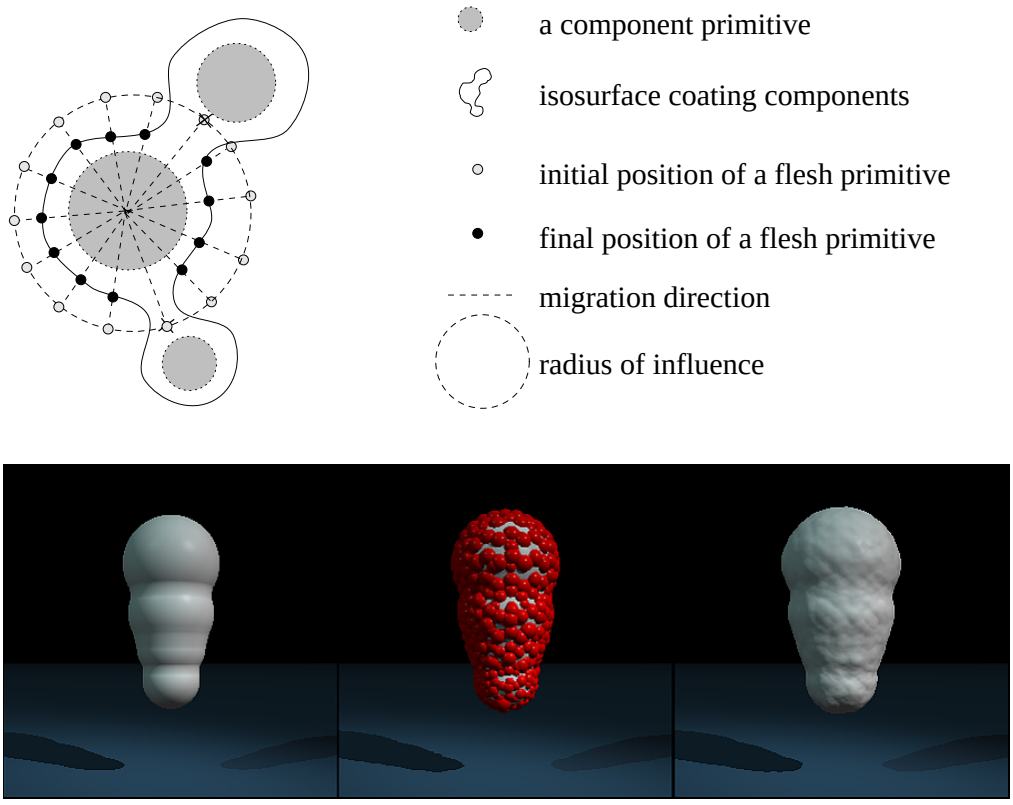


Figure 4.2: *Coat of flesh generation process: Top: creating flesh primitives, Bottom: component, component coated with flesh primitives (represented by spheres) and component blended with flesh*

generated at the specified attachment point (Figure 4.3). It then migrates along a line connecting it with the parent until it reaches the isosurface. The motion is towards the parent if the attachment point is outside the isovolume and away from the parent if it is inside it. If the root reaches the parent without intersecting the isosurface, it is marked as invalid (a cross in Figure 4.3) and the strand is not created. After placing the root on the isosurface, the remaining strand primitives are evenly distributed along the strand direction $\overrightarrow{d}$. The centre P_i of i-th primitive in the strand, in relation to the root P_0, is given by:

$$P_i = P_0 + iR_0 \frac{\overrightarrow{d}}{\| \overrightarrow{d} \|}$$

where $R_0 = f^{-1}(Iso)$ is the radius of a flesh primitive in isolation. It is an arbitrary distance that ensures coherence of the strand. Any other value from the coherence preserving interval $[d_{min}, d_{max}]$ (see Section 3.4.2) can be used. Each strand primitive can be tested against the scalar field due to the components to verify if it does not interpenetrate the isosurface. When a flesh primitive created is inside the isovolume, a shorter strand is generated. Suppressed strand primitives are marked with squares in Figure 4.3.

This kind of flesh is useful for modelling "boneless" parts of a character, *e.g.* hair or a soft tail. All strand primitives are assumed to be the same size. Interesting effects may be achieved by introducing varying radii of primitives and modelling for instance a tapering tail. For the blending properties for a strand of flesh, it is assumed that the root is blendable with the associated component and the next strand primitive. The remaining strand primitives are only blendable with their two neighbouring strand primitives.

4.6 Choreographic nature of animation

Creating an animation sequence has many common points with choreographing a dance piece. Both these creative processes start with a vague, often intangible idea and through a series of stages transform it into

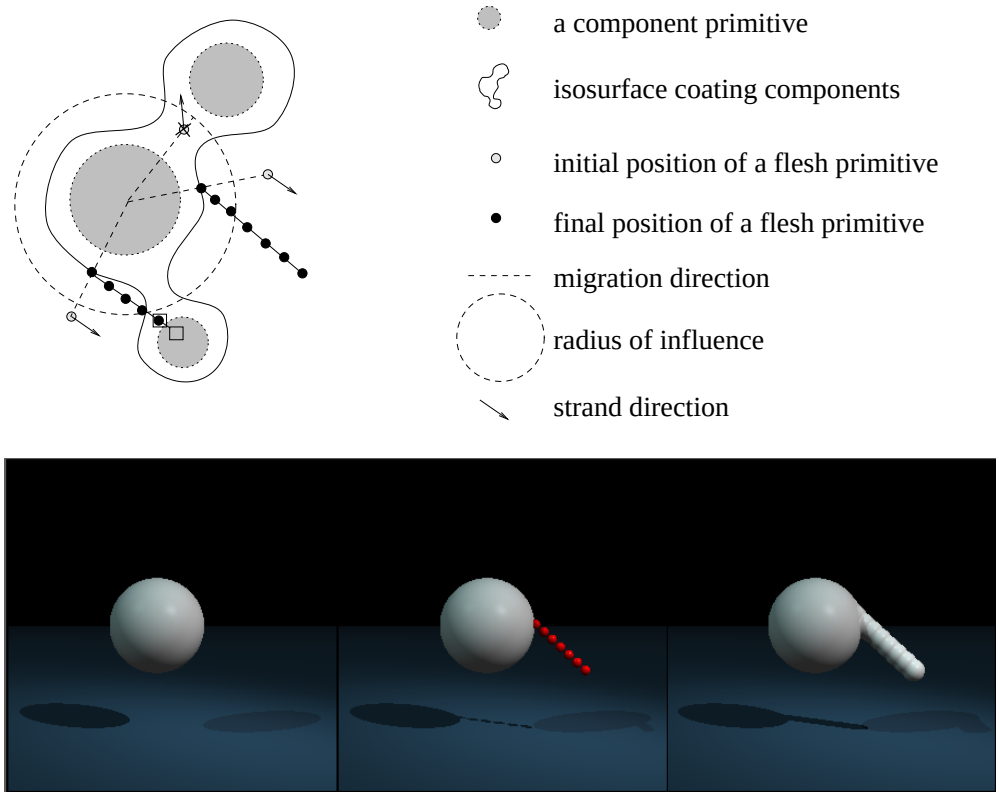


Figure 4.3: *Strand of flesh generation process: Top: creating flesh primitives, Bottom: component, component with strand primitives (represented by spheres) and component blended with flesh*

a motion specification for characters in the case of animation, or for dancers in the case of choreography. There are several formalisms (*dance notations*) developed to record an existing dance piece. Some of them have been computerised and there exists software capable of editing and interpreting them [Calv96]. This section will point out the similarities between choreography and animation and review the most popular dance notations looking for their application to the composition of an animation sequence.

4.6.1 Choreographic process

A choreographer starts with an inspiration, an abstract idea to be conveyed through the dance. The manner of transforming such abstraction into a dance performance defines a unique artistic style of a choreographer. One possible path is to start from a general sketch of the dance, a "conceptual storyboard", in which movements are referred to in abstract terms, *e.g.* dance solitude, walk into a new perspective, comment on the deception of time, leave a thought of fatalism. The next stage is to develop a series of dance phrases that express required feelings and moods. These phrases are then taught to the dancers at rehearsals. The following stage is the recursive refinement of the phrases, the creation of their variations often suggested by the dancers and improvised interaction between them. The dancers can therefore be active participants in the creative process. By the end of this stage, the phrases are combined into monologues and dialogues between dancers. These dance elements can be now fitted into the conceptual storyboard, that itself may have evolved during the creative process. If the choreographer is satisfied with the resulting dance, the process gets to its final stage: the performance rehearsal, during which small movements might be modified but whose main purpose is to prepare the dancers to *dance* the feelings in the choreographed movements rather than simply performing them. The choreographic process may go back to an earlier stage of creation many times and commence its refinement from there.

4.6.2 Animation process

Creating an animation sequence also starts from an initial idea, a message to be given to the audience. Again, a way of transforming this vague goal into the final animation depends strongly on the animator's style of work. A few sketches are usually drawn at the beginning to present the artist's concept of the story. These are then developed further into a detailed storyboard for the animation and a study of characters in the animation. Character modelling is the next stage and it can be compared to an audition for dancers in the choreographic process. Then, each sequence described in the storyboard is developed into a series of shots or phrases, refined until the required emotion is conveyed through them. For large animation projects, the active participation of dancers in the creative process can be compared to the hierarchical development of the animation, in which a separate animator can be assigned to the task of animating a particular character or a background event and be allowed to create it according to his or her individual artistic sense. Finally, the developed sequences are tried out together, any possible discrepancies or non-smooth transitions between them removed and the animation is ready for performance. At any time, the animator may decide to go back to a previous stage to refine the creation process from there.

4.6.3 Dance notation

Most visual recordings of dance pieces are recordings of an interpretation of a choreography performed by a group of dancers. Such interpretations will differ from performance to performance and from one group of dancers to another. Recreating the initial choreography from it cannot be free from the influence of the personalities of performing dancers. Notwithstanding the importance of this style of recording, there is a need for saving a pure, uninterpreted choreography, that can be taken up by a choreographer and a group of dancers for re-interpretation.

One way of recording a choreography is writing it up as a formalised score, analogous to writing up a musical piece. However, there is no established notation for dance that could be equivalent to the musical score. Moreover, the existing dance notations are difficult to master and they require a trained notator to record a choreography. The three most popular dance notations are Benesh notation [Bene56], Eshkol-Wachmann notation [Eshk58] and Labanotation [Hutc70]. They will be briefly described in the following sections.

Benesh notation

Benesh notation [Bene56] was developed mainly for classical ballet. It records the dance movements on the five-line music stave aligned with the musical score used for a choreography. The bottom line in the stave becomes the floor and the following ones represent the levels of the knees, the waist, the shoulders and the top of the head of a dancer. A set of symbols is used to mark the placement and the orientation of these body parts for any given dance position. Due to its use of free symbols, *e.g.* curves to describe limb movements or body transitions between dance positions, Benesh notation is difficult to learn and automate.

Labanotation

Labanotation [Hutc70] represents a human figure as a set of joints connected by limbs. The motion of each joint in the body is drawn as a vertical stripe describing the evolution of its positions in time. Each time stripe contains a series of symbols that give the place, orientation and the level in space for each joint. The size of symbols gives the duration of each movement and additional diagrams are drawn to show the relationship between the dancers in the stage space. Because of its highly formalised syntax, Labanotation is most often used in computer systems for dance [Bare77, Smol77, Calv80, Hall95].

Eshkol-Wachmann notation

Eshkol and Wachmann [Eshk58] define a model of a human body as an articulated structure of limbs connected by joints. They then specify local rotations for each limb using three types of transformations: *rotatory movements*, *plane movements* and *curved surface movements*, each described as the angles necessary to perform the relevant rotation. Each dance position can be detailed as a sequence of these three

types of movements starting from the neutral position in which the amounts of transformations for all limbs are zero. The given rotations are performed for the relevant links in the articulated structure describing a dance position for the model. This makes Eshkol-Wachmann notation particularly suitable for computer implementation, since such a description of dance positions directly uses the *forward kinematics* principle in which the angles between the limbs (links) in the articulations (joints) of the model are given to describe a state of an articulated structure (see *e.g.* [Watt92]). Moreover, although Eshkol-Wachmann dance notation was created for human figures, it can be trivially extended for any character by specifying the elementary movements for all links in an appearance graph of a character. For these reasons Eshkol-Wachmann notation has been chosen as a method for animating components in the layered model for implicit surfaces (Section 4.8).

4.7 Animation control

At the lowest level of abstraction, controlling an animation of a model means specifying the time evolution of all parameters required to specify a state of the model. For rigid objects, the position in space and the orientation is sufficient. For deformable objects, the change of the geometry and possibly the change of the topology have to be specified. For complex models, detailing all these parameters becomes a tedious manual procedure that requires a great deal of effort to create an animation. A higher level of control is thus necessary. At a higher level of control, the animator deals with more abstract description of the animation, normally passing a set of rules or constraints to the algorithms implemented in software.

Physics-based simulation is one example of higher level motion control. The physical conditions in a scene are specified and numerical solutions to the Newtonian equations of motion give the new positions of all objects. Particle systems are another example. Each particle behaves according to specified rules and a set of particles interacting with each other create randomised behaviours of the model. For objects defined using an articulated structure, one control method is *kinematics* (see *e.g.* [Watt92]) which requires a description of the positions of the links in the articulations (joints) of the model. The positions can be specified directly (*forward kinematics*) or calculated from the required placing of the ends of the links, so called end-effectors, *e.g.* hands or feet of a character(*inverse kinematics*).

The coherence preserving animation control method, described in Section 3.4, is a low level method in which an animation is created by specifying the motion for individual primitives in a model. To specify a higher level of control for the layered model described in Section 4.4, a hybrid control method in proposed, in which the two layers are animated separately. The motion of components is specified using forward kinematics (Section 4.8) and for the flesh, a procedural method is developed (Section 4.9) that links the flesh motion with the component motion by having the flesh "follow" the movement of components. The flesh motion is generated automatically and a set of parameters, used to control the process, can be adjusted to refine the result.

4.8 Component motion

This section details a method of motion specification for the object component layer using forward kinematics. The inspiration for this particular implementation of forward kinematics comes from the world of dance. It uses the Eshkol-Wachmann dance notation [Eshk58] to define poses of a character that can be combined into a more complex movement (see Section 4.6). Perlin [Perl95] used a similar approach to choreograph dance sequences for a virtual puppet. He created a library of movements using low level forward kinematics and combined them to create a dance. Implementing the Eshkol-Wachmann dance notation allows for the state of an articulated structure to be specified at a higher level of abstraction, using a more intuitive set of elementary movements (rotatory, plane and curved surface movements) rather than directly specifying limb angles at joints.

Figure 4.4 presents the modelling script and a model of a dancer in the neutral position. This dancer will perform the elementary movements and some of the dance sequences, created using the dance notation, that will illustrate the results presented in this chapter.

```
iso 0.8
object Dancer colour Grey
component head r (0,0,0)
    point (-1.5,6,0) POV 5 1.2
    point (1.5,6,0) POV 5 1.2
    point (0,2,0) POV 5 1.2
    blend SUM
component neck r (0,0,0)
    point (0,0,0) POV 3 1.2
    blend SUM
component torso r (0,0,0)
    point (0,6.5,0) POV 5 1.2
    point (-4,6.5,0) POV 5 1.2
    point (4,6.5,0) POV 5 1.2
    point (0,2,0) POV 4 1.2
    point (-2,2,0) POV 4 1.2
    point (2,2,0) POV 4 1.2
    point (0,0,0) POV 3 1.2
    point (-2,3.5,-2.5) POV 2.5 1.2
    point (2,3.5,-2.5) POV 2.5 1.2
    blend SUM
component pelvis r (0,0,0)
    point (0,-1,0) POV 5 1.2
    point (-1.5,-5.5,0) POV 3 1.2
    point (-1.5,-4.5,0) POV 3 1.2
    point (1.5,-5.5,0) POV 3 1.2
    point (1.5,-4.5,0) POV 3 1.2
    point (-2.5,-4,1) POV 4 1.2
    point (2.5,-4,1) POV 4 1.2
    blend SUM
component l_upper_arm r (0,0,0)
    point (0,-2.5,0) POV 4 1.2
    point (-0.5,-6,0) POV 4 1.2
    point (-1,-8,0) POV 3 1.2
    blend SUM
component l_forearm r (0,0,0)
    point (-1,-1.5,0) POV 3 1.2
    point (-1,-3.5,0) POV 2.5 1.2
    point (-0.5,-4.5,0) POV 1.5 1.2
    blend SUM
component l_hand r (0,0,0)
    point (-0.5,-1,0) POV 2 1.2
    point (0,-2.5,0) POV 1.5 1.2
    blend SUM
component r_upper_arm r (0,0,0)
    point (0,-2.5,0) POV 4 1.2
    point (0.5,-6,0) POV 4 1.2
    point (1,-8,0) POV 3 1.2
    blend SUM
component r_forearm r (0,0,0)
    point (1,-1.5,0) POV 3 1.2
    point (1,-3.5,0) POV 2.5 1.2
    point (0.5,-4.5,0) POV 1.5 1.2
    blend SUM
component r_hand r (0,0,0)
    point (0.5,-1,0) POV 2 1.2
    point (0,-2.5,0) POV 1.5 1.2
    blend SUM
component l_thigh r (0,0,0)
    point (0,-0.5,0) POV 5 1.2
    point (0,-2.5,0) POV 4 1.2
    point (0,-5,0) POV 4 1.2
    point (0,-8,0) POV 3 1.2
    blend SUM
component l_lower_leg r (0,0,0)
    point (0,0.5,0) POV 4 1.2
    point (0,-3,0) POV 4 1.2
    point (0,-6,0) POV 3 1.2
    blend SUM
component l_foot r (0,0,0)
    point (0,0,-1) POV 2 1.2
    point (0,0,-3) POV 2 1.2
    point (0,0,-4) POV 1.5 1.2
    blend SUM
component r_thigh r (0,0,0)
    point (0,-0.5,0) POV 5 1.2
    point (0,-2.5,0) POV 4 1.2
    point (0,-5,0) POV 4 1.2
    point (0,-8,0) POV 3 1.2
    blend SUM
component r_lower_leg r (0,0,0)
    point (0,0.5,0) POV 4 1.2
    point (0,-3,0) POV 4 1.2
    point (0,-6,0) POV 3 1.2
    blend SUM
component r_foot r (0,0,0)
    point (0,0,-1) POV 2 1.2
    point (0,0,-3) POV 2 1.2
    point (0,0,-4) POV 1.5 1.2
    blend SUM
component top_spine r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component low_neck r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component waist r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component l_elbow r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component l_wrist r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component r_elbow r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component r_wrist r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component l_knee r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component l_ankle r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component groin r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component r_knee r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
component r_ankle r (0,0,0)
    point (0,0,0) POV 1 1
    blend SUM
joint top_spine between
    head (0,0,0) neck (0,3,0)
joint low_neck between
    neck (0,-2,0) torso (0,6.5,0)
joint waist between
    pelvis (0,2,0) torso (0,0,0)
joint low_neck between
    l_upper_arm (0,0,0) torso (-6,7,0)
joint l_elbow between
    l_upper_arm (0,-8.5,0) l_forearm (0,0,0)
joint l_wrist between
    l_forearm (0,-5.5,0) l_hand (0,0,0)
joint low_neck between
    r_upper_arm (0,0,0) torso (6,7,0)
joint r_elbow between
    r_upper_arm (0,-8.5,0) r_forearm (0,0,0)
joint r_wrist between
    r_forearm (0,-5.5,0) r_hand (0,0,0)
joint groin between
    pelvis (-3,-5.5,0) l_thigh (0,0,0)
joint l_knee between
    l_lower_leg (0,0,0) l_thigh (0,-9,0)
joint l_ankle between
    l_lower_leg (0,-7,0) l_foot (0,0,0)
joint groin between
    pelvis (3,-5.5,0) r_thigh (0,0,0)
joint r_knee between
    r_lower_leg (0,0,0) r_thigh (0,-9,0)
joint r_ankle between
    r_lower_leg (0,-7,0) r_foot (0,0,0)
anchor waist (0,0,0)
end
```

Figure 4.4: *A script file and a model of a dancer*

4.8.1 Elementary movements

In the following implementation of the three types of elementary movements an assumption is made that the longitudinal axis of a limb is aligned with the Y-axis of the local coordinate system of the limb, the top of the limb, *i.e.* the articulation around which any limb rotation is performed, is positioned at the origin of the limb's local coordinate system and the front of the limb is the negative Z-axis. In Figure 4.5 two sample limbs, an "arm" and a "head", are shown in their respective local coordinate systems.

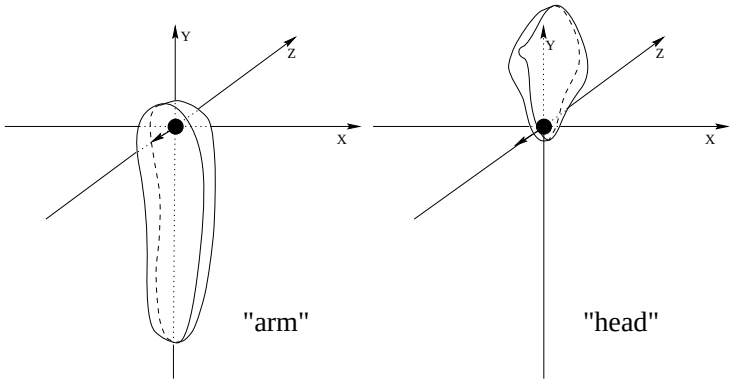


Figure 4.5: *Designing a limb in local coordinates: the longitudinal axis is aligned with the Y-axis, the top of the limb lies at the origin and its front is the negative Z-axis*

Rotatory movements

Rotatory movements are performed around the longitudinal axis of a limb. A user specifies an amount of rotation $\alpha \in [-2\pi, 2\pi]$. A negative α describes a clockwise rotation and a positive α gives an anti-clockwise rotation. Since the longitudinal axis is aligned with the Y-axis, a rotatory movement is a rotation by α around Y-axis described by the following rotation matrix:

$$\begin{bmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Plane movements

A limb moves in a plane movement when it describes a plane in space with its longitudinal axis. A user specifies a plane of motion and an amount of rotation $\alpha \in [-2\pi, 2\pi]$. The plane of motion is chosen from a family of planes perpendicular to the XZ-plane and going through the Y-axis (Figure 4.7). It can be described by an angle $\beta \in [0, \pi]$ which gives the required amount of anti-clockwise rotation of the XY-plane around the Y-axis. A plane movement is thus a combination of a rotation by β around Y-axis and a rotation by α around Z-axis. Its local transformation matrix is the multiplication of:

$$\begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Conical movements

A limb moves in a conical movement when its longitudinal axis describes a cone. A conical movement is a simple case of a more general *curved surface movement* in which a limb describes any curved surface in

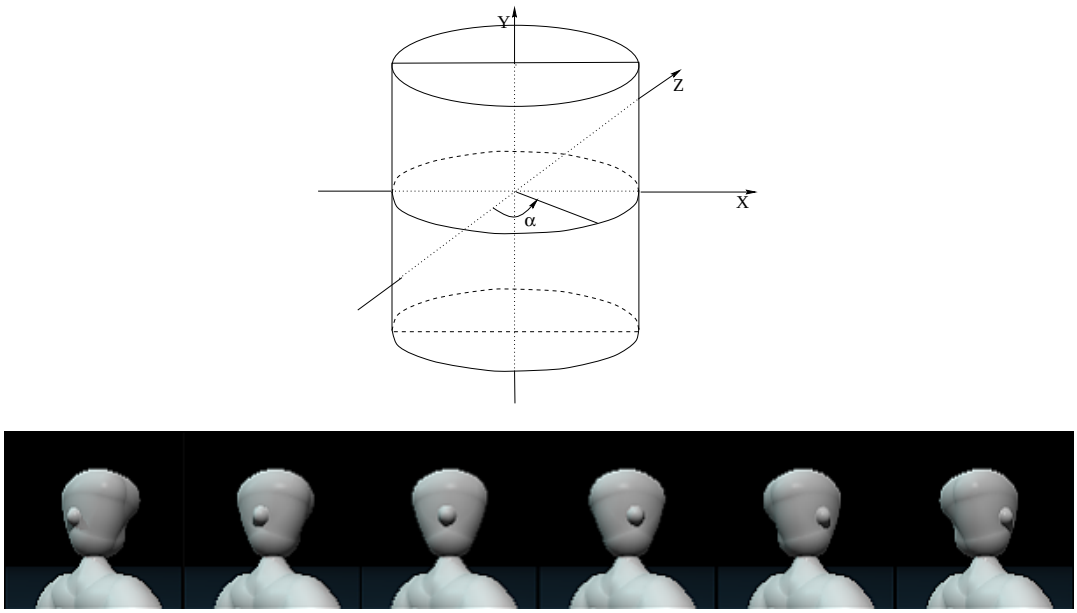


Figure 4.6: *Specifying a rotatory movement for a head ($\alpha = -\frac{\pi}{3}$)*

space. For a conical movement a user specifies a cone and an amount of rotation α along a cone, where $\alpha \in [-2\pi, 2\pi]$. The cone is described by its angular size $\beta \in [0, \frac{\pi}{2}]$ and its position in space, which is a planar movement of the axis of the cone with parameters γ and δ (Figure 4.8). A conical movement is thus a combination of a rotation by β around Z-axis, a rotation by α around Y-axis (these two rotations give the motion along the cone as if it was aligned with the Y-axis) and then a plane movement described by γ and δ to transform the cone to its position. The matrix of this motion is the multiplication of:

$$\begin{bmatrix} \cos\beta & \sin\beta & 0 & 0 \\ -\sin\beta & \cos\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\delta & 0 & -\sin\delta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\delta & 0 & \cos\delta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.8.2 Choreographing a sequence

Elementary movements can be combined for each limb by multiplying their corresponding matrices. A *motion element* is a set of elementary movements to be applied to limbs. Each motion element is assigned a time step at which it starts and a time step at which it ends. Starting from the neutral position, motion elements are progressively applied to the model. For each elementary movement in a motion element, the angle $\Delta\alpha$ is calculated by dividing the specified angle α by the number of frames during which the movement takes place. For each frame of animation, the current position of the model is calculated by applying a $\Delta\alpha$ part of all elementary movements in motion elements currently taking place. In Figure 4.9 two motion elements are specified. The first one takes place between times $t = 1$ and $t = 2$ and the second between times $t = 1$ and $t = 5$. For instance, for the frame $t = 3$, the entire first motion element has been applied and $\frac{3}{5}$-th of each movement from the second motion element.

Figure 4.10 shows three variations of a dance movement specified using the dance notation. During the first motion element (times 0-2), Dancer leans to the side (plane movements of the *torso* and the *head*). At the same time she opens her arms (plane movements), turns her *left leg* outwards (rotatory movement) and launches the *right leg* to the side (plane movement). The first motion element stays the same for all three sequences. Slight changes in choreography are achieved by adjusting the last two motion elements. The second motion element (times 2-4) is a rotatory movement of the *torso*. The angle is increased in each sequence. Frame 4 in each sequence shows the difference in the amount of turn of the Dancer's body. The

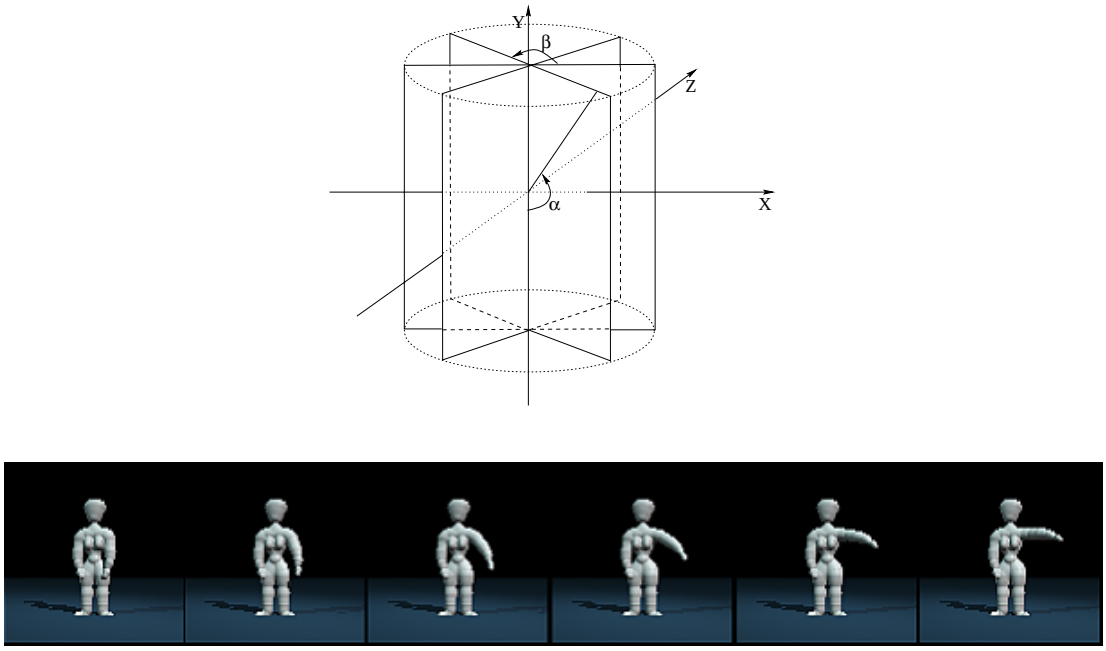


Figure 4.7: *Specifying a plane movement for an arm (*$\alpha = \frac{5\pi}{12}$ *and* $\beta = \frac{11\pi}{12}$*)*

last motion element (times 4-5) describes leaning of the *torso* backwards (plane movement) and bending of the *right leg* (plane movements). The amount of the lean is increased in each sequence. Similarly, more leg bending is achieved by increasing the amount of the plane movement and adding a plane movement for the *right calf*.

4.8.3 Path following

The choreography specified so far was performed with the dancer standing in one place. In this section another type of an elementary motion is added, *location change*. It is implemented as a translation of the anchor of the model given by a vector $T = (x_t, y_t, z_t)$. It is treated in the same way as the other three elementary movements, and performed at each time step as a translation ΔT calculated by dividing the coordinates of vector T by the number of frames during which the location change takes place.

Figure 4.11 shows a sample motion specified using location change: a jump in the first dance position (feet outwards, legs straight, knees together, stomach pressed against the spine, shoulders down, relaxed arms) landing in the second dance position (feet outwards, legs apart, knees bent, body resting along its axis) with arms open and chest facing the ceiling. The anchor of the model (*waist*) is moved in the vertical direction by the location change movement while the remaining elementary movements specify the pose of the dancer during different phases of the jump.

