FACULTY of MATHEMATICS, PHYSICS and INFORMATICS
COMENIUS UNIVERSITY
BRATISLAVA

MASTER THESIS

Bratislava, 2003. april, Gabriel Farkas

FACULTY of MATHEMATICS, PHYSICS and INFORMATICS
COMENIUS UNIVERSITY
BRATISLAVA
DEPARTMENT of GEOMETRY

BLENDING of CANAL SURFACES

Author: Gabriel Farkas
Consultant: RNDr. Pavel Chalmovianský, PhD.

I honestly allege that i have developed the master thesis independently only with the use of the literature listed in the bibliography

.....
Gabriel Farkas

I would like to thank to my diploma thesis consultant Pavel Chalmovianský for his valuable advices, remarks and suggestions.

Contents

1 Introduction **3**

2 Preliminaries/Basic notions **4**
2.1 Boundary representation (b-rep) 4
2.1.1 Mesh 4
2.1.2 Representation of the mesh 6
2.2 Functional representations 10
2.2.1 Implicit surfaces 10
2.3 Bézier curves 13
2.3.1 Bernstein polynomials 13
2.3.2 Bézier segments 14

3 The blending algorithm **15**
3.1 An intuitive overview 15
3.2 Basic definitions 15
3.3 Input-output characterization 17
3.3.1 Input 17
3.3.2 Output 18
3.4 Blending between two selections 18
3.4.1 calculate the rings from the selections 19
3.4.2 remove the selections 19
3.4.3 generate the worms 19
3.4.4 determine a suitable form for the worm 23
3.5 Blending between three or more selections 30
3.5.1 Creating the virtual sphere 30
3.5.2 Mapping a ring to a sphere 30
3.5.3 Connecting the mapped rings with the original rings . 32
3.5.4 Eroding the sphere mesh 32

4 Summary and Future Work **35**

5 Results **37**

Chapter 1

Introduction

Computer graphics offers many approaches to represent the reality or the unreal if needed: boundarily represented objects, functional representations, fractals, particle systems just to name a few. Everyone of these approaches (techniques) offer different advantages and disadvantages, some are excellent to represent organic materials for example but lack the ability to construct technical object from them. Other representations are easily used to create strictly defined clearly shaped objects, but are unable to capture the characteristics of the more chaotic systems.

That is the reason why the users of computer graphics usually select the approach based on the problem to solve. In our work, we will try to break the barrier between some of those representations and will try to show that with a little help from the computer and the user representations can have properties that usually belong to other ones.

The implicit surfaces are well known for their ease to use to create smooth blendings. This effect comes naturally because of their very definition. Our goal is to mimic this behaviour with boundarily represented objects (meshes to be more precise).

But curiousity or lust to adventures are not the only reasons why we are planning to explore this possibility. Many graphic designers prefer to work with meshes, because of their easier to understand properties (it is made of triangles after all). We hope that with our work even they will be able to experiment with features usually reserved to implicit surfaces.

Our work is organized in the following manner: first in Chapter 2 we introduce the mathematical background needed for our work, then in Chapter 3 we show our approach to blend triangular surfaces. In Chapter 4 we summarize our work and list ideas on possible future improvements, and in Chapter 5 we show some characteristic results of our algorithm.

Chapter 2

Preliminaries/Basic notions

2.1 Boundary representation (b-rep)

One of the most used object representations deals with the description of the boundary of the object, because:

- it is the most natural representation (people usually draw the boundary of the objects)
- the inner part can be calculated from the boundary. ??? HOW ???

As always, we begin with defining our basic working structure, the mesh [1].

2.1.1 Mesh

Let $\mathcal{A}$ denote a set. The set of all possible selections of n elements created from the elements of the set $\mathcal{A}$ without repetition will be denoted $\mathcal{A}^n$. By $(a_0, \ldots, a_{n-1})$ we denote the elements of $\mathcal{A}$, called **ordered n − tuples**. Let $\mathcal{A}^{[n]}$ be the set of all *cyclically* and *reverse unordered* $n - tuples$. The elements of $\mathcal{A}^{[n]}$ are denoted $[a_0, \ldots, a_{n-1}]$, where $a_i \in \mathcal{A}, i \in Z_i$. The set $\mathcal{A}^{[n]}$ is the set $\mathcal{A}^n$ factored by relation of equivalence $\simeq$ generated by relations:

$$(a_0, \ldots, a_{n-2}, a_{n-1}) \simeq (a_0, a_{n-1}, \ldots, a_{n-2})$$

$$(a_0, a_1, \ldots, a_{n-2}, a_{n-1}) \simeq (a_{n-1}, a_{n-2}, \ldots, a_1, a_0) \text{for all } (a_0, \ldots, a_{n-1}$$

According to this $\mathcal{A}^{[n]} = \mathcal{A}^n/_{\simeq}$.

We say that the element $[b_0, \ldots, b_{k-1}]$ is a **sub − element** of $[a_0, \ldots, a_{n-1}], a_i \in \mathcal{A}, k \leq n$ and denote $[b_0, \ldots, b_{k-1}] \subset [a_0, \ldots, a_{n-1}]$, iff there exist elements $c_0, \ldots, c_{n-k-1} \in \mathcal{A}$ such that $(a_0, \ldots, a_{n-1}) \simeq (b_0, \ldots, b_{k-1}, c_0, \ldots, c_{n-k-1}$. We say that $[b_0, \ldots, b_{k-1}]$ is **incident** with $[a_0, \ldots, a_{n-1}]$ and vice versa.

Definition 2.1.1 *An **abstract 2-dimensional cell complex** $\mathcal{C} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ is a triple of sets:*

$\mathcal{V} \neq \emptyset$ *is a countable set of vertices - called also the set of 0-faces,*

$\mathcal{E} \subset \mathcal{V}^{[2]}$ *is a set of edges - called also the set of 1-faces,*

$\mathcal{F} \subset \bigcup_{i=3}^{\infty} \mathcal{V}^{[i]}$ *is a set of faces, or 2-faces,*

satisfying the following conditions:

1. *if $u \in \mathcal{V}$, there exists $v \in \mathcal{V}, u \neq v$ such that $[u, v] \in \mathcal{E}$ (that is, there are no isolated vertices),*
2. *if $[v_0, v_1, \ldots, v_{k-1}] \in \mathcal{F}$, then $[v_0, v_1], [v_1, v_2], \ldots, [v_{n-2}, v_{n-1}], [v_{n-1}, v_0] \in \mathcal{E}$ ($\mathcal{E}$ must contain all edges of a face)*
3. *$[u, v] \in \mathcal{E}$, then there is a face $F \in \mathcal{F}$ such that $[u, v] \subset F$ (all edges are contained in a face),*
4. *if $[u, v] \in \mathcal{E}$, there are no more than two faces $F_1, F_2 \in \mathcal{F}$ such that $[u, v] \subset F_1$ and $[u, v] \subset F_2$ (maximally 2 faces are sharing an edge),*
5. *if $v \in \mathcal{V}$ and the set $Loop(v) = \{u \in \mathcal{V} \mid \exists F \in \mathcal{F} \ \wedge \ v \in F \ \wedge \ u \in F \ \wedge \ u \neq v\}$ is finite with k elements $u_0, \ldots, u_{k-1}$, then there exists an ordering of elements $(u_0, \ldots, u_{k-1})$ of Loop(v) so that $[u_i, u_{i+1}]$ is edge of $F \in \mathcal{F}$ with $v \in F \ \forall i = 0, \ldots, k-2$.*

Definition 2.1.2 *A **Mesh M(V,E,F)** is a special case of cell complex (see Definition 2.1.1)- the vertex set is made up from points in R^3. An edge e is called a boundary edge of M(V,E,F) if it is not shared by two faces. A vertex v is called boundary vertex if it belongs to a boundary edge. The mesh is said to be closed if it has no boundary edges. Otherwise, M(V,E,F) is an open mesh.*

We list several functions which will be used in the following text:

vnv(v), $v \in V$ the neighboring vertices of a vertex:

$$\text{vnv}(v) = \{w \in V | [w, v] \in E\} \tag{2.1}$$

vnf(v),$v \in V$ the neighboring faces of a vertex:

$$\text{vnf}(v) = \{f \in F | \exists w_{i_1}, \ldots, w_{i_n} \in V : [v, w_{i_1}, \ldots, w_{i_n}] = f\} \tag{2.2}$$

vne(v),$v \in V$ the neighboring edges of a vertex:

$$\text{vne}(v) = \{e \in E | \exists w \in V : [w, v] = e\} \tag{2.3}$$

env(e),$e \in E$ the vertices of an edge:

$$\text{env}(e) = \{v, w\}, \text{where } \{v, w\} \subset V \wedge [v, w] = e \tag{2.4}$$

ene(e),$e \in E$ the neighboring edges of an edge:

$$\text{ene}(e) = \{e' \in E | \exists w_1, w_2, w_3 \in V : [w_1, w_2] = e \wedge [w_2, w_3] = e'\} \tag{2.5}$$

enf(e),$e \in E$ the neighboring faces of an edge:

$$\text{enf}(e) = \text{vnf}(v) \cap \text{vnf}(w), \text{ where } [v, w] = e \tag{2.6}$$

fnv(f),$f \in F$ the vertices of a face:

$$\text{fnv}(f) = \{v_{i_1}, \ldots, v_{i_n}\}, \text{where } \{v_{i_1}, \ldots, v_{i_n}\} \subset V \wedge [v_{i_1}, \ldots, v_{i_n}] = f \tag{2.7}$$

fne(f),$f \in F$ the neighboring edges of a face:

$$\text{fne}(f) = \{[v, w] \in E | \exists w_{i_1}, \ldots, w_{i_n} \in V : [v, w, w_{i_1}, \ldots, w_{i_n}] = f\} \tag{2.8}$$

fnf(f),$f \in F$ the neighboring faces of a face:

$$\text{fnf}(f) = \{f' \in F | f' \neq f \wedge fnv(f) \cup fnv(f') \neq \emptyset\} \tag{2.9}$$

As an example see Figure 2.1 for an example of a valid and a not valid mesh. The second mesh is not valid because the edge e belongs to more than two faces, not satisfying thereby condition 4 of Definition 2.1.1.An additional Figure 2.2 shows an interesting valid mesh, a Klein bottle, non-orientable and self-intersecting.

2.1.2 Representation of the mesh

In our algorithm we often need to query the mesh for different kinds of neighborhood-related information (get the list of the neighboring vertices of a vertex, the neighboring faces of a vertex, the neighboring faces of a face etc.). Therefore it is essential for us to be able to query fast for these information. As fast enough we consider constant time or linear dependent on some local characteristic (valence of a vertex, number of the vertices of a face) time responses. Here we explore different representations:

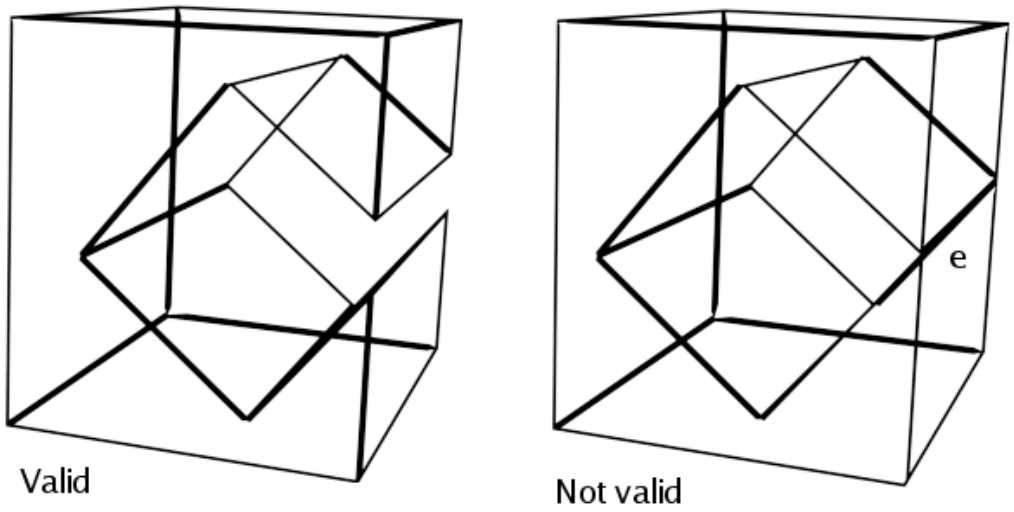

Figure 2.1: A valid and a non-valid mesh. The mesh on the left is not valid because the edge e is shared by more than two faces

Face list

Every face is defined as a list of its vertices. The face is defined as a list of edges. The reason for his popularity is the fact, that this structure is well suited for hardware accelerated displaying. Its disadvantage is bad performance in adjacency queries ($o(n)$ for most queries).

Winged Edge

Proposed by B.Baumgart [2] .It was originally developed for manifold meshes with no holes in the faces, and later extended to handle those situations, but we will not explore those modifications because they are not needed for our algorithm.

