Exploring the Blobtree

de Groot, Erwin

Award date:
2003

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER's THESIS

Exploring the Blobtree

by
Erwin de Groot

Supervisors:
Huub van de Wetering
Brian Wyvill

Eindhoven, November 2003

Contents

Chapter 1

Introduction

Implicit modelling is a way of creating volumes which are called implicit surfaces. The most important property of implicit modelling is the simplicity with which complex models can be created. Despite of the simplicity of the model, visualization of an implicit surface takes a long time compared to the more conventional mesh representation. The increase in speed of computers in the last years, makes implicit modelling a more interesting alternative nowadays. Although the interest in implicit modelling is growing, still a lot of research needs to be done to make implicit modelling more useful.

The goal of this thesis is to explore different aspects of implicit models, with one kind of implicit modelling in particular: the Blobtree.

1.1 Motivation

Within computer science, I always was particularly interested in computer graphics. Before I started my master thesis, I already worked a little bit with implicit surfaces, so I knew implicit modelling was a subject I would enjoy. Furthermore, there are a lot of things left to explore in the field of implicit modelling, so implicit modelling makes a perfect research subject.

I did my research on implicit modelling at the University of Calgary (Canada). The first reason to do my research in Calgary is the expertise of their computer science department on implicit modelling. The second reason was to gain the experience of living abroad.

1.2 Implicit surfaces

Implicit surfaces are volume representations that in general have an organic shape. Several kinds of implicit surfaces exist (Metaballs, Soft Objects [14], R-functions [9]), but this thesis is especially focussed on the Blobtree [13].

1.2.1 Implicit surfaces in general

Implicit surfaces are volume representations where the surface of the volume is defined by an iso-value c of a real function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$. The volume is then defined by the collection of points p in $\mathbb{R}^3$ for which $f(p) \geq c$ and the surface is defined by the points p for which $f(p) = c$. For example, a sphere with radius R could be defined with $f(p) = R^2 - p_x^2 - p_y^2 - p_z^2$ and $c = 0$ (figure 1.1).

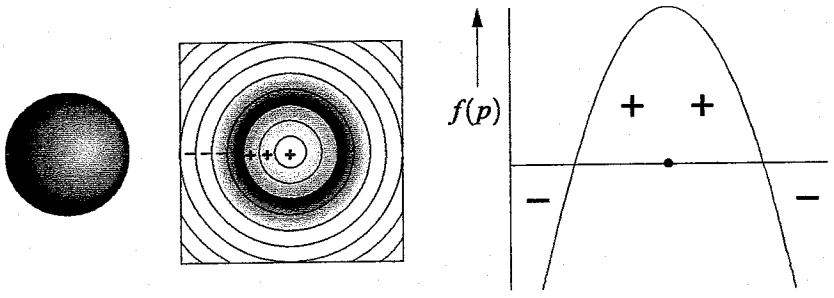

Figure 1.1: An implicit sphere. From left to right: the sphere, a 2D cross section and a 1D cross section, both through the center of the sphere.

The mathematical definition of implicit surfaces makes it possible to apply certain operations on them. For example, to take the union of two implicit surfaces defined by f_1 and f_2, in each point the maximum can be taken. The union can thus be defined by $f(p) = max(f_1(p), f_2(p))$ (figure 1.2). Depending on the kind of implicit surfaces that are used, several operations can be defined like intersection, difference and blend operations.

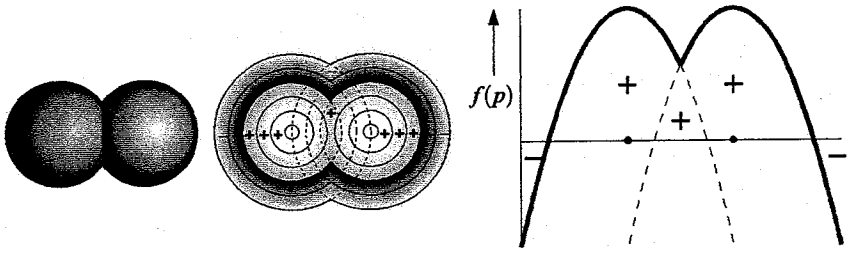

Figure 1.2: The union of two spheres. From left to right: the union, a 2D cross section and a 1D cross section, both through the centers of the spheres.

Implicit surfaces can in general be divided into two groups: bounded and unbounded implicit surfaces. A function that defines a bounded implicit surface returns a constant value outside a certain boundary. The values of a function that defines an unbounded implicit surface vary over the whole space. While unbounded implicit surfaces have less restrictions to meet, bounded implicit surfaces are usually faster to evaluate because their influence is restricted to a small part of the space.

1.2.2 The Blobtree

A Blobtree is a hierarchical tree structure (figure 1.4). The nodes of the tree represent operations (like union, intersection or blend) and the leaves of the tree represent simple implicit surfaces called primitives (like a sphere or a line).

The primitives are skeletal implicit surfaces. Skeletal implicit surfaces are bounded implicit surfaces that have a simple skeleton like a point, a line or a circle. The real function f, called field function, takes the distance to the skeleton d and returns a positive value. The output values of the field functions are called field values and they decrease when the distance increases. Furthermore there exists a distance R for which all distances equal to or bigger then R map to 0 (figure 1.3).

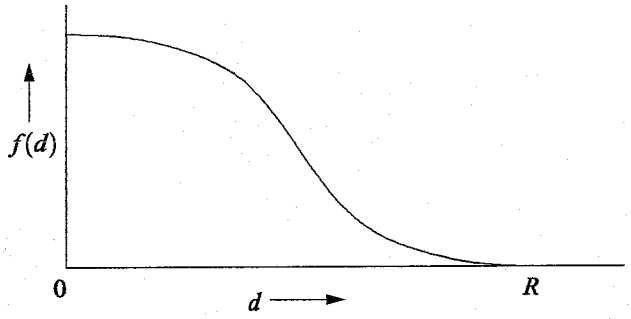

Figure 1.3: Example field function

The nodes of a Blobtree represent operations that act on their child nodes. For example, the field function of a blend could be the sum of the field values of its child nodes.

1.2.3 Existing Blobtree tools

Few tools for implicit surfaces exist and the support for implicit surfaces in public software is usually limited. Nevertheless, I was able to use the Blobtree software of the computer science department of the University of Calgary: JSP [1]. However, JSP has some disadvantages. It is hard to install and get it to work, because there doesn't exist a decent installer. Another problem is that the JSP software is still in development. Several times JSP seized to work for a certain amount of time (sometimes up to days) because changes were made to the software. Furthermore, the features of JSP are limited, for example, a raytracer (see section 2.3) is missing.

1.3 Projects

My assignment was to explore different aspects of the Blobtree. I started with experimenting with the Blobtree software and on my way I picked up some projects that came along. That is why the work for this thesis is divided among a number of projects.

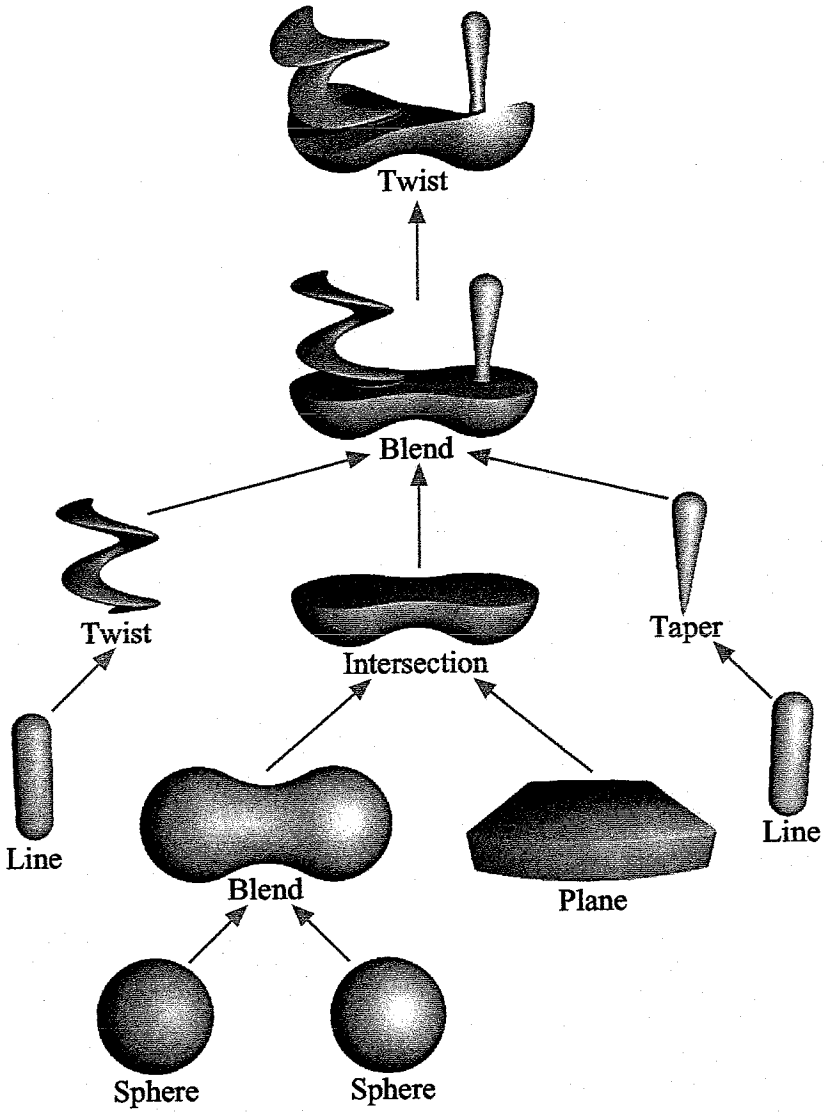

Figure 1.4: An example of a Blobtree structure

1.3.1 Blobtree.NET

To get familiar with the Blobtree, I started to write my own Blobtree software: Blobtree.NET (I implemented it using Microsoft's .NET framework [5]). In the beginning Blobtree.NET was only a test project that made it possible for me to experiment with the Blobtree and I did not have intentions to keep using it.

But after a short period of time Blobtree.NET became a quite sophisticated tool with several advantages over the existing Blobtree software. One of the biggest advantages was that I wrote the software myself, which made it easy for me to change or add features to it. This and other advantages made Blobtree.NET the perfect tool for my research.

Whenever I needed it, I added new features to Blobtree.NET. Soon Blobtree.NET became a quite advanced tool with several useful features like an adaptive polygonizer (section 2.2) and a raytracer (section 2.3).

1.3.2 Binary CSG operators for soft objects

At a certain moment my supervisor Brian asked me to implement the operations described in a paper Loïc Barthe and he were working on: Binary CSG operators for soft objects. The goal was to create some nice pictures illustrating the features of those operations. I used Blobtree.NET to implement the operations and I extended Blobtree.NET to produce pictures of higher quality (section 2.4).

1.3.3 Animation class

My supervisor Brian taught a course at the University of Calgary called 'Animation'. The goal of this course was to produce a short animation using various techniques. This year the main character 'Blobbie' was created using Blobtree.NET. I helped out the students of that class with the Blobtree.NET software and helped creating software that used Blobtree.NET (section 2.5). Having other people using my software gave me some insight in how to improve it and how to make it more user friendly.

1.3.4 Better blending between multiple nodes of the Blobtree

Blending is one of the most important features of implicit surfaces. One of the problems when blending two or more implicit surfaces is the control over the blending parts. Some solutions exist, but these solutions are only applicable in special cases. I wrote a paper that presents an intuitive method that works in all cases (chapter 3). The implementation of this method and the experimentations have been done in the Blobtree.NET software as well as creating the pictures.