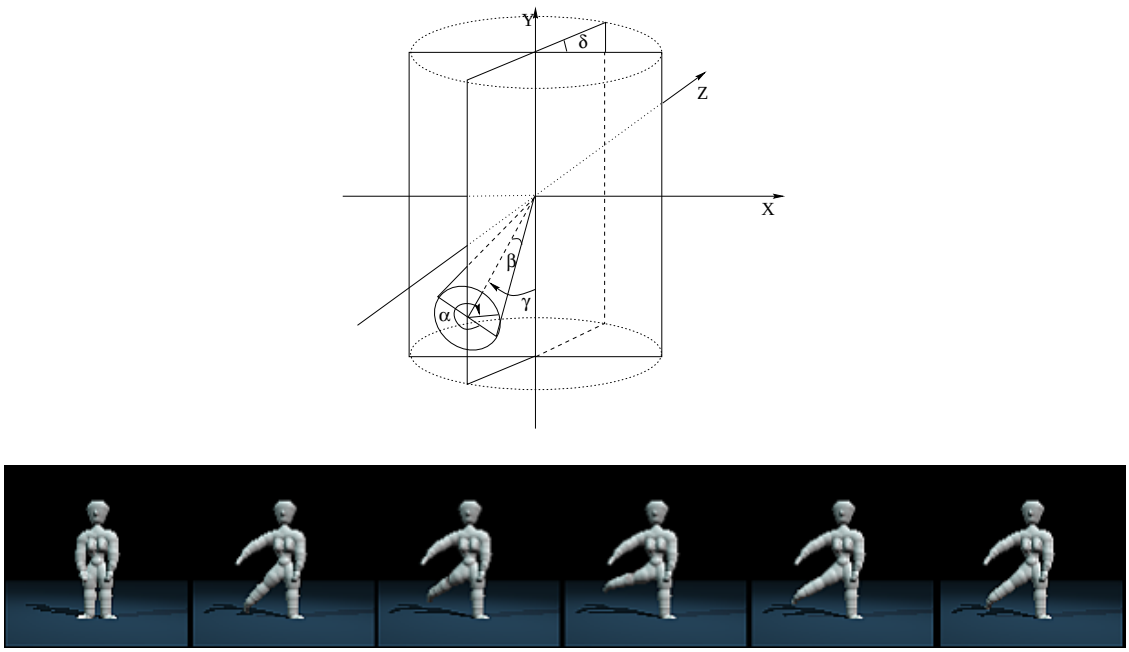
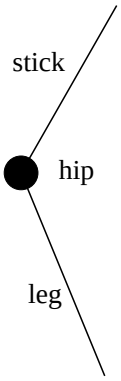


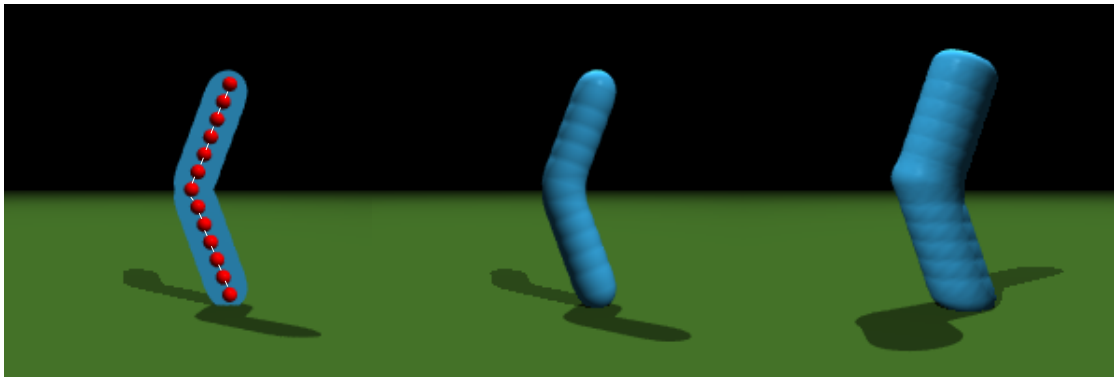
Figure 4.8: *Specifying a conical movement for a leg ($\alpha = -2\pi$, $\beta = \frac{\pi}{8}$, $\gamma = \frac{\pi}{12}$ and $\delta = \frac{\pi}{12}$)*

```
iso 0.8

object Stickleg colour SkyBlue
    component stick rotation (0,0,-20)
        point (0,5,0) POVfunction 10 1
        point (0,10,0) POVfunction 10 1
        point (0,15,0) POVfunction 10 1
        point (0,20,0) POVfunction 10 1
        point (0,25,0) POVfunction 10 1
        point (0,30,0) POVfunction 10 1
        blend SUM
    component joint rotation (0,0,0)
        point (0,0,0) POVfunction 10 1
        blend SUM
    component leg rotation (0,0,20)
        point (0,-5,0) POVfunction 10 1
        point (0,-10,0) POVfunction 10 1
        point (0,-15,0) POVfunction 10 1
        point (0,-20,0) POVfunction 10 1
        point (0,-25,0) POVfunction 10 1
        point (0,-30,0) POVfunction 10 1
        blend SUM
    joint joint between stick (0,0,0) leg (0,0,0)
    anchor joint (0,0,0)

end
```





```
object Dancer
    times 1 2
        plane l_upper_arm (-100,0)
        plane l_forearm (-20,0)
        plane l_hand (-20,0)
    times 1 5
        plane torso (-30,0)
        plane r_upper_arm (30,10)
        plane r_forearm (15,0)
        plane l_upper_arm (-50,0)
        plane l_forearm (-20,0)
        plane l_hand (-5,0)
        plane l_thigh (-30,0)
        plane l_foot (90,90)
    end
```

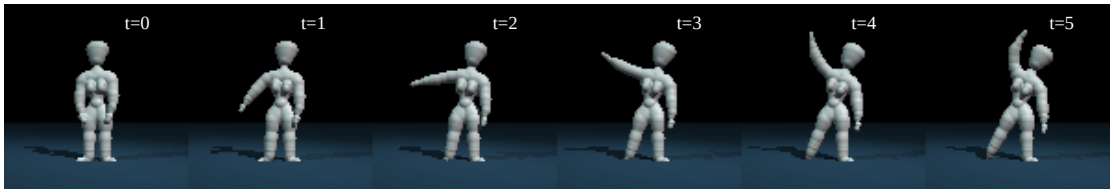


Figure 4.9: *A lean to the side specified using a dance notation script*

4.9 Flesh motion

Each flesh primitive has a parent either among the component primitives, as for skin and the root in a strand, or among other flesh primitives, as for the remaining primitives in a strand. Motion for a flesh primitive is generated by considering its position in relation to its parent. A simple approach is to use the coherence preservation algorithm to calculate flesh motion in a geometric manner. However, the effect obtained is unnaturally regular and gives a "wooden" look to a character. To introduce some irregularity to flesh behaviour, physical interaction can be added to its motion generation. The results of these two approaches to flesh motion control are presented in the following sections. Figure 4.12 presents a simple character, Stickleg, chosen to illustrate the difference between the two methods. Stickleg is built of two links and a joint between them and covered with a coat of flesh.

4.9.1 Geometric restriction

A connection between a flesh primitive and its parent can be treated as another distance constraint (see Section 3.4) to be maintained by the system. Thus, each flesh primitive exhibits push-pull action in relation to its parent.

Figure 4.14a shows the result of animating Stickleg coated with flesh, modelled in Figure 4.12, using this algorithm. The coat of flesh contains approximately 200 primitives. The firmness parameters specified for each connection between a flesh primitive and its parent are $f_{min} = 0.5$ and $f_{max} = 0.9$. Note the high degree of regularity in the motion. For instance, in frame 12, the flesh looks like a series of primitives translated behind the components. Moreover, at the end of the motion of the components, the flesh ceases its movement. This behaviour seems unnatural, hence some physical interaction between primitives is taken into account in the next method of flesh motion generation.

4.9.2 Adding physical interaction

The Lennard-Jones attraction/repulsion forces (see *e.g.* [Munc85]) between flesh primitives (Figure 4.13) model interaction forces which act between molecules. At the equilibrium distance between two primitives, r_{ij}, the interaction forces between them are equal to zero. When the distance between the primitives decreases, a force pushes them away from each other (repulsion force). When the distance increases, a force pulls them towards each other (attraction force). The interaction forces tend to zero when the distance

a

```
object Dancer
    times 0 2
        plane head (30,0)
        plane torso (45,0)
        rotatory l_thigh 30
        rotatory l_calf -45
        plane l_upper_arm (-130,0)
        plane l_forearm (-40,0)
        plane l_hand (-25,0)
        plane r_upper_arm (150,0)
        plane r_forearm (45,0)
        plane r_hand (30,0)
        plane r_thigh (30,0)
        plane r_foot (-30,-30)
    times 2 4
        rotatory torso 45
    times 4 5
        plane torso (30,0)
        plane r_thigh (45,0)
end
```

b

```
object Dancer
    times 0 2
        plane head (30,0)
        plane torso (45,0)
        rotatory l_thigh 30
        rotatory l_calf -45
        plane l_upper_arm (-130,0)
        plane l_forearm (-40,0)
        plane l_hand (-25,0)
        plane r_upper_arm (150,0)
        plane r_forearm (45,0)
        plane r_hand (30,0)
        plane r_thigh (30,0)
        plane r_foot (-30,-30)
    times 2 4
        rotatory torso 90
    times 4 5
        plane torso (45,0)
        plane r_thigh (45,0)
        plane r_calf (30,0)
end
```

c

```
object Dancer
    times 0 2
        plane head (30,0)
        plane torso (45,0)
        rotatory l_thigh 30
        rotatory l_calf -45
        plane l_upper_arm (-130,0)
        plane l_forearm (-40,0)
        plane l_hand (-25,0)
        plane r_upper_arm (150,0)
        plane r_forearm (45,0)
        plane r_hand (30,0)
        plane r_thigh (30,0)
        plane r_foot (-30,-30)
    times 2 4
        rotatory torso 100
    times 4 5
        plane torso (60,0)
        plane r_thigh (60,0)
        plane r_calf (30,0)
end
```

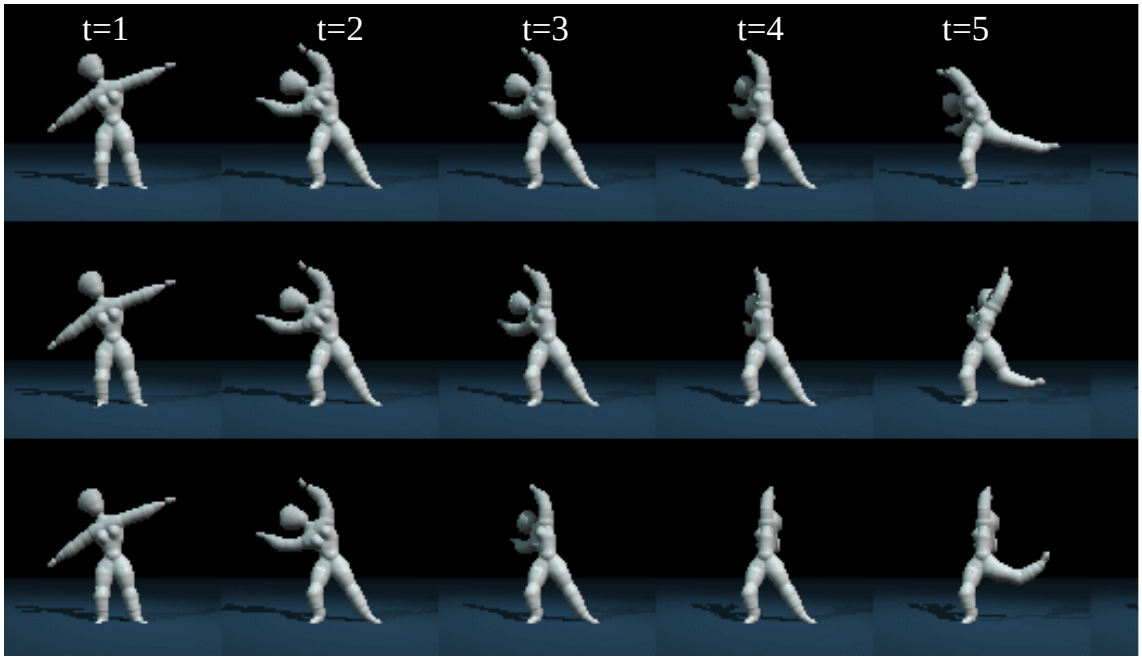


Figure 4.10: *Choreography using dance notation: three variations of a dance movement*

```
object Dancer
    times 1 1
        translate waist (0,-1,0)
        plane l_thigh (-20,0)
        plane r_thigh (20,0)
        plane l_calf (30,0)
        plane r_calf (-30,0)
        plane l_upper_arm (-5,0)
        plane r_upper_arm (5,0)
    times 2 2
        plane l_foot (80,0)
        plane r_foot (-80,0)
        plane l_thigh (25,0)
        plane r_thigh (-25,0)
        plane l_calf (-30,0)
        plane r_calf (30,0)
times 2 3
        translate waist (0,10,0)
        plane l_upper_arm (15,10)
        plane r_upper_arm (-15,10)
times 4 6
        translate waist (0,-10,0)
times 5 6
        plane l_upper_arm (60,10)
        plane r_upper_arm (-60,100)
        plane l_foot (-80,0)
        plane r_foot (80,0)
        plane l_thigh (5,0)
        plane r_thigh (-5,0)
        plane l_calf (30,0)
        plane r_calf (-30,0)
    times 6 6
        plane l_thigh (-5,0)
        plane r_thigh (5,0)
        plane l_calf (10,0)
        plane r_calf (-10,0)
        conical l_upper_arm (60,45,45,60)
        conical r_upper_arm (-60,45,135,60)
    times 7 7
        plane torso (20,45)
end
```

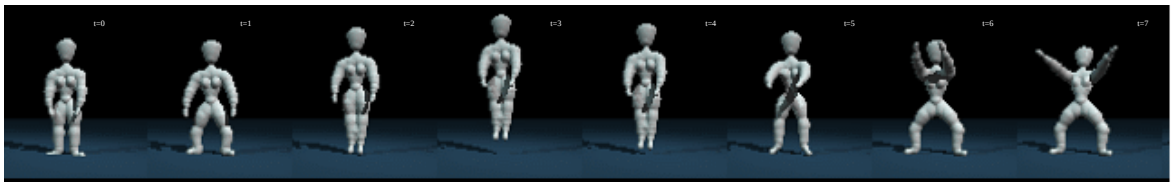


Figure 4.11: *A jump in "first" landing in "second" specified using dance notation with location change*

between primitives tends to infinity.

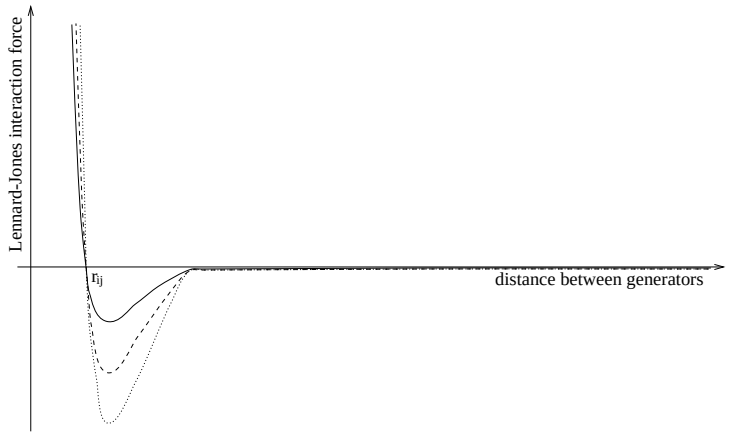


Figure 4.13: *Lennard-Jones interaction forces between two point masses*

In order to introduce such physical interaction to flesh behaviour, a mass m_i, a velocity $\overrightarrow{v}_i$ and a current acting force $\overrightarrow{F}_i$ are attributed to each flesh primitive. At the same time a gravity vector $\overrightarrow{g}$ is added as a global characteristic of a scene. At each time step, a flesh primitive will move according to these parameters. The position s_i and the velocity $\overrightarrow{v}_i$ of the i-th primitive change after time Δt according to the Newtonian equations of motion as follows:

$$\begin{aligned} s_i(t+\Delta t) &= s_i(t) + \overrightarrow{v}_i(t)\Delta t + \frac{\overrightarrow{F}_i(t)}{m_i}\Delta t^2 \\ \overrightarrow{v}_i(t+\Delta t) &= \overrightarrow{v}_i(t) + \frac{\overrightarrow{F}_i(t)}{m_i}\Delta t \end{aligned}$$

The Lennard-Jones forces L_{ij} between two flesh primitives, with current positions of generators G_i and

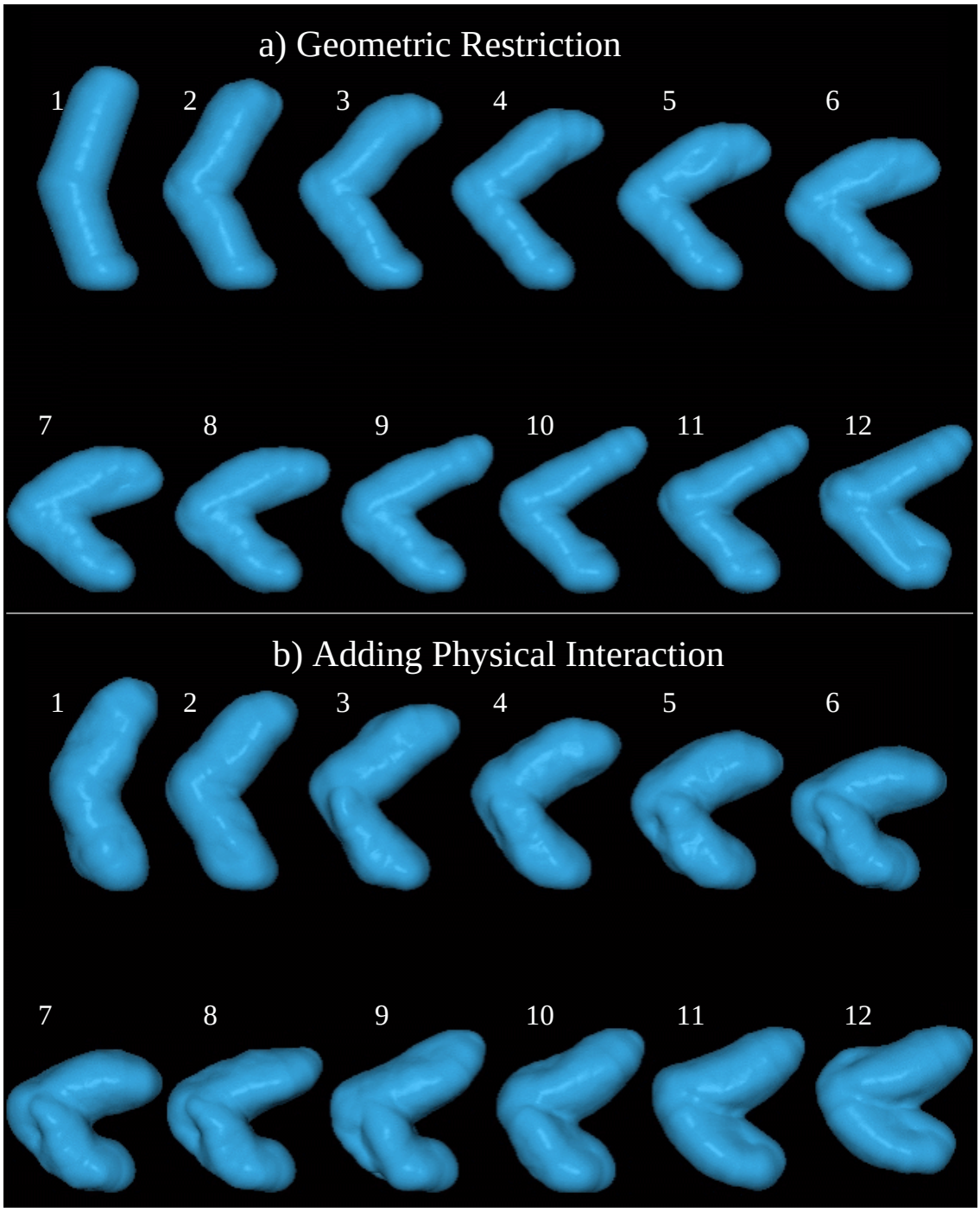


Figure 4.14: *Stickleg jumps with flesh controlled using a) the geometric restriction method and b) a combination of geometric restriction and the Lennard-Jones forces*

G_j, are computed during the calculation of the current acting force $\overrightarrow{F}_i$:

$$\overrightarrow{F}_i(t) = m_i \overrightarrow{g} + \sum_j \overrightarrow{L}_{ij}(t)$$

The $\overrightarrow{L}_{ij}$ calculation is implemented after Miller and Pearce [Mill89]:

$$\overrightarrow{L}_{ij}(t) = \begin{cases} a(\frac{b_1}{D_{ij}^m} - \frac{b_2}{D_{ij}^n})\overrightarrow{G_iG_j} & i \neq j \\ 0 & i = j \end{cases}$$

where a is a scaling factor, $D_{ij} = \|\overrightarrow{G_iG_j}\|$ is the distance between the two generators, $m = 2n$ are constants, b_1 and b_2 are variables, $b_2 = b_1 r_{ij}^{n-m}$, and r_{ij} is the equilibrium distance between the generators. The equilibrium distance r_{ij} depends on the initial distance between the two primitives. Recall the minimum and maximum coherence preserving distances between two primitives (Section 3.4):

$$\begin{aligned} d_{min} &= 0 \\ d_{max} &= f_i^{-1}(\tfrac{Iso}{2}) + f_j^{-1}(\tfrac{Iso}{2}) \end{aligned}$$

For primitives initially close to each other ($R_{init} \in [d_{min}, d_{max}]$), the equilibrium distance r_{ij} is set to their initial separation R_{init}. When the initial distance is larger than d_{max}, r_{ij} is set to d_{max}. As the result the interaction between flesh primitives is not homogeneous. The advantage of this approach is that there is little interaction between primitives which are initially far from each other, *i.e.* the primitives do not attempt to achieve the equilibrium state in which the distance between each pair of primitives is equal to a constant equilibrium value. Thus, the equilibrium state matches the initial configuration of primitives.

To calculate the new position of a flesh primitive due to the motion of its parent, the physical interaction method is combined with the geometric restriction method. At each time step Lennard-Jones forces are first computed between all flesh primitives and the above equations are used to find the potential new position for the i-th flesh primitive, P_{i_0}:

$$P_{i_0} \leftarrow s_i(t + \Delta t)$$

The geometric restriction method is then applied to adjust this position to maintain it within the distance constraints of the relevant connection, $[R_{min}, R_{max}]$. If such an adjustment is required, the primitive's velocity $\overrightarrow{v}_i$ is changed according to the adjusting translation:

$$\overrightarrow{v}_i(t + \Delta t) \leftarrow \overrightarrow{v}_i(t + \Delta t) + c\frac{\overrightarrow{P_{i_0}P_{i_1}}}{\|\overrightarrow{P_{i_0}P_{i_1}}\|}$$

where P_{i_1} is the adjusted flesh primitive position and c is a scaling factor.

When a flesh primitive reaches the R_{min} or R_{max} distance from its parent, its velocity $\overrightarrow{v}_i$ is reversed and dampened in the next time step:

$$\overrightarrow{v}_i(t + \Delta t) \leftarrow -D\overrightarrow{v}_i(t + \Delta t)$$

where $D \in [0, 1]$ is a damping factor.

Figure 4.14b shows the result of animating Stickleg coated with flesh using the above algorithm. The firmness parameters are not changed ($f_{min} = 0.5$, $f_{max} = 0.9$). The physical interaction between flesh primitives is calculated with gravity $\overrightarrow{g} = (0, 10, 0)$. The mass of a primitive is proportional to its radius in isolation $r = f^{-1}(Iso)$, $mass = \frac{4\pi}{3}r^3$. The Lennard-Jones interaction forces are calculated with parameters $b_1 = 10$, $n = 4$, $m = 2$. The factor a depends on the distance between the two primitives D_{ij}:

$$\begin{cases} a = 1 - \frac{D_{ij}^2}{r_{ij}^2(r_i + r_j)^2} & if \quad D_{ij}^2 \text{ ¿ } r_{ij}^2(r_i + r_j)^2 \\ a = 0 & if \quad D_{ij}^2 < r_{ij}^2(r_i + r_j)^2 \end{cases}$$

where r_{ij} is the equilibrium distance and r_i, r_j are the respective radii in isolation of the two primitives. The damping factor used in the simulation is $D = 0.5$.

The flesh primitives are attracted towards each other by Lennard-Jones interaction forces, thus they do not follow the motion of their respective parent primitives in the same manner. For instance, in frames 7-9, "muscle-like" irregularity of Stickleg's behaviour can be observed. The flesh is distributed more evenly around components and generator overlapping is prevented by repulsion forces. The flesh no longer ceases its motion at the same time as object components, it continues to move until its velocity is dampened to zero. The damping rate can be controlled by the damping factor.

Figure 4.15 shows an example of motion generation using this method for strands of flesh. A dancer performs a lean to the side and her hair, modelled as strands of flesh, automatically follows the motion. The firmness of "hair" is set to $f_{min} = 0.2$ and $f_{max} = 0.7$. The parameters of the physical interaction are the same as for Stickleg. In the frame 0, the hair hangs down due to gravity. All the strands are blendable and the last primitive in each strand is given a larger radius and is therefore heavier. As the dancer starts to lean to the side, the hair strands stretch within their geometric restriction limits (frames 1-5). At the same time, hair undergoes Lennard-Jones interaction, which causes it to join and separate into strands. For instance, in the frame 2 the strands of flesh are blended while in the frame 5 separate strands can be seen. Comparing Figure 4.15 and Figure 4.9, note how adding flesh to the model enhances the liveliness of an animation sequence, with no additional motion specification required.

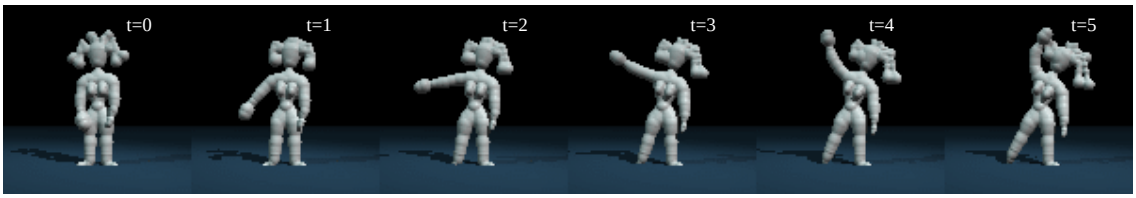


Figure 4.15: *A lean to the side specified using dance notation: hair is modelled using strands of flesh and follows the motion of the head*

The hybrid animation control method proposed offers two stages of creating an animation sequence. The first approximation of the required motion is achieved by animating object components. The final motion for the object is obtained when the motion for the flesh layer is automatically generated based on th component motion and a set of control parameters. The flesh behaviour can be refined by adjusting these parameters and regenerating its motion until the desired effect is reached. The following section will look into ways of controlling the adjustment of parameters in order to add traditional animation effects to the behaviour of flesh.

4.10 Traditional animation effects

In the hybrid animation method proposed large parts of motion are generated automatically. It allows the animator to focus on the general aspects of motion (animating components) and leave the software to produce the detailed motion accordingly (animating flesh). It is essential however to provide the animator with a means to refine the motion proposed by the software. One way to achieve such a refinement is to directly manipulate each of the available control parameters. This section proposes heuristics that express some traditional animation effects (Squash and Stretch, Follow Through and Exaggeration) in terms of control parameters. The animator is thus given a higher level of control in which the amount of the traditional animation effects in the final animation can be increased or decreased.

Such a manner of parameter adjustment can be called "knob" control. The aim is to give the animator a set of abstract "knobs" that express high level qualities of an animation sequence, *e.g.* the amount of exaggeration. Such an abstract quality can be translated into lower level control parameters and increasing or decreasing it means the appropriate adjustment of these parameters. One implementation of such a mapping is to represent each abstract quality ("knob") by the $[-1, 1]$ interval with zero marking the reference "normal" setting for all relevant parameters, specified by the animator, and provide a function that for each value $k \in [-1, 1]$ gives a value of a parameter p within its limits $[p_{min}, p_{max}]$. This correspondence can

be given by:

$$p(k) = -0.5k\,(k-1)\,p_{min} - (k-1)\,(k+1)\,p_{init} + 0.5k\,(k+1)\,p_{max}$$

where p_{init} is the initial value of the parameter p. It is a non-linear mapping and it can be applied for any parameter with finite limits p_{min}, p_{max} and for any initial value p_{init} (Figure 4.16).

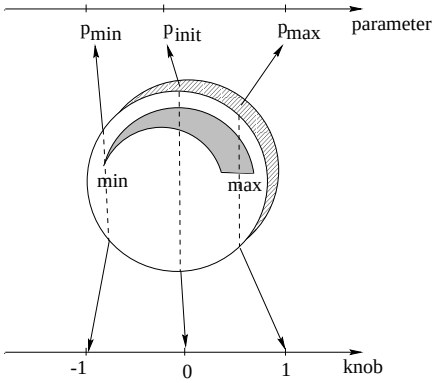


Figure 4.16: *Knob control of animation parameters*

Increasing an abstract quality may mean a decrease of a parameter's value. For instance, if the abstract quality if "lightness", the decrease in gravity will increase the level of lightness. To map a decrease of a parameter p to the $[-1, 1]$ interval, p is considered as a parameter $p' \in [-p_{max}, -p_{min}]$ with the initial value $-p_{init}$. For any given $k \in [-1, 1]$ the corresponding value of the parameter p is $p(k) = -p'(k)$ where $p'(k)$ is the mapping for p'.

The traditional animation qualities that are implemented in this thesis are: Squash and Stretch, Follow Through and Exaggeration. Heuristics converting each of these three qualities into the parameters of the hybrid animation control proposed are detailed in the following sections.

4.10.1 Squash and Stretch

The amount of squash and stretch in a character's behaviour can be adjusted by controlling the firmness of the object components. Two parameters specify firmness: f_{min}, which decides the allowed amount of squash that can be performed on the character, and f_{max} which decides the allowed amount of stretch (see Section 3.4). The range goes from a "loose" character for low values of both firmness parameters, to a "firm" character for high values. Special effects, *e.g.* a character that can easily be squashed but not stretched are also possible. Thus, the Squash and Stretch "knob" maps the $[-1, 1]$ interval into the values of f_{min} and f_{max} for object components.

In Figure 4.17 Stickleg is dancing a samba, choreographed using the coherence preserving algorithm, *i.e.* by pulling and pushing the *hip* joint sideways. The values of firmness parameters are: $f_{min} = 0.8$ and $f_{max} = 0.6$, which means that the primitives are not allowed to get too close to each other (f_{min} high) but they can get away from each other (f_{max} medium). When *hip* passes through its middle point, it pushes *stick* because the component primitives get too close to each other. As a result the character stretches upwards (frames 3,9,14). In the left or right extreme of *hip*'s motion, it pulls both *stick* and *leg* when the distance between the component primitives becomes too big. Thus, the character squashes (frames 6,12,16). The motion of flesh is calculated according to the motion of components and will be discussed in the next section.

4.10.2 Follow Through

Whereas for Squash and Stretch the firmness for object components was adjusted, the effect of Follow Through is achieved by controlling the behaviour of the flesh. More Follow Through means that flesh follows the motion of components while staying further behind. This can be achieved by decreasing flesh firmness, *i.e.* by setting both f_{min} and f_{max} to low values. The flesh with such a characteristic will allow

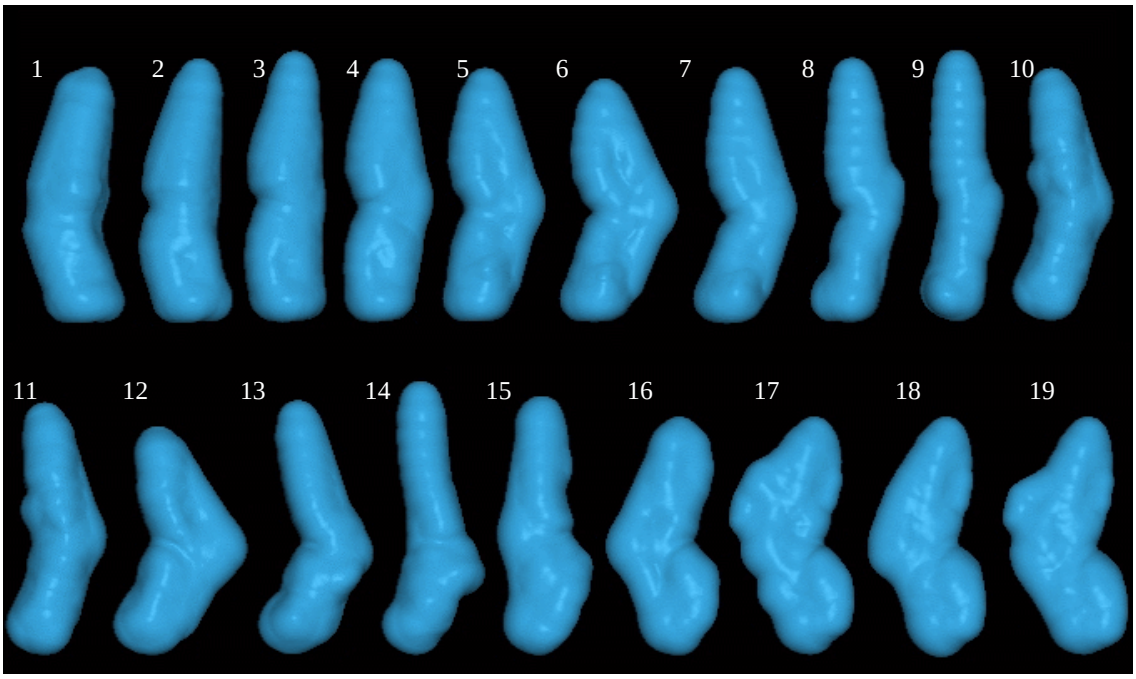


Figure 4.17: *Stickleg dancing a samba: the character stretches and squashes and flesh is following the motion of the hip joint to the left and to the right (follow through)*

the components to get close to it before being "pushed" and to get further away before being "pulled". This is the effect sought for Follow Through. Frames 1-10 in Figure 4.17 show how the components push the flesh (frames 1-4), then start to pull it behind them (frames 5-6) and push them again when the direction of the *hip* movement changes (frames 7-10).