Its basic idea is to use edges as the central information holders. For each edge we store the following information (see Figure 2.3: (for edge a))

- vertices of this edge (X,Y)
- its left and right faces (1,2)
- the predecessor and successor edges of this edge when traversing its left face (b,d)
- the predecessor and successor edges of this edge when traversing its right face (e,c)

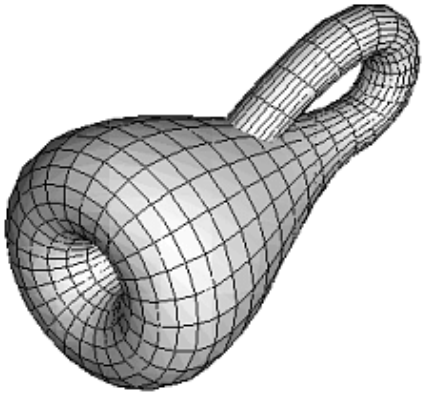

Figure 2.2: The Klein bottle mesh

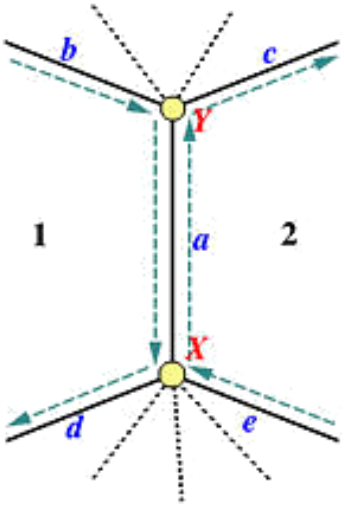

Figure 2.3: The winged edge data structure

With those informations we can build a edge table. This example shows one row of the edge table which corresponds to Figure 2.3

We need two more, very simple tables, which contain:

- for each vertex $v \in V$ one edge that is incident to this vertex and the position of the vertex
- for each face one of it's boundary edges

Edge	Vertices		Faces		Left Traverse		Right Traverse	
Name	Start	End	Left	Right	Pred	Succ	Pred	Succ
a	X	Y	1	2	b	d	e	c

Table 2.1: Winged edge representation

This representation is able to answer the necessary queries (Section 2.1.2) fast enough. Another advantage of him is its constant sized data structures.

Halfedge

Published by Eastman [3], also called as "Doubly-connected edge list". Every edge is represented by 2 directed "halfedges". For every halfedge we have a record of (see Figure 2.4): (edge e)

- the vertex at the end of the halfedge (V)
- the oppositely oriented halfedge (f)
- the face the halfedge borders (P)
- the next halfedge around the face (g)

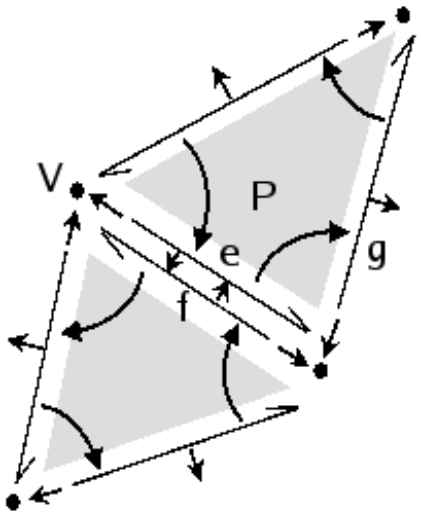

Figure 2.4: The halfedge data structure

We create two more tables, one for the vertices. The first table serves as a storage for every vertex: we store one of the halfedges that start from each vertex and the position of the vertex. The other table contains for every face one of its bordering halfedges. With those information the time complexity of the queries (Section 2.1.2) is fast enough.

2.2 Functional representations

2.2.1 Implicit surfaces

See Figure 2.5 and Figure 2.6 as examples of this representation.

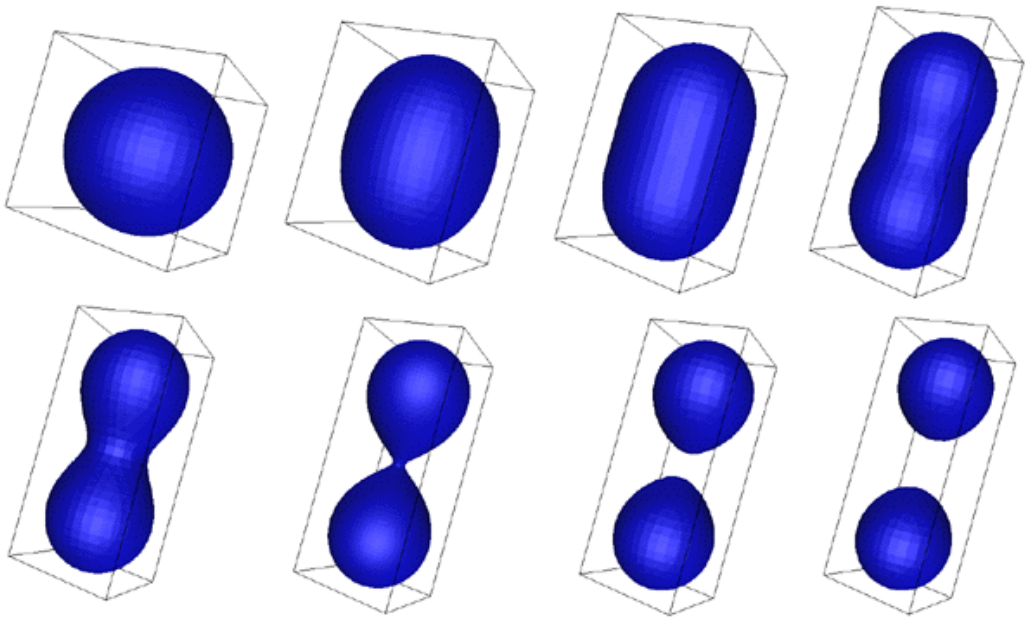

Figure 2.5: Changing of the implicit surface by two moving control points

An implicit surface is defined by a continuous function $f\colon \mathbb{R}^3 \to \mathbb{R}$ which assigns a scalar value to each point of the space. The surface S is defined as:

$$S = \{p \in R^3 | f(p) = 0\} \tag{2.10}$$

TODO: POINTS AND VECTORS SHOULD BE BOLD!

As we can see implicit surfaces are contours (isosurfaces) of a scalar field. The function f can be seen as the field potential at a given point in space. The isosurface is usually constructed from a set of control points ($v_0, \ldots, v_n \in \mathbb{R}^3$

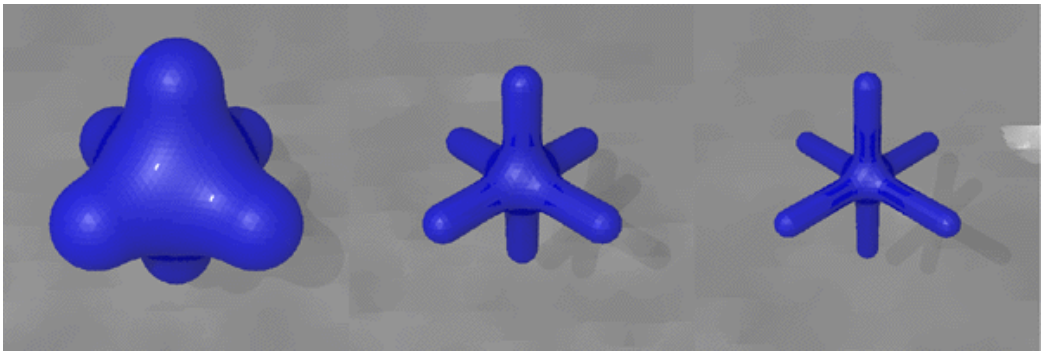

Figure 2.6: More complex example of the natural blending of the implicit surfaces

), and the field potential is $f(p) = \sum_{i=1}^{n} D(r_i)$, where $D(r_i)$ is the field function and r_i denotes the square of the euclidean distance between the points p and v_i ($r_i(p, v_i) = \|p - v_i\|^2$).

There are plenty of ways for the definition of the function $D(r)$. The methods are classified according to them.

Distance based approach : The numerically simplest solution is to define:

$$D(r) = 1/r^2. \tag{2.11}$$

Blobs Another field function was proposed by Jim Blinn (it is based on the electron density fields):

$$D(r) = ae^{-br^2}, \tag{2.12}$$

where a is called threshold, and relates to the height, and b is called *blobbyness*, and relates to the standard deviation of the curve.

There are two problems with the above solutions:

1. To calculate the field value in one point, we have to summarize the contribution from all the control points. That means the the computation time linearly depends on the number of control points.

2. The usual rendering method of implicit surfaces uses ray-tracing, which involves finding the intersections between rays and the implicit surface. Because of that it's desired to use simple functions, what is definitely not the case for the above mentioned two solutions.

These two problems are solved in the following approaches.

Metaballs Nishimura [7] uses the following formula:

$$D(r) = \begin{cases} a(1 - 3r^2/b^2) & 0 \leq r \leq b/3 \\ 3a/2(1 - r/b)^2 & b/3 \leq r \leq b \\ 0 & b \leq r \end{cases} \tag{2.13}$$

Soft Objects Wivil [8] proposes the following field function:

$$D(r) = \begin{cases} a(1 - (4r^6/9b^6) + (17r^4/9b^4) - (22r^2/9b^2)) & r^2 < b^2 \\ 0 & r^2 \geq b^2 \end{cases} \tag{2.14}$$

a scales the function, and b specifies the distances after which the given control point has no influence on the field strength. One of its advantages over the metaballs is that it only uses squares, so it is not necessary to compute square roots (which are usually slower to compute).

For a comparison between the different approaches see Figure 2.7 For

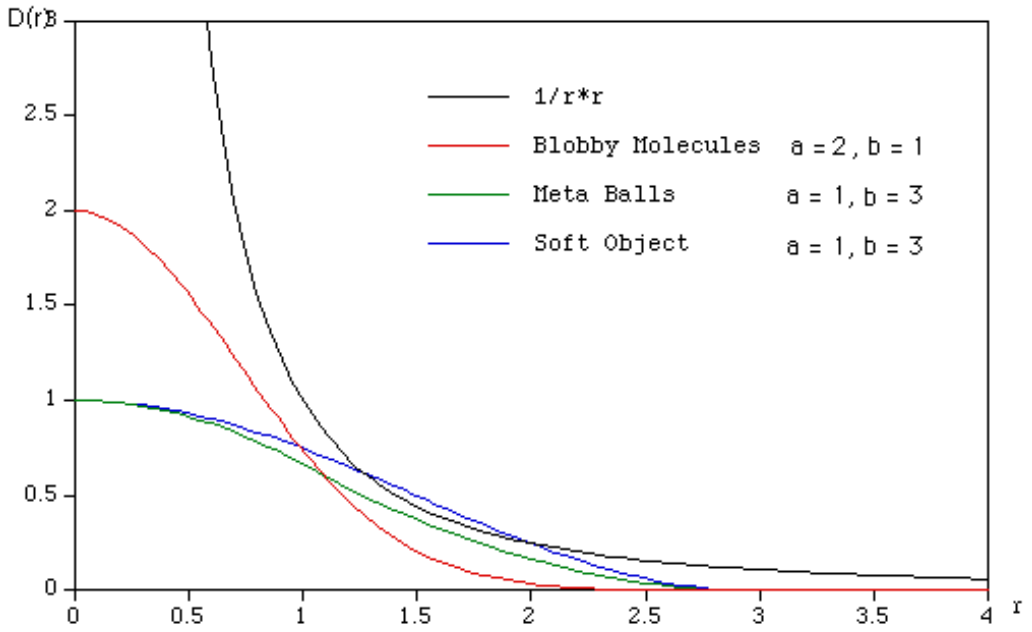

Figure 2.7: A Comparison of the value of the field function ($D(r)$) depending on the distance(r)for the different approaches

more information see [4] and [5] As we have seen using implicit surfaces it's relatively simple to achieve the geometric operation we are trying to create.

2.3 Bézier curves

2.3.1 Bernstein polynomials

The Bernstein polynomial of degree n is defined as

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, i \in \{0, \ldots, n\}, t \in \mathbb{R} \tag{2.15}$$

They have a number of useful properties[9]. They have extreme values at

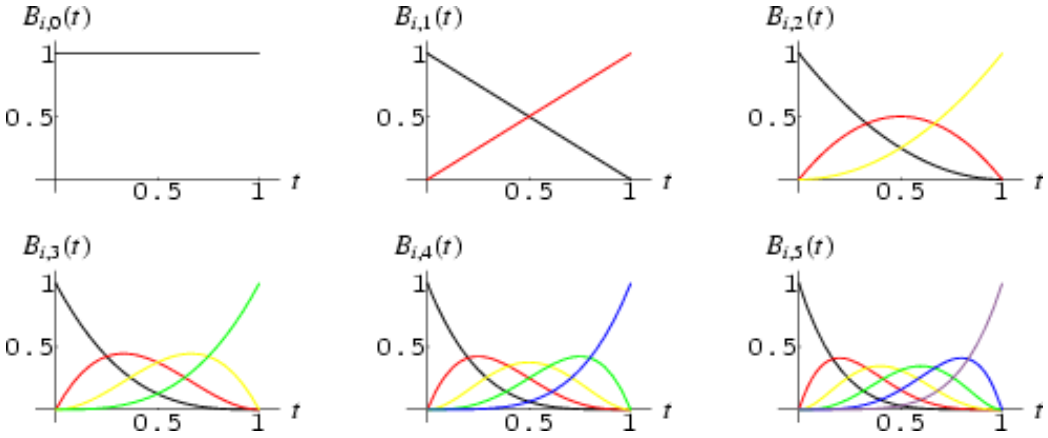

Figure 2.8: Examples of of the simpler bernstein polynomials

$$B_{i,n}(0) = \begin{cases} 0 & \text{for } i = 1, 2, \ldots, n \\ 1 & \text{for } i = 0 \text{ or } 1, \end{cases} \tag{2.16}$$

$$B_{i,n}(1) = \begin{cases} 0 & \text{for } i = 1, 2, \ldots, n \\ 1 & \text{for } i = 0 \text{ or } 1, \end{cases} \tag{2.17}$$

satisfy symmetry

$$B_{i,n}(t) = B_{n-i,n}(1-t), \tag{2.18}$$

positivity

$$B_{i,n}(t) \geq 0 \text{ for } 0 \leq t \leq 1, \tag{2.19}$$

normalization

$$\sum_{i=0}^{n} B_{i,n}(t) = 1, \tag{2.20}$$

can be recursively calculated as

$$B_{i,n}(t) = (1-t)B_{i,n-1}(t) + tB_{i-1,n-1}(t), \tag{2.21}$$

and $B_{i,n}$ with $i \neq 0, n$ has a single unique local maximum of

$$i^i n^{-n} (n-i)^{n-i} \binom{n}{i} \tag{2.22}$$

occurring at $t = i/n$.