Chapter 2

Blobtree.NET

Blobtree.NET is the new Blobtree tool I created during my project. It has been developed using C♯ [4] and runs on the Microsoft .NET platform [5]. The user interface and the Blobtree functionality are implemented in separate libraries. This makes it possible for other application to use the Blobtree functionality without the user interface (see sections 2.5.2 and 2.5.3).

The features of Blobtree.NET can roughly be divided into the following parts:

The Blobtree structure This includes the object structure, the evaluation of a Blobtree and related optimizations.

The polygonizer The method that converts a Blobtree into a mesh. Because of the short computation time this method needs, this method is especially used to see if a Blobtree has the wanted appearance.

The Raytracer The method that creates an accurate image of a Blobtree. This method is very time consuming, but creates really high quality pictures. That is why this method is usually just used to create the final pictures (after making sure with the polygonizer that the Blobtree has the wanted appearance).

Component features Features that make it possible to use the Blobtree.NET functionality inside other applications. The component features will be discussed in more detail in sections 2.5.2 and 2.5.3.

2.1 The Blobtree structure

A Blobtree is represented with a tree of objects. Each object either represents a node or a leaf in the tree and the objects representing a node have references to their child objects. Each object represents a Blobtree by itself, and can be evaluated for certain properties (section 2.1.1).

2.1.1 Evaluating the Blobtree

To get certain information from a Blobtree object, the object can be evaluated. In an evaluation the Blobtree object is queried for a certain property. The objects in Blobtree.NET can be evaluated for three different properties:

The field value The value returned by the field function in a given point

The normal The normal of the iso surface in a given point

The material The material of the Blobtree in a given point. The material includes colors (ambient, diffuse and specular) and other features like shininess, transparency and reflection.

The implementation of a Blobtree node could look as follows:

```
public class Node
{
    // Array of child nodes
    public Node[] Children;

    // Function to get the field value
    public float GetFieldvalue(Vector p);
    // Function to get the normal
    public Vector GetNormal(Vector p);
    // Function to get the material
    public Vector GetMaterial(Vector p);

    ...
}
```

The properties of a leaf of the Blobtree can be calculated directly, using only the data within that object. To calculate the properties of a node, the child objects usually need to be evaluated as well. The following pieces of code calculate the field values for a sphere primitive and a blend node:

```
// Sphere primitive
public float GetFieldvalue(Vector p) {
    return Fieldfunction(p.X*p.X + p.Y*p.Y + p.Z*p.Z);
}

// Blend node
public float GetFieldvalue(Vector p) {
    float f = 0;
    foreach (Node n in Children)
        f += n.GetFieldvalue(p);
    return f;
}
```

Evaluating the Blobtree is very time consuming and therefore some optimalizations have been build in (sections 2.1.2 and 2.1.3).

2.1.2 Bounding volumes

Bounding volumes are usually simple shapes for which easily can be tested whether a point is inside the volume or not (Blobtree.NET uses boxes and spheres). Common shapes for bounding volumes are spheres and boxes. When a bounding volume encapsulates a more complex volume, evaluations can be sped up by evaluating the bounding volume first. Only if a point happens to be inside the bounding volume, the more complex volume needs to be evaluated too. Usually a bounding volume only occupies a small part of the space where the evaluations take place in. Therefore most of the points that are evaluated lie outside the bounding volume and evaluating the complex volume in these points can be avoided.

When bounded implicit volumes are used (like in Blobtree.NET), bounding volumes can be used to encapsulate the influence of a node. The bounding volumes for the leaves of the Blobtree have to defined, but this is usually a simple task since the leaves have simple shapes. The bounding volumes of the nodes of the Blobtree have to be calculated by combining the bounding volumes of its child nodes. The bounding volume of a union node, for example, at least encapsulates the bounding volumes of its child nodes.

The bounding volumes greatly reduce the number of evaluations inside the Blobtree. For example, a blend node does not have to evaluate a certain child node when the point to be evaluated lies outside the bounding volume of that child.

```
// Blend node
public float GetFieldvalue(Vector p) {
    float f = 0;
    foreach (Node n in Children)
        if (n.BoudingVolume.Inside(p))
            f += n.GetFieldvalue(p);
    return f;
}
```

2.1.3 Caching values

Usually a Blobtree is evaluated for its field values first, maybe followed by evaluation for the normal and material. For example, a polygonizer (section 2.2) and a raytracer (section 2.3) try to find the surface of the Blobtree first by checking certain field values. Once the surface is found, the Blobtree is also evaluated for the normal and the material (for rendering purposes).

When for a certain point the field value, the normal and the material have to be calculated, there is a high probability some of the calculations are at least done twice. For example, when a union node is evaluated for its field value, it needs to calculate the field values of its child nodes. When the union node also needs to be evaluated for the normal or the material in the same point, the field values of the child nodes need to be evaluated again.

A solution could be to always calculate the field value, the normal and the material together. But in most cases it is unnecessary to calculate the normal or material, so this results in a loss of speed.

The solution used in Blobtree.NET works as follows. When the Blobtree is evaluated in a certain point for the first time, a new object is passed: the cache. When a node of the Blobtree is evaluated, it can store certain values in the cache object. For example, when a union node is evaluated it can store a reference to the child node with the maximum field value. When the union node is evaluated for a second time, there is no need to evaluate all the child nodes. Instead, only the child node for which the reference is stored in the cache object needs to be evaluated.

```
// Evaluating field values of the union node
public float GetFieldvalue(Vector p, Cache c) {
    // The maximum field value
    float maxf = float.MinValue;
    // The node with the maximum field value
    Node maxNode;
    // The cache object used to evaluate maxNode
    Cache maxCache;

    foreach (Node n in Children)
    {
        Cache cn = new Cache();
        f = n.GetFieldvalue(p, cn);
        if (f > maxf)
        {
            maxf = f;
            maxNode = n;
            maxCache = cn;
        }
    }

    // Store found values in cache
    c[0] = maxNode;
    c[1] = maxCache;

    return f;
}

// Evaluating normals of the union node
public Vector GetNormal(Vector p, Cache c) {
    Node maxNode = (Node)c[0];
    Cache maxCache = (Cache)c[1];
    return maxNode.GetNormal(p, maxCache);
}
```

2.2 The polygonizer

A polygonizer [7][8][12] is a method to visualize implicit surfaces. It tries to approximate the implicit surface with a mesh of polygons (triangles in this case). Once a mesh is created, it is easy to visualize that mesh using existing graphics drivers like OpenGL. An advantage of using this method is that the resulting mesh can be viewed from any angle, without having to evaluate the implicit surface again.

2.2.1 The algorithm

There are several possible ways to create a mesh from an implicit surface. In Blobtree.NET the algorithm is split up into the following number of steps.

1. **Loop through a grid** The space that needs to be searched will be divided into a regular grid of small cubes (figure 2.1). The idea is to create a little bit of the mesh in each cube. The smaller the cubes are, the more precise the result will be, but also the longer the algorithm will take. To avoid holes in the mesh, the edges of the separate parts of the mesh have to match up with their neighbors. The following steps are applied to each of the cubes in the grid.

2. **Split cube into tetrahedra** Because it is still too complicated to create a good set of polygons approximating the surface inside the cube, the cube will be split up into six tetrahedra (figure 2.2). The split up will be done in a way that the edges of the tetrahedra match up with the edges of the tetrahedra of the neighboring cubes. This is to prevent gaps in the resulting mesh. The next steps are applied on each of the tetrahedra.

3. **Check edges for intersections** For each of the corners of the tetrahedron the field values of the Blobtree are calculated. When a corner has a field value equal or higher than the iso-value c, the corner is inside the volume represented by the Blobtree. If the field value is lower than c, the corner is outside the volume. An edge intersects the surface of the volume when one of its endpoint is inside the volume and one is outside. When an intersection is detected, a more precise location of the intersection is calculated by using a binary search algorithm along the edge (until the wanted precision is reached). In figure 2.3 the intersections are marked as spheres on the edges.

4. **Create polygons** Now the intersections between the Blobtree volume and the edges of the tetrahedron are found, polygons can be created depending on the number of intersections. When there are no intersections, there is no need to create any polygons (first picture in figure 2.3). When there are three intersections, one polygon is created with the intersections as its corners (second picture in figure 2.3). When there are four intersections, two polygons that share an edge are created with the intersections as their corners (third picture in figure 2.3). This can actually be done in two ways, but in this algorithm both ways are equally good.

```
Mesh Polygonize(Node Blobtree){
    Mesh mesh = new Mesh();
    <Split up search space into grid of cubes>;
    foreach (Cube cube in grid)
    {
        <Split up cube into 6 tetrahedra>;
        foreach (Tetrahedron tetrahedron in cube)
        {
            <Check wether the corners of the tetrahedron
            are inside or outside the Blobtree>;
            foreach (Edge edge in tetrahedron)
            {
                if (<endpoints of edge
                    not on the same side>)
                {
                    <approximate intersection of edge
                    with the Blobtree with a binary
                    search along the edge>;
                }
            }
            if (<one intersection found>)
            {
                <create one polygon with intersections
                as corners>;
                <add polygon to mesh>;
            }
            if (<two intersections found>)
            {
                <create two polygons with intersections
                as corners>;
                <add both polygons to mesh>;
            }
        }
    }
    return mesh;
}
```

2.2.2 Refining the mesh

The previous algorithm produces a mesh with its points almost equally distributed along the surface of the Blobtree volume. The parts of the surface where the curvature is low, will probably be approximated fine. But the parts of the surface with high curvature (for example edges and creases), need more precision (figure 2.4).

Blobtree.NET contains another algorithm that refines the result of the polygonizer. This algorithm loops through all the edges of the mesh. Each edge will be checked for

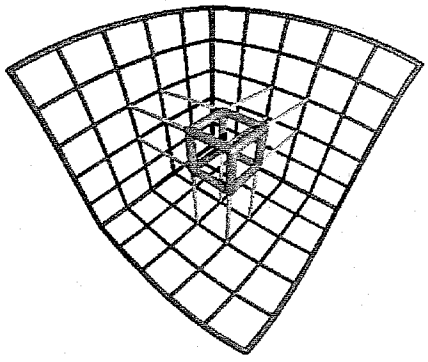

Figure 2.1: Step 1: each cube in the grid will be examined separately

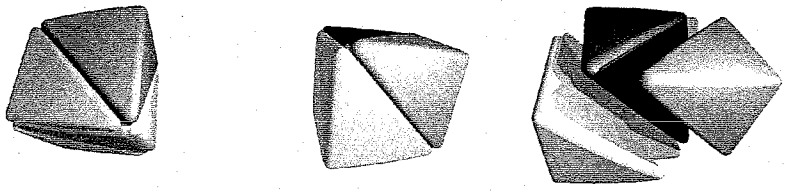

Figure 2.2: Step 2: splitting up the cube into six tetrahedra

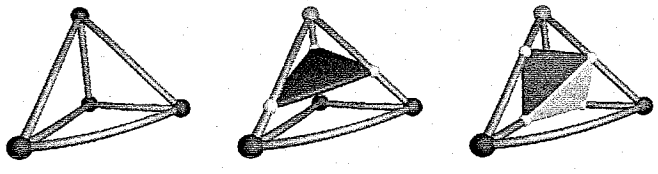

Figure 2.3: Step 3 and 4: Calculating the intersections with the Blobtree and creating the polygons. From left to right: 0, 3 and 4 intersections respectively.