Another quality of the flesh with more Follow Through is that it continues its motion for longer after components stop moving. This can be achieved by using a larger damping factor. In Figure 4.17 this effect can be seen in frames 16-19. Frame 16 is the last frame, for which the motion of components occur, the next three frames present the flesh that bounces around static components until it slowly comes down to rest in frame 19. The effect was generated using a high damping factor $D = 0.9$.

Various Follow Through effects can also be created by increasing the mass of a flesh primitive or the gravity in the environment to obtain heavier or lighter flesh. For higher values of gravity or for increased mass, the flesh will stay attracted towards the ground and for lower values of gravity, the flesh will be more bouncy, flying upwards. For instance, Figure 4.18 contrasts a jumping sequence for Stickleg with firmer flesh (Figure 4.18a) with its version with looser flesh (Figure 4.18b). Comparing frames number 7 in the two sequences shows how the looser flesh exhibits more follow through. It drops down to the ground and stays there even when Stickleg is still in the air (frames 7-12 in Figure 4.18b). To create the impression that the flesh is attached at the ends of the components, connections were created between flesh primitives in a form of chains (seen clearly in the frame 12 in Figure 4.18b). The flesh primitives at the end of each chain are attached by firm connections to the ends of the components and the rest of the chain hangs loosely below. The disadvantage of this approach is that naked components can be seen. For instance, frames 9-12 in Figure 4.18b show the naked *stick*.

Summarising, the Follow Through "knob" maps the $[-1, 1]$ interval into the values of firmness (f_{min} and f_{max}), the damping factor, gravity and flesh primitive mass. Each of the parameters is using its own mapping equation and the adjustment is performed for the same value of $k \in [-1, 1]$ resulting in a gradual change of all the initial parameter values. Since lower values of firmness are required to increase Follow Through, the decreasing mapping is used for f_{min} and f_{max}.

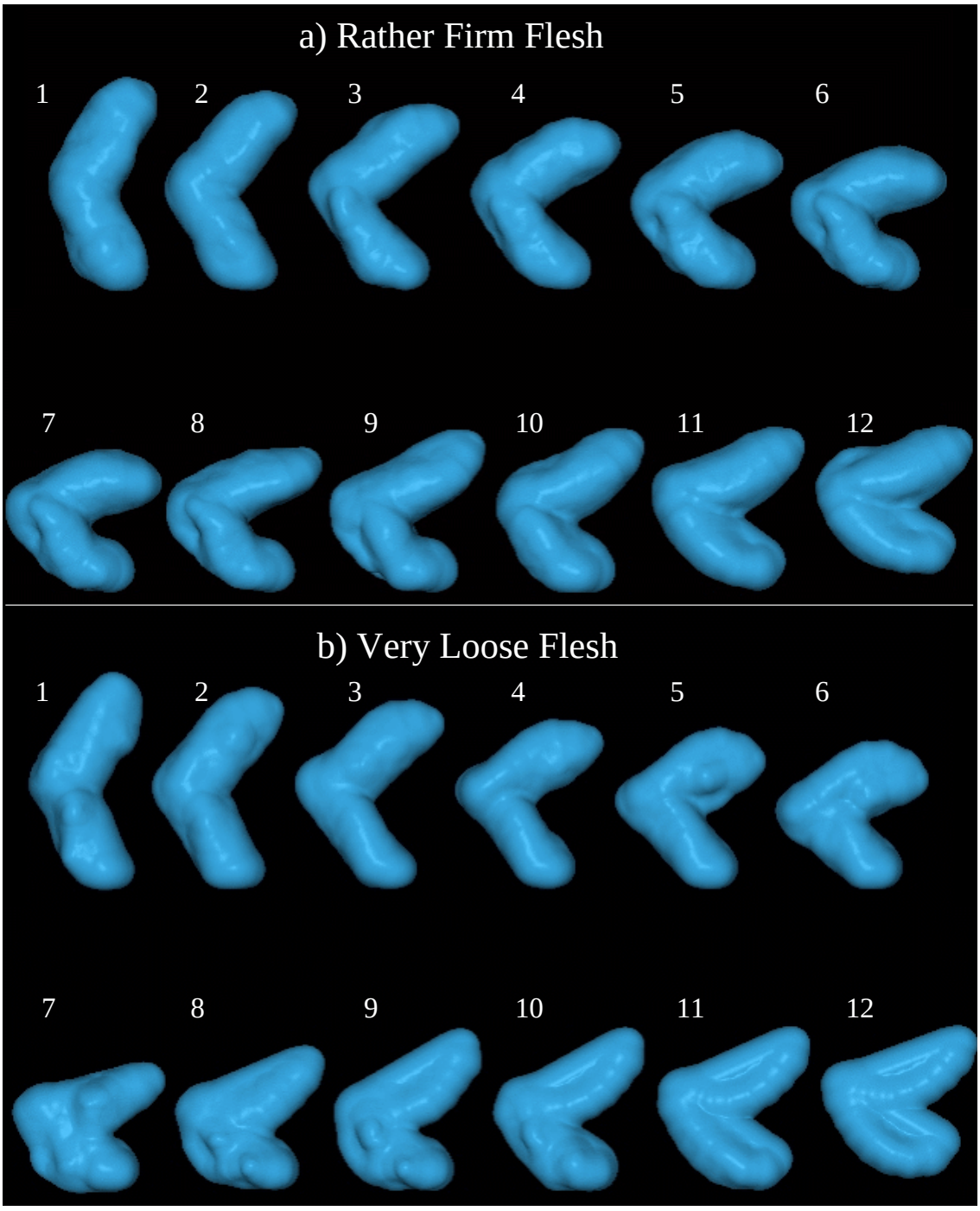


Figure 4.18: *Stickleg is jumping with a) firm flesh and b) with very loose flesh, that drops on the floor before following the motion*

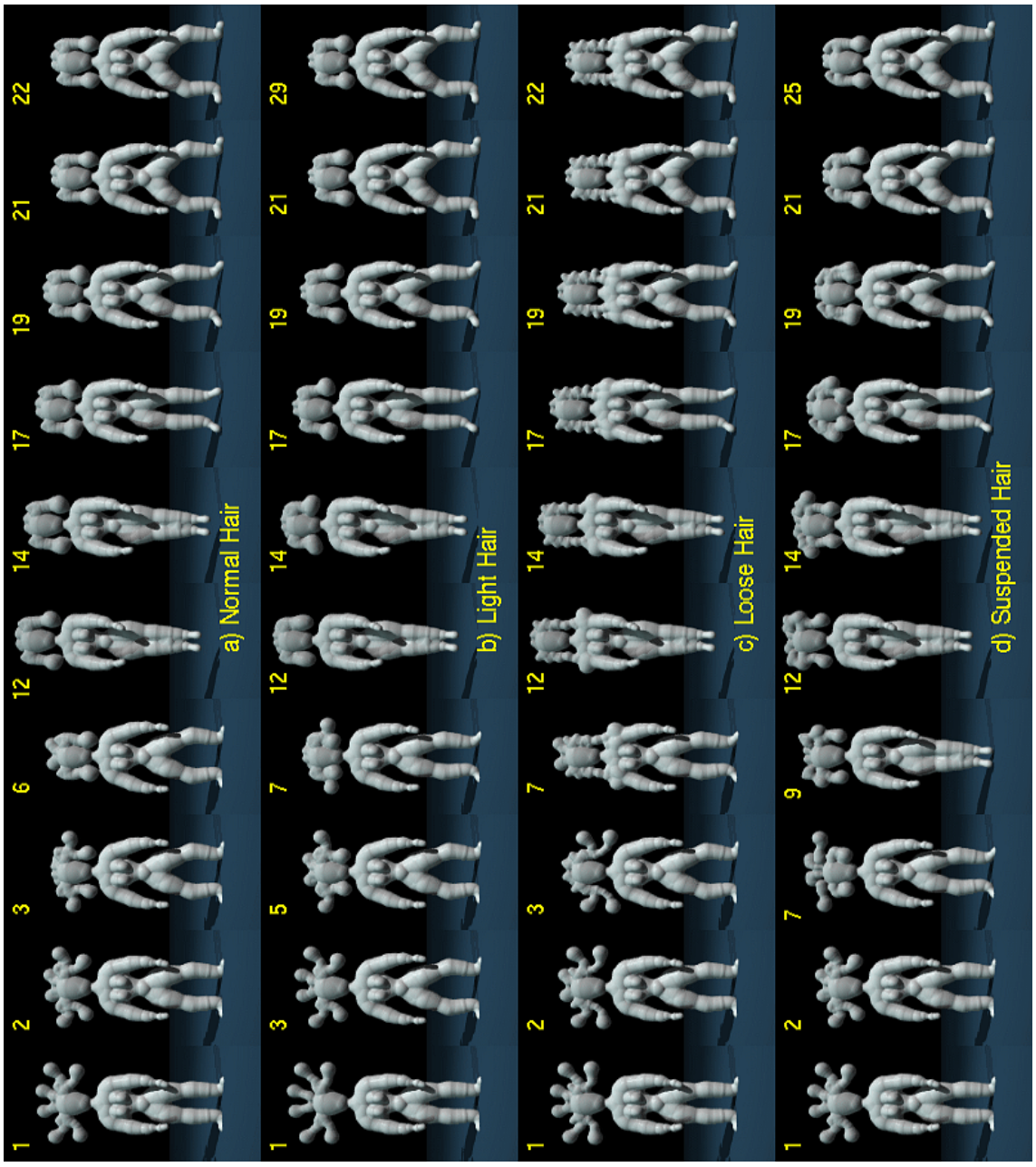


Figure 4.19: *Tuning the traditional animation effects: a) a reference animation with "normal" flesh, b) the flesh can get "lighter", c) "looser" or d) "suspended"*

4.10.3 Exaggeration

According to traditional animation principles [Thom81] any action is more convincing when exaggerated. The Exaggeration "knob" therefore converts the current animation sequence into the same sequence with all its qualities (parameters) increased (the "less" direction) or decreased (the "more" direction). The set of all initial parameters is used to mark the zero position, *i.e.* the sequence at its normal, not exaggerated state. The animator needs to specify if for a particular parameter the increasing or the decreasing mapping should be used.

Figure 4.19 shows the variety of animation effects that can be achieved using parameter control. It shows a Dancer, modelled in Figure 4.4, with strands of flesh representing hair. Figure 4.19a will serve as a reference sequence. It was animated using the dance notation, as in the example in Figure 4.11 with arm movement removed to show the hair motion more clearly. The direction of strands during their generation was set to the direction outwards from the head to avoid interpenetration between flesh and components (frame a1). During frames a2-a6 the Dancer bends her knees in preparation for a jump while her hair falls down under gravity. During the motion up (frames a6-a12) the hair continues to fall down. When the Dancer returns to the ground (frames a14-a21), the hair moves slightly upwards due to the head movement. Finally during the last two frames, a21-a22, the Dancer has completed the jump and the hair bounces gently sideways in the follow through stage.

Figure 4.19b shows a sequence of a Dancer with "light" hair. It has been achieved by decreasing the gravity of the scene to $\overrightarrow{g} = (0, 5, 0)$. The hair stays longer in the air before falling down (frames b1-b7) and during the jump (frames b7-b12). During the follow through phase (frames b21-b29), the hair stays further from the head and higher above the shoulders.

In Figure 4.19c the hair is "loose", *i.e.* it has low level of firmness ($f_{max} = 0.1$). The strand primitives can stay further from each other, creating longer and thinner chains of primitives.

Figure 4.19d presents a "special effect" - hair that stays suspended in the air before falling down. It has been achieved by using a combination of a negative value for the damping factor $D = -0.1$ and a positive value $D = 0.5$. The values are alternated every other time step. When the negative damping factor is used, instead of changing the direction when reaching the minimum of maximum coherence preserving distance, the flesh primitive attempts to continue motion in the same direction, being dragged back in the next time step, when the damping factor changes. This effect results in hair staying above the head, not falling flatly on it.

4.11 Dino: a case study

The animation process starts from a thought to be conveyed through an animated sequence. The initial stages include pencil studies for the characters and rough sketches for the main actions in the sequence, a storyboard. Then, the characters are modelled using a chosen modelling technique and the sequences from the storyboard are developed using a variety of animation control methods. This section will present the development of an animation sequence from the initial ideas using the layered model proposed for in the realm of implicit surfaces. The components of the chosen character, Dino, will be animated by applying the Eshkol-Wachmann notation to the articulated figure (appearance graph) of the character, including new "limbs": the neck and the tail of a dinosaur. The flesh motion will be generated by using a combination of the geometric restriction method and physical interaction between primitives. The animation sequence can then be refined by adjusting the amount of traditional effects in it.

4.11.1 Designing the character

After a few initial charcoal sketches, the characteristic parts of the character were transformed into an implicit surface with blending properties, *i.e.* the appearance graph to be used for the character was chosen (Figure 4.20). The factors which were taken into consideration are: what are the important features of the character (*i.e.* what makes up its appearance), where will bending occur in the character (*i.e.* where are the joints) and how much flexibility is needed in each limb (*i.e.* how many joints are in a limb). The chosen character has a long bendable neck (3 joints) and a long bendable tail (3 joints).

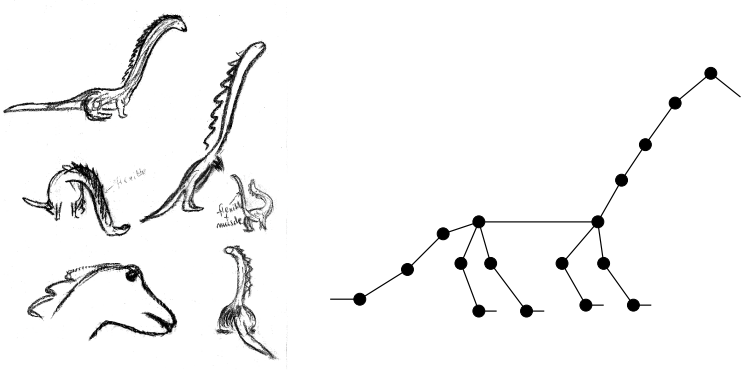


Figure 4.20: *Initial sketches and the designed appearance graph*

4.11.2 Modelling the character

The next stage is converting the designed character into a model. In the traditional 3D character animation, plasticine or clay would be used to create it. For implicit surfaces, each component in the appearance graph has to be modelled using two layers of primitives: a component layer which defines the general look of the bone and muscle structure for the character, and a flesh layer which models soft and loose parts, *e.g.* hair or tail. Each limb is defined in its local coordinate system and then the complete model is assembled by constraining the limbs to meet at relevant joints. In the prototyping stage, components alone may be used for motion definition, making the experiments faster to perform. Flesh can be added later when the motion for components is decided.

Modelling components

A simple approach to model bone/muscle would be to replace each link in the appearance graph (Figure 4.20) by a line segment generator and each joint by a point generator. It gives a good starting point for the appearance of each limb. It can be treated as the bone structure and the muscle layer has to be added around each such bone to shape it into the required look for the limb. For some applications, *e.g.* when modelling a toy dinosaur, the crude approximation may even be sufficient. However, if a more sophisticated dinosaur is required, each component has to be manually refined. Figure 4.21 shows the script used to produce the bone/muscle layer for the dinosaur in Figure 4.22a. The limbs were approximated by chains of point generators rather than line segments.Figure 4.22c shows its 2D cross section with primitives marked as spheres.

```
iso 0.4

object dino colour Wheat
    component head rotation (0,0,0)
        point (16,-19,0) POVfunction 40 1
        point (36,-34,0) POVfunction 20 1
        blend SUM
    component neck1 rotation (0,0,0)
        point (30,30,0) POVfunction 40 1
        blend SUM
    component neck2 rotation (0,0,0)
        point (18,31,0) POVfunction 40 1
        point (43,55,0) POVfunction 40 1
        blend SUM
    component neck3 rotation (0,0,0)
        point (12,36,0) POVfunction 40 1
        point (23,72,0) POVfunction 40 1
        blend SUM
    component trunk rotation (0,0,0)
        point (53,1,0) POVfunction 80 1
        point (108,1,0) POVfunction 80 1
        blend SUM
    component tail1 rotation (0,0,0)
        point (-67,-58,0) POVfunction 45 1
        point (-35,-31,0) POVfunction 45 1
        point (0,0,0) POVfunction 50 1
        blend SUM
    component tail2 rotation (0,0,0)
        point (-55,-22,0) POVfunction 35 1
        point (-28,-15,0) POVfunction 35 1
        blend SUM
    component tail3 rotation (0,0,0)
        point (-28,-3,0) POVfunction 30 1
        blend SUM
    component l_fleg rotation (0,0,0)
        point (25,-87,0) POVfunction 10 1
        point (14,-82,0) POVfunction 20 1
        point (3,-65,0) POVfunction 25 1
        point (-4,-44,0) POVfunction 25 1
        point (0,-22,0) POVfunction 30 1
        blend SUM
    component l_bleg rotation (0,0,0)
        point (3,-74,0) POVfunction 12 1
        point (-11,-72,0) POVfunction 15 1
        point (-31,-67,0) POVfunction 30 1
        point (-6,-46,0) POVfunction 35 1
        point (16,-24,0) POVfunction 35 1
        blend SUM
    component r_fleg rotation (0,0,0)
        point (25,-87,0) POVfunction 10 1
        point (14,-82,0) POVfunction 20 1
        point (3,-65,0) POVfunction 25 1
        point (-4,-44,0) POVfunction 25 1
        point (0,-22,0) POVfunction 30 1
        blend SUM
    component r_bleg rotation (0,0,0)
        point (3,-74,0) POVfunction 12 1
        point (-11,-72,0) POVfunction 15 1
        point (-31,-67,0) POVfunction 30 1
        point (-6,-46,0) POVfunction 35 1
        point (16,-24,0) POVfunction 35 1
        blend SUM
    component top_spine rotation (0,0,0)
        point (0,0,0) POVfunction 20 1
        blend SUM
    component neck_j1 rotation (0,0,0)
        point (0,0,0) POVfunction 40 1
        blend SUM
    component neck_j2 rotation (0,0,0)
        point (0,0,0) POVfunction 40 1
        blend SUM
    component low_neck rotation (0,0,0)
        point (0,0,0) POVfunction 45 1
        blend SUM
    component back rotation (0,0,0)
        point (0,0,0) POVfunction 80 1
        blend SUM
    component tail_j1 rotation (0,0,0)
        point (0,0,0) POVfunction 40 1
        blend SUM
    component tail_j2 rotation (0,0,0)
        point (0,0,0) POVfunction 30 1
        blend SUM
    component l_fhip rotation (0,0,0)
        point (0,0,0) POVfunction 35 1
        blend SUM
    component l_bhip rotation (0,0,0)
        point (0,0,0) POVfunction 40 1
        blend SUM
    component r_fhip rotation (0,0,0)
        point (0,0,0) POVfunction 35 1
        blend SUM
    component r_bhip rotation (0,0,0)
        point (0,0,0) POVfunction 40 1
        blend SUM
    joint top_spine between neck1 (20,20,0) head (0,0,0)
    joint neck_j1 between neck1 (0,0,0) neck2 (54,50,0)
    joint neck_j2 between neck2 (0,0,0) neck3 (40,103,0)
    joint tail_j1 between tail1 (-97,-86,0) tail2 (0,0,0)
    joint tail_j2 between tail2 (-85,-25,0) tail3 (0,0,0)
    joint trunk between neck3 (0,0,0) low_neck (-140,-45,0)
    joint trunk attachedto back (0,0,0)
    joint trunk attachedto tail1 (-30,12,0)
    joint trunk attachedto l_fhip (-105,50,30)
    joint trunk attachedto l_bhip (-25,60,40)
    joint trunk attachedto r_fhip (-105,50,-30)
    joint trunk attachedto r_bhip (-25,60,-40)

    anchor back (0,0,0)

    skin
        radius 30
        cover trunk neck2 neck3 tail1 tail2 tail3
end
```

Figure 4.21: *The modelling script for Dino*

Modelling flesh

In this step, the soft and "floppy" parts of Dino, *e.g.* hair or loose skin, are modelled. Coats of flesh on the neck, trunk and tail are added to make the Dino's body deformable during animation. There are 45 primitives in all components. 153 flesh primitives were used to generate the flesh layer. Flesh radius 50 was used, which is greater than most of the radii of component primitives. Using larger flesh primitives produces a less "bumpy" surface. Figure 4.22b shows the final dinosaur and Figure 4.22d shows its 2D

cross section with primitives marked as spheres.

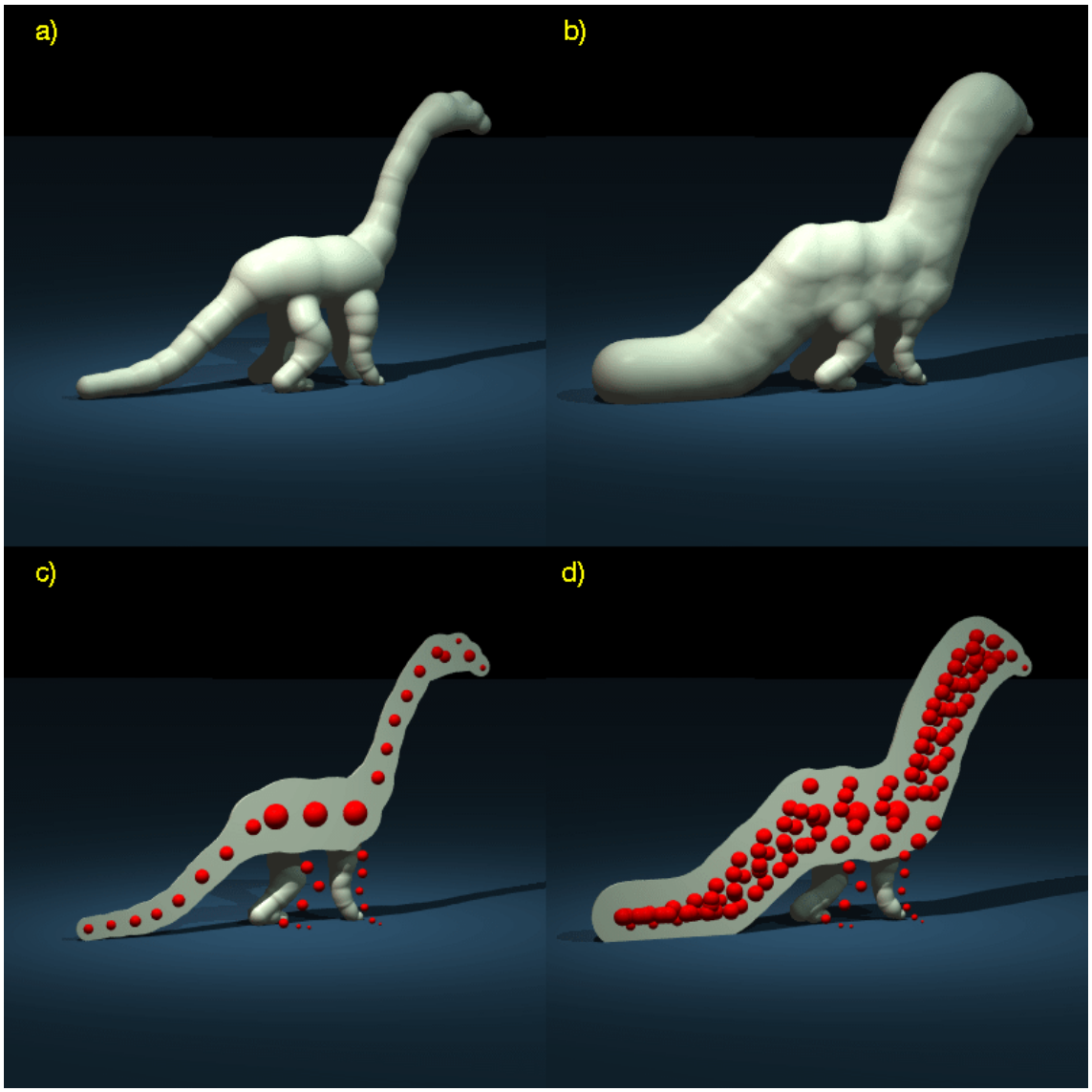


Figure 4.22: *Modelling the Dino: a) components b) components with flesh c) a 2D cross section showing primitives in the components d) a 2D cross section showing primitives in the components and flesh*

4.11.3 Animating the character

This is the most complex stage of the animation process. It should be ensured that, instead of a photo realistic look, the motion is used to convey a character's personality to the audience. The use of traditional animation effects, such as squash and stretch, follow through or exaggeration results in a more captivating animation sequence. This stage is performed in three steps: drawing a storyboard, specifying motion elements for the articulated figure and generating flesh motion. The last step may be adjusted until the satisfactory motion for the flesh is obtained.

Storyboard

The goal of an animation sequence should be defined first. It is typically sketched out as a storyboard, in which the story to be told is constructed. The emotion to be expressed in each scene is suggested in the storyboard. Figure 4.23 presents the storyboard sketched for this animation. The dino-dance movements performed by the character are clearly defined in each frame.

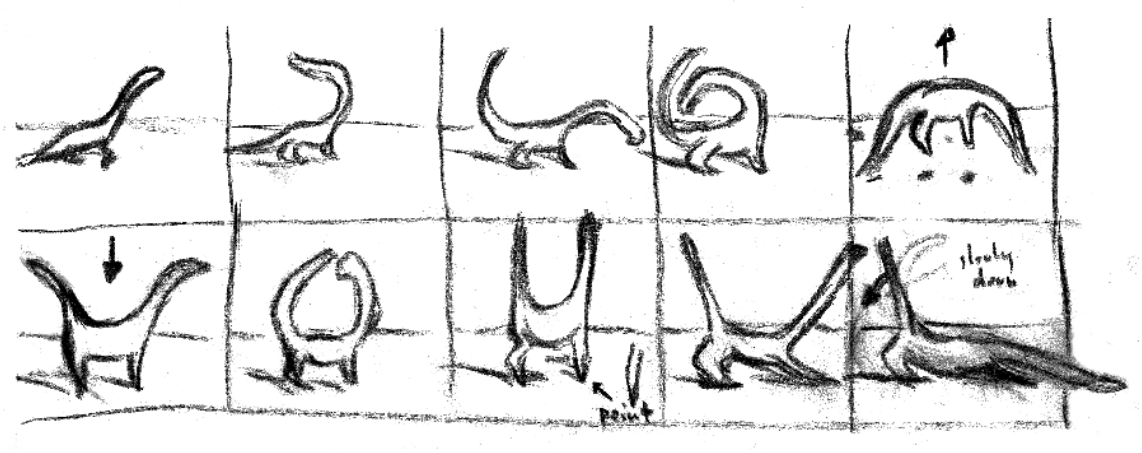


Figure 4.23: *A storyboard*

Component motion

In this stage, the animator can focus on choosing ways of achieving emotions for each motion element. Motion elements for the articulated figure are specified using the Eshkol-Wachmann dance notation, applying it to all Dino's components, including the tail and the neck components. Such an extended notation can be in this particular case called a “dinotation”. Figure 4.24 presents the dinotation script for the motion in Figure 4.25. It describes the first few movements in the storyboard.

```
object dino
times 1 2
    plane neck1 (30,0)
times 2 4
    plane neck2 (45,0)
times 3 6
    plane neck3 (60,0)
    plane head (60,0)
times 6 8
    translate trunk (0,-10,0)
    plane r_fleg (-30,90)
    plane r_bleg (-30,90)
    plane l_fleg (30,90)
    plane l_bleg (30,90)
    plane neck1 (-30,0)
    plane neck2 (-45,0)
    plane neck3 (-60,0)
    plane head (-60,0)
times 6 7
    plane tail1 (-30,0)
times 7 8
    plane tail2 (-30,0)
times 8 10
    translate trunk (0,10,0)
    plane r_fleg (20,90)
    plane r_bleg (20,90)
    plane l_fleg (-20,90)
    plane l_bleg (-20,90)
    plane tail1 (-30,0)
    plane tail3 (-45,0)
times 9 12
    translate trunk (0,-15,0)
    plane r_fleg (-20,90)
    plane r_bleg (-20,90)
    plane l_fleg (20,90)
    plane l_bleg (20,90)
times 13 15
    translate trunk (0,20,0)
    plane neck1 (-30,0)
    plane neck2 (-45,0)
    plane neck3 (-30,0)
    plane head (60,0)
    plane tail1 (60,0)
    plane tail2 (30,0)
    plane tail3 (45,0)
end
```

Figure 4.24: *The animation script for Dino*

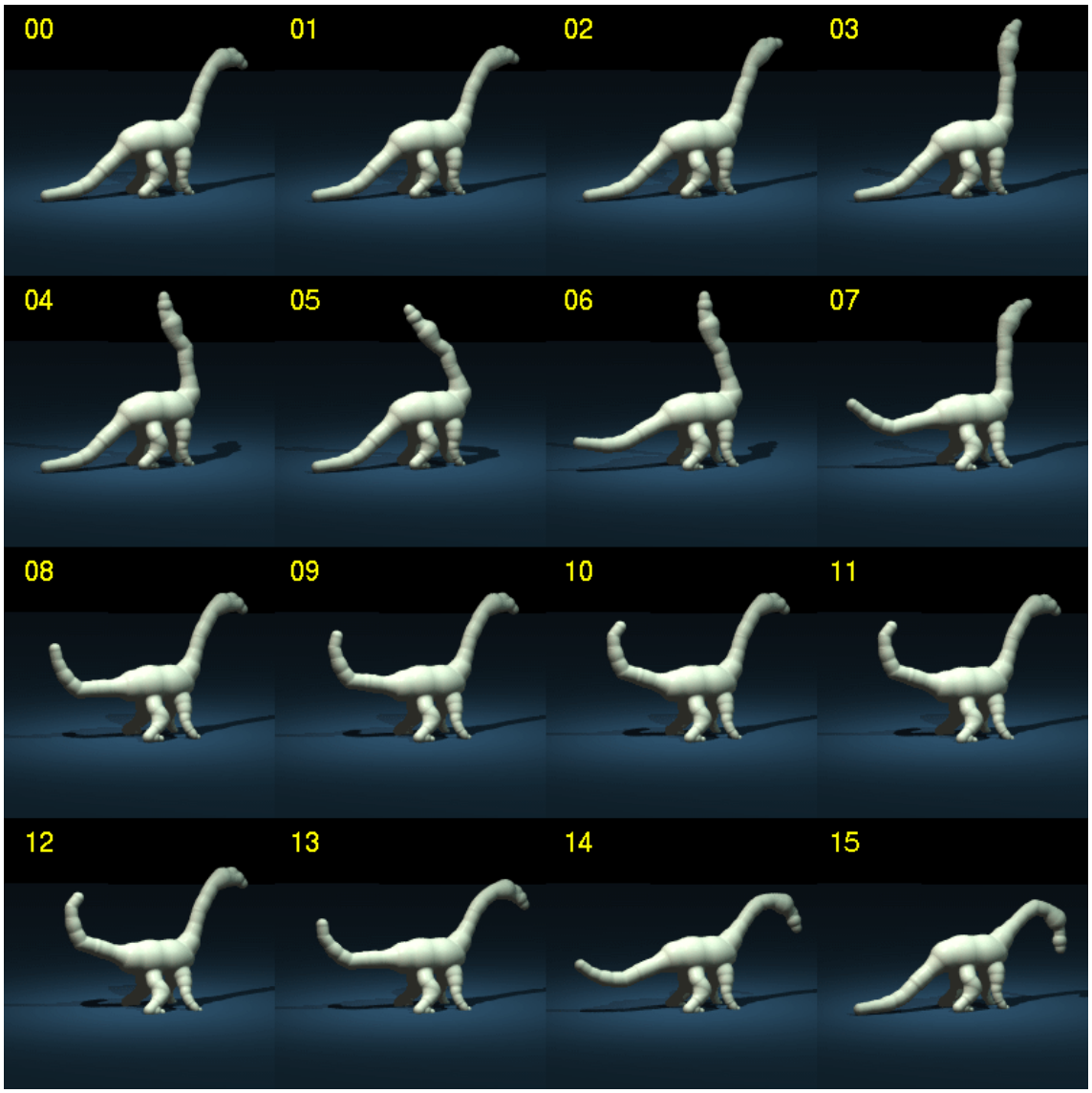


Figure 4.25: *A dancing dinosaur: no flesh*

Flesh motion

Animating the dinosaur modelled using only components results in very little deformation and a quite rigid behaviour of the character. For instance, in frame 12 in Figure 4.25 the individual links in the tail are bent showing the articulations. When the components are covered with a layer of flesh, such artifacts do not appear. Figure 4.26 presents the same animation sequence with added flesh motion. The flesh firmness is set to $f_{min} = 0.9$ and $f_{max} = 0.5$. Gravity use in the scene is $(0, 5, 0)$, therefore the flesh stays above components during its motion (*e.g.* the flesh on the tail in frames 08-12). The "bubbling" effect is due to the size and distribution of flesh primitives that were used. A more dense flesh or larger flesh primitives would produce a smoother result.

