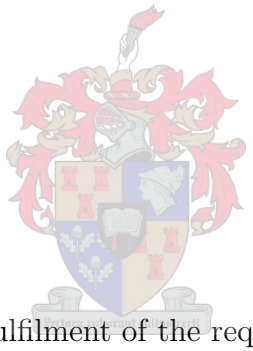
Spherical parameterisation methods for 3D surfaces

Willie Brink



Thesis presented in partial fulfilment of the requirements for the degree of
Master of Engineering Science at the Department of Applied Mathematics of the
University of Stellenbosch

Supervisor: Dr MF Maritz
Co-supervisor: Prof JAC Weideman

December 2005

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work, and that I have not previously in its entirety, or in part, submitted it at any university for a degree.

Signature: . . . . . .

Date: . . . . . .

Abstract

The surface of a 3D model may be digitally represented as a collection of flat polygons in $\mathbb{R}^3$. The collection is known as a polygonal mesh. This representation method has become standard in computer graphics.

Parameterising the surface of such a 3D model is an important phase in numerous applications, including filtering, compression, recognition, texture mapping and morphing.

In this thesis we consider surfaces that are topologically equivalent to a sphere. A natural parameter domain for these surfaces is the surface of the unit sphere. A continuous one-to-one mapping between the surface of a 3D model and the surface of a sphere is known as spherical parameterisation.

Some existing methods of spherical parameterisation are discussed. A novel method that is both theoretically sound and numerically efficient is introduced. We call this method the θ–ϕ method.

The idea behind the θ–ϕ method is to cut a given mesh open along a specific line on the surface. The "open" mesh is embedded in the 2D plane, and then folded onto the surface of the unit sphere, yielding a spherical parameterisation.

Results from applying the different methods are given, and briefly analysed. The θ–ϕ method is then compared to the existing methods.

A short description of some of the applications mentioned above is also given.

Opsomming

Die oppervlak van 'n 3D model kan digitaal voorgestel word as 'n versameling van veelhoeke in $\mathbb{R}^3$. Die versameling staan bekend as 'n veelhoekverbinding. Hierdie voorstellingsmetode is die standaard in rekenaargrafika.

Die parameterisering van die oppervlak van so 'n 3D model is 'n belangrike fase in verskeie toepassings, soos filtrering, kompaktering, herkenning, tekstuurafbeelding en morfering.

In hierdie tesis oorweeg ons oppervlakke wat topologies ekwivalent aan 'n sfeer is. 'n Natuurlike parametergebied vir hierdie oppervlakke is die oppervlak van die eenheidsfeer. 'n Kontinue een-tot-een afbeelding tussen die oppervlak van 'n 3D model en die oppervlak van 'n sfeer staan bekend as sferiese parameterisering.

Enkele bestaande metodes van sferiese parameterisering word bespreek. 'n Nuwe metode wat teoreties begrond en numeries doeltreffend is, word voorgestel. Ons noem hierdie metode die θ–ϕ metode.

Die idee agter die θ–ϕ metode is om 'n gegewe veelhoekverbinding langs 'n spesifieke lyn op die oppervlak oop te sny. Hierdie "oop" veelhoekverbinding word in die 2D vlak ingebed, en dan oor die oppervlak van die eenheidsfeer gevou. Sodoende word 'n sferiese parameterisering verkry.

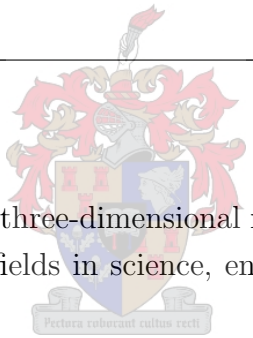
Resultate van die toepassing van die verskillende metodes word gegee en kortliks analiseer. Die θ–ϕ metode word dan met die bestaande metodes vergelyk.

'n Kort beskrywing van sommige van die bogenoemde toepassings word ook gegee.

Contents

CHAPTER 1

Introduction



Research in the digital processing of three-dimensional models has increased tremendously over the last two decades. Various fields in science, engineering, medicine and entertainment benefit from this research.

The surface of a 3D model may be described by a collection of adjacent flat polygons. Such a collection is known as a mesh. Meshes have become a standard respresentation method for 3D models, due to their simplicity, flexibility, and the fact that they are widely supported by computer graphics software and hardware.

Parameterising the surface of such a mesh is a central issue in computer graphics. A parameterisation allows complex operations to be performed directly on the parameter domain, rather than on the surface of a mesh, which has an arbitrary shape. These operations include remeshing, surface filtering, compression, recognition, morphing, and texture mapping, all of which are important applications in computer graphics.

Parameterisation amounts to establishing a one-to-one correspondence between the surface of a mesh and some parameter domain.

Only surfaces topologically equivalent to a sphere will be considered in this thesis. A natural parameter domain for these surfaces is the surface of the unit sphere (see for example Figure 1.1).

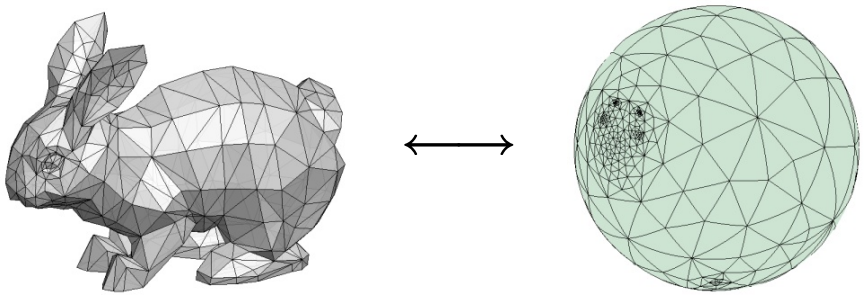


Figure 1.1

To date, the problem of parameterising the surface of an arbitrary mesh remains challenging. Various methods have been proposed [1, 3, 9, 16, 19, 21, 29, 34, 35]. These methods have limitations, however, and there is still a need for a robust and efficient method.

There are two major classes of parameterisation methods. The first of these rely on positioning vertices on the sphere at convex combinations of neighbouring vertices [1, 3, 9, 16]. We call these methods the *convex combination methods*. The second class of methods is those that simplify the mesh to a convex mesh by removing vertices, and then replacing the vertices in such a way that the mesh remains convex [19, 29, 35]. These methods are known as *progressive mesh methods*.

This thesis is devoted to the convex combination methods only. Some existing methods are discussed, and a novel method is presented. This new method is compared to the existing methods with the aid of some experimental results.

The rest of the thesis is organised as follows.

- Chapter 2 introduces some general definitions and notation for meshes. It also shows how meshes are related to graphs.
- Chapter 3 formally defines the problem of spherical parameterisation, and discusses the standard procedure towards finding solutions.
- Chapter 4 provides some theory on convex combinations. A method that implements convex combinations for embedding planar graphs in the 2D plane is discussed.
- Chapter 5 shows how a number of existing methods extend the 2D planar embedding algorithm to the surface of the sphere. The aim of these methods is to solve the problem of parameterisation.

- Chapter 6 explains the new method in detail.
- Chapter 7 gives some experimental results after applying the different methods discussed in the thesis to a number of test models. Measured running times are given and compared.
- Chapter 8 briefly discusses a number of applications for surface parameterisation and illustrates them by means of examples.
- Chapter 9 gives some concluding remarks and possibilities for future research.
- Appendix A discusses an algorithm from graph theory, known as Dijkstra's algorithm, used in various parts of this thesis.
- Appendix B supplies MATLAB code for the different algorithms and methods in this thesis.

CHAPTER 2

Graphs and meshes

Polygonal meshes (or simply "meshes") have become a standard representation method for three-dimensional objects in computer graphics. A mesh is a collection of polygons or "faces" that forms the outer surface of an object or shape in three-dimensional space. Figure 2.1 shows an example of a 3D model (a rabbit) represented as a polygonal mesh.

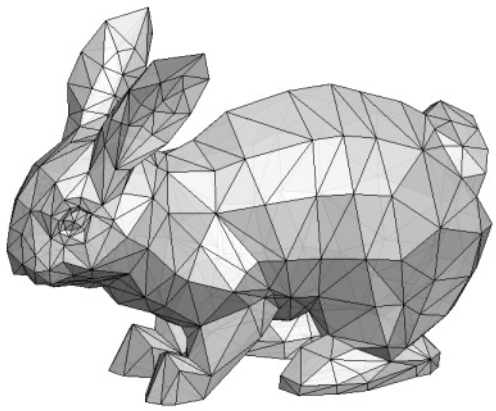


Figure 2.1

Clearly, the collection of distinct vertices of all the faces in a mesh has a connectivity structure — pairs of vertices are connected by means of edges. This structure suggests the use of methods and techniques from the field of graph theory.

This chapter provides some basic definitions from graph theory, and shows more formally how polygonal meshes are related to graphs.

2.1 Definitions from graph theory

In this section, definitions and concepts from graph theory that are needed for the purpose of this thesis are briefly discussed.

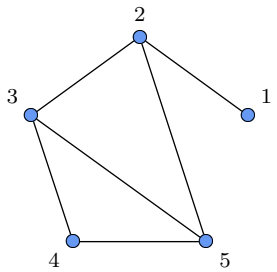
A (simple) *graph* G is uniquely defined by two sets: a *vertex set* $V(G)$, and an *edge set* $E(G)$. Both these sets are finite, and it is assumed that $V(G)$ is non-empty.

The elements of $V(G)$ are called *vertices*. Geometrically, a vertex may be thought of as a point. Each vertex in $V(G)$ can be labeled, and we follow the convention that the vertices are labeled with the integers from 1 to $|V(G)|$, where $|A|$ denotes the cardinality of a set A. We denote the vertex with label i simply by the number i. Therefore, for a graph G with n vertices, we follow the convention that

$$V(G) = \{1, 2, \ldots, n\}. \tag{2.1}$$

The elements of $E(G)$ are called *edges*. An edge may be thought of as a curve joining two vertices. Each edge is an unordered pair of the form (i, j), where $i, j \in V(G)$.

Figure 2.2 shows the graphical representation of a graph with its corresponding vertex set and edge set. The graph has 5 vertices and 6 edges.



$V(G) = \{1, 2, 3, 4, 5\}$
$E(G) = \{(1, 2), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5)\}$

Figure 2.2

Two vertices i and j in $V(G)$ are said to be *adjacent* in G if $(i, j) \in E(G)$. Also, we say i is a *neighbour* of j, and vice versa. The edge (i, j) is said to be an adjacent edge of both i and j.

The *neighbourhood* of a vertex $i \in V(G)$ is the set of vertices

$$N(i) \;=\; \{j \in V(G) : (i, j) \in E(G)\}, \tag{2.2}$$

in other words, the set of all the vertices adjacent to i.

The *degree* of a vertex $i \in V(G)$, denoted by $\deg(i)$, is defined as

$$\deg(i) = |N(i)|, \tag{2.3}$$

that is, the number of vertices adjacent to i.

Two graphs G_1 and G_2 are said to be *isomorphic* if there exists a bijective mapping $f : V(G_1) \mapsto V(G_2)$ such that the edge (i, j) is an element of $E(G_1)$ if and only if $(f(i), f(j))$ is an element of $E(G_2)$. If such a mapping exists, the function f is an *isomorphism* between G_1 and G_2. Figure 2.3 gives an example of two isomorphic graphs.

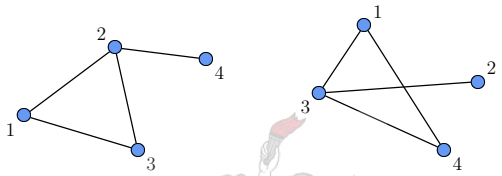


Figure 2.3

Each edge in the graph G may be associated with a *weight*. It is useful to assign zero weights to any pair of non-adjacent vertices. We therefore define the weights of a graph as

$$\rho_{ij} \begin{cases} > 0, & (i, j) \in E(G), \\ = 0, & (i, j) \notin E(G), \end{cases} \qquad i, j = 1, 2, \ldots, |V(G)|. \tag{2.4}$$

A *walk* in a graph G is a sequence $W = (v_1, v_2, \ldots, v_n)$ of vertices such that every pair of consecutive vertices in W are adjacent in G. The walk W may also be referred to as a v_1–v_n walk in G. The *length* of the walk W is denoted by $\ell(W)$, and is defined as

$$\ell(W) = \sum_{i=1}^{n-1} \rho_{v_i v_{i+1}}. \tag{2.5}$$

Vertices and edges may appear more than once in a walk. A *path* in a graph G is a walk in which none of the vertices are repeated. Every vertex in the graph appears either once or never in a path. A *cycle* is a walk with at least 4 vertices in which the first vertex is the same as the last one, but in which no other vertices are repeated. A cycle with n edges is called an n–cycle.

The *distance* between two vertices i and j in a graph G is defined as the length of a shortest possible i–j path, and is denoted by $d(i, j)$. If G does not contain an i–j path, we define $d(i, j) = \infty$.

A vertex i in a graph G is said to be *connected* to a vertex j if the graph contains an i–j path. A graph is *connected* if i is connected to j for all possible (i, j) pairs in the graph. A graph that is not connected is called disconnected.

We define a *vertex removal* from a graph G to be the result after a vertex $v \in V(G)$, together with all its adjacent edges, is removed from G. This resulting graph is denoted by $G - v$. Thus, if $H = G - v$, then

$$V(H) = V(G) \setminus v, \tag{2.6}$$

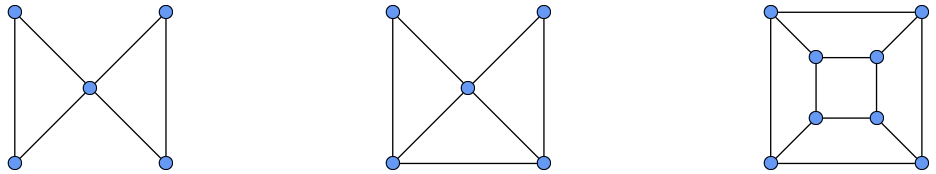
and

$$E(H) = \{(i, j) \in E(G) : v \notin \{i, j\}\}. \tag{2.7}$$

For a graph G and $S \subset V(G)$, we write $G - S$ to denote the resulting graph after every vertex in S has been removed from G.

A graph G is said to be k–connected, with k a positive integer, if it is possible to remove any set of $k - 1$ vertices from G, such that the resulting graph remains connected. We require that a k–connected graph have at least $k+1$ vertices. By definition, any connected graph is at least 1–connected.

Figure 2.4 shows three graphs. The graph in (a) is 1–connected, but not 2–connected. The graph in (b) is 2–connected, but not 3–connected. The graph in (c) is 3–connected, but not 4–connected.



(a) 1–connected (b) 2–connected (c) 3–connected

Figure 2.4

A *planar* graph is a graph that can be drawn in the two-dimensional plane such that none of its edges intersect. Such a drawing of a planar graph is called a planar embedding of the graph. Chapter 4 deals with finding planar embeddings. The graphs in Figures 2.2, 2.3 and 2.4 are examples of planar graphs.

2.2 Polygonal meshes

This section defines the concept of a mesh and gives some properties of meshes. Two specific classes of meshes are also introduced. In the chapters that follow we will only be interested in meshes from these classes.

2.2.1 Definition of a mesh

A *mesh* M is defined by the following two structures: a *coordinate set* $X(M)$ and a *face set* $F(M)$.

The elements of the coordinate set $X(M)$ are called *nodes*. Suppose a given mesh has n nodes. The nodes are points in $\mathbb{R}^3$, labeled with the integers from 1 to n. Therefore

$$X(M) = \{\mathbf{x}_i \in \mathbb{R}^3,\ i \in \{1, 2, \ldots, n\}\}, \tag{2.8}$$

where $\mathbf{x}_i$ denotes the coordinates of node i. It is assumed that the coordinates of the nodes are distinct, therefore $\mathbf{x}_i \neq \mathbf{x}_j$, for any $i \neq j$.

The elements of the face set $F(M)$ are called *faces*. Every face defines a planar polygon in $\mathbb{R}^3$. Suppose a given mesh has m faces. The faces are labeled with the integers 1 to m, with f_i denoting the face with label i. A face $f_i \in F(M)$ is an ordered tuple

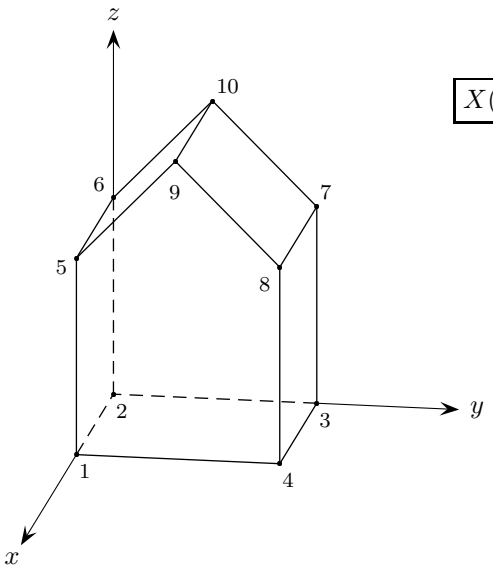
$$f_i = (f_{i,1}, f_{i,2}, \ldots, f_{i,k(i)}), \tag{2.9}$$

with $k(i) \geq 3$ and $f_{i,j} \in \{1, 2, \ldots, n\}$, $j = 1, 2, \ldots, k(i)$, such that the points $\mathbf{x}_j$, $j \in f_i$ are co-planar. The elements $f_{i,j}$ correspond to the vertices of the polygon f_i. The number $k(i)$ therefore denotes the number of vertices, or equivalently the number of sides, of the polygon f_i. We refer to $k(i)$ as the *degree* of the face f_i, and denote it by $\deg(f_i)$.

Figure 2.5 shows an example of a mesh M, with given coordinate set and face set. The mesh has 10 nodes and 7 faces.

The convention for the order of traversing the vertices for each face is counterclockwise as seen from outside the mesh. Note that the example in Figure 2.5 follows this convention.

A mesh in which all the faces have the same degree is called a *uniform* mesh. In particular, if all the faces of a mesh have degree 3, the mesh is said to be *triangular*.



$X(M)$

i	$\mathbf{x}_i$
1	$[2,0,0]$
2	$[0,0,0]$
3	$[0,2,0]$
4	$[2,2,0]$
5	$[2,0,2]$
6	$[0,0,2]$
7	$[0,2,2]$
8	$[2,2,2]$
9	$[2,1,3]$
10	$[0,1,3]$

$F(M)$

i	f_i
1	$(1,2,3,4)$
2	$(1,5,6,2)$
3	$(3,7,8,4)$
4	$(5,9,10,6)$
5	$(7,10,9,8)$
6	$(1,4,8,9,5)$
7	$(2,6,10,7,3)$

Figure 2.5

2.2.2 The underlying graph of a mesh

Since the sides of every face in a mesh M may be thought of as edges connecting the nodes of M, a relationship between meshes and graphs exists. This relationship is explained next.

Every mesh M is associated with a graph, called its *underlying graph*. The graph is denoted by G_M, and is sometimes referred to as the node-neighbourhood graph, or the skeleton of the mesh.

Figure 2.6 shows a graphical representation of the underlying graph of the mesh from Figure 2.5.

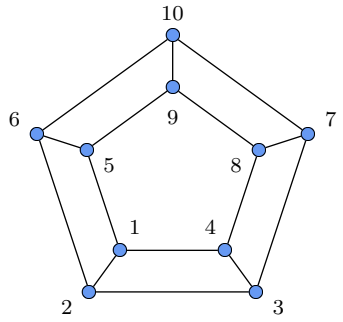


Figure 2.6

The graph G_M is constructed as follows. Every vertex in G_M corresponds to a node of the

mesh M, and every edge in G_M corresponds to a side of one or more of the faces of M.

In order to obtain the edge set of G_M, we define the set $E(f_i)$ to be the set of sides (or edges) of the face $f_i \in F(M)$. Therefore,

$$E(f_i) = \{(f_{i,j}, f_{i,j+1}),\ j = 1, 2, \ldots, \deg(f_i) - 1\} \cup \{(f_{i,\deg(f_i)}, f_{i,1})\}. \tag{2.10}$$

Using also (2.1), a formal definition of the graph G_M follows.

Definition 2.1 : *For a mesh M with n nodes and m faces, the unique underlying graph G_M is defined by*

$$V(G_M) = \{1, 2, \ldots, n\}, \quad \textit{and} \quad E(G_M) = \bigcup_{i=1}^{m} E(f_i),$$

where $E(f_i)$ is given by (2.10).

Definition 2.1 can be used to obtain the graph G_M from the sets $X(M)$ and $F(M)$. Note that there exist at most one edge between any pair of vertices in G_M. Faces of M may therefore share the same edge in G_M.

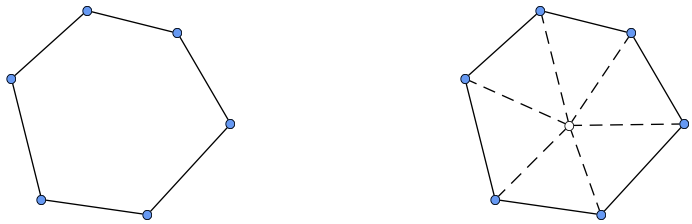
From Figure 2.6, it is clear that every face $f_i \in F(M)$ corresponds to a cycle of length $\deg(f_i)$ in the graph G_M.

2.2.3 Mesh triangulation

In the chapters that follow, only triangular meshes will be considered. This is not a major restriction, since any polygon can be divided into a number of triangles. For a general mesh with non-triangular faces, a triangulation algorithm may be applied as part of a preprocessing phase.

There is a number of different ways to triangulate a given polygon. A simple method of triangulating a convex polygon is illustrated in Figure 2.7. A new vertex is created in the centre of the polygon and connected to the original vertices as shown.

Note that for some concave polygons the new vertex may lie outside of the polygon. This problem may be solved by first partitioning the polygon into a number of convex polygons. In practice, however, very few cases arise where concave polygonal faces occur in meshes.



(a) non-triangular (b) triangulated

Figure 2.7

De Berg, et. al. [6] devote an entire chapter to polygon triangulation, and provide some interesting algorithms and proofs.

2.2.4 The surface of a mesh

The main focus of this thesis is to parameterise the surface of a given mesh. This section provides the formal definition of, and a formula for, the surface of a triangular mesh.

The *surface* of a mesh M, denoted by S_M, is defined to be the set of all points in $\mathbb{R}^3$ that lie in one or more of the polygonal faces of M.

For a triangular mesh M with m faces, let $T(f_i) \subset \mathbb{R}^3$ denote the set of points inside or on the boundary of the triangular face f_i. Then

$$S_M = \bigcup_{i=1}^{m} T(f_i). \tag{2.11}$$

Next we derive an explicit formula for describing $T(f_i)$ in terms of the coordinates of the vertices of f_i.

Figure 2.8 shows a triangular face f_i. Let $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$ denote the vectors from the origin to the three vertices of f_i.

Let $\mathbf{d} \in \mathbb{R}^3$ be the vector from the origin to an arbitrary point on the line segment between $\mathbf{a}$ and $\mathbf{b}$, as shown in Figure 2.8. Then

$$\mathbf{d} = \lambda\mathbf{a} + (1-\lambda)\mathbf{b}, \quad \lambda \in [0,1]. \tag{2.12}$$

Let $\mathbf{v} \in \mathbb{R}^3$ be the vector from the origin to an arbitrary point on the line segment joining $\mathbf{c}$ and $\mathbf{d}$. Therefore

$$\mathbf{v} = \mu\mathbf{d} + (1-\mu)\mathbf{c} = \lambda\mu\mathbf{a} + (1-\lambda)\mu\mathbf{b} + (1-\mu)\mathbf{c}, \tag{2.13}$$

with $\mu \in [0,1]$.

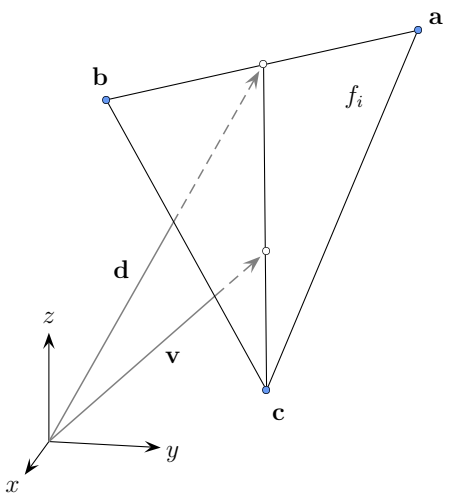


Figure 2.8

Varying λ and μ between 0 and 1 in (2.13) results in all possible points inside or on the boundary of f_i. The set $T(f_i)$ may therefore be expressed as

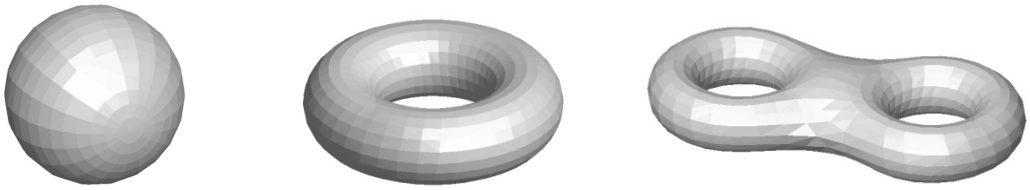
$$T(f_i) = \{\lambda\mu\mathbf{x}_{f_{i,1}} + (1-\lambda)\mu\mathbf{x}_{f_{i,2}} + (1-\mu)\mathbf{x}_{f_{i,3}} \in \mathbb{R}^3 : \lambda, \mu \in [0,1]\}, \quad f_i \in F(M). \quad (2.14)$$

Equation (2.14) is used in equation (2.11) to obtain an explicit formula for the surface of the mesh M.

2.2.5 Homeomorphism and 2–manifoldness

This section deals with some topological features of meshes.

A *homeomorhism* is a one-to-one (or bijective) mapping $h : U \to V$ between the points of two geometric objects U and V that is continuous in both directions, i.e. h and h^{-1} are both continuous. If such a mapping exists, it is said that U and V are homeomorphic (or topologically equivalent).



(a) sphere (b) torus (c) figure eight

Figure 2.9

None of the surfaces of the meshes in Figure 2.9 are homeomorphic to each other, since continuous mappings between the three different sets of points cannot be found. The mesh from Figure 2.5, however, is homeomorphic to the mesh in Figure 2.9(a).

The following two geometric objects will be used throughout the rest of this thesis. The *surface of the unit sphere*, denoted by S_0, is defined to be the continuous set of points given by

$$S_0 = \{\mathbf{x} \in \mathbb{R}^3 : ||\mathbf{x}|| = 1\}. \tag{2.15}$$

The *closed disc*, denoted by D_0, is defined to be the continuous set of points given by

$$D_0 = \{\mathbf{x} \in \mathbb{R}^2 : ||\mathbf{x}|| \leq 1\}. \tag{2.16}$$

In the rest of this thesis we will be concerned with meshes with surfaces homeomorphic to either S_0 or D_0. This motivates the need for the introduction of the following two classes of meshes.

Definition 2.2 : *We write $\mathcal{M}$ to denote the set of meshes such that a mesh M belongs to $\mathcal{M}$ if and only if:*

(1) M is triangular, and

(2) S_M is homeomorphic to S_0,

with S_M defined as in (2.11) and S_0 as in (2.15).

It turns out that the mesh from Figure 2.1 is an example of a mesh in $\mathcal{M}$. The mesh from Figure 2.5, however, is not, since it is not triangular. The mesh from Figure 2.9(b) is not in $\mathcal{M}$, since its surface is not homeomorphic to S_0.

Definition 2.3 : *We write $\mathcal{M}_B$ to denote the set of meshes such that a mesh M belongs to $\mathcal{M}_B$ if and only if:*

(1) M is triangular, and

(2) S_M is homeomorphic to D_0,

with S_M defined as in (2.11) and D_0 as in (2.16).

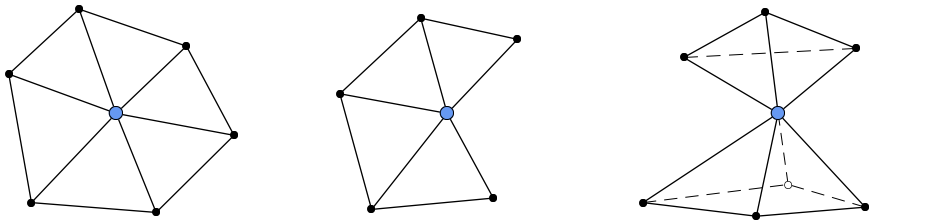
The subscript B in the notation $\mathcal{M}_B$ emphasises the fact that meshes in this class have boundaries. These boundaries are discussed next, with some preliminary definitions.

The *star* of a vertex i, denoted by $\text{star}(i)$, is defined as the set of all points on the faces containing i. Hence

$$\text{star}(i) = \{T(f) : f \in F(M),\ i \in f\}, \quad i \in V(G_M), \tag{2.17}$$

with $T(f)$ defined as in (2.14).

A vertex i is called *2–manifold* if (a) the number of faces sharing i is equal to $\deg(i)$, and (b) $\text{star}(i)$ is homeomorphic to the disc D_0 as defined in (2.16). A vertex i is said to be a *boundary vertex* if condition (b) is satisfied, but instead of (a), the number of faces sharing i is equal to $\deg(i) - 1$. Figure 2.10 shows a few vertices, each with its star, and whether it is 2–manifold, boundary, or neither.



(a) 2–manifold vertex (b) boundary vertex (c) non–manifold vertex

Figure 2.10

A mesh M is called *2–manifold* if every vertex $i \in V(G_M)$ is a 2–manifold vertex. Clearly, all the meshes in $\mathcal{M}$ are 2–manifold. But note that a mesh such as the one in Figure 2.9(b) is 2–manifold, but does not belong to the class $\mathcal{M}$.

For an arbitrary mesh M, any cycle of boundary vertices in G_M is refered to as a *boundary* of M. In particular, a mesh in $\mathcal{M}$ has no boundaries, while a mesh in $\mathcal{M}_B$ has exactly one boundary (hence the subscript B). We use the symbol B to denote the boundary of a mesh $M \in \mathcal{M}_B$.

Figure 2.11 shows a mesh belonging to the class $\mathcal{M}_B$, from two different viewpoints. The boundary is clearly visible.

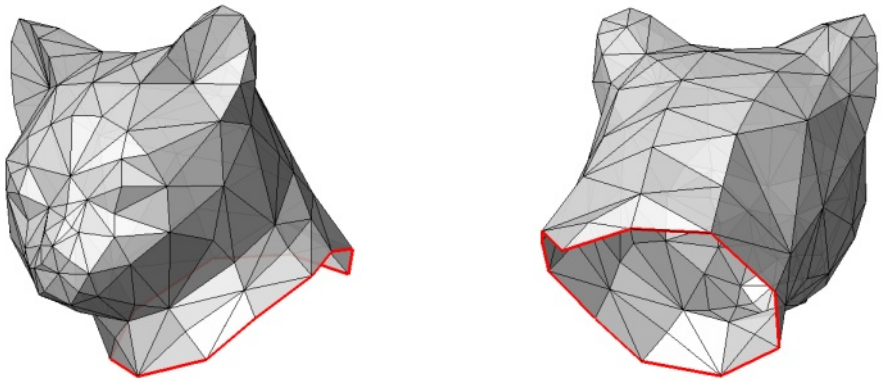


Figure 2.11

Another consequence of Definitions 2.2 and 2.3 is the fact that the underlying graph of a mesh in either $\mathcal{M}$ or $\mathcal{M}_B$ is necessarily connected. If this were not the case, a continuous mapping could not be established from the surface of the mesh to either S_0 or D_0.

We conclude this section on polygonal meshes by discussing two well-known results, namely Steinitz's theorem and Euler's formula.

2.2.6 Steinitz's theorem

This section deals with a result due to Ernst Steinitz [31], that relates some properties of a mesh to that of its underlying graph.

The result, originally published in 1934, is reformulated here in the context and notation of this chapter. The proof is not given, and may for example be found in [36, p. 103].

Theorem 2.4 : (Steinitz's theorem) *A simple graph G is the underlying graph of a mesh M, where $S(M)$ is homeomorphic to S_0, if and only if G is planar and 3–connected.*

The theorem translates to the following. If a mesh M has a surface homeomorphic to S_0, then the underlying graph of M is necessarily planar and 3–connected. Conversely, given a planar, 3–connected graph G, a mesh M can be constructed such that G is equal to the underlying graph of M, and such that the surface of M is homeomorphic to S_0.

Note that this theorem holds for meshes in the class $\mathcal{M}$.

The surface of a mesh $M \in \mathcal{M}_B$ is homeomorphic to the disc, therefore the graph G_M can be embedded in the 2D plane without edge intersections, implying that G_M is planar.

We have thus established the following important result:

Corollary 2.5 : *The underlying graph of any mesh in either $\mathcal{M}$ or $\mathcal{M}_B$ is planar.*

2.2.7 Euler's formula

This section deals with an equality known as Euler's formula, that relates the number of vertices, edges and faces of certain types of meshes.

Euler discovered this formula around 1750. A proof may be found in for example [5, p. 189].

Theorem 2.6 : (Euler's formula) *For any mesh M, with $S(M)$ homeomorphic to S_0, it holds that*

$$v - e + f = 2,$$

where v denotes the number of vertices, e the number of edges, and f the number of faces in M.

Note that the formula holds for meshes in $\mathcal{M}$. We aim to establish a similar result for meshes in $\mathcal{M}_B$.

Consider a mesh in $M \in \mathcal{M}_B$. Since the edges in the boundary B of M corresponds to the edges of a simple polygon, it is possible to reposition all the boundary vertices of M to the same plane in $\mathbb{R}^3$. It is then possible to add a face to M, with vertices exactly those of B. This results in a mesh from the class $\mathcal{M}$.

Therefore, if follows from Theorem 2.6 that

$$v - e + (f + 1) = 2, \tag{2.18}$$

where v, e and f denote the number of vertices, edges and faces in the original mesh M (before the new face is added). We have thus established the following important result.

Corollary 2.7 : *Consider a mesh M with v vertices, e edges and f faces. The following hold:*

(a) if $M \in \mathcal{M}$, then $v - e + f = 2$,
(b) if $M \in \mathcal{M}_B$, then $v - e + f = 1$.

The mesh in Figure 2.1 belongs to $\mathcal{M}$. For this example, $v = 453$, $f = 902$, $e = 1353$, and since $453 - 1353 + 902 = 2$, the formula holds. The mesh in Figure 2.11 belongs to $\mathcal{M}_B$, and for that example $v = 135$, $f = 257$ and $e = 391$, and we have $135 - 391 + 257 = 1$.

Consider a mesh $M \in \mathcal{M}$, with v vertices, f faces, and e edges. Since every face in M has degree 3, and every edge is shared by 2 faces, it follows that $e = \frac{3}{2}f$. Combining this with Euler's formula yields

$$f = 2(v - 2), \quad \text{and} \quad e = 3(v - 2). \tag{2.19}$$

Therefore, if the number of vertices in an arbitrary mesh in $\mathcal{M}$ is known, then (2.19) may be used to determine how many faces and edges the mesh necessarily has.

Table 2.1 lists the number of meshes in $\mathcal{M}$ for specific numbers of vertices. Note that this table counts the number of meshes where the underlying graphs differ up to isomorphism, i.e., regardless of different labelling or positioning of nodes. The numbers in the last column are taken from [25].

For obvious reasons, there are no meshes in $\mathcal{M}$ with fewer than 3 vertices. The unique mesh with 4 vertices is the tetrahedron (a pyramid with triangular base).

Vertices	Faces	Edges	Number of meshes in $\mathcal{M}$
4	4	6	1
5	6	9	1
6	8	12	2
7	10	15	5
8	12	18	14
9	14	21	50
10	16	24	233
11	18	27	1,249
12	20	30	7,595
13	22	33	49,566
14	24	36	339,722
15	26	39	2,406,841

Table 2.1

There is a well-known extension to Euler's formula. For any 2–manifold mesh, it holds that

$$v - e + f = 2(c - g), \tag{2.20}$$

where v, e and f denote the number of vertices, edges and faces respectively. The integer c denotes the number of connected components, and g denotes the so-called genus of the mesh, which is defined next.

The *genus* of a connected 2–manifold mesh M is defined to be the maximum number of cuttings along closed simple curves on the surface of M without rendering the resultant mesh disconnected. The genus of a mesh with surface homeomorphic to S_0, is 0. The mesh in Figure 2.9(b) has genus 1, and the one in Figure 2.9(c) has genus 2.

2.3 The validity of a mesh

This section discusses some techniques for examining the properties of a given mesh, for testing whether a given mesh belongs to the class $\mathcal{M}$.

The following result gives three conditions which must be satisfied for the surface of a mesh to be homeomorphic to the sphere.

Proposition 2.8 : *If a mesh M, with v vertices, e edges and f faces, satisfies the following three conditions,*

(1) M is 2–manifold,

(2) G_M is a connected graph, and

(3) $v - e + f = 2$,

then S_M is homeomorphic to S_0.

Proof

If M is 2–manifold, then equation (2.20) holds. If G_M is connected, then $c = 1$. It then follows from (2.20) that

$$v - e + f = 2 - 2g, \tag{2.21}$$

with g the genus of the mesh. If condition (3) is satisfied, then $g = 0$, implying that $S(M)$ is homeomorphic to S_0. ■

Proposition 2.8 implies that if a given mesh M satisfies the three conditions listed, and M is triangular, then $M \in \mathcal{M}$. Techniques to test those conditions will now be discussed. Consider a given mesh M.

The condition of triangularity, as well as condition (3), are easily tested.

Next, a method for testing the condition of 2–manifoldness is discussed. From the definition of 2–manifoldness, every vertex of M must be a 2–manifold vertex. This means that for every $i \in V(G_M)$, (a) the number of faces sharing i must equal $\deg(i)$, and (b) $\text{star}(i)$ must be homeomorphic to the disc.

We define the *face neighbourhood* of a vertex i, denoted by $N_f(i)$, to be the set of faces sharing i. Thus

$$N_f(i) = \{f \in F(M) : i \in f\}. \tag{2.22}$$

Now, condition (a) holds if

$$|N_f(i)| = \deg(i), \quad i \in V(G_M). \tag{2.23}$$

Condition (b) is tested as follows. If condition (a) is satisfied, and all the neighbours of i lie on a single cycle of length $\deg(i)$ in the graph G_M, then condition (b) is satisfied. See for example Figure 2.10(a).

If the two conditions for 2–manifoldness hold for every vertex in M, then M is 2–manifold.

A technique to test the connectivity of the graph G_M of a mesh M will now be discussed. Recall that a graph G is said to be connected if there exists a path in G between every possible pair of vertices.

The following recursive algorithm is used. We start with an arbitrary vertex $i \in V(G_M)$. Let D be the set $\{i\}$. Every neighbour of i is inserted into D. Then every neighbour of every vertex $j \in D$ is inserted. This process is continued until no more vertices are inserted into D.

Clearly, if after the execution of this algorithm D is equal to $V(G_M)$, then G_M is a connected graph.

2.4 Orientation of faces

The last section in this chapter discusses a method for testing, and correcting, the ordering of vertices in every face of a given mesh in $\mathcal{M}$.

Recall from the definition of a mesh that the following convention is assumed to be implemented. The vertices of every face are ordered to be counterclockwise, as seen from outside the mesh. In some of the arguments in subsequent chapters this assumption is crucial.

If the vertices of a face f are ordered according to this convention, we say that f is *orientated correctly*. If not, then f is orientated incorrectly.

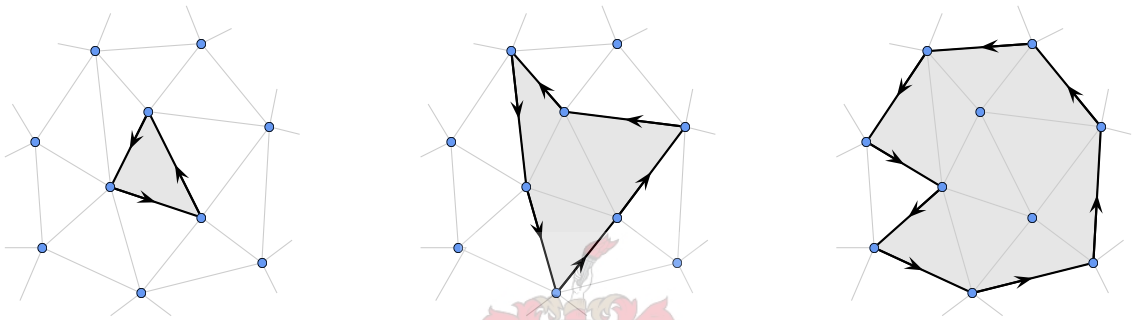
An algorithm has been developed for this purpose. Before a formal description of this algorithm is given, we illustrate it with an example. Consider a mesh $M \in \mathcal{M}$.

The algorithm assumes that one face, say $f^* \in F(M)$, is known to be orientated correctly. It then runs through every face $f \in F(M) \setminus \{f^*\}$, checking that f is orientated correctly, and if not, correcting the orientation of f by reversing the ordering of its vertices.

Figure 2.12 gives an illustration of the first few steps of this algorithm. It shows some vertices and edges of a mesh M. Assume the face $f^* \in F(M)$ is known to be correctly orientated. The grey face in Figure 2.12(a) depicts the face f^*. The arrows on the edges of f^* denote the order in which the vertices of f^* are traversed in the face set.

Every face sharing an edge with f^* is tested next. Since it is known that the orientation of f^* is correct, these faces are easily tested, and corrected if necessary. Figure 2.12(b) shows the face f^*, as well as the three faces sharing edges with f^*, in grey. Again, the arrows on the edges denote the order in which vertices of grey faces are traversed.

Figure 2.12(c) shows the result after another step. The grey faces indicate all the faces that have already been checked, and the arrows indicate the direction of vertex traversal in these faces.



(a) (b) (c)

Figure 2.12

This process is continued until every face in $F(M)$ has been checked, resulting in an updated face set, where every face is orientated according to the orientation of f^*.

The only part of this algorithm that may need further investigation is obtaining the face f^*. We recommend a manual search for such a face. Since only one face is needed, this would probably not be such a tedious task.

Also, if the face f^* is orientated incorrectly, then the result of applying the algorithm would be that all the faces in the mesh are orientated incorrectly. The reversal of the ordering of vertices of every face in the mesh would then produce the desired result.

Algorithm 2.9 gives the algorithmic form of the procedure described above. Some notation used in this algorithm is introduced next. Consider a mesh $M \in \mathcal{M}$, with a given face $f^* \in F(M)$.

Let F' denote the set of faces that have already been checked and corrected if necessary. The algorithm is terminated as soon as $|F'| = |F(M)|$, implying that every face in $F(M)$ has been checked.

Let F_T denote the list of faces that can be checked next, i.e. all faces sharing edges with grey faces in Figure 2.12. This list will be implemented as a "last-in-first-out" list.

Let E_T denote the last-in-first-out list of edges corresponding to the faces in F_T (the bold edges in the figure). We assume that faces and corresponding edges are inserted into F_T and E_T in the same order.

The function remove(L) is defined to return the element in the list L that was inserted last. This element is then also removed from L. The function add(L, i) adds an element i to the list L, and returns the updated list. These two functions will be applied to the lists F_T and E_T.

The following functions are also used in the algorithm. The function edges$[F(M), f]$ is defined as follows, for a face $f = (f_1, f_2, f_3) \in F(M)$,

$$\text{edges}[F(M), f] = \{(f_1, f_2), (f_2, f_3), (f_3, f_1)\}. \tag{2.24}$$

This gives the three edges contained in the face f. The ordering of vertices in edges are very important here.

The function otherface$[F(M), e, f]$, where f contains the (unordered) edge e, returns the face in $F(M) \setminus \{f\}$ containing the edge e. Therefore, for a face $f \in F(M)$ and an edge $e = (i, j) \in E(G_M)$,

$$\text{otherface}[F(M), e, f] = \{f_p \in F(M) \setminus \{f\} : \{i, j\} \subset f_p\}. \tag{2.25}$$

The function flipface(f) returns the face f, where the ordering of the vertices of f has been reversed. For example, if $f = (7, 1, 11)$, then flipface(f) $= (11, 1, 7)$. Hence, for a face $f = (f_1, f_2, f_3) \in F(M)$,

$$\text{flipface}(f) = (f_3, f_2, f_1). \tag{2.26}$$

Algorithm 2.9 performs the checking, and correcting, of the orientation of faces in a given mesh M, with the aid of a specific face f^*, which is known to be correct.

A face $f \in F_T$ is checked for correct orientation in the following way. If the corresponding edge $e \in E_T$ appears in f in the same order as it appears in the other face sharing e, then f is orientated incorrectly. If not, then f is orientated correctly. See for instance Figure 2.12.

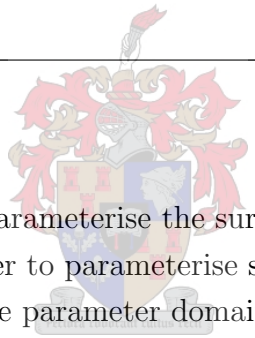
Algorithm 2.9 :
INPUT: A mesh $M \in \mathcal{M}$, and a face $f^* \in F(M)$
OUTPUT: A new face set F', such that every face in F' is orientated as f^*

1. *Let* $F' \leftarrow \{f^*\}$ *and* $F_T \leftarrow \emptyset$
2. $\{e_1, e_2, e_3\} \leftarrow \text{edges}[F(M), f^*]$
3. **for** $i = 1$ *to* 3 **do**
   $f_p \leftarrow \text{otherface}[F(M), e_i, f^*]$
   $F_T \leftarrow \text{add}(F_T, f_p)$ *and* $E_T \leftarrow \text{add}(E_T, e_i)$
   end
4. **while** $|F'| < |F(M)|$ **do**
   $f \leftarrow \text{remove}(F_T)$ *and* $e \leftarrow \text{remove}(E_T)$
   $e_1 \leftarrow \text{edge}[\text{F(M)}, \text{f}, \text{e}]$
   if $e_1 = e$ **then**
   $F' \leftarrow F' \cup \{\text{flipface}(f)\}$
   else
   $F' \leftarrow F' \cup \{f\}$
   end
   $\{e_1, e_2, e_3\} \leftarrow \text{edges}[F', f]$
   for $i = 1$ *to* 3 **do**
   $f_p \leftarrow \text{otherface}[F(M), e_i, f]$
   if $f_p \notin F'$ **then**
   $F_T \leftarrow \text{add}(F_T, f_p)$ *and* $E_T \leftarrow \text{add}(E_T, e_i)$
   end
   end
   end
5. **return** F'

This concludes the chapter on graphs and meshes. In the next chapter, the main problem addressed in this thesis is presented.

CHAPTER 3

Surface parameterisation



The main focus of this thesis is to parameterise the surface of an arbitrary mesh from the class $\mathcal{M}$ (see Definition 2.2). In order to parameterise such a mesh, a one-to-one mapping from the surface of the mesh to some parameter domain is needed.

Recall from section 2.2.5 that a homeomorphism is a bijective mapping h such that both h and h^{-1} are continuous.

Since we consider meshes with surfaces homeomorphic to the sphere, a natural choice for a parameter domain for these meshes is the surface of the sphere, S_0, as defined in (2.15). We define the concept of a *spherical parameterisation* as follows.

Definition 3.1 : *A spherical parameterisation of a mesh $M \in \mathcal{M}$ is a homeomorphism h between S_M and S_0, where S_M denotes the surface of M, as given by (2.11), and S_0 denotes the surface of the unit sphere, defined in (2.15).*

The problem of finding a valid spherical parameterisation for an arbitrary mesh in $\mathcal{M}$ is referred to as the *problem of parameterisation*. It follows directly from the definition of the class $\mathcal{M}$ (Definition 2.2, on page 13) that a solution to this problem always exists.

Parameterising the surface of a 3D model is regarded as an important problem in computer graphics. Applications include remeshing, filtering, texture mapping, and morphing.

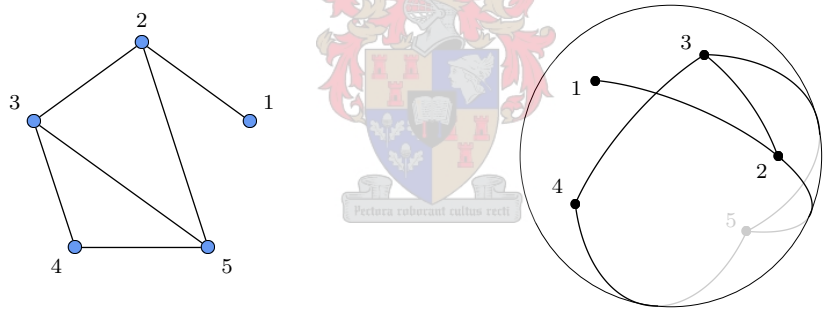
Chapter 8 briefly discusses some of these applications.

The rest of this chapter deals with a specific spherical drawing of the underlying graph G_M of a mesh $M \in \mathcal{M}$, and shows that the problem of parameterisation is easily solved once such a drawing is obtained.

3.1 GC-embeddings

In this section a specific embedding of the underlying graph of a mesh in $\mathcal{M}$ is discussed. We will show that this embedding leads to a valid spherical parameterisation of the mesh.

We define a *spherical drawing* of any graph G as follows. Every vertex $i \in V(G)$ is positioned at a point $\mathbf{v}_i \in S_0$, and every edge $(i, j) \in E(G)$ is drawn as a simple curve on the surface of the unit sphere, between $\mathbf{v}_i$ and $\mathbf{v}_j$. Figure 3.1 shows a graph in (a), and a spherical drawing of the graph in (b).



(a) a graph G (b) a spherical drawing of G

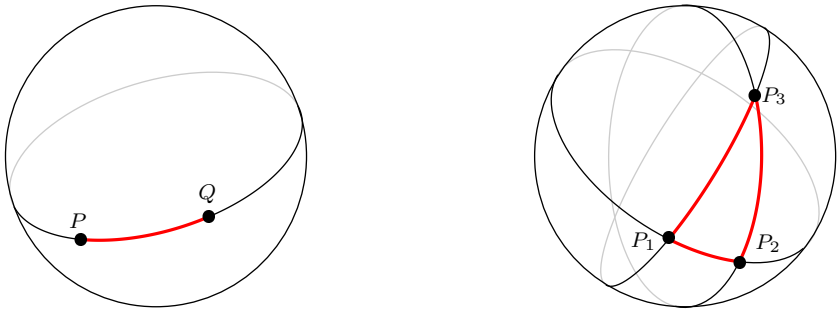
Figure 3.1

A *great circle* is a circle on the surface of a sphere with the same radius and midpoint as that sphere. Consider the great circle C passing through two points P and Q on a sphere. The shorter of the two arcs of C between P and Q is called the *minor arc* between P and Q. This is the shortest curve on the surface of the sphere between points P and Q.

A *spherical polygon* through the points P_i, $i = 1, 2, \ldots, k$, on the surface of a sphere is defined to be the region on the surface of the sphere enclosed by the minor arcs of the great circles connecting P_i with P_{i+1}, $i = 1, 2, \ldots, k-1$, and P_k with P_1. Specifically, if $k = 3$, it is called a *spherical triangle*.

Figure 3.2(a) shows a sphere and two points P and Q on the surface, with the great circle through P and Q. The arc shown in bold is the minor arc between P and Q. Figure 3.2(b)

shows a sphere with three points P_1, P_2 and P_3 on the surface. The edges of the spherical triangle defined by these three points are shown in bold.



(a) minor arc (b) spherical triangle

Figure 3.2

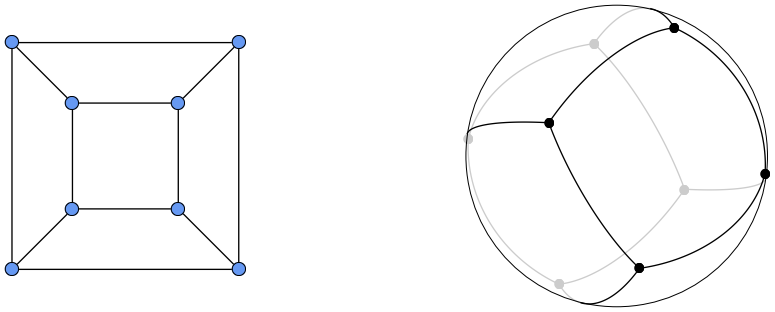
We define a *GC-drawing* of a graph G as a spherical drawing of G where every edge is drawn as a minor arc between its endpoints. "GC" is an abbreviation for Great Circle.

Consider a drawing of a graph G on some surface S. If none of the edges in that drawing intersect, the drawing is called an *embedding* of G on S. This concept gives rise to the following definition.

Definition 3.2 : *A GC-embedding of a graph G is any GC-drawing of G such that:*

(1) the vertices are positioned at distinct points in S_0, and
(2) none of the edges in the drawing intersect.

Figure 3.3 shows a graph G in (a), and a GC-embedding of G in (b).



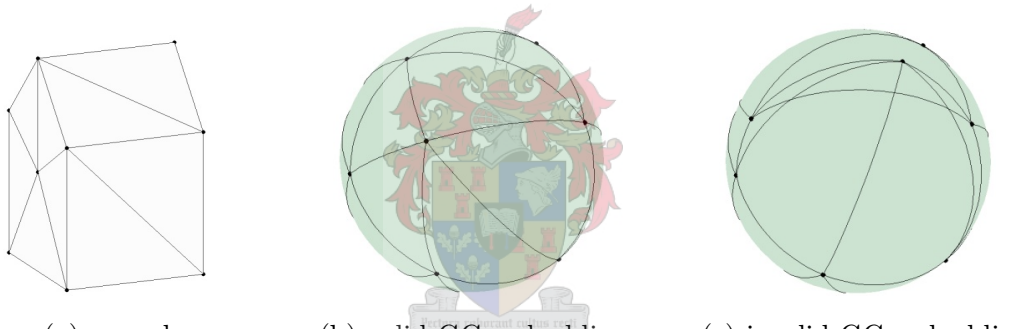
(a) a graph G (b) a GC-embedding of G

Figure 3.3

Note that in Figure 3.3(b), the surface of the sphere is divided into a number of spherical polygons, such that none of these polygons overlap.

In particular, a GC-embedding of the underlying graph G_M of any mesh $M \in \mathcal{M}$ partitions the surface of the sphere into a number of non-overlapping spherical triangles, such that every face of M corresponds to a unique spherical triangular region on the surface of the sphere.

Figure 3.4 shows a mesh $M \in \mathcal{M}$ in (a) and two GC-drawings of the graph G_M in (b) and (c). The postitions of the vertices in (b) are all distinct and none of the edges intersect. The drawing in (b) is therefore a valid GC-embedding of G_M. The drawing in (c), however, is not a GC-embedding, since edge intersections clearly occur. Also, the GC-embedding in (b) divides the surface of the sphere into non-overlapping spherical triangles, while some of the triangles in (c) overlap.



(a) a mesh (b) valid GC-embedding (c) invalid GC-embedding

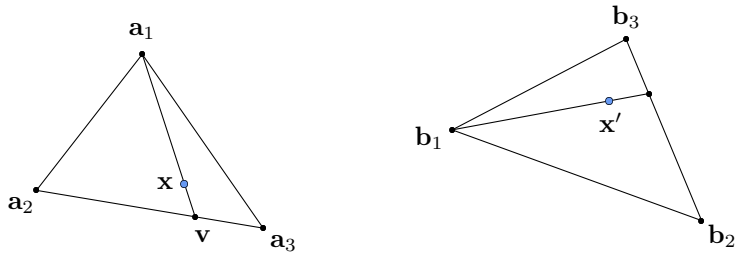
Figure 3.4

Before we prove that a GC-embedding of G_M leads to a spherical parameterisation of M, the following section provides some theory on mapping between the sets of points inside two different triangles.

3.2 Mapping between two triangles in $\mathbb{R}^3$

Consider two arbitrary planar traingles in $\mathbb{R}^3$, each of which has a strictly positive area. Suppose the set T_A contains all points inside or on the boundary of the one triangle, and the set T_B contains all points inside or on the boundary of the other triangle. This section explains how a homeomorphic mapping between T_A and T_B may be constructed.

Suppose the three vertices of triangle T_A are positioned at $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3 \in \mathbb{R}^3$, and the three vertices of T_B are positioned at $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3 \in \mathbb{R}^3$. Figure 3.5 gives an example.



(a) T_A (b) T_B

Figure 3.5

Consider a point $\mathbf{x} \in T_A$. The line through $\mathbf{a}_1$ and $\mathbf{x}$ intersects the line segment between $\mathbf{a}_2$ and $\mathbf{a}_3$, at say $\mathbf{v}$. See Figure 3.5(a). Hence, for some fixed $\lambda, \mu \in [0, 1]$,

$$\begin{aligned} \mathbf{x} &= \lambda\mathbf{a}_1 + (1-\lambda)\mathbf{v} \\ &= \lambda\mathbf{a}_1 + (1-\lambda)\left[\mu\mathbf{a}_2 + (1-\mu)\mathbf{a}_3\right] \\ &= w_1\mathbf{a}_1 + w_2\mathbf{a}_2 + w_3\mathbf{a}_3, \end{aligned} \tag{3.1}$$

with $w_1 = \lambda$, $w_2 = (1-\lambda)\mu$ and $w_3 = (1-\lambda)(1-\mu)$. Note that $w_1 + w_2 + w_3 = 1$. We rewrite (3.1) as the following linear system,

$$A\mathbf{w} = \mathbf{x}, \tag{3.2}$$

with $A = [\mathbf{a}_1, \ \mathbf{a}_2, \ \mathbf{a}_3]$ and $\mathbf{w} = [w_1, \ w_2, \ w_3]^T$. Since the vectors $\mathbf{a}_1$, $\mathbf{a}_2$ and $\mathbf{a}_3$ are linearly independent, A is non-singular. The solution $\mathbf{w}$ of (3.2) therefore exists, and is unique.

The vector $\mathbf{w}$ is referred to as the *barycentric coordinates* of $\mathbf{x}$, with respect to the triangle T_A.

We define the point $\mathbf{x}'$ as

$$\mathbf{x}' = B\mathbf{w}, \tag{3.3}$$

with $B = [\mathbf{b}_1, \ \mathbf{b}_2, \ \mathbf{b}_3]$. Clearly, as Figure 3.5(b) illustrates, $\mathbf{x}'$ lies inside the triangle T_B. Also, $\mathbf{x}'$ is the unique point in T_B corresponding to the point $\mathbf{x}$. Note, for example, that $\mathbf{a}_i$ maps to $\mathbf{b}_i$, $i = 1, 2, 3$.

The mapping discussed above is a continuous mapping from T_A to T_B. Since the triangles are arbitrary, a similar mapping is used for the inverse mapping from T_B to T_A. Therefore the mapping is homeomorphic. We call this mapping the *barycentric mapping* between the two triangles.

In the following section it is shown that the problem of parameterisation may be solved by finding a valid GC-embedding of the corresponding underlying graph. We use the barycentric mapping established here, for this purpose.

3.3 Spherical parameterisation from a GC-embedding

For a mesh $M \in \mathcal{M}$ we proceed to construct a continuous bijective mapping h between the surface of M, S_M, and the surface of the unit sphere, S_0, given a valid GC-embedding of G_M. This mapping would then be a solution to the problem of parameterisation.

Consider a mesh $M \in \mathcal{M}$, with a given GC-embedding of G_M. Suppose $\mathbf{v}_i$ denotes the coordinates of vertex $i \in V(G_M)$ in this GC-embedding.

First, it is shown how any point in S_M can be uniquely and continuously mapped to a point in S_0, using the given GC-embedding. Then it is also shown how any point in S_0 maps uniquely and continuously to a point in S_M.

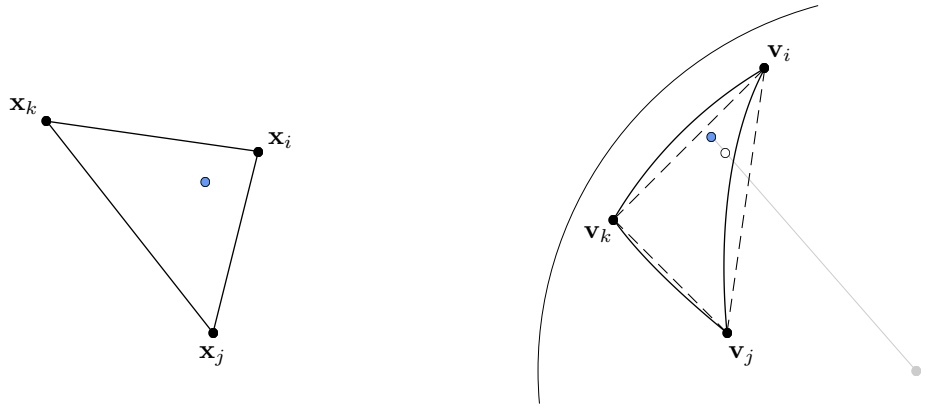
Consider a point $\mathbf{x} \in S_M$. Since the surface of M is partitioned into a number of flat triangular faces in $\mathbb{R}^3$, this point $\mathbf{x}$ lies inside or on the boundary of one of these faces, say $f \in F(M)$. Suppose f has vertices with indices i, j and k, such that $i, j, k \in V(G_M)$.

From the given GC-embedding of G_M, we know that the face f corresponds to a unique spherical triangle, say f_s, on the surface of the unit sphere. This spherical triangle has vertices positioned at $\mathbf{v}_i$, $\mathbf{v}_j$ and $\mathbf{v}_k$.

The point $\mathbf{x}$ is mapped with a barycentric mapping to a unique point $\mathbf{v}'$ in the flat triangle with vertices $\mathbf{v}_i$, $\mathbf{v}_j$ and $\mathbf{v}_k$. We then map the point $\mathbf{v}'$ to the surface of the sphere by means of normalisation. Hence

$$\mathbf{v} = \frac{\mathbf{v}'}{||\mathbf{v}'||} \tag{3.4}$$

denotes the unique point in S_0 corresponding to the point $\mathbf{x} \in S_M$.



(a) surface of M (b) surface of unit sphere

Figure 3.6

Figure 3.6 illustrates the mapping. The dot inside the triangle in (a) indicates $\mathbf{x}$. This point is mapped to $\mathbf{v}'$, the white dot in (b), with a barycentric mapping. The point $\mathbf{v}'$ is then normalised to yield $\mathbf{v}$ on the surface of the sphere.

This establishes a continuous mapping h from S_M to S_0.

Next, consider a point $\mathbf{v} \in S_0$. Since the GC-embedding of G_M partitions the surface of the unit sphere into a number of spherical triangles, the point $\mathbf{v}$ lies inside or on the boundary of one of these spherical triangles, say f_s. This triangle corresponds to a unique triangular face of M, say $f \in F(M)$.

Suppose f_s has vertices positioned at $\mathbf{v}_i$, $\mathbf{v}_j$ and $\mathbf{v}_k$. Then f has vertices positioned at $\mathbf{x}_i$, $\mathbf{x}_j$ and $\mathbf{x}_k$.

Since $\mathbf{v}$ lies inside or on the boundary of f_s, the vector $\mathbf{v}$ intersects the flat triangle with vertices $\mathbf{v}_i$, $\mathbf{v}_j$ and $\mathbf{v}_k$. It is then possible to write

$$\alpha\mathbf{v} = w_1\mathbf{v}_i + w_2\mathbf{v}_j + w_3\mathbf{v}_k, \tag{3.5}$$

with $\alpha \in (0, 1)$, and $w_1, w_2, w_3 \in [0, 1]$. Since $\mathbf{v}$ is inside the triangle f, it follows that $w_1 + w_2 + w_3 = 1$. Equation (3.5) is rewritten as

$$\mathbf{v} = \frac{w_1}{\alpha}\mathbf{v}_i + \frac{w_2}{\alpha}\mathbf{v}_j + \frac{w_3}{\alpha}\mathbf{v}_k. \tag{3.6}$$

Values for w_1/α, w_2/α and w_3/α are obtained by solving the 3×3 linear system (3.6), which is similar to finding barycentric coordinates.

In order to obtain the value of α, note that

$$\frac{w_1}{\alpha} + \frac{w_2}{\alpha} + \frac{w_3}{\alpha} = \frac{1}{\alpha}(w_1 + w_2 + w_3) = \frac{1}{\alpha}. \tag{3.7}$$

This gives an expression to solve for α, and then to obtain values for w_1, w_2 and w_3.

We define the point $\mathbf{x}$ to be

$$\mathbf{x} = w_1\mathbf{x}_i + w_2\mathbf{x}_j + w_3\mathbf{x}_k, \tag{3.8}$$

which is the unique point in S_M corresponding to $\mathbf{v} \in S_0$. Figure 3.6 may also serve as an illustration for this inverse mapping. The dot on the surface of the sphere in (b) indicates $\mathbf{v}$. The white dot in (b) is $\alpha\mathbf{v}$, which is mapped to the triangle in (a), yielding the point $\mathbf{x}$.

This establishes the inverse mapping h^{-1} from S_0 to S_M.

Both h and h^{-1} are continuous and bijective. The mapping h is therefore a solution to the problem of parameterisation, and we have the following result.

Proposition 3.3 : *For a mesh $M \in \mathcal{M}$, it is possible to solve the problem of parameterisation with a valid GC-embedding of the underlying graph G_M.*

This seems to be the standard procedure for parameterising the surface of a mesh in the literature [1, 3, 16, 17, 21, 29, 35]. In Chapter 5, a few existing methods for obtaining GC-embeddings will be discussed.

The importance of the requirement that no edges may intersect in a GC-embedding is stressed in the construction of h. As soon as an edge intersection occurs, spherical triangles overlap, and a one-to-one mapping cannot be established between $S(M)$ and S_0.

The following section provides two techniques to test whether a given GC-drawing of G_M is in fact a valid GC-embedding of G_M.

3.4 Validity of a GC-embedding

Suppose a GC-drawing of G_M, for a given mesh $M \in \mathcal{M}$, is given. Suppose therefore that a set $V = \{\mathbf{v}_i \in \mathbb{R}^3 : i = 1, 2, \ldots, n\}$ is given, such that every vertex $i \in V(G_M)$ is positioned at $\mathbf{v}_i$, and the edges are drawn as minor arcs between endpoints. This section provides two possible methods to test whether this GC-drawing is a valid GC-embedding of G_M.

According to Definition 3.2, it is necessary to test that the points $\mathbf{v}_i$ are distinct, and that no edges intersect in the drawing. Note that an edge intersection implies that at least two spherical triangles overlap, and vice versa.

It is thus sufficient to test that the points $\mathbf{v}_i$ are distinct, and that the spherical triangles defined by these points do not overlap. The first condition is easily tested. The following two methods test the second condition.

3.4.1 The orientation test

The first test is based on the fact that a valid spherical parameterisation is obtained if all the spherical triangles are orientated correctly. That is, for a face $f \in F(M)$ the side that is on the outside of the mesh must be on the outside of the sphere. There can be no foldovers in the embedding without at least one face being upside down.

Suppose the convention in section 2.2 is followed, namely that the labels of the vertices of every face are ordered to be counterclockwise as seen from outside the mesh. For a face $f \in F(M)$ corresponding to the spherical triangle f_s with vertices $\mathbf{v}_i$, $\mathbf{v}_j$ and $\mathbf{v}_k$, we then calculate

$$s(f_s) = \text{sign}\left[(\mathbf{v}_i \times \mathbf{v}_j) \cdot \mathbf{v}_k\right], \tag{3.9}$$

where the function $\text{sign}(\cdot) : \mathbb{R} \mapsto \{-1, 0, 1\}$ is defined as

$$\text{sign}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0, \end{cases} \qquad x \in \mathbb{R}. \tag{3.10}$$

Now, if $s(f_s) = 1$, then f_s is orientated correctly. If $s(f_s) = -1$, then f_s is upside down, and if $s(f_s) = 0$ then f_s is either a single point or an arc on the sphere, in other words, f_s is a collapsed triangle.

In order to test the entire GC-drawing, (3.9) is evaluated for every spherical face f_s in the GC-drawing. If the result is 1 for all the faces, then all the faces are orientated correctly, and the drawing is a valid GC-embedding. We call this the *orientation test*.

The following theorem validates the correctness of the orientation test.

Theorem 3.4 : *Given a spherical triangle f with vertices located at $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$, the value of $s(f) = \text{sign}\left[(\mathbf{v}_1 \times \mathbf{v}_2) \cdot \mathbf{v}_3\right]$ satisfies the following:*

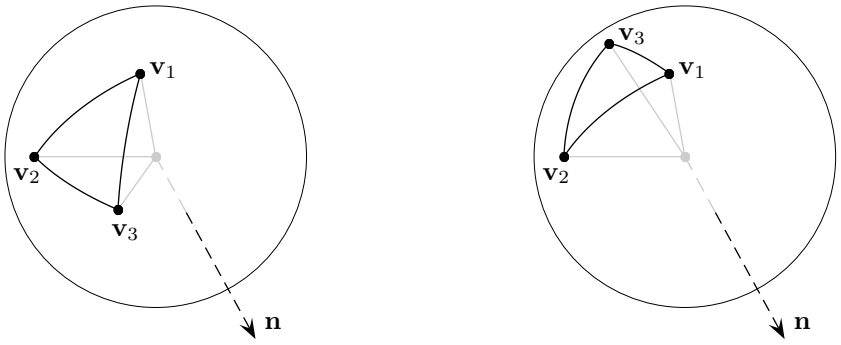
(a) $s(f) = 0$ *if the area of the triangle is zero,*
(b) $s(f) = 1$ *if the vertices are ordered counterclockwise,*
(c) $s(f) = -1$ *if the vertices are ordered clockwise.*

By ordering we mean as viewed from outside the sphere.

Proof

In order to prove (a), suppose first that $\mathbf{v}_1 = \mathbf{v}_2$. Then $\mathbf{v}_1 \times \mathbf{v}_2 = \mathbf{0}$, implying that $s(f) = 0$. If $\mathbf{v}_1 \neq \mathbf{v}_2$, but $\mathbf{v}_3$ lies on the great circle through $\mathbf{v}_1$ and $\mathbf{v}_2$, then $(\mathbf{v}_1 \times \mathbf{v}_2)$ is perpendicular to $\mathbf{v}_3$. This also implies that $s(f) = 0$.

In order to prove (b) and (c), suppose the area of the triangle is strictly positive. The vector $\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2$ is perpendicular to the plane spanned by $\mathbf{v}_1$ and $\mathbf{v}_2$. This plane partitions $\mathbb{R}^3$ into two open halfspaces, say H_1 and H_2, with $\mathbf{n} \in H_1$.



(a) counterclockwise (b) clockwise

Figure 3.7

If the vertices of f are ordered counterclockwise, then $\mathbf{v}_3 \in H_1$ (see Figure 3.7(a)). Thus $\mathbf{n} \cdot \mathbf{v}_3 > 0$, and $s(f) = 1$. If the vertices of f are ordered clockwise, then $\mathbf{v}_3 \in H_2$ (see Figure 3.7(b)). Thus $\mathbf{n} \cdot \mathbf{v}_3 < 0$, and $s(f) = -1$. ■

The advantage of this test is that it identifies the faces that are upside down. A disadvantage is that all the faces in the face set of the mesh must be correctly orientated, i.e. the mesh must be preprocessed with Algorithm 2.9.

3.4.2 The area test

The second test is based on the fact that a valid GC-embedding is obtained if the sum of the areas of all the spherical triangles is exactly the area of the unit sphere, which is 4π. As soon as an overlapping occurs, this sum would become larger than 4π.

For a given GC-drawing, with $V = \{\mathbf{v}_i \in \mathbb{R}^3 : i = 1, 2, \ldots, n\}$, we calculate

$$A = \sum_{f=1}^{|F(M)|} a(f), \tag{3.11}$$

where the function $a : F(M) \mapsto \mathbb{R}^3$ gives the area of the spherical triangle corresponding to the face $f \in F(M)$. There are no overlappings in a GC-drawing of G_M if and only if $A = 4\pi$. We call this the *area test*.

Next, an explicit formula for $a(f)$ is given. Suppose the face $f \in F(M)$ maps to the spherical triangle with vertices positioned at $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$. Then, according to [30, p. 18],

$$a(f) = \angle_{312} + \angle_{123} + \angle_{231} - \pi, \tag{3.12}$$

where $\angle_{ijk}$ denotes the angle in radians at vertex j of the spherical triangle with vertices labelled i, j and k. This angle is measured as the angle between the tangents to the two incident arcs of the triangle. See for example Figure 3.8. The angle $\angle_{312}$ is shown.

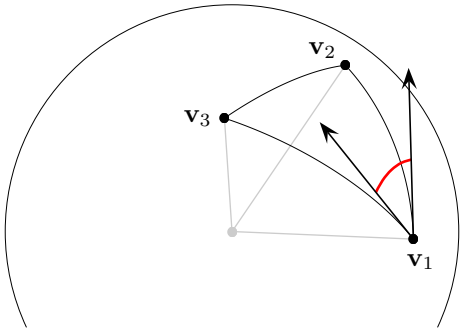


Figure 3.8

In order to obtain a formula for this angle, consider for instance the angle $\angle_{312}$. A vector $\mathbf{a}$ parallel to the line through $\mathbf{v}_1$, tangent to the arc between $\mathbf{v}_1$ and $\mathbf{v}_2$, is given by

$$\mathbf{a} = (\mathbf{v}_1 \times \mathbf{v}_2) \times \mathbf{v}_1. \tag{3.13}$$

Using the identity $(\mathbf{p} \times \mathbf{q}) \times \mathbf{r} = (\mathbf{p} \cdot \mathbf{r})\mathbf{q} - (\mathbf{q} \cdot \mathbf{r})\mathbf{p}$ and the fact that $\mathbf{v}_1 \cdot \mathbf{v}_1 = 1$ yields

$$\mathbf{a} = \mathbf{v}_2 - (\mathbf{v}_1 \cdot \mathbf{v}_2)\mathbf{v}_1. \tag{3.14}$$

Similarly, a vector $\mathbf{b}$ parallel to the line through $\mathbf{v}_1$, tangent to the arc between $\mathbf{v}_1$ and $\mathbf{v}_3$, is given by

$$\mathbf{b} = (\mathbf{v}_1 \times \mathbf{v}_3) \times \mathbf{v}_1 = \mathbf{v}_3 - (\mathbf{v}_1 \cdot \mathbf{v}_3)\mathbf{v}_1. \tag{3.15}$$

It follows that the angle $\angle_{312}$ is equal to the angle between $\mathbf{a}$ and $\mathbf{b}$, and therefore,

$$\cos(\angle_{312}) = \frac{\mathbf{a} \cdot \mathbf{b}}{||\mathbf{a}||\,||\mathbf{b}||}, \tag{3.16}$$

which gives a formula for calculating $\angle_{312}$. Similar expressions can be derived for $\angle_{123}$ and $\angle_{231}$, and are then used in (3.12) to calculate the area of the spherical triangle.

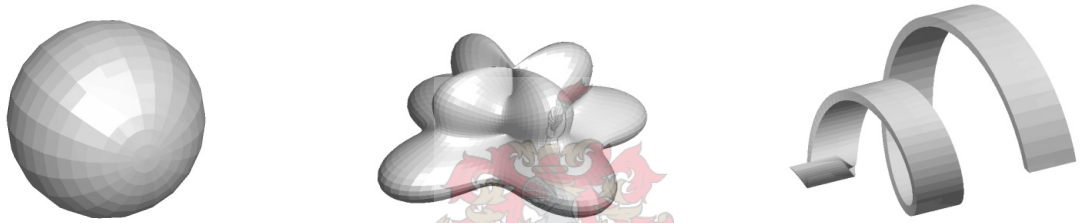
The advantage of this method is that correct orientation of the faces is not required a priori. A disadvantage is that numerical round-off errors are bound to occur. A possibility is to evaluate $|A - 4\pi|$, and if it is less than some fraction of the area of the smallest spherical triangle, the embedding can be said to be valid.

3.5 Parameterising convex or star-shaped meshes

For some meshes in $\mathcal{M}$, a valid GC-embedding is obtained by simply normalising every vertex of the original mesh, with respect to some point inside the mesh.

A *star-shaped* mesh has the property that a point inside the mesh can be found such that any ray originating at that point intersects the surface of the mesh exactly once. Such a point is called a *radial centroid* of the mesh. The radial centroid of a mesh is usually not unique.

A *convex* mesh is a mesh where any point strictly inside the mesh is a radial centroid. Any mesh that is not convex is called *concave*. See for example Figure 3.9.



(a) convex (b) star-shaped (c) concave

Figure 3.9

A convex or star-shaped mesh in $\mathcal{M}$ has the advantage that a valid GC-embedding is easily obtained. Suppose that for a convex or star-shaped mesh M one of its radial centroids is positioned at $\mathbf{m} \in \mathbb{R}^3$. Then

$$\mathbf{v}_i = \frac{\mathbf{x}_i - \mathbf{m}}{||\mathbf{x}_i - \mathbf{m}||}, \quad i \in V(G_M), \tag{3.17}$$

with $\mathbf{x}_i$ the coordinates of node i in the original mesh, yields the positions for the vertices of a valid GC-embedding of G_M.

The embedding in Figure 3.4(b) was obtained by using (3.17) and choosing the point $\mathbf{m}$ to be the mean of the coordinates of all the nodes, i.e.

$$\mathbf{m} = \frac{1}{|V(G_M)|} \sum_{i \in V(G_M)} \mathbf{x}_i. \tag{3.18}$$

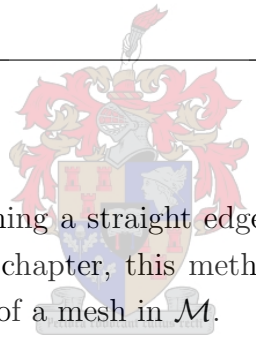
For convex meshes, this method for finding $\mathbf{m}$ is sufficient, but for some star-shaped meshes it can be rather difficult to obtain a radial centroid.

The interesting problem of parameterisation arises when a general mesh, such as the rabbit model in Figure 2.1, is considered.

Before we look at methods for finding valid GC-embeddings of arbitrary meshes in $\mathcal{M}$, we deal, in the following chapter, with an algorithm for finding valid 2D planar embeddings. In Chapter 5, this algorithm is then extended to find valid GC-embeddings.

CHAPTER 4

2D planar embeddings



In this chapter, a method for obtaining a straight edge drawing of a planar graph in the 2D plane is discussed. In the next chapter, this method is adapted to find a valid GC-embedding of the underlying graph of a mesh in $\mathcal{M}$.

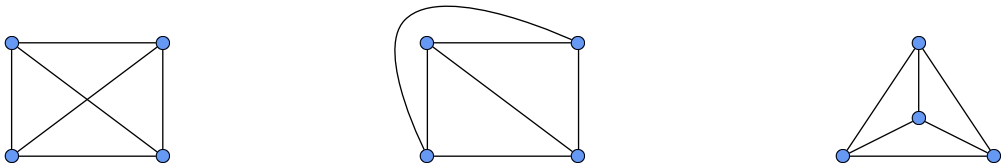
4.1 2D Tutte embedding

The definition in section 2.1 of a planar graph translates to the following: a graph G is planar if it can be embedded in the 2D plane such that each vertex $i \in V(G)$ is mapped to a point in $\mathbb{R}^2$, each edge $(i, j) \in E(G)$ is mapped to a curve whose endpoints are the points i and j, and none of these edges intersect. Such an embedding is called a *planar embedding* of the graph.

If a planar embedding of G has the property that every edge is drawn as a segment of a straight line, and no edge intersections occur, we call it a *straight line embedding* of the graph G. It was first shown by Fáry [8] that every planar graph has a valid straight line embedding.

Figure 4.1 shows a planar graph in (a). This embedding of the graph is not a planar embedding, because of the two edges intersecting. A planar embedding of this graph is

shown in (b), and a straight line embedding is shown in (c).



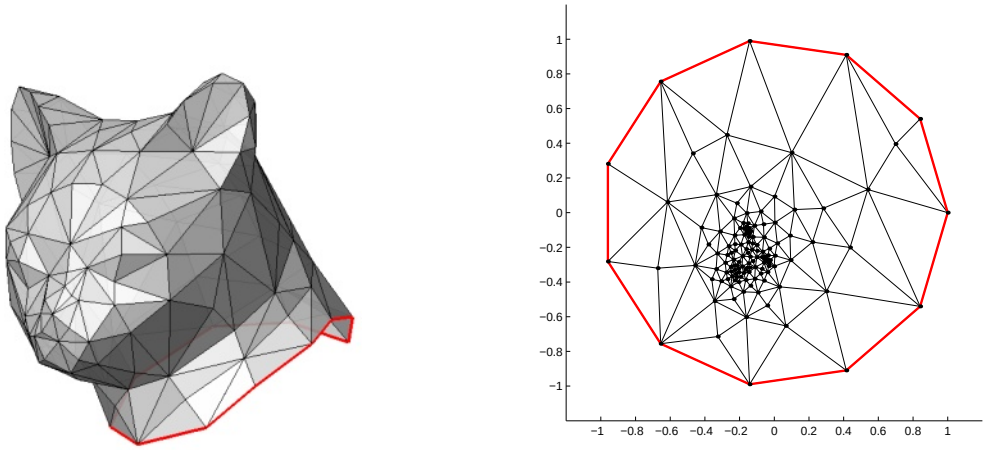
(a) a planar graph (b) planar embedding (c) straight line embedding

Figure 4.1

Tutte [33] proposed a method for finding a straight line embedding of the graph G_M of a mesh $M \in \mathcal{M}_B$ (i.e. a mesh with a boundary, see Definition 2.3). We call an embedding obtained by his algorithm a *2D Tutte embedding*.

Obtaining a 2D Tutte embedding of G_M involves two steps. First, the boundary vertices of G_M are fixed on a convex boundary in the 2D plane, such that the ordering of these vertices is preserved. Next, every interior vertex of the graph is positioned at the centroid of its neighbours. Tutte proved that this results in a drawing of the graph with no edge intersections [33].

Figure 4.2(a) shows an example of a mesh in $\mathcal{M}_B$. Figure 4.2(b) shows a 2D Tutte embedding of the underlying graph, where the boundary vertices are fixed on the boundary of the unit circle centred at the origin.



(a) a mesh M in $\mathcal{M}_\mathcal{B}$ (b) a 2D Tutte embedding of G_M

Figure 4.2

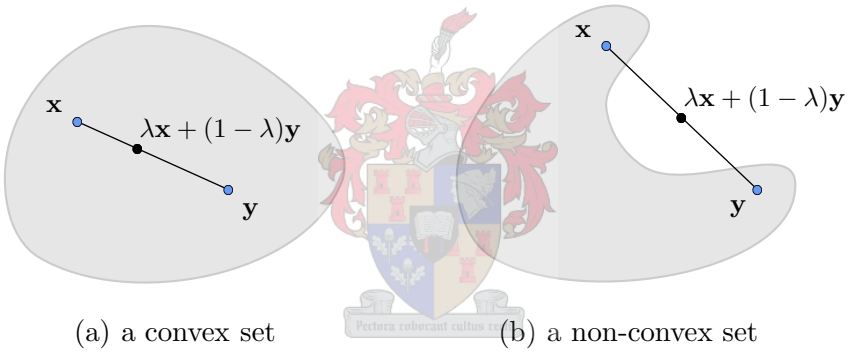
In Tutte's algorithm, the positioning of a vertex at the centroid of its neighbours may be generalised to any convex combination of the positions of the neighbours. Before this

generalisation is discussed, the following section provides some theory on convex combinations. This theory contributes to proving that the generalisation of Tutte's method always yields a valid straight line embedding for the graph G_M of any mesh $M \in \mathcal{M}_B$.

4.2 Convex combinations

In this section the concepts of a convex set, a convex hull and a convex combination in $\mathbb{R}^2$ are defined, and relationships between these concepts are derived.

A set of points $P \subset \mathbb{R}^2$ is said to be a *convex set* in $\mathbb{R}^2$ if, for any points $\mathbf{x}$, $\mathbf{y} \in P$, and any $\lambda \in (0, 1)$, the point $\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}$ is also in P. Figure 4.3 shows an example of a convex set and a non-convex set.



(a) a convex set (b) a non-convex set

Figure 4.3

The *convex hull* of a set of points $S \subset \mathbb{R}^2$ is defined to be the convex set in $\mathbb{R}^2$ with smallest possible area containing S. We denote the set of points strictly inside the convex hull of S by $\mathrm{CH}(S)$. Figure 4.4 shows an example of the convex hull of a set of points.

(a) a set S of points (b) $\mathrm{CH}(S)$

Figure 4.4

The following lemma gives an explicit formula for any point in the convex hull of a set of vectors, in terms of those vectors. This lemma is used to prove a subsequent theorem.

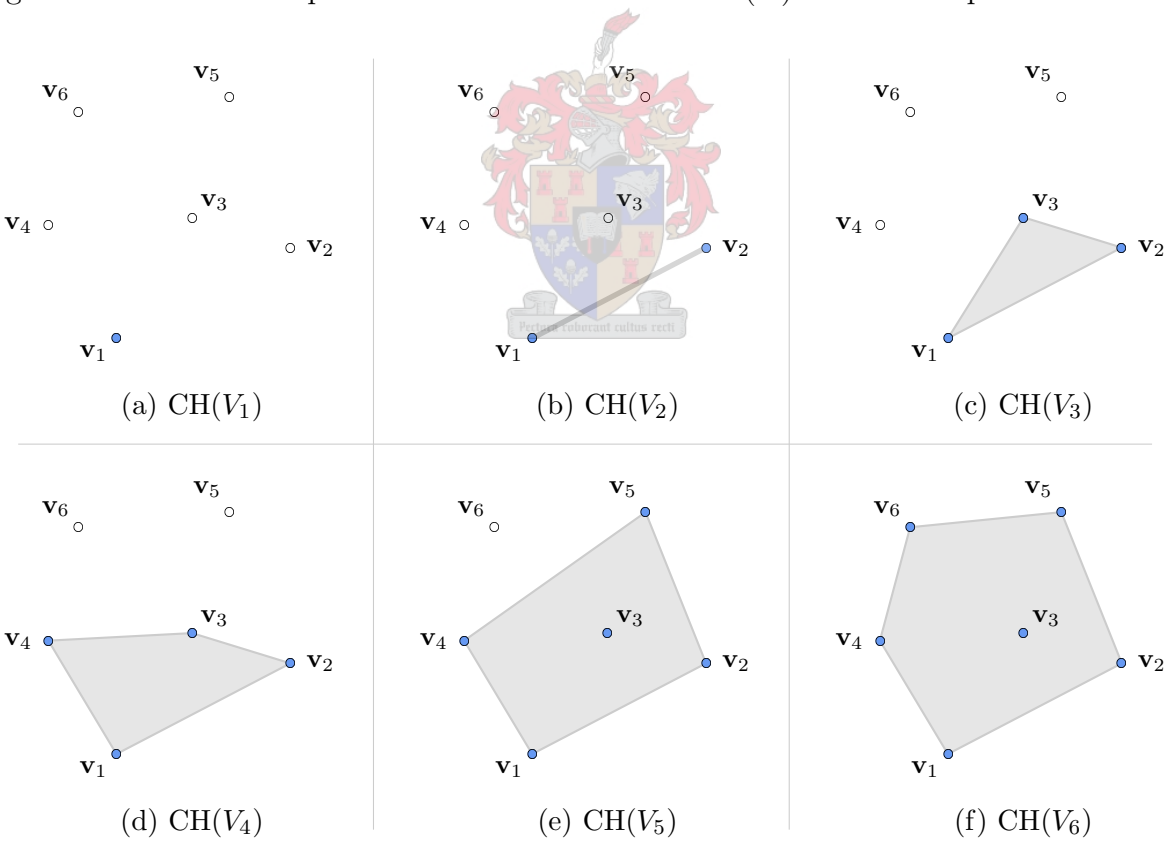
Lemma 4.1 : *For a set of vectors* $V = \{\mathbf{v}_i \in \mathbb{R}^2,\ i = 1, 2, \ldots, n\}$, *any point* $\mathbf{v} \in \mathrm{CH}(V)$ *can be expressed as*

$$\mathbf{v} = \sum_{i=1}^{n} \left[\mathbf{v}_i (1 - \lambda_i) \prod_{j=i+1}^{n+1} \lambda_j \right], \tag{4.1}$$

with $\lambda_1 = 0$, $\lambda_i \in (0, 1)$, $i = 2, 3, \ldots, n$, *and* $\lambda_{n+1} = 1$. *Moreover, for* $\lambda_1 = 0$, $\lambda_{n+1} = 1$, *and any choice of* $\lambda_i \in (0, 1)$, $i = 2, 3, \ldots, n$, *the point* $\mathbf{v}$ *as given by (4.1) is an element of* $\mathrm{CH}(V)$.

Proof

Figure 4.5 shows the steps of a method to construct $\mathrm{CH}(V)$ for an example set V.



(a) $\mathrm{CH}(V_1)$ (b) $\mathrm{CH}(V_2)$ (c) $\mathrm{CH}(V_3)$

(d) $\mathrm{CH}(V_4)$ (e) $\mathrm{CH}(V_5)$ (f) $\mathrm{CH}(V_6)$

Figure 4.5

The method starts with the first vector, $\mathbf{v}_1$. Any point $\mathbf{x}_1$ in the convex hull of the set $V_1 = \{\mathbf{v}_1\}$ may be expressed as

$$\mathbf{x}_1 = \mathbf{v}_1(1 - \lambda_1), \tag{4.2}$$

where we choose $\lambda_1 = 0$ to simplify the expressions that follow.

If we consider the next vector, $\mathbf{v}_2$, any point $\mathbf{x}_2$ in the convex hull of the set $V_2 = \{\mathbf{v}_1, \mathbf{v}_2\}$ may be expressed as a convex combination of $\mathbf{v}_2$ and a point in $\mathrm{CH}(V_1)$. Therefore

$$\begin{aligned} \mathbf{x}_2 &= \mathbf{x}_1\lambda_2 + \mathbf{v}_2(1-\lambda_2) \\ &= \mathbf{v}_1(1-\lambda_1)\lambda_2 + \mathbf{v}_2(1-\lambda_2), \end{aligned} \tag{4.3}$$

with $\lambda_2 \in (0,1)$. Note that any value of λ_2 in the interval $(0,1)$ yields a point in $\mathrm{CH}(V_2)$.

If we consider the next vector, $\mathbf{v}_3$, any point $\mathbf{x}_3$ in the convex hull of $V_3 = \{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$ may be expressed as a convex combination of $\mathbf{v}_3$ and a point in $\mathrm{CH}(V_2)$. Thus

$$\begin{aligned} \mathbf{x}_3 &= \mathbf{x}_2\lambda_3 + \mathbf{v}_3(1-\lambda_3) \\ &= \mathbf{v}_1(1-\lambda_1)\lambda_2\lambda_3 + \mathbf{v}_2(1-\lambda_2)\lambda_3 + \mathbf{v}_3(1-\lambda_3), \end{aligned} \tag{4.4}$$

with $\lambda_3 \in (0,1)$. Again, any value of λ_3 in the interval $(0,1)$ yields a point in $\mathrm{CH}(V_3)$.

This process is continued until the last vector in V, $\mathbf{v}_n$, is reached. Then we know that any point $\mathbf{x}_n$ in the convex hull of the set $V_n = V$ may be expressed as

$$\mathbf{x}_n = \sum_{i=1}^{n} \left[\mathbf{v}_i(1-\lambda_i) \prod_{j=i+1}^{n+1} \lambda_j \right], \tag{4.5}$$

with $\lambda_1 = 0$, $\lambda_i \in (0,1)$, $i = 2, 3, \ldots, n$, and $\lambda_{n+1} = 1$. Also, any values for λ_i, $i = 1, 2, \ldots, n+1$ satisfying these conditions yield a point in $\mathrm{CH}(V)$. ■

A *strict convex combination* of the vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n \in \mathbb{R}^d$, is defined as a linear combination

$$\mathbf{v} = \sum_{i=1}^{n} w_i \mathbf{v}_i, \tag{4.6}$$

with weights $w_i \in \mathbb{R}$, $i = 1, 2, \ldots, n$, such that

$$\sum_{i=1}^{n} w_i = 1, \quad \text{and} \quad w_i > 0, \quad i = 1, 2, \ldots, n. \tag{4.7}$$

For the purpose of this chapter, we will consider convex combinations of vectors in $\mathbb{R}^2$.

For a set of vectors $V = \mathbf{v}_i \in \mathbb{R}^2$, $i = 1, 2 \ldots, n$, we define the set CC(V) to be the set of all possible strict convex combinations of the vectors in V. Therefore

$$\mathrm{CC}(V) = \left\{ \sum_{i=1}^{n} w_i \mathbf{v}_i \ : \ \sum_{i=1}^{n} w_i = 1 \ \text{ and } \ w_i > 0, \ i = 1, 2, \ldots, n \right\}. \tag{4.8}$$

The following fundamental theorem relates the concepts of a convex hull and convex combinations. This theorem is needed in proving that the generalisation of Tutte's method yields a valid straight line embedding.

Theorem 4.2 : *For any set of vectors* $V = \{\mathbf{v}_i \in \mathbb{R}^2,\ i = 1, 2, \ldots, n\}$,

$$\mathrm{CH}(V) = \mathrm{CC}(V),$$

where $\mathrm{CH}(V)$ *denotes the set of points strictly inside the convex hull of* V, *and* $\mathrm{CC}(V)$ *is defined by (4.8).*

Proof

Let $\mathbf{v} \in \mathrm{CH}(V)$. According to Lemma 4.1, $\mathbf{v}$ can be expressed as

$$\mathbf{v} = \sum_{i=1}^{n} h_i \mathbf{v}_i, \tag{4.9}$$

where

$$h_i = (1 - \lambda_i) \prod_{j=i+1}^{n+1} \lambda_j, \quad i = 1, 2, \ldots, n, \tag{4.10}$$

with $\lambda_1 = 0$, $\lambda_i \in (0, 1)$, $i = 2, 3, \ldots, n$, and $\lambda_{n+1} = 1$. Note that with λ_i, $i = 1, 2, \ldots, n$, satisfying these conditions, it follows that $h_i > 0$, $i = 1, 2, \ldots, n$. Also,

$$\sum_{i=1}^{n} h_i = \sum_{i=1}^{n} \left[\prod_{j=i+1}^{n+1} \lambda_j - \prod_{j=i}^{n+1} \lambda_j \right] = -\prod_{j=1}^{n+1} \lambda_j + \prod_{j=n+1}^{n+1} \lambda_j = 0 + 1 = 1. \tag{4.11}$$

Therefore $\mathbf{v}$ is a strict convex combination of the vectors in V. Thus $\mathbf{v} \in \mathrm{CC}(V)$, and we have

$$\mathrm{CH}(V) \subset \mathrm{CC}(V). \tag{4.12}$$

Next, let $\mathbf{v} \in \mathrm{CC}(V)$. Suppose therefore that

$$\mathbf{v} = \sum_{i=1}^{n} w_i \mathbf{v}_i, \tag{4.13}$$

where weights w_i, $i = 1, 2, \ldots, n$, are given such that (4.7) is satisfied. According to Lemma 4.1, in order to show that $\mathbf{v}$ is an element of $\mathrm{CH}(V)$, it is only necessary to show that $\mathbf{v}$ can be written in the form (4.1), such that $\lambda_1 = 0$, $\lambda_i \in (0, 1)$, $i = 2, 3, \ldots, n$, and $\lambda_{n+1} = 1$.

By comparing (4.13) with (4.1), we choose $\lambda_1 = 0$ and $\lambda_{n+1} = 1$. For $n \geq 2$, the values of λ_i, $i = 2, 3, \ldots, n$ must be chosen to satisfy

$$w_i = (1 - \lambda_i) \prod_{j=i+1}^{n+1} \lambda_j, \quad i = 1, 2, \ldots, n. \tag{4.14}$$

For $n = 2$, we choose $\lambda_2 = 1 - w_2$. Note that $\lambda_2 \in (0, 1)$, and this specific choice is therefore valid. Suppose now that $n \geq 3$. Since $w_n = 1 - \lambda_n$, it follows that

$$\lambda_n = 1 - w_n. \tag{4.15}$$

Next, $w_{n-1} = (1 - \lambda_{n-1})\lambda_n$, and we have

$$\lambda_{n-1} = 1 - \frac{w_{n-1}}{\lambda_n}. \tag{4.16}$$

This backwards recursion is continued, and the following is obtained,

$$\lambda_i = 1 - w_i / \left[\prod_{j=i+1}^{n+1} \lambda_j \right], \quad i = n-1, n-2, \ldots, 2. \tag{4.17}$$

This, together with (4.15), gives a solution to (4.14). Note that from (4.15), $\lambda_n \in (0, 1)$. If follows from (4.17) that

$$\begin{aligned} \prod_{j=i}^{n+1} \lambda_j &= \left[\prod_{j=i+1}^{n+1} \lambda_j \right] - w_i \\ &= \left[\prod_{j=i+2}^{n+1} \lambda_j \right] - w_{i+1} - w_i \\ &= \left[\prod_{j=n+1}^{n+1} \lambda_j \right] - \sum_{j=i}^{n} w_j \\ &= 1 - \sum_{j=i}^{n} w_j, \quad i = 2, 3, \ldots, n-1. \end{aligned} \tag{4.18}$$

Therefore

$$\lambda_i = 1 - w_i / \left[1 - \sum_{j=i+1}^{n} w_j \right], \quad i = 2, 3, \ldots, n-1. \tag{4.19}$$

Clearly, since (4.7) holds, $\sum_{j=i}^{n} w_j < 1$, $i = 2, 3, \ldots, n-1$, therefore

$$1 - \sum_{j=i+1}^{n} w_j > w_i, \quad i = 2, 3, \ldots, n-1, \tag{4.20}$$

which, together with (4.19) establishes that $\lambda_i \in (0,1)$, $i = 2, 3, \ldots, n-1$.

It is therefore possible to write $\mathbf{v}$, as defined in (4.13), in the form (4.1), thereby proving that $\mathbf{v} \in \mathrm{CH}(V)$, and we have

$$\mathrm{CC}(V) \subset \mathrm{CH}(V). \tag{4.21}$$

Combining (4.12) and (4.21) proves the theorem. ■

We conclude this section by giving two results that follow from Theorem 4.2.

Corollary 4.3 : *Suppose $\mathbf{v} \in \mathbb{R}^2$ is a strict convex combination of the vectors in $V = \{\mathbf{v}_i \in \mathbb{R}^2,\ i = 1, 2, \ldots, n\}$, and suppose ℓ is any line through $\mathbf{v}$. Then either*

(a) the points $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ all lie on ℓ, or
(b) at least one of these points lies strictly on one side of ℓ, and at least one of the points lies strictly on the other side of ℓ.

Proof

Note that $\mathbf{v} \in \mathrm{CC}(V)$, and thus from Theorem 4.2, $\mathbf{v} \in \mathrm{CH}(V)$. Therefore $\mathbf{v}$ lies stricly inside the convex hull of V.

Suppose that all the points $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ lie strictly on the one side of ℓ. Such a situation is shown in Figure 4.6(a).

Figure 4.6

But then the convex hull of V, $\mathrm{CH}(V)$, is also strictly on that side of ℓ, implying that ℓ does not pass through $\mathrm{CH}(V)$. This implies that $\mathbf{v} \notin \mathrm{CH}(V)$, which is a contradiction.

Next, suppose some of the points $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ lie on ℓ, and the others strictly on one side of ℓ. This implies that ℓ coincides with one of the sides of $\mathrm{CH}(V)$. But then $\mathbf{v}$ lies on the boundary of $\mathrm{CH}(V)$, which is also a contradiction. ■

Corollary 4.4 : *Suppose* $\mathbf{v} \in \mathbb{R}^2$ *is a strict convex combination of the* n *linearly independent vectors in* $V = \{\mathbf{v}_i \in \mathbb{R}^2,\ i = 1, 2, \ldots, n\}$*, with* $n \geq 3$*. Then integers* $a, b, c \in \{1, 2, \ldots, n\}$ *exist, such that* $\mathbf{v}$ *lies strictly inside the triangle with vertices* $\mathbf{v}_a$*,* $\mathbf{v}_b$ *and* $\mathbf{v}_c$*.*

Proof

We proceed by constructing such a triangle. Figure 4.7 gives an illustration of the idea behind this construction.

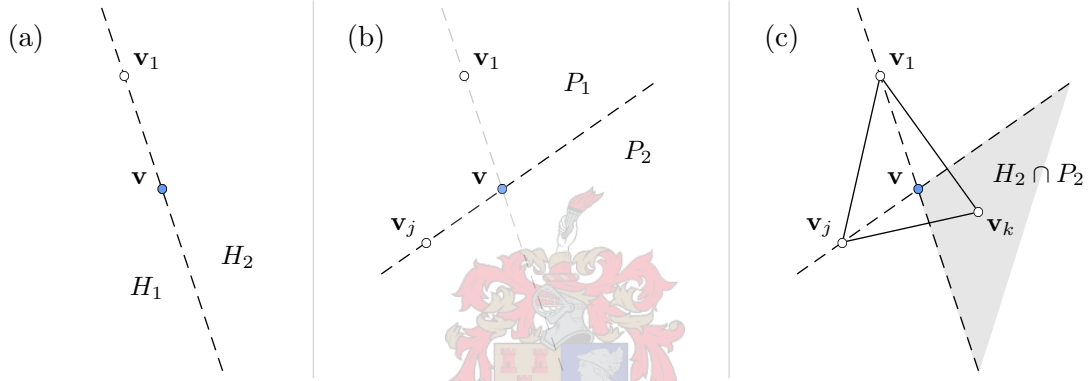


Figure 4.7

Consider the line passing through $\mathbf{v}_1$ and $\mathbf{v}$ that divides $\mathbb{R}^2$ into two open halfplanes, H_1 and H_2. See Figure 4.7(a). If follows from Corollary 4.3 that at least one of the vectors in $V \setminus \{\mathbf{v}_1\}$ lies in H_1 and at least one in H_2.

Consider a point of $V \setminus \{\mathbf{v}_1\}$ that lies in H_1, say $\mathbf{v}_j$. The line passing through $\mathbf{v}_j$ and $\mathbf{v}$ divides $\mathbb{R}^2$ into two open halfplanes, say P_1 and P_2. See Figure 4.7(b). Again, Corollary 4.3 states that at least one element of $V \setminus \{\mathbf{v}_j\}$ lies in P_1 and at least one in P_2.

It follows that there exists at least one point of $V \setminus \{\mathbf{v}_1, \mathbf{v}_j\}$ in the intersection of H_2 and P_2. Consider one of these points, say $\mathbf{v}_k$.

Clearly, the point $\mathbf{v}$ lies strictly inside the triangle with vertices $\mathbf{v}_1$, $\mathbf{v}_j$ and $\mathbf{v}_k$. See Figure 4.7(c). ■

The next section deals with a generalisation of 2D Tutte embedding. Some of the results of convex combinations proven here are used to prove that the algorithm of generalised 2D Tutte embedding always yields a valid straight line embedding of the graph G_M of a mesh $M \in \mathcal{M}_B$.

4.3 Generalised 2D Tutte embedding

In this section a generalisation of the 2D Tutte embedding algorithm, as briefly discussed in section 4.1, is investigated. In this generalisation every vertex is no longer positioned at the centroid, but rather at any strict convex combination of the positions of its neighbours.

Consider the graph G_M of a mesh $M \in \mathcal{M}_B$, with N vertices. There exists a cycle B in the graph G_M such that every vertex in B is a boundary vertex in M. All the other vertices of G_M will be called *interior vertices*. We denote the number of interior vertices by n, where $n < N$.

We assume, by relabelling the vertices if necessary, that $1, 2, \ldots, n$ are the labels of the interior vertices, and $n+1, n+2, \ldots, N$ are the labels of the boundary vertices.

The first step of the generalised 2D Tutte embedding algorithm is to map the boundary vertices to the vertices of some convex polygon Q in $\mathbb{R}^2$, such that the ordering of these vertices is preserved. Let these coordinates be given by $\mathbf{u}_i$, $i = n+1, n+2, \ldots, N$. One possibility is to choose Q as a regular $(N-n)$–gon, i.e.

$$\mathbf{u}_i = \begin{bmatrix} \cos[h(i-n-1)] \\ \sin[h(i-n-1)] \end{bmatrix}, \quad i = n+1, n+2, \ldots, N, \tag{4.22}$$

with $h = 2\pi/(N-n)$.

Next, for each interior vertex $i \in \{1, 2, \ldots, n\}$, weights w_{ij}, $j = 1, 2, \ldots, N$, are chosen such that

$$w_{ij} > 0, \quad (i,j) \in E(G_M), \quad \text{and} \quad w_{ij} = 0, \quad (i,j) \notin E(G_M), \tag{4.23}$$

and also

$$\sum_{j=1}^{N} w_{ij} = 1, \quad i = 1, 2, \ldots, n. \tag{4.24}$$

We define the points $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n \in \mathbb{R}^2$ to be the solutions of the following linear system of equations,

$$\mathbf{u}_i = \sum_{j=1}^{N} w_{ij}\mathbf{u}_j, \quad i = 1, 2, \ldots, n. \tag{4.25}$$

This yields a drawing of the graph G_M. Every vertex i of the graph G_M is positioned at the point $\mathbf{u}_i$ in the plane, for $i = 1, 2, \ldots, N$. Edges are drawn as segments of straight lines between endpoints. We will call this drawing a *generalised 2D Tutte embedding* of G_M.

Note that with the weights chosen to satisfy (4.23) and (4.24), every interior vertex is positioned at a strict convex combination of the positions of its neighbours.

The question remains whether this embedding is always a valid straight line embedding. To prove that it is we have to show that the points $\mathbf{u}_i$, $i = 1, 2, \ldots, n$, as given by (4.25), are well-defined and distinct, and that no edges intersect in the embedding.

First, it is shown that the linear system (4.25) has a unique solution. In order to accomplish this, (4.25) is rewritten as

$$\mathbf{u}_i - \sum_{j=1}^{n} w_{ij}\mathbf{u}_j = \sum_{j=n+1}^{N} w_{ij}\mathbf{u}_j, \quad i = 1, 2, \ldots, n. \tag{4.26}$$

By considering the two components of $\mathbf{u}_i$, say x_i and y_i, separately, the equations in (4.26) are equivalent to the following two matrix equations,

$$A\mathbf{x} = \mathbf{b}_1, \quad \text{and} \quad A\mathbf{y} = \mathbf{b}_2, \tag{4.27}$$

where $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, $\mathbf{y} = [y_1, y_2, \ldots, y_n]^T$, and the matrix A is $n \times n$ with elements

$$\{A\}_{ij} = \begin{cases} 1, & i = j \\ -w_{ij}, & i \neq j, \end{cases} \quad i, j = 1, 2, \ldots, n. \tag{4.28}$$

It follows from (4.26) that $\mathbf{b}_1 = [b_{1,1}, b_{1,2}, \ldots, b_{1,n}]^T$ and that $\mathbf{b}_2 = [b_{2,1}, b_{2,2}, \ldots, b_{2,n}]^T$ have elements

$$b_{1,i} = \sum_{j=n+1}^{N} w_{ij}x_j, \quad \text{and} \quad b_{2,i} = \sum_{j=n+1}^{N} w_{ij}y_j, \quad i = 1, 2, \ldots, n. \tag{4.29}$$

Existence and uniqueness of a solution to the system (4.25) is therefore equivalent to the non-singularity of the matrix A.

Theorem 4.5 : *The matrix A, as defined by (4.28), is non-singular.*

Proof

We prove the theorem by showing that the only solution of $A\mathbf{v} = \mathbf{0}$ is $\mathbf{v} = \mathbf{0}$. Let $\mathbf{v} = [v_1, v_2, \ldots, v_n]^T$. The equation $A\mathbf{v} = \mathbf{0}$ may be written as

$$v_i - \sum_{j=1}^{N} w_{ij}v_j = 0, \quad i = 1, 2, \ldots, n, \tag{4.30}$$

with $v_{n+1} = v_{n+2} = \ldots = v_N = 0$. Let $v_{\max}$ be the maximum of $v_1, v_2, \ldots, v_n$, and suppose $v_{\max} = v_k$, for some $k \in \{1, 2, \ldots, n\}$.

Consider any vertex $j \in N(k)$. Since $v_k = v_{\max}$, we have $v_j \leq v_k$. But if $v_j < v_k$, then (4.30) together with (4.23) and (4.24), imply that there must exist a vertex $\ell \in N(k)$ such that $v_\ell > v_k$, which is impossible. Therefore $v_j = v_k = v_{\max}$.

In a similar way, every neighbour i of vertex j must satisfy $v_i = v_{\max}$, and so on.

Since the graph G_M is connected, a boundary vertex is eventually reached, with the result that $v_j = v_{\max}$ for some $j \in \{n+1, n+2, \ldots, N\}$. Therefore $v_{\max} = 0$.

A similar argument shows that for the minimum of $v_1, v_2, \ldots, v_n$, denoted by $v_{\min}$, we have $v_{\min} = 0$, and therefore $\mathbf{v} = \mathbf{0}$. Hence the theorem holds. ■

Theorem 4.5 implies that (4.25) has a unique solution. Before we prove that this solution yields a valid straight line embedding, the following result is needed.

Lemma 4.6 : *If every internal vertex i is positioned at $\mathbf{u}_i$, where $\mathbf{u}_i$ are solutions of (4.25), $i \in \{1, 2, \ldots, n\}$, then every internal vertex lies strictly inside the convex polygon Q with vertices at $\mathbf{u}_{n+1}, \mathbf{u}_{n+2}, \ldots, \mathbf{u}_N$.*

Proof

Suppose at least one internal vertex does not lie strictly inside Q. Let $\mathbf{u}_i$ be any such vertex whose shortest distance to Q is maximal. Then $\mathbf{u}_i$ either lies strictly outside of Q, or on the boundary of Q.

First, suppose $\mathbf{u}_i$ lies strictly outside of Q. Let $\mathbf{v}$ be the point on the boundary of Q nearest to $\mathbf{u}_i$. The point $\mathbf{v}$ is either a vertex of Q, or a point on one of the edges of Q.

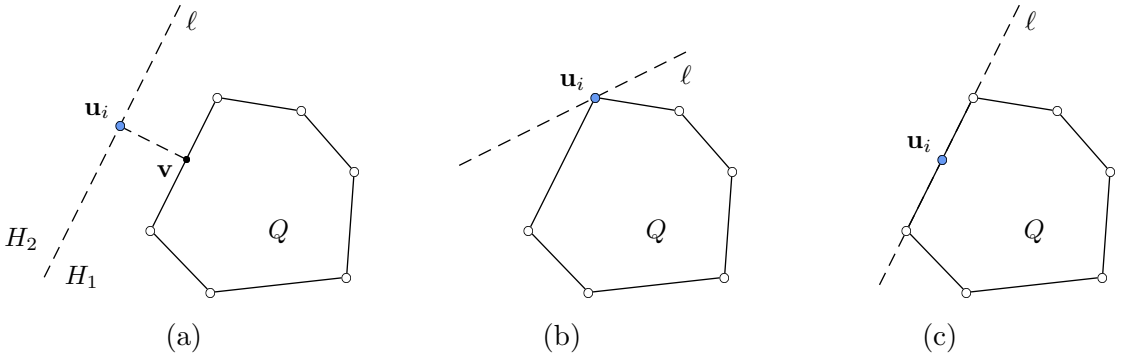


Figure 4.8

Let ℓ denote the line passing through $\mathbf{u}_i$, perpendicular to $\mathbf{u}_i - \mathbf{v}$. This line divides $\mathbb{R}^2$ into two open halfplanes, say H_1 and H_2 such that $\mathbf{v} \in H_1$. Note that because $\mathbf{u}_i$ lies outside

of Q and Q is convex, the vertices of Q all belong to H_1. See for example Figure 4.8(a).

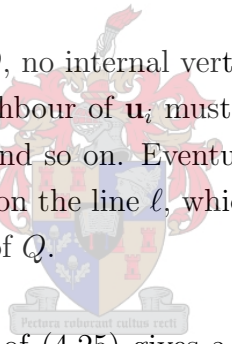
Since $\mathbf{u}_i$ has maximal shortest distance to Q, it follows that $\mathbf{u}_j \notin H_2$, $j = 1, 2, \ldots, n$. But $\mathbf{u}_i$ is a strict convex combination of its neighbours, thus, according to Corollary 4.3, the neighbours of $\mathbf{u}_i$ must all lie on the line ℓ.

Similarly, the neighbours of the neighbours of $\mathbf{u}_i$ must all lie on ℓ, and so on. Since the graph G_M is connected, the vertices of Q must all lie on ℓ, which results in a contradiction. Therefore no internal vertices lie strictly outside of Q.

Next, suppose $\mathbf{u}_i$, the point with maximal distance from Q, lies on the boundary of Q. Then $\mathbf{u}_i$ is either a vertex of Q, or a point on one of the edges of Q.

If $\mathbf{u}_i$ coincides with a vertex of Q, let ℓ be any line through $\mathbf{u}_i$, but not through Q, as in Figure 4.8(b). If $\mathbf{u}_i$ is a point on an edge of Q, let ℓ be the infinite extension of that edge, as shown in Figure 4.8(c).

Since $\mathbf{u}_i$ has maximal distance from Q, no internal vertex lies strictly outside of Q. Thus, according to Corollary 4.3, every neighbour of $\mathbf{u}_i$ must lie on ℓ. Also, every neighbour of every neighbour of $\mathbf{u}_i$ must lie on ℓ, and so on. Eventually the vertices of Q are reached, implying that all the vertices of Q lie on the line ℓ, which is a contradiction. Therefore no internal vertices lie on the boundary of Q. ■



It remains to prove that the solution of (4.25) gives a valid straight line embedding, i.e. that no edges intersect. Note that an edge intersection implies that at least two triangular faces overlap in the embedding. The converse is also true: overlapping faces imply edge intersections.

The following theorem states that the solution of (4.25) results in an embedding where none of the faces overlap, which is equivalent to stating that a valid straight line embedding is obtained. The proof of this theorem is partly due to Colin de Verdière, Pocchiola and Vegter [4].

Theorem 4.7 : *The solution of (4.25) yields a planar embedding of G_M in which no triangles overlap.*

Proof

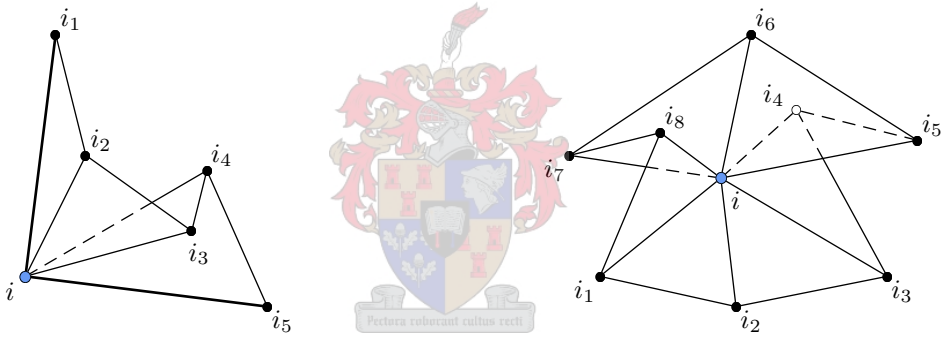
Let $B(G_M)$ denote the set of $N - n$ boundary vertices, and $I(G_M)$ the set of n interior vertices of G_M. We denote the boundary polygon (with vertices at $\mathbf{u}_{n+1}, \mathbf{u}_{n+2}, \ldots, \mathbf{u}_N$) by Q.

Let the function $\alpha(i) : V(G_M) \mapsto \mathbb{R}$ be defined as

$$\alpha(i) = \begin{cases} 2\pi, & i \in I(G_M), \\ \text{the angle of } Q \text{ at } i, & i \in B(G_M), \end{cases} \qquad i \in V(G_M). \tag{4.31}$$

Let the function $\beta(i) : V(G_M) \mapsto \mathbb{R}$ be defined such that $\beta(i)$ gives the sum over all triangles incident to vertex i of the angle of such a triangle at i. We aim to show that $\alpha(i) = \beta(i)$, for all $i \in V(G_M)$.

Consider a fixed vertex $i \in V(G_M)$. The vertices adjacent to i form a cycle if $i \in I(G_M)$, and a path if $i \in B(G_M)$. We denote the neighbours of i by $i_1, i_2, \ldots, i_p$, in the order of this cycle or path. See for example Figure 4.9. Note that for an interior vertex i, (4.25) implies that i is positioned at a strict convex combination of the positions of the vertices $i_1, i_2, \ldots, i_p$.



(a) path adjacent to boundary vertex i (b) cycle adjacent to interior vertex i

Figure 4.9

First, we prove that $\alpha(i) \leq \beta(i)$, with equality if and only if the triangles incident to i do not overlap. To this end, the following notation is introduced. For a triangle with vertices u, v and w, let $\sigma(u, v, w)$ be the angle of the triangle at v, with $0 \leq \sigma(u, v, w) \leq \pi$.

Suppose $i \in B(G_M)$. Let a and b denote the labels of the two boundary vertices adjacent to i. According to Lemma 4.6, every interior vertex lies strictly inside the polygon Q. Therefore

$$\sum_{j=1}^{p-1} \sigma(i_j, i, i_{j+1}) \geq \sigma(a, i, b), \quad i \in B(G_M), \tag{4.32}$$

and $\beta(i) \geq \alpha(i)$, $i \in B(G_M)$.

Next, consider an interior vertex $i \in I(G_M)$, with neighbours $i_1, i_2, \ldots, l_p$. It follows from Corollary 4.4 that there exist integers a, b and c with $1 \leq a < b < c \leq p$ such that vertex

i lies strictly inside the triangle with vertices labelled i_a, i_b and i_c. Therefore

$$\sigma(i_a, i, i_b) + \sigma(i_b, i, i_c) + \sigma(i_c, i, i_a) = 2\pi. \tag{4.33}$$

We also have

$$\sum_{j=a}^{b-1} \sigma(i_j, i, i_{j+1}) \geq \sigma(i_a, i, i_b), \tag{4.34}$$

$$\sum_{j=b}^{c-1} \sigma(i_j, i, i_{j+1}) \geq \sigma(i_b, i, i_c), \tag{4.35}$$

$$\sum_{j=c}^{p+a-1} \sigma(i_j, i, i_{j+1}) \geq \sigma(i_c, i, i_a), \tag{4.36}$$

where we define $i_{j+cp} = i_j$, $c \in \mathbb{Z}$. Adding these three inequalities yields

$$\sum_{j=1}^{p-1} \sigma(i_j, i, i_{j+1}) \geq \sigma(i_a, i, i_b) + \sigma(i_b, i, i_c) + \sigma(i_c, i, i_a) = 2\pi, \tag{4.37}$$

and thus $\beta(i) \geq \alpha(i)$, $i \in I(G_M)$.

Since $\alpha(i) \leq \beta(i)$ for every boundary vertex and every interior vertex i, we have

$$\alpha(i) \leq \beta(i), \quad i \in V(G_M). \tag{4.38}$$

Moreover, it is clear from Figure 4.9 that equality holds only if the triangles incident to vertex i do not overlap.

In order to prove that equality does hold in (4.38), let t denote the total number of triangular faces of the original mesh M. Then

$$\sum_{i=1}^{N} \beta(i) = \pi t, \tag{4.39}$$

and

$$\sum_{i=1}^{N} \alpha(i) = 2\pi n + \pi(N - n - 2) = \pi(N + n - 2). \tag{4.40}$$

Since the mesh M belongs to $\mathcal{M}_B$, it follows from Corollary 2.7 that

$$N - e + t = 1, \tag{4.41}$$

where e denotes the number of edges in G_M. Because all the faces are triangular, $e = (3t + N - n)/2$, and equation (4.41) becomes

$$N - \frac{3t + N - n}{2} + t = 1, \tag{4.42}$$

which simplifies to

$$N + n - 2 = t. \tag{4.43}$$

Substituting (4.43) into (4.40) and using also (4.39) yield

$$\sum_{i=1}^{N} \alpha(i) = \pi t = \sum_{i=1}^{N} \beta(i). \tag{4.44}$$

But since $0 < \alpha(i) \leq \beta(i)$, $i \in V(G_M)$, it follows that $\alpha(i) = \beta(i)$, $i \in V(G_M)$. Hence no triangles overlap, and the theorem holds. ■

Theorems 4.5 and 4.7 imply that the generalised 2D Tutte embedding algorithm always results in a valid planar embedding for the graph G_M of any mesh $M \in \mathcal{M}_B$, provided that the weights satisfy (4.23) and (4.24), and that the boundary vertices are positioned at the vertices of a convex polygon.

Note that Tutte's original algorithm [33] as briefly explained in section 4.1, is a special case of this algorithm where the weights are chosen as

$$w_{ij} = \begin{cases} \dfrac{1}{\deg(i)}, & (i,j) \in E(G_M), \\ 0, & \text{otherwise}, \end{cases} \qquad i,j \in V(G_M). \tag{4.45}$$

The next section gives some more possibilities for choosing weights.

4.4 Choosing weights

In the previous section, the generalised 2D Tutte algorithm was given and proven to be correct under the condition that the weights satisfy (4.23) and (4.24). This section provides a number of choices for these weights.

4.4.1 Tutte weights

An obvious choice of weights is given by (4.45). Since Tutte used these weights in his 2D planar embedding algorithm [33], we call these weights *Tutte weights*. It follows directly from (2.3) that (4.24) is satisfied.

4.4.2 Chord length weights

Another choice is *chord length weights*, where every weight is scaled to represent the Euclidean length of the corresponding edge in the original mesh.

These weights are defined as

$$w_{ij} = \frac{\omega_{ij}}{\text{wdeg}(i)}, \quad (i,j) \in E(G_M), \quad \text{and} \quad w_{ij} = 0, \quad (i,j) \notin E(G_M), \tag{4.46}$$

where

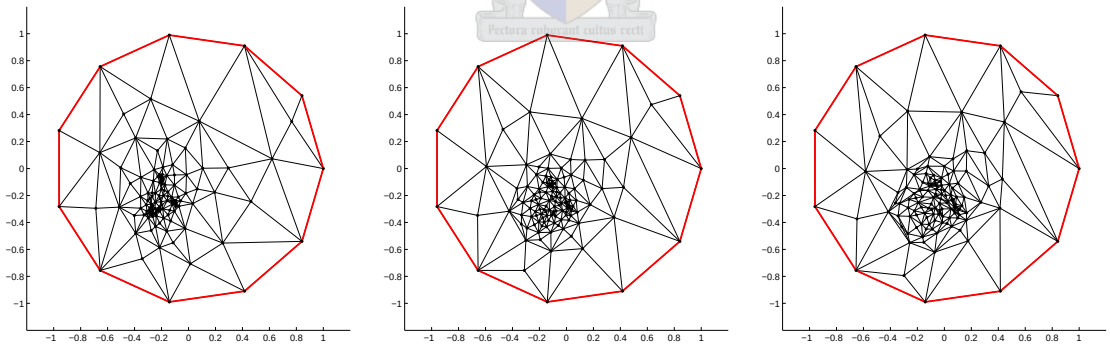
$$\omega_{ij} = ||\mathbf{x}_i - \mathbf{x}_j||^\rho, \quad i,j \in V(G_M), \tag{4.47}$$

with $\rho \in \mathbb{R}$, and

$$\text{wdeg}(i) = \sum_{j \in N(i)} \omega_{ij}, \quad i \in V(G_M). \tag{4.48}$$

The position vector $\mathbf{x}_i$, $i \in V(G_M)$, denotes the coordinates of node i in the original mesh M. Note that Tutte weights are the special case of $\rho = 0$. Choosing for example $\rho = -1$ reduces the influence that longer edges have on the position of a vertex, while $\rho = 1$ increases this influence.

Figure 4.10 shows the result of embedding the graph G_M of the mesh M from Figure 4.2(a), using different values of ρ in (4.47).



$\rho = -1$ $\rho = 1$ $\rho = 2$

Figure 4.10

4.4.3 Other weights

A number of alternative choices for weights may be found in the literature. Floater [9] proposed the so-called *shape-preserving weights* which, as its name suggests, attempts to attain the shape of the original mesh relatively, by minimising a distortion function.

In [10], Floater proposed the *mean-value weights*. These weights are derived from the mean-value theorem for harmonic functions. They also attain certain shape-parameters of the original mesh.

For the remainder of this thesis, we will make use of chord length weights only.

4.5 An iterative scheme for 2D Tutte embedding

In section 4.3 it was shown that a valid straight line embedding of the graph G_M can be obtained by solving a linear system. This section deals with the issue of numerical efficiency that arises in the implementation of the generalised 2D Tutte algorithm.

From a computational viewpoint, solving the two systems in (4.27),

$$A\mathbf{x} = \mathbf{b}_1, \quad \text{and} \quad A\mathbf{y} = \mathbf{b}_2, \tag{4.49}$$

by a direct method for a graph with thousands of vertices is not desirable. The matrix A is sparse, but it is not, in general, possible to rearrange the non-zero entries into a band structure. The sparseness of A suggests that an iterative method should be more computationally efficient.

For this purpose, we use Gauss-Seidel iteration, which is defined next. Consider a square linear system $C\mathbf{v} = \mathbf{d}$. The matrix C is split up as

$$C = L + D + U, \tag{4.50}$$

where L denotes the strictly lower triangular part, D the diagonal part, and U the strictly upper triangular part of C. The linear system is then written as

$$\begin{aligned} (L + D + U)\mathbf{v} &= \mathbf{d} \\ (L + D)\mathbf{v} &= -U\mathbf{v} + \mathbf{d}. \end{aligned} \tag{4.51}$$

This leads to the following iteration scheme,

$$(L + D)\mathbf{v}^{(r+1)} = -U\mathbf{v}^{(r)} + \mathbf{d}, \quad r = 1, 2, \ldots, \tag{4.52}$$

where $\mathbf{v}^{(0)}$ is some chosen initial value. Note that since $L + D$ is a lower triangular matrix, solving for $\mathbf{v}^{(r+1)}$ in (4.52) is a matter of forward substitution. The iteration (4.52) is known as *Gauss-Seidel iteration*. If (4.52) converges, it converges to the solution $\mathbf{v} = C^{-1}\mathbf{d}$.

The following well-known theorem gives a condition under which (4.52) converges. A proof may for example be found in [12, p. 512].

Theorem 4.8 : *If C is symmetric and positive definite, then the Gauss-Seidel iteration (4.52) converges for any $\mathbf{v}^{(0)}$.*

Consider the two linear systems in (4.49). We assume, for the purpose of the arguments that follow, that A was constructed using chord length weights (see section 4.4.2), for some $\rho \in \mathbb{R}$.

In order to apply Theorem 4.8 to the linear system (4.49), we first rewrite the system as follows. Let D be the $n \times n$ diagonal matrix, with diagonal entries $\{D\}_{ii} = \text{wdeg}(i)$, $i = 1, 2, \ldots, n$. Then

$$B\mathbf{x} = \mathbf{c}_1, \quad \text{and} \quad B\mathbf{y} = \mathbf{c}_2, \tag{4.53}$$

with $B = DA$, $\mathbf{c}_1 = D\mathbf{b}_1$ and $\mathbf{c}_2 = D\mathbf{b}_1$. Note that from (4.47), we have $\omega_{ij} = \omega_{ji}$, and therefore, B is symmetric.

We aim to show that B is also positive definite. The following theorem, known as *Gerschgorin's circle theorem*, identifies a region in the complex plane that contains all the eigenvalues of a given square matrix. The theorem was originally proposed and proved in [11]. A proof may also be found in [12, p. 320].

Theorem 4.9 : (Gerschgorin's circle theorem) *For an $n \times n$ matrix A, with $\{A\}_{ij} = a_{ij}$, let*

$$R_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|, \quad i = 1, 2, \ldots, n.$$

Then every eigenvalue of A lies in the set Λ defined as

$$\Lambda = \bigcup_{i=1}^{n} \{z \in \mathbb{C} : |z - a_{ii}| \leq R_i\}.$$

The following proposition states that the Gauss-Seidel iteration method converges for the two linear systems (4.53) for any initial values $\mathbf{x}^{(0)}$ and $\mathbf{y}^{(0)}$.

Proposition 4.10 : *For the linear systems in (4.53), the Gauss-Seidel iteration converges, for any $\mathbf{x}^{(0)} \in \mathbb{R}^n$ and $\mathbf{y}^{(0)} \in \mathbb{R}^n$.*

Proof

We have already established that the matrix B is symmetric. It is well-known that the

eigenvalues of a symmetric matrix are real (see for instance [2, p. 376]). Thus the eigenvalues of B, denoted by $\lambda_1, \lambda_2, \ldots, \lambda_n$, are all real numbers.

Applying Gerschgorin's circle theorem and using also (4.46) and (4.48) yield

$$R_i = \sum_{j \in N(i)} \omega_{ij} = \text{wdeg}(i). \tag{4.54}$$

Since $\{B\}_{ii} = \text{wdeg}(i)$, $i = 1, 2, \ldots, n$, it follows that $\lambda_i \geq 0$, $i = 1, 2, \ldots, n$.

But from Theorem 4.5, A is non-singular, and therefore B is also non-singular. Thus $\lambda_i \neq 0$, $i = 1, 2, \ldots, n$, implying that $\lambda_i > 0$, $i = 1, 2, \ldots, n$. Therefore, B is positive definite.

It follows from Theorem 4.8 that the Gauss-Seidel iteration converges for any $\mathbf{x}^{(0)} \in \mathbb{R}^n$ and $\mathbf{y}^{(0)} \in \mathbb{R}^n$. ■

Therefore, the iteration scheme given by

$$(L + D)\mathbf{x}^{(r+1)} = -U\mathbf{x}^{(r)} + \mathbf{c}_1, \quad (L + D)\mathbf{y}^{(r+1)} = -U\mathbf{y}^{(r)} + \mathbf{c}_2, \quad r = 1, 2, \ldots, \tag{4.55}$$

with $B = L + D + U$, and where $\mathbf{x}^{(0)}$ and $\mathbf{y}^{(0)}$ are some chosen initial values, always converges, and may be used in solving for $\mathbf{x}$ and $\mathbf{y}$ in (4.49).

Figure 4.11 illustrates this iteration, for the example mesh from Figure 4.2(a). We chose $\mathbf{x}^{(0)} = \mathbf{y}^{(0)} = \mathbf{0}$, and used Tutte weights (i.e. chord length weights with $\rho = 0$). Clearly, the iteration approaches the exact solution depicted in Figure 4.2(b).

It should be noted that we have only proved convergence in the case where the weights are chosen as chord length weights, and not for all weights satisfying (4.23) and (4.24). For arbitrary weights satisfying these conditions, the corresponding matrix in (4.49) is not necessarily symmetric and positive definite.

In the following chapter, we attempt to extend the notion of 2D Tutte embedding to the surface of the sphere, to ultimately obtain a valid GC-embedding of the underlying graph of a mesh.

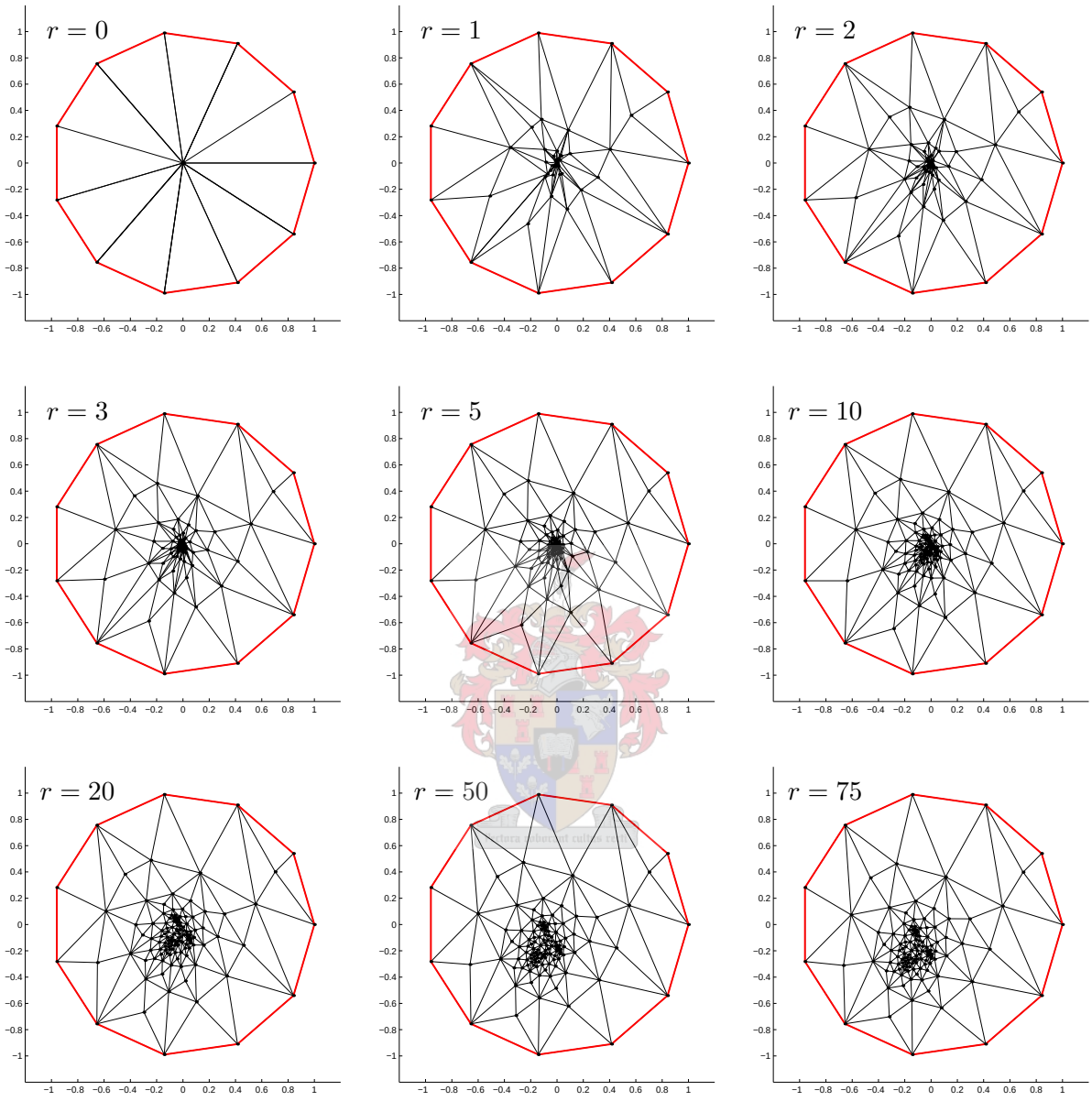
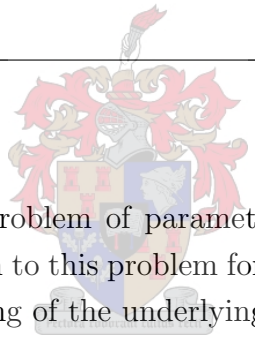


Figure 4.11

CHAPTER 5

Iterative methods for spherical embedding



In this chapter, we return to the problem of parameterisation as defined in Chapter 3. Proposition 3.3 states that a solution to this problem for a mesh $M \in \mathcal{M}$, may be obtained by first finding a valid GC-embedding of the underlying graph G_M.

Recall from Definition 3.2 that a valid GC-embedding of G_M assigns distinct coordinates $\mathbf{v}_i \in S_0$ to every vertex $i \in V(G_M)$, such that when we draw all the edges as minor arcs joining endpoints, none of them intersect.

This chapter gives a brief discussion of some existing methods to find valid GC-embeddings of an underlying graph.

In Chapter 4, the generalized 2D Tutte embedding algorithm for finding valid planar embeddings of a graph G_M, where $M \in \mathcal{M}_B$, was discussed. In the following section, we attempt to extend that algorithm to find a valid GC-embedding of the underlying graph of a mesh in $\mathcal{M}$.

5.1 Spherical Tutte embedding

The generalised 2D Tutte embedding algorithm involves the positioning of every vertex at some convex combination of its neighbours. When extending this idea to the surface of the

sphere, there are two aspects to be considered. First, a boundary is no longer of any use, since the surface of the sphere is a closed set.

Second, every vertex can no longer be positioned at a convex combination of its neighbours, but rather at a projection of this combination on the surface of the unit sphere. This projection is a radial projection through the center of the sphere.

Figure 5.1 illustrates this notion. The figure shows the unit sphere centred at the origin with four neighbours located at $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ of a vertex i. The centroid of the four neighbours is denoted by $\mathbf{u}$, hence

$$\mathbf{u} = \tfrac{1}{4}(\mathbf{a} + \mathbf{b} + \mathbf{c} + \mathbf{d}). \tag{5.1}$$

This point lies inside the sphere. The position of vertex i, denoted by $\mathbf{v}_i$, is then taken to be the projection of $\mathbf{u}$ on the surface of the sphere, i.e.

$$\mathbf{v}_i = \frac{\mathbf{u}}{||\mathbf{u}||}. \tag{5.2}$$

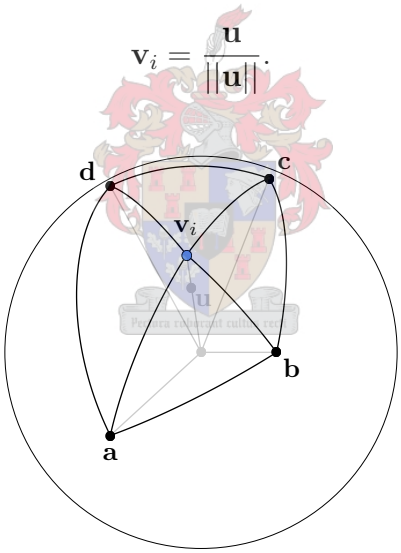


Figure 5.1

For a mesh $M \in \mathcal{M}$, the goal is therefore to obtain a GC-drawing of the graph G_M on the surface of the unit sphere, such that every vertex is positioned at the projection of some convex combination of its neighbours. We will refer to such a drawing as a *spherical Tutte embedding* of G_M.

5.1.1 The non-linear system for spherical Tutte embedding

In this section a system of non-linear equations is set up. A non-trivial solution of this system would then give a spherical Tutte embedding.

Consider a given mesh $M \in \mathcal{M}$ with N vertices. As with 2D Tutte embedding, weights w_{ij} are assigned such that for each vertex $i \in V(G_M)$,

$$w_{ij} > 0, \quad (i,j) \in E(G_M), \quad \text{and} \quad w_{ij} = 0, \quad (i,j) \notin E(G_M), \tag{5.3}$$

and also

$$\sum_{j=1}^{N} w_{ij} = 1. \tag{5.4}$$

The points $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N \in \mathbb{R}^3$ are defined to be solutions of the following non-linear system of equations,

$$\mathbf{v}_i = \frac{\mathbf{u}_i}{||\mathbf{u}_i||}, \quad \text{with} \quad \mathbf{u}_i = \sum_{j=1}^{n} w_{ij}\mathbf{v}_j, \quad i = 1, 2, \ldots, N. \tag{5.5}$$

Note that $\mathbf{v}_i \in S_0$, $i = 1, 2, \ldots, N$, and every vertex is a projection of a convex combination of its neighbours. A solution to (5.5) therefore yields a spherical Tutte embedding of G_M, where each vertex i is positioned at point $\mathbf{v}_i$, $i = 1, 2, \ldots, N$.

The question of whether this embedding is in fact a valid GC-embedding of G_M still remains. Before this question is addressed, (5.5) is rewritten in another form. Note that $\mathbf{u}_i$ is a scalar multiple of $\mathbf{v}_i$, for each $i \in \{1, 2, \ldots, N\}$. It is therefore possible to write

$$\mathbf{u}_i = \alpha_i \mathbf{v}_i, \quad \text{where} \quad \alpha_i = ||\mathbf{u}_i||, \quad i = 1, 2, \ldots, N. \tag{5.6}$$

If the $N \times 3$ matrix U is defined as the matrix with row i equal to $\mathbf{u}_i^T$, and the $N \times 3$ matrix V is defined as the matrix with row i equal to $\mathbf{v}_i^T$, $i = 1, 2, \ldots, N$, then

$$U = AV, \tag{5.7}$$

where A is the diagonal $N \times N$ matrix with diagonal elements $\{A\}_{ii} = \alpha_i$, $i = 1, 2, \ldots, N$. But, from (5.5), we also have

$$U = WV, \tag{5.8}$$

where the $N \times N$ matrix W has elements

$$\{W\}_{ij} = w_{ij}, \quad i, j = 1, 2, \ldots, N. \tag{5.9}$$

Combining (5.7) and (5.8), and forcing the vertices to lie on the surface of the sphere yield the following non-linear system of equations,

$$WV = AV, \quad \text{such that} \quad ||\mathbf{v}_i|| = 1, \quad i = 1, 2, \ldots, N, \tag{5.10}$$

with $\mathbf{v}_i^T$ equal to the ith row of V. The first equation, $WV = AV$, states that every vertex must be positioned at some dilation of a convex combination of its neighbours. The second set of equations ensures that every vertex is positioned on the surface of the unit sphere.

A non-trivial solution of (5.10), for A and V, therefore yields a spherical Tutte embedding of G_M. Note that (5.10) represents $4N$ equations in $4N$ unknowns. The unknowns are the coordinates $\mathbf{v}_i \in \mathbb{R}^3$ and the value of $\alpha_i = \{A\}_{ii}$ for every vertex $i \in \{1, 2, \ldots, N\}$.

A trivial solution of (5.10) is a case where every vertex is positioned at the same point on the surface of the unit sphere, that is, the solution $\mathbf{v}_i = \mathbf{q}$, where $\mathbf{q} \in S_0$ is a fixed point, and $\alpha_i = 1$, for all $i \in \{1, 2, \ldots, N\}$.

The following section discusses the question of whether a spherical Tutte embedding is in fact a valid GC-embedding. In subsequent sections, the matter of solving the system (5.10) will be considered.

5.1.2 Correctness of the non-linear system

The following theorem states that any spherical embedding of G_M where every vertex is distinctly positioned at the projection of some convex combination of its neighbours is a valid GC-embedding of G_M. A direct consequence of this theorem is that any non-trivial solution to (5.10) is in fact a valid GC-embedding of G_M.

The proof of this theorem relies on results from spectral graph theory, and falls beyond the scope of this thesis. A proof may be found in Gotsman, Gu and Sheffer [16].

Theorem 5.1 : *Given the graph G_M of a mesh $M \in \mathcal{M}$, a valid GC-embedding is obtained if and only if each vertex of G_M is distinctly positioned at the projection on the surface of the unit sphere of some convex combination of the positions of its neighbours.*

Theorem 5.1 implies that a non-trivial solution of (5.10) forms a valid GC-embedding, provided such a solution exists. The problem of finding a solution of this non-linear system is addressed in the following section.

5.1.3 Solving the non-linear system

Solving (5.10) is not straightforward. Gotsman et. al. [16] proposed using the MATLAB

procedure `fsolve`, which is a numerical method for solving non-linear systems of equations. This method uses preconditioned gradients and is based on Newton's method.

When implementing this procedure, an initial guess of the solution is needed. Initial values for $\mathbf{v}_i$ may be obtained by centering the mesh at the origin, and then normalising every vertex. Thus

$$\mathbf{v}_i^{(0)} = \frac{\mathbf{x}_i - \mathbf{m}}{||\mathbf{x}_i - \mathbf{m}||}, \quad i = 1, 2, \ldots, N, \tag{5.11}$$

where $\mathbf{x}_i$ denotes the coordinates of vertex i in the original mesh, and $\mathbf{m}$ is some point inside the mesh, usually taken to be the mean of $\mathbf{x}_i$, $i = 1, 2, \ldots, N$. It should be noted that for some meshes, such a point may not be easy to obtain.

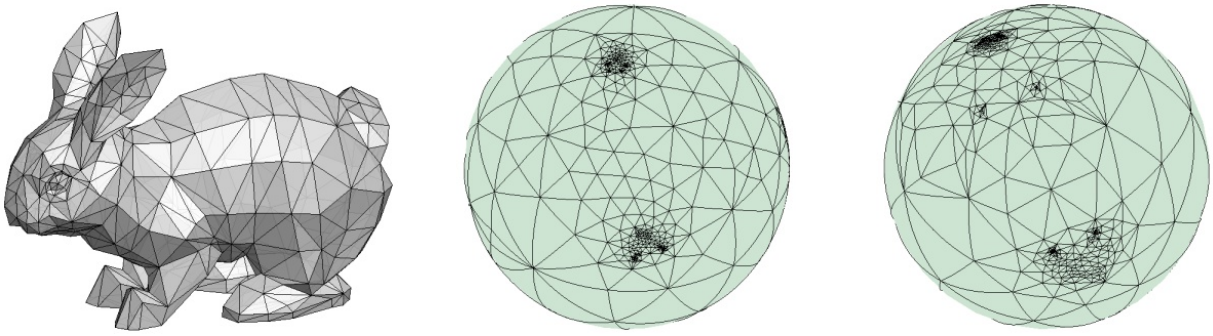
Gotsman et. al. proposed setting the initial value of α_i to

$$\alpha_i^{(0)} = \frac{4\pi}{\sqrt{3}N}, \quad i = 1, 2, \ldots, N, \tag{5.12}$$

which represents the average total curvature at the points on the sphere [16].

Solving (5.10) with `fsolve`, using initial guesses (5.11) and (5.12), is hereafter referred to as *Gotsman's method* for finding a valid GC-embedding of the graph G_M.

Figure 5.2 shows the result of applying Gotsman's method to the rabbit model, for two different choices of weights, namely Tutte weights and chord length weights with $\rho = 1$ (see section 4.4).



(a) original model (b) Tutte weights (c) chord length weights

Figure 5.2

The problem with Gotsman's method is that it becomes computationally expensive even for meshes with only a few hundred vertices. The example above required about 17 minutes for one embedding.

Sections 5.2 and 5.3 provide two iterative schemes based on heuristic arguments that aim to obtain solutions to (5.10) faster than Gotsman's method.

5.2 The method of iterative relaxation

In this section an iterative method for finding a spherical Tutte embedding of the graph G_M of a mesh $M \in \mathcal{M}$, is discussed. Inspired by the iterative scheme for obtaining 2D Tutte embeddings (section 4.5), this method involves the iterative replacement of the position of every vertex with the projection of a convex combination of its neighbours, thereby hopefully converging to a non-trivial solution of (5.10).

For a given mesh $M \in \mathcal{M}$ with N vertices, let the $N \times 3$ matrix $V^{(r)}$ contain the position vectors of the vertices at iteration step $r \in \{1, 2, \ldots\}$, such that the ith row of $V^{(r)}$, denoted by $[\mathbf{v}_i^{(r)}]^T$, yields the position vector of vertex $i \in \{1, 2, \ldots, N\}$, at step r.

The iteration is started with an initial embedding of the graph on the surface of the unit sphere. The initial embedding from Gotsman's method is used, therefore

$$\mathbf{v}_i^{(0)} = \frac{\mathbf{x}_i - \mathbf{m}}{||\mathbf{x}_i - \mathbf{m}||}, \quad i = 1, 2, \ldots, N, \tag{5.13}$$

where $\mathbf{x}_i$ denotes the coordinates of vertex i in the original mesh M, and $\mathbf{m} \in \mathbb{R}^3$ is some point in the interior of the mesh M. As with Gotsman's method, obtaining a valid choice for $\mathbf{m}$ may be difficult in some cases. A possibility is to let $\mathbf{m}$ be the arithmetic mean of the nodes of M, i.e.

$$\mathbf{m} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i. \tag{5.14}$$

For some "extremely concave" meshes, however, this point may not lie in the interior. In such a case, the point $\mathbf{m}$ may be user-specified.

Next, weights w_{ij} are chosen to satisfy (5.3) and (5.4). The coordinates of the vertices at every iteration step is defined recursively by

$$\mathbf{v}_i^{(r+1)} = \frac{\mathbf{u}_i^{(r)}}{||\mathbf{u}_i^{(r)}||}, \quad \text{where} \quad U^{(r)} = WV^{(r)}, \quad i = 1, 2, \ldots, N, \quad r = 1, 2, \ldots, \tag{5.15}$$

with W the $N \times N$ matrix defined by (5.9), and where $[\mathbf{u}_i^{(r)}]^T$ gives the ith row of the matrix $U^{(r)}$. For each iteration, the position of every vertex is updated to be the projection of a convex combination of its neighbours.

The process of replacing vertices by convex combinations of neighbours is known as *relaxation*. For this reason, the algorithm defined by (5.13) and (5.15) will be called *the method of iterative relaxation*.

If the only objective is to find a valid GC-embedding of the graph G_M, the iteration may be terminated as soon as such an embedding is reached. The orientation test from section 3.4.1, or the area test from section 3.4.2 may be used to determine whether a given spherical drawing of G_M is a valid GC-embedding. Testing a given spherical drawing may become computationally expensive if the mesh has thousands of faces. For this reason, we opt to test for the validity of the embedding every R iterations, where R is user-specified.

Figure 5.3 shows the result of applying the method of iterative relaxation to the rabbit model. Embeddings after certain numbers of iterations are shown. The point $\mathbf{m}$ was calculated according to (5.14), and Tutte weights (from section 4.4.1) were used.

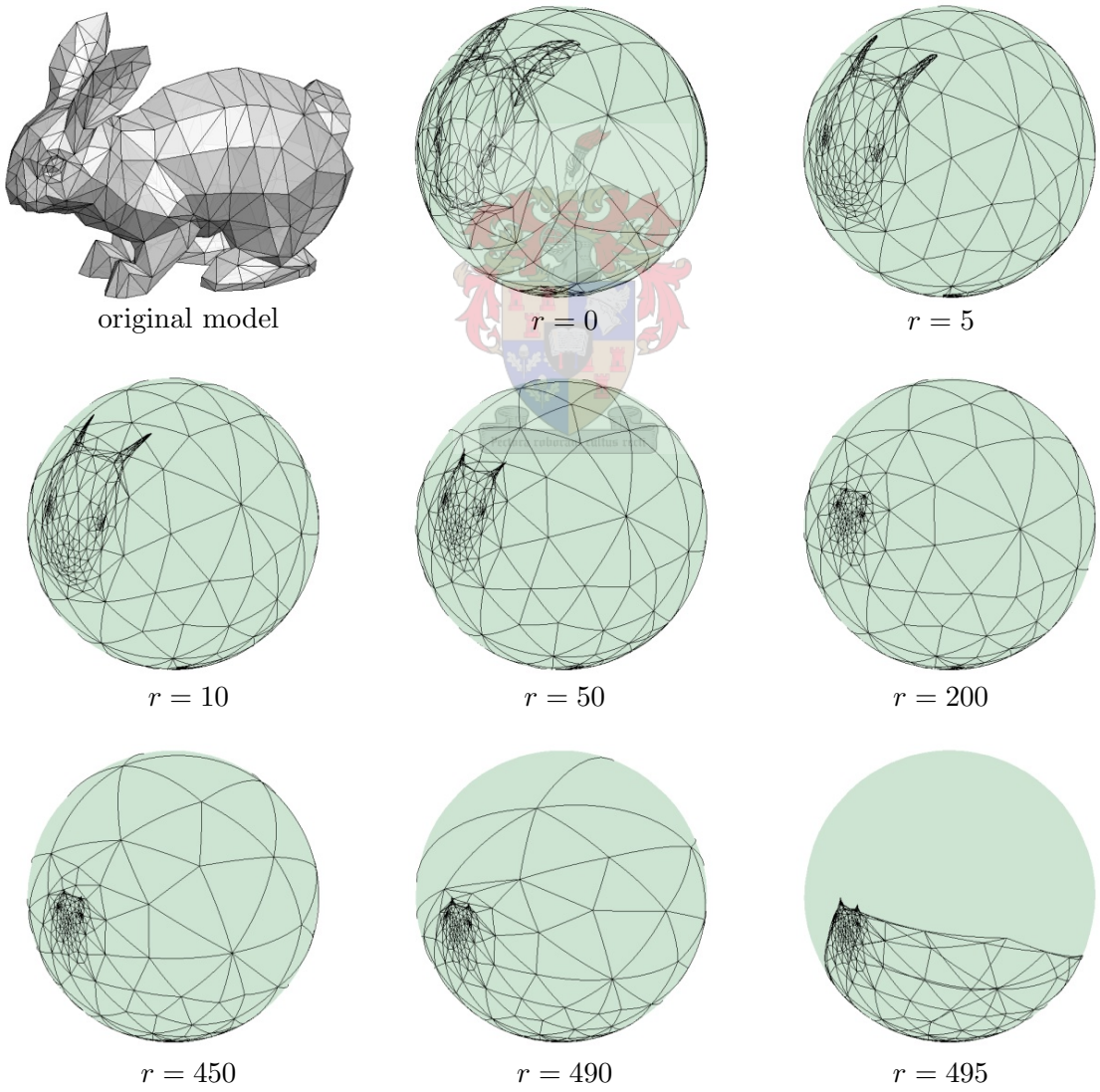


Figure 5.3

Foldovers are clearly visible in the initial embedding, at $r = 0$. Applying the iteration scheme seems to smooth out these foldovers. After about 200 iterations the embedding seems to tend towards a valid GC-embedding. Before a valid embedding is reached, however, the drawing of the graph collapses, ultimately to a single point on the surface of the sphere (which is a trivial solution of (5.10)).

It seems that the problem with this method is that faces in the embedding become relatively large in some areas, and small in other areas. These small faces tend to "pull" all the vertices of the graph towards them, resulting in a collapse of the graph.

Chapter 7 shows a few more examples, some where the method of iterative relaxation succeeds, and some where it fails. Issues such as execution time are also considered.

The next section deals with an adaptation of the method of iterative relaxation. Weights are updated at every iteration, whereby it is hoped that the collapse of the graph can be avoided.

5.3 Alexa's method

The method described in this section was established by Alexa [1]. The main difference between this method and the method discussed in the previous section is that the weights are updated at every step in the iteration. Heuristic arguments determine the specific updating.

Alexa [1] gives the following motivation for his method. Consider a spring embedding, where every edge simulates a spring. The idea is to minimise a potential defined by

$$W = \sum_{(i,j)\in E(G_M)} ||\mathbf{v}_i - \mathbf{v}_j||^2, \tag{5.16}$$

where $\mathbf{v}_i$ denotes the coordinates of the vertex i. Squaring the edge lengths ensures that longer edges are penalised, and the solution strives towards a reasonably uniform distribution over the surface of the sphere.

The following iteration scheme was proposed by Alexa [1]. An initial embedding may be chosen according to (5.13). The scheme is then defined as

$$\mathbf{v}_i^{(r+1)} = \frac{\mathbf{v}_i^{(r)} - \mathbf{q}_i^{(r)}}{||\mathbf{v}_i^{(r)} - \mathbf{q}_i^{(r)}||}, \quad i \in V(G_M), \quad r = 1, 2, \ldots, \tag{5.17}$$

with

$$\mathbf{q}_i^{(r)} = \frac{c_i^{(r)}}{\deg(i)} \sum_{j \in N(i)} \left(\mathbf{v}_i^{(r)} - \mathbf{v}_j^{(r)} \right) ||\mathbf{v}_i^{(r)} - \mathbf{v}_j^{(r)}||, \quad i \in V(G_M), \quad r = 1, 2, \ldots, \tag{5.18}$$

where $c_i^{(r)} \in \mathbb{R}$ for all $i \in V(G_M)$ and $r \in \{1, 2, \ldots\}$. The choice of $c_i^{(r)}$ will be discussed presently.

Note that multiplying $(\mathbf{v}_i^{(r)} - \mathbf{v}_j^{(r)})$ by its length results in a quadratic weight for each edge, as is implied by (5.16). Longer edges are thereby shortened in subsequent steps. Also, the point $\mathbf{v}_i^{(r)} - \mathbf{q}_i^{(r)}$ is not necessarily on the surface of the sphere, and must therefore be normalised to obtain $\mathbf{v}_i^{(r)}$.

According to Alexa [1], choosing $c_i^{(r)} = 1$, $i \in V(G_M)$ and $r \in \{1, 2, \ldots\}$, yields a robust, but not very efficient, relaxation process. For short edges, the quadratic length weights cause vertices to move very slowly from one iteration to the next. The move lengths can be increased by choosing $c_i^{(r)}$ as

$$c_i^{(r)} = \left[\max_{j \in N(i)} \left\{ ||\mathbf{v}_i^{(r)} - \mathbf{v}_j^{(r)}|| \right\} \right]^{-1}, \quad i \in V(G_M), \ r = 1, 2, \ldots. \tag{5.19}$$

If all adjacent edges are short, the corresponding $c_i^{(r)}$ become large. Alexa argues that this would ensure that short edges do not slow down the iteration process.

The following proposition relates Alexa's method to the method of iterative relaxation from the previous section.

Proposition 5.2 : *Choosing $c_i^{(r)} = 1$, $i = 1, 2, \ldots, N$, and omitting the factor $||\mathbf{v}_i^{(r)} - \mathbf{v}_j^{(r)}||$ in Alexa's method result in a special case of the method of iterative relaxation from section 5.2.*

Proof

The method of iterative relaxation, as given by (5.15), may be written as

$$\mathbf{u}_i^{(r)} = \sum_{j=1}^{N} w_{ij} \mathbf{v}_j^{(r)}, \quad i = 1, 2, \ldots, N, \tag{5.20}$$

for a fixed $r \in \{1, 2, \ldots\}$. By choosing the weights in (5.20) as Tutte weights from section 4.4.1, equation (5.20) becomes

$$\mathbf{u}_i^{(r)} = \frac{1}{\deg(i)} \sum_{j \in N(i)} \mathbf{v}_j^{(r)}, \quad i = 1, 2, \ldots, N. \tag{5.21}$$

Choosing $c_i^{(r)} = 1$, $i = 1, 2, \ldots, N$, and omitting the factor $||\mathbf{v}_i^{(r)} - \mathbf{v}_j^{(r)}||$ in (5.18), yield

$$\begin{aligned} \mathbf{q}_i^{(r)} &= \frac{1}{\deg(i)} \sum_{j \in N(i)} \left(\mathbf{v}_i^{(r)} - \mathbf{v}_j^{(r)} \right) \\ &= \mathbf{v}_i^{(r)} - \frac{1}{\deg(i)} \sum_{j \in N(i)} \mathbf{v}_j^{(r)}, \quad i = 1, 2, \ldots, N, \end{aligned}$$

by virtue of (2.3). Substituting (5.21) into this equation then gives

$$\mathbf{v}_i^{(r)} - \mathbf{q}_i^{(r)} = \mathbf{u}_i^{(r)}, \quad i = 1, 2, \ldots, N. \tag{5.22}$$

Normalising both sides of this equation gives the desired result. ■

Results from applying Alexa's method on the rabbit model are shown in Figure 5.4, after certain numbers of iterations.

A valid GC-embedding is obtained after 104 iterations. Continuing with the iteration process after this does not seem to improve the embedding significantly.

It should be stressed that the arguments in this section are all based on heuristics, and cannot guarantee convergence of the iteration process, or the validity of solutions thus obtained. However, Alexa's method does seem to be more stable than the method of iterative relaxation.

Chapter 7 provides some interesting results obtained by applying Alexa's method to various models.

In the next chapter a new method for parameterising the surface of a given mesh is discussed in detail. It will be shown that this method guarantees a valid spherical parameterisation for any mesh $M \in \mathcal{M}$.

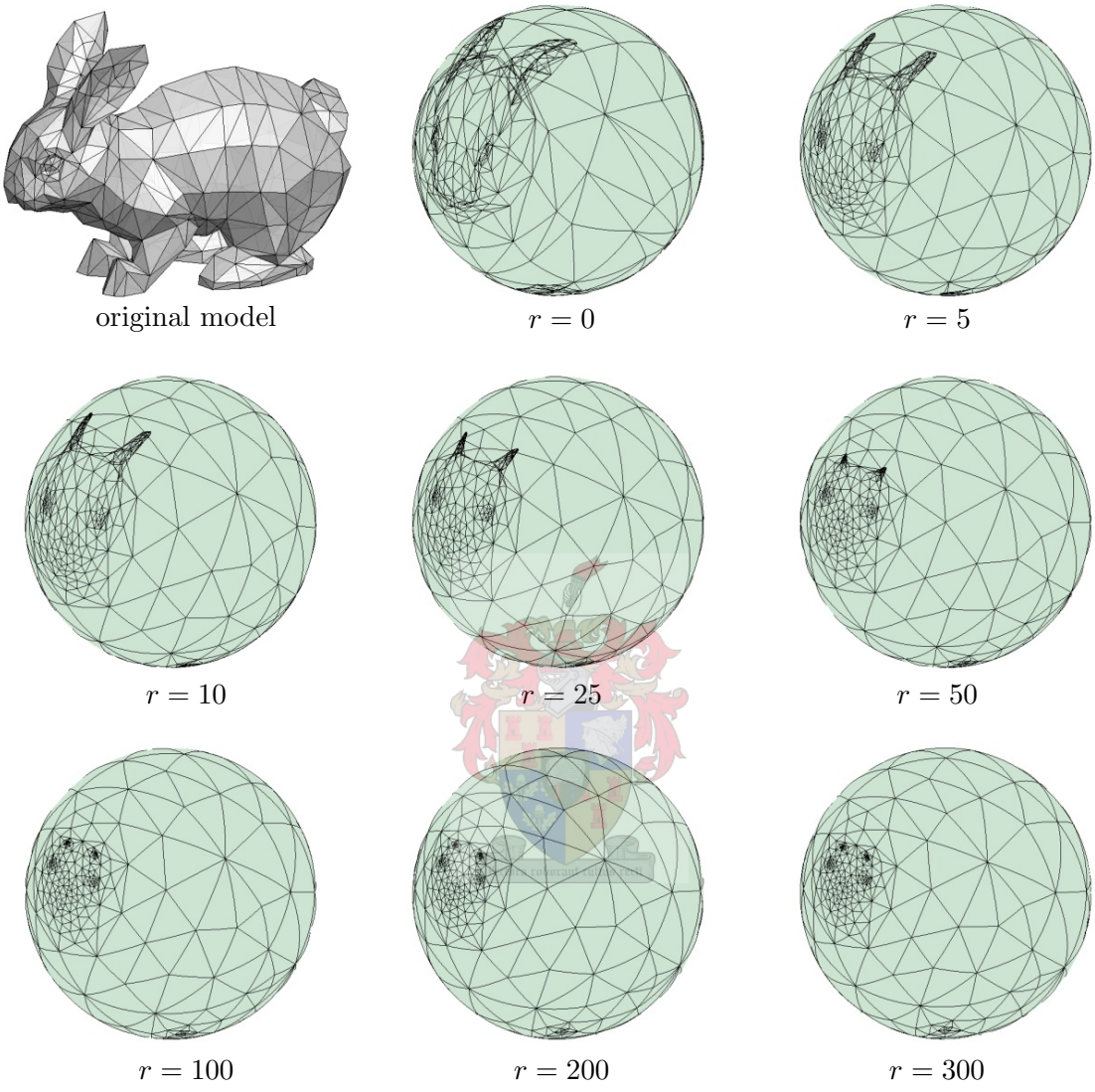
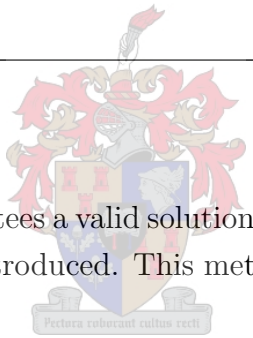


Figure 5.4

CHAPTER 6

The θ–ϕ method



In this chapter a method that guarantees a valid solution to the problem of parameterisation for any mesh from the class $\mathcal{M}$ is introduced. This method is the main novel contribution of this thesis.

The concept behind the method is to alter the given mesh in $\mathcal{M}$ to a mesh in $\mathcal{M}_B$, and then to use a 2D planar embedding algorithm to embed the underlying graph in the 2D plane. This 2D embedding is then folded onto the sphere, thereby yielding a spherical drawing of the underlying graph of the original mesh. We will call this method the θ–ϕ *method*, for reasons that will soon become clear.

This method was originally inspired by a method of Brechbühler, Kübler and Gerig [3]. Their method was designed for voxel-based objects (a 3D model built up from unit cubes), and they used heat diffusion to motivate their arguments.

We adapt their method to polygonal meshes, show that the 2D planar embedding algorithm from Chapter 4 may be incorporated, and prove that the spherical drawing thus obtained leads to a solution for the problem of parameterisation.

The following section gives a brief overview of the steps followed in the θ–ϕ method. Detailed discussion of these steps follow in subsequent sections.

6.1 Overview of the θ–ϕ method

This section provides an overview of the θ–ϕ method. In subsequent sections, the method will be explained in full detail. Figure 6.1 shows the basic strategy.

We start with a given mesh $M \in \mathcal{M}$. Two vertices of M are chosen, and a path through G_M connecting these vertices is found. See Figure 6.1(a).

Next, the mesh is cut open along this path, yielding a mesh M' similar to the one shown in Figure 6.1(b). This mesh M' belongs to the class $\mathcal{M}_B$ (see Definition 2.3).

The generalised 2D Tutte embedding algorithm from Chapter 4 is then used to embed $G_{M'}$ in the 2D plane, as shown in Figure 6.1(c).

This embedding is then folded onto the surface of the sphere, as shown in Figure 6.1(d). This yields a spherical embedding of the graph G_M, as shown in Figure 6.1(e).

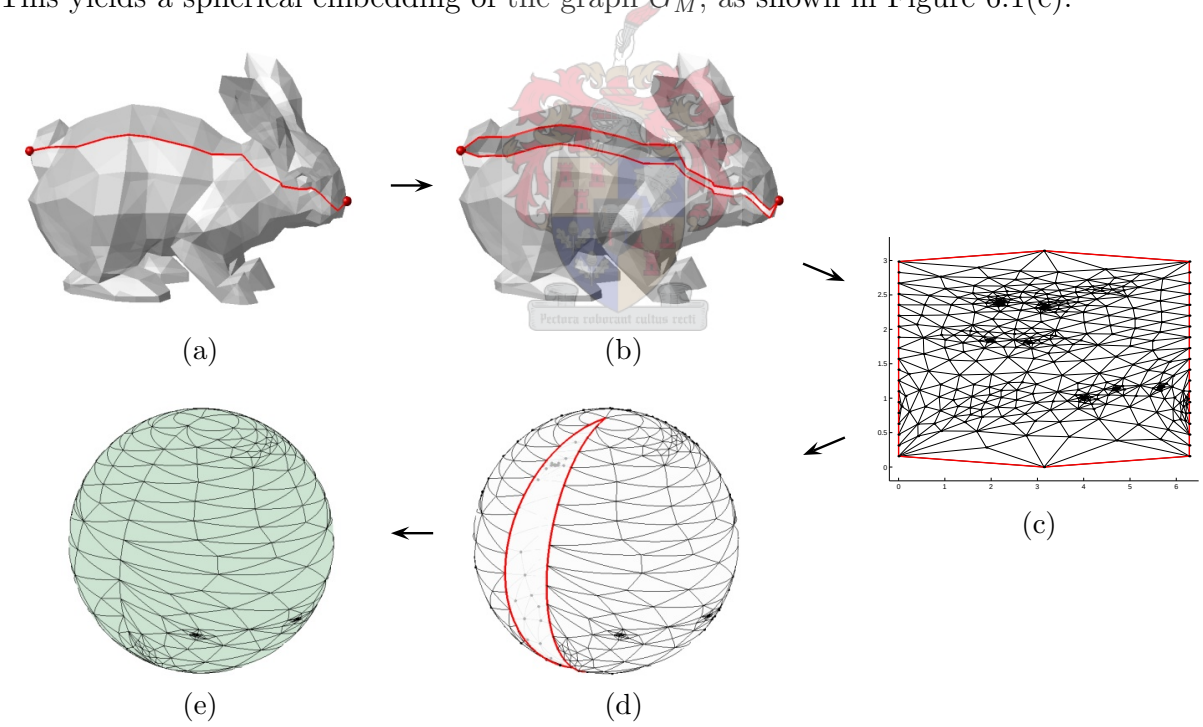


Figure 6.1

The first step of the method is to select a path between two specific vertices, along which the mesh is cut open. Section 6.2 discusses a few techniques for the selection of two such vertices, and also how a path between them may be chosen.

The second step is to cut the mesh open along the selected path. Section 6.3 discusses the issues that need to be considered in that step.

Section 6.4 involves the embedding of the open mesh in the 2D plane, and section 6.5 explains how this 2D embedding is folded onto the sphere, to yield a spherical embedding.

In section 6.6 it is shown that this spherical embedding solves the problem of parameterisation (as defined in Chapter 3).

6.2 Selecting poles and a cut path

As explained in the previous section, the θ–ϕ method involves cutting the given mesh M open, along a specific path in G_M. This section deals with selecting that path.

The path along which the mesh is cut open is referred to as the *cut path*, and denoted by P. In the final spherical embedding of G_M the two endpoints of P map to the north and south pole of the unit sphere. We therefore call these two vertices the *north pole* and *south pole* of G_M, and denote their labels by n and s, where $n, s \in V(G_M)$.

The vertices n and s may be chosen arbitrarily, but different choices of these vertices lead to different spherical embeddings of G_M. In order to avoid an uneven distribution of points in the embedding, it is advisable to choose the two vertices as far apart as possible. A number of different methods for selecting the poles follows. Selecting a path between these poles is discussed in section 6.2.4.

6.2.1 The physical distance method

In order to select two vertices in the graph G_M that are farthest apart, some measure of distance between a pair of vertices may be defined and calculated for every pair. A pair with maximum distance may then be selected as the two poles of the graph.

The first measure of distance is called *physical distance*, and is denoted by $d_p(i, j)$, $i, j \in V(G_M)$. It is defined as the physical distance between vertices i and j in the original mesh M. Therefore,

$$d_p(i, j) = ||\mathbf{x}_i - \mathbf{x}_j||, \quad i, j \in V(G_M), \tag{6.1}$$

where $\mathbf{x}_i \in \mathbb{R}^3$ denotes the coordinates of node i in M. The two poles of G_M are then defined to be two distinct vertices n and s in $V(G_M)$, satisfying

$$d_p(n, s) = \max\{d_p(i, j), \quad i, j \in V(G_M)\}. \tag{6.2}$$

The value of $d_p(i,j)$ is calculated for every possible (i,j)–pair, and the two poles are then chosen to be a pair satisfying (6.2). This method will be called the *physical distance method*.

The problem with choosing poles based on this scheme is that it may result in selecting poles that are not really "anti-podal" in the underlying graph G_M. Consider for example a "C"-shaped mesh, similar to the one depicted in Figure 6.2. Two poles satisfying (6.2) are shown in (a). Although this may not be such a bad choice, it is evident that choosing poles as those shown in (b), may result in a more uniform embedding of the graph G_M.

(a) physical distance method (b) a better choice?

Figure 6.2

It is therefore suggested that in some cases, the distance between vertices should rather be defined as the distance through the underlying graph, and not as the physical distance.

6.2.2 The graph distance method

The second measure of distance is called *graph distance*, and is denoted by $d(i,j)$. Recall from section 2.1 that the distance between two vertices i and j in a graph is defined to be the length of a shortest possible path between i and j.

According to equation (2.5), the length of a path is defined as the sum of the weights of every edge on that path. It is thus necessary to assign weights to every edge. We suggest using either constant weights,

$$\rho_{ij} = 1, \quad (i,j) \in E(G_M), \tag{6.3}$$

or edge-length weights,

$$\rho_{ij} = ||\mathbf{x}_i - \mathbf{x}_j||, \quad (i,j) \in E(G_M). \tag{6.4}$$

In this scheme, the two poles of G_M are defined to be two vertices n and s in $V(G_M)$,

satisfying

$$d(n, s) = \max\{d(i, j), \quad i, j \in V(G_M)\}. \tag{6.5}$$

Appendix A gives an overview of an algorithm, called Dijkstra's algorithm, that can be used to obtain shortest paths between vertices in a graph. The distances between every pair of vertices in G_M are calculated. The two poles are then selected to be a pair satisfying (6.5). This method will be called the *graph distance method*.

For small meshes this method may be sufficient, but the execution time for calculating distances increases dramatically as the number of vertices in the graph increase. The following scheme gives a faster alternative that generally does not produce a solution to (6.5), but does yield fairly distant poles.

6.2.3 The fast-graph method

The following pole selection scheme is much faster than the one discussed in the previous paragraph. It is based on the method suggested by Brechbühler et. al. [3].

The method begins with the selection of a fixed vertex, say $i \in V(G_M)$. This vertex is chosen arbitrarily. The north pole of G_M is then chosen to be the vertex n satisfying

$$d(i, n) = \max\{d(i, j), \quad j \in V(G_M)\}, \tag{6.6}$$

i.e. a vertex farthest from i in G_M. Next, the south pole of G_M is chosen to be a vertex s satisfying

$$d(n, s) = \max\{d(n, j), \quad j \in V(G_M)\}, \tag{6.7}$$

i.e. a vertex farthest from n in the graph.

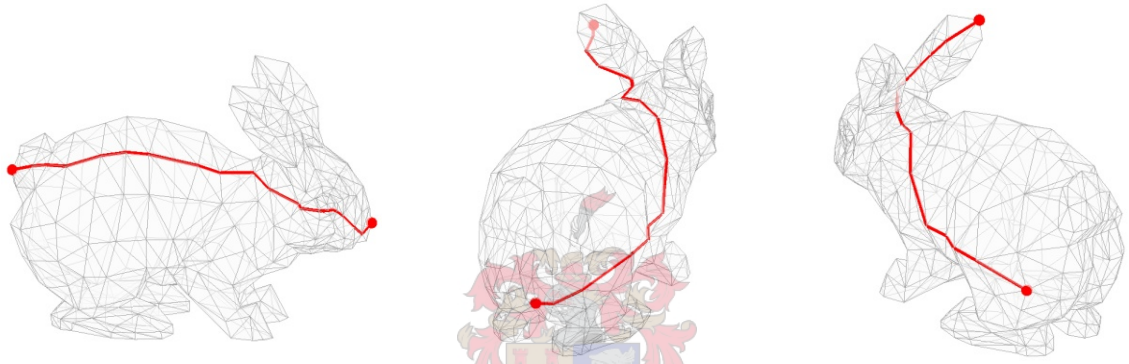
Thus, it is only necessary to find distances from two vertices to every other vertex, instead of distances between all possible pairs of vertices. This results in a much faster algorithm. We call this method the *fast-graph method*.

The method avoids selecting a pole near the middle of a long thin tubular mesh, but it generally does not produce a solution to (6.5) for any graph. In selecting poles, the idea is to avoid choosing two vertices close to one another, and this method returns vertices fairly far apart. Because of the great improvement in execution time, it is suggested that this method is chosen above the previous one.

6.2.4 Selecting the cut path

After the two poles have been selected using any of the three methods above, the cut path P is chosen to be any shortest path between vertices n and s. Dijkstra's algorithm (see Appendix A) may be used for this purpose.

Figure 6.3 shows the rabbit model with the chosen cut paths, using different pole selection methods. In each case, the poles are indicated by dots. From these examples, it is evident that the different methods produce poles that are fairly far apart.



(a) physical distance (b) graph distance (c) fast-graph

Figure 6.3

Let $|P|$ denote the number of vertices in P. We denote every vertex in P, except the two poles, by p_i, $i = 1, 2, \ldots, |P| - 2$, such that $P = \{n, p_1, p_2, \ldots, p_k, s\}$, with $k = |P| - 2$. For the remainder of this chapter, we assume that the cut path is chosen such that $k \geq 2$, i.e. there are at least 4 vertices in the cut path.

A symmetry plane is clearly visible in the rabbit model. Since the θ–ϕ method cuts the mesh open on a path, and then later reattaches it, we might hope to select a cut path close to this symmetry plane (possibly for aesthetic reasons). The next method attempts to select poles and a cut path based on this argument.

6.2.5 The symmetry method

The following method for selecting poles and a cut path attempts to find symmetry planes in a given mesh. A path in or close to such a plane is then chosen to be the cut path.

First, a weighted confidence ellipsoid is fitted to points on the surface of the mesh M. Pairs of the three principal axes of this ellipsoid span three different planes in $\mathbb{R}^3$. These

planes are tested for symmetry, and the best candidate is then chosen to be the potential symmetry plane. Two poles and a path between them are then selected from a set of vertices that are on or close to this plane. This method of finding poles and a cut path will be called the *symmetry method*.

Consider a given set of N coordinates in $\mathbb{R}^3$, denoted by $\mathbf{v}_i = [x,\ y,\ z]^T$. Let $\mathbf{m}$ denote the mean of these coordinates, i.e.

$$\mathbf{m} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{v}_i. \tag{6.8}$$

The coordinates are translated, and packed into the matrix V as follows,

$$V = \left[\mathbf{v}_1 - \mathbf{m},\quad \mathbf{v}_2 - \mathbf{m},\quad \ldots,\quad \mathbf{v}_N - \mathbf{m}\right]^T. \tag{6.9}$$

The *covariance matrix* of V, denoted by C_V, is calculated as

$$C_V = V^T V. \tag{6.10}$$

Because C_V is symmetric, there exist three orthogonal eigenvectors $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{s}_3$. It is well-known that these eigenvectors give the directions of the three principal axes of the *confidence ellipsoid* of V, centred at $\mathbf{m}$.

Suppose however that the coordinates of V have associated weights, i.e. some points are more important than others. Suppose therefore that every vector $\mathbf{v}_i$ has an associated weight $w_i \in \mathbb{R}$.

The value of $\mathbf{m}$ is redefined to be the weighted average of the points, i.e.

$$\mathbf{m} = \frac{\sum_{i=1}^{N} w_i \mathbf{x}_i}{\sum_{i=1}^{N} w_i}. \tag{6.11}$$

This vector $\mathbf{m}$ is used in the construction (6.9) of V.

The diagonal matrix W is defined to be the $N \times N$ matrix with diagonal entries $\{W\}_{ii} = w_i$, $i = 1, 2, \ldots, N$. The *weighted confidence ellipsoid* is then the ellipsoid centred at $\mathbf{m}$, with principal axes in the directions of the eigenvectors of the matrix C, where

$$C = V^T W V. \tag{6.12}$$

Consider a mesh M. The symmetry method operates as follows. A weighted confidence ellipsoid is fitted to points on the surface of M, and the three planes spanned by pairs of the principal axes of this ellipsoid are tested for potential symmetry.

Because the surface of M is made up of triangles with different areas, we opt to choose the centroids of all the faces as points, each weighed with the area of the corresponding face.

Let the centroid of a face $i \in \{1, 2, \ldots, |F(M)|\}$ be denoted by $\mathbf{y}_i$, then

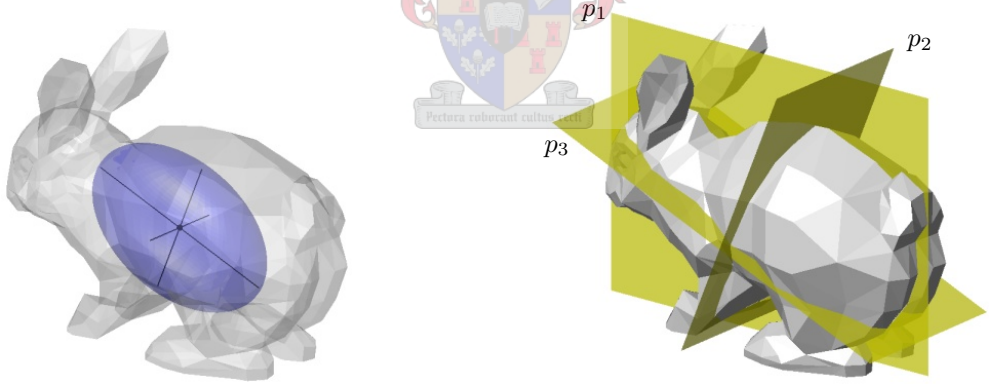
$$\mathbf{y}_i = \tfrac{1}{3}(\mathbf{x}_{f_{i,1}} + \mathbf{x}_{f_{i,2}} + \mathbf{x}_{f_{i,3}}), \quad i = 1, 2, \ldots, |F(M)|, \tag{6.13}$$

where $\mathbf{x}_i$ gives the coordinates of node i in M. Let w_i denote the area of the face f_i, i.e.

$$w_i = \tfrac{1}{2}||(\mathbf{x}_{f_{i,2}} - \mathbf{x}_{f_{i,1}}) \times (\mathbf{x}_{f_{i,3}} - \mathbf{x}_{f_{i,1}})||, \quad i = 1, 2, \ldots, |F(M)|. \tag{6.14}$$

The weighted confidence ellipsoid is calculated for the coordinates $\mathbf{y}_i$, weighed by w_i, $i = 1, 2, \ldots, |F(M)|$. This ensures that larger faces have a greater impact on the orientation of the ellipsoid than smaller faces. Figure 6.4(a) shows the rabbit model (slightly transparent), with a scaling of the corresponding weighted confidence ellipsoid.

Let $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{s}_3$ denote the normalised eigenvectors of C. Let p_1, p_2 and p_3 be the three planes spanned respectively by $\mathbf{s}_2$ and $\mathbf{s}_3$, $\mathbf{s}_1$ and $\mathbf{s}_3$, and $\mathbf{s}_1$ and $\mathbf{s}_2$. Figure 6.4(b) shows the rabbit model with the planes p_1, p_2 and p_3.



(a) confidence ellipsoid (b) potential symmetry planes

Figure 6.4

Suppose that C is diagonalised as

$$C = Q\Lambda Q^T, \tag{6.15}$$

where Q is an orthogonal matrix with columns $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{s}_3$, and Λ is the diagonal matrix with diagonal entries equal to the corresponding eigenvalues of C.

Next, the planes p_1, p_2 and p_3 are tested for potential symmetry. Consider for instance plane p_1. We wish to calculate the signed distance between p_1 and every point $\mathbf{y}_i - \mathbf{m}$.

By signed distance we mean the positive distance if the point lies on one side of the plane, and the negative of the distance if it lies on the other side. These distances should, ideally, sum to zero if p_1 is an exact symmetry plane.

Since Q is an orthogonal matrix, the vector $\mathbf{s}_1$ is a normalised vector perpendicular to the plane p_1. The projection of $\mathbf{y}_i - \mathbf{m}$ on the line spanned by $\mathbf{s}_1$ is given by

$$\mathbf{p}_i = \mathbf{s}_1\mathbf{s}_1^T(\mathbf{y}_i - \mathbf{m}). \tag{6.16}$$

Note that $\mathbf{p}_i$ is a vector in the direction of $\mathbf{s}_1$. Therefore the signed distance from $\mathbf{y}_i - \mathbf{m}$ to plane p_1 is given by

$$d_{p_1}(\mathbf{y}_i) = \mathbf{s}_1^T(\mathbf{y}_i - \mathbf{m}). \tag{6.17}$$

Calculating

$$D_{p_1} = \sum_{i=1}^{|F(M)|} d_{p_1}(\mathbf{y}_i), \tag{6.18}$$

yields the sum of the signed distances from all the points $\mathbf{y}_i$ to the plane p_1. The values of D_{p_2} and D_{p_3} are calculated similarly. For an exact symmetry plane p the value of D_p would be 0. We define the *best potential symmetry plane* of the mesh M to be the plane p_k such that

$$|D_{p_k}| = \min\{|D_{p_1}|, |D_{p_2}|, |D_{p_3}|\}, \tag{6.19}$$

with $k \in \{1, 2, 3\}$. For the rabbit model we get $D_{p_1} = 0.65$, $D_{p_2} = -228.7$ and $D_{p_3} = -699.2$. Clearly, plane p_1 is the best potential symmetry plane for this example.

Next, the vertices of M close to p_k are selected. The set of vertices $R \subset V(G_M)$, defined by

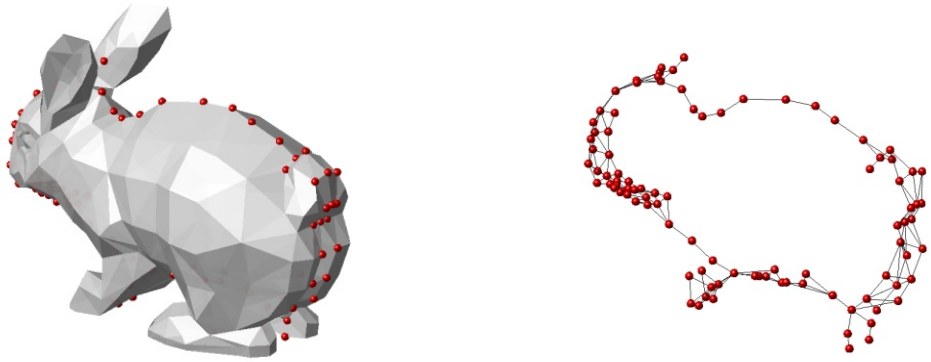
$$R = \{i \in V(G_M) : d_{p_k}(\mathbf{x}_i) < \varepsilon\} \tag{6.20}$$

with $\mathbf{x}_i$ the coordinates of node i in the mesh, and ε some predefined tolerance, contains all the vertices close to the plane p_k. Figure 6.5(a) shows the rabbit model with the vertices in R. For this example, $\varepsilon = 0.3$ was chosen.

We select two poles, and a path between them, from the following subgraph of G_M. Let H be the graph with vertex set $V(H) = R$, and edge set

$$E(H) = \{(i, j) \in E(G_M) : i, j \in R\}. \tag{6.21}$$

The graph H is therefore a graph with vertices close to the symmetry plane, and edges from G_M connecting them. Figure 6.5(b) shows the graph H for the rabbit model.



(a) vertices close to p_1 (b) the graph H

Figure 6.5

For some models the graph H may be disconnected. If this is the case, we choose the connected component of H with the most vertices.

The fast-graph method from section 6.2.3 is implemented next, to select two poles n and s in H. A shortest path between n and s, through the graph H, is then selected to be the cut path. Figure 6.6 shows the cut path of the rabbit model, selected according to the symmetry method.

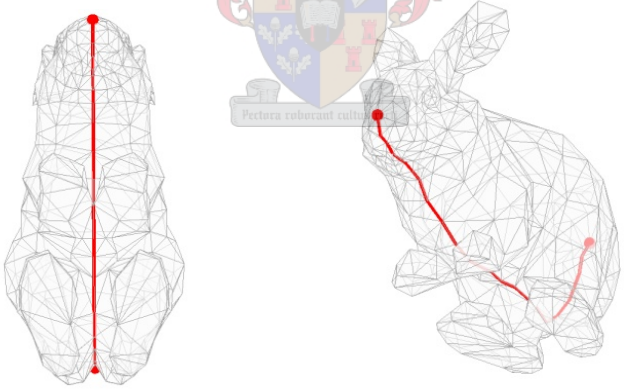


Figure 6.6

It should be noted that the symmetry method may identify a non-symmetry plane as a potential symmetry plane, since the only criterion for a plane to be classified as a potential symmetry plane is that the signed distances from every vertex to the plane should cancel out and ideally sum to 0. Of course, this criterion does not hold only if the plane is a symmetry plane.

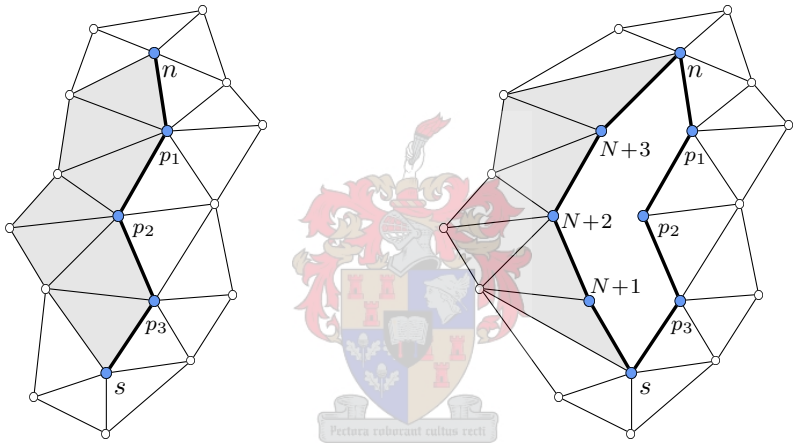
Also, if there does not seem to be a symmetry plane in the model, that is, if none of the values $|D_{p_1}|$, $|D_{p_2}|$ or $|D_{p_3}|$ are sufficiently small, it is advisable to use another method for selecting the poles.

6.3 Cutting the mesh open

The next step in the θ–ϕ method is to cut the mesh M open along the cut path, resulting in a mesh $M' \in \mathcal{M}_B$.

In this section, we derive expressions for the coordinate set $X(M')$, and the face set $F(M')$. These two sets uniquely define the mesh M', as well as the underlying graph $G_{M'}$. Let the number of vertices in M be denoted by N.

Figure 6.7 shows how M is cut open along the path P. The figure shows part of a graph G_M in (a), with P in bold, and part of the resulting graph $G_{M'}$ in (b).



(a) part of G_M (b) part of $G_{M'}$

Figure 6.7

For every vertex p_i, $i = 1, 2, \ldots, k$, a new vertex is created. The labels of the new vertices, $N{+}1, N{+}2, \ldots, N{+}k$, are ordered such that $B = (n, p_1, p_2, \ldots, p_k, s, N{+}1, N{+}2, \ldots, N{+}k)$ forms the boundary cycle in M'. See for example Figure 6.7(b). That is, vertex p_i is duplicated with the label $N + k - i + 1$, $i = 1, 2, \ldots, k$.

We position the new vertices at exactly the same points as the vertices they duplicate. The coordinate set $X(M')$ is therefore defined as

$$X(M') = X(M) \cup \{\mathbf{x}_{N+1}, \mathbf{x}_{N+2}, \ldots, \mathbf{x}_{N+k}\}, \tag{6.22}$$

where we choose $\mathbf{x}_{N+i} = \mathbf{x}_{p_{k-i+1}}$, $i = 1, 2, \ldots, k$.

In order to construct the face set $F(M')$, note that some of the faces in the original mesh M need to be changed. See for example the grey faces in Figure 6.7. The faces of M that need to be changed are shown in (a). The changed faces of M' are shown in (b).

We denote the set of faces in the original mesh M that need to be changed by F_C, where $F_C \subset F(M)$. Also, let F_U denote the set of faces in M that remain unchanged in M'. Therefore

$$F_U = F(M) \setminus F_C. \tag{6.23}$$

The faces in F_C need to be identified. Algorithm 6.1 may be used for this purpose. The algorithm returns the set F_C.

The algorithm uses a function faces$[G_M, (i,j)]$, where G_M is the underlying graph of a mesh $M \in \mathcal{M}$, and $(i,j) \in E(G_M)$. The function returns the set $\{f_1, f_2\}$, such that f_1 and f_2 are the two faces in M that share the edge (i,j). The function is therefore defined as

$$\text{faces}[G_M, (i,j)] = \{f \in F(M) : i,j \in f\}. \tag{6.24}$$

Note also that for the purpose of the algorithm, each face is seen as a set containing all its vertices.

Algorithm 6.1 :
INPUT: A graph G_M with cut path $P = \{n, p_1, p_2, \ldots, p_k, s\}$
OUTPUT: The set F_C of faces that need to be changed

1. *Let* $p_{k+1} \leftarrow s$, $F_C \leftarrow \{\}$, *and* $i \leftarrow 1$
2. $\{f_1, f_2\} \leftarrow \text{faces}[G_M, (n, p_1)]$
3. $f \leftarrow f_1$
4. $q_i \leftarrow f \setminus \{n, p_1\}$
5. **for** $j = 1$ *to* k **do**
   while $p_{j+1} \notin q_i$ **do**
   $F_C \leftarrow F_C \cup f$
   $g \leftarrow \text{faces}[G_M, (q_i, p_j)]$
   $f \leftarrow g \setminus f$
   $i \leftarrow i + 1$
   $q_i \leftarrow f \setminus \{q_{i-1}, p_j\}$
   end
   $i \leftarrow i - 1$
   end
6. $F_C \leftarrow F_C \cup f$
7. **return** F_C

Figure 6.8 illustrates this algorithm for the example from Figure 6.7. At each step, the edge (q_i, p_j) is shown in bold, and the face f in grey. The algorithm runs through every vertex in P, and finds the labels of all the neighbouring vertices, until the next vertex in P is reached. This ensures that only the faces to the one side of P are put in F_C.

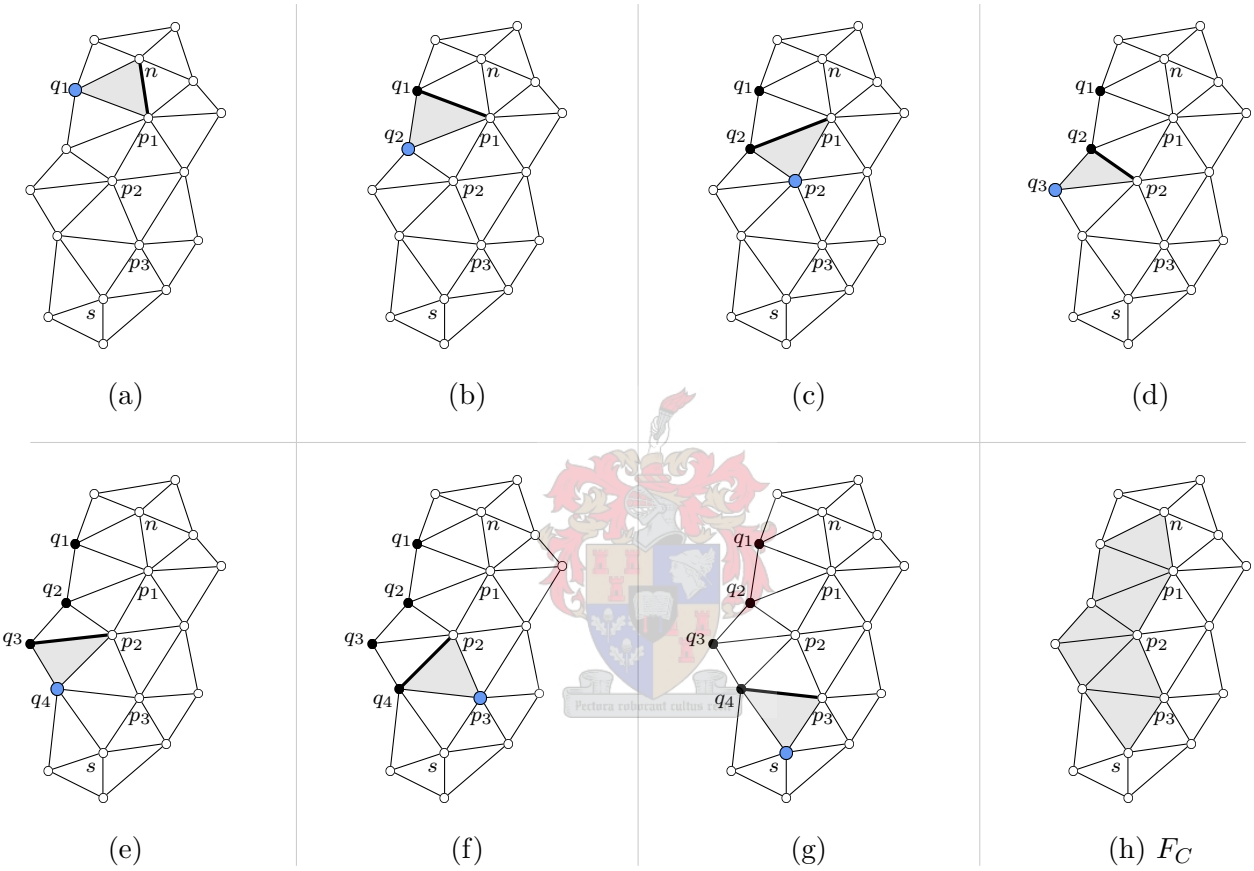


Figure 6.8

Note that in step 3 of Algorithm 6.1, the face f_1 is chosen at random. If f_2 had been chosen, the vertices on the other side of P would have been selected.

Let $\overline{P}$ denote the set of vertices on the cut line, excluding the two poles. Thus

$$\overline{P} = P \setminus \{n, s\}. \tag{6.25}$$

Then every face $f \in F_C$ has either 1, 2 or 3 vertices belonging to the set $\overline{P}$. The index of each of these vertices is updated to the index of the new vertex that duplicates it.

Thus, for every face $f \in F_C$, the following is performed. Every vertex in the set $f \cap \overline{P}$ has a label of the form p_i, with $i \in \{1, 2, \ldots, k\}$. These labels are changed to $N + k - i + 1$, for every $p_i \in f \cap \overline{P}$. Let F'_C denote the set of faces resulting from this updating.

The face set of M' is then given by

$$F(M') = F'_C \cup F_U, \tag{6.26}$$

with F_U defined by (6.23). Equations (6.22) and (6.26) yield the mesh $M' \in \mathcal{M}_B$. The underlying graph $G_{M'}$ is then constructed according to Definition 2.1.

The next section deals with embedding $G_{M'}$ in the 2D plane.

6.4 Embedding the open mesh in the 2D plane

This section discusses a specific 2D planar embedding for the graph $G_{M'}$, as used in the θ–ϕ method. We implement the generalised 2D Tutte embedding (from section 4.3) for this purpose.

The first step in embedding $G_{M'}$ in the 2D plane is to assign every boundary vertex of M' coordinates, such that these coordinates lie on the boundary of a convex polygon in $\mathbb{R}^2$. We choose the coordinates of the boundary vertices specifically to aid in simplifying the final step of the θ–ϕ method, which is to fold the 2D embedding onto the unit sphere.

Consider the spherical coordinate system, as illustrated in Figure 6.9. Any point in $\mathbb{R}^3$ is described by a triplet (r, θ, ϕ), where r is the length of the position vector of the point, and θ and ϕ are the two angles as shown.

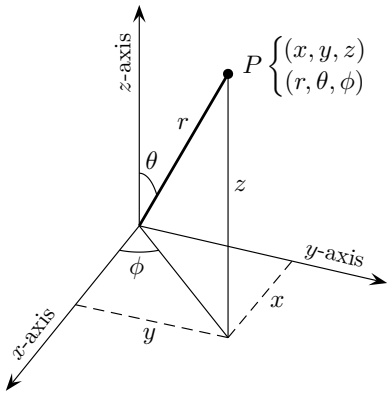


Figure 6.9

It can be shown that spherical coordinates are related to cartesian coordinates by

$$x = r\sin\theta\cos\phi, \quad y = r\sin\theta\sin\phi, \quad z = r\cos\theta, \tag{6.27}$$

and inversely,

$$r = \sqrt{x^2+y^2+z^2}, \quad \theta = \arccos(z/\sqrt{x^2+y^2+z^2}), \quad \phi = \arctan(y/x). \tag{6.28}$$

Note that (2.15), the definition of S_0, can be written in spherical coordinates as

$$S_0 = \{(1, \theta, \phi) : \theta \in [0, \pi],\ \phi \in [0, 2\pi)\}. \tag{6.29}$$

Therefore, any point on the surface of the unit sphere is uniquely defined by the two parameters θ and ϕ, where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi)$.

We call the rectangle in $\mathbb{R}^2$ defined by the cartesian coordinates $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi)$ the θ–ϕ *rectangle*. We aim to embed the graph $G_{M'}$ in the θ–ϕ rectangle, because any point in that rectangle, excluding the lines $\theta = 0$ and $\theta = \pi$, maps uniquely to a point in S_0 by virtue of (6.27), with $r = 1$.

Recall from the previous section that the boundary cycle of M' is of the form

$$B = (n, p_1, p_2, \ldots, p_k, s, N+1, N+2, \ldots, N+k). \tag{6.30}$$

The (θ, ϕ)–coordinates of the boundary vertices are chosen as follows,

$$\begin{aligned} &\mathbf{u}_n = (0, \pi), \quad \mathbf{u}_s = (\pi, \pi), \\ &\mathbf{u}_{p_i} = (ih, 0), \quad \mathbf{u}_{N+i} = ((k+1-i)h, 2\pi), \quad i = 1, 2, \ldots, k, \end{aligned} \tag{6.31}$$

with $h = \pi/(k+1)$. Every vertex $i \in B$ is positioned at coordinates $\mathbf{u}_i$ in the θ–ϕ rectangle, thereby yielding a boundary for the 2D embedding. This boundary is clearly convex, as can be seen from the example in Figure 6.10. In this example, $k = 4$.

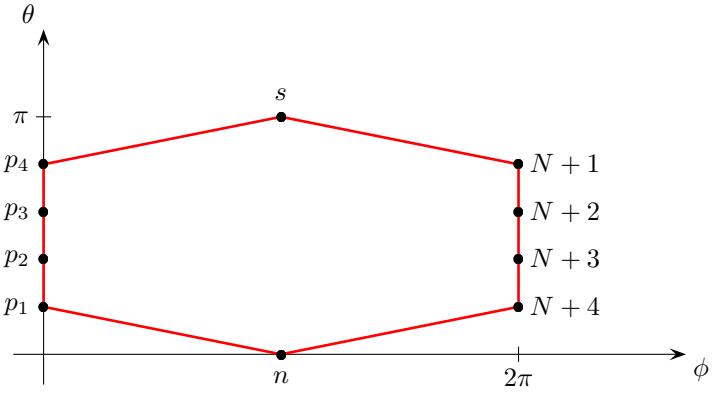


Figure 6.10

With the boundary vertices fixed, every interior vertex $i \in V(G_{M'}) \setminus B$ is positioned at coordinates $\mathbf{u}_i$, which are solutions to the linear system

$$\mathbf{u}_i = \sum_{j=1}^{N+k} w_{ij}\mathbf{u}_j, \quad i \in V(G_{M'}) \setminus B. \tag{6.32}$$

The weights w_{ij}, $i, j \in V(G_{M'})$ are chosen to satisfy

$$w_{ij} > 0, \quad (i,j) \in E(G_{M'}), \quad \text{and} \quad w_{ij} = 0, \quad (i,j) \notin E(G_{M'}), \tag{6.33}$$

and also

$$\sum_{j=1}^{N+k} w_{ij} = 1, \quad i \in V(G_{M'}) \setminus B. \tag{6.34}$$

This yields a 2D planar embedding for the graph $G_{M'}$. No edges intersect in this embedding, by virtue of Theorems 4.5 and 4.7, and because the boundary is convex, and (6.33) and (6.34) hold.

Figure 6.11 shows a 2D embedding in the θ–ϕ rectangle of the rabbit model. The cut line was chosen to be the one shown in Figure 6.6, and Tutte weights (see section 4.4.1) were used in (6.32).

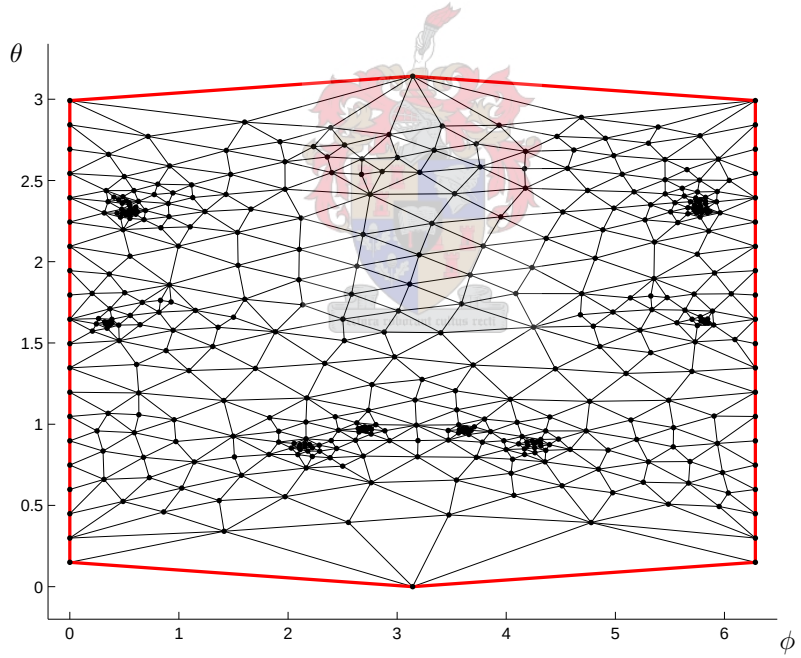


Figure 6.11

The fact that the graph $G_{M'}$ is embedded in the θ–ϕ rectangle is the reason why we call this method the θ–ϕ method.

The next section deals with folding this 2D embedding onto the unit sphere to give a spherical drawing of the graph $G_{M'}$.

6.5 Folding the 2D embedding onto the sphere

In this section the last step of the θ–ϕ method is discussed. The 2D embedding of the graph $G_{M'}$ is folded onto the unit sphere. In section 6.6 it will be shown that this leads to a solution for the problem of parameterisation.

Since the boundary of the 2D embedding was chosen to embed the graph $G_{M'}$ in the θ–ϕ rectangle, every vertex $i \in G_{M'}$ has values (θ_i, ϕ_i). It follows from (6.27) that the mapping $m : [0, \pi] \times [0, 2\pi) \mapsto S_0$, defined by

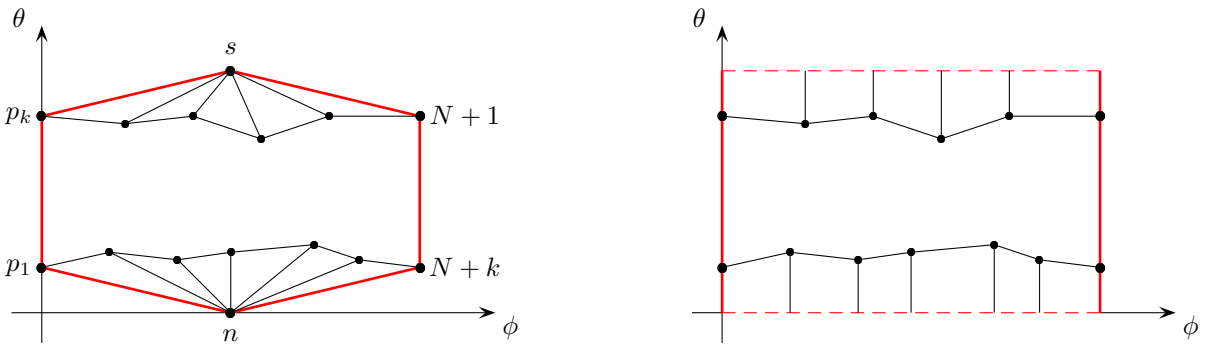
$$m(\mathbf{u}) = \begin{bmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{bmatrix}, \quad \mathbf{u} = (\theta, \phi) \in [0, \pi] \times [0, 2\pi), \tag{6.35}$$

maps any point in the θ–ϕ rectangle, excluding the lines $\theta = 0$ and $\theta = \pi$, to its corresponding point on the surface of the unit sphere.

Note from Figure 6.10 that there are four triangular regions in the θ–ϕ rectangle that are not covered by the embedding of $G_{M'}$. Before the mapping m is applied to the embedding of $G_{M'}$, the embedding is transformed to cover the entire θ–ϕ rectangle.

6.5.1 Transforming the 2D embedding to cover the entire θ–ϕ rectangle

This section explains how the 2D embedding of $G_{M'}$ may be transformed, to cover the entire θ–ϕ rectangle. Figure 6.12 illustrates the aim of the chosen transformation.



(a) embedding (b) transformed embedding

Figure 6.12

The original embedding of $G_{M'}$ is shown in Figure 6.12(a), and the transformed embedding

in (b). Only the boundary edges and edges connecting the poles and neighbours of the poles are shown.

Since the mapping m maps any point on the line $\theta = 0$ to the point $(0, 0, 1) \in S_0$, the position of the north pole, $\mathbf{u}_n = (0, \pi)$, is transformed to the line $\theta = 0$. The edges connecting the vertex n are changed in the following manner.

Consider an edge (n, i) connecting n with some vertex $i \in N(n)$. The coordinates of n in the θ–ϕ plane is $\mathbf{u}_n = (0, \pi)$, and that of i is $\mathbf{u}_i = (\theta_i, \phi_i)$. Instead of drawing this edge as the straight line segment connecting $\mathbf{u}_i$ and $\mathbf{u}_n$, we draw it as the vertical line connecting the points $\mathbf{u}_i$ and $(0, \phi_i)$.

A similar transformation is applied to the edges connecting the south pole s. Instead of drawing a straight line segment between $\mathbf{u}_j = (\theta_j, \phi_j)$ and $\mathbf{u}_s = (\pi, \pi)$, we draw it as the line connecting $\mathbf{u}_j$ and (π, ϕ_j), for some $j \in N(s)$.

Consider the path $P_n = (p_1, v_1, v_2, \ldots, v_\ell, N + k)$, where $\ell = \deg(n) - 2$, and $v_i \in N(n)$, $i = 1, 2, \ldots, \ell$. This is the path of all vertices adjacent to the north pole. Clearly, the transformed embedding contains no edge intersections if the sequence $\{\phi_i\}$ increases strictly monotonically, $i \in P_n$. Figure 6.13(a) shows an example where this is not the case. Figure 6.13(b) shows the resulting transformed embedding.

(a) embedding (b) transformed embedding

Figure 6.13

A similar restriction holds for the path P_s, which is the path of all vertices adjacent to the south pole. In practice it rarely happens that the ϕ–values do not increase strictly monotonically, with the result that the transformed embedding contains no edge intersections. If it does happen, however, we change the initial boundary of the 2D embedding as follows.

Suppose the ϕ–values on the path $P_n = (p_1, v_1, v_2, \ldots, v_\ell, N + k)$ do not increase monotonically. We define the graph H as

$$H = G_{M'} - n, \tag{6.36}$$

i.e. $G_{M'}$, with vertex n removed (together with all its adjacent edges). The boundary cycle

B' of H is then given by

$$B' = (p_1, p_2, \ldots, p_k, s, N+1, N+2, \ldots, N+k, v_\ell, v_{\ell-1}, \ldots, v_1). \tag{6.37}$$

For the generalised 2D Tutte embedding algorithm the (θ, ϕ)–coordinates of these boundary vertices are chosen as

$$\begin{aligned} &\mathbf{u}_s = (\pi, \pi), \\ &\mathbf{u}_{p_i} = (ih_1, 0), \quad \mathbf{u}_{N+i} = ((k+1-i)h_1, 2\pi), \quad i = 1, 2, \ldots, k, \\ &\mathbf{u}_{v_j} = (h_1, jh_2), \quad j = 1, 2, \ldots, \ell, \end{aligned} \tag{6.38}$$

with $h_1 = \pi/(k+1)$ and $h_2 = 2\pi/(\ell+1)$. Figure 6.14 shows an example of such a boundary, where $k = 4$ and $\ell = 5$.

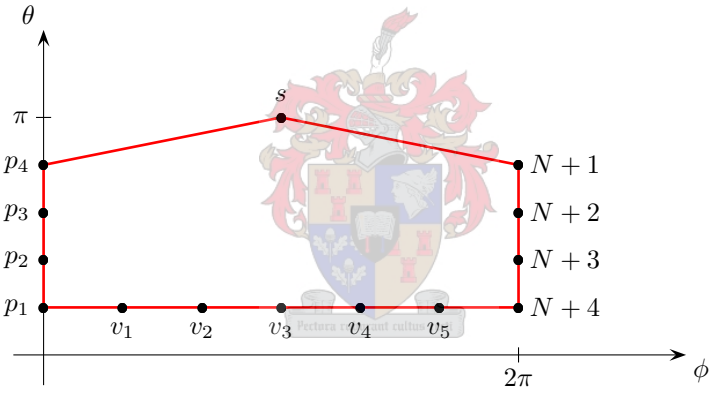


Figure 6.14

The boundary is clearly convex. The generalised 2D Tutte embedding algorithm is implemented on the graph H with this boundary, yielding a 2D embedding of H in the θ–ϕ rectangle. The vertex n is positioned at $(0, \pi)$, and connected to its neighbours in $G_{M'}$. This yields a 2D embedding of $G_{M'}$.

Note that the ϕ–values of the neighbours of n do increase strictly monotonically in this embedding. It is therefore possible to transform the embedding as illustrated in Figure 6.12, without resulting in edge intersections.

A similar procedure is applied to the embedding of $G_{M'}$ in the case where the ϕ–values of the neighbours of s do not increase strictly monotonically.

Thus, no edge intersections occur in the transformed embedding, and the embedding covers the entire θ–ϕ rectangle.

6.5.2 Mapping the transformed embedding to S_0

Next, the transformed embedding is mapped to the surface of the sphere. The mapping m, given by (6.35), is applied to every vertex $\mathbf{u}_i$, $i = 1, 2, \ldots, N + k$, as follows,

$$\mathbf{v}_i = m(\mathbf{u}_i), \quad i = 1, 2, \ldots, N + k. \tag{6.39}$$

This gives coordinates $\mathbf{v}_i \in S_0$ for every vertex $i \in V(G_{M'})$.

The edges in the (transformed) embedding are also mapped with m, as follows. An arbitrary point $\mathbf{x}$ on the straight line segment joining the points $\mathbf{u}_1 = (\theta_1, \phi_1)$ and $\mathbf{u}_2 = (\theta_2, \phi_2)$ in the θ–ϕ rectangle may be represented as

$$\mathbf{x} = \lambda\mathbf{u}_1 + (1 - \lambda)\mathbf{u}_2, \quad \lambda \in [0, 1]. \tag{6.40}$$

Allowing λ to vary between 0 and 1 yields every point on that line segment. Therefore, mapping the line segment to the sphere is a matter of calculating $m(\mathbf{x})$ for every $\lambda \in [0, 1]$, where $\mathbf{x}$ is given by (6.40).

Mapping the vertices and edges of the 2D embedding to the surface of the sphere yields a spherical drawing of the graph $G_{M'}$. We refer to this spherical drawing as a *TP-embedding* of $G_{M'}$. Figure 6.15 shows the TP-embedding of the graph $G_{M'}$ from Figure 6.11.

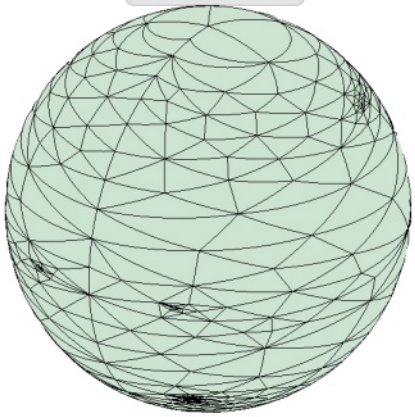


Figure 6.15

Note the following. Since the positions of the boundary vertices were chosen according to (6.31), it follows that $\mathbf{u}_{N+k-i+1} = (2\pi, ih)$, and hence

$$m(\mathbf{u}_{p_i}) = m(\mathbf{u}_{N+k-i+1}), \quad i = 1, 2, \ldots, k. \tag{6.41}$$

Any edge connecting a vertex $N+i$ with a neighbour is therefore also a curve on S_0 between that neighbour and the vertex that was originally duplicated, namely p_{k-i+1}.

Also, both the lines $\phi = 0$ and $\phi = 2\pi$ map to the same line on the sphere. Thus, under the mapping m the graph $G_{M'}$ is closed on exactly the same line as it was originally cut open.

In the following section we show that a TP-embedding of $G_{M'}$ leads to a solution for the problem of parameterising the surface of the original mesh M.

6.6 Solving the problem of parameterisation

Recall that the problem of parameterisation for a given mesh $M \in \mathcal{M}$ is to find a continuous bijective mapping (or homeomorphism) between S_M and S_0. In section 3.3 a homeomorphism between S_M and S_0 was constructed with the aid of a valid GC-embedding of the graph G_M.

In this section we proceed to construct a homeomorphism h between S_M and S_0 by using a TP-embedding of the graph $G_{M'}$ obtained with the θ–ϕ method. This homeomorphism would then be a solution to the problem of parameterisation.

Consider a mesh $M \in \mathcal{M}$, with a given TP-embedding of the graph $G_{M'}$. It is therefore assumed that M has been cut open (as explained in section 6.3), to produce a mesh $M' \in \mathcal{M}_B$, and that the graph $G_{M'}$ was embedded in the θ–ϕ rectangle (as explained in section 6.4).

Some notation is needed. The set of points in the θ–ϕ rectangle covered by the embedding of $G_{M'}$ is denoted by T'. We define $\overline{T'}$ as

$$\overline{T'} = T' \setminus \{(0, \pi), (\pi, \pi)\}. \tag{6.42}$$

We will write T to denote the entire θ–ϕ rectangle, that is, all the points in $[0, \pi] \times [0, 2\pi)$, and define $\overline{T}$ as

$$\overline{T} = T \setminus \{\mathbf{u} = (\theta, \phi) \in T : \theta \in \{0, \pi\}\}. \tag{6.43}$$

We write $\overline{S_{M'}}$ to denote the set $S_{M'} \setminus \{\mathbf{x}_n, \mathbf{x}_s\}$, which is the set of all points on the surface of M', excluding the two poles.

We will also write $\overline{S_0}$ to denote the set $S_0 \setminus \{(0, 0, -1), (0, 0, 1)\}$.

Figure 6.16 gives an illustration of how the construction of h proceeds. In section 6.6.1 a mapping h_1 between $\overline{S_{M'}}$ and $\overline{T'}$ is established. Then, in section 6.6.2, a mapping h_2 between $\overline{T'}$ and $\overline{T}$ is established. Section 6.6.3 establishes a mapping h_3 between $\overline{T}$ and $\overline{S_0}$.

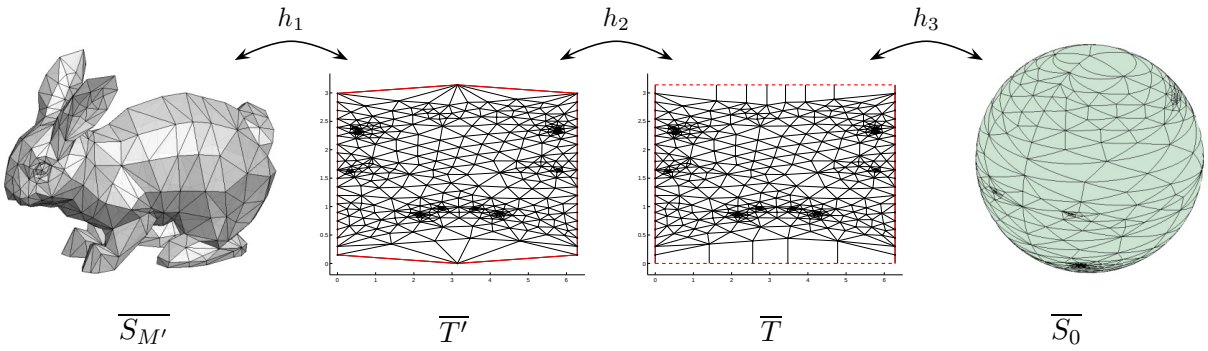


Figure 6.16

Combining the mappings h_1, h_2 and h_3 leads to a mapping between S_M and S_0. This is discussed in section 6.6.4.

6.6.1 Mapping between $\overline{S_{M'}}$ and $\overline{T'}$

In this section, a continuous mapping h_1 between the sets $\overline{S_{M'}}$ and $\overline{T'}$ is established. Recall that $\overline{S_{M'}}$ denotes the set of points on the surface of M', excluding the two poles, and $\overline{T'}$ denotes the set of points in the θ–ϕ rectangle covered by the embedding of G'_M and excluding the poles.

An arbitrary point in $\overline{S_{M'}}$ lies inside or on the boundary of a triangular face $f \in F(M)$. Since the 2D embedding of G_M partitions $\overline{T'}$ into a number of non-overlapping triangles, f corresponds to a unique triangle t in the θ–ϕ plane.

We use the barycentric mapping, discussed in section 3.2, to obtain a homeomorphism between f and t. Consideration of all the faces of M' and all the triangles in the θ–ϕ plane yield a homeomorphic mapping h_1 between $\overline{S_{M'}}$ and $\overline{T'}$.

6.6.2 Mapping between $\overline{T'}$ and $\overline{T}$

In this section, a continuous mapping h_2 between the sets $\overline{T'}$ and $\overline{T}$ is established. Recall that $\overline{T'}$ denotes the set of points in the θ–ϕ rectangle, covered by the embedding of G'_M and

excluding the poles, and $\overline{T}$ denotes the set of points in the entire θ–ϕ rectangle, excluding the lines $\theta = 0$ and $\theta = \pi$.

We use the transformation from section 6.5.1 to transform the embedding of $G_{M'}$ to cover the entire θ–ϕ rectangle. We assume, by changing the embedding as discussed in section 6.5.1 if necessary, that there are no edge intersections in this transformed embedding.

Consider a point $\mathbf{u} \in \overline{T'}$. We will establish a mapping h_2 that maps $\mathbf{u}$ to a point $\mathbf{u}' \in \overline{T}$. If $\mathbf{u}$ does not lie in one of the triangles affected by the transformation, then h_2 maps $\mathbf{u}$ to exactly the same position. That is, we let $\mathbf{u}' = \mathbf{u}$.

Suppose that the point $\mathbf{u}$ lies in one of the triangles affected by the transformation. Suppose the vertices of this triangle are positioned at $\mathbf{u}_1 = (\theta_1, \phi_1)$, $\mathbf{u}_2 = (\theta_2, \phi_2)$ and $\mathbf{u}_3 = (\theta_3, \phi_3)$, such that $\mathbf{u}_3$ is equal to either $\mathbf{u}_n$ or $\mathbf{u}_s$. See for example Figure 6.17(a). We denote the set of points inside this triangle by Q_3.

The point $\mathbf{u}$ in the triangle described above must be mapped continuously and uniquely to a point in the trapezium, with vertices positioned at $\mathbf{u}_1$, $\mathbf{u}_2$, (θ_3, ϕ_1) and (θ_3, ϕ_2). See for example Figure 6.17(b). We denote the set of points in this trapezium by Q_4.

(a) triangle Q_3 (b) trapezium Q_4

Figure 6.17

Let Q_3' be the set of points given by

$$Q_3' = Q_3 \setminus \{\mathbf{u}_3\}, \tag{6.44}$$

and Q_4' the set of points given by

$$Q_4' = Q_4 \setminus \{(\theta, \phi) \in Q_4 : \theta \neq \theta_3\}. \tag{6.45}$$

We aim to establish a continuous one-to-one mapping between Q_3' and Q_4'. Consider the given point $\mathbf{u} = (\theta_u, \phi_u) \in Q_3'$. It is possible to write

$$\mathbf{u} = w_1\mathbf{u}_1 + w_2\mathbf{u}_2 + w_3\mathbf{u}_3, \tag{6.46}$$

where $w_1, w_2 \in [0,1]$, $w_3 \in [0,1)$, and $w_1 + w_2 + w_3 = 1$. The weights w_1, w_2 and w_3 are obtained by solving the non-singular linear system (6.46).

The point $\mathbf{u}$ is also a linear combination of the points $\mathbf{v}$ and $\mathbf{u}_3$, as shown in Figure 6.17(a), where $\mathbf{v} = (\theta_v, \phi_v)$ is some combination of $\mathbf{u}_1$ and $\mathbf{u}_2$. Thus

$$\mathbf{v} = \lambda \mathbf{u}_1 + (1-\lambda)\mathbf{u}_2, \tag{6.47}$$

for some $\lambda \in [0,1]$. Hence

$$\mathbf{u} = \mu \mathbf{v} + (1-\mu)\mathbf{u}_3 = \lambda\mu \mathbf{u}_1 + (1-\lambda)\mu \mathbf{u}_2 + (1-\mu)\mathbf{u}_3, \tag{6.48}$$

for some $\mu \in (0,1]$. Comparing (6.48) with (6.46) yields $\lambda = w_1/(1-w_3)$ and $\mu = 1 - w_3$.

Since $\mathbf{u}_3$ is transformed to the line segment between (θ_3, ϕ_2) and (θ_3, ϕ_1), let $\mathbf{u}'$ be the following linear combination of $\mathbf{v}$ and $\mathbf{w}$, with $\mathbf{w} = (\theta_3, \phi_u)$,

$$\mathbf{u}' = \mu \mathbf{v} + (1-\mu)\mathbf{w}. \tag{6.49}$$

See for example Figure 6.17(b). Some manipulation yields

$$\mathbf{u}' = \begin{bmatrix} \theta_u \\ \lambda\phi_1 + (1-\lambda)\phi_2 \end{bmatrix} = \begin{bmatrix} \theta_u \\ \frac{w_1}{1-w_3}\phi_1 + \frac{w_2}{1-w_3}\phi_2 \end{bmatrix}. \tag{6.50}$$

The point $\mathbf{u}'$ is then the unique point in Q_4' corresponding to the point $\mathbf{u} \in Q_3'$. Note that (6.50) gives a formula for calculating $\mathbf{u}'$ from the weights that satisfy (6.46).

This establishes the mapping h_2.

Next, to establish the inverse mapping h_2^{-1}, consider a point $\mathbf{u}' = (\theta_{u'}, \phi_{u'}) \in \overline{T}$. If the point $\mathbf{u}'$ lies in any triangle t of the embedding of $G_{M'}$, such that n or s is not a vertex of t, then the inverse of h_2 does not affect the position of $\mathbf{u}'$, that is, we let $\mathbf{u} = \mathbf{u}'$.

Assume now that this is not the case. Then $\mathbf{u}'$ lies in one of the trapezoidal regions created by transforming the embedding of $G_{M'}$, say in the trapezium Q_4 from Figure 6.17(b). Note that $\mathbf{u}'$ is a linear combination of $\mathbf{v}$ and $\mathbf{w}$, as shown in Figure 6.17(b). Therefore

$$\mathbf{u}' = \lambda \mathbf{v} + (1-\lambda)\mathbf{w}, \tag{6.51}$$

for some $\lambda \in (0,1]$. The point $\mathbf{w}$ is defined to be equal to $(\theta_3, \phi_{u'})$, and $\mathbf{v} = (\theta_v, \phi_v)$ is some linear combination of $\mathbf{u}_1$ and $\mathbf{u}_2$, thus

$$\mathbf{v} = \mu \mathbf{u}_1 + (1-\mu)\mathbf{u}_2, \tag{6.52}$$

for some $\mu \in [0,1]$. Since $\phi_v = \phi_{u'}$, it follows that

$$\mu = \frac{\phi_{u'} - \phi_2}{\phi_1 - \phi_2}. \tag{6.53}$$

Substituting this value into (6.52) yields the point $\mathbf{v}$. By considering the θ–values in (6.51) we have

$$\theta_{u'} = \lambda[\mu\theta_1 + (1-\mu)\theta_2] + (1-\lambda)\theta_3, \tag{6.54}$$

and therefore

$$\lambda = \frac{\theta_{u'} - \theta_3}{\mu(\theta_1 - \theta_2) - \theta_2 - \theta_3}. \tag{6.55}$$

The point $\mathbf{u}$, defined as

$$\mathbf{u} = \lambda\mathbf{v} + (1-\lambda)\mathbf{u}_3, \tag{6.56}$$

is the unique point in Q_3' corresponding to the point $\mathbf{u}' \in Q_4'$. Again, note that (6.56), together with (6.55), (6.53) and (6.52), yield a formula for calculating $\mathbf{u}$, given the point $\mathbf{u}'$.

This also establishes the continuous inverse mapping h_2^{-1}.

6.6.3 Mapping between the $\overline{T}$ and $\overline{S_0}$

In this section, a mapping h_3 between the sets $\overline{T}$ and $\overline{S_0}$, is established. Recall that $\overline{T}$ denotes the set of points in the entire θ–ϕ rectangle, excluding the lines $\theta = 0$ and $\theta = \pi$, and $\overline{S_0}$ denotes the set of points on the surface of the unit sphere, excluding the points $(0,0,-1)$ and $(0,0,1)$.

Recall from section 6.5 that the mapping m, as defined in (6.35), maps any point in $\overline{T}$ to a point in $\overline{S_0}$. According to (6.28), the inverse of m is given by

$$m^{-1}(\mathbf{v}) = \begin{bmatrix} \arccos(z/\sqrt{x^2+y^2+z^2}) \\ \arctan(y/x) \end{bmatrix}, \quad \mathbf{v} = (x,y,z) \in S_0. \tag{6.57}$$

We therefore choose $h_3 = m$, and this gives a mapping between $\overline{T}$ and $\overline{S_0}$. Note that the mapping h_3 is not continuous, since the surface of the sphere is "cut open" to form the θ–ϕ plane. This is not a problem, as will be seen shortly.

6.6.4 Mapping between S_M and S_0

Finally, the mappings h_1, h_2 and h_3 are combined to establish a mapping h between S_M and S_0. We construct h as follows.

Consider a point $\mathbf{x} \in S_M$. If $\mathbf{x} = \mathbf{x}_n$, then h maps $\mathbf{x}$ to the point $(0, 0, 1) \in S_0$. If $\mathbf{x} = \mathbf{x}_s$, then h maps $\mathbf{x}$ to the point $(0, 0, -1) \in S_0$.

Suppose now that $\mathbf{x} \notin \{\mathbf{x}_n, \mathbf{x}_s\}$. The point $\mathbf{x}$ lies inside or on the boundary of a face in $F(M)$, and therefore also inside or on the boundary of a face in $F(M')$. This point maps uniquely to a point $\mathbf{u}$ in the θ–ϕ plane, under the mapping h_1 from section 6.6.1. The mapping h_2 from section 6.6.2 is used to map $\mathbf{u}$ uniquely to a point $\mathbf{u}'$ in the entire θ–ϕ plane. This point is then mapped with h_3 from section 6.6.3 to a point $\mathbf{v} \in S_0$.

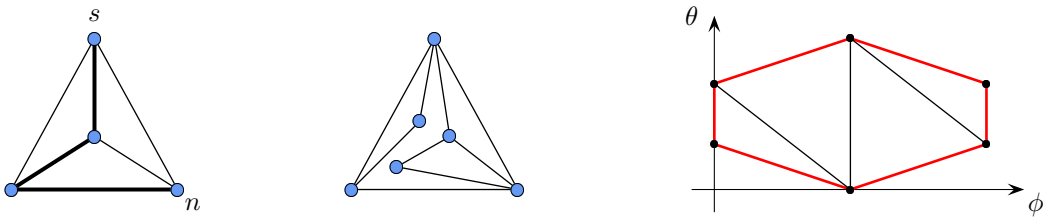
The inverse mapping, h^{-1}, is defined as follows. Consider a point $\mathbf{v} \in S_0$. If $\mathbf{v} = (0, 0, 1)$, then h^{-1} maps it to $\mathbf{x}_n$, and if $\mathbf{v} = (0, 0, -1)$, then h^{-1} maps it to $\mathbf{x}_s$.

Assume now that $\mathbf{v}$ is neither $(0, 0, 1)$ nor $(0, 0, -1)$. The point $\mathbf{v}$ is mapped with the function h_3^{-1} from section 6.6.3, to yield a point $\mathbf{u}'$ somewhere in the θ–ϕ rectangle. The mapping h_2^{-1} from section 6.6.2 is used to map $\mathbf{u}'$ to the point $\mathbf{u}$ in the region of the θ–ϕ rectangle that is covered by the embedding of $G_{M'}$. This point $\mathbf{u}$ maps uniquely to a point $\mathbf{x}$ in $S_{M'}$ under the mapping h_1^{-1} from section 6.6.1. Note that $\mathbf{x}$ is then also a point in S_M.

Since the mapping h_3 closes the 2D embedding of $G_{M'}$ on exactly the same line as the mesh M was originally cut open, it follows that h is a continuous mapping between S_M and S_0.

This mapping h can be constructed for any mesh $M \in \mathcal{M}$, as long as it is possible to cut M open to obtain $M' \in \mathcal{M}_B$, and its underlying graph $G_{M'}$. From the beginning of the discussion of the θ–ϕ method, the only assumption that was made on the original mesh M was that the cut path should contain at least 4 vertices.

Next we show that it is possible to obtain a path containing 4 vertices, through the graph G_M of any mesh $M \in \mathcal{M}$. Recall from section 2.2.7 that there does not exist a mesh in $\mathcal{M}$ with fewer than 4 vertices. Also, there is only one mesh in $\mathcal{M}$ with 4 vertices (up to isomorphism of the underlying graphs). Let M denote this mesh.



(a) the graph G_M (b) the graph $G_{M'}$ (c) 2D embedding of $G_{M'}$

Figure 6.18

Figure 6.18(a) shows the graph G_M, with a path indicated in bold. This path has four vertices, and is therefore a valid cut path. Figure 6.18(b) shows the graph $G_{M'}$ that results from cutting M open along this path. The 2D embedding of $G_{M'}$ in the θ–ϕ rectangle is shown in Figure 6.18(c). Therefore, the θ–ϕ method succeeds for this mesh.

Clearly, adding vertices to M would not result in the failure of the θ–ϕ method. Hence, the θ–ϕ method can be applied to any mesh in $\mathcal{M}$.

We have therefore proven the main result of this chapter, which is the following.

Proposition 6.2 : *The θ–ϕ method, together with the mapping h established in this section, solves the problem of parameterisation for any mesh in $\mathcal{M}$.*

This result is a great advantage of the θ–ϕ method. Recall from Chapter 5 that the method of iterative relaxation, as well as Alexa's method, cannot guarantee a valid solution to the problem of parameterisation.

Chapter 7 gives a more comprehensive comparison of these methods with the θ–ϕ method. An issue such as execution time is considered.

This concludes the description of the θ–ϕ method. In the last section of this chapter the possibility of obtaining a valid GC-embedding from the θ–ϕ method is discussed.

6.7 GC-embeddings from the θ–ϕ method

Recall from Chapter 3 that a GC-embedding of a graph G is defined as a drawing of G on the sphere, such that every vertex $i \in V(G_M)$ is positioned at a distinct point $\mathbf{v}_i \in S_0$, and that every edge is drawn as the minor arc of a great circle between its endpoints. For this GC-embedding to be valid, we require that none of these edges intersect.

It was shown in section 3.3 that the problem of parameterisation for a mesh $M \in \mathcal{M}$ is easily solved once such a GC-embedding of G_M is obtained.

The question this section addresses is whether it is possible to obtain a valid GC-embedding of G_M from the 2D embedding of $G_{M'}$, yielded by the θ–ϕ method.

Note that the spherical drawing obtained by mapping the 2D embedding of $G_{M'}$ in the θ–ϕ rectangle onto the surface of the sphere is not a GC-embedding. The edges are not drawn as minor arcs, as is visible in for example Figure 6.15. The edges are obtained by mapping

a straight line segment in the θ–ϕ rectangle with the function m defined in (6.35).

Next, we consider what happens if the edges are drawn as minor arcs, in the hope of obtaining a GC-embedding of G_M.

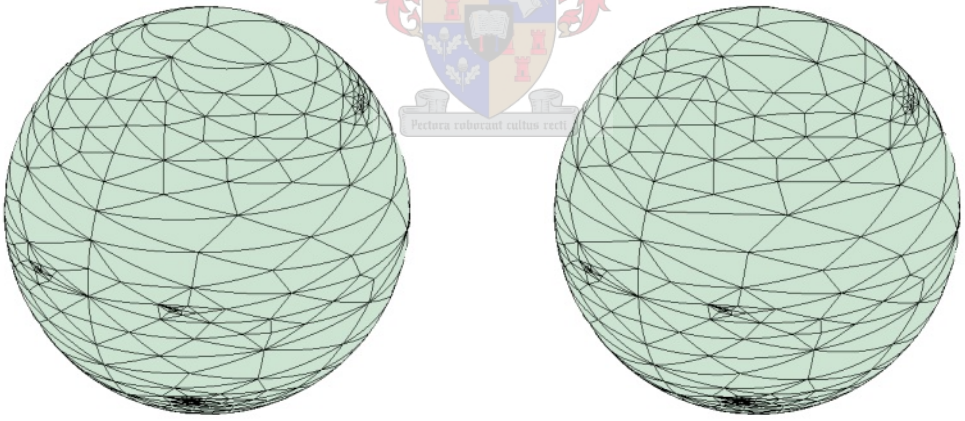
Consider for this purpose a mesh $M \in \mathcal{M}$. Suppose this mesh is cut open as prescribed by the θ–ϕ method to produce the mesh $M' \in \mathcal{M}_B$. Suppose the graph $G_{M'}$ is embedded in the θ–ϕ rectangle, such that the coordinates $\mathbf{u}_i$ give the position of vertex $i \in V(G_{M'})$.

The function m from (6.35) is used to map every vertex $i \in V(G_{M'})$ to the point $\mathbf{v}_i \in S_0$. Let every vertex $i \in V(G_M)$ be positioned at $\mathbf{v}_i$, with

$$\mathbf{v}_i = m(\mathbf{u}_i), \quad i \in V(G_{M'}). \tag{6.58}$$

The edges are simply drawn as minor arcs between endpoints. This yields a GC-drawing of the graph $G_{M'}$. The question of whether it is also a valid GC-embedding of $G_{M'}$ remains.

Figure 6.19(a) shows the TP-embedding of Figure 6.15, for comparison. Figure 6.19(b) shows a drawing of the same graph, where the edges were simply drawn as minor arcs between endpoints.



(a) edges mapped with m (b) edges as minor arcs

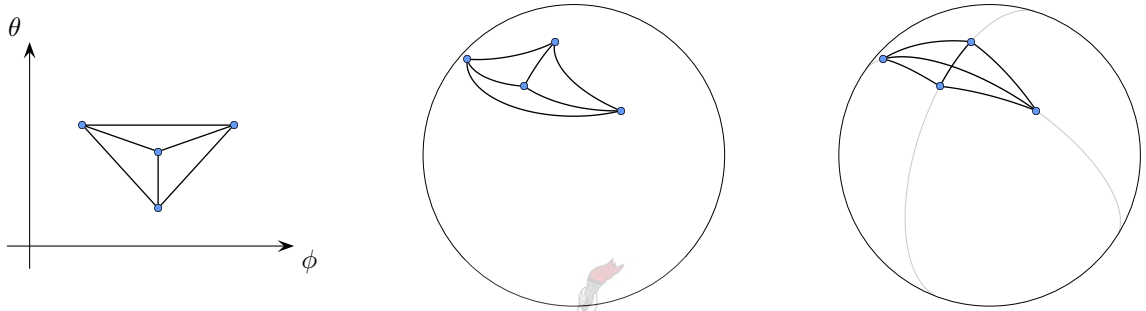
Figure 6.19

For this example, inspection shows that there are no edge intersections in the GC-drawing depicted in Figure 6.19(b). Hence, a valid GC-embedding is obtained and Proposition 3.3 implies that this embedding solves the problem of parameterisation (with a different mapping than the one established in section 6.6.4).

It is important to note, however, that in general a valid GC-embedding may not be obtained with this procedure. Indeed, the following example shows that an embedding in the θ–ϕ

rectangle where no edges intersect does not necessarily yield a GC-drawing with no edge intersections.

Figure 6.20(a) shows four vertices in the θ–ϕ plane, connected by straight line segments. Clearly, no edges intersect. Figure 6.20(b) shows this embedding mapped to the surface of the sphere. Here the lines were also mapped with the function m. In Figure 6.20(c), these curves are replaced by minor arcs of great circles. An edge intersection occurs.



(a) the θ–ϕ plane (b) edges mapped with m (c) minor arcs

Figure 6.20

The preceding example shows that the θ–ϕ method may not always yield a valid GC-embedding. However, as Figure 6.19 illustrates, there are meshes for which a valid GC-embedding may be obtained from the θ–ϕ method.

The next chapter shows some experimental results. Examples where the θ–ϕ method gives a valid GC-embedding, and some where it does not, are also mentioned.

CHAPTER 7

Experimental results

In this chapter some experimental results are given and discussed. The methods and algorithms described in this thesis were implemented in MATLAB 6.5 Release 13. Appendix B gives a short description of this implementation. All the experiments mentioned in this chapter were executed on a 2.8 GHz Intel Pentium 4 processor with 512 MB memory.

The following section shows the different test models that were used, and provides some properties of these models.

7.1 Test models

We experimented on eight different test models, all varying in size and shape. These models are shown on the following pages, each with its number of vertices, faces and edges.

The models are all freely available on the internet, in different formats. The different formats were converted to MATLAB variables. Every model listed belongs to the class $\mathcal{M}$ (see Definition 2.2). Also, the faces of every model are all orientated correctly (see section 2.4).

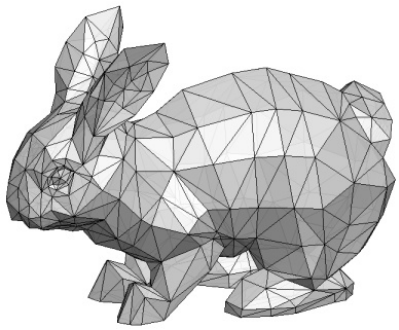
Note that the edges of the last two models are not drawn, simply because the faces are too small.

PAWN

vertices	154
faces	304
edges	456

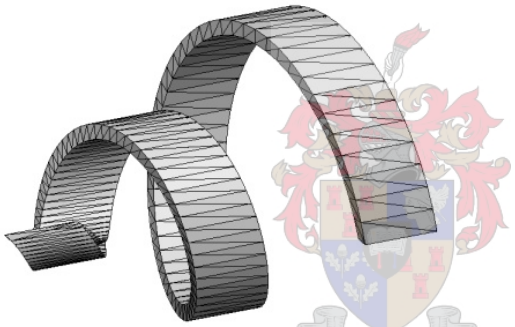
RABBIT

vertices	453
faces	902
edges	1,353



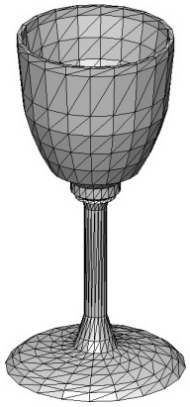
HELIX

vertices	505
faces	1,006
edges	1,509



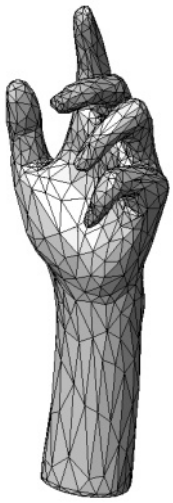
GLASS

vertices	662
faces	1,320
edges	1,980



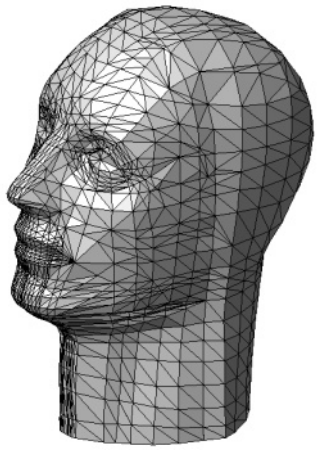
HAND

vertices	1,002
faces	2,000
edges	3,000



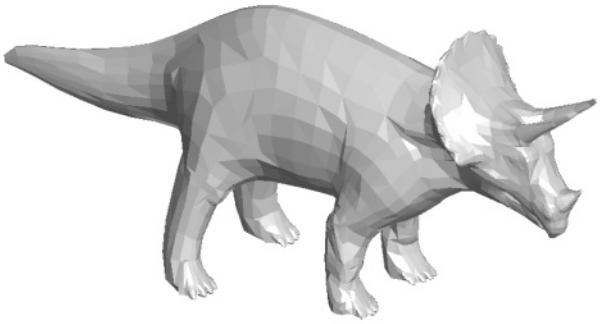
HEAD

vertices	1,490
faces	2,976
edges	4,464



TRICERATOPS

vertices	2,832
faces	5,660
edges	8,490



FEMALE

vertices	4,039
faces	8,074
edges	12,111

Model	Vertices	Faces	Edges
pawn	154	304	456
rabbit	453	902	1,353
helix	505	1,006	1,509
glass	662	1,320	1,980
hand	1,002	2,000	3,000
head	1,490	2,976	4,464
triceratops	2,832	5,660	8,490
female	4,039	8,074	12,111

The different parameterisation methods discussed in this thesis were tested on the models above. Results of these tests follow. The issue of execution time is considered in section 7.6.

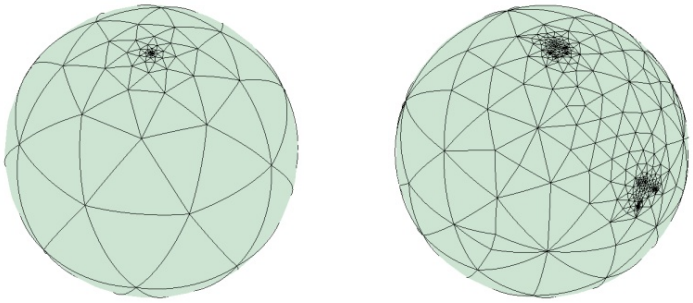
7.2 Gotsman's method

Recall from section 5.1.3 that Gotsman's method involves solving the non-linear system (5.10) with the MATLAB function `fsolve`.

The method as we implemented it was only able to solve the system for PAWN and RABBIT.

The other models have too many vertices, and MATLAB quickly runs out of memory.

Figure 7.1 shows the results of applying Gotsman's method to PAWN and RABBIT. In each case, Tutte weights were used (see section 4.4.1).



PAWN RABBIT

Figure 7.1 — Gotsman's method

7.3 Iterative relaxation

Results from applying the method of iterative relaxation (from section 5.2) are given in this section. The method is defined by (5.15).

Recall that an initial embedding on the sphere is needed. This embedding may be obtained by (5.13), with $\mathbf{m}$ some point inside the mesh. Choosing $\mathbf{m}$ as the arithmetic mean of the nodes yields a valid interior point for every model except HELIX. For this model an interior point was selected manually. Figure 7.2 shows the initial embedding of every model.

The method of iterative relaxation yields valid GC-embeddings for five of the eight models. For RABBIT, HELIX and FEMALE, however, the graph collapses (ultimately to a single point on the surface of the sphere) before a valid embedding is reached. Figure 5.3 illustrates this collapsing behaviour for RABBIT.

Figure 7.3 shows the valid GC-embeddings of the five models for which the iterative relaxation method does succeed. The orientation test in section 3.4.1 was implemented to terminate the process when a valid embedding is reached. Tutte weights (section 4.4.1) were used.

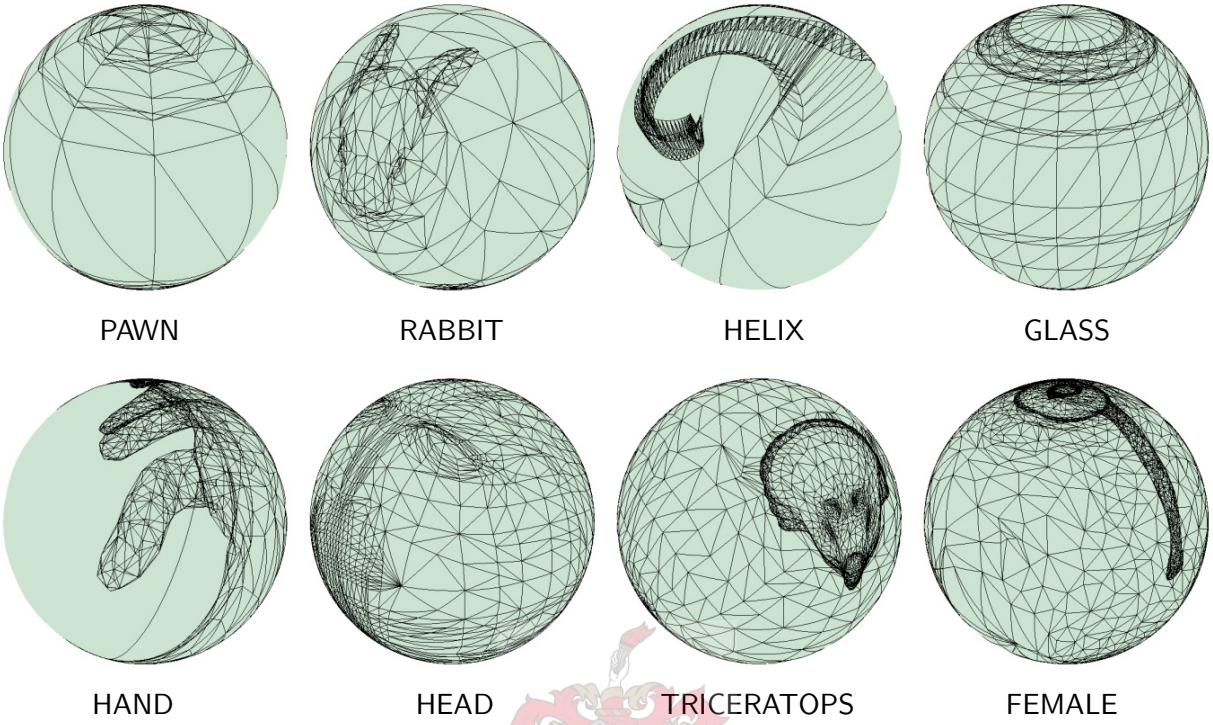


Figure 7.2 — Initial embeddings

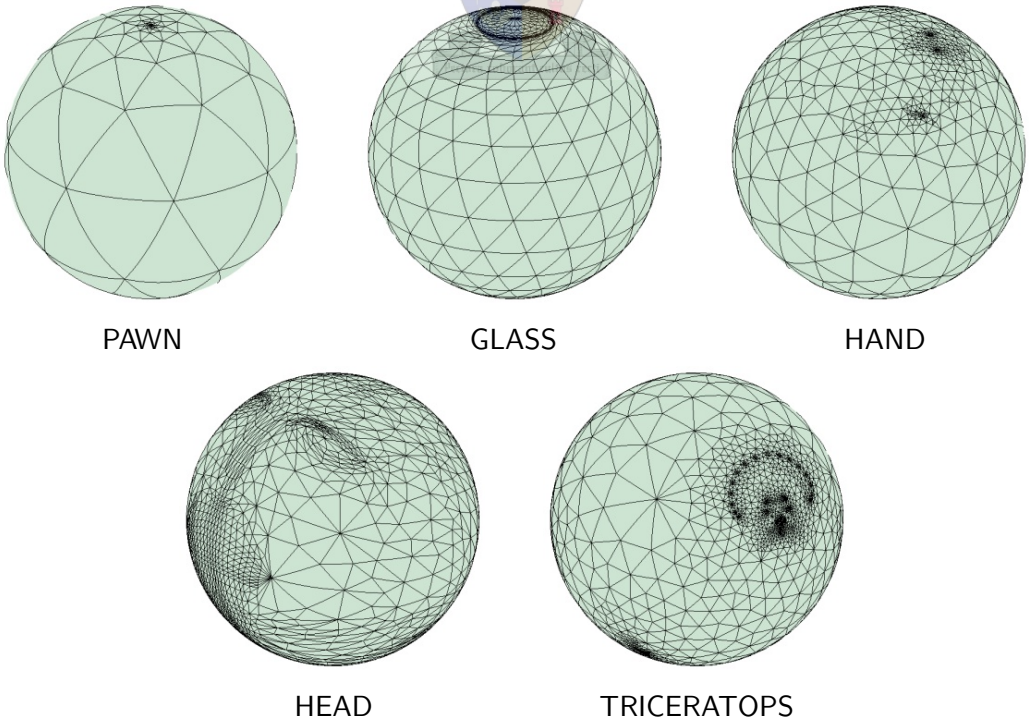


Figure 7.3 — Iterative relaxation method

7.4 Alexa's method

Results from applying Alexa's method (from section 5.3) are given in this section. The method is defined by (5.17) and (5.18). The initial embeddings depicted in Figure 7.2 were used.

The method yielded valid GC-embeddings for all the models, except for HELIX and FEMALE. These two models have long tubular parts, and an iterative scheme such as Alexa's method requires an immense number of iterations to smooth out these parts.

For HELIX we terminated the process after 200,000 iterations, which required about 2 hours of computation. For FEMALE the process was terminated after 20,000 iterations, which lasted for about 10 hours. It is not certain whether a valid embedding would ever be reached for these two models (clearly not in any desirable amount of time). Also, the collapsing behaviour of the iterative relaxation method seems less likely to occur with Alexa's method.

Figure 7.4 shows the results of the six models where Alexa's method does succeed. The results look similar to those obtained by the method of iterative relaxation.

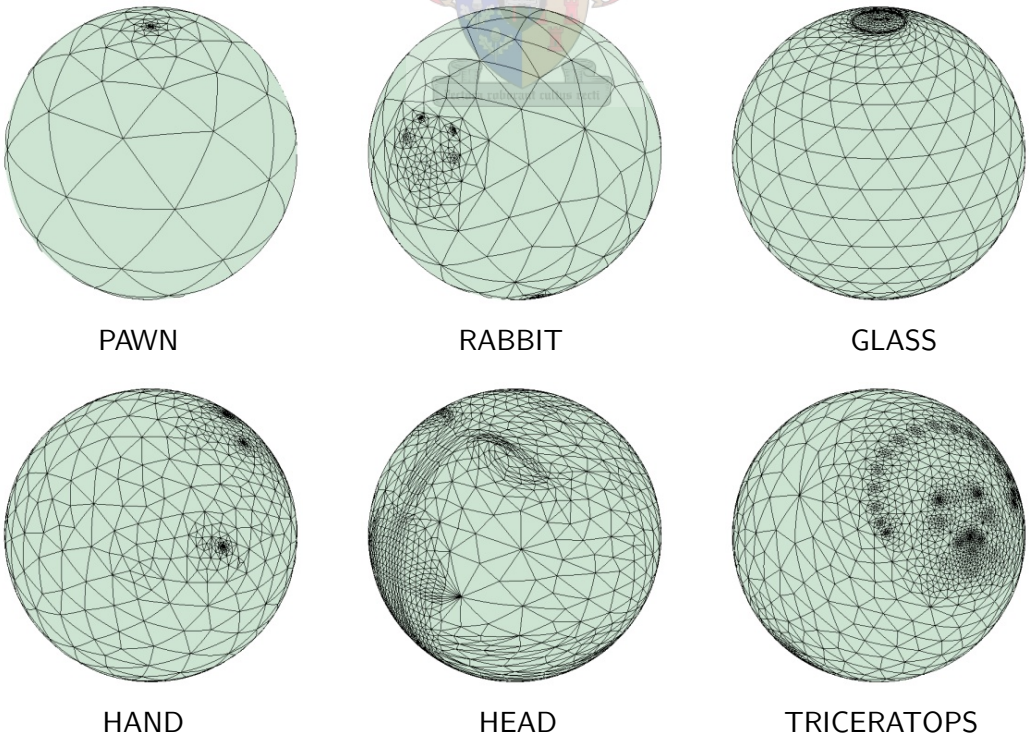