2 criteria (figure 2.5):

1. 1. The normals of the endpoints of the edge must roughly point in the same direction. To check this, the dot product of the normals must be greater than a

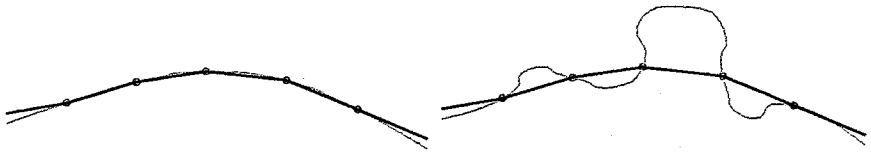

Figure 2.4: The surface of the Blobtree approximated with a mesh with equally distributed points. Left: a surface with low curvature. Right: a surface with high curvature. The grey line indicates the surface of the Blobtree and the black line indicates the mesh.

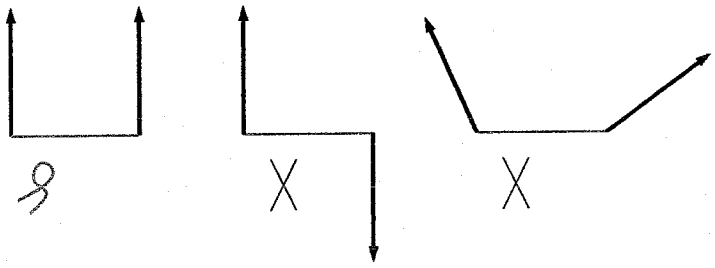

Figure 2.5: Edge check. From left to right: a good edge, an edge violating criterium 1 and an edge violating criterium 2

certain threshold (specified by the user).

2. 2. The normals of the endpoints of the edge must be close to perpendicular to the edge itself. To check this, the absolute value of the dot product of either normal and the edge, must be smaller than a certain threshold (specified by the user).

If an edge does not meet the criteria, the edge will be split (figure 2.6). The point that will be created to split the edge, has to be close to the surface of the Blobtree. To accomplish this, we will start with the point in the middle of the edge. In each step the normal in the point will be calculated and the point will be moved over a small distance in the direction of the normal. These steps will be repeated until the point ends up inside the Blobtree (figure 2.7).

```
void RefineMesh(Node Blobtree, ref Mesh mesh,
    float threshold1, float threshold2)
{
    foreach (Edge e in mesh)
    {
        // check if edge e meets the criteria
        bool isValid = true;
        if (e.point1.normal * e.point2.normal
            < threshold1)
        {
```

```
            isValid = false;
        }
        if (e.point1.normal * (e.point2 - e.point1)
            > threshold2)
        {
            isValid = false;
        }
        if (!isValid)
        {
            // edge e does not meet the criteria
            // create the initial point in the
            // middle of the edge
            Vector newpoint =
                (e.point1 + e.point2) * 0.5;
            // create cache object (see section 2.1.3)
            Cache cache = new Cache();
            // calculate the field value of the
            // new point
            float fieldvalue =
                Blobtree.GetFieldvalue(newpoint, cache);
            Vector normal;
            // do while the new point is outside the
            // Blobtree
            while (fieldvalue < 0)
            {
                // move the new point a little bit
                // along the normal
                normal = Blobtree
                    .GetNormal(newpoint, cache);
                newpoint += normal * <small number>;
                cache = new Cache();
                fieldvalue = Blobtree
                    .GetFieldvalue(newpoint, cache);
            }
            <remove polygons with edge e from mesh>;
            <create new edges and polygons as
            in figure 2.6>;
            <add new polygons to mesh>;
        }
    }
}
```

Figure 2.8 shows the effect of the extra refinement of the mesh. The mesh after the refinement is far more precise than before. Comparable results can also be accomplished by making the polygonization grid finer, but this will result in a relatively long computation time (figure 2.9).

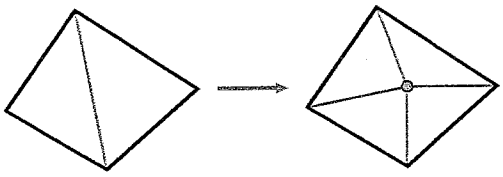

Figure 2.6: Splitting an edge

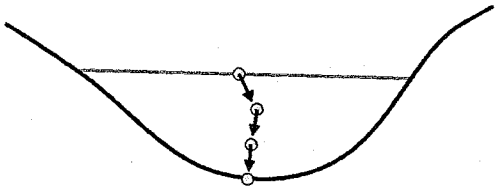

Figure 2.7: Finding a new point near the surface

2.3 The raytracer

The raytracer is another method that can be used to visualize a Blobtree. It creates a flat image of the Blobtree, but such an image is usually very detailed. Although raytracing a Blobtree is time consuming, is gives high quality results. Another advantage is that it is easy to add special effects, like bump mapping, transparency and reflection.

Most of the pictures I created for the papers 'Binary CSG operators for soft objects' (section 2.4) and 'Better blending between multiple nodes of the Blobtree' (chapter 3) are created using the raytracer of Blobtree.NET.

2.3.1 The algorithm

The idea behind the raytracing algorithm is to follow rays of light and look where they end up on the screen. To speed things up, in this algorithm the rays are followed in the opposite direction. The only rays that have influence on the result, are the ones that go through the screen and end up in the eye of the viewer (figure 2.10). These rays are followed backwards to see if they originated from a light source (possibly bouncing off objects in between).

The algorithm can be split up into the following steps:

1. **Creating initial rays** For each pixel p in the screen a ray is created that starts in the eye of the viewer and is pointed in the direction of p (figure 2.10). Step 2 will be applied on each of the rays. Pixel p will get the color returned by step 2.

2. **Traversing the ray to get its color** The ray is traversed until it either hits the surface of the Blobtree, hits a light source or leaves the scene. This is done by checking the field values along the ray ,starting in the origin of the ray and moving forward a little bit each time (see figure 2.11). When the ray does not hit

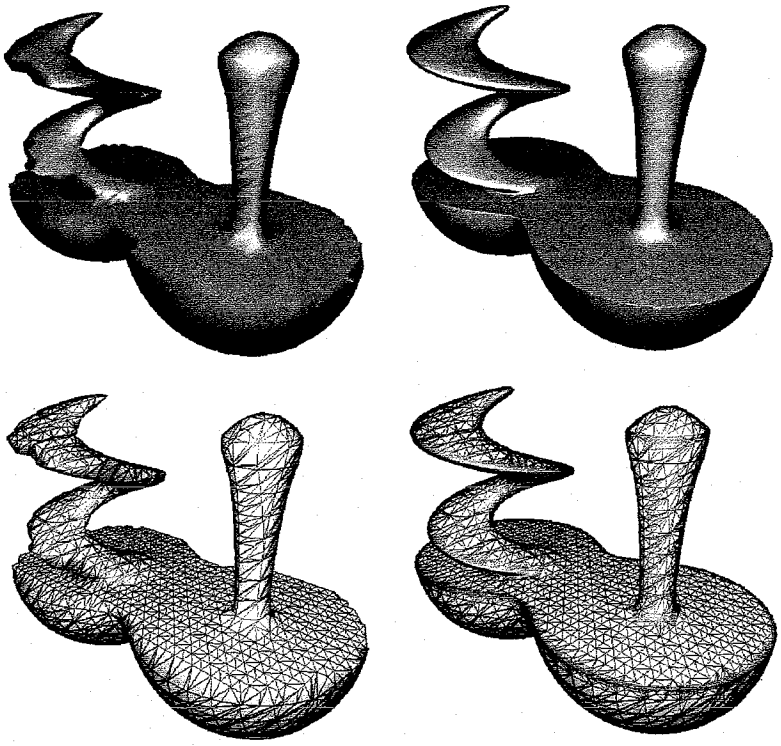

Figure 2.8: Resulting meshes. Left: without refinement. Right: with refinement. In the bottom row the edges of the polygons are drawn.

	coarse grid	coarse grid + refinement	fine grid
Computation time (seconds)	1.3	7.5	35.6
Evaluations of the Blobtree	43374	110478	1370998
Polygons	6676	56358	115600
Result	poor	really good	good

Figure 2.9: Comparison between different sets of parameters for the polygonization, evaluating the Blobtree of figure 2.8.

anything before leaving the scene, the ray does not carry any light (the return value is black). When the surface of the Blobtree or a light source is hit, the return value is calculated with step 3 or 4 respectively.

3. **Hitting the surface** When the surface of the Blobtree is hit in a certain point, the Blobtree is evaluated for both the normal and the material in that point. Depending on the material, several new rays are created. Usually for each light source a

ray is created pointing at that light source. Extra rays may be created for reflection or transparency. Step 2 will be applied on each of the new rays. Based on the returned colors a new color can be calculated (taking into account the length of the rays and the directions of the rays). This new color will be returned as the result (figure 2.12).

4. Hitting a light source When a light source is hit, simply the color of the light source will be returned.

```
Image Raytrace(Node Blobtree)
{
    foreach (Pixel p in screen)
    {
        <create ray r from the eye to Pixel p>;
        p.color = TraverseRay(Blobtree, r);
    }
} Color TraverseRay(Node Blobtree, Ray r) {
    bool hit = false;
    float stepsize = <small number>;
    while (!hit)
    {
        // move the current position a little bit
        // further along the ray
        r.position += r.direction * stepsize;
        // create cache object (see section 2.1.3)
        Cache c = new Cache();
        // calculate fieldvalue
        float f = Blobtree.GetFieldvalue(r.position, c);
        // check if the Blobtree is hit
        if (f >= 0)
        {
            // make sure the current position is
            // outside the Blobtree (undo last step)
            // to make sure new rays originating
            // from this point won't instantly
            // hit the surface
            r.position -= r.direction * stepsize;
            <create new rays as explained in step 3>;
            // get the colors returned by the new rays
            Collection colors = new Collection();
            foreach (Ray newray in newrays)
            {
                colors.Add(
                    TraverseRay(Blobtree, newray));
            }
            <Calculate a new color clr as
```

```
            explained in step 3>;
            return clr;
        }
        else if (<lightsource l is hit>)
            return l.color;
    }
}
```

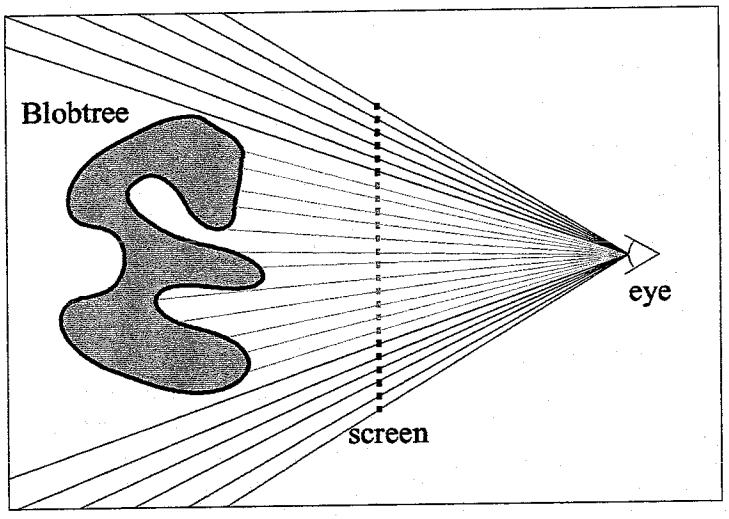

Figure 2.10: Casting rays through the pixels of the screen

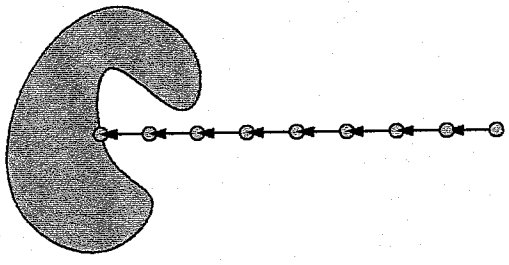

Figure 2.11: traversing the ray to detect intersections

2.4 Binary CSG operators for soft objects

Together with Loïc Barthe and Brian Wyvill, I worked on the paper 'Binary CSG operators for soft objects' [3]. When I started working on it, most of the paper was