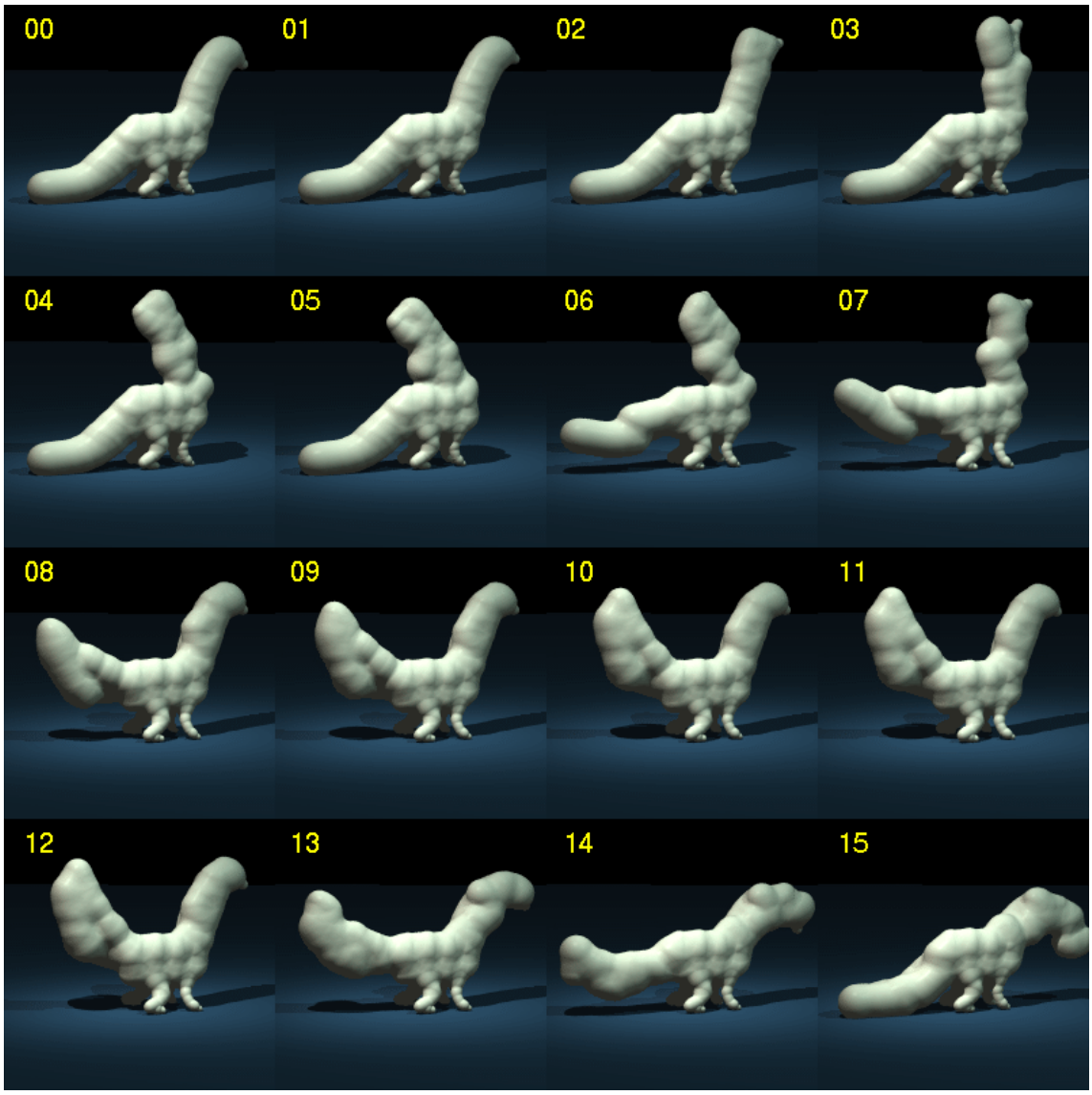


Figure 4.26: *A dancing dinosaur: with flesh*

4.12 Summary

This chapter introduced a layered model for character animation based on implicit surfaces with blending properties. Three layers in the model are: user-defined object components, user-controlled automatically generated flesh and automatically created implicit skin. The model offers the advantages of a layered model, *e.g.* the possibility of designing, viewing and animating the model at various levels of detail. Additionally, it offers the advantages of implicit surfaces, *e.g.* simple deformation, precise collision and self-collision modelling and automatic continuous skin generation.

A hybrid motion control, *i.e.* a combination of manual and automatic control, is provided to animate this model. An animator specifies motion for object components of a character using a dance notation. Based on this motion, the system generates flesh motion. Skin motion does not need to be calculated since it automatically deforms and fits the time-varying configuration of components and flesh according to the extended model for implicit surfaces with blending properties described in Chapter 3. An animator can change a set of parameters that control the automatic motion generation to refine the final behaviour of the character. This adjustment can be performed at a low level of abstraction, by directly manipulating all

available parameters. A higher level of adjustment control is also provided. It translates the parameters controlling the flesh motion generation into an abstraction for traditional animation effects: Squash and Stretch, Follow Through and Exaggeration. For instance, a request for more follow through is automatically translated into decreasing the firmness of flesh, increasing the damping factor, increasing gravity and increasing flesh mass. These traditional animation effects are incorporated automatically and the animator has control over their amount. The animator can also switch between high and low levels of animation control by adjusting both abstract qualities of the flesh motion and individual control parameters.

Such a hybrid method offers a balance between manual animation control, when total control and motion specification is left to the animator, and automatic animation control, when the system generates the motion based on some general rules given by the animator. It provides a way of animating a multi-layered model with only a few parameters to control and it allows the animator to describe the animation in abstract terms.

Chapter 5

Discussion and Future Work

5.1 Introduction

In this chapter, the experience gained through this research will be put in a broader context, attempting to establish the current and future role of skeleton based implicit surfaces in modelling and character animation. Some improvements to the methods developed will be proposed and potential applications of the results achieved will be discussed.

Skeleton based implicit surfaces offer a natural layered approach to modelling. An object is specified as a set of geometric generators, each of them defining a scalar field around itself. The surface of the object is represented by an isosurface of the global scalar field, calculated by combining the scalar fields around generators. Any type of geometric generators, *e.g.* points, line segments, curves or polygons, can be used. They create an intuitive *skeleton* for an object. The resulting surface is smoothly wrapped around such a skeleton, giving the object an attractive, "organic" look. Arbitrary surface topology for an object can be achieved. An implicit model is compact, because only the description of the primitives has to be stored. It also facilitates rapid prototyping of objects for design or animation, since most of the initial work can be done in real time by manipulating skeletons only and the surfaces can be added later. Another advantage of implicit surfaces is their deformability. Surface deformation can be easily performed by changing the position or the radius of influence of some primitives in a model. Motion fluidity can be introduced to the behaviour of a model in this way. Implicit primitives can be moved within a certain limit without the loss of the topological integrity of the model. These qualities of implicit surfaces were the initial motivation for their use in the area of computer graphics investigated in this research: character modelling and animation.

One of the essential qualities of a character is its *appearance*, *i.e.* a set of characteristic features, both static (look) and dynamic (behaviour), that make it recognisable throughout its life. The first part of this research dealt with a definition and conservation of such characteristics for models constructed using implicit surfaces (Chapter 3). As a result, an extended implicit model has been proposed. It defined the characteristic parts of a character using an appearance graph. It offered solutions to the problem of these parts blending together (*unwanted blending*) and the problem of them separating and thus destroying the topological coherence of the character (*coherence loss*). In Section 5.2 these aspects of modelling using implicit surfaces will be discussed. The general advantages of modelling a character using a layered approach will be presented and some particular problems encountered for a layered implicit model will be discussed. Some generalisations of the algorithms proposed in this thesis will be presented and a few attractive extensions, including tangent continuous deformation and volume preserving deformation will be proposed. Finally, two ways of specifying a model, a script file and an interactive editor, will be contrasted and their relationship examined.

A character has to be coloured and textured to appeal to the audience as a convincing personality. These are aspects of rendering and they remain an open research area for implicit surfaces. Before moving to the issues of animating implicit surfaces, some of the problems of texturing will be discussed in Section 5.3.

The experience of traditional character animation was an inspiration for the second part of this research that dealt with animation control of implicit surfaces (Chapter 4). The aim was to introduce traditional animation effects to computer generated animation. The hybrid control proposed offers a balance between

automatic (high level) and manual (low level) animation control. In this thesis traditional animation effects are achieved automatically and their extent is fine-tunable by the user. Section 5.4 will present a reflection on the animation control of implicit surfaces, in particular hybrid animation control of a multi-layered model. The possibility of simplifying a model and using multiple levels of detail during character design, animation prototyping and interactive animation will be considered. A suggestion of an application of the approach presented in this thesis to the task of 3D metamorphosis of deformable objects will also be presented. Finally, a possible combination of implicit surfaces with other modelling and animation environments, *e.g.* the interaction between smooth and sharp objects, will be discussed.

5.2 Modelling using implicit surfaces

Most life forms possess some kind of a skeleton. A natural approach to modelling living or animated objects is therefore to use skeletal design, *i.e.* to specify a skeleton for an object and have software automatically clothe it with a surface. The skeleton can then be moved during the object's life and the surface always follows this motion, deforming accordingly. Implicit surfaces are a good way of modelling such skeletal objects, including forms with arbitrary topology. They also offer a compact computer representation, and, because of their multi-layered nature, allow for fast prototyping of models since not all layers have to be displayed.

5.2.1 Layered character modelling

Using a layered model is particularly suitable for computer character modelling. The first layer forms the skeleton for a character, typically an articulated structure of links and joints. Then one or more of the following layers are added to sculpt the body for the character: bones, muscles, fat tissue, skin. This strategy, based on the skeletal nature of life forms, may suggest that only characters existing in the world of animals can be modelled. However, the analogy can be taken further to model inanimate forms in order to *animate* them, *e.g.* designing the skeleton, the muscles and the skin for a tree or a table. A character would usually be “humanised”, *i.e.* it will have in its skeleton equivalents of legs, arms and heads.

The most common approach to creating layered characters in computer animation is to represent the skin as a polygonal or parametric surface and fit it around an articulated structure [Magn88, Chad89, Fors91]. The skin deforms during each movement of the skeleton thus modelling skin effects, *e.g.* creases and skin folds. The skin layer is often also used to model the deformation due to the muscles, *e.g.* bulging during muscle flexion, without the explicit muscle layer representation. The deformable muscle layer may be represented explicitly in the model as in [Shen95] where implicit surfaces are used to model muscles. These skin models are geometric, elastic skin can also be modelled, incorporating some physics based rules of it deformation. Gascuel *et al.* [Gasc91] use a rigid bone layer and covers it with an elastic skin modelled with B-spline surfaces and attached to the bone layer by springs. Turner [Turn95] uses superellipsoids model muscle and fat tissue layers. He attaches an elastic polygonal skin layer to them by spring forces. The deformation of the skin is in these cases guided by the response obtained from the spring connections by applying the current acting forces to them.

The definition of an implicit surface uses the concept of a skeleton (a set of generators) and a skin (an isosurface). An advantage of this representation is that the skin is calculated automatically and intuitively around a given set of generators, with arbitrary topology supported. Creating the skin layer is therefore a simple procedure as opposed to the creation of a polygonal or a parametric surface around a skeleton which is required for existing layered character models. Another advantage is the simplicity of deforming an implicit surface which is achievable by changing the parameters that define implicit primitives. For instance, to model a bulging muscle, the radius of a primitive can be increased and the implicit surface will locally bulge within the primitive's radius of influence.

To benefit from these qualities, this thesis investigated the use of implicit surfaces for developing an alternative layered model for character animation. Initially, the properties of a simple model consisting of two layers (a set of generators and an isosurface) were examined. It was observed that objects modelled using this approach had little structure imposed on them. They were treated as a collection of primitives that blended with each other. At any time, these primitives could cluster together and blend into an amorphous shape. They could also move apart, causing the loss of the topological integrity of the surface. Imposing a

structure on a set of generators in a model using an appearance graph provides a basis for solutions proposed to these two problems: avoiding unwanted blending and preventing coherence loss. The new formulation for implicit surfaces, extended by these solutions, was named the ABC approach: Appearance, Blending and Coherence.

5.2.2 The ABC approach

An appearance graph serves the purpose of an articulated structure for a model. It consists of links which define the limbs of a character and joints which define the articulations between the limbs. It is used during the design stage of character modelling to define the characteristic parts of a character. Contrary to the existing layered models, an appearance graph does not define a rigid skeleton for an object. Therefore, the same appearance graph can be used throughout the animated life of a character, even if it undergoes extreme deformations, *e.g.* limb stretching which is an effect often required in cartoons.

To deal with unwanted blending, an extended formulation for implicit surfaces was introduced (see Section 3.3). It defined the blending properties for a scene and modified the way of calculating the global scalar field to model deformation due to collisions between objects and between parts of an object. Modelling precise contact surfaces in self-collisions is a novel development for computer animation.

Although simple primitives and simple blending were used, the proposed solution to the unwanted blending problem, *i.e.* a model for implicit surfaces with blending properties, is extensible to distance surfaces defined around any geometric skeletons, *e.g.* line segments, curves or polygons. It is not applicable to other blending methods because the algorithm for calculating the deformed surface relies on summation to provide surface continuity.

Extending the unwanted blending algorithm

The extension of the formulation for implicit surfaces with blending properties (Section 3.3) is trivial. The proposed solution is based on regions of influence and scalar field values at a given point P. The field value at P due to a deformed component C, $F_{C_{def}}(P)$, is calculated according to the following algorithm (Section 3.3):

$$\begin{array}{lll}
\text{if} & P \notin \mathcal{M}(\mathcal{C}) & \text{then } F_{C_{def}}(P) = 0 \\
\text{elsif} & \exists_{C_u \in \mathcal{U}(\mathcal{C})} F_{C_u}(P) \text{ ¿ } Iso \wedge F_C(P) \text{ ¿ } Iso & \text{then } F_{C_{def}}(P) = F_C(P) + F_{\mathcal{B}(\mathcal{C})} - F_{\mathcal{U}(\mathcal{C})} + Iso \\
\text{else} & & F_{C_{def}}(P) = F_C(P) + F_{\mathcal{B}(\mathcal{C})}
\end{array}$$

where $\mathcal{M}(\mathcal{C})$ is the maximal influence area for C, $\mathcal{B}(\mathcal{C})$ is the set of components blendable with C, $\mathcal{U}(\mathcal{C})$ is the set of components unblendable with it, $F_C(P)$ is the undeformed field value for C, $F_{\mathcal{U}(\mathcal{C})}(P)$ is the collision term:

$$F_{\mathcal{U}(\mathcal{C})}(P) = \sum_{C_u \in \mathcal{U}(\mathcal{C})} F_{C_u}(P)$$

and $F_{\mathcal{B}(\mathcal{C})}(P)$ is the blending term:

$$F_{\mathcal{B}(\mathcal{C})}(P) = \sum_{\substack{C_b \in \mathcal{B}(\mathcal{C}) \wedge \\ \forall_{C_\sqcap \in \mathcal{U}(C_\sqcup)} \mathcal{F}_{C_\sqcap}(P) \leq \mathcal{F}_{C_\sqcup}(P)}} F_{C_b}(P)$$

For any component C, its undeformed field function $F_C(P)$ is calculated as:

$$F_C(P) = \sum_{i=1}^{n} f_i(P)$$

where $f_i, i = 1..n$ are field functions of primitives in the component. To use a new type of a primitive, only the calculation of the undeformed field function for a component changes. The proposed algorithm remains intact.

Although it has been developed for characters, the ABC approach can be used for other applications, for instance for simulating deformable objects that undergo collisions and self-collisions. For simulation purposes the two important areas where the methods proposed need to be further developed are tangent continuous deformation and volume conservation during deformation.

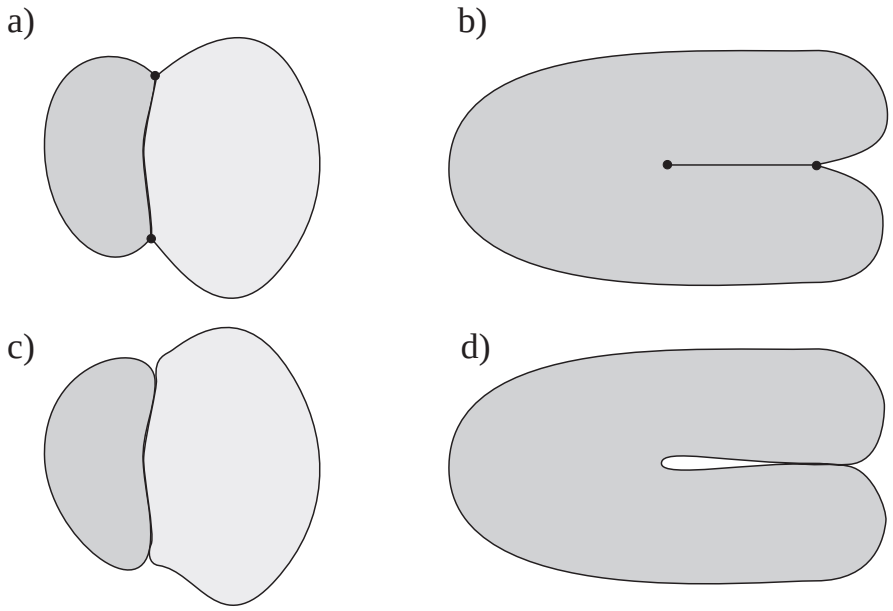


Figure 5.1: *Differences between continuous and discontinuous surface deformation*
a) Tangent discontinuity in a collision
b) Tangent discontinuity in a self-collision
c) Tangent continuous deformation during a collision
d) Tangent continuous deformation during a self-collision

Tangent continuous deformation

One problem with the proposed formulation is that during collisions, tangent discontinuities occur in the deformed surfaces. Figure 5.1a presents a diagram of the deformation during a collision and a self-collision. The points of tangent discontinuity are marked. Figure 5.1b shows a perhaps more natural behaviour of the deformed surface, with tangent continuity preserved. Marie-Paule Gascuel [Gasc93] proposed a method that added a displacement function to the scalar field to preserve C^1-continuity of deformation during collisions. The displacement function was calculated in a post-processing stage and it required calculation of the closest point on a surface, a computationally expensive operation. Moreover, collisions with the deformed surface could not be detected, since the field function calculation remained unchanged. To our knowledge, no deformation method that preserves tangent continuity has been proposed for self-collisions. This problem is one of the possible directions to explore in future research.

For instance, one possible solution to tangent discontinuity is the use of an alternative blending method. Figure 5.2 shows a few ideas that may serve as a starting point. The light grey areas mark the initial inside of an object with a tangent discontinuity (a "corner"). The dark grey areas show local blend surfaces that were added (Figures 5.2a, 5.2b and 5.2c) or subtracted from the initial surface (Figures 5.2d, 5.2e and 5.2f) to achieve tangent continuity of the surface. The final blended surface of the object is marked with lines. Any local blending method can be used, *e.g.* the method of Middleditch and Sears [Midd85], Rockwood and Owen [Owen89, Rock90a, Rock90b], Hoffman and Hopcroft [Hoff86] or Rossignac and Requicha [Ross84]. Each of these methods offers a way of controlling the range of blend, marked in Figure 5.2 by radii R_1 and R_2. Figures 5.2a, 5.2b and 5.2c present a sequence of blends applied to a "corner" of an convex object. The idea is to allow for the radius R_1 to take negative values and therefore add a blend that will create a tangent continuous surface which fits the "ground" (Figure 5.2c). This method can be used to create tangent continuous deformation during collisions.

Figures 5.2d, 5.2e and 5.2f illustrate a reasoning that can be applied to achieve tangent continuity during self-collisions. In Figure 5.2d, a local subtractive blend was created around the "corner" of a concave object. The idea is to apply such a blend to the "corner" while the two parts of the object approach each other and collide (Figures 5.2e and 5.2f). One problem that may occur is blend surface interpenetration.

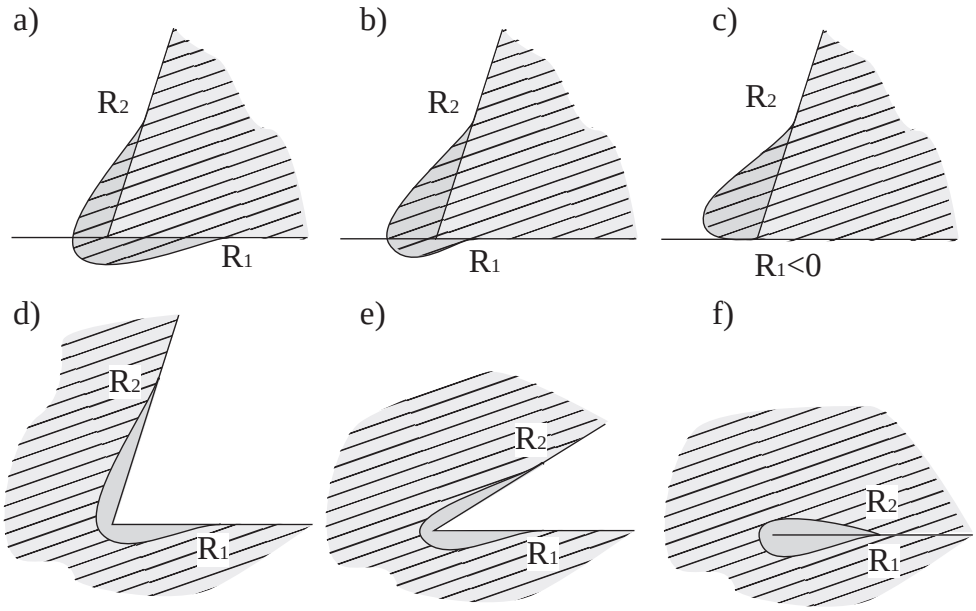


Figure 5.2: *A suggestion for tangent continuity conserving deformation*
a) b) c) during a collision
d) e) f) during a self-collision

Volume preserving deformation

A significant problem with the model presented in this thesis is that volume is not conserved during deformation. The first attempt to introduce volume conservation to animation of implicit surfaces was done by Wyvill *et al.* [Wyvi86a]. The isovalue, *magic*, was chosen so that the volume of two blended primitives is approximately equal to the sum of the initial two volumes. However, this solution does not generalise to more than two primitives. Moreover, it controls only the initial and the final frames of an animation. The volume in the inbetween frames is not controlled.

Desbrun and Gascuel [Desb95a] proposed a solution for any number of primitives. During surface sampling, a set of seeds is maintained on the implicit surface for each generator. These seeds can be used to define a set of "pyramids" with the top on a generator and the base on a surface triangle defined by the sampling points. The volume of an implicit object is approximated by the volume of such pyramids. It is maintained at a near constant level by adjusting the field function of primitives for which the sampling changes during animation. The volume is thus controlled locally.

As stressed in one of the traditional animation principles, a character needs to conserve its volume in order to preserve its personality [Thom81]. As an example, a series of drawings showing a half-filled sack of flour in different "moods" was shown to animators at the Disney studio. Squashed and stretched to express feelings, it always retained its half-filled state. A method that provides volume control should be introduced in the model proposed in this thesis to make the animations more convincing.

5.2.3 Specifying a model

To develop a model proposed in this thesis, a user needs to define a set of primitives in 3D space, form them into components and position the components in relation to each other to obtain parts of an appearance graph. Two possible contrasting methods of model specification are: (i) off-line batch processing of a modelling language script and (ii) on-line interactive design of a model. The former method can handle any size of models but does not offer an interactive environment. The latter gives an immediate visualisation of the form being designed but the more complex the object the more computationally intensive its display.

The two methods of model specification are closely related (Figure 5.3). A model in both cases has to be stored in a file, which can either be a simple script file or can contain some programming commands, *e.g.* iteration. Both an interactive modelling program and a text editor can load existing models, modify

them and save them. An often used approach is to use an interactive modeller to specify the first version of a model and then to manually modify the resulting modelling file in a text editor.

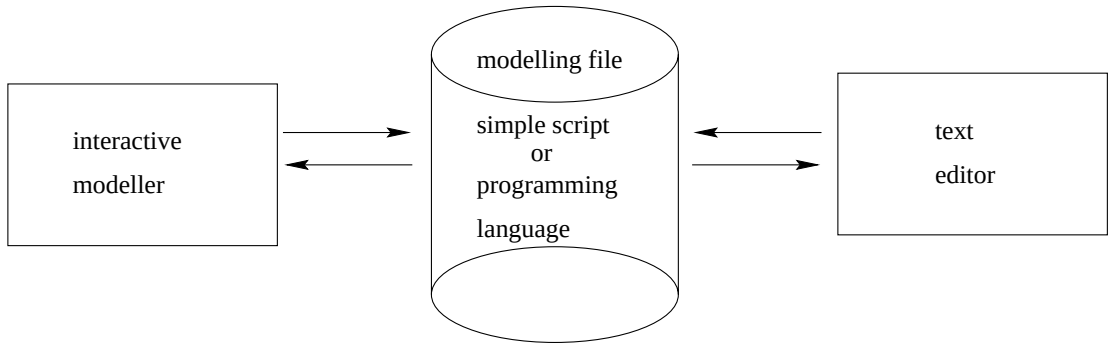


Figure 5.3: *A relationship between scripting and interactive systems*

Interactive editing

Implicit surfaces do not offer an easy way of rendering the surface of an object. Direct raytracing needs to be used (Appendix A) or the surface needs to be polygonised and then shaded using a standard polygon renderer. Neither of these methods is interactive. Wireframe-based, interactive implicit surface modellers exist commercially (*e.g.* MetaEditor [Meta94]) and as freeware (*e.g.* Blob Sculptor [Herm94]). They offer tools to create models built of simple primitives, defined by point-generators. They are useful for creating simple models of organic looking shapes. However, they become computationally expensive for more complex models.

Bloomenthal and Wyvill [Bloo90b] suggested a set of methods for improving the interactivity of modelling using implicit surfaces. Instead of a 3D surface, a series of 2D slices can be displayed. This requires good interpretation skills to perceive the shape of the 3D object. Another alternative is to directly display an octree used to locate the surface in space, *i.e.* in a given adaptive grid, cubes containing the surface are identified but surface polygons are not computed and the cubes displayed directly, with their back faces removed. Another interesting technique is *scattering*, in which a cloud of points, initially close to the surface, travels towards it until it achieves a required proximity. The surface is approximated by displaying the seeds.

The bottleneck of implicit surface visualisation is surface sampling. Witkin and Heckbert [Witk94] developed an adaptive sampling algorithm that used a set of *floaters* with repulsion forces that maintained their position on the surface and adapted their density to surface changes. For instance, if the surface was stretched, new surface floaters would be born and when the surface was squashed, some floaters would die. The floaters could be displayed at near real time rates, giving a good impression of the shape of the surface. A set of such sampling points could be used to obtain a polygonisation of the sampled surface. However, since a set of floaters did not have a fixed topology, *i.e.* during the life of each floater, it could travel anywhere on the surface, such a polygonisation would have to be recalculated after each modification.

Desbrun *et al.* [Desb95b] have developed a method of approximate display of implicit surfaces, based on piecewise polygonisation. The primitives were sampled and polygonised in their area of maximal influence. The topology of the resulting set of sampling points was fixed, since a set of seeds had always been sent in fixed directions from a generator to the surface. The sampling points had to be modified only in the areas where the surface had been modified. Thus, the algorithm could take advantage of temporal coherence between deformed versions of the same model. The surface could be visualised as sample points or the polygonisation of the blended implicit surface can be approximated by displaying the polygonised primitives. Such piecewise polygonal display was not continuous but it was interactive and offered an important advantage of opaque visualisation, that gave the information about depth that could not be achieved by displaying only the sampling points.

A short study of interactive display of implicit surfaces has been performed in [Opal95c]. The study investigated polygonisation of implicit surfaces using the marching cubes algorithm [Lore87]. The technique

of marching cubes divides a 3D space into grid cells. Then, there are three stages in the polygonisation algorithm: (i) generating grid-data, (ii) identifying grid cells intersecting the implicit surface and (iii) calculating the polygons. In the first stage (i), the scalar field values at the grid points (corners of grid cells) have to be calculated. During stage (ii) all cells which are intersected by the surface (boundary cells) are found and in the stage (iii) polygons in each boundary cell are produced according to the scalar field values in the corners of the cell. Previous attempts have been made to speed up steps (ii) and (iii) of polygonisation [Wyvi86a, Bloo88, Bloo90b]. The experiments in [Opal95c] were concerned with the first phase of the polygonisation process, the grid-data generation. During deformation, the grid-data will only change in the areas where the surface has been modified. Therefore, there is no need to regenerate all the data. A secondary data structure can be used to determine the grid parts that need processing. Appendix B presents the results of the study showing that the local updating of the grid-data gives a significant reduction in computation time for smaller primitives (half the size of the world and smaller) for both the static image and animation. For larger primitives (larger than half the size of the world) the complete regeneration of the grid-data appears to be more efficient.

This approach could be useful during the design of an implicit model since a user would possibly modify a small part of an object at a time. Similarly, during animation, only the moving objects need to be updated, the static rigid background stays unchanged and should only be rendered once. The method described in Appendix B was proposed for generation of grid-data but it could be used for any stage in the rendering process, *e.g.* if polygonisation is performed during the boundary cell search, the search does not have to be performed for the entire grid; only the modified grid cells need to be re-polygonised.

Model scripting

The approach taken in this thesis is the use of script files to specify models. A specification language was developed and writing a modelling script using the syntax of this language was left as a manual task to an animator. Although this is a quick method of prototyping the proposed algorithms, it proves unsatisfactory for modelling complex characters. Specifying components and their relative position is a tedious task and requires skill to create an interesting shape.

For instance, Figure 5.4 shows a sample scene and a script file used to specify it. Specifying such a scene manually is a very complex task. The primitives in this example were actually generated using supporting tools. The implicit "floor" was generated by a simple C program that used random radii for primitives and distributed the primitives at a fixed grid within a rectangle. The model of a head was constructed using a 2D interactive modelling tool [Opal93a]. The depth in it was achieved by assigning the Z-coordinate value to zero. The two characters were obtained by rotation and translation of the model obtained.

Using the experience from this example, one possible improvement to the modelling language is to convert it to a programming language that would allow for instance to distribute primitives along simple geometric primitives, *e.g.* a line segment, a curve or a polygon. The language could be purely text based, *i.e.* a program file would be compiled to produce a model, or an interesting possibility would be to consider visual programming, which could use an interactive tool to specify certain programming structures, *e.g.* an array of primitives.