2.3.2 Bézier segments

Given a set $n+1$ control points $P_0, P_1, \ldots, P_n \in R^3$, the corresponding Bézier segment is given by

$$C(t) = \sum_{i=0}^{n} P_i B_{i,n}(t) \tag{2.23}$$

where $B_{i,n}(t)$ is a Bernstein polynomial and $t \in [0, 1]$.

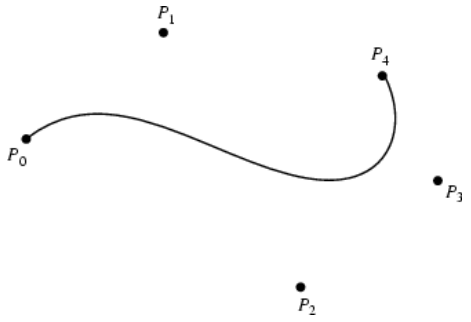

Figure 2.9: Example of a fifth order two dimensional Bézier curve

They have some useful properties inherited from the Bernstein polynomials such that,

- they pass through the first and last control points because of (2.16)
- they lie within the convex hull of the control points (because the curve is a convex combination of the control points, see (2.19) and (2.20))
- they can be split at an arbitrary $t \in (0, 1)$ because of (2.21). This algorithm is called the *de Casteljau algorithm.*

Their undesirable properties are their numerical instability for large number of control points, and the fact that moving a control point changes the whole curve. That can be avoided by using multiple lower degree Bézier curves.

Chapter 3

The blending algorithm

The aim of this chapter is to describe the algorithm we developed. We give an informal overview of the algorithm, define the input-output constrains, formally describe every step of the algorithm and show some basic time estimations.

3.1 An intuitive overview

The input consist of a mesh and a list of "selections". A selection is an area on the surface of the mesh selected by the user. Our algorithm basically forms smoothly blended canal surfaces between these selections, thereby mimicking the blending of implicit surfaces. We call the boundary of the selections "rings" (because of their circular shape), and the canal surfaces "worms".

For the case when there are only two selections, the forming of the blending canal surface is intuitive (see Figure 3.1)

For the case when there are three or more selections we choose the following method: We add a virtual sphere to the scene and create the blending surfaces between the selections and the sphere (see Figure 3.2) We call the blending surface formed by the virtual sphere and the canal surfaces ("worms") a "hydra" (after the serpent-like creature from the Greek mythology with multiple heads, see Figure 3.3).

3.2 Basic definitions

Definition 3.2.1 *A selection $S_M = \{f_1, \ldots, f_n\}$ is a subset of the faces of mesh M ($M = (V, E, F) : \forall i \in \{1, \ldots, n\} : f_i \in F\}$, where between every two faces of the selection there is a list of connected faces from the selection:*

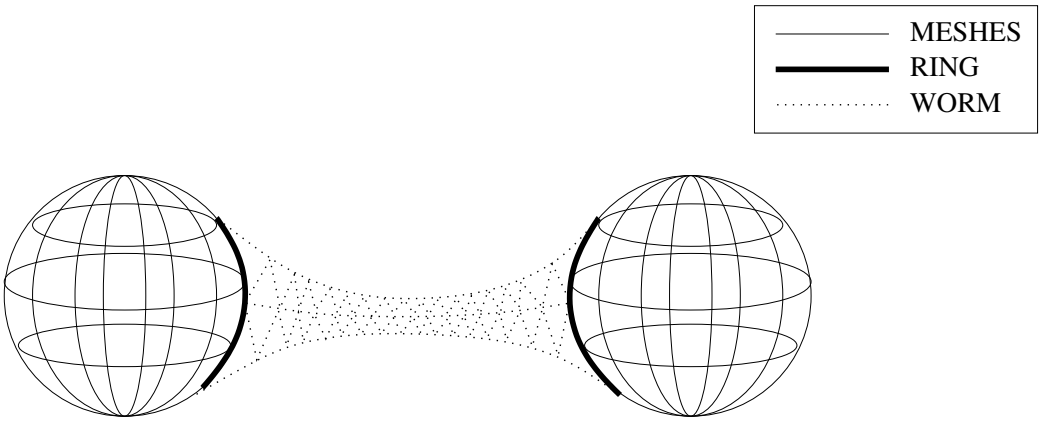

Figure 3.1: Creating a blended canal surface (a "worm") between two surfaces

$$\forall f_1, f_2 \in S_M : \exists f'_{i_1}, \ldots, f'_{i_n} \in S_M : f_{i_{k+1}} \in \text{fnf}(i_k), k \in \{1, \ldots, n-1\} \quad (3.1)$$

Definition 3.2.2 *The boundary of the selection can be defined as the set of those vertices, which have neighboring faces from both the selection, and the complement of the selection:*

$$\text{boundary}(S_M) = \{v \in V | \exists f_1, f_2 \in F : \{f_1, f_2\} \subset \text{vnf}(v) \wedge f_1 \in S_M \wedge f_2 \notin S_M \quad (3.2)$$

Because of (3.1) the boundary vertices can be ordered into a circular list, where every two consecutive vertices are neighbors.

Definition 3.2.3 *The* ring *is a circular list of the boundary vertices of a selection. For the sake of simplicity we define them as tuples:*

$$\text{ring}(S_M) = (v_1, \ldots, v_n), \text{where } v_k \in \text{boundary}(S_M) \wedge k \in \{1, \ldots, n\} \\ \wedge v_{k+1} \in \text{vnv}(v_k) \forall k \in \{1, \ldots, n-1\} \wedge v_1 \in \text{vnv}(v_n) \quad (3.3)$$

Definition 3.2.4 *The center of a ring $R = (v_1, \ldots, v_n)$ is a point $P_r \in \mathbb{R}^3$ defined as:*

$$P_r = \sum_{i=1}^{n} (1/n) v_i \quad (3.4)$$

Definition 3.2.5 *The normal vector of a ring $R = (v_1, \ldots, v_n\}$ is the vector $\overrightarrow{n_R}$ defined as the normal vector of the plane p_R defined by three noncolinear vertices from R for which plane the sum of the distances of the other vertices from the plane is the least.*

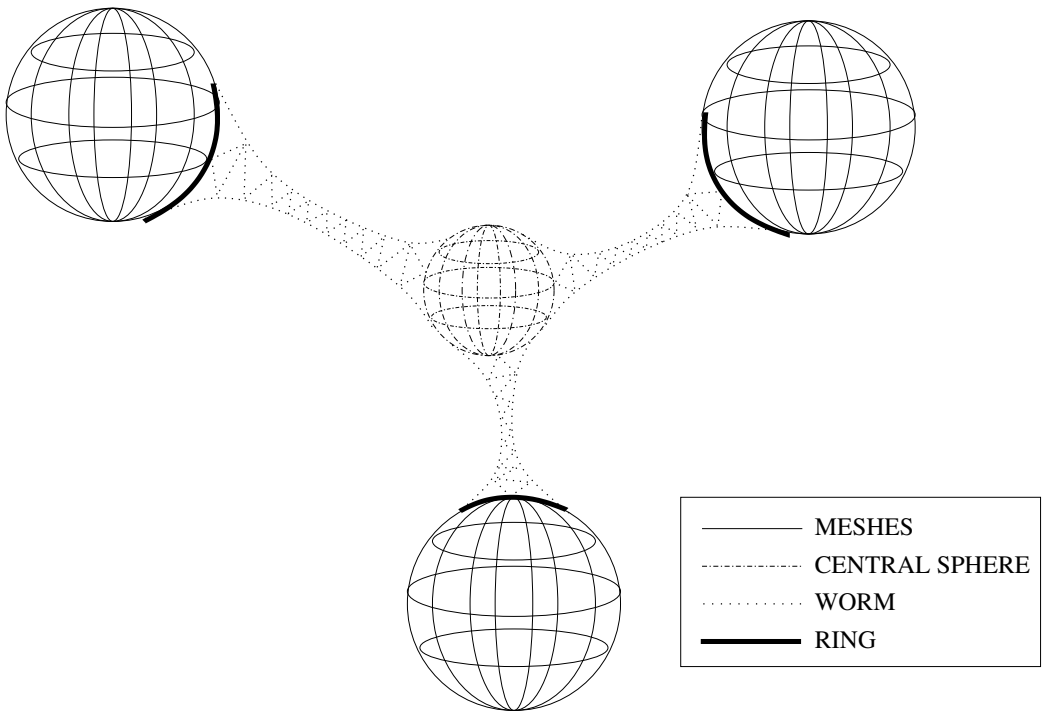

Figure 3.2: Creating a blended canal surface (a "hydra") between three surfaces

3.3 Input-output characterization

3.3.1 Input

mesh M The mesh[1] may contain boundary edges because our algorithm only modifies the surface[2] locally,but must not contain them around the selections (the neighbors of the vertices of the rings must not be boundary vertices).

set of selections $L = \{S_1, \ldots, S_n\}$

[1]One must realize that there is no fundamental difference between blending together the selected parts of multiple meshes, and blending together the selections of a single mesh. The blending between multiple meshes can simply be achieved by generating the union of all the input meshes and working with it as a single mesh. This is the reason why our algorithm works on a single mesh and not on multiple meshes

[2]In the following text we will use the term mesh and surface interchangeably

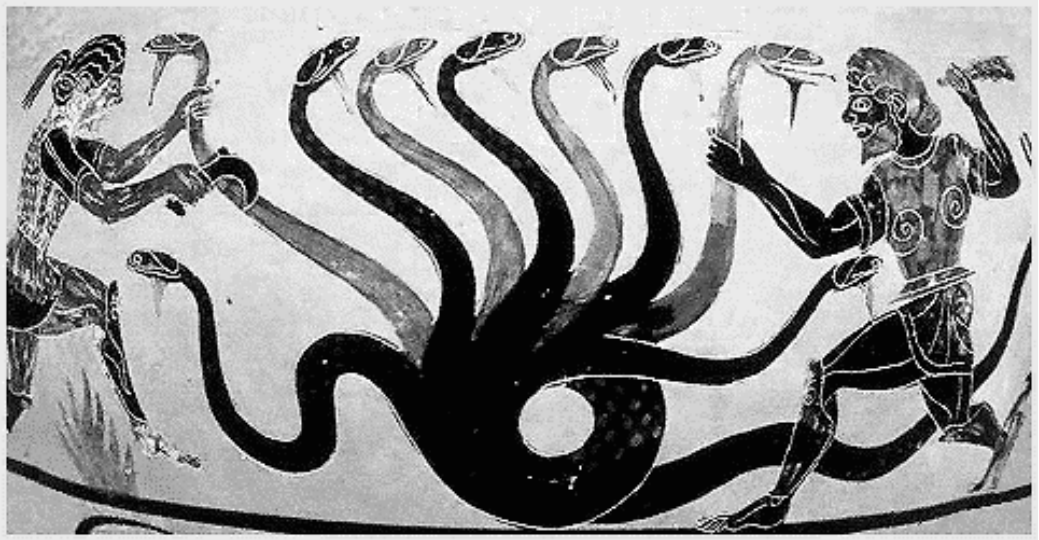

Figure 3.3: *(Iolaos, Hydra , Herakles)* THE HYDRA LERNAIA was a gigantic serpent-like creature living in water with nine heads, one of which was immortal. She was slain by Herakles who buried her immortal head beneath a boulder

3.3.2 Output

Mesh N A mesh defined as a list of faces, which is formed by the input mesh deformed the way as required by the user (the selected parts blended together)

3.4 Blending between two selections

The algorithm can be shortly described as:

1. calculate the rings from the selections
2. remove the selections from the mesh
3. determine a suitable form for the worm[3]
4. generate the worm

[3]In the following text this step will be discussed after the fourth step, because to understand the mechanism for selecting a suitable form for the worm, one must understand the way the worm is defined

3.4.1 calculate the rings from the selections

First observe that determining whether a vertex, edge or face lies on the boundary of the selection involves determining whether it has a neighboring face which is not part of the selection. It can be done in three steps:

1. for every face check whether it is a boundary face or not
2. for every vertex of the boundary faces check whether they are boundary vertices or not
3. order the boundary vertices in the order required by Definition 3.2.3

3.4.2 remove the selections

This step is necessary because without it the edges forming the boundary of the selection would be shared by more than two faces, thereby violating Condition 4 of Definition 2.1.2. It is a simple face removal algorithm which can be done fast enough on the halfedge data structure[4].

3.4.3 generate the worms

The problem can be defined as generate a canal surface, the ends of which touch the two rings. The form of the worm is defined using cubic bezier curves. Refer to Figure 3.4 where you can see three cubic bezier curves (B_1, B_2, B_3), and their control polygons $(A_{i,k})$. In the following we will use n to denote the number of bezier curves.

The input for our algorithm for generating the worm consists of:

- The control points for the bezier curves:

  $A_{i,k}$,where $i \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, 4\}$

- The value defining the number of samples we take from the bezier curve:

  r (example: if $r = 4$, we take four samples (at $t = 0$,$t = 0.33$,$t = 0.66$,$t = 1$)

We denote the worms as $W_{n,r}$ (example: the Figure 3.5 refers to a $W_{3,9}$: three bezier curves sampled at nine points)

The algorithm to generate the worms is the following:

[4]When using the term "fast enough" we refer to the time constrains defined in Section 2.1.2

sample the value of the bezier curves at the given intervals We denote the sampled values as $C_{i,k}$, where $i \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, r\}$. Therefore:

$C_{i,k} = B_{0,3}(t)A_{i,1} + B_{1,3}(t)A_{i,2} + B_{2,3}(t)A_{i,3} + B_{3,3}A_{i,4}$,

where $t = (k-1)(1/r)$, $k \in \{1, \ldots, r\}$ and $B_{j,l}$ is the j-th Bernstein polynomial of degree l.

Refer to Figure 3.5 where the value of the Bezier curves is already sampled at uniform intervals ($C_{i,k}$).

generate the triangle strips which form the surface of the worm A triangle strip is a list of triangle where each two consecutive triangles share an edge.

The worms surface is formed by triangle strips inserted in between every two consecutive triangle strips. Refer to Figure 3.6 for an example of one of the triangle strips created, and to Figure 3.7 for an example of all the strips created.