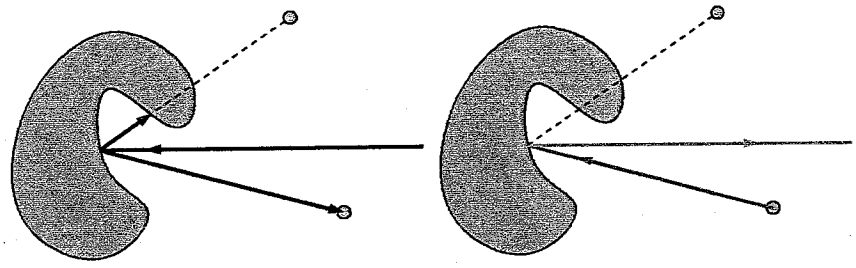

Figure 2.12: Casting new rays and calculate a return color

already written. My job was to implement the techniques discussed in the paper in Blobtree.NET and to make pictures using those techniques. Furthermore I corrected a bunch of spelling mistakes, rephrased sentences when needed and wrote an example section. The example section includes the pictures I created and shows how the techniques of the paper are used to create these pictures.

2.4.1 New Blobtree nodes

The paper 'Binary CSG operators for soft objects' describes 6 new nodes for the Blobtree. Basically the nodes are the binary CSG operations union, intersection and difference. For each of those operations there are two variants: a smooth and a sharp one. The smooth nodes apply the CSG operation with a smooth transition between the child nodes, whereas the sharp nodes apply the CSG operations with a sharp transition. Furthermore, the smooth nodes also have controlpoints to control the shape of the transition.

Usually CSG operation introduce discontinuities where the child nodes meet. The paper describes a technique to implement the previous nodes while keeping the number of discontinuities in the field function to a bare minimum.

2.4.2 Adapting Blobtree.NET

I extended Blobtree.NET software with several features.

New Blobtree nodes The nodes described in section 2.4.1 are added (see section 2.4.3).

A raytracer A raytracer is implemented to create high quality pictures (see section 2.3).

Visualization for cross sections In order to show what exactly is going on when using the new nodes, I implemented a simple algorithm to visualize a cross section of a Blobtree (see figures 2.13). The algorithm evaluated a Blobtree for a lot of points in the XY-plane and maps the field values to a color.

Texture mapping I added texture mapping functionality to several primitives and nodes of Blobtree.NET. To achieve this, I added a function to each node that returned

a 2D point given a point in space. The 2D point is used to look up a color in the texture map. The use of textures made the final pictures look a lot nicer.

2.4.3 Implementing the new nodes

The implementation of the nodes was not easy. Because of the amount of math, it was easy to make mistakes. It took a lot of time to create a good implementation.

The implementation pretty much reflects the math from the paper, except for some calculations in the sharp nodes. Some of the math for sharp nodes in the paper require huge calculations with complex numbers. In my implementation these calculations are replaced with a recursive approximation. In this case the approximation is a lot faster and even more precise (a 'precise' complex implementation would result in a bigger error due to a lot of calculations with limited floating point precision). An explanation of what the recursive approximation does, would be too complicated without including the original math of the paper.

2.4.4 The pictures

Figure 2.15 shows the final result created with the raytracer (section 2.3). It is a teddy bear constructed from simple primitives (mostly spheres) and the nodes described in the paper. Figures 2.14 and 2.13 show some parts of the model: an ear and an arm. The ear demonstrates a sharp node (The cross section shows no discontinuities except for the sharp edges) and the arm demonstrates a smooth node (The cross section shows no discontinuities at all).

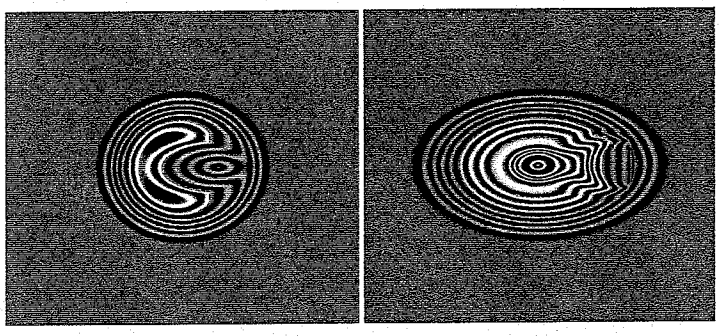

Figure 2.13: Cross sections of the parts of figure 2.14.

2.5 Computer Animation

One of the assignments for the course Computer Animation (CPSC 601.94, University of Calgary) is to create a computer animated version of the short movie 'Things that go beep'. To make things more interesting, the animation has to be made using several different techniques. This year the participating students choose to model the

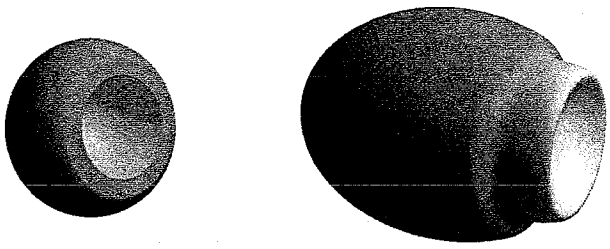

Figure 2.14: Parts of the final model. Left: one of the ears, right: one of the arms.

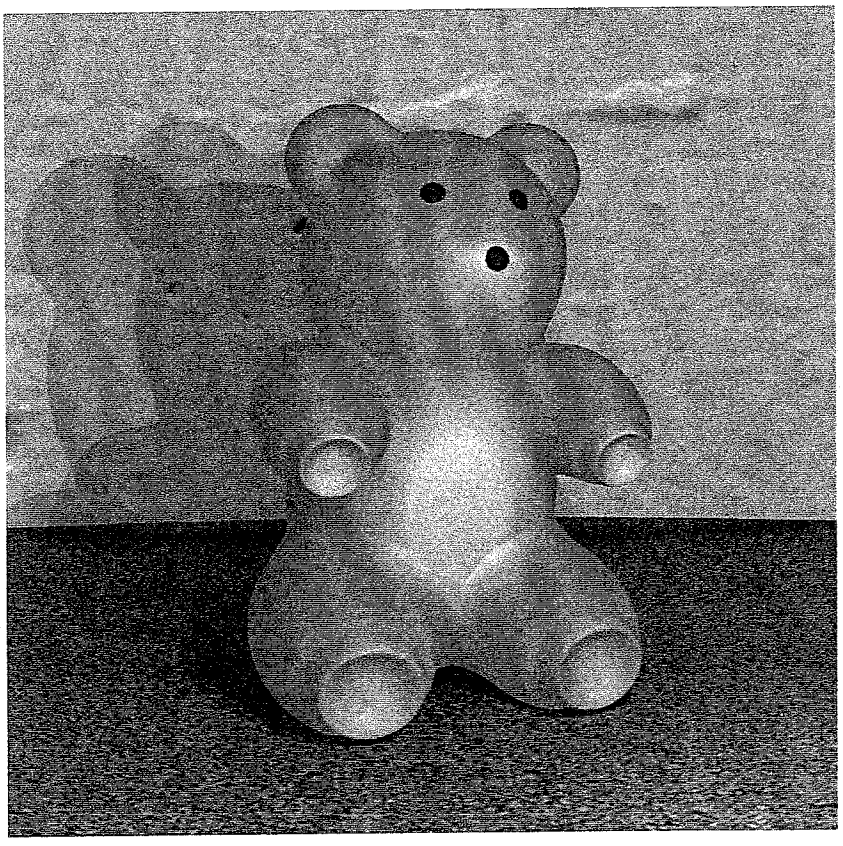

Figure 2.15: The final picture.

main character 'Blobbie' using Blobtree.NET. Because I was helping the students using Blobtree.NET, I also got involved in creating the animation.

2.5.1 Blobbie

The main character of the short movie is Blobbie. Blobbie is built with a Blobtree using sphere and cylinder primitives and taper, blend and union nodes. The modelling was done by creating an input file for Blobtree.NET using a simple text editor. The input file was visualized by quickly polygonizing it (section 2.2). In between the polygonizations, changes to the input file were made until the desired shape appeared.

I helped setting up the Blobtree.NET software on other computers, gave a number of examples for the input file and helped making a rough model of Blobbie (which was later on refined by others).

After creating a good model for Blobbie (see figure 2.16), there were two issues left to tackle: Blobbie had to move (see section 2.5.2) and Blobbie had to wear a t-shirt (see section 2.5.3).

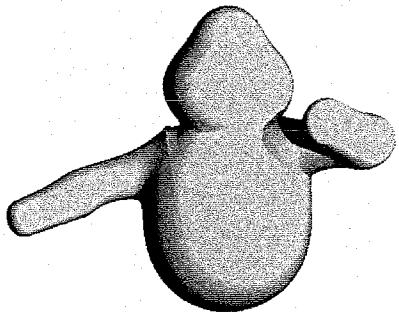

Figure 2.16: Blobbie

2.5.2 Trackmaker

Trackmaker [11] is the application developed to easily create movement for Blobbie. The basic idea is to create a series of Blobtree.NET input files that have a description of Blobbie for each frame in the animation. The only difference between the input files are some numbers, usually representing a position or a rotation (the separate parts of Blobbie do not change, they only move). Blobtree.Net polygonizes the resulting input files and saves them as an obj-file (which is used by several software packages, including Maya [2]).

Trackmaker starts with a semi Blobtree.NET input file where some numbers are not specified yet. Trackmaker calculates the values of the unspecified numbers for each frame of the animation and produces the resulting Blobtree.NET input files. The calculation of the values of the unspecified numbers is done as follows: the user can specify values for certain frames and Trackmaker will create the values for the other frames by interpolating between the specified ones.

The user can supply values for certain frames in an interactive way: a frame can be selected and while the values are being specified, the resulting shape of Blobbie will be shown instantly (figure 2.17). This all is being done using the Blobtree.NET functionality. Blobtree.NET is written in a way the functionality can also be used in other applications. Trackmaker simply had to include the Blobtree.NET library and it could use the Blobtree.NET polygonizer and a Blobtree.NET component to show the polygonization result on the screen.