5.3 Rendering implicit surfaces

Implicit surfaces suffer from a serious drawback - they cannot be easily rendered at interactive speeds. Two techniques that have been most widely used for rendering them are polygonisation and direct ray- tracing. Since they cannot be so far be performed in real-time, the usual trade-off between high quality and speed occurs. Hence, alternative techniques which display the approximation of the surface have been developed (see Section 5.2.3). However, perhaps a more serious problem affecting the rendering of models built using implicit surfaces is the problems of texturing them.

Texturing general models built with implicit surfaces is considered an open research problem. The possibility of constant changes of the surface of a model (including topology changes) is one of the main advantages of the technique. However, this causes problems with texturing such models, in particular with parametrisation of the deforming surface.

object *floor* **colour** *Flesh*
component *surface* **rotation** (0,0,0)
point (0,0,0) **POV** 158 1
point (0,0,80) **POV** 115 1
point (0,0,160) **POV** 154 1
point (0,0,240) **POV** 128 1
point (0,0,320) **POV** 114 1
point (0,0,400) **POV** 197 1
point (0,0,480) **POV** 117 1
point (0,0,560) **POV** 153 1
point (0,0,640) **POV** 200 1
point (0,0,720) **POV** 122 1
point (160,0,0) **POV** 156 1
point (160,0,80) **POV** 140 1
point (160,0,160) **POV** 129 1
point (160,0,240) **POV** 200 1
point (160,0,320) **POV** 134 1
point (160,0,400) **POV** 101 1
point (160,0,480) **POV** 102 1
point (160,0,560) **POV** 126 1
point (160,0,640) **POV** 114 1
point (160,0,720) **POV** 119 1
point (320,0,0) **POV** 198 1
point (320,0,80) **POV** 136 1
point (320,0,160) **POV** 118 1
point (320,0,240) **POV** 179 1
point (320,0,320) **POV** 178 1
point (320,0,400) **POV** 182 1
point (320,0,480) **POV** 113 1
point (320,0,560) **POV** 148 1
.
.
. 100 primitives
joint *surface*
anchor *surface* (0,0,0)

object *head1* **colour** *Brown*
component *head* **rotation** (-25,0,0)
point (265,0,743) **POV** 28 1
point (238,0,777) **POV** 22 1
point (220,0,778) **POV** 22 1
point (265,0,682) **POV** 11 -1
point (277,0,709) **POV** 22 1
point (263,0,718) **POV** 14 1
point (251,0,717) **POV** 11 1
point (242,0,716) **POV** 11 1
point (240,0,693) **POV** 15 1
point (251,0,696) **POV** 15 1
point (266,0,699) **POV** 15 1
point (304,0,751) **POV** 60 1
point (259,0,652) **POV** 24 1
point (311,0,694) **POV** 65 1
point (272,0,660) **POV** 21 1
point (287,0,672) **POV** 31 1
point (290,0,884) **POV** 43 1
point (410,0,620) **POV** 100 -1
point (214,0,828) **POV** 62 1
point (244,0,849) **POV** 62 1
point (432,0,748) **POV** 90 1
point (376,0,722) **POV** 90 1
point (272,0,794) **POV** 52 1
point (642,0,744) **POV** 5 1
point (642,0,749) **POV** 7 1
point (642,0,755) **POV** 9 1
point (642,0,761) **POV** 9 1
point (612,0,772) **POV** 9 1
.
.
. 66 primitives
joint *head*
anchor *head* (0,0,0)

object *head2* **colour** *Gold*
component *head* **rotation** (25,180,0)
point (265,0,743) **POV** 28 1
point (238,0,777) **POV** 22 1
point (220,0,778) **POV** 22 1
point (265,0,682) **POV** 11 -1
point (277,0,709) **POV** 22 1
point (263,0,718) **POV** 14 1
point (251,0,717) **POV** 11 1
point (242,0,716) **POV** 11 1
point (240,0,693) **POV** 15 1
point (251,0,696) **POV** 15 1
point (266,0,699) **POV** 15 1
point (304,0,751) **POV** 60 1
point (259,0,652) **POV** 24 1
point (311,0,694) **POV** 65 1
point (272,0,660) **POV** 21 1
point (287,0,672) **POV** 31 1
point (290,0,884) **POV** 43 1
point (410,0,620) **POV** 100 -1
point (214,0,828) **POV** 62 1
point (244,0,849) **POV** 62 1
point (432,0,748) **POV** 90 1
point (376,0,722) **POV** 90 1
point (272,0,794) **POV** 52 1
point (642,0,744) **POV** 5 1
point (642,0,749) **POV** 7 1
point (642,0,755) **POV** 9 1
point (642,0,761) **POV** 9 1
point (612,0,772) **POV** 9 1
.
.
. 66 primitives
joint *head*
anchor *head* (0,0,0)

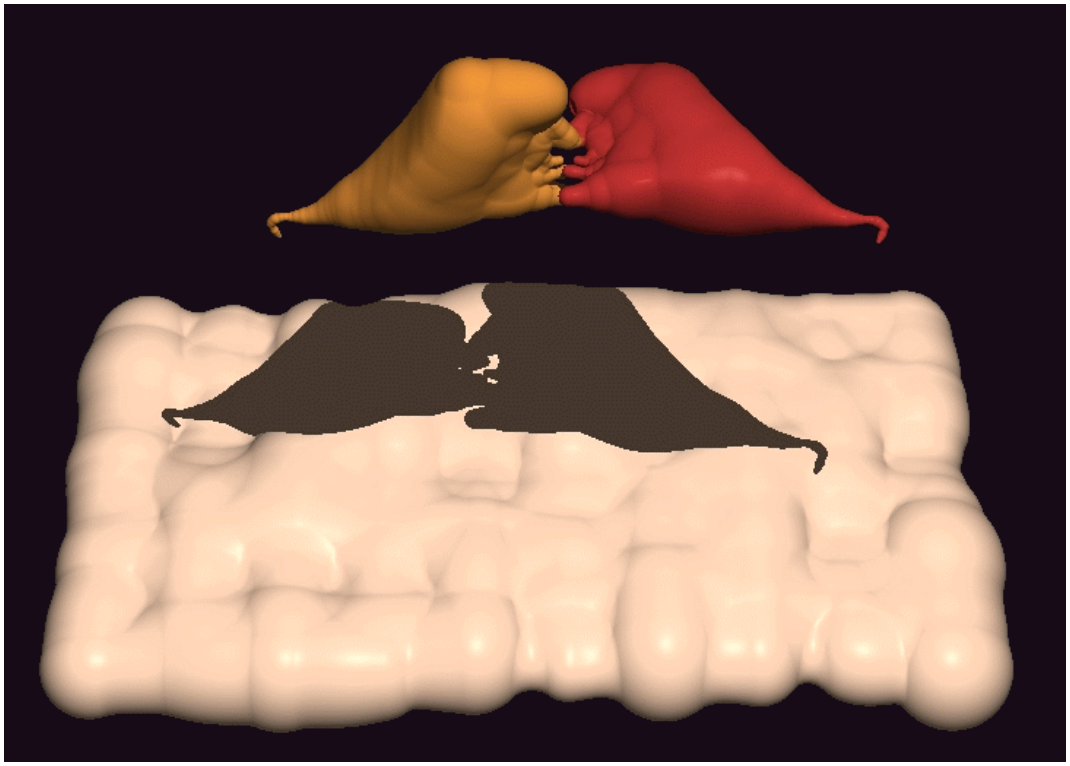


Figure 5.4: *Dali's Kiss No 7*

Colour is the simplest "texture" which can be applied to the skin of a character. It has been used in all implementations of implicit surfaces. The most common approach is to apply the weighted sum of colour contributions from all primitives to find the colour of the areas of multiple influence [Blin82, Nish85, Wyvi86a, McPh90]. This results in a smooth colour transition in the blending area. Each object component in a character can be assigned a colour to simulate items of "clothing". Flesh primitives can be assigned separate colour.

McPheeters [McPh90] summarises possible colouring metaphors for implicit surfaces. He suggests three simple approaches: all primitives change colour to the blended colour (primitives are like drops of coloured water), the division between primitives is visible (primitives are made of clay-like solids) or there is a smooth transfer of colour (semi-solid primitives which exhibit limited mixing). He also mentions a more complex metaphor: the effect of sculpting an object from differently coloured pieces of clay, the result being streaks of two initial colours and shades of the blended colour around the connection area.

One of the most popular texturing techniques is two-dimensional texture mapping. It finds a texture value for a surface point from the parametric representation of the surface. The values of parameters are passed to a 2D function which returns the required texture value. Implicit surfaces are not easily parametrised. Nevertheless parametrisation can be obtained in some cases. Bloomenthal [Bloo88] parametrises implicit surfaces using their boundary representation (polygons). McPheeters [McPh90] uses direct parametrisation of primitives applying complex mathematical procedures. An efficient patch based method of polygonising and decorating implicit surfaces was proposed by Pedersen [Pede95]. For character animation, it is important that the texture "sticks" to the surface during a character's life, including its deformation. Smets-Solanes [Smet93, Smet96] proposes a method that offers this property for implicit surfaces. He observes that parametrisation is not essential for applying a 2D texture mapping. What is sufficient is a homeomorphism between instances of a moving and deforming implicit surface. He proposes an algorithm for finding such a homeomorphism.

Solid texturing is another popular method of texturing, in which textured objects are "carved" in a 3D texture space. Problems arise when a textured object moves. Wyvill *et al.* [Wyvi90a] attempted to apply solid texturing to deformable implicit surfaces. However, squashing and stretching of a character could not be naturally handled by their solution.

5.4 Animating implicit surfaces

There are many similarities between dance choreography and character animation. Both of them start from an abstract idea to be delivered through the visual channel. The structure of a creation is sketched by dividing it into dance phrases or animation events. The artist is then developing each part, possibly simultaneously and often going back to the structure and modifying it according to any new ideas that appeared during the production. In this research, the choreographic nature of animation was recognised and used by implementing a dance notation as one level of a hybrid animation control method for animation creation. In this level, the animator has full control over the movement of all components in a character. The inspiration for the animation control for flesh, comes from the traditional character animation. The motion of flesh is used to create automatic animation effects, notably Squash and Stretch, Follow Through and Exaggeration. In this section the results obtained using this approach to animation control will be discussed and some new directions and extensions for it will be proposed.

5.4.1 Hybrid animation control

Characters modelled in thesis are built around articulated structures. Each link and joint in the structure contains a group of primitives (an object component). One method proposed for animation control of such a model is the coherence preserving algorithm (Section 3.4). Individual primitives are moved and the motion for the remaining primitives is computed accordingly. The motion specification is fairly low level since it has to be given by positioning the primitives in space. However, the system is capable of calculating the movement for all the primitives but one at each time step. Nevertheless, for more complex models specifying motion of individual primitives is a tedious procedure.

An alternative way of animating an articulated structure is to specify the positions and orientations for all joints and links using, for example, forward kinematics. It is a higher level of abstraction with relation to

primitives since there is no need to specify individual positions for any of the primitives in a model. They are derived from the motion of relevant links and joints. Moreover, the chosen implementation of forward kinematics, based on Eshkol-Wachmann dance notation, is based on an intuitive principle of dance position specification which makes choosing the link positions easier. However, a character animated in this way undergoes very little deformation, only the surface around joints changes slightly due to blending. With the goal of keeping the high level of abstraction for motion specification while keeping the fluidity of motion obtained using the coherence preserving algorithm, the model was extended by an automatically generated layer of flesh.

The motion control for components was specified using the dance notation, the motion for flesh was automatically derived from the motion of components. To provide the animator with control over the flesh motion, a set of adjustable parameters was used to define the flesh behaviour. Heuristics were proposed to adjust these parameters in order to achieve automatic and controllable traditional animation effects: Squash and Stretch, Follow Through and Exaggeration. Each of these abstract qualities of movement is adjusted using "knob" control (see Section 4.10). First, the flesh motion is created according to a set of low-level parameters given by an animator (*e.g.* the gravity or firmness of components or flesh). The motion created for this configuration of parameters is considered to be the "normal" version of a sequence. The animator can then increase or decrease the achieved effect on the high level by "turning" the relevant knob, *i.e.* by proportionally modifying all parameters manipulated by each abstract quality.

The heuristics work well for simple qualities, *e.g.* for Squash and Stretch which adjusts the firmness of object components. The results are predictable. A "looser" object will undergo more squash and stretch. A "firmer" one will deform less. For more complex qualities of Follow Through and Exaggeration, the results obtained in the animations are less intuitive. The parameters may easily interfere with each other if they are mapped as forward and inverse mapping in two different "knobs". The parameter interference was one of the reasons for introducing a layer of flesh, since increasing the amount of Squash and Stretch at the same time as increasing the amount of Follow Through cancelled the influence on the firmness parameter when used for object components.

One limitation of using the dance notation for specifying motion of components is that links are treated as rigid line segments, so spine-like effects cannot be modelled. For instance, in Figure 4.25, this effect can be clearly seen on the tail of the dinosaur. An interesting extension would be to combine the dance notation with the coherence preservation algorithm. A spine could be modelled as a chain of primitives connected by coherence preserving connections. The behaviour of such a "spine" could be either directly scripted by moving one of the primitives and making the remaining ones follow the motion, or it could be derived from a dance notation script for the components adjacent to the "spine" in the articulated structure. In this context, a useful extension to the coherence preserving algorithm is to use primitives other than spherical so that rigid links can be used in place of the rigid chains of primitives used at the moment. This extension is detailed in the following section.

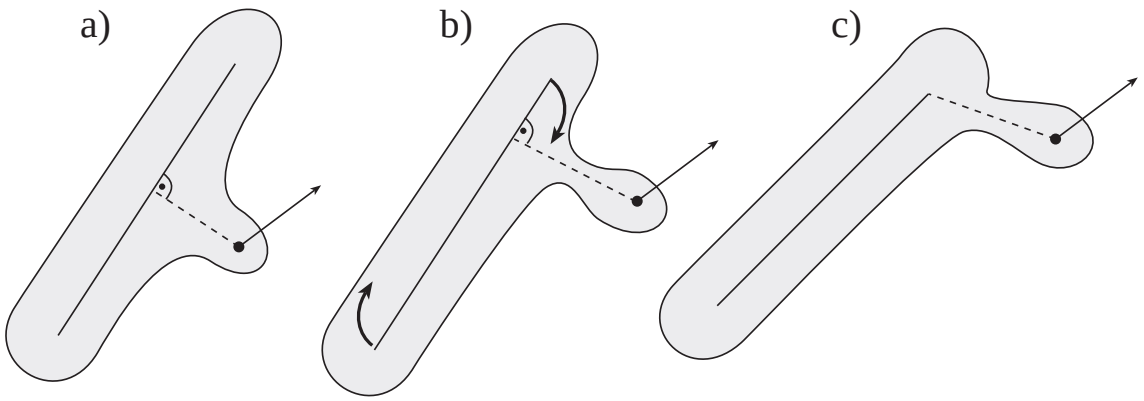


Figure 5.5: *A point-to-line connection*

Extending the coherence preserving algorithm

The coherence preserving algorithm that allows for clay-like material to be modelled, is extensible for other types of skeletons and other blending methods. To extend it to other types of generators, it is required to specify new types of connections between pairs of primitives. For example, to introduce a line segment as a generator, a point-to-line constraint has to be defined. It will restrict the distance from a point to a line segment. A rotation can be introduced to the "push/pull" action instead of translation only response. Figure 5.5 shows a possible behaviour of a point-to-line connection. Figure 5.5a show the initial configuration of two generators: a line segment and a point. The point is moved in the direction shown by the arrow. In the new configuration (Figure 5.5b) the distance between two two generators is still within the allowed limits, therefore the line does not move. However, when the point is moved again in the same direction (Figure 5.5c), the allowed coherence preserving distance is breached and the line generator follows the point. Instead of a simple translation, rotation should also be used to make the connection realistic. Similar cases would have to be developed for all possible pairs of generators. The work of Gascuel and Gascuel [Gasc94] proposes a method of maintaining displacement constraints between solids. They treat each connection as an elastic "string" and use this physical property to calculate the expected behaviour of connected solids. Small translations and rotations are applied to adjust each connection. An important advantage of this method is that it works for cycles in the connection network.

The coherence preservation algorithm is also extensible to other blending methods, provided that the new limiting conditions are found. For instance, if union blending is used, the maximum coherence preserving distance is the sum of the radii of the primitives in isolation:

$$d_{max} = f_1^{-1}(Iso) + f_2^{-1}(Iso)$$

This is an intuitive result since maximum-blending is equivalent to taking a union of the primitives. Thus, the two primitives are blended only if they overlap in space when considered in isolation.

5.4.2 Simplifying a model and metamorphosis

In an animation scene, objects sufficiently far away from the view point need not be shown with the same level of detail as the objects close to the viewer. Using a simplified version of a model to represent a distant object will reduce the time required to perform an animation step and render the scene. This strategy has been used in applications requiring a real time fly-through of an environment, *e.g.* in flight simulators.

In the realm of implicit surfaces this can be achieved by building a few models for each object, from a coarse one to be displayed at a distance to a fine one for close-ups. A multi-levelled design for implicit surfaces was proposed by Beier [Beie90] who suggested the use of a few primitives to approximate a character in the initial stages of character design and animation production and refining the representation after deciding on the desired sequence.

A similar approach can be used during an animation sequence, in which a less detailed representation of a character can be utilised for characters sufficiently far away from the view point. However, this introduces problems during switching between different levels of detail, since some primitives might suddenly appear or vanish. Therefore, a smooth transition between versions of a model has to be provided. It can be achieved by interpolation between the representations that will progressively vary the size of appearing or vanishing primitives. Since simplified versions of a character are built using the same appearance graph, such transition should produce natural results.

A more complex problem of graph simplification can also be considered. For objects far away, maintaining all the branches of the appearance graph and processing collisions between them is a superfluous costly operation that will not have a significant effect on the animated scene. Therefore, for distant objects, surface deformation due to collisions may be ignored. However, the collisions should be detected and influence the animation so it is up to date when the character comes closer to the viewpoint. This leads to a very complex problem of simplification of a model during motion. It is desired to save computation power by not calculating the motion of distance objects. However, when at a given time step an object is near the viewpoint, it should have completed its animation script until this time step. This may be understood as a very general problem of motion simplification, for example, a character moving at a distance along a spline may be moved along a straight line. Again, the question of possible collisions missed during such a simplification needs to be addressed.

A generalisation of the smooth transition between simplified models is 3D metamorphosis, *i.e.* finding a smooth transition between two arbitrary objects. When two two objects are specified as skeleton based implicit surfaces with graphs describing the blending properties of each model, a generalisation of the algorithm applied to perform smooth transitions between the simplified models of an object is required. One of the desired properties of the sought algorithm is consistent topology changes. If the topologies of the start and end objects are the same, no topology change should occur during the metamorphosis. If the topologies differ, the transition should not reverse to the topology of a previous stage in the metamorphosis (*e.g.* during the metamorphosis from a sphere to a torus, a hole should appear at one stage and it should not disappear until the end object is reached). Work on a metamorphosis algorithm would need to consider the shortest chain of changes required to perform the transition and whether or not all intermediate stages are valid models, *i.e.* include the skeletons, field functions and a graph of blending properties.

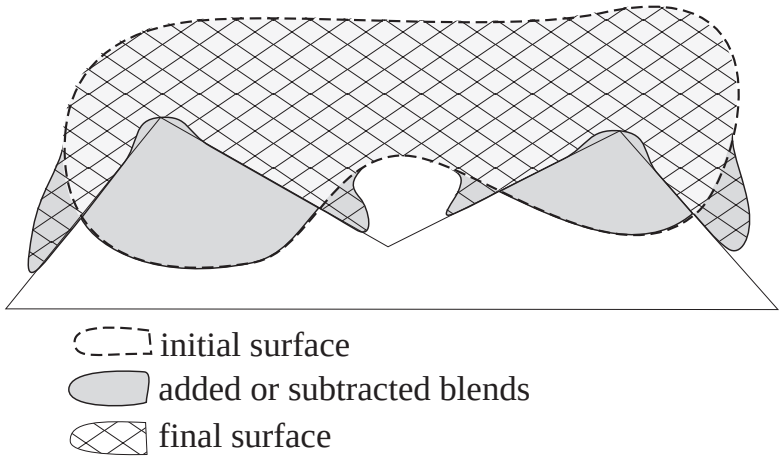


Figure 5.6: *Interaction of a smooth deformable object with a sharp landscape*

5.4.3 Combining implicit surfaces with other modelling techniques

Each modelling technique is better suited to a particular task. For instance, it would be difficult to model a cube with implicit surfaces, while it is a simple task using polygons or CSG operations. Similarly, modelling an expressive deformable character is easier using implicit surfaces and non-trivial using polygons. To provide a complete animation environment, as many modelling and animation techniques as possible should be supported. It is thus important to explore their interaction.

To combine implicit surfaces with polygonal models, the behaviour of an implicit character in an environment of sharp rigid objects defined using *e.g.* CSG methods should be examined. The character would need to deform itself against the sharp corners of a rigid object while keeping its tangent continuity. More complex precise collision surfaces would have to be created. Figure 5.6 shows a possible behaviour in such a situation. A mixture of additive and subtractive blends (dark grey areas) could possibly be used to achieve tangent continuous deformation around vertices and edges of a sharp landscape. The added surface blends which model the surface "dripping" down the sides of the rigid solid in Figure 5.6 could be modelled as suggested in Figure 5.2. The subtracted surface blends need to model the surface wrapped around sharp corners and edges on the rigid object. It is a more general case of precise contact surface creation during collisions. A rigid object can be represented by an implicit function, as for instance in the work by Pasko *et al.* [Pask88, Pask95b], in which a function representation of objects was proposed to represent a wide variety of primitives, including skeleton based implicit surfaces and CSG objects. In this case, a method of precise contact modelling between different classes of objects, based on scalar field modification, could be developed.

5.5 Summary

In this chapter the main points of the ABC approach have been summarised and some suggestions for extending it to more cases of skeletally based implicit surfaces have been given. Some initial ideas to provide tangent continuous deformation, an issue important for character animation as well as more general deformable clay-like substance deformation have been detailed.

The chapter also looked at ways of specifying an implicit character. A significant drawback of implicit surface for design and animation, is the lack of an interactive way of visualising them. The chapter reviewed the related research work in that area and summarised the results of a short study on the use of a secondary data structure to detect and process only the modified parts of a scene. The problem of interactive character sculpting and animation using implicit surfaces still remains an open research area.

A hybrid method of controlling the motion for a layered model based on providing a separate method for controlling the components (dance notation) and flesh motion (the coherence preserving algorithm combined with physical interaction), has been discussed. The main goal of this method has been to provide a user with a tool for creating automatic but fine-tunable traditional animation effects. Some extensions to the coherence preserving algorithm and to component animation specification have also been proposed.

A suggestion for the use of simplified models for character design and animation has been given. Some initial ideas about the advantages of using an appearance graph for creating a smooth transition between the versions of a model have been listed. It has also been suggested that a generalisation of this approach could be applied to 3D metamorphosis between moving characters.

Finally, the importance of integrating a variety of modelling techniques in an animation environment for character modelling has been stressed. As an example, the interaction between smooth implicit characters and sharp polygonal landscape has been discussed and some initial ideas for its implementation have been given.

Chapter 6

Conclusions

This thesis has presented the results of research into modelling and animation of characters using implicit surfaces. The initial motivation was the visual appeal of the technique: smoothly blended organic-looking shapes with arbitrary topology. The main goal was to adapt this method to the problems specific to character animation and to provide automatic creation of traditional animation effects. The findings of this research prove that extending implicit surfaces, as proposed in this thesis, creates a useful tool for modelling and animating simple but expressive characters. For modelling, the ABC approach introduces clay-like properties to the behaviour of a block of implicit material. Characters that can be sculpted in this material and they can be coherently deformed while retaining their appearance throughout their animated lives. For animation, the hybrid animation control method, developed for the multi-layered implicit model, offers a way of automatically creating and controlling cartoon effects such as squash and stretch, follow through and exaggeration.

In the ABC (Appearance, Blending, Coherence) approach, a means of specifying a character's appearance using a graph was proposed, blending properties were incorporated into the technique and coherence preservation constraints were imposed on the spacial relationship between the primitives in a model. A character is assembled from components connected according to its appearance graph and animated by moving its individual primitives. The novelty of this way of introducing a structure into a model lies in the fact that it does not restrict the number or the spacial relationship between the primitives that are assigned to the components. A certain amount of deformation, automatically controlled so as not to destroy the topological integrity, is allowed and therefore special cartoon effects (*e.g.* extreme stretching of limbs) can be modelled.

The unwanted blending algorithm detects and processes collisions between characters and self-collisions for a character. The response to collisions results in the creation of a precise contact surface between the interacting parts. Deformation during collisions is essential for animation, especially for implicit surfaces which, in their initial form, would blend the colliding parts, destroying the appearance of a model. The extended implicit formulation proposed in this thesis offers a simple algorithm that models deformation due to collisions by modifying the calculation of the global scalar field. The algorithm is general and extensible to more complex skeletons. The self-collision deformation processing is a novel development for implicit surfaces.

The coherence preserving algorithm ensures that all the primitives in a model stay connected during its deformation. The firmness for such a “clay” character can be specified. It changes the amount of squashing and stretching allowed to occur for it. The algorithm implements simple and efficient geometric distance constraints in a model. They control the distance between primitives and maintain it within the topological integrity preserving limits. As a result, the entire surface remains coherent without breaking into pieces. Distance constraints for implicit surfaces benefit from the fact that, within certain limits, surface topology remains unchanged during the relative motion of primitives. An extension incorporating more variety of generators to this algorithm has been proposed.

Animating characters by moving individual primitives is a tedious process and it is impractical for more complex models and longer animation sequences. Looking into the process of creating an animation, similarities to dance choreography can be observed. With this comparison in mind, a higher level of animation

control for characters modelled using an appearance graph was proposed. It implements the Eshkol-Wachmann dance notation and uses it to specify the positions of links in the appearance graph in relation to the joints. This is the principle of using forward kinematics for animating an articulated structure. However, it offers a slightly higher level of abstraction by expressing the link rotations in terms of shaping the body of a character into a dance position. Although a more intuitive way of motion specification, dance notation does not take advantage of the natural deformability of implicit surfaces and animations created using it exhibit little deformation.

To combine the advantages of using a higher level animation control method with the benefits of the coherence preserving algorithm, an extra layer was introduced into the model. A hybrid animation control method was then proposed to specify an animation sequence for this model. Dance notation was used to animate object components and define the general movement of a character. The motion of the flesh layer was generated based on a combination of the coherence preserving algorithm and physical interaction between primitives. This approach generates much of the animation for a character automatically. A set of parameters controls the generated motion and can be adjusted by a user. Heuristics are proposed that translate adding traditional animation effects into modifying a subset of these parameters. The method produces pleasing results for short animation sequences starring simple characters. Its intuitiveness needs to be assessed in a more complex animation project, that would create a longer animation sequence featuring a number of characters interacting with each other and with the environment.

A series of future directions for implicit surfaces was proposed. In particular, tangent continuity conserving and constant volume deformation needs to be further explored for character animation and for other applications of implicit surfaces, *e.g.* simulation of deformable substances. The limitations of the rendering process of implicit surfaces make them less attractive for interactive design and character animation. Thus, methods need to be developed to improve their display rate and provide consistent texturing of implicit objects during deformation. Model simplification and 3D feature based metamorphosis for moving characters are also interesting problems to explore.

Implicit surfaces offer a way of modelling smooth, organic-looking forms that can be locally modified to suit the animator's design of a character. The set of animation techniques proposed in this thesis enhances the artistic vocabulary of an animator by offering an intuitive way of working with implicit “clay” to model and animate three-dimensional characters using a computer. An improvement of the interactivity of the algorithms and a study integrating them into a standard, polygon-based animation environment is required to assess the usability of the methods for a real-life character animation project.

Bibliography

[Agin72] G. Agin. “Representation and Description of Curved Objects”. Stanford Artificial Intelligence Report MEMO AIM-173, Stanford University, 1972.

[Arvo91] J. Arvo, Ed. *Graphics Gems II*. Academic Press, 1991.

[Bare77] J. Barenholtz, Z. Wolofsky, I. Ganapathy, T. Calvert, and P. O'Hara. “Computer Interpretation of Dance Notation”. S. Lusignan and J. North, Eds., *Computing in the humanities*, pp. 235–240, University of Waterloo Press, Waterloo, 1977.

[Barr81] A. H. Barr. “Superquadrics and Angle-Preserving Transformations”. *IEEE Computer Graphics and Applications*, Vol. 1, No. 1, pp. 11–23, 1981.

[Beie90] T. Beier. “Practical Uses for Implicit Surfaces in Animation”. *Modeling and Animating with Implicit Surfaces*, pp. 20.1–20.11, 1990. SIGGRAPH Course Notes 23.

[Bene56] R. Benesh and J. Benesh. *An Introduction to Benesh Dance Notation*. Adam and Charles Black, London, 1956.

[Blan95] C. Blanc and C. Schlick. “Extended fields functions for soft objects”. *Implicit Surfaces'95*, pp. 21–32, Grenoble, France, Apr. 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Blin82] J. F. Blinn. “A Generalization of Algebraic Surface Drawing”. *ACM Transactions on Graphics*, Vol. 1, No. 3, pp. 235–256, July 1982.

[Bloo88] J. Bloomenthal. “Polygonisation of implicit surfaces”. *Computer Aided Geometric Design*, Vol. 5, pp. 341–355, 1988.

[Bloo90a] J. Bloomenthal. “Techniques for Implicit Modeling”. *Modeling and Animating with Implicit Surfaces*, pp. 13.1–13.18, 1990. SIGGRAPH Course Notes 23.

[Bloo90b] J. Bloomenthal and B. Wyvill. “Interactive Techniques for Implicit Modeling”. *Computer Graphics*, Vol. 24, No. 2, pp. 109–116, March 1990.

[Bloo91] J. Bloomenthal and K. Shoemake. “Convolution Surfaces”. *Computer Graphics*, Vol. 25, No. 4, pp. 251–256, July 1991. Proceedings of SIGGRAPH'91.

[Bloo92] J. Bloomenthal. “Hand Crafted”. *4th Annual Western Computer Graphics Symposium*, Banff, Alberta, Apr. 1992.

[Bloo95a] J. Bloomenthal. “Bulge Elimination in Implicit Surface Blends”. *Implicit Surfaces'95*, pp. 7–20, Grenoble, France, Apr. 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Bloo95b] J. Bloomenthal. *Skeletal Design of Natural Forms*. PhD thesis, The University of Calgary, Department of Computer Science, Jan. 1995.

[Bron85] I. Bronsztejn and K. Siemiendiajew. *Matematyka: poradnik encyklopedyczny (tom 1-6)*. Panstwowe Wydawnictwo Naukowe, 1985. In Polish.

[Calv80] T. Calvert, J. Chapman, and A. Patla. “The Integration of Subjective and Objective Data in the Animation of Human Movement”. *Computer Graphics*, pp. 198–203, 1980.

[Calv96] T. Calvert and S. Mah. “Choreographers as Animators: Systems to support composition of dance””. N. Magnenat-Thalmann and D. Thalmann, Eds., *Interactive Computer Animation*, Chap. 5, pp. 100–125, Prentice Hall, 1996.

[Chad89] J. E. Chadwick, D. R. Haumann, and R. E. Parent. “Layered Construction for Deformable Animated Characters”. *Computer Graphics*, Vol. 23, No. 3, pp. 243–252, 1989. Proceedings of SIGGRAPH'89.

[Desb94] M. Desbrun and M.-P. Gascuel. “Highly Deformable Material for Animation and Collision Processing”. *Fifth Eurographics Workshop on Animation and Simulation*, Oslo, Norway, Sep. 1994.

[Desb95a] M. Desbrun and M.-P. Gascuel. “Animating Soft Substances with Implicit Surfaces”. *Computer Graphics*, pp. 287–291, 1995. Proceedings SIGGRAPH'95.