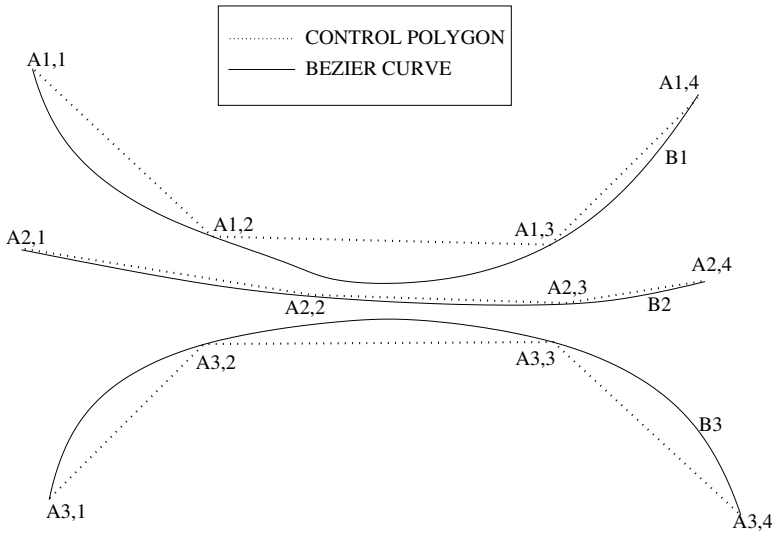

Figure 3.4: Constructing the worm: the first step: showing the bezier curves and the control polygons

The worms surface is formed by as many triangle strips as the number of the bezier curves. Therefore the worm $W_{n,r}$ is made of n triangle strips.

The algorithm for generating the triangle strips in pseudo-code:

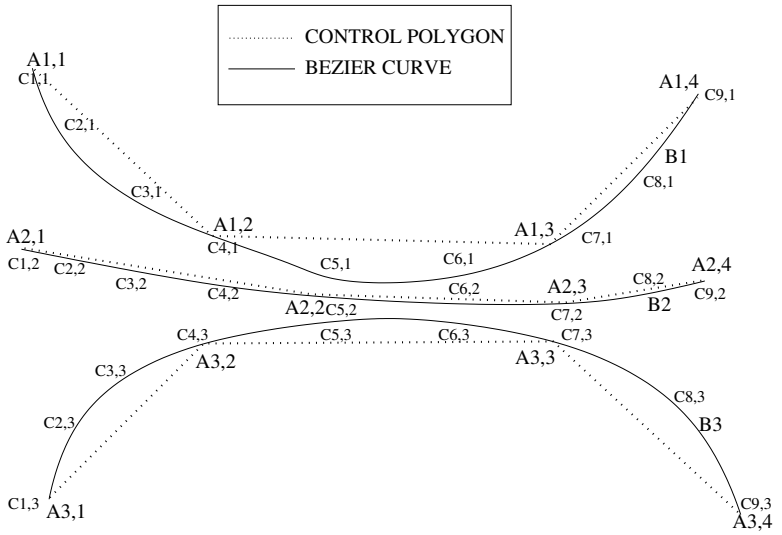

Figure 3.5: Constructing the worm: the second step: showing the bezier curves, the control polygons and the points sampled uniformly on the curves

```
for curve = 1 to n:
    for sample = 1 to k:
        nextcurve = (curve + 1) mod n
        p0 = C[curve][sample]
        p1 = C[nextcurve][sample]
        p2 = C[nextcurve][sample+1]
        addTriangle(p0,p1,p2)

        q0 = C[nextcurve][sample+1]
        q1 = C[curve][sample+1]
        q2 = C[curve][sample]
        addTriangle(q0,q1,q2)
```

On Figure 3.6 the first triangle strip is made, and on Figure 3.7 the worm is completed.

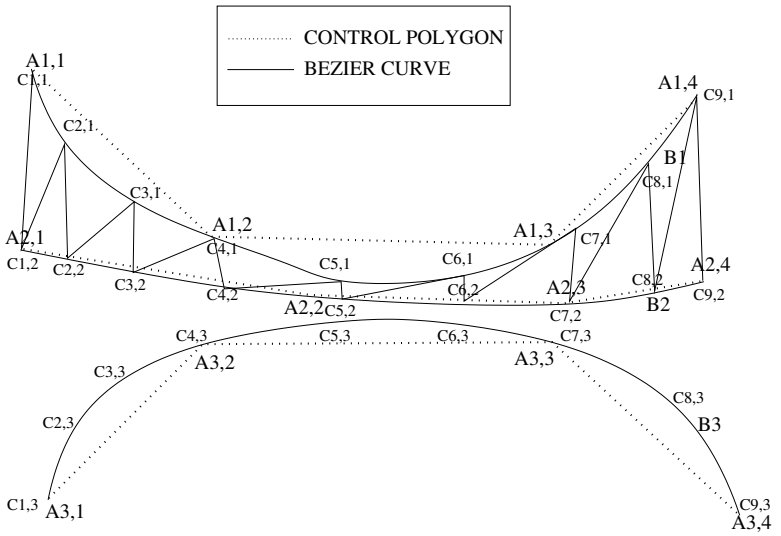

Figure 3.6: Constructing the worm: the third step: showing the bezier curves, the control polygons, the sampled points and one of the triangle strips

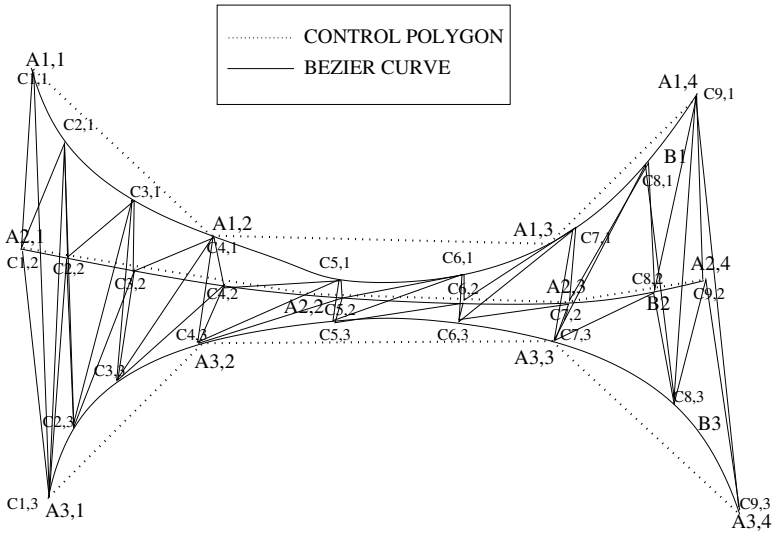

Figure 3.7: Constructing the worm: the fourth, final step: showing the bezier curves, the control polygons, the sampled points and all the triangle strips

3.4.4 determine a suitable form for the worm

As we have seen in the previous section positioning the bezier curves suitably is crucial for achieving smooth blendings, because the bezier curves define the form of the worm. The number of bezier curves used is defined by the number of vertices contained in the longer ring:

ring1= $\{v_1, \ldots, v_h\}$, ring2= $\{w_1, \ldots, w_j\} : n = \max(h, j)$

For every curve we have to calculate four control points:A_1, A_2, A_3, A_4.

The calculation algorithm can be described by the following steps:

Calculate the first and the last control point The first and the last control point has to be a vertex of the rings to actually connect the worm to the surface.

Two problems arise when selecting the first and the last control point:

- Which vertex do we select from the first and which vertex from the second ring, or more formally: how to find the function which maps the vertices of the first ring into the vertices of the second ring? The solution is described in Section 3.4.4
- How do we select the bezier curves if the number of the vertices in the two rings is not the same? The solution is described in the Section 3.4.4

Calculate the second and the third control point Theoretically the second and the third control point can be at an arbitrary point in the space, the worm will still connect the two rings. But to achieve smooth blending, more care is needed. Our proposed algorithm for selecting the second and third vertex is described in Section 3.4.4

Finding the correct mapping

We consider two cases:

both rings contain the same number of vertices In this case we use the most straightforward mapping:

we map the i-th vertex from the first ring into the i=th vertex of the second ring.

ring1= $\{v_1, \ldots, v_l\}$ and ring2= $\{w_1, \ldots, w_l\}$.

Then we will use l bezier curves $B_1, \ldots, B_l$, for which: $A_{j,1} = v_j$ and $A_{j,4} = w_j, j \in \{1, \ldots, l\}$

one of the rings contains more vertices than the second ring Let the first ring have more vertices (the solution is symmetric if the second ring has more vertices).

let us denote the first ring as *ring1* and the second ring as *ring2*.

We solve the problem by adding vertices into ring2. We expand ring2 to contain the same number of vertices as ring1 and then use the algorithm for mapping two identically long rings.

The expanding algorithm is described in Section 3.4.4

Expanding a ring to a given length

As an explanation of the expanding algorithm refer to Figure 3.8. Here we have a ring containing four vertices $(V_0, \ldots, V_4)$, which has to be expanded to contain twelve vertices $(W_0, \ldots, W_11)$.

We have to warn the reader that the figure is wrong in one fact: vertices W_0, W_1, W_2 are at the same position as vertex V_0, vertices W_3, W_4, W_5 at the same place as vertex V_1 and so on. The Figure is drawn this way to make the expanding algorithm easier to understand.

What we do is the following:

1. We replace the ring containing the four vertices with a ring containing twelve vertices, where the twelve vertices are not yet defined
2. we assign $W_0 = V_0$, $W_3 = V_1$, $W_6 = V_2$, $W_9 = V_3$
3. we sequentially examine every vertex of the ring beginning with W_0 and ending with W_11, and if the vertex is not yet defined, we define it to be the same as the previous vertex.

   This way we get $V_0 = W_0 = W_1 = W_2, V_1 = W_3 = W_4 = W_5, V_2 = W_6 = W_7 = W_8, V_3 = W_9 = W_10 = W_11$

The pseudo-code of the algorithm used here:
The ring is defined as $R = (v_0, \ldots, v_{l-1})$ and has to be expanded to contain m vertices.
Our algorithm produces a new ring (P) which is a copy of the ring on input expanded to the required length.
the function "int" is defined as $\text{int}(x) = \lfloor x \rfloor$

```
step = m/l
P = empty-ring-of-length(m)
for i=0 to l-1:
```

```
    index = int(i*step)
    P[index] = R[i]

lastValue = -1

for i=0 to m-1:
    if P[i] != -1:
        lastValue = P[i]
    if P[i] == -1:
        P[i] = lastValue
```

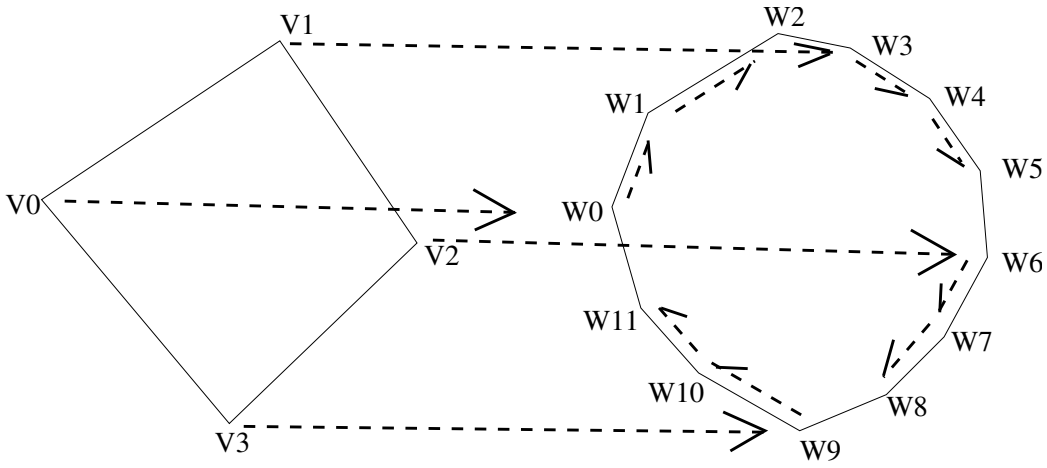

Figure 3.8: Expanding the ring: creating a copy of the ring which contains the same vertices as the original but is longer because contains some vertices multiple times

Placing the middle vertices

Refer to Figure 3.9. This is an example with two rings (T_1, T_2, T_3, T_4) and (T_1', T_2', T_3', T_4'). The vertex normals in the given points are denoted as T_iN_i and $T_i'N_i'$. We will use nearly the same notation for the general case:
ring1= $(T_1, \ldots, T_j)$ and ring2= $(T_1', \ldots, T_j')$[5]
the normals are denoted as $\overrightarrow{n_i} = T_iN_i$ and $\overrightarrow{n_i'} = T_i'N_I'$, where $i \in \{1, \ldots, j\}$

We calculate the following points:

- $T = \sum_{i=1}^{j}(1/j)T_i$

[5]At this point both rings contain the same number of points because of the application of the algorithm described in Section 3.4.4