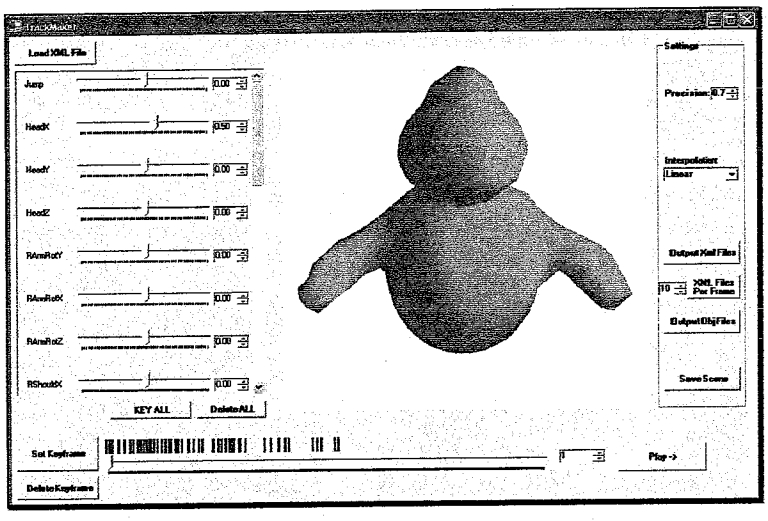

Figure 2.17: The Trackmaker user interface

I had to change Blobtree.NET in a way that certain components were also available for other applications. Furthermore I helped building the Blobtree.NET functionality into Trackmaker and provided some ideas on how to build Trackmaker itself.

2.5.3 Cloth

To create a t-shirt for Blobbie (figure 2.18), a piece of cloth is simulated using a mass spring system. In the simulation collisions are detected between the cloth and a Blobtree (Blobbie).

Kelly Poon created an application to simulate cloth movement [10]. I helped her porting the application to .NET and adding interaction with Blobtrees using Blobtree.NET. The ported application reads a series of input files that define the volume

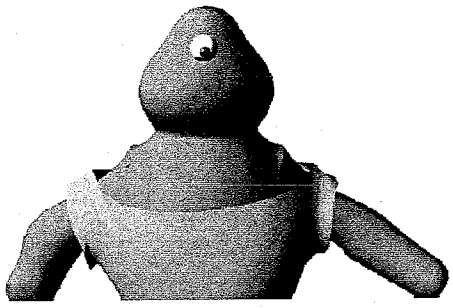

Figure 2.18: Blobbie with a t-shirt

of Blobbie in each frame. For each frame a number of simulation steps are calculated to make the cloth move. For example, when Blobbie moves up an arm, the sleeve of the T-shirt moves accordingly due to collisions with the arm. For each frame the current state of the cloth is saved as an obj-file (figure 2.19).

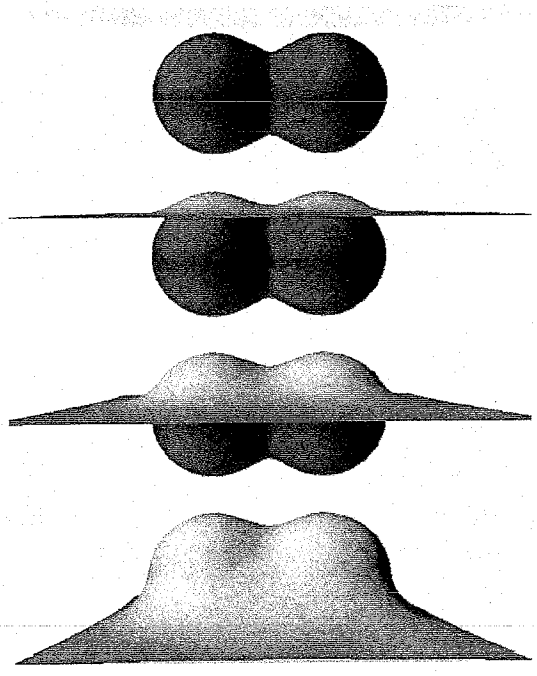

Figure 2.19: Different states of a piece of cloth falling on a Blobtree.

2.5.4 The movie

The separate parts of the movie (Blobbie, the t-shirt, the scene and other stuff) were merged using Maya [2]. Figure 2.20 shows a few frames from the movie.

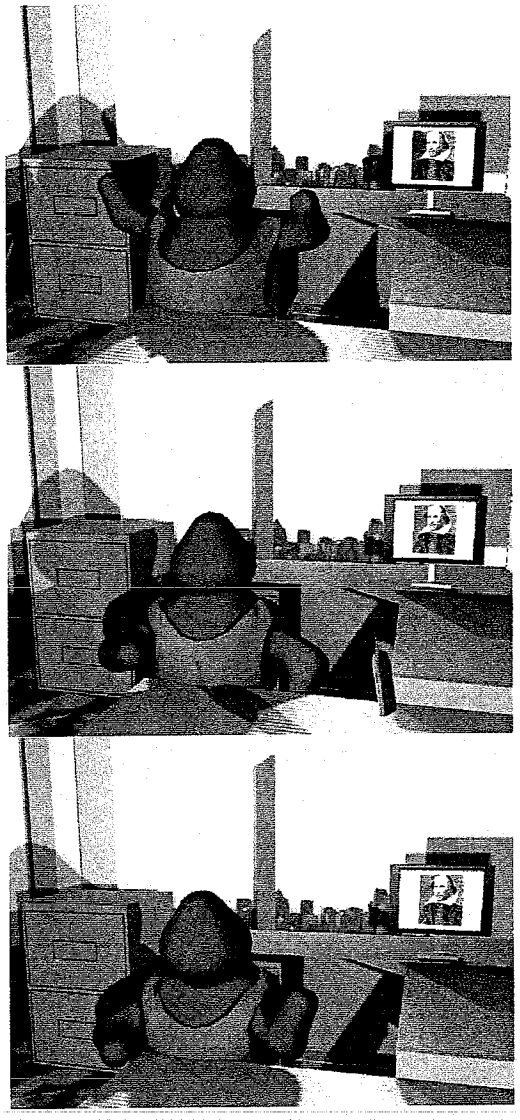

Figure 2.20: A few frames from the animated movie 'Things that go beep'.

Chapter 3

Better blending between multiple nodes of the Blobtree

3.1 abstract

A Blobtree is an implicit surface defined by a tree of objects. Each node of the tree can either be a primitive or an operation. Primitives are skeletal implicit surfaces defined by a skeleton (like a point or a line) and a potential function which maps the distance to the primitive to $\mathbb{R}^+$. Operations (like union, twist or blend) are nodes that act on one or more other nodes.

Skeletal implicit surfaces are volume representations particularly useful for models which require smooth blending. In some cases problems arise when blending is used. One of those problems occurs when the nodes involved in the blend have large differences in their blending ranges. In this case the influence of the relatively small node is negligible compared to the larger ones. Because of this, in certain places the result will look more like a union then a blend.

The method presented in this paper deals with the blend problem, even in the case where multiple nodes (either primitives or operations) are used. The idea behind this method is to locally deform the blending ranges of each node in order to make the blending conditions between each pair of nodes more suitable for smooth blending.

3.2 Introduction

Implicit surfaces are volume representations where the surface of the volume is defined by an iso-value of a real function. There are several kinds of implicit surfaces and they can generally be divided into two groups: bounded implicit surfaces (Metaballs, Soft Objects [14]) and unbounded implicit surfaces (R-functions [9]). A bounded implicit surface returns a constant value outside a certain boundary and the values of an unbounded implicit surface vary over the whole space.

The blending method presented in this paper (referred to as better blend) relies on

the local properties of each of the volumes to be blended. For this reason a bounded representation of the volumes is chosen: Blobtrees. The method can also be applied on unbounded representations, but this may require different calculations.

A Blobtree [13] is a tree structure built from primitives and operations. The primitives (the leafs of the tree) are skeletal implicit surfaces [6] and the other nodes are operations that work on one or more primitives or nodes.

A primitive is defined by a non-zero iso-value c of a field function. The field function f is composed of a potential function $p(d)$ and a distance function, where the distance function provides the distance d to a skeletal element (like a point or a line) and the potential function maps the distance to $\mathbb{R}^+$.

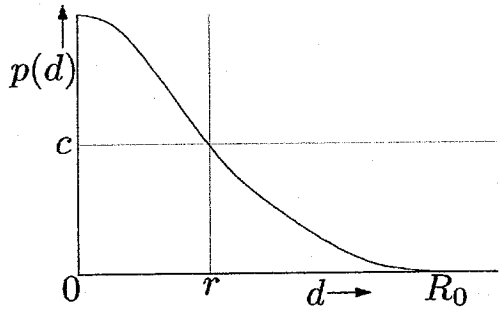

Figure 3.1: Example potential function

Figure 3.1 shows a bounded potential function. When the distance from a point to the skeleton is less than or equal to r, the point is inside the volume of the primitive. The range between r and R_0 is called the blending range. The blending range isn't part of the volume of the primitive, but has influence on certain operations like blending.

Operation nodes can be any operation that works on one or more other nodes. Possible operations are union, intersection, twist and blend [13]. In this paper we will take a closer look at the blend operation.

The easiest way to blend the nodes of a collection I is to add up their field values (a field value is the value of the field function in a given point). Nodes will blend when they are close enough to each other, so that the sum of the field values in between the nodes exceed the iso-value. The field value of the blend f_b in a point q will be calculated as follows:

$$f_b(q) = \sum_{i \in I} f_i(q)$$

As pointed out in [15], a problem occurs when the nodes in a blend have large size differences. The influence of a relatively small node is negligible compared to the influence of a large node. This results in a blend that looks like a union. Another side effect is that the size of the small node increases because of the large influence of the large node (figure 3.2).

In this paper we will present a solution which will work for an arbitrary number of primitives or nodes. In this new solution the blending ranges of the nodes will be locally

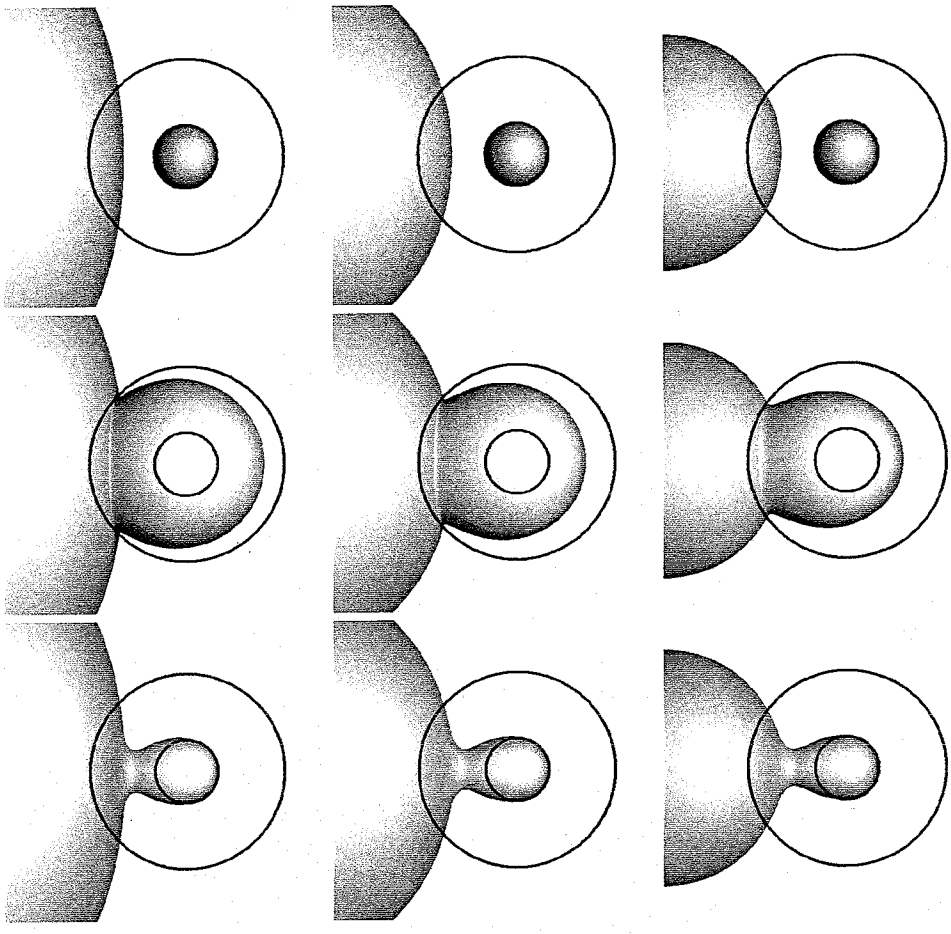

Figure 3.2: The blending problem. From top to bottom: union, regular blend, better blend. Ratio of radii of large to small point primitives are (from left to right) 16, 8, 4. The black circles indicate the shape and the blending range of the small primitive.

deformed. The amount of deformation for a node will depend on the influence of other nodes. The user can define weights between each pair of nodes, which can influence the amount of deformation. With this method the user can control the blending between each separate pair of nodes without interfering with the blending between other nodes.