[Desb95b] M. Desbrun, N. Tsingos, and M.-P. Gascuel. “Adaptive Sampling of Implicit Surfaces for Interactive Modelling and Animation”. *Implicit Surfaces'95*, pp. 171–186, Grenoble, France, Apr. 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Eshk58] N. Eshkol and A. Wachmann. *Movement Notation*. Weidenfeld and Nicolson, 1958.

[Espo95] J. M. de Espona. “The Basics of Building a Dinosaur”. *Computer Graphics World*, Vol. 18, No. 7, pp. 79–83, July 1995.

[Fors91] D. R. Forsey. “A Surface Model for Skeleton-based Character Animation”. *Second Eurographics Workshop on Animation and Simulation*, pp. 55–73, Vienna, Austria, Sep. 1991.

[Fuji90] T. Fujita, K. Hirota, and K. Murakami. “Representation of Splashing Water using Metaball Model”. *Fujitsu*, Vol. 41, No. 2, pp. 159–165, 1990. in Japanese.

[Gard65] M. Gardener. “M. Gardener's Mathematical Games Column: The Superellipse”. *Scientific American*, No. 213, pp. 222–234, 1965.

[Gasc91] M.-P. Gascuel, A. Verroust, and C. Puech. “Animation and Collisions between Complex Deformable Bodies”. *Graphics Interface'91*, pp. 263–270, 1991.

[Gasc93] M.-P. Gascuel. “An Implicit Formulation for Precise Contact Modelling between Flexible Solids”. *Computer Graphics*, Vol. 27, pp. 313–320, 1993. Proceedings of SIGGRAPH'93.

[Gasc94] J.-D. Gascuel and M.-P. Gascuel. “Displacement constraints for interactive modelling and animation of articulated structures”. *The Visual Computer*, Vol. 10, pp. 191–204, 1994.

[Gasc95] J.-D. Gascuel. “Implicit Patches: An optimised and powerful ray intersection algorithm”. *Implicit Surfaces'95*, pp. 143–160, Grenoble, France, Apr. 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Glas89] A. S. Glassner. *An Introduction to Ray Tracing*. Academic Press Ltd, 1989.

[Grav93] G. L. Graves. “The Magic of Metaballs”. *Computer Graphics World*, pp. 27–32, May 1993.

[Guy95] A. Guy and B. Wyvill. “Controlled Blending for Implicit Surfaces”. *Implicit Surfaces'95*, pp. 107–112, Grenoble, France, Apr. 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Hall95] N. Hall-Marriot. *DANCE: Describing, Animating and Notating Classical Enchaianements*. PhD thesis, University of Technology, Sydney, 1995.

[Hans88] A. Hanson. “Hyperquadrics: Smoothly Deformable Shapes with Convex Polyhedral Bounds”. *Computer Vision, Graphics, and Image Processing*, Vol. 44, pp. 191–210, 1988.

[Hart93] J. Hart. “Ray Tracing Implicit Surfaces”. *Modeling, Visualizing and Animating Implicit Surfaces*, pp. 13.1–13.15, 1993. SIGGRAPH Course Notes 25.

[Hart95] J. Hart. “Implicit Representation of Rough Surfaces”. *Implicit Surfaces'95*, pp. 33–44, Grenoble, France, Apr. 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Herm94] A. Hermida, S. Anger, and T. Brown. “Blob Sculptor version 1.0a”. 1994. Available at Pi Square BBS (301)725-9080.

[Hoff86] C. Hoffman and J. Hopcroft. “Quadratic blending surfaces”. *Computer Aided Design*, Vol. 18, pp. 301–306, 1986.

[Hutc70] A. Hutchinson. *Labanotation*. New York: Theatre Arts Books, 1970.

[Kaci91] Z. Kacic-Alesic and B. Wyvill. “Controlled Blending of Procedural Implicit Surfaces”. *Graphics Interface'91*, pp. 236–245, Calgary, Canada, June 1991.

[Karl89] D. Karla and A. H. Barr. “Guaranteed Ray Intersections with Implicit Surfaces”. *Computer Graphics*, Vol. 23, No. 3, pp. 297–306, 1989. Proceedings of SIGGRAPH'89.

[Lass87] J. Lasseter. “Principles of Traditional Animation Applied to Computer Animation”. *Computer Graphics*, Vol. 21, No. 4, pp. 35–43, 1987.

[Lore87] W. Lorensen and H. Cline. “Marching cubes: a high resolution 3D surface construction algorithm”. *Computer Graphics*, Vol. 21, No. 4, pp. 163–169, July 1987. Proceedings of SIGGRAPH'87.

[Magn88] N. Magnenat-Thalmann, R. Laperriere, and D. Thalmann. “Joint-Dependent Local Deformations for Hand Animation and Object Grasping”. *Graphics Interface'88*, pp. 26–33, 1988.

[Mand82] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman, 1982.

[Max91] N. L. Max and G. Wyvill. “Shapes and Textures for Rendering Coral”. N. M. Patrikalakis, Ed., *Scientific Visualisation of Physical Phenomena*, pp. 333–343, Springer-Verlag, 1991.

[McPh90] C. W. McPheeters. *Isosurface Modelling of Soft Objects in Computer Graphics*. PhD thesis, National Centre for Computer Animation, Department of Communication and Media, Dorset Institute, 1990.

[Meta94] “Meta Editor”. Personal communication from Bruno Tsuchiya, META Corporation Internatianal, 1994.

[Midd85] A. E. Middleditch and K. H. Sears. “Blend Surfaces for Set Theoretic Volume Modelling Systems”. *Computer Graphics*, Vol. 19, No. 3, pp. 161–170, July 1985.

[Mill89] G. Miller and A. Pearce. “Globular Dynamics: A Connected Particle System for Animating Viscous Fluids”. *Computers and Graphics*, Vol. 13, No. 3, pp. 305–309, 1989.

[Munc85] R. Muncaster. *A-level physics*. Stanley Thornes (Publishers) Ltd., 1985.

[Mura91] S. Muraki. “Volumetric Shape Description of Range Data using “Blobby Model””. *Computer Graphics*, Vol. 25, No. 4, pp. 227–235, July 1991. Proceedings of SIGGRAPH'91.

[Nish85] H. Nishimura, M. Hirai, T. Kawai, T. Kawata, I. Shirakawa, and K. Omura. “Object Modeling by Distribution Function and a Method of Image Generation”. *The Transactions of the Institute of Electronics and Communication Engineers of Japan*, Vol. J68-D, No. 4, pp. 718–725, 1985. In Japanese (translated into English by Takao Fujiwara while at Centre for Advanced Studies in Computer Aided Art and Design, Middlesex Polytechnic, England, 1989).

[Opal93a] A. Opalach and S. Maddock. “Computer Animation using Implicit Surfaces”. Tech. Rep. CS-93-15, Department of Computer Science, University of Sheffield, Nov. 1993. 52 pages.

[Opal93b] A. Opalach and S. Maddock. “Implicit Surfaces: Appearance, Blending and Consistency”. *Fourth Eurographics Workshop on Animation and Simulation*, pp. 233–245, Barcelona, Spain, Sep. 1993.

[Opal94] A. Opalach and S. Maddock. “Disney Effects using Implicit Surfaces”. *Fifth Eurographics Workshop on Animation and Simulation*, Oslo, Norway, Sep. 1994.

[Opal95a] A. Opalach. “How to create and animate a dancing dinosaur with implicit surfaces”. *Introduction to Modelling and Animation using Implicit Surfaces*, pp. 14.1–14.4, Leeds, UK, June 1995. Course Notes No 3.

[Opal95b] A. Opalach and S. Maddock. “High Level Control of Implicit Surfaces for Character Animation”. *Implicit Surfaces'95*, pp. 223–232, Grenoble, France, Apr. 1995. First International Eurographics Workshop on Implicit Surfaces.

[Opal95c] A. Opalach and S. Maddock. “Speeding Up Grid-Data Generation for Polygonisation of Implicit Surfaces”. *Eurographics UK Chapter*, pp. 153–160, Loughborough, UK, March 1995.

[Owen89] J. C. Owen and A. P. Rockwood. “Intersection of General Implicit Surfaces”. *ACM Transactions on Graphics*, pp. 335–345, Oct. 1989.

[Pask88] A. Pasko, V. Pilyugin, and V. Pokrovskij. “Geometric Modeling in the Analysis of Trivariate Functions”. *Computers and Graphics*, Vol. 12, No. 3/4, pp. 457–465, 1988.

[Pask93] A. Pasko, V. Savchenko, V. Adziev, and A. Sourin. “Multidimensional geometric modeling and visualization based on the function representation of objects”. Tech. Rep. 93-1-008, Department of Computer Software, The University of Aizu, Japan, Sep. 1993.

[Pask95a] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. “Function Representation in Geometric Modeling: Concepts, Implementation and Applications”. *The Visual Computer*, Vol. 11, No. 8, pp. 429–446, 1995.

[Pask95b] A. Pasko and V. Savchenko. “Constructing Functionally Defined Surfaces”. *Implicit Surfaces'95*, pp. 97–106, Grenoble, France, Apr. 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Payn92] B. A. Payne and A. W. Toga. “Distance Field Manipulation of Surface Models”. *IEEE Computer Graphics and Applications*, pp. 65–71, Jan. 1992.

[Pede95] H. K. Pedersen. “Decorating Implicit Surfaces”. *Computer Graphics*, pp. 291–300, 1995. Proceedings of SIGGRAPH'95.

[Pela94] C. Pelachaud, C. van Overweld, and C. Seah. “Modeling and Animating the Human Tongue during Speech Production”. *Computer Animation'94*, 1994.

[Perl95] K. Perlin. “Real Time Responsive Animation with Personality”. *IEEE Visualisation and Computer Graphics*, Vol. 1, No. 1, pp. 5–15, March 1995.

[POV 93] POV-team. “POV-Ray v.2.2”. 1993. Available by ftp from alfred.ccs.carleton.ca.

[Pres92] W. Press, S. Teukolski, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992.

[Reed94] T. Reed and B. Wyvill. “Visual Simulation of Lightning”. *Computer Graphics*, pp. 359–364, 1994. Proceedings of SIGGRAPH'94.

[Ricc73] A. Ricci. “A constructive geometry for computer graphics”. *The Computer Journal*, Vol. 16, No. 2, pp. 157–160, May 1973.

[Rock90a] A. P. Rockwood. “Using Implicit Surfaces to Blend Arbitrary Solid Models”. *Modeling and Animating with Implicit Surfaces*, pp. 9.1–9.22, 1990. SIGGRAPH Course Notes 23.

[Rock90b] A. P. Rockwood and J. C. Owen. “Blending Surfaces in Solid Modeling”. *Modeling and Animating with Implicit Surfaces*, pp. 10.1–10.17, 1990. SIGGRAPH Course Notes 23.

[Ross84] J. R. Rossignac and A. A. Requicha. “Constant-Radius Blending in Solid Modelling”. *Computers in Mechanical Engineering*, Vol. 3, No. 1, pp. 65–73, July 1984.

[Rvac87] V. L. Rvachev. *Theory of R-functions and Some Applications*. Naukova Dumka, Kiev, 1987. In Russian.

[Savc95] V. V. Savchenko, A. A. Pasko, O. G. Okunev, and T. L. Kunii. “Function Representation of Solids Reconstructed from Scattered Surface Points and Contours”. *Computer Graphics Forum*, Vol. 14, No. 4, pp. 181–188, 1995.

[Scla91] S. Sclaroff and A. Pentland. “Generalized Implicit Functions For Computer Graphics”. *Computer Graphics*, Vol. 25, No. 4, pp. 247–250, July 1991. Proceedings of SIGGRAPH'91.

[Sede86] T. W. Sedeberg and S. R. Parry. “Free Form Deformations of Solid Geometric Models”. *Computer Graphics*, Vol. 20, No. 4, pp. 151–160, 1986. Proceedings of SIGGRAPH'86.

[Shap94] V. Shapiro. “Real functions for representation of rigid solids”. *Computer Aided Geometric Design*, Vol. 11, pp. 153–175, 1994.

[Shen95] J. Shen and D. Thalmann. “Interactive Shape Design Using Metaballs and Splines”. *Implicit Surfaces'95*, pp. 187–196, Grenoble, France, Apr. 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Sing95] K. Singh and R. Parent. “Implicit Function Based Deformations of Polyhedral Objects”. *Implicit Surfaces'95*, pp. 113–128, Grenoble, France, Apr. 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Smet93] J.-P. Smets-Solanes. “Surfacic Textures for Animated Implicit Surfaces: the 2D case”. *Fourth Eurographics Workshop on Animation and Simulation*, pp. 221–232, Barcelona, Spain, 1993.

[Smet96] J.-P. Smets-Solanes. “Vector Field Based Texture Mapping of Animated Implicit Objects”. *Eurographics'96*, 1996. To appear.

[Smol77] S. Smoliar and L. Weber. “Using the Computer for a Semantic Representation of Labanotation”. S. Lusignan and J. North, Eds., *Computing in the humanities*, pp. 253–261, University of Waterloo Press, Waterloo, 1977.

[Tats90] H. Tatsumi, E. Takaoki, K. Omura, and H. Fujita. “A New Method for Three-Dimensional Reconstruction from Serial Sections by Computer Graphics Using “Meta-Ball”: Reconstruction of “Hepatoskeletal System” Formed by Ito Cells in the Cod Liver”. *Computers and Biomedical Research*, Vol. 23, No. part 1, pp. 37–45, 1990.

[Terz87] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. “Elastically Deformable Models”. *Computer Graphics*, Vol. 21, No. 4, pp. 205–214, July 1987. Proceedings of SIGGRAPH'87.

[Terz89] D. Terzopoulos, J. Platt, and K. Fleisher. “Heating and Melting Deformable Models (From Goop to Glop)”. *Graphics Interface'89*, pp. 219–226, June 1989.

[Thom81] F. Thomas and O. Johnston. *Disney Animation: The Illusion of Life*. Abbeville Press, 1981.

[Tonn91] D. Tonnesen. “Modelling liquids and solids using termal particles”. *Graphics Interface'91*, pp. 255–262, 1991.

[Tsin95] N. Tsingos, E. Bittar, and M.-P. Gascuel. “Implicit Surfaces for Semi-Automatic Medical Organs Reconstruction”. *Computer Graphics International'95*, pp. 3–15, Leeds, UK, June 1995.

[Turn95] R. Turner. “LEMAN: A System for Constructing and Animating Layered Elastic Characters”. *Computer Graphics International'95*, pp. 185–203, Leeds, UK, June 1995.

[Watt92] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, 1992.

[Watt93] A. Watt. *3D Computer Graphics (second edition)*. Addison-Wesley, 1993.

[Witk94] A. Witkin and P. Heckbert. “Using Particles to Sample and Control Implicit Surfaces”. *Computer Graphics*, pp. 269–278, July 1994. Proceedings of SIGGRAPH'94.

[Wood87] J. R. Woodwark. “Blends in Geometric Modelling”. R. R. Martin, Ed., *The mathematics of surfaces II*, pp. 255–297, Clarendon Press, 1987.

[Wyvi86a] B. Wyvill, C. McPheeters, and G. Wyvill. “Animating Soft Objects”. *The Visual Computer*, Vol. 2, pp. 235–242, Aug. 1986.

[Wyvi86b] G. Wyvill, C. McPheeters, and B. Wyvill. “Data Structure for Soft Objects”. *The Visual Computer*, Vol. 2, pp. 227–234, Aug. 1986.

[Wyvi88] B. Wyvill. “The Great Train Rubbery”. SIGGRAPH'86 Electronic Theatre and Video Review, 1988. Issue 26.

[Wyvi89] B. Wyvill and G. Wyvill. “Field Functions for Implicit Surfaces”. *The Visual Computer*, Vol. 5, pp. 75–82, Dec. 1989.

[Wyvi90a] G. Wyvill. “Texturing Implicit Surfaces”. *Modeling and Animating with Implicit Surfaces*, pp. 15.1–15.5, 1990. SIGGRAPH Course Notes 23.

[Wyvi90b] G. Wyvill and A. Trotman. “Ray-Tracing Soft Objects”. *Computer Graphics International'90*, pp. 469–475, 1990.

[Wyvi92] B. Wyvill. “Warping Implicit Surfaces for Animation Effects”. *Western Computer Graphics Symposium (SKIGRAPH'92)*, pp. 55–63, 1992.

Appendix A

Raytracing Implicit Surfaces

A.1 Introduction

The raytracing technique is based on simulating light rays or photon paths. Photons leave a light source and travel around a scene bouncing off objects. Some of them will reach the eye (the *view point*) and therefore contribute to the image of the scene. This is called *forward raytracing*. In computer graphics, *backward raytracing* is commonly used, where rays from the viewpoint are "fired" into the scene. Figure A.1 shows the general principle of backward raytracing. The *primary rays* (solid lines) are sent into a scene through each pixel in the screen. They are tested for intersections with the objects in the scene. If an intersection is found, it is tested for shadows by sending *shadow rays* (dotted lines) towards all light sources. The process is recursively repeated for rays reflected or transmitted through transparent objects (dashed lines). To calculate the shading at the intersection points, a vector normal to the surface in the intersection points needs to be computed. The colour of each pixel is calculated by combining the shading from all intersection points resulted from all levels of recursion after sending a ray to the scene. Glassner [Glas89] gives a very good introduction to raytracing.

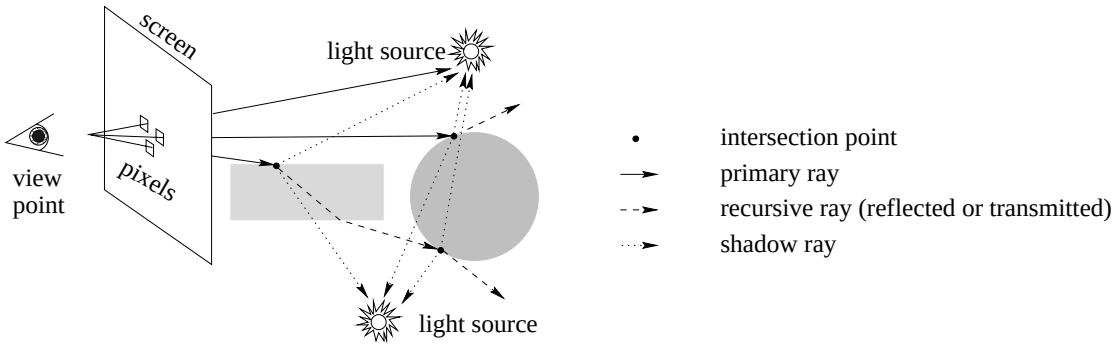


Figure A.1: *The principle of backward raytracing*

Implicit surfaces are well suited to raytracing because of the inside/outside function used to define them which has a simple normal vector calculation. Using inside/outside function, an intersection along a ray can be calculated with iterative methods, *e.g.* Newton's method or *regula falsi* as proposed by Blinn [Blin82]. A normal vector for an implicit surface is aligned with the gradient of the field, which is easily calculated using partial derivatives of the scalar field function. Blinn [Blin82] and Nishimura *et al.* [Nish85] visualise their implicit models using raytracing. More general raytracing algorithms for implicit surfaces have also been developed. Karla and Barr [Karl89] propose an algorithm which guarantees ray-surface intersections even with the finest details on an implicit surface. Wyvill and Trotman [Wyvi90b] suggest an algorithm which is less general but finds all intersections along a ray, not only the closest one, a feature useful for CSG operations on implicit surfaces. Dominique Gascuel [Gasc95] describes a based on "implicit patches"

extension to Karla and Barr's algorithm which handles skeletal elements and various blending properties. Hart [Hart93] presents a good review of raytracing applied to the implicit surfaces modelling technique.

This appendix gives details of an extension that implements implicit surfaces with blending properties proposed in this thesis in a freeware raytracer called POVray [POV 93]. Section A.2 will give some preliminary details, including calculation of the value of the scalar field function used, the ray-sphere intersection to determine which primitives influence the space along a ray, the data structures used and the preprocessing required. Section A.3 will describe the two-stage algorithm for finding ray-surface intersections: finding intervals along the ray in which the same scalar field function used (Section A.3.1) and calculating intersections in each interval (Section A.3.2). Section A.4 will show the computation of a normal vector at a given point.

A.2 Preliminaries

A.2.1 Field function

The potential function used is:

$$f(r) = s\left(1 - \frac{r^2}{R^2}\right)^2$$

where r is the distance from a given point to the generator of the primitive being considered, s is a scaling factor and R is the radius of influence of the considered primitive. This function can be transformed as follows:

$$f(r) = s\left(1 - 2\frac{r^2}{R^2} + \frac{r^4}{R^4}\right)$$

and further:

$$f(r) = \underbrace{\frac{s}{R^4}}_{c_0} r^4 + \underbrace{\frac{-2s}{R^2}}_{c_1} r^2 + \underbrace{s}_{c_2}$$

The function is stored as the coefficients c_0, c_1, c_2. Its value for a given r^2 is simply calculated as:

$$f(r) = c_0(r^2)^2 + c_1 r^2 + c_2$$

A.2.2 Radius in isolation

The radius of a primitive in isolation, used for instance in the coherence preserving algorithm (see Section 3.4) or for simple collision detection between spherical primitives, can be calculated as follows:

$$r_{Iso} = f^{-1}(Iso)$$

The radius in isolation therefore fulfills the following equation:

$$s\left(1 - \frac{r_{Iso}^2}{R^2}\right)^2 = Iso$$

Since $r_{Iso}\ 6\ R$, the square root of both sides of the equation can be taken:

$$1 - \frac{r_{Iso}^2}{R^2} = \sqrt{\frac{Iso}{s}}$$

Transforming further:

$$r_{Iso}^2 = R^2\left(1 - \sqrt{\frac{Iso}{s}}\right)$$

Because $r_{Iso}\ ¿\ 0$, the solution, after taking the square root, is:

$$r_{Iso} = R\sqrt{1 - \sqrt{\frac{Iso}{s}}}$$

A.2.3 Calculating field value

The following procedure implements field value calculation described in Section 3.3:

```
/*
 * Calculate the field value of a implicit surface - the position vector
 * "Pos" must already have been transformed into implicit space.
 */
static DBL
calculate_field_value(obj, Pos)
OBJECT *obj;
vector *Pos;
{
        int i, j, indef_flag, maxi;
        double len, density, defval, defv, defdensity, bldens,maxbldens;
        Primitive *ptr;
        IMPLICIT *implicit = (IMPLICIT *) obj;
        vector V;

        density = 0.0;
        for (i = 0; i < implicit->count; i++) {
                ptr = implicit->list[i];
                len = calculate_skeleton_distance2(&(implicit->list[0]), i, Pos, &V);
                if (len < ptr->radius2) {
                        density += len * (len * ptr->coeffs[0] + ptr->coeffs[1]) + ptr->coeffs[2];
                }
        }

        /* this is the undeformed density, calculate deformation */

        defdensity = 0.0;
        indef_flag = FALSE;
        for (i = 0; i < implicit->defcount; i++) {
                defval = 0.0;
                for (j = 0; j < implicit->deformlist[i]->count; j++) {
                        ptr = implicit->deformlist[i]->list[j];
                        len = calculate_skeleton_distance2(&(implicit->deformlist[i]->list[0]),
                              j, Pos, &V);
                        if (len < ptr->radius2) {
                                defv = len * (len * ptr->coeffs[0] + ptr->coeffs[1]) + ptr->coeffs[2];
                                if (defv >= implicit->isovalue) indef_flag = TRUE;
                                defval += defv;
                        }
                }
```

```
            if (defval > density) return 0;                    /* outside maximal influence */
            defdensity += defval;
        }

        /* max blending components density */

        maxbldens = 0.0;
        maxi = -1;
        for (i = 0; i < implicit→blcount; i++) {
            bldens = 0.0;
            for (j = 0; j < implicit→blendlist[i]→count; j++) {
                ptr = implicit→blendlist[i]→list[j];
                len = calculate_skeleton_distance2(&(implicit→blendlist[i]→list[0]),
                    j, Pos, &V);
                if (len ≤ ptr→radius2) {
                    bldens += len * (len * ptr→coeffs[0] + ptr→coeffs[1])+ptr→coeffs[2];
                }
            }
            if (bldens > density) return 0;
            if (bldens > maxbldens) {
                maxi = i;
                maxbldens = bldens;
            }
        }

        if (maxi>-1) density += maxbldens;


        if (density ≥ implicit→isovalue && indef_flag)
            return density-defdensity+implicit→isovalue;
        else
            return density;
}
```

A.2.4 Ray

A ray is represented by its starting point P_0 and its normalised direction $\overrightarrow{D}$ (Figure A.2). Any point P along the ray can be represented by:

$$P = P_0 + t\,\overrightarrow{D}$$

where $t \in \mathcal{R}$.

A.2.5 Ray-sphere intersection

To determine which primitives contribute to the scalar field function along a ray, the intersection of the ray with the bounding spheres, *i.e.* spheres at the radius of influence, are calculated. It is a simple ray-sphere intersection algorithm detailed in [Glas89].

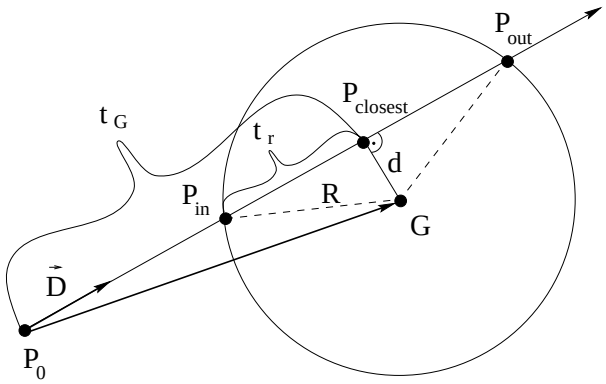


Figure A.2: *Ray-sphere intersection algorithm*

The steps of the algorithm are as follows (see Figure A.2):

1. Calculate the position vector from the ray starting point P_0 to the centre of the sphere G:

$$\overrightarrow{P_0G} = G - P_0$$

2. Calculate the distance from P_0 to the point on the ray closest to G. The point closest to G belongs to the ray, it can therefore be written as:

$$P_{closest} = P_0 + t_G \overrightarrow{D}$$

for a $t_G \in \mathcal{R}$. Thus, the distance from P_0 to $P_{closest}$ is:

$$\|P_{closest} - P_0\| = \|t_G \overrightarrow{D}\| = t_G \|\overrightarrow{D}\| = t_G$$

because $\overrightarrow{D}$ is normalised. The value of t_G can be calculated by projecting $\overrightarrow{P_0G}$ on the ray, using scalar product:

$$t_G = \overrightarrow{P_0G} \cdot \overrightarrow{D}$$

3. Calculate the distance d from G to $P_{closest}$. It can be calculated from the Pythagorean theorem using the right-angle triangle $\triangle P_0 P_{closest} G$:

$$d^2 = \|\overrightarrow{P_0G}\|^2 - t_G^2 = \overrightarrow{P_0G} \cdot \overrightarrow{P_0G} - t_G^2$$

4. Calculate the distance t_r from $P_{closest}$ to each of the two intersection points, P_{in} and P_{out}. It can be calculated from the Pythagorean theorem using one of the right-angle triangles $\triangle P_{in} P_{closest} G$ or $\triangle P_{out} P_{closest} G$:

$$t_r^2 = R^2 - d^2$$

where R is the radius of the sphere.

5. Calculate the two intersection points P_{in} and P_{out}. They are points lying on the ray, thus they can be expressed as:

$$\begin{aligned} P_{in} &= P_0 + t_{in} \overrightarrow{D} \\ P_{out} &= P_0 + t_{out} \overrightarrow{D} \end{aligned}$$

where $t_{in}, t_{out} \in \mathcal{R}$ can be calculated using previously computed values of t_G and t_r:

$$\begin{aligned} t_{in} &= t_G - t_r \\ t_{out} &= t_G + t_r \end{aligned}$$

Figure A.3 presents an implementation of this algorithm given by Cychosz in Graphics Gems II [Arvo91].

```
int intsph (raybase,raycos,center,radius,rin,rout)

Point3 raybase;                                          /* Base of the intersection ray */
Vector3 raycos;                                          /* Direction cosines of the ray */
Point3 center;                                           /* Center of the sphere */
double radius;                                           /* Radius of the sphere */
double *rin;                                             /* Entering distance */
double *rout;                                            /* Leaving distance */

{
        int hit;                                         /* True if ray intersects sphere*/
        double dx, dy, dz;                               /* Ray base to sphere center */
        double bsq, u, disc;
        double root;


        dx = raybase.x - center.x;
        dy = raybase.y - center.y;
        dz = raybase.z - center.z;
        bsq = dx*raycos.x + dy*raycos.y + dz*raycos.z;
        u = dx*dx + dy*dy + dz*dz - radius*radius;
        disc = bsq*bsq - u;

        hit = (disc >= 0.0);

        if (hit) {                                       /* If ray hits sphere */
                root = sqrt(disc);
                *rin = -bsq - root;                      /* entering distance */
                *rout = -bsq + root;                     /* leaving distance */
        }

        return (hit);
}
```

Figure A.3: *A routine calculating ray intersection with a sphere*

A.2.6 Data structures

Figure A.4 shows the data structures required. The type vector (Lines 1-3) is presented for completeness. It stores four coordinates of a vector in the homogeneous representation.

The next two types *implicitstackptr (Lines 4-9) and *componentstackptr (Lines 10-16) are used by the parser and return unsorted lists of primitives and components.

This data is then passed to the procedure called MakeImplicit (included in Section A.2.7) that converts them into Primitive records (Lines 18-23), Component dynamically allocated arrays (Lines 25-28) and a IMPLICIT record (Lines 35-47), which represents a component with blending properties.

The type Implicit_Interval (Lines 30-33) is used for storing intervals along a ray (Section A.3.1).

```
1: typedef struct {                                  /*————— A vector in homogeneous coordinates —————*/
2: double x,y,z,one;                                                          /* vector coordinates */
3: } vector;
4:                                                   /*—- An unsorted list of primitives returned by the parser —-*/
5: typedef struct implicit_list_struct *implicitstackptr;
6: struct implicit_list_struct {
7: Primitive prim;                                                        /* a description of a primitive */
8: implicitstackptr next;                                                 /* next primitive in the list */
9: };
10:                                                  /*—- An unsorted list of components returned by the parser —*/
11: typedef struct component_list_struct *componentstackptr;
12: struct component_list_struct {
13: implicitstackptr list;                                          /* a list of descriptions of primitives */
14: int count;                                       /* number of elements in the list (dynamic memory allocation) */
15: componentstackptr next;                                               /* next list of primitives */
16: };
17:
18: typedef struct {                                 /*————————— A primitive —————————-*/
19: double radius2, radius_iso;                                  /* radius of influence and radius in isolation */
20: double coeffs[3];                                          /* coefficients of the potential function */
21: double tcoeffs[5];                                /* coefficients required for intersection search */
22: vector generator;                                          /* the position of the (point-)generator */
23: } Primitive;
24:
25: typedef struct {                                 /*————— A component = a array of primitives —————*/
26: Primitive **list;                                          /* a dynamically allocated array of primitives */
27: int count;                                                          /* number of primitives */
28: } Component;
29:
30: typedef struct {                                 /*————— An interval for intersection search —————-*/
31: int type, index, index2;                         /* interval type and indices in the relevant arrays of primitives */
32: double bound;                                                  /* the value of t, along the ray */
33: } Implicit_Interval;
34:
35: struct Implicit_Struct                           /*————— A component with blending properties —————-*/
36: {
37: STURM_FIELDS                                                /* POVray variable - stores info for Sturm sequences */
38: TRANSFORM *Trans;                                           /* POVray variable - transformation matrix */
39: short Inverted;                                    /* POVray variable - boolean flag (inverted or not) */
40: double isovalue;                                                 /* the isovalue for the component */
41: int defcount, blcount;                             /* number of deforming and blending components */
42: Component *list;                                              /* an array of main primitives */
43: Component **deformlist;                                    /* an array of deforming components */
44: Component **blendlist;                                    /* an array of blending components */
45: Implicit_Interval *intervals;                  /* an array of intervals (used in intersection search) */
46: short collision;                                          /* a boolean flag (indicates a collision) */
47: } IMPLICIT;
```

Figure A.4: *Data structures for implicit surfaces*

A.2.7 Preprocessing

Creating components

For each component C its array of primitives list (line 42 in Figure A.4) is created. For each primitive its radius in isolation radius_iso (line 19 in Figure A.4) is calculated as described in Section A.2.2 and the coefficients c_0, c_1 and c_2 (line 20 in Figure A.4) are calculated as detailed in Section A.2.1.