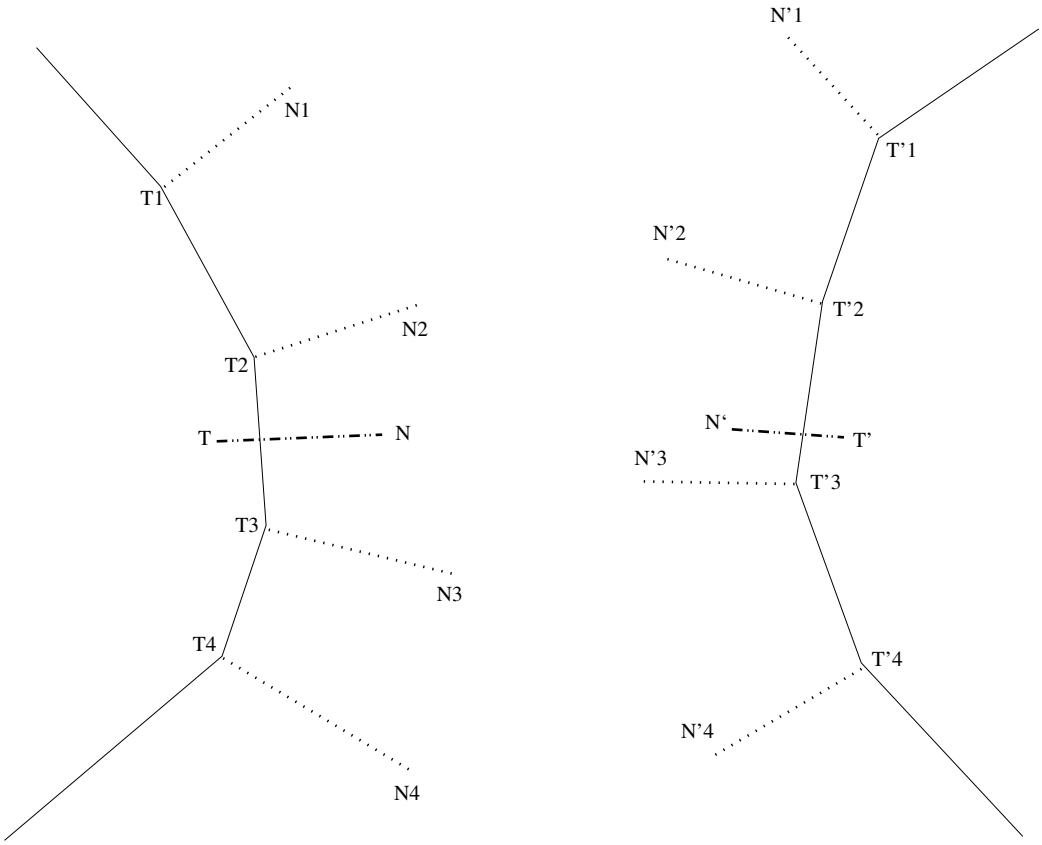

Figure 3.9: Constructing the bezier curves for the worm: first step: the vertices, the normals, and the centers shown

- $T' = \sum_{i=1}^{j}(1/j)T'_i$
- $N = \sum_{i=1}^{j}(1/j)N_i$
- $N' = \sum_{i=1}^{j}(1/j)N'_i$
- $\overrightarrow{n} = \overrightarrow{TN}$
- $\overrightarrow{n'} = \overrightarrow{T'N'}$

With these information calculated the position of the second ($A_{j,2}$) and the third ($A_{j,3}, j \in \{1, \ldots, n\}$ can be defined as:

- $A_{j,2} = T + t\overrightarrow{n}$, where $t \in < 0, 1 >$
- $A_{j,3} = T' + t'\overrightarrow{n'}$, where $t' \in < 0, 1 >$

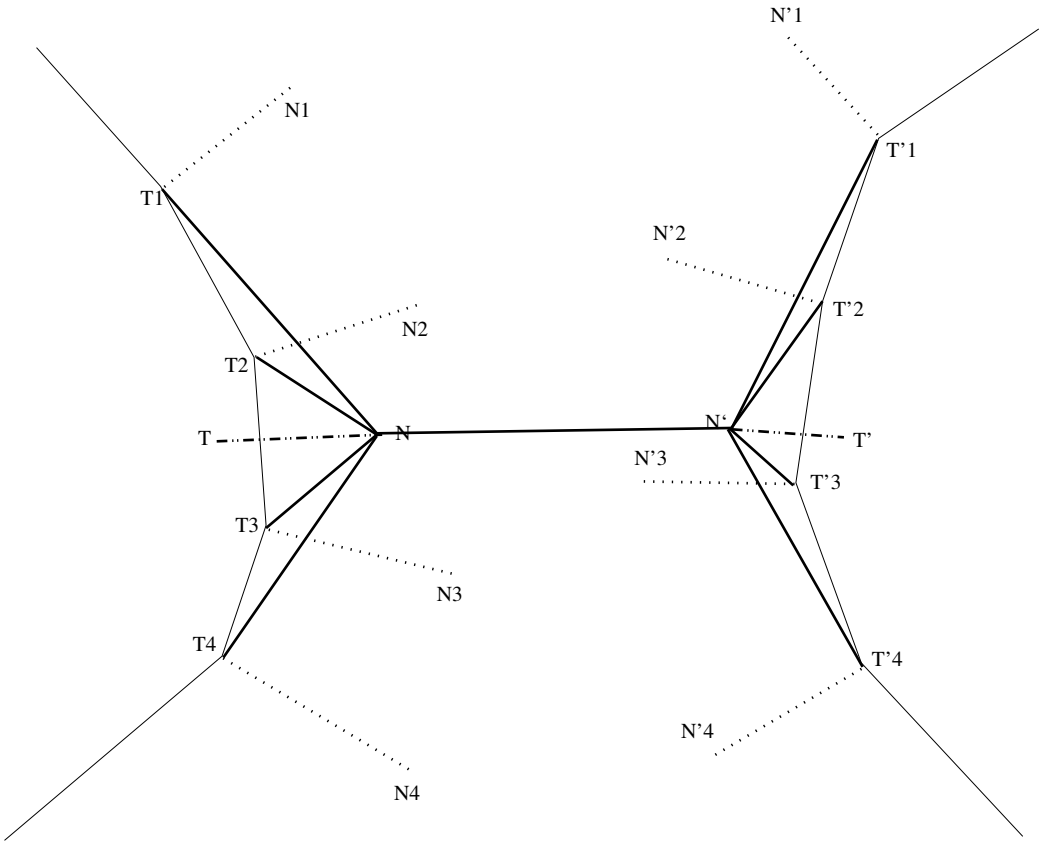

Figure 3.10: Constructing the bezier curves for the worm: second step: the vertices, the normals, the centers and the control polygons shown

The variables t and t' are left for the user to adjust the smoothness of the blending.
Refer to Figures 3.10, 3.11, 3.12 as examples of this algorithm, and to Figure 3.13 for a comparison between the results of Figure 3.11 and 3.12.

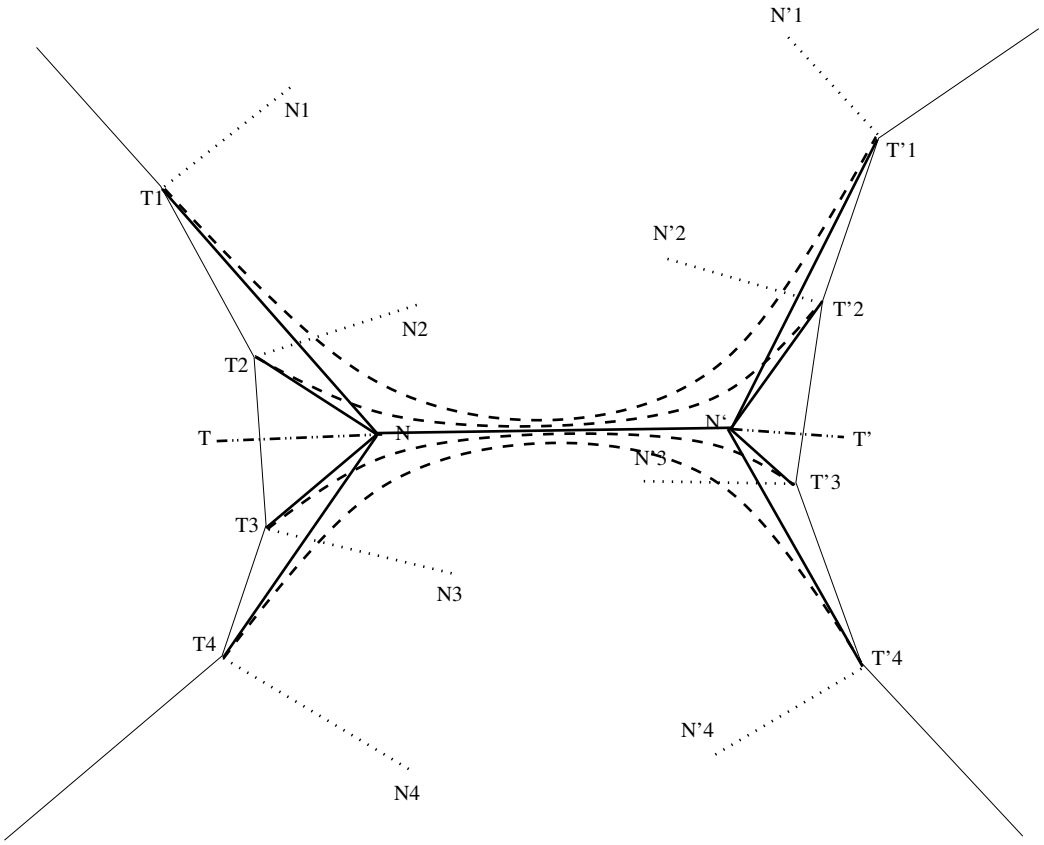

Figure 3.11: Constructing the bezier curves for the worm: third step: the vertices, the normals, the centers the control polygons and the bezier curves shown

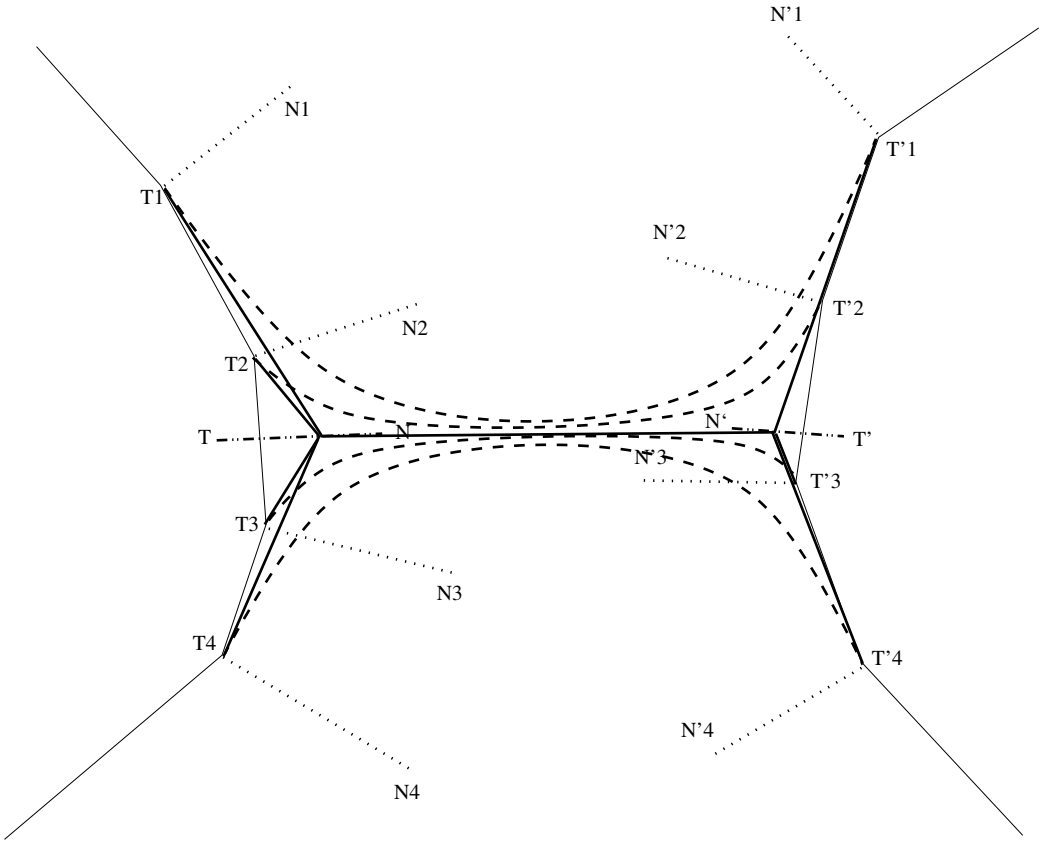

Figure 3.12: Constructing the bezier curves for the worm: fourth step: the vertices, the normals, the centers and the control polygons and the bezier curves shown in a different position

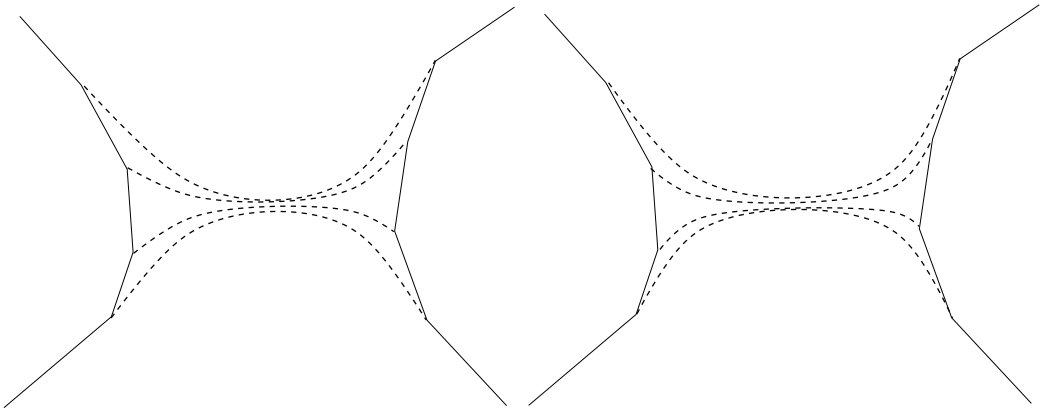

Figure 3.13: Constructing the bezier curves for the worm: a comparison shown between the two previous steps

3.5 Blending between three or more selections

For the sake of simplicity we will only discuss the issue of blending between three surfaces. Blending between more than three surfaces can be done in the same way.