3.3 Deforming the blending range

One way of solving the blending problem is to change the blending range of a primitive by changing the potential function [15]. The problem with this approach is that this

only works with primitives and that the primitives themselves have to be changed. This approach does not work for operation nodes, because these nodes do not have a potential function. The field function of an operation node can therefore only be changed by changing the potential functions of its underlying primitives, which in most cases is hard to control and not always possible without changing the shape of the node itself.

Changing primitives in order to change the blending range can be avoided by using deformation. This can be done by mapping the field values of a node with $g : \mathbb{R}^+ \rightarrow \mathbb{R}^+$. The new field function f' will now be calculated as follows: $f'(x) = g(f(x))$.

The mapping g has to meet the following requirements:

1. the range of g has to be in $\mathbb{R}^+$, because mapping g has to produce valid field values

2. $g(0) = 0$ in order not to create an influence everywhere outside the range of the node

3. $g(c) = c$ to preserve the shape of the node

4. g must have a parameter to control the amount of deformation

5. g and its derivative have to be continuous to preserve continuity in the field values

First we construct g in the domain $[0, c]$. To satisfy requirements 2 and 4, a parameter $k \in [0, c)$ is introduced. This parameter indicates where the new blending range should start compared to the original one. Field values smaller than k will be mapped to 0, because they won't be within the new blending range anymore. To satisfy requirement 5, $g_k(k) = 0$ and $\dot{g}_k(k) = 0$. A solution satisfying these requirements is the square polynomial $c(\frac{t-k}{c-k})^2$. Other solutions are also possible, but the square polynomial is easy to implement and can be calculated fast.

For the range (c, ∞) the same polynomial is used as in the range $[0, c]$, but now reflected over the point (c, c). This ensures continuity in (c, c).

We construct the following mapping, which meets all of the requirements (see also figure 3.3):

$$g_k(t) = \begin{cases} 0 & \text{if } t \leq k \\ h_k(t) & \text{if } k < t \leq c \\ 2c - h_k(2c - t) & \text{if } c < t < 2c - k \\ 2c & \text{if } t \geq 2c - k \end{cases}$$

$$\text{where } h_k(t) = c(\frac{t-k}{c-k})^2$$

Effects of different values for k are shown in figure 3.5.

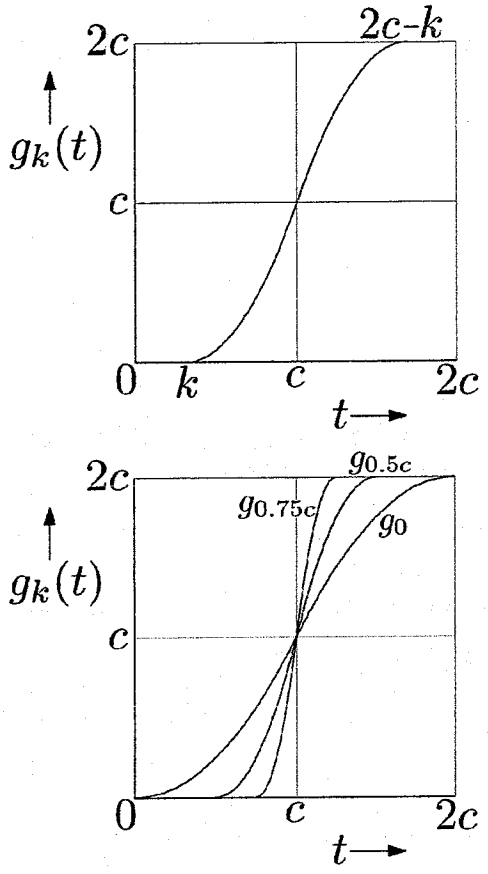

Figure 3.3: Top: Plot of g_k. Bottom: g_0, $g_{0.5c}$ and $g_{0.75c}$.

3.4 Assigning weights to influence blending

Now we have control over the blending ranges of the nodes, we can use this to influence the blending itself. The deformations can be controlled by assigning weights to the nodes, where each weight indicates the amount of influence in the blend. This can be done in several ways. We will present three methods that solve the blending problem discussed in [15].

3.4.1 Solution 1: Deforming nodes separately

The easiest way to use deformations to solve the blending problem is to assign weights $w_i \in (0,1]$ to each of the nodes i in the collection of nodes I. A lower value for w_i means more deformation and less influence in the blend. The nodes will be deformed

Figure 3.4: Color reference for cross sections.

using mapping $g_{c(1-w_i)}$ and then blended together using a summation. The field value of the blend in a point q can be calculated with:

$$f_b(q) = \sum_{i \in I} g_{c(1-w_i)} f_i(q)$$

Figure 3.6 shows the case where four spheres with different sizes are blended. Before blending, the blending ranges of each sphere are deformed.

In many cases this approach may give reasonable results, but it is not possible to control the blending between each pair of primitives and nodes separately.

3.4.2 Solution 2: Blending all possible pairs of nodes separately

To solve the problem of the previous solution, this time each pair of nodes is blended separately. Each pair of nodes (i, j) with i, j in I and $i \neq j$ has its own weights $w_{i,j}$ and $w_{j,i}$ to blend the nodes together. The field value of the blend can be calculated by adding up the intermediate results:

$$f_b(q) = \frac{\sum_{i,j \in I \wedge i \neq j} \left(g_{c(1-w_i)} f_i(q) + g_{c(1-w_j)} f_j(q) \right)}{|I| - 1}$$

Although it is possible to control the blending between each pair of nodes, it is not possible to control the final blend of the intermediate results. Figure 3.7 shows the relevant intermediate results of the blend of figure 3.8. Although the blends between each pair of nodes are controllable, the final blend is not. Therefore the final blend is not related to the intermediate results. In this case the final result shows unwanted bulging at the blending places (e.g. the blend between the two largest spheres in figure 3.7 is smaller than in figure 3.8). This problem is especially visible in the cases where the intermediate results have large size differences themselves. Using a union instead of a blend in the final step is not an option since the union may introduce discontinuities.

3.4.3 Solution 3: Locally deforming primitives and nodes

To solve the problem of section 3.4.2, this time the weights are used to locally apply deformations. Each of the primitives and nodes will be deformed locally depending on the weights and the field functions of other primitives and nodes (figures 3.9 and 3.10). This approach will be explained in more detail in the next section.

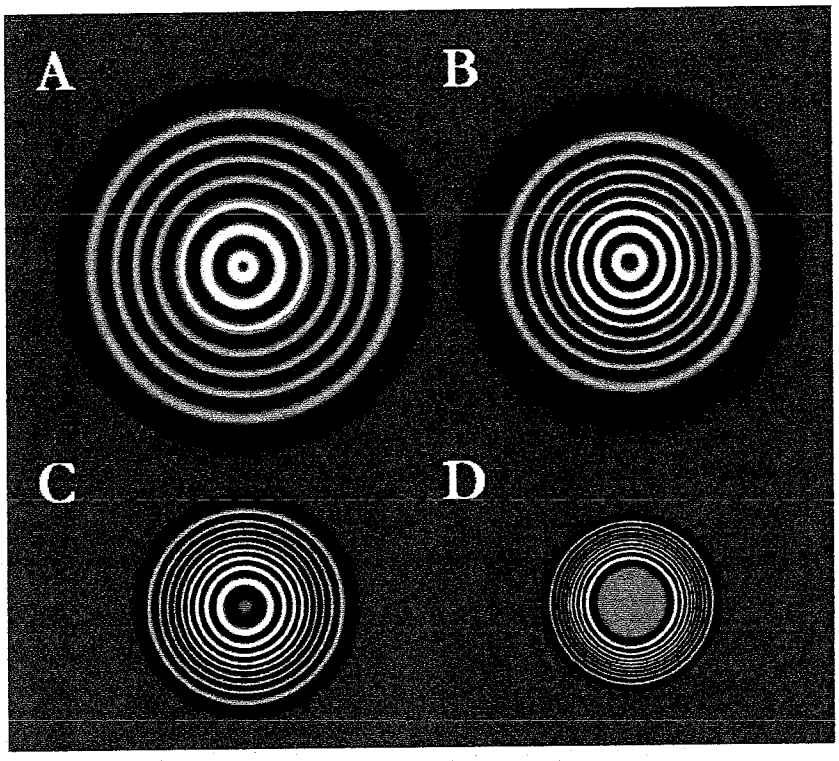

Figure 3.5: Cross sections of a point primitive. A: the original primitive. B, C and D: deformed primitives using g_k with respectively $k = 0$, $k = 0.5c$ and $k = 0.75c$.