Creating $\mathcal{B}(\mathcal{C})$ and $\mathcal{U}(\mathcal{C})$

For each component C, $\mathcal{B}(\mathcal{C})$ and $\mathcal{U}(\mathcal{C})$ are assigned based on the modelling script. They are stored as arrays of components blendlist and deformlist (lines 43-44 in Figure A.4) in the IMPLICIT record.

The following procedure implements the creation of components an their relevant sets $\mathcal{B}(\mathcal{C})$ and $\mathcal{U}(\mathcal{C})$ from the information passed from the parser:

```
void
MakeImplicit(implicit, isovalue, implicitlist, deforminglist,
blendinglist, blendtable, npoints, ndlists, nblists, sflag)
IMPLICIT *implicit;
double isovalue;
implicitstackptr implicitlist;
componentstackptr deforminglist;
```

```
componentstackptr blendinglist;
blendtableptr *blendtable;
int npoints, ndlists, nblists;
int sflag;
{
        int i, j;
        double rad, coeff, xs, ys, zs, xe, ye, ze, len;
        double coeffs[5], roots[4];
        int root_count;
        implicitstackptr temp;
        componentstackptr dtemp, btemp;
        vector mins, maxs, V;
        int ndpoints=0, nbpoints=0;                                    /* total number of deforming primitives */

        if (npoints < 1) {
                printf("Need at least one component in an implicit surface\n");
                exit(1);
        }
        implicit->isovalue = isovalue;
        implicit->list = (Primitive **) malloc(npoints * sizeof(Primitive *));
        implicit->deformlist = (Component **) malloc(ndlists * sizeof(Component *));
        implicit->blendlist = (Component **) malloc(nblists * sizeof(Component *));
        if (implicit->list == NULL || implicit->deformlist == NULL || implicit->blendlist == NULL)
                MAError("implicit data");

        for (i = 0; i < (unsigned) npoints; i++) {
                implicit->list[i] = (Primitive *) malloc(sizeof(Primitive));
                if (implicit->list[i] == NULL)
                        MAError("implicit data");
        }

        dtemp = deforminglist;
        for (i = 0; i < (unsigned) ndlists; i++) {
                implicit->deformlist[i] = (Component *) malloc(sizeof(Component));
                if (implicit->deformlist[i] == NULL)
                        MAError("implicit data");
                implicit->deformlist[i]->list = (Primitive **) malloc(dtemp->count * sizeof(Primitive *));
                if (implicit->deformlist[i]->list == NULL)
                        MAError("implicit data");
                implicit->deformlist[i]->count = dtemp->count;
                for (j = 0; j < (unsigned) dtemp->count; j++) {
                        ndpoints++;
                        implicit->deformlist[i]->list[j] = (Primitive *) malloc(sizeof(Primitive));
                        if (implicit->deformlist[i]->list[j] == NULL)
                                MAError("implicit data");
                }
                dtemp = dtemp->next;
        }

        btemp = blendinglist;
        for (i = 0; i < (unsigned) nblists; i++) {
                implicit->blendlist[i] = (Component *) malloc(sizeof(Component));
                if (implicit->blendlist[i] == NULL)
                        MAError("implicit data");
                implicit->blendlist[i]->list = (Primitive **) malloc(btemp->count * sizeof(Primitive *));
                if (implicit->blendlist[i]->list == NULL)
                        MAError("implicit data");
                implicit->blendlist[i]->count = btemp->count;
                for (j = 0; j < (unsigned) btemp->count; j++) {
                        nbpoints++;
                        implicit->blendlist[i]->list[j] = (Primitive *) malloc(sizeof(Primitive));
                        if (implicit->blendlist[i]->list[j] == NULL)
                                MAError("implicit data");
                }
                btemp = btemp->next;
        }

        implicit->count = npoints;
        implicit->defcount = ndlists;
        implicit->blcount = nblists;
        implicit->Sturm_Flag = sflag;

        /* Initialize the implicit data */
        for (i = 0; i < npoints; i++) {
                temp = implicitlist;
                if (fabs(temp->elem.coeffs[2]) < EPSILON ||
                    temp->elem.radius2 < EPSILON) {
                        perror("Degenerate implicit element\n");
                }
                /* Store implicit specific information */
                rad = temp->elem.radius2;
```

```
            implicit→list[i]→radius_iso = rad*(sqrt(1 - sqrt(Iso/coeff)));
            rad *= rad;
            coeff = temp→elem.coeffs[2];
            implicit→list[i]→radius2 = rad;
            implicit→list[i]→coeffs[2] = coeff;
            implicit→list[i]→coeffs[1] = -(2.0 * coeff) / rad;
            implicit→list[i]→coeffs[0] = coeff / (rad * rad);

            implicit→list[i]→generator.x = temp→elem.generator.x;
            implicit→list[i]→generator.y = temp→elem.generator.y;
            implicit→list[i]→generator.z = temp→elem.generator.z;

            implicitlist = implicitlist→next;
            free(temp);
    }

    /* Initialize the deforming data */
    for (i = 0; i < ndlists; i++) {
            dtemp = deforminglist;
            for (j=0; j<dtemp→count; j++) {
                    temp = deforminglist→list;
                    if (fabs(temp→elem.coeffs[2]) < EPSILON ||
                        temp→elem.radius2 < EPSILON) {
                            perror("Degenerate deforming element\n");
                    }
                    /* Store implicit specific information */
                    rad = temp→elem.radius2;
                    implicit→deformlist[i]→list[j]→radius_iso = rad*(sqrt(1 - sqrt(Iso/coeff)));
                    rad *= rad;
                    coeff = temp→elem.coeffs[2];
                    implicit→deformlist[i]→list[j]→radius2 = rad;
                    implicit→deformlist[i]→list[j]→coeffs[2] = coeff;
                    implicit→deformlist[i]→list[j]→coeffs[1] = -(2.0 * coeff) / rad;
                    implicit→deformlist[i]→list[j]→coeffs[0] = coeff / (rad * rad);

                    implicit→deformlist[i]→list[j]→generator.x = temp→elem.generator.x;
                    implicit→deformlist[i]→list[j]→generator.y = temp→elem.generator.y;
                    implicit→deformlist[i]→list[j]→generator.z = temp→elem.generator.z;

                    dtemp→list = dtemp→list→next;
                    free(temp);
            }
            deforminglist = deforminglist→next;
            free(dtemp);
    }

    /* Initialize the blending data */
    for (i = 0; i < nblists; i++) {
            btemp = blendinglist;
            for (j=0; j<btemp→count; j++) {
                    temp = blendinglist→list;
                    if (fabs(temp→elem.coeffs[2]) < EPSILON ||
                        temp→elem.radius2 < EPSILON) {
                            perror("Degenerate blending element\n");
                    }
                    /* Store implicit specific information */
                    rad = temp→elem.radius2;
                    implicit→blendlist[i]→list[j]→radius_iso = rad*(sqrt(1 - sqrt(Iso/coeff)));
                    rad *= rad;
                    coeff = temp→elem.coeffs[2];
                    implicit→blendlist[i]→list[j]→radius2 = rad;
                    implicit→blendlist[i]→list[j]→coeffs[2] = coeff;
                    implicit→blendlist[i]→list[j]→coeffs[1] = -(2.0 * coeff) / rad;
                    implicit→blendlist[i]→list[j]→coeffs[0] = coeff / (rad * rad);

                    implicit→blendlist[i]→list[j]→generator.x = temp→elem.generator.x;
                    implicit→blendlist[i]→list[j]→generator.y = temp→elem.generator.y;
                    implicit→blendlist[i]→list[j]→generator.z = temp→elem.generator.z;

                    btemp→list = btemp→list→next;
                    free(temp);
            }
            blendinglist = blendinglist→next;
            free(btemp);
    }

    /* Allocate memory for intersection intervals */
    implicit→intervals = (Implicit_Interval *)
                            malloc((npoints+nbpoints+ndpoints)*
                                                                    NO_OF_INTERSECTIONS * sizeof(Implicit_Interval));
    if (implicit→intervals == NULL)
```

```
        MAError("implicit data");
}
```

Collision detection

Collision detection can be performed without surface sampling, taking advantage of the fact that primitives are built around point-generators. Therefore, a collision occurs when any two generators G_1 and G_2 from two unblendable components are closer to each other than the sum of their respective radii in isolation (as detailed in Section 3.4), *i.e.* when:

$$d(G_1, G_2) \ 6 \ r_{1\ Iso} + r_{2\ Iso}$$

To detect collisions, the program iterates over all primitives from all unblendable components and checks the distances between generators. If a collision occurs, it is recorded in the component (line 46 in Figure A.4).

A.3 Ray-implicit surface intersections

The intersections are found in two stages: (i) determine which primitives influence the space along the given ray and create the intervals based on this information and (ii) calculate intersections in each interval.

A.3.1 Determining influences along a ray

For each component C a list of its primitives and the primitives in its blendable and unblendable sets are traversed, looking for intersections with the radii of influence of primitives using the routine intsph (Figure A.3). The values of t_{in} and t_{out}, calculated by intsph, mark the limits of influence of a primitive. A list oft_{in} and t_{out}, sorted along the ray, is created and stored in *intervals (line 45 in Figure A.4). The following fragment of code presents calculation of intervals:

```
/*
 * Store the points of intersection of this implicit with the ray. Keep
 * track of: whether this is the start or end point of the hit, which
 * component was pierced by the ray, and the point along the ray that the hit
 * occured at.
 */
static
add_hit_interval(intervals, cnt, type, index, index2, bound)
Implicit_Interval **intervals;
int *cnt;
int type, index, index2;
DBL bound;
{
        int k, j;
        for (k = 0; k < *cnt && bound > (*intervals)[k].bound; k++);
        if (k < *cnt) {

        /*
         * This hit point is smaller than one that already exists -
         * bump the rest and insert it here
         */
                for (j = *cnt; j > k; j--)
                        memcpy(&((*intervals)[j]), &((*intervals)[j - 1]), sizeof(Implicit_Interval));
                (*intervals)[k].type = type;
                (*intervals)[k].index = index;
                (*intervals)[k].index2 = index2;
                (*intervals)[k].bound = bound;
                (*cnt)++;
        } else {
        /*
         * Just plop the start and end points at the end of the list
         */
                (*intervals)[(*cnt)].type = type;
                (*intervals)[(*cnt)].index = index;
                (*intervals)[(*cnt)].index2 = index2;
                (*intervals)[(*cnt)].bound = bound;
        }
}

static int
valid(t0,t1,mindist)
double *t0,*t1,mindist;
```

```
{
        double disc;

        if (t1 < mindist) t1 = 0.0;
        if (t0 < mindist) t0 = 0.0;
        if (t1 == t0) return FALSE;
        else if (t1 < t0) {
                disc = t0;
                t0 = t1;
                t1 = disc;
        }
        return TRUE;
}

/* types of intervals */
#define IN 0
#define OUT 1
#define IN_DEF 3
#define OUT_DEF 4
#define IN_BL 5
#define OUT_BL 6

static int
determine_influences(P, D, implicit, mindist)
VECTOR *P, *D;
IMPLICIT *implicit;
DBL mindist;
{
        int i, j, cnt;
        double t0, t1;
        Implicit_Interval *intervals = implicit->intervals;
        Primitive *temp;

        cnt = 0;
        for (i = 0; i < implicit->count; i++) {

                if (intsph(*P, *D, implicit->list[i]->generator, sqrt(implicit->list[i]->radius2), &t0, &t1)) {
                        if (!valid(&t0,&t1,mindist))
                                break;
                        add_hit_interval(&intervals, &cnt, IN, i, 0, t0);
                        add_hit_interval(&intervals, &cnt, OUT, i, 0, t1);
                }                                                                   /* if intsph */
        }                                                                           /* for */

        /* now find intersections with deforming primitives */

        for (i = 0; i < implicit->defcount; i++) {
                for (j = 0; j < implicit->deformlist[i]->count; j++) {

                        temp = (implicit->deformlist[i]->list)[j];
                        if (intsph(*P, *D, temp->generator, sqrt(temp->radius2), &t0, &t1)) {
                                if (!valid(&t0,&t1,mindist))
                                        break;
                                add_hit_interval(&intervals, &cnt, IN_DEF, i, j, t0);
                                add_hit_interval(&intervals, &cnt, OUT_DEF, i, j, t1);
                        }
                }                                                                   /* for j */
        }                                                                           /* for i */

        /* and intersections with blending primitives */

        for (i = 0; i < implicit->blcount; i++) {
                for (j = 0; j < implicit->blendlist[i]->count; j++) {

                        temp = (implicit->blendlist[i]->list)[j];
                        if (intsph(*P, *D, temp->generator, sqrt(temp->radius2), &t0, &t1)) {
                                if (!valid(&t0,&t1,mindist))
                                        break;
                                add_hit_interval(&intervals, &cnt, IN_BL, i, j, t0);
                                add_hit_interval(&intervals, &cnt, OUT_BL, i, j, t1);
                        }
                }                                                                   /* for j */
        }                                                                           /* for i */

        return cnt;
}
```

Figure A.5 presents a diagram of a scene discussed in Section 3.3.4. It is composed of one object, containing three components: a joint C_1 and two links C_2 and C_3. Each of the three components contains one primitive, whose radius of influence is marked with a dashed circle.

The example shows calculating the intersections for component C_2. The set of its unblendable components $\mathcal{U}(\mathcal{C}_\in) = \{\mathcal{C}_\ni\}$ and the set of its blendable components $\mathcal{B}(\mathcal{C}_\in) = \{\mathcal{C}_\infty\}$. The intersections of the ray with all spheres of influence are marked with letters A, B, C, D, E, F and the type of each intersection is indicated on the left of the figure.

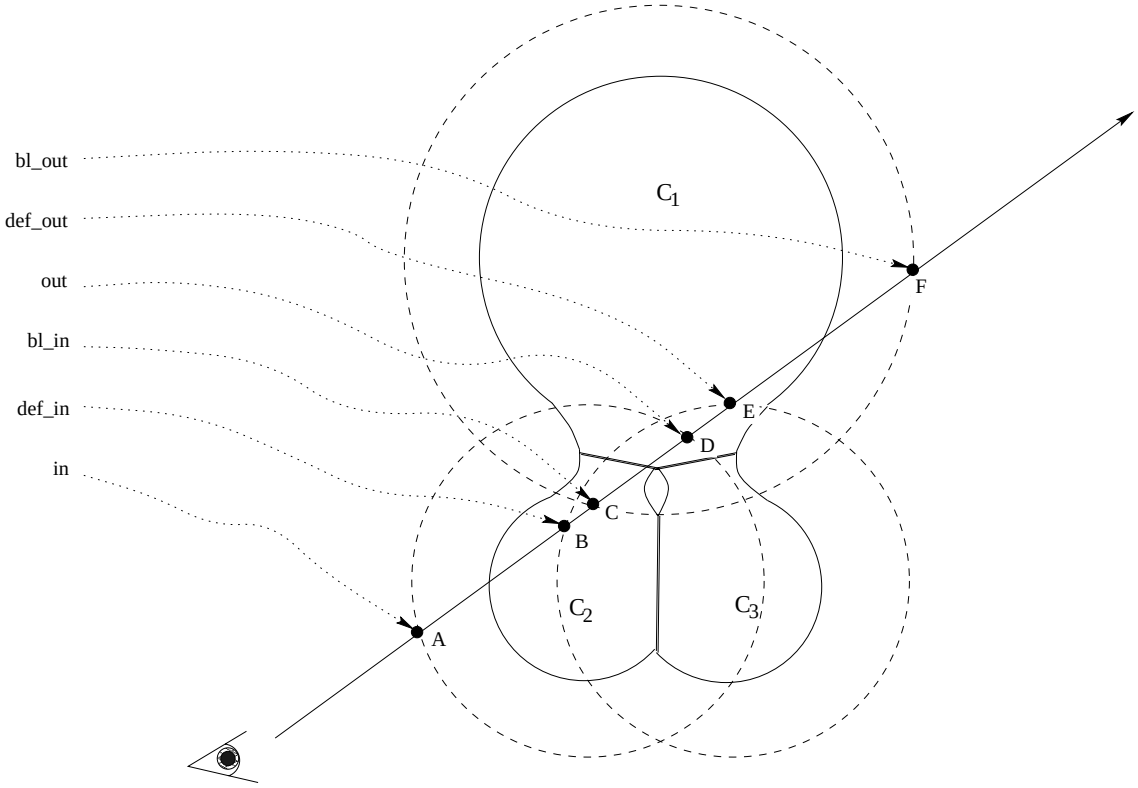


Figure A.5: *Calculating ray-surface intersections for component* C_2

A.3.2 Finding intersections in each interval

When the intervals are calculated, each of them is processed looking for intersection with the isosurface of the component being considered.

A point of the ray P is represented by:

$$P = P_0 + t\overrightarrow{D}$$

for $t \in \mathcal{R}$. The distance r^2 from such a point P on the ray to the generator G of a primitive can be found from the Pythagorean theorem:

$$r^2 = (P_0 + t\overrightarrow{D} - G) \cdot (P_0 + t\overrightarrow{D} - G) = \overrightarrow{D} \cdot \overrightarrow{D}t^2 + 2t\overrightarrow{D} \cdot (P_0 - G) + (P_0 - G) \cdot (P_0 - G)$$

Using the following substitutions:

$$t_0 = (P_0 - G) \cdot (P_0 - G)$$
$$t_1 = \overrightarrow{D} \cdot (P_0 - G)$$

and noticing $\overrightarrow{D} \cdot \overrightarrow{D} = 1$ it can be rewritten as:

$$r^2 = t_0 + 2tt_1 + t^2$$

This distance can then be inserted into the coefficient form of a primitive's field function:

$$f(r) = c_0(r^2)^2 + c_1 r^2 + c_2$$

resulting in the following equation:

$$f(r) = c_0(t_0 + 2tt_1 + t^2) + c_1(t_0 + 2tt_1 + t^2) + c_2$$

Expanding the terms and grouping with respect to t gives:

$$f(r) = t^4 \underbrace{c_0}_{T_0} + t^3 \underbrace{(4c_0t_1)}_{T_1} + t^2 \underbrace{(c_1 + 2c_0t_0 + 4c_0t_1^2)}_{T_2} + t \underbrace{2(c_1t_1 + 2c_0t_0t_1)}_{T_3} + \underbrace{c_2 + c_1t_0 + c_0t_0^2}_{T_4}$$

This formulae can be solved for t by any quartic root solvers [Pres92] to find the values of t which fulfill $f(r) = Iso$. For multiple primitives, the coefficients T_0, T_1, T_2, T_3, T_4 are added:

$$f_1(r) + f_2(r) = t^4(T_{10} + T_{20}) + t^3(T_{11} + T_{21}) + t^2(T_{12} + T_{22}) + t(T_{13} + T_{23}) + (T_{14} + T_{24})$$

By solving this formulae, intersection with the implicit surface due to the blend between the two primitives are found.

To find intersections, intervals are processed one by one. Each time a new primitive starts to influence the ray, its coefficients T_0, T_1, T_2, T_3, T_4 are calculated and stored in the array tcoeffs (Line 21 in Figure A.4) in the relavant Primitive record.

Three lists of currently influencing components are constructed as arrays of sums of coefficients tcoeffs from all primitives belonging to the component: the sum of coefficients from the component being considered (called coeffs in the procedure All_Implicit_Intersections quoted further in this section), an array of coefficients from the components blendable with it (called bcoeffs[] in All_Implicit_Intersections) and the sum of coefficients from the components unblendable with it (called dcoeffs in All_Implicit_Intersections). The blendable components are treated separately because in the algorithm for implicit surface with blending properties (Section 3.3) the blending term only considers components that are within their maximal areas of influence.

When a primitive starts to influence the ray (interval types IN, IN_DEF, IN_BL), its tcoeffs coefficients are added to the relevant array of coefficients. When a primitive stops to influence the ray (interval types OUT, OUT_DEF, OUT_BL) its coefficients are subtracted from the relevant coefficient array.

After updating the coefficients, the function that describes the scalar field in the current interval along the ray is computed using the coefficients coeffs, bcoeffs and dcoeffs. Let us look again at the example in Figure A.5.

The influence of components and the field functions in the intervals are as follows:

interval	main	blendable	unblendable	
AB	C_2	-	-	
BC	C_2	-	C_3	sc
CD	C_2	C_1	C_3	solve
DE	-	C_1	C_3	
EF	-	C_1	-	

To validate the roots, the values of t are checked against the bounds of the current interval. If they are outside the bounds, the t in invalid. Then, the potential intersection point $P_{intersect}$ is calculated from:

$$P_{intersect} = P_0 + t\overrightarrow{D}$$

To verify is $P_{intersect}$ lies on the implicit surface, the field value is calculated and tested against the isovalue. If $P_{intersect}$ lies on the implicit surface, *i.e.* its field value is equal to the isovalue within certain precision, it is passed to POVray as an intersection point.

The following procedure implements root finding in intervals:

```
int
All_Implicit_Intersections(Object, Ray, Depth_Stack)
OBJECT *Object;
RAY *Ray;
ISTACK *Depth_Stack;
{
        IMPLICIT *implicit = (IMPLICIT *) Object;
        INTERSECTION Local_Element;
        DBL dist, len, *tcoeffs, dcoeffs[5], **bcoeffs, coeffs[5], roots[4];
        int i, j, k, b, cnt, dcnt, bcnt, binx;
        vector P, D, V;
        int root_count, in_flag, indef_flag, inbl_flag, *inbl_flag_array;
        Primitive *element;
        DBL t0, t1, c0, c1, c2;
        vector IPoint, dv;
        Implicit_Interval *intervals = implicit->intervals,
        int Intersection_Found = FALSE, New_Intersection_Found = FALSE;

        Ray_Implicit_Tests++;

        /* Trans the ray into the implicit surface space */

        if (implicit->Trans != NULL) {
                MInvTransPoint(&P, &Ray->Initial, implicit->Trans);
                MInvTransDirection(&D, &Ray->Direction, implicit->Trans);
        } else {
                P.x = Ray->Initial.x;
                P.y = Ray->Initial.y;
                P.z = Ray->Initial.z;
                D.x = Ray->Direction.x;
                D.y = Ray->Direction.y;
                D.z = Ray->Direction.z;
        }

        len = sqrt(D.x * D.x + D.y * D.y + D.z * D.z);
        if (len == 0.0)
                return 0;
        else {
                D.x /= len;
                D.y /= len;
                D.z /= len;
        }

        /*
    * Figure out the intervals along the ray where each component of the
    * implicit surface has an effect.
    */
        if ((cnt = determine_influences(&P, &D, implicit, 0.01)) == 0)
                /*
           * Ray doesn't hit the sphere of influence of any of its
           * component elements
           */
                return 0;

        /* Clear out the coefficients */
        for (i = 0; i < 4; i++)
                coeffs[i] = 0.0;
        coeffs[4] = -implicit->threshold;

        for (i=0; i<5; i++)
                dcoeffs[i] = 0.0;

        bcoeffs = (DBL**)malloc(implicit->blcount*sizeof(DBL*));
        for (i=0; i<implicit->blcount; i++)
                bcoeffs[i] = (DBL *)malloc(5*sizeof(DBL));

        inbl_flag_array = (int*)malloc(implicit->blcount*sizeof(int));

        for (i=0; i<implicit->blcount; i++) {
                inbl_flag_array[i]=0;
                for (j=0; j<5; j++)
                        bcoeffs[i][j] = 0.0;
        }

        /*
    * Step through the list of influence points, adding the influence of
    * each implicit component as it appears
    */
        in_flag = 0;
        indef_flag = 0;
        inbl_flag = 0;
```

```
for (i=0; i < cnt; i++) {

        /* examine each interval, update coeffs, dcoeffs and bcoeffs */

        switch (intervals[i].type) {

        case IN:

                in_flag++;
                element = implicit→list[intervals[i].index];

                c0 = element→coeffs[0];
                c1 = element→coeffs[1];
                c2 = element→coeffs[2];

                VSub(V, P, element→skeleton.pos);
                VDot(t0, V, V);
                VDot(t1, V, D);
                tcoeffs = &(element→tcoeffs[0]);

                tcoeffs[0] = c0;
                tcoeffs[1] = 4.0 * c0 * t1;
                tcoeffs[2] = 2.0 * c0 * (2.0 * t1 * t1 + t0) + c1;
                tcoeffs[3] = 2.0 * t1 * (2.0 * c0 * t0 + c1);
                tcoeffs[4] = c0 * t0 * t0 + c1 * t0 + c2;

                for (j = 0; j < 5; j++)
                        coeffs[j] += tcoeffs[j];

                break;
        case OUT:

                tcoeffs = &(implicit→list[intervals[i].index]→tcoeffs[0]);
                for (j = 0; j < 5; j++)
                        coeffs[j] -= tcoeffs[j];
                --in_flag;
                break;

        case IN_DEF:
                indef_flag++;
                element = implicit→deformlist[intervals[i].index]→list[intervals[i].index2];

                c0 = element→coeffs[0];
                c1 = element→coeffs[1];
                c2 = element→coeffs[2];

                VSub(V, P, element→skeleton.pos);
                VDot(t0, V, V);
                VDot(t1, V, D);
                tcoeffs = &(element→tcoeffs[0]);

                tcoeffs[0] = c0;
                tcoeffs[1] = 4.0 * c0 * t1;
                tcoeffs[2] = 2.0 * c0 * (2.0 * t1 * t1 + t0) + c1;
                tcoeffs[3] = 2.0 * t1 * (2.0 * c0 * t0 + c1);
                tcoeffs[4] = c0 * t0 * t0 + c1 * t0 + c2;

                for (j = 0; j < 5; j++)
                        dcoeffs[j] -= tcoeffs[j];

                break;
        case OUT_DEF:

                tcoeffs = &(implicit→deformlist[intervals[i].index]→list[intervals[i].index2]→tcoeffs[0]);
                for (j = 0; j < 5; j++)
                        dcoeffs[j] += tcoeffs[j];
                --indef_flag;
                break;

        case IN_BL:
                element = implicit→blendlist[intervals[i].index]→list[intervals[i].index2];

                c0 = element→coeffs[0];
                c1 = element→coeffs[1];
                c2 = element→coeffs[2];

                VSub(V, P, element→skeleton.pos);
                VDot(t0, V, V);
                VDot(t1, V, D);
                tcoeffs = &(element→tcoeffs[0]);
```

```
            tcoeffs[0] = c0;
            tcoeffs[1] = 4.0 * c0 * t1;
            tcoeffs[2] = 2.0 * c0 * (2.0 * t1 * t1 + t0) + c1;
            tcoeffs[3] = 2.0 * t1 * (2.0 * c0 * t0 + c1);
            tcoeffs[4] = c0 * t0 * t0 + c1 * t0 + c2;

            for (j = 0; j < 5; j++)
                bcoeffs[intervals[i].index][j] += tcoeffs[j];

            if (!(inbl_flag_array[intervals[i].index])++)
                inbl_flag++;
            break;
        case OUT_BL:

            tcoeffs = &(implicit→blendlist[intervals[i].index]→list[intervals[i].index2]→tcoeffs[0]);

            for (j = 0; j < 5; j++)
                bcoeffs[intervals[i].index][j] -= tcoeffs[j];

            if (!--(inbl_flag_array[intervals[i].index]))
                --inbl_flag;
            break;

        default:
            Error("Bl: Unrecognized interval type");
            break;
        }                                                    /* switch interval type */

        /* now find intersections */

        if (in_flag) {

            /* for each component in the blendable interval list we'll add it to
             F(P), look for intersections (both undeformed and deformed)
             check if it's Inside Implicit then subtract its influence - linear... */

            b = inbl_flag; /* current bcoeffs considered */
            binx = 0; /* to find the element in bcoeffs array */

            do {

                if (b>0) {
                    while (binx < implicit→blcount && !inbl_flag_array[binx]) binx++;
                    for(k=0; k<5; k++)
                        coeffs[k] += bcoeffs[binx][k];
                }

                /* first the undeformed component */

                /* Figure out which root solver to use */
                if (implicit→Sturm_Flag == 0)
                    /* Use Ferrari's method */
                    root_count = solve_quartic(coeffs, &roots[0]);
                else
                    /* Sturm sequences */
                    if (fabs(coeffs[0]) < COEFF_LIMIT)
                        if (fabs(coeffs[1]) < COEFF_LIMIT)
                            root_count = solve_quadratic(&coeffs[2], &roots[0]);
                        else
                            root_count = polysolve(3, &coeffs[1], &roots[0]);
                    else
                        root_count = polysolve(4, coeffs, &roots[0]);

                /* See if any of the roots are valid */
                for (j = 0; j < root_count; j++) {
                    dist = roots[j];
                    /*
* First see if the root is in the interval of
* influence of the currently active components of
* the implicit
*/
                    if ((dist ≥ intervals[i].bound) &&
                        (dist ≤ intervals[i+1].bound) &&
                        (dist > Implicit_Tolerance)) {
                            VScale(IPoint, D, dist);
                            VAdd(IPoint, IPoint, P);
                            /* Transform the point into world space */
                            if (implicit→Trans ≠ NULL)
                                MTransPoint(&IPoint, &IPoint, implicit→Trans);
                            if (Inside_Implicit(&IPoint, Object)) {
```

```
                                        VSub(dv, IPoint, Ray→Initial);
                                        VLength(len, dv);
                                        if (Point_In_Clip(&IPoint, Object→Clip)) {
                                                push_entry(len, IPoint, Object, Depth_Stack);
                                                Intersection_Found = TRUE;
                                        }
                                }
                        }
                }                                                                       /* for root...*/
                if (Intersection_Found)
                        Ray_Implicit_Tests_Succeeded++;


                if (indef_flag) {

                        coeffs[4] += implicit→threshold;
                        for (k=0; k<5; k++)
                                coeffs[k] += dcoeffs[k];

                        /* Figure out which root solver to use */
                        if (implicit→Sturm_Flag == 0)
                                /* Use Ferrari's method */
                                root_count = solve_quartic(coeffs, &roots[0]);
                        else
                                /* Sturm sequences */
                                if (fabs(coeffs[0]) < COEFF_LIMIT)
                                        if (fabs(coeffs[1]) < COEFF_LIMIT)
                                                root_count = solve_quadratic(&coeffs[2], &roots[0]);
                                        else
                                                root_count = polysolve(3, &coeffs[1], &roots[0]);
                                else
                                        root_count = polysolve(4, coeffs, &roots[0]);

                        /* get rid of deformation coefficients */

                        coeffs[4] -= implicit→threshold;
                        for (k=0; k<5; k++)
                                coeffs[k] -= dcoeffs[k];


                        /* See if any of the roots are valid */
                        for (j = 0; j < root_count; j++) {
                                dist = roots[j];
                                /*
 * First see if the root is in the interval of
 * influence of the currently active components of
 * the implicit
 */
                                if ((dist ≥ intervals[i].bound) &&
                                    (dist ≤ intervals[i+1].bound) &&
                                    (dist > Implicit_Tolerance)) {
                                        VScale(IPoint, D, dist);
                                        VAdd(IPoint, IPoint, P);
                                        /* Transform the point into world space */
                                        if (implicit→Trans ≠ NULL)
                                                MTransPoint(&IPoint, &IPoint, implicit→Trans);
                                        if (Inside_Implicit(&IPoint, Object)) {
                                                VSub(dv, IPoint, Ray→Initial);
                                                VLength(len, dv);
                                                if (Point_In_Clip(&IPoint, Object→Clip)) {
                                                        push_entry(len, IPoint, Object, Depth_Stack);
                                                        New_Intersection_Found = TRUE;
                                                }
                                        }                                               /* if Inside_. */
                                }                                                       /* if dist >= int... */
                        }                                                               /* for j<root_count */
                        if (New_Intersection_Found)
                                Ray_Implicit_Tests_Succeeded++;
                }                                                                       /* if indef_flag */

                /* clear blending coefficients */

                if (b>0)
                        for(k=0; k<5; k++)
                                coeffs[k] -= bcoeffs[binx][k];

                b--;
        } while (b>0);                                                  /* if b ... == consider next blendable component */
    }                                                                                   /* if in_flag */

}                                                                                       /* for i<cnt */
```

```
    return Intersection_Found || New_Intersection_Found;
}
```

A.4 Normal vectors

At any point of a scalar field a normal vector to the isosurface going through this point is aligned with the gradient of the field at this point [Bron85]. The gradient can be simply calculated as partial derivatives of the function F which describes the scalar field:

$$N = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial z}, \frac{\partial F}{\partial z} \right)$$

To calculate the gradient in POVray, for each intersection point the field function is computed according to the algorithm for implicit surfaces with blending properties, presented in Section 3.3. The result is a function F that contains a sum and difference of field functions due to all primitives that contributed to its value:

$$F(P) = \sum_{i \in C} f_i(P) + \sum_{i \subseteq \mathcal{B}(\mathcal{C})} f_i(P) - \sum_{i \subseteq \mathcal{U}(\mathcal{C})} f_i(P) + constant$$

The gradient of such a function is calculated as:

$$\nabla F(P) = \sum_{i \in C} \nabla f_i(P) + \sum_{i \subseteq \mathcal{B}(\mathcal{C})} \nabla f_i(P) - \sum_{i \subseteq \mathcal{U}(\mathcal{C})} \nabla f_i(P)$$

Each partial derivative $\frac{\partial F}{\partial x}$, $\frac{\partial F}{\partial y}$ and $\frac{\partial F}{\partial z}$ can be therefore calculated as:

$$\frac{\partial F}{\partial x}(P) = \sum_{i \in C} \frac{\partial f_i}{\partial x}(P) + \sum_{i \subseteq \mathcal{B}(\mathcal{C})} \frac{\partial f_i}{\partial x}(P) - \sum_{i \subseteq \mathcal{U}(\mathcal{C})} \frac{\partial f_i}{\partial x}(P)$$

Therefore, it is sufficient to calculate all partial derivatives due to field functions contributing to the point P and combine them according to the blending properties algorithm.