The creating of the hydra (the blending consisting of the central sphere and the worms) can be described in the following way:

1. We create a sphere mesh and insert it into the center of the scene
2. We map the rings of the selections into the surface of the sphere
3. We Connect the mapped rings with their originating ring using worms
4. We erode the sphere mesh to be better blended to the worms

3.5.1 Creating the virtual sphere

1. We create a spherical mesh (for possible algorithms to create a tessellation of a sphere see [11]).
2. The suitable place to place the sphere is the center of the centers of the rings.
   We can calculate it as
   $C = \sum_{j=1}^{i}(1/j)C_i$, or for our case $C = (C_1 + C_2 + C_3)/3$, where C_i is the center of the i-th ring.
3. We move the sphere mesh to C.

3.5.2 Mapping a ring to a sphere

This process involves:

Moving the ring Involves moving the ring to a position where its center touches the surface of the sphere mesh. We will use a real sphere as an approximation of the sphere mesh.
This way the needed point can be quickly calculated:
let D denote the center of the sphere mesh, r the approximate radius of the sphere and C the center of the ring. the vector of the move ($\overrightarrow{t}$) can be calculated as: $\overrightarrow{t} = (D - C) - ((D - C)/|D - C|)r$

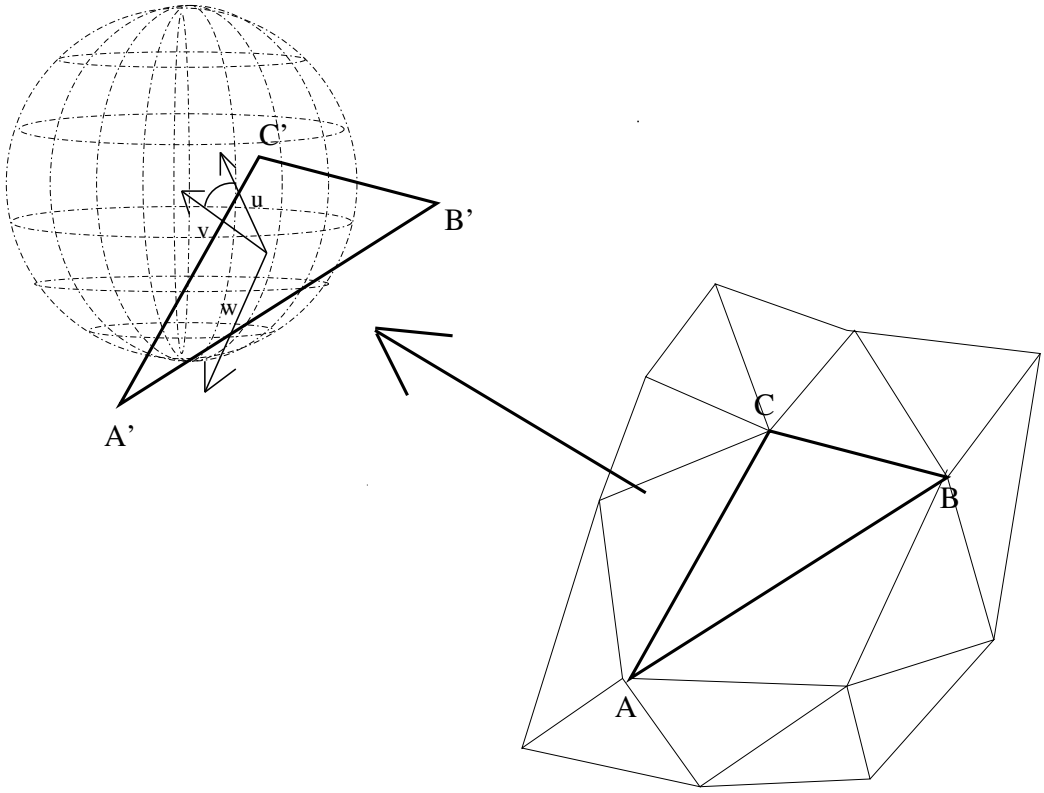

Figure 3.14: Mapping a simple triangular ring into the surface of the sphere mesh. Showing the needed transformations: translating and rotating

Rotating the ring Refer to Figure 3.14: The ring $R = (A, B, C)$ first has to be moved to the place denoted by (A', B', C') and then rotated to make his normal vector point to the center of the sphere.
Let $\overrightarrow{u}$ denote the normal vector of the ring, and $\overrightarrow{v}$ the vector pointing to the center of the sphere.
The needed rotation has to transform $\overrightarrow{u}$ into $\overrightarrow{v}$, and can be calculated as follows:
$\overrightarrow{w} = \overrightarrow{u} \times \overrightarrow{v}$ defines the axis of the rotation,and
$\alpha = \overrightarrow{u} \cdot \overrightarrow{v}$ defines the angle of the rotation.

Mapping the vertices of the ring to vertices of the sphere At this step we have a ring touching the sphere. Now we simply map each vertex of the ring into the nearest vertex of the sphere mesh.

Create a ring from the mapped vertices and expand the ring if needed The list of mapped vertices obtained after finishing the previous step

is not necessarily a ring. The problem is that the vertices can be so distant from each other that they are not neighbors anymore.
See Figure 3.15 as an example of this situation. The ring consisting of vertices V_1, V_2, V_3 has been mapped to the surface, but the vertices does not satisfy the neighborhood condition in Definition 3.2.3. Therefore we use a simple breadth-first path finding algorithm to find the missing vertices:
for every vertex, if the next vertex, defined by the ordering of the original ring is not the neighbor of him, then has to find a way to that vertex and add the acquired vertices to the ring:
In pseudocode:
$R = (v_1, \ldots, v_n)$ denotes the ring-like structure (a list of vertices) obtained by mapping the vertices of the ring into the spherical mesh
P the output of the algorithm: a ring which contains all the vertices from R

```
P = empty-list
for i = 1 to n:
    next = (i+1) mod n
    path = findPathBetweenVertices(R[i],R[next])
    addVerticesToList(P,path)
```

3.5.3 Connecting the mapped rings with the original rings

At this step we use the algorithm we developed for blending between two rings as discussed in Section 3.4.

3.5.4 Eroding the sphere mesh

To improve the visual appearance of the blending (to make the central sphere less noticeable, we "erode" the central sphere. That means that the "not needed" parts (the vertices of the sphere not contained in any of the rings) will be moved in the direction of the center of the sphere, this way making them less noticeable. We will do it in the following steps:

1. For every vertex of the central mesh we calculate its minimal distance from the rings on the mesh (that means the minimum from the distances from all the rings on the mesh). We can do that with a simply breadth-first path finding algorithm.

2. Let d_i denote the minimal distance from the rings for vertex $v_i, i \in \{1, \ldots, |V|\}$. For every non-ring vertex v_i we move the vertex in the direction of the center of the mesh by an amount of $p_i = f(d_i)$, where $f(d_i)$ is a function mapping the vertex-ring distance into distance in $\mathbb{R}^3$. In our algorithm we used $p_i = r(d_i/k)$ where $k = \max(d_i), i \in I$, and r is the radius of the sphere.

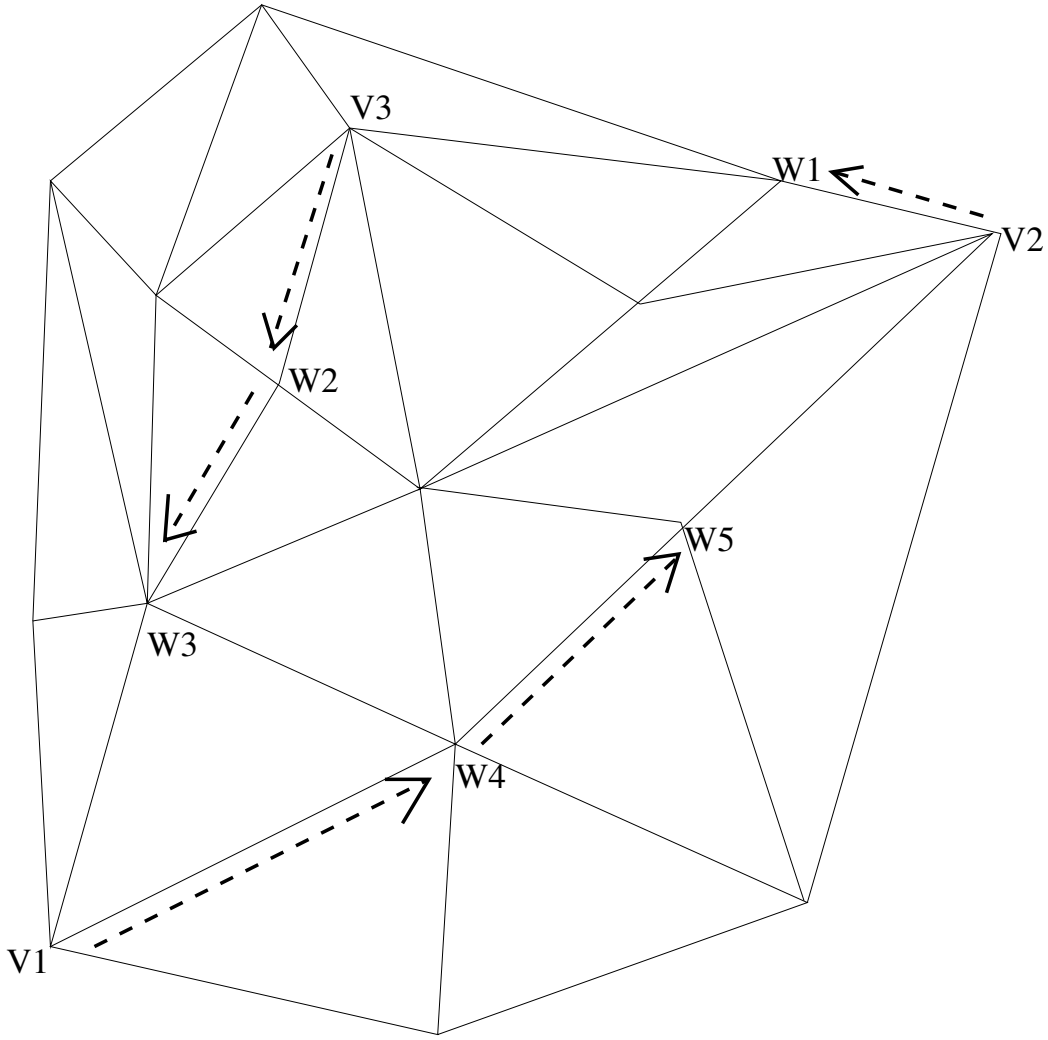

Figure 3.15: A ring (V_1, V_2, V_3) mapped to the surface. Expanding the ring-like structure to fulfill the requirements of the definition of the ring needed. Showing the searched paths for the vertices used in the expansion.

Chapter 4

Summary and Future Work

The aim of this work was to implement on boundary based objects one of the artistically most appealing abilities of implicit surfaces: the ability to naturally form smooth blendings.
In the previous chapter we have shown that it is possible to emulate or mimic that behavior: to form smooth blendings between different parts of a mesh, with as little help from the user as possible.
In this work we believed that a mixed approach gives the best efficiency: automatically find a good way to create the blendings, and after that let the user to tune the solution to his taste. At the end there are many unresolved problems and possible enhancements:

- Our ring-vertex-mapping algorithm often finds incorrect mappings. Mappings that at the end produce distorted canal surfaces with self-intersection. The user can always correct these errors, but automatizing this process is definitely a problem worth to explore.

- Explore different ways to let the user control the form of the worm. For now we proposed only two ways: either control all the middler control points manually or let them in the hands of the algorithm only allowing a little modification of the result to the user. There could be better tools or better approaches to help the user modifying the control points. Perharps group based controls or magnet like controls as can be seen in different three dimensional modeling applications.

- The usage of the bezier curves is a two-edged sword: on one hand it helps to make the calculations simple, but on the other hand the user has only two control points for every curve to modify. A possibility would be to use B-spline or rational curves.

- There is the possibility to mimic the implicit surfaces more closely. It should be possible to just move different meshes near to each other and form the complete blending manually (without the need to specify the blending areas manually)

Chapter 5

Results

In this chapter we give an example of the results achieved by our work. We demonstrate various types of connections on various types of meshes.

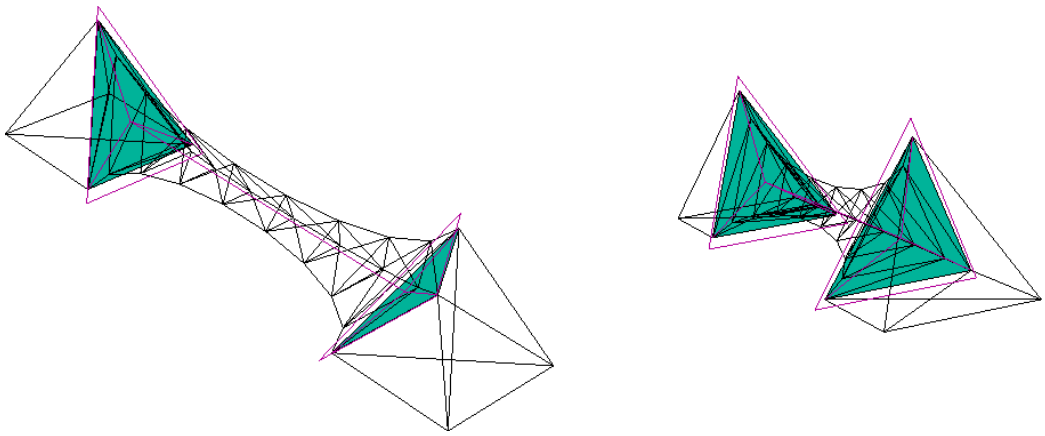

Figure 5.1: One of the simplest blending from different views

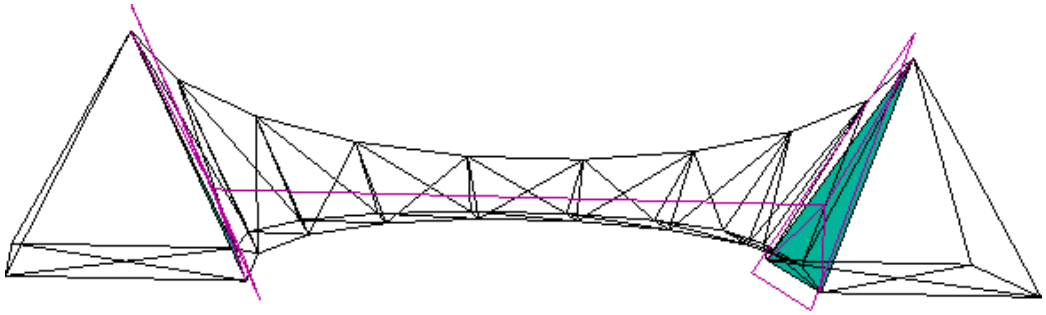

Figure 5.2: One of the simplest blending from different views

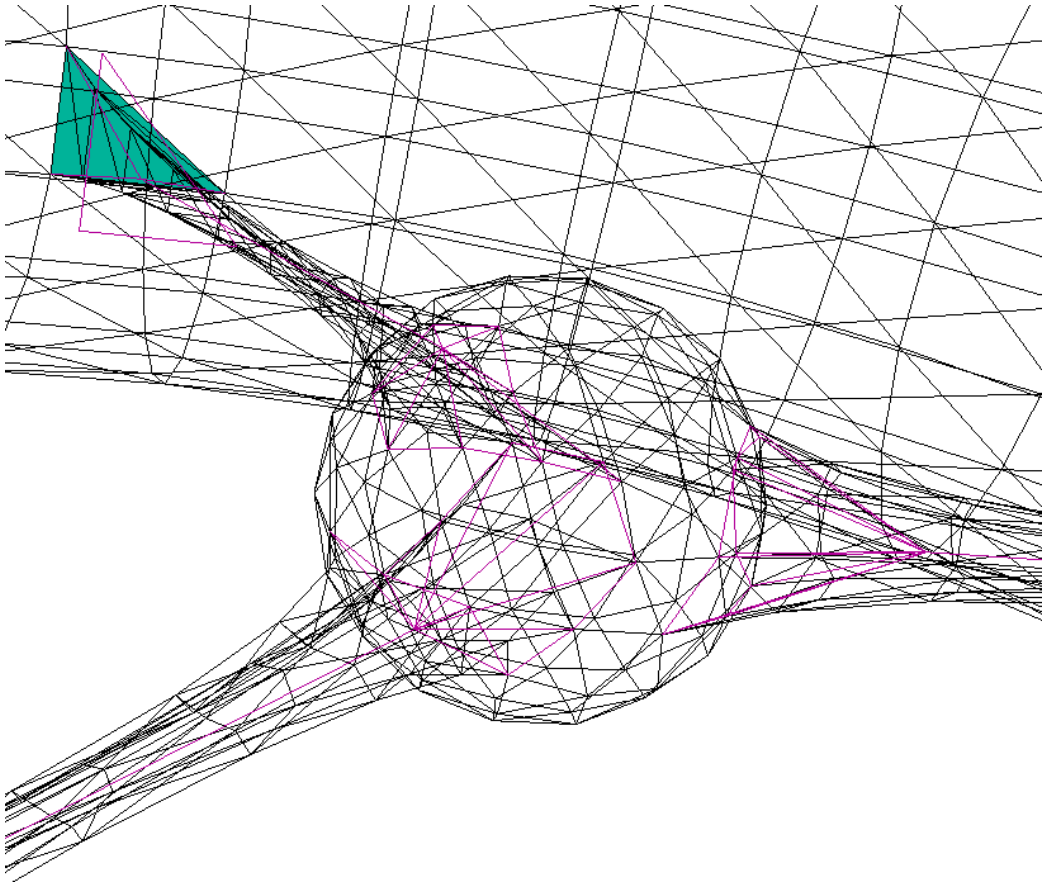

Figure 5.3: Eroding the central sphere: step1

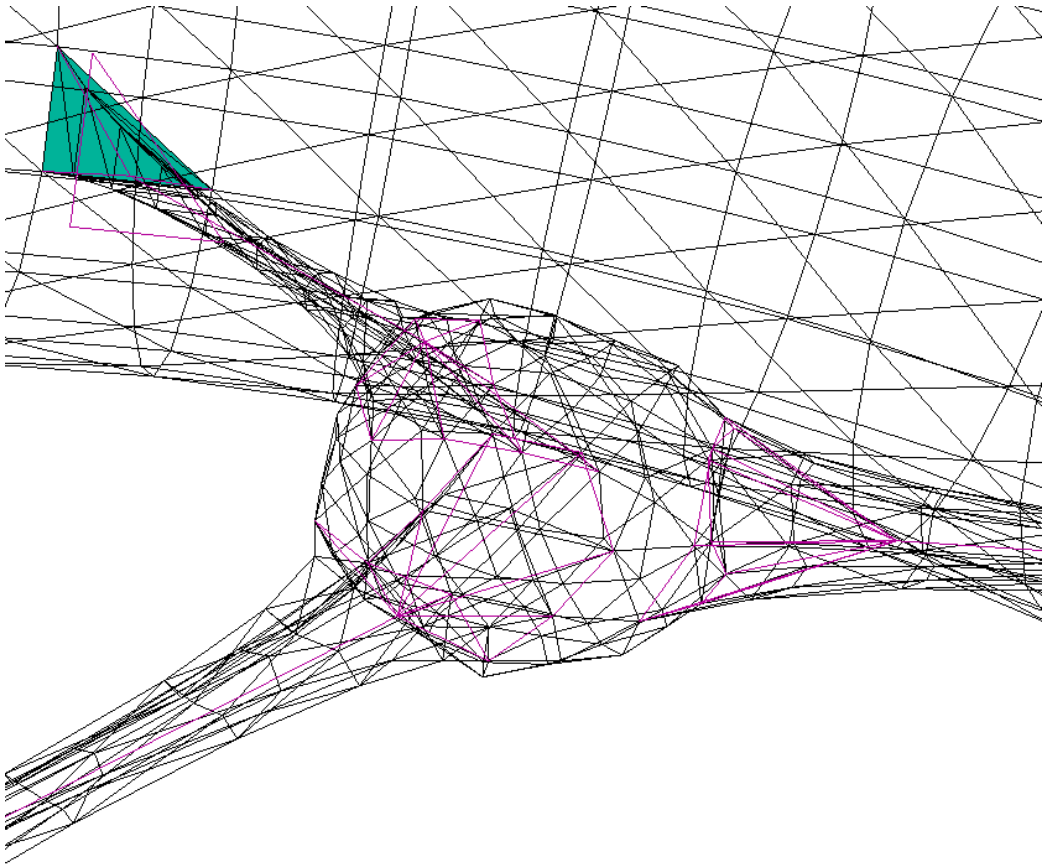

Figure 5.4: Eroding the central sphere: step2

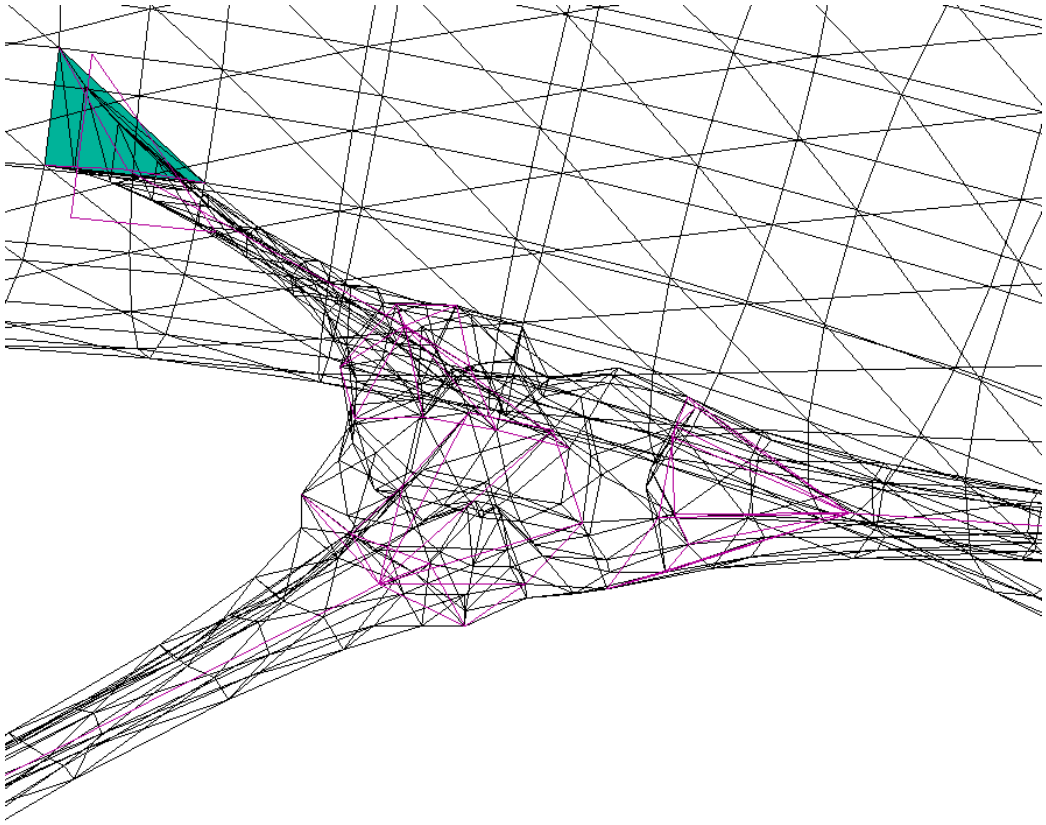

Figure 5.5: Eroding the central sphere: step3

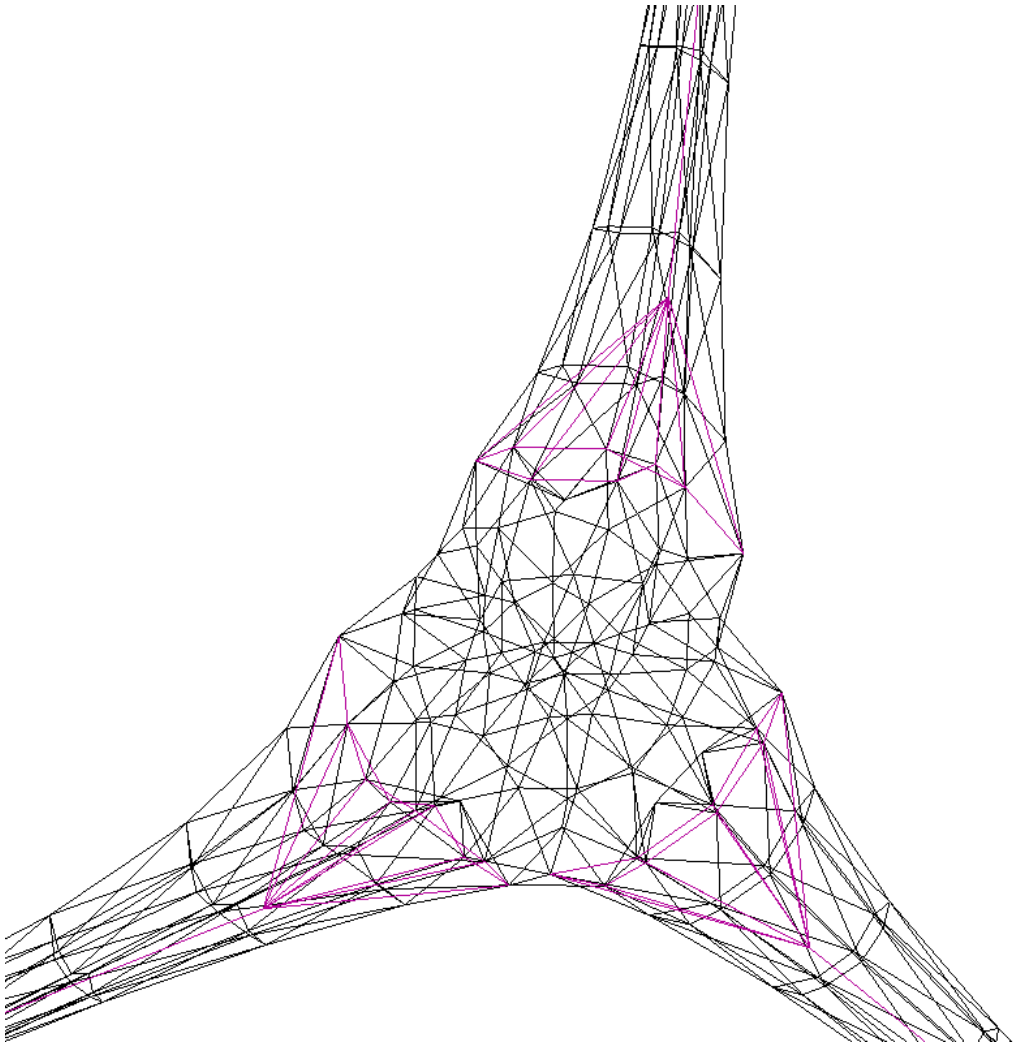

Figure 5.6: Eroding the central sphere: final step

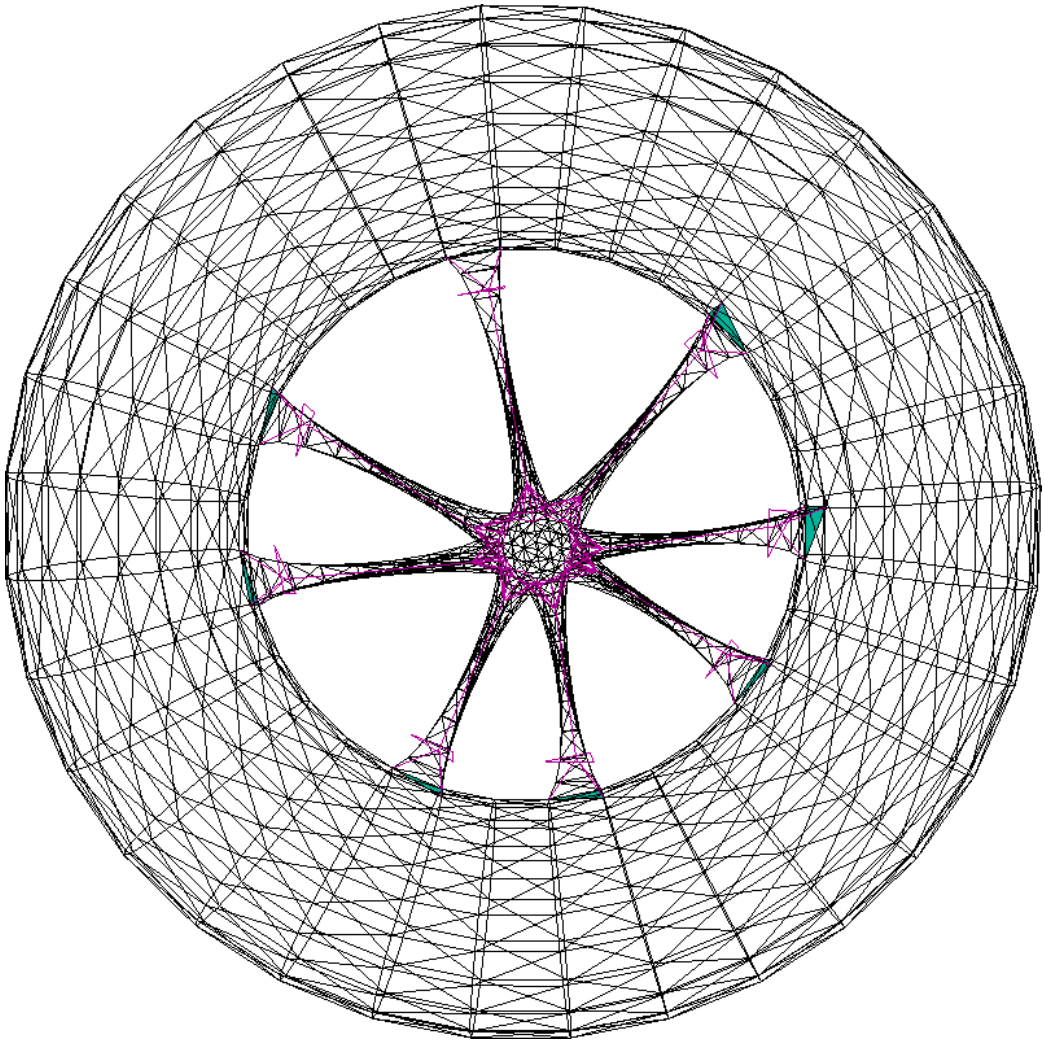

Figure 5.7: An example of connecting 8 blending areas

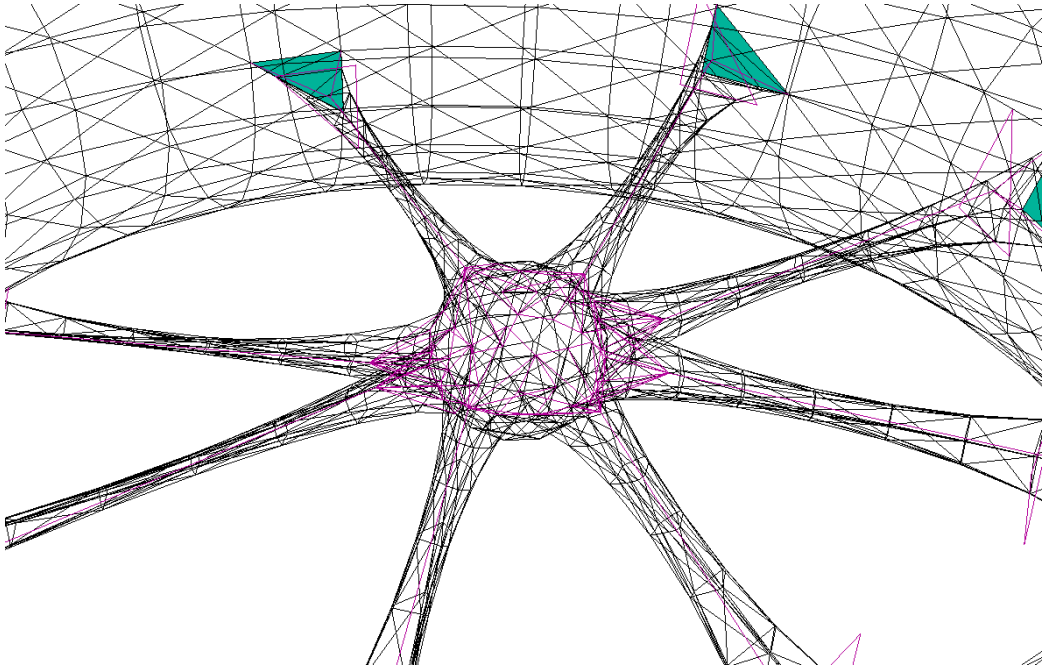

Figure 5.8: An example of connecting 8 blending areas

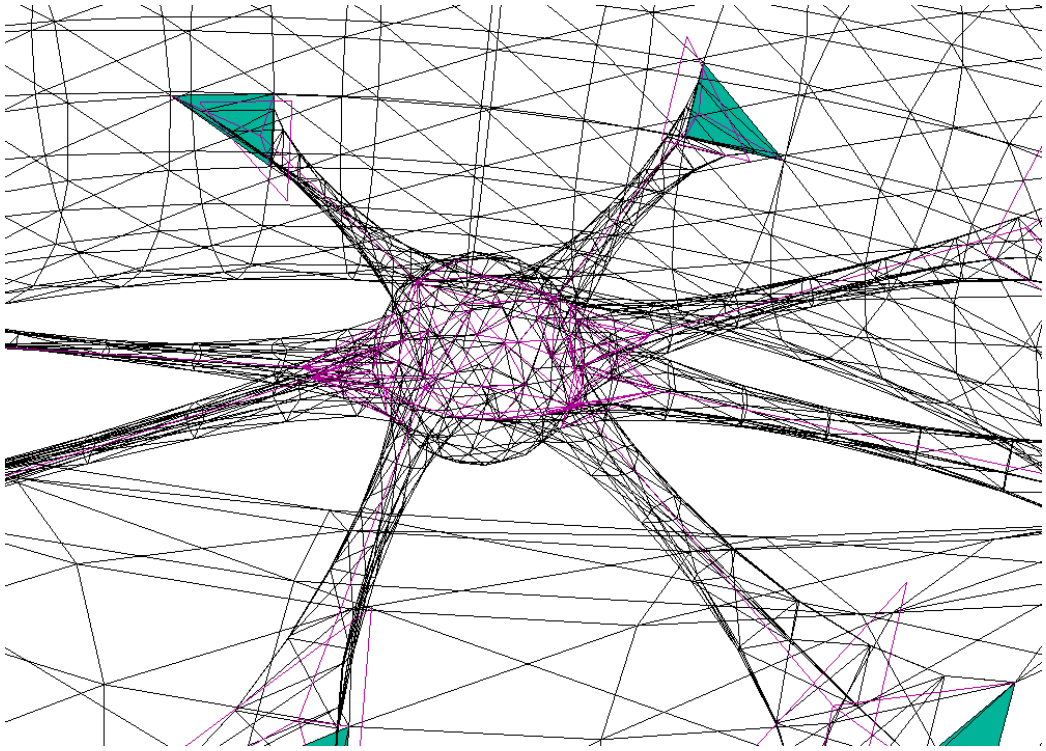

Figure 5.9: An example of connecting 8 blending areas

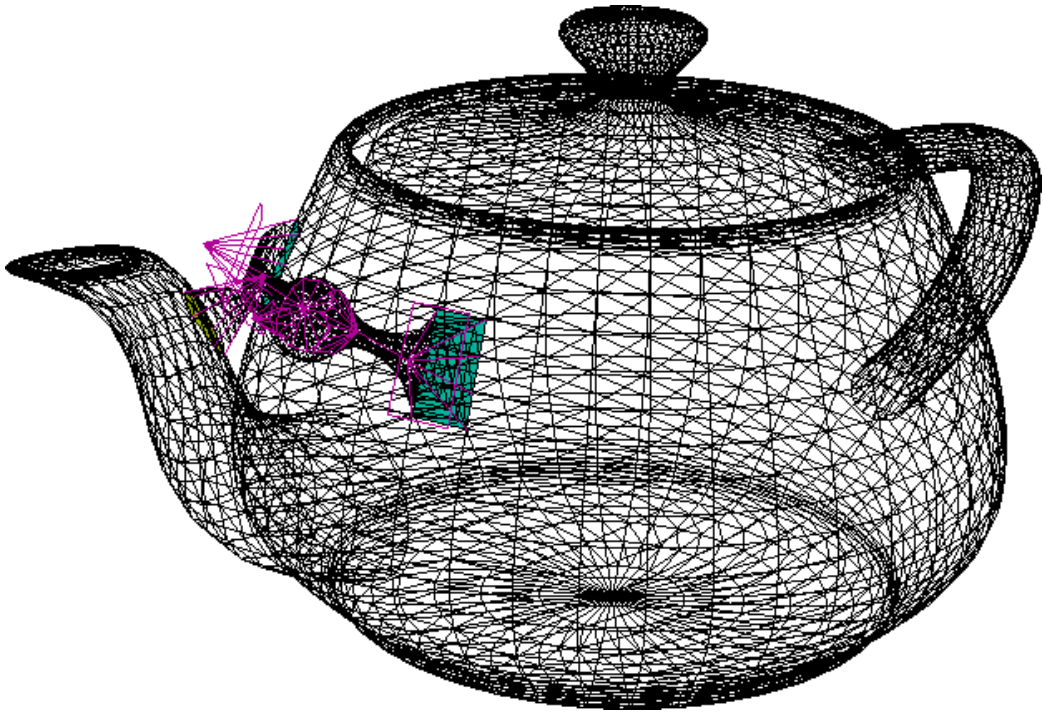

Figure 5.10: The obligatory teapot

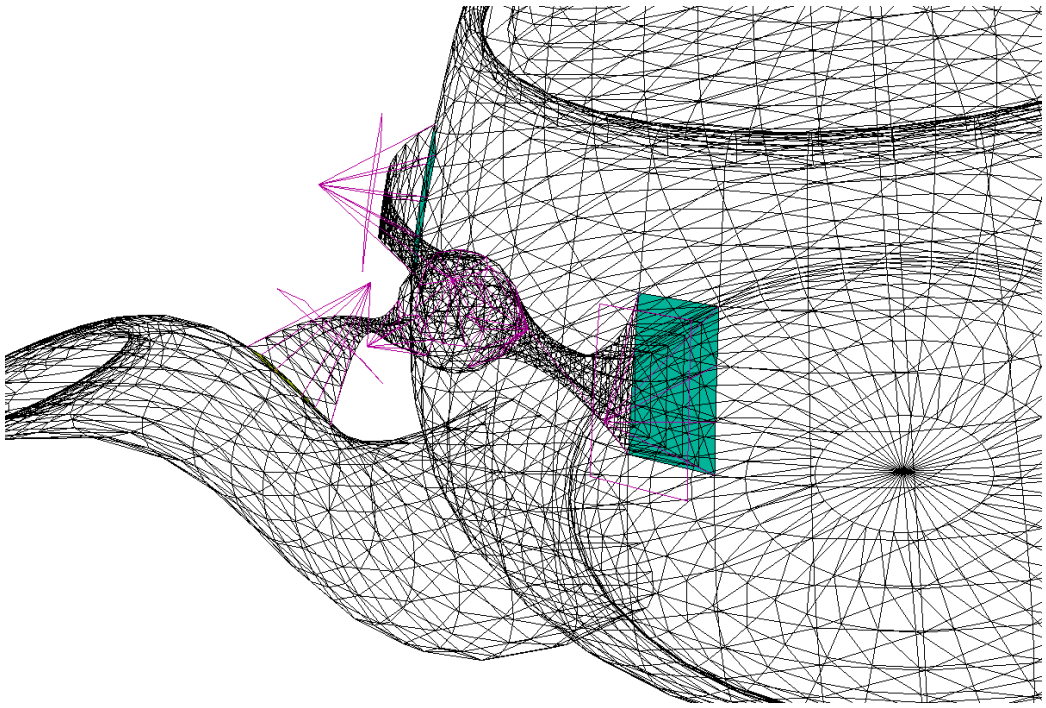

Figure 5.11: The obligatory teapot zoomed

Bibliography

[1] Chalmovianský, P.: *Mathematical Methods in Subdivision Surfaces*, dissertation thesis (2001)

[2] Baumgart,B.G.: *Winged-Edge Polyhedron Representation for Computer Vision*, National Computer Conference (1975)

[3] Eastman,C.M., Weiss,S.F.: *Tree structures for high dimensionality nearest neighbor searching*, Information Systems, vol. 7, no. 2 (1982)

[4] Bourke,P.: *Implicit surfaces*, http://astronomy.swin.edu.au/pbourke/modelling/implicitsurf (1997)

[5] Hart, J.: *Ray Tracing Implicit Surfaces*, WSU Technical Report EECS-93-014 (1993)

[6] Blinn, J.F: *A generalization of algebraic surface drawing*, ACM Transactions on Graphics, (1982)

[7] Nishimura H., Hirai, M., Kawai, T., Kawata, T., Shirakawa, I. and Omura, K., *Object Modelling by Distribution Function and a Method of Image Generation*, The Transactions of the Institute of Electronics and Communication Engineers of Japan (1985)

[8] Wyvill,G., Trotman,A.: *Ray tracing soft objects*, Proceedings of Computer Graphics International (1990)

[9] Farin, G.: *Curves and Surfaces for Computer Aided Geometric Design*, San Diego: Academic Press (1993)

[10] Shoemake, K.: *ARCBALL: A User Interface for Specifying Three-Dimensional Orientation Using a Mouse*, Graphics Interface (1992)

[11] *comp.graphics.algorithms Frequently Asked Questions*, http://isc.faqs.org/faqs/graphics/algorithms-faq/