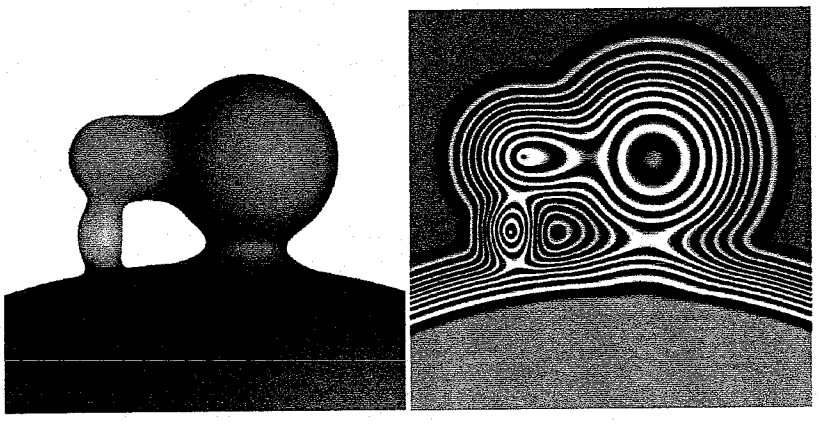

Figure 3.6: Blending after deforming each node

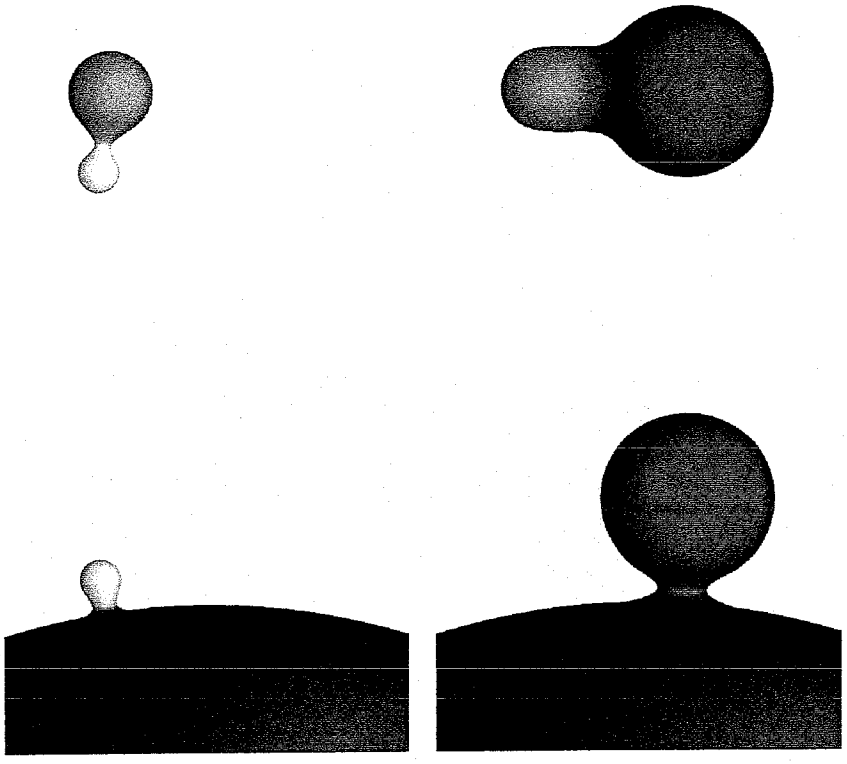

Figure 3.7: Relevant intermediate blends

3.5 Better blending

A better approach to influence the blending between each pair of nodes is to locally deform each of the nodes based on the proximity of other nodes and their weights. When a node has no influence in a certain point, then this node won't have any influence on the deformation of other nodes in that point. Furthermore, higher values for weights result in less deformation and thus more influence in the blend.

The amount of deformation of a node i in a point q depends on the field values and weights of the other nodes. The amount of deformation $k_i(q)$ of node i in a point q is calculated as follows:

$$k_i(q) = c(1 - w_{i,i} \prod_{j \in I \setminus \{i\}} d_{i,j}(q))$$

Here $w_{i,i}$ controls the global deformation of the blending range of node i and $d_{i,j}(q)$ is the contribution of node j to the local deformation of node i in point q. When

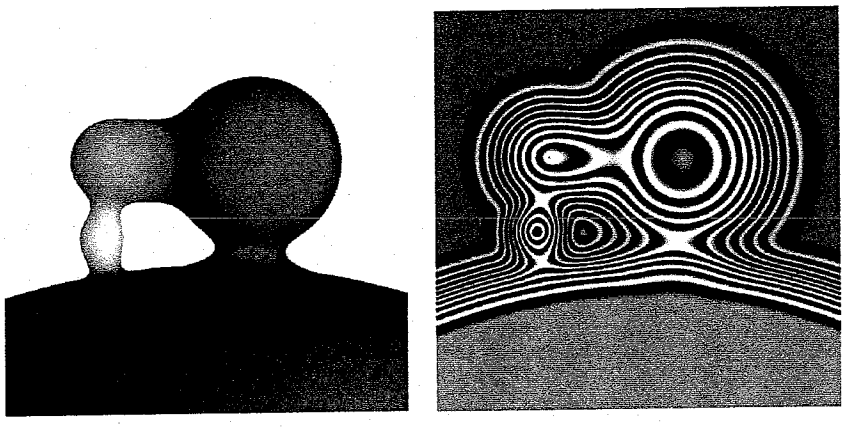

Figure 3.8: Final blend of the intermediate blends

$d_{i,j}(q) = 1$ node j does not have any influence on the deformation of i and the influence grows when $d_{i,j}(q)$ gets smaller. $d_{i,j}(q)$ is calculated as follows:

$$d_{i,j}(q) = \begin{cases} 1 + (1 - w_{i,j}) \frac{f_j(q)}{c} (\frac{f_j(q)}{c} - 2) & \text{if } f_j(q) < c \\ w_{i,j} & \text{if } f_j(q) \geq c \end{cases}$$

Figure 3.11 shows how $d_{i,j}(q)$ decreases as $f_j(q)$ increases. To make the values of the weights as intuitive as possible, the value of a weight $w_{i,j}$ indicates the contribution of j to the deformation i in the points q where $f_j(q) = c$. To avoid negative values for $d_{i,j}(q)$, we set $d_{i,j}(q) = w_{i,j}$ inside node j.

As with g_k, many definitions for $d_{i,j}$ are possible. We choose this definition because it is simple.

Now we know the amount of deformation for each node i in a point q, we can apply the deformations and blend the whole thing together:

$$f_b(q) = \sum_{i \in I} g_{k_i(q)}(q)$$

3.6 Examples

Figures 3.12 through 3.15 show a vase built from just a few primitives and a blend operation. Figure 3.12 shows the separate deformed nodes and a cross section of the final result. Figures 3.13 and 3.14 show the difference between using a regular blend and the better blend. The regular blend clearly shows the blending problem. Figure 3.15 shows the vase within a scene of implicit objects.

Figures 3.16 and 3.17 show the difference between using a regular blend and the better blend for the plane model. The difference is not as extreme as in the first example, but better blending is necessary to model the details. Figure 3.18 shows the plane within a scene of implicit objects.

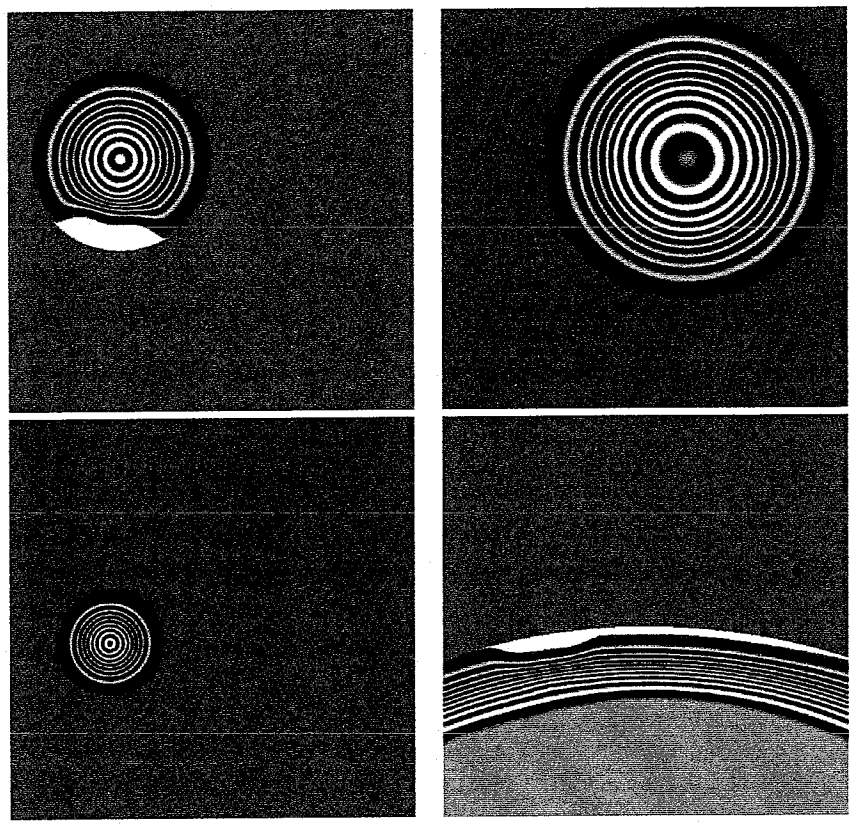

Figure 3.9: Cross section of each deformed node. The plain white area shows the blending range before applying local deformation (top-left and bottom-right picture).

3.7 Adapting Blobtree.NET

After working on the paper 'Binary CSG operators for soft objects' (section 2.4), Blobtree.NET was already quite capable of producing nice pictures. Still I added a few new features to make the result even nicer.

New blending node Of course I had to implement the new blending node as described in section 3.5.

Extensions for visualizing cross sections In order to show some of the features of the new node, I had to add some features in visualizing cross sections (like marking the difference between two cross sections; see figure 3.9).

Reflection I implemented reflection as a property of a material. Furthermore I extended the raytracer so that it would support reflections. Figure 3.18 shows a picture that is created using reflection.

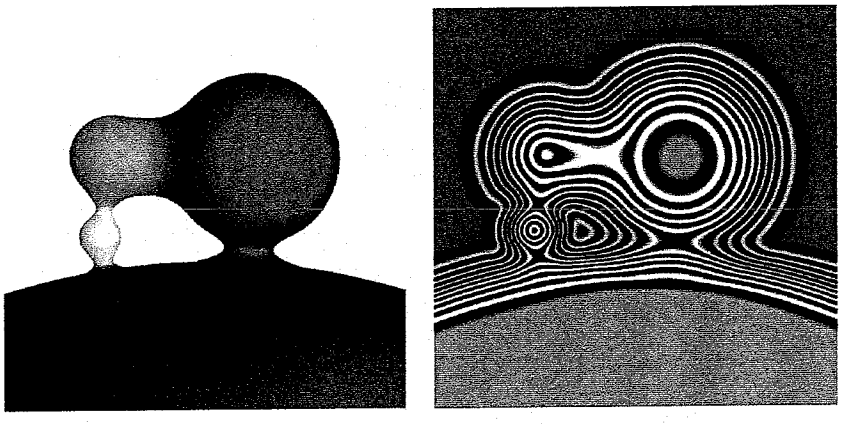

Figure 3.10: Final blend after deforming the nodes

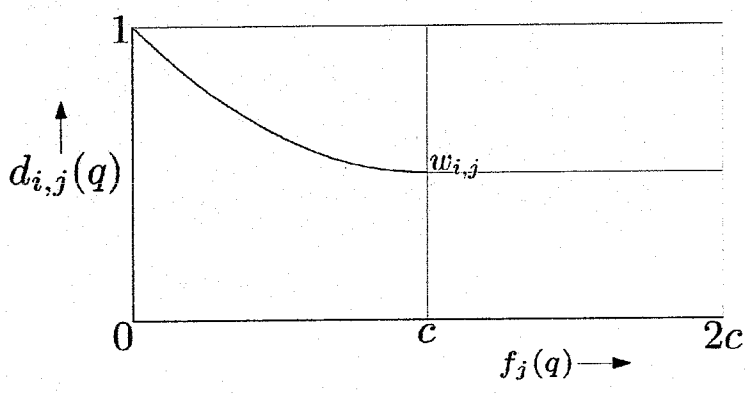

Figure 3.11: The contribution $d_{i,j}(q)$ of node j to the deformation of node i in point q.

3.8 Implementing the new node

When I first implemented my solution for the blending problem, the results were just reasonable. It was still hard to control the amount of bulging, so the solution still needed to be refined and the math had to be optimized. Every time after changing the implementation of the node, the results were visualized using the polygonizer (section 2.2). Based on these results, I adapted the current implementation to make the results better. After several changes, the results were satisfying and the new blending node was ready to be used for creating the final pictures (figures 3.15 and 3.18).