Let us consider the field function used in the coefficient form:

$$f(r) = c_0(r^2)^2 + c_1 r^2 + c_2$$

where $r^2 = (x_G - x)^2 + (y_G - y)^2 + (z_G - z)^2$ for a generator $G = (x_G, y_G, z_G)$. The partial derivative of such a function can be calculated as:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial r} \frac{\partial r}{\partial x}$$

The partial derivative $\frac{\partial f}{\partial r}$ is calculated as follows:

$$\frac{\partial f}{\partial r} = 4c_0 r^3 + 2c_1 r$$

Since:

$$r = \sqrt{(x_G - x)^2 + (y_G - y)^2 + (z_G - z)^2}$$

its partial derivative $\frac{\partial r}{\partial x}$ is given by:

$$\frac{\partial r}{\partial x} = \left(\frac{\frac{\partial A}{\partial x}}{2\sqrt{A}} \right) = \left(\frac{-2\,(x_G - x)}{2\sqrt{A}} \right)$$

where $\sqrt{A} = r$. The derivative $\frac{\partial f}{\partial x}$ is therefore:

$$\frac{\partial f}{\partial x} = \frac{\left(4c_0 r^3 + 2c_1 r\right)\left(-\left(x_G - x\right)\right)}{r} = \left(4c_0 r^2 + 2c_1\right)\left(-\left(x_G - x\right)\right)$$

The following procedure calculates the normal vector at a given intersection point:

```
void
Implicit_Normal(Result, Object, IPoint)
OBJECT *Object;
VECTOR *Result, *IPoint;
{
        VECTOR New_Point, V, DefResult;
        int i, j, maxi;
        double dist, val, density, defdensity, defval, defdens, maxbldens, bldens;
        IMPLICIT *implicit = (IMPLICIT *) Object;
        Component *temp;
        int indef_flag;

        /* Transform the point into the blobs space */
        if (implicit->Trans != NULL)
                MInvTransPoint(&New_Point, IPoint, implicit->Trans);
        else
                New_Point = *IPoint;

        Make_Vector(Result, 0, 0, 0);

        /*
         * For each component that contributes to this point, add its bit to
         * the normal
         */
        density = 0.0;
        for (i = 0; i < implicit->count; i++) {
                temp = implicit->list[i];
                dist = calculate_skeleton_distance2(&(implicit->list[0]), i, &New_Point, &V);
                if (dist <= temp->radius2) {
                        density += dist * (dist * temp->coeffs[0] + temp->coeffs[1]) + temp->coeffs[2];
                        val = -2.0 * (2.0 * temp->coeffs[0] * dist + temp->coeffs[1]);
                        Result->x += val * V.x;
                        Result->y += val * V.y;
                        Result->z += val * V.z;
                }
        }

        /* this is the normal based on undeformed density, add deformation */

        Make_Vector(&DefResult, 0, 0, 0);
        defdensity = 0.0;
        indef_flag = FALSE;
        for (i = 0; i < implicit->defcount; i++) {
                for (j = 0; j < implicit->deformlist[i]->count; j++) {
                        temp = implicit->deformlist[i]->list[j];
                        dist = calculate_skeleton_distance2(&(implicit->deformlist[i]->list[0]),
                                j, &New_Point, &V);
                        if (dist <= temp->radius2) {
                                defdens = dist * (dist * temp->coeffs[0] + temp->coeffs[1]) + temp->coeffs[2];
                                if (defdens >= implicit->isovalue) indef_flag = TRUE;
                                defdensity += defdens;
                                defval = -2.0 * (2.0 * temp->coeffs[0] * dist + temp->coeffs[1]);
                                DefResult.x += defval * V.x;
                                DefResult.y += defval * V.y;
                                DefResult.z += defval * V.z;
                        }
                }
        }

        if (indef_flag && density >= implicit->isovalue) {
                Result->x -= DefResult.x;
                Result->y -= DefResult.y;
                Result->z -= DefResult.z;
        }

        maxbldens = 0.0;
        maxi = -1;
        for (i = 0; i < implicit->blcount; i++) {
                bldens = 0.0;
                for (j = 0; j < implicit->blendlist[i]->count; j++) {
                        temp = implicit->blendlist[i]->list[j];
                        dist = calculate_skeleton_distance2(&(implicit->blendlist[i]->list[0]),
                                j, &New_Point, &V);
                        if (dist <= temp->radius2) {
                                bldens += dist * (dist * temp->coeffs[0] + temp->coeffs[1])+temp->coeffs[2];
                        }
                        if (bldens > maxbldens) {
                                maxi = i;
                                maxbldens = bldens;
                        }
```

```
                }
        }

        if (maxi>-1)
                for (j = 0; j < implicit→blendlist[maxi]→count;j++) {
                        temp = implicit→blendlist[maxi]→list[j];
                        dist = calculate_skeleton_distance2(&(implicit→blendlist[maxi]→list[0]),
                                j, &New_Point, &V);
                        if (dist ≤ temp→radius2) {
                                val = -2.0 * (2.0 * temp→coeffs[0] * dist + temp→coeffs[1]);
                                Result→x += val * V.x;
                                Result→y += val * V.y;
                                Result→z += val * V.z;
                        }
                }

        val = (Result→x * Result→x + Result→y * Result→y + Result→z * Result→z);
        if (val < EPSILON) {
                Result→x = 1.0;
                Result→y = 0.0;
                Result→z = 0.0;
        } else {
                val = 1.0 / sqrt(val);
                Result→x *= val;
                Result→y *= val;
                Result→z *= val;
        }

        /* Transform back to world space */
        if (implicit→Trans ≠ NULL)
                MTransNormal(Result, Result, implicit→Trans);
        VNormalize(*Result, *Result);
}
```

Appendix B

Speeding Up Grid-Data Generation for Polygonisation of Implicit Surfaces

This appendix presents a paper published in the proceedings of the 13th Annual Eurographics UK Chapter Conference, in Loughborough, 28th-30th March 1995.

Speeding Up Grid-Data Generation for Polygonisation of Implicit Surfaces

Agata Opalach Steve Maddock

Department of Computer Science, The University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK
Tel: +44 114 282 5577 Fax: +44 114 278 0972
E-mail: a.opalach@dcs.shef.ac.uk

Abstract

Implicit surfaces are particularly suitable for design and animation of smoothly blended objects that deform during motion. Such objects change their shape due to external forces, deform during collisions with other objects or conform to their environment. They are also able to squash and stretch, an effect essential in traditional animation yet often lacking in computer animation. However, visualisation of implicit surfaces is not yet achieved at interactive rates. This limits the practical application of implicit surfaces in the area of interactive design and animation. One way of displaying implicit surfaces is to polygonise them and use a fast hardware polygon renderer to create a high quality image. This paper proposes a way of speeding up grid-data generation, one of the phases of the polygonisation process.

1 Introduction

Implicit surfaces as a modelling technique were initially described by Blinn (blobby molecules) [2], Nishimura *et al.* (metaballs) and Wyvill *et al.* (soft objects) [19]. All three approaches used scalar fields around 3D points to build their models. The technique was later generalised for arbitrary skeletal elements [20, 5, 4] and has found practical applications in a wide variety of areas. For instance, Tatsumi et al. present the use of metaballs in the process of modelling the cod liver [16], Fujita *et al.* represent splashing water using metaballs [7], Muraki uses a blobby model for volumetric shape description (including human face modelling) [12], Max and Wyvill render coral as soft objects [10] and Payne and Toga model a rat's brain using distance fields [15].

Implicit surfaces have also been appreciated in the field of computer animation. In Japan, metaballs have been widely used for modelling and animation of shapes as complex as human anatomy parts [9] Another use for the technique is in visualising deformable material modelled as particles [11, 18] or mass points [17]. Implicit surfaces have also been used for modelling precise contact between colliding implicitly defined solids [8] and simulating highly deformable material which can break into pieces [6].

Beier at Pacific Data Images commented on the usefulness of blobby primitives for character animation [1]. Our work [13] describes the ABC of implicit surfaces: appearance, blending and coherence. These

three properties are important for character animation with implicit surfaces. In [14] we used implicit surfaces to implement cartoon effects (or traditional animation principles) in a character animation system. The growing popularity of implicit surfaces calls for their interactive visualisation.

First let us briefly introduce the terminology of implicit surfaces. Consider a set of skeletal elements (points, lines, polygons) with a scalar field function $f_i : \mathcal{R}^3 \rightarrow \mathcal{R}$ defined for them. An implicit surface can be defined as $\{P \in \mathcal{R}^3 \mid \mathcal{F}(\mathcal{P}) - \mathcal{I}\int\wr = \prime\}$ where Iso is the isovalue at which an isosurface is extracted and $F(P) = \sum_{i=1}^{n}(f_i(P))$. Figure 1 shows an implicit surface created around two points. We will refer to *skeletal elements* as ¡

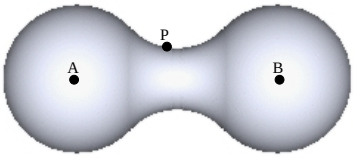


Figure 1: *An implicit surface around two skeletal elements, or primitives, A and B*

Two main techniques are used to visualise implicit surfaces. The first is polygonisation, in which an implicit model is converted to a polygonal representation and then a standard polygon renderer is used to render the polygons. The second is ray tracing. Here, ray-surface intersections are computed using the implicit definition of a surface and the intersections are then used by a ray tracing program to render the surface.

Since fast polygon rendering hardware already exists, we shall consider techniques to speed up the polygonisation process. One standard way to polygonise an implicit surface is to divide a 3D space into grid cells. There are three stages in this method: (i) generating grid-data, (ii) identifying grid cells intersecting the implicit surface and (iii) calculating the polygons. In the first stage (i), the scalar field values at the grid points (corners of grid cells) have to be calculated. During stage (ii) all cells which are intersected by the surface (boundary cells) (Figure 2) are found and in the stage (iii) polygons in each boundary cell are produced according to the scalar field values in the corners of the cell. Previous attempts have been made to speed up steps (ii) and (iii) of polygonisation [21, 3, 5]. Our experiments are concerned with the first phase of the polygonisation process, the grid-data generation (i).

2 Grid-data generation algorithms

An implicit surface is defined by primitives positioned in space. If we divide the space into grid cells, some grid points will be influenced by a number of primitives (Figure 2), *i.e.* the primitives give non-zero contributions to the scalar field value at those grid points. We can identify two techniques for generating grid-data: *the naive method* and *the update method*.

2.1 Naive method

The naive method of generating the grid-data is to iterate over all primitives (including non-influencing primitives) and sum the scalar field contribution from them for each grid point. In this simple approach the process of calculating scalar field values at all grid points has to be repeated each time primitives are added, deleted or moved in the scene. When only a few primitives are involved, large parts of the scene remain unchanged, yet all the grid-data is recalculated. We can address such computation inefficiency using the update method.

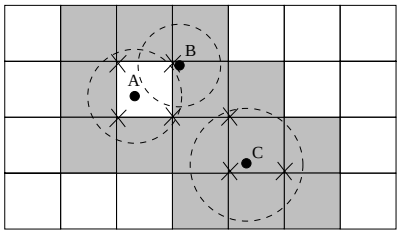


Figure 2: *(2D case) Three primitives A, B and C defining an implicit surface in a 2D grid. A circle around each primitive represents the radius of influence. Grid points within each radius of influence are shown as crosses. The boundary cells are shaded in grey.*

2.2 Update method

In the update method the initial stage is to store a list of influencing primitives for each grid point. Then, at a given grid point, only primitives influencing it are considered in the calculation of the scalar field value. The update method requires additional data structures to be created and maintained for a scene. Once the data structure is created, it has to be updated for each change occurring in the scene. There is no need to recalculate the parts of the scene which have not been altered. The primitives added, deleted or moved have to be inserted, deleted or moved to the influence lists of relevant grid points and the field value at the affected grid points has to be recalculated.

We will compare the two methods of grid-data generation in two experiments: (i) generating a static image of a scene and (ii) generating the frames of an animation sequence.

3 Static image experiment

The first experiment was designed to compare the performance of the two grid-data generation techniques in the case of polygonising a single image of a static scene.

3.1 Test sets

The test data for the static image experiment consisted of a randomly generated cluster of primitives (each of the same radius) in the middle of a cubic world (Figure 3). The world dimensions were 100x100x100 units and primitives were randomly placed in the 50x50x50 units cube in the centre of the world. For each test case the number of primitives in the scene and their radius were independently varied. In one set of tests the number of primitives was fixed at 32 and their radius was varied from 10 to 100 (Table 1 in the Appendix). In the second set, the number of primitives was fixed at 128 (Table 2 in the Appendix). The time required for generating grid-data for both grid-data generation methods was measured. For the update method, the time required for creating the additional data structure was also added. Ten tests in each configuration were run and the results averaged.

3.2 Results and Discussion

We expected that the update method would be considerably faster than the naive method. However, the results presented in Figure 4 for 32 and 128 primitives show that the update method only works better for small (radius 10-50) and fewer (32) primitives. When the size of primitives gets larger (close to the size of the world), the update method proves to be very inefficient.

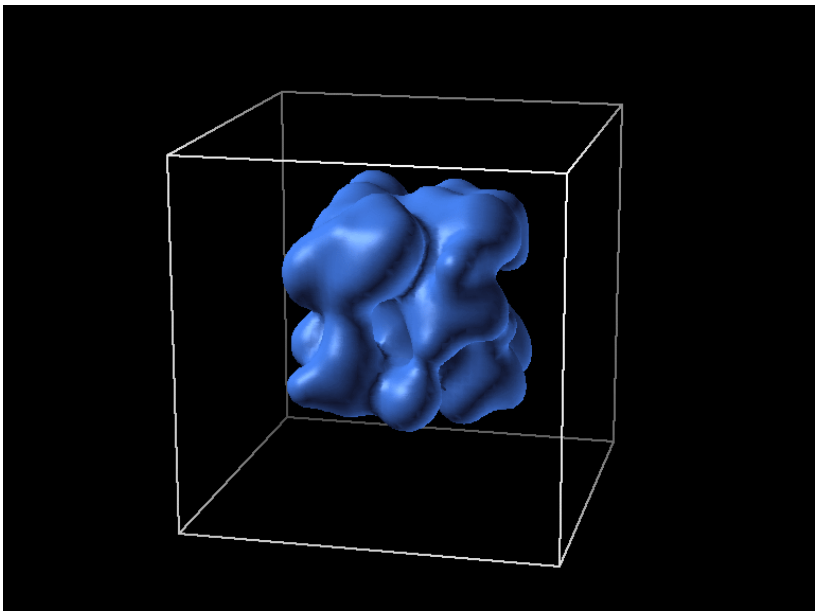


Figure 3: *Static image test: 128 primitives, radius 25*

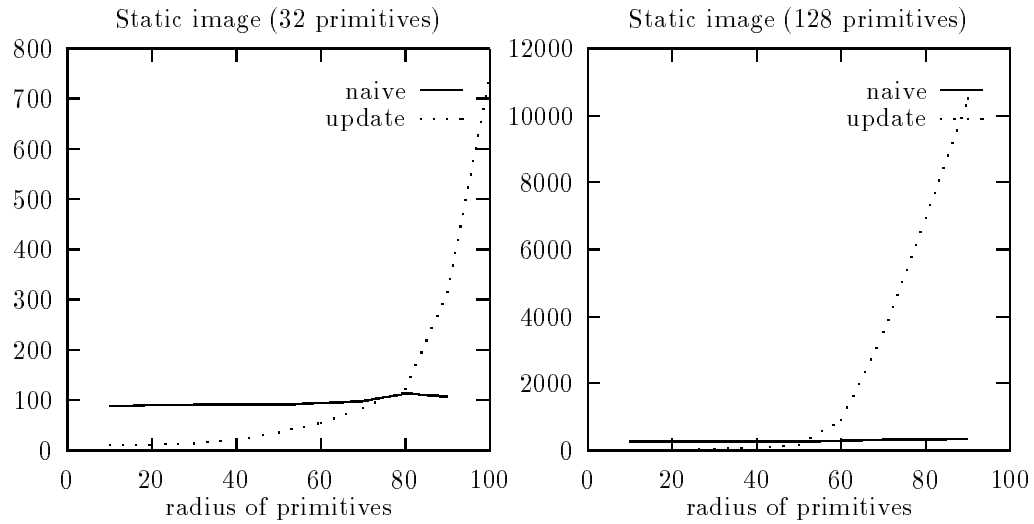


Figure 4: *The results of the static image experiment: The plots show the time in seconds of grid-data generation for the scene from Figure 3 for varying radius of primitives in two cases: 32 and 128 primitives*

The naive method has a constant cost which depends only on the number of primitives in the scene. The update method depends on the radius of primitives in the scene: the larger the primitives, the more grid points they influence and therefore the longer the time required to create the additional data structure. In a practical situation most of the primitives in a scene would probably be small in relation to the world size. In these cases using the update method for grid generation would reduce the computation time.

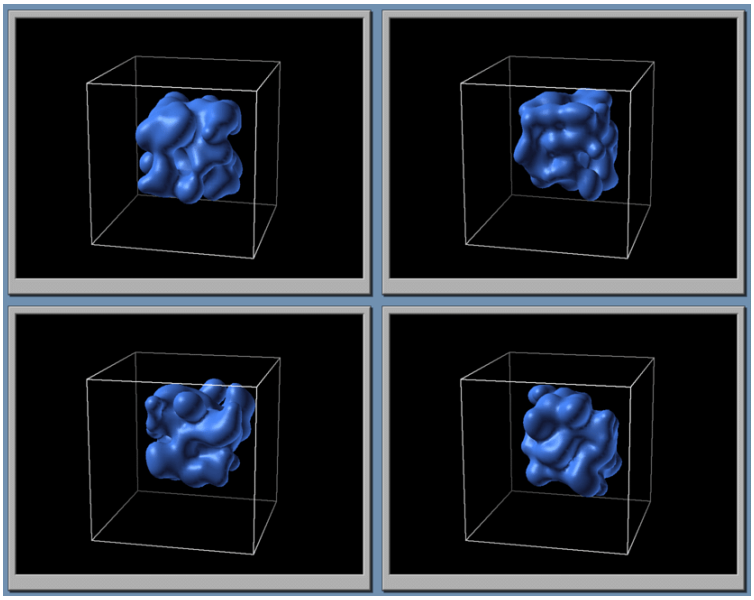


Figure 5: *Animation test: 128 primitives, radius 25, randomly moving 100 primitives*

4 Animation experiment

The animation experiment was designed to compare the two techniques during the creation of several frames in an animation sequence. In this case the update method takes advantage of the fact that only some of the primitives move during each animation step (Note: all primitive can move in the worst case).

4.1 Test sets

The animation experiment used the same scene as in the static image experiment. For each animation step a number of primitives in the scene were moved in random directions. The translation vector had a random length within the range from one to two grid cells. Figure 5 shows frames from the tested animation sequence. For each test case the number of primitives in the scene, their radius and the number of primitives moving during the animation were independently varied. The number of primitives changed from 32 to 128, their radius from 10 to 60 and the number of moving primitives from 10 to the total number of primitives in the scene. In each test ten frames of animation were created and the average time for grid-data generation for one frame was calculated. In the update method the time for creating and updating the secondary data structure was also added to the total time. Each test was run ten times and the results averaged.

4.2 Results and Discussion

Figures 6 and 7 and Tables 3-8 in the Appendix show the results of the animation experiment for 32 and 128 primitives respectively. The plots show how the computation time changes depending on the number of primitives moved in an animation step. The plots represent different radii of the primitives in the scene. The update method proves more efficient in most cases. However, when the radius of primitives in the scene grows larger (radius 50+, world size 100x100x100) and nearly all primitives in the scene are moved, the naive method works better.

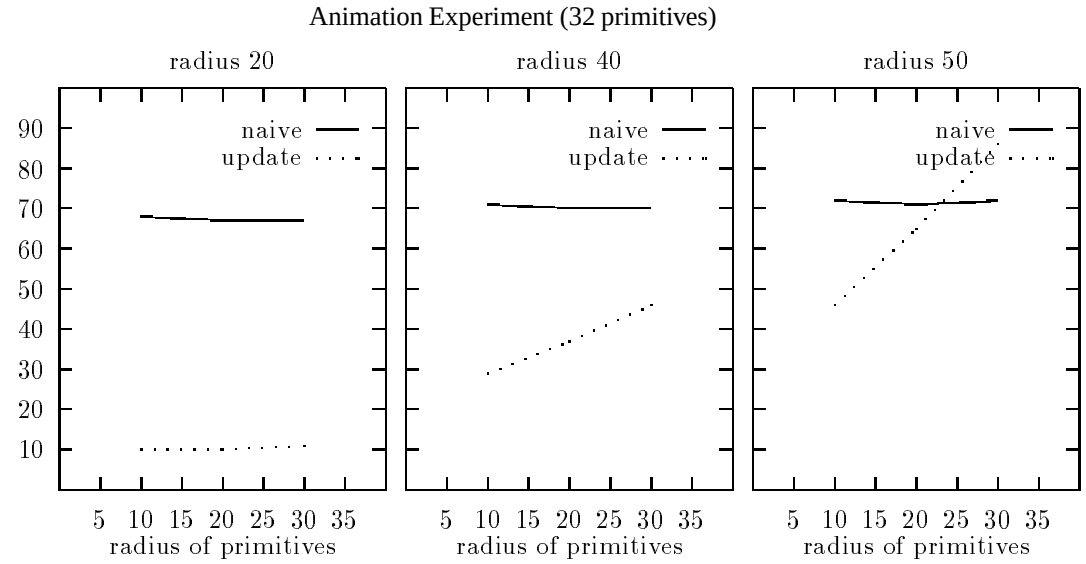


Figure 6: The results of the animation experiment: The plots show the average time in seconds of polygonising one frame of animation from Figure 5 for the radius of primitives 32 and a varied number of moved primitives

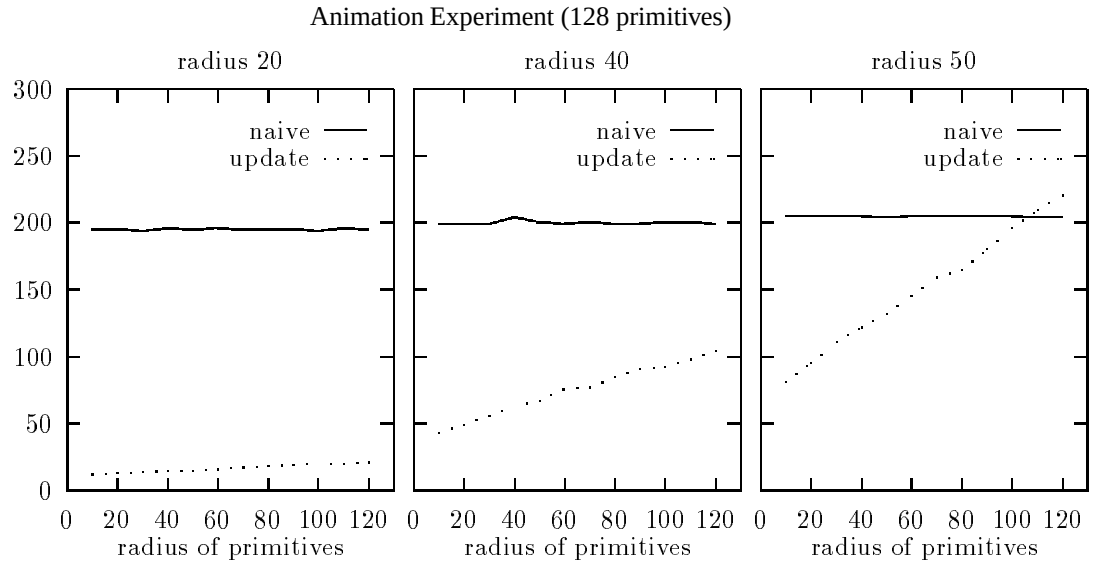


Figure 7: The results of the animation experiment: The plots show the average time in seconds of polygonising one frame of animation from Figure 5 for the radius of primitives 128 and a varied number of moved primitives

5 Conclusions

We have run a series of tests comparing two methods of grid-data generation for polygonising implicit surfaces: the naive method and the update method. The update method gives a significant reduction in computation time for smaller primitives (half the size of the world and smaller) for both the static image and animation experiments. For larger primitives (larger than half the size of the world) the naive method appears to be more efficient since it has a constant cost for all cases. The main advantage of the update method is shown during animation, when it uses the fact that only some primitives in the scene move. Hence, only grid- relation to the size of the world) primitives in the scene and only some of them will be moving in each animation step. In these cases the update method considerably speeds up the grid-data generation process.

The use of implicit surfaces for modelling and animation is growing in popularity. Speeding up their display is important for interactive applications. For instance, a designer or an animator have to view their creations in real time in order to be able to use implicit surfaces for interactive design or creating computer animation. When interactively sculpting an object an artist or designer is likely to change only one part of it at a time. Similarly an animation sequence will usually consist of a static background and only a few objects moving. In these cases the update method of grid-data generation can accelerate the visualisation process.

References

[1] Thad Beier. Practical uses for implicit surfaces in animation. In *Modeling and Animating with Implicit Surfaces*, pages 20.1–20.11, 1990. SIGGRAPH Course Notes 23.

[2] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[3] Jules Bloomenthal. Polygonisation of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.

[4] Jules Bloomenthal and Ken Shoemake. Convolution surfaces. *Computer Graphics*, 25(4):251–256, July 1991. Proceedings of SIGGRAPH'91.

[5] Jules Bloomenthal and Brian Wyvill. Interactive techniques for implicit modeling. *Computer Graphics*, 24(2):109–116, March 1990. Proceedings of SIGGRAPH'90.

[6] Mathieu Desbrun and Marie-Paule Gascuel. Highly deformable material for animation and collision processing. In *Fifth Eurographics Workshop on Animation and Simulation*, Oslo, Norway, September 1994.

[7] Takushi Fujita, Katsuhiko Hirota, and Kouichi Murakami. Representation of splashing water using metaball model. *Fujitsu*, 41(2):159–165, 1990. in Japanese.

[8] Marie-Paule Gascuel. An implicit formulation for precise contact modelling between flexible solids. *Computer Graphics*, 27:313–320, 1993. Proceedings of SIGGRAPH'93.

[9] Gaye L. Graves. The magic of metaballs. *Computer Graphics World*, pages 27–32, May 1993.

[10] Nelson L. Max and Geoff Wyvill. Shapes and textures for rendering coral. In N. M. Patrikalakis, editor, *Scientific Visualisation of Physical Phenomena*, pages 333–343. Springer-Verlag, 1991.

[11] Gavin Miller and Andrew Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers and Graphics*, 13(3):305–309, 1989.

[12] Shigeru Muraki. Volumetric shape description of range data using "blobby model". *Computer Graphics*, 25(4):227–235, July 1991. Proceedings of SIGGRAPH'91.

[13] Agata Opalach and Steve Maddock. Implicit surfaces: Appearance, Blending and Consistency. In *Fourth Eurographics Workshop on Animation and Simulation*, pages 233–245, Barcelona, Spain, September 1993.

[14] Agata Opalach and Steve Maddock. Disney effects using implicit surfaces. In *Fifth Eurographics Workshop on Animation and Simulation*, Oslo, Norway, September 1994.

[15] Bradley A. Payne and Arthur W. Toga. Distance field manipulation of surface models. *IEEE Computer Graphics and Applications*, pages 65–71, January 1992.

[16] Haruyuki Tatsumi, Eiji Takaoki, Koichi Omura, and Hisao Fujita. A new method for three-dimensional reconstruction from serial sections by computer graphics using "meta-ball": Reconstruction of "hepatoskeletal system" formed by ito cells in the cod liver. *Computers and Biomedical Research*, 23(part 1):37–45, 1990.

[17] Demetri Terzopoulos, John Platt, and Kurt Fleisher. Heating and melting deformable models (from goop to glop). In *Graphics Interface'89*, pages 219–226, June 1989.

[18] David Tonnesen. Modelling liquids and solids using termal particles. In *Graphics Interface'91*, pages 255–262, 1991.

[19] Brian Wyvill, Craig McPheeters, and Geoff Wyvill. Animating soft objects. *The Visual Computer*, 2:235–242, August 1986.

[20] Brian Wyvill and Geoff Wyvill. Field functions for implicit surfaces. *The Visual Computer*, 5:75–82, December 1989.

[21] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2:227–234, August 1986.

Appendix. Experimental Results

Static Image Experiment

radius / method	10	20	30	40	50	60	70	80	90
naive	88	90	91	92	92	94	98	113	107
update	10	11	14	22	36	55	84	122	316

Table 1: Static image experiment results [time] (32 primitives)

radius / method	10	20	30	40	50	60	70	80	90
naive	262	263	265	267	274	289	299	326	345
update	10	16	32	71	145	912	3532	6936	10501

Table 2: Static image experiment results [time] (128 primitives)

Animation Experiment

moved / method	10	20	30
naive	68	67	67
update	10	10	11

Table 3: Animation experiment results [time] (32 primitives, radius 20)

moved / method	10	20	30
naive	72	71	72
update	46	65	86

Table 4: Animation experiment results [time] (32 primitives, radius 40)

moved / method	10	20	30
naive	71	70	70
update	29	37	46

Table 5: Animation experiment results [time] (32 primitives, radius 50)

moved / method	10	20	30	40	50	60	70	80	90	100	110	120
naive	195	195	194	196	195	196	195	195	195	194	196	195
update	12	13	14	15	15	16	17	18	19	20	20	21

Table 6: Animation experiment results [time] (128 primitives, radius 20)

moved / method	10	20	30	40	50	60	70	80	90	100	110	120
naive	199	199	199	204	200	199	200	199	199	200	200	199
update	43	49	56	63	67	76	77	85	91	92	98	104

Table 7: Animation experiment results [time] (128 primitives, radius 40)

moved / method	10	20	30	40	50	60	70	80	90	100	110	120
naive	205	205	205	205	204	205	205	205	205	205	204	204
update	81	95	111	122	132	145	159	165	180	196	209	220

Table 8: Animation experiment results [time] (128 primitives, radius 50)