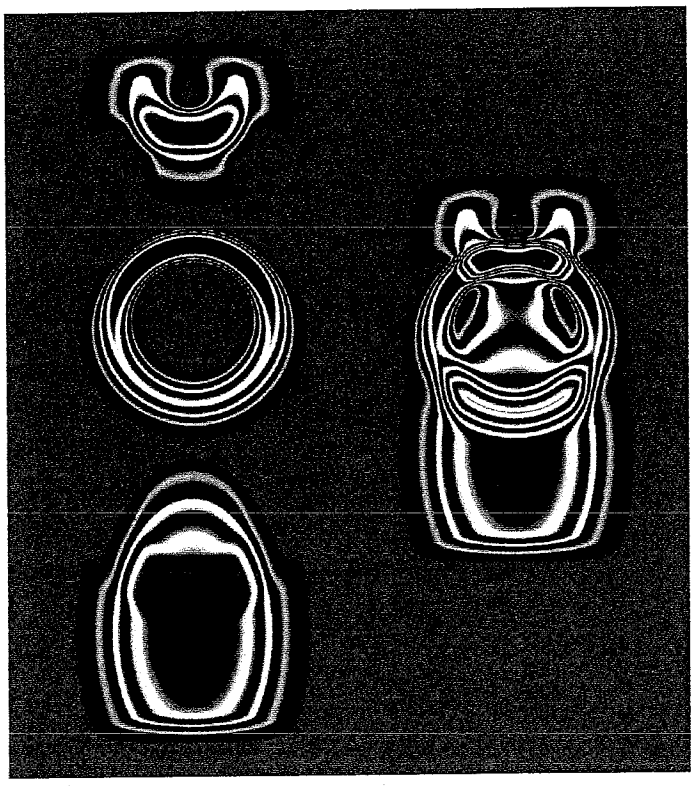

Figure 3.12: Cross sections of the separate deformed nodes and the final blend.

3.9 Conclusion

The method presented in this paper is a solution for the blending problem which occurs when nodes have large size differences. Not only solves it the problem in the cases with more then two primitives, but it also works with operation nodes.

The method greatly increases control over the blending by the use of weights, which control the blending between each separate pair of primitives and nodes.

The evaluation time of this method is comparable to the evaluation time of a regular blend, because it only evaluates each of the primitives and nodes once. The difference in evaluation time will mainly be caused by the calculation of the amount of deformation and the use of the mapping g_k, but this difference will in most cases be negligible compared to the evaluations of the nodes.

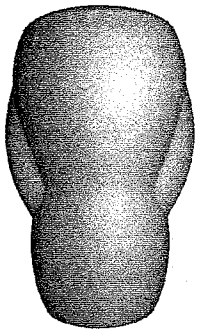

Figure 3.13: The vase using regular blending.

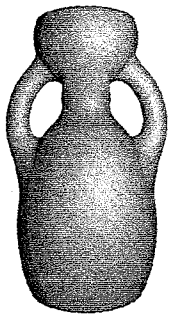

Figure 3.14: The vase using better blending.

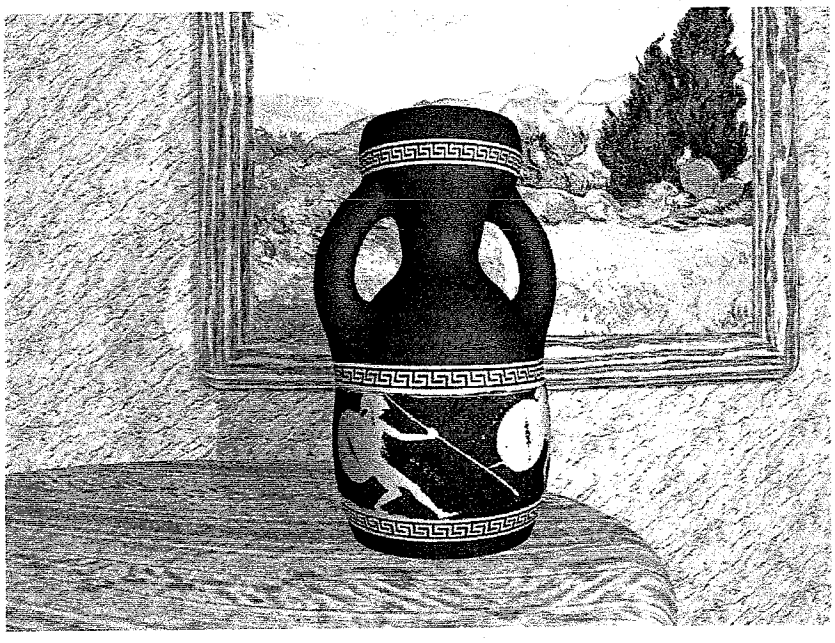

Figure 3.15: The final result within a scene.

3.10 Future work

Future work may include finding a method to automatically generate weight values which result in 'good' blends. This also includes defining what a 'good' blend exactly

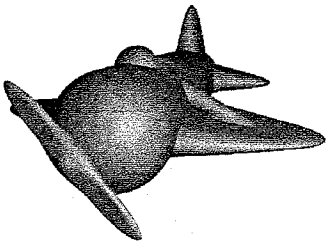

Figure 3.16: The plane using regular blending.

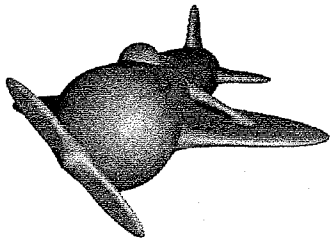

Figure 3.17: The plane using better blending.

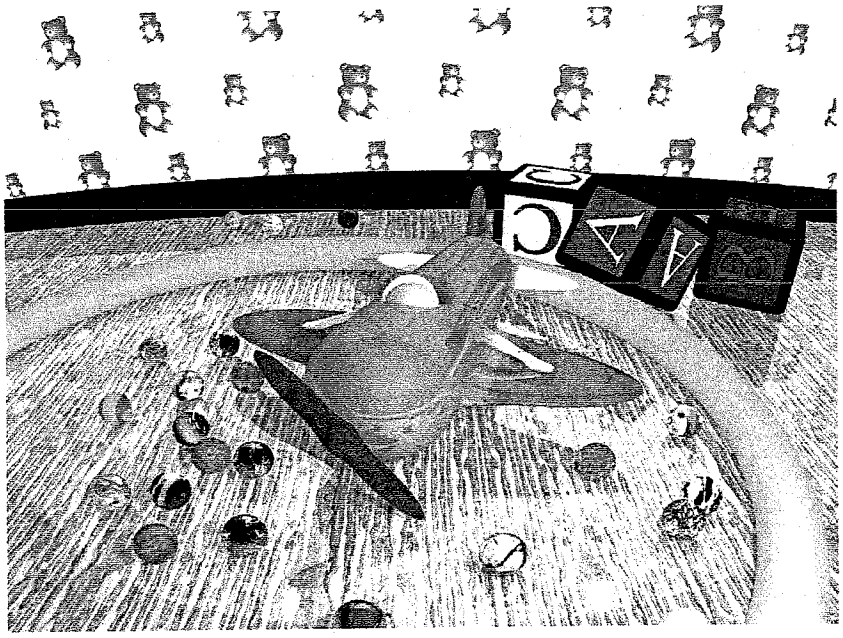

Figure 3.18: The final result within a scene.

is. This may result in different ways to generate weight values depending on what the user defines as a 'good' blend.

Although the formulas in the calculation give reasonable results, there is still a lot of room for improvement. A big improvement would be to replace the mapping g_i

by a mapping which doesn't change the field value when $t = 0$. Such a new mapping would result in a blend which preserves the original nodes in places where no blending occurs. This way, the resulting blend keeps the proximity blending property, even after applying blending several times.

Chapter 4

Conclusion

My assignment was to explore different aspects of implicit surfaces, with the Blobtree in particular. With the creation of Blobtree.NET I learned a lot on implicit surfaces and Blobtrees (see chapters 1 and 2). Furthermore Blobtree.NET made it possible to explore more specialized aspects of implicit surfaces and Blobtrees, which resulted in a contribution to a paper (see section 2.4) and a paper of my own (see chapter 3).

The Blobtree.NET software turned out to be very useful for research purposes. I was able to do all the research for this thesis without the use of other software packages. Furthermore it is easy to use Blobtree.NET in other projects. For the project of the animation class (see section 2.5) the Blobtree.NET software was used instead of the traditional JSP [1] software, because Blobtree.NET was far more easy to use.

In the paper "Better blending between multiple nodes of the Blobtree" I showed a solution to the well known "blending problem". With this paper I showed that my knowledge about implicit surfaces is more than sufficient and that I am able to solve problems myself.

Before I came to the University of Calgary I did not know much about implicit surfaces or Blobtrees, but now I consider myself specialized in this subject. Now I finished my thesis, I'm ready to do research in more specialized and unexplored areas of implicit surfaces.

Chapter 5

Future work

Of course still a lot of research can be done on implicit surfaces. For me, Blobtree.NET would be a really helpful tool to help me do that research. And although Blobtree.NET is easy to use by other people (see section 2.5), it is still hard for others to change or expand Blobtree.NET. In the future Blobtree.NET could be rewritten in a way that it is more accessible for other researchers, so that they also could use Blobtree.NET for their research.

Implicit surfaces greatly improves the simplicity with which a model can be created. The creation of models would be even better if there was a interactive editor. Although Blobtree.NET supports the functionality behind such an editor, the editor itself hasn't been created yet.

At this moment Blobtree.NET does not support any animation features (although section 2.5 shows that there are ways to work around that). In the future better support for animation could be integrated into Blobtree.NET.

Implicit surfaces will always be time consuming compared to polygonal volume representations. The polygonizer and the raytracer of Blotree.NET could be sped up a lot, possibly by using graphics accelerating hardware.

The methods described in the paper "Better blending between multiple nodes of the Blobtree" can still be approved. Also other methods to blend nodes should be considered, since different blending situations could need different solutions.

Chapter 6

Acknowledgements

I want to thank the University of Calgary for giving me a place to do my research. Furthermore I want to thank my supervisors Huub van de Wetering and Brian Wyvill for all their help and support.

Bibliography

[1] JSP. *http://pages.cpsc.ucalgary.ca/ jungle/software/JungleSP/index.html*

[2] Alias. Maya. *http://www.alias.com/eng/products-services/maya/*

[3] L. Barthe, B. Wyvill, and E. de Groot. Binary csg operators for 'soft objects'.

[4] Microsoft. C♯. *http://msdn.microsoft.com/vcsharp/*

[5] Microsoft. .NET. *http://www.microsoft.com/net/*

[6] J. Bloomenthal. *Introduction to Implicit Surfaces*. Morgan Kaufmann, ISBN 1-55860-233-X. Edited by Jules Bloomenthal With Chandrajit Bajaj, Jim Blinn, Marie-Paule Cani-Gascuel, Alyn Rockwood, Brian Wyvill, and Geoff Wyvill.

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

[8] J. Bloomenthal. An Implicit Surface Polygonizer. *Graphics Gems IV*, pages 324–349, 1994. Edited by paul Heckbert.

[9] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, 1995.

[10] K. Poon. Cloth animation. 2003.

[11] J. Tayler-Hell. Trackmaker: Animating Blobtree.NET. 2003.

[12] K. van Overveld and B. Wyvill. Shrinkwrap: An efficient adaptive algorithm for triangulating an iso-surface . *The Visual Computer (In Press)*, 2002.

[13] B. Wyvill, E. Galin, and A. Guy. The blob tree, warping, blending and boolean operations in an implicit surface modeling system. *Implicit Surfaces*, 3, June 1998. Chosen for inclusion in a special issue of Computer Graphics Forum.

[14] B. Wyvill, C. McPheeters, and G. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

[15] B. Wyvill and G. Wyvill. Better blending of implicit objects at different scales. *ACM Siggraph 2000 presentation*, 2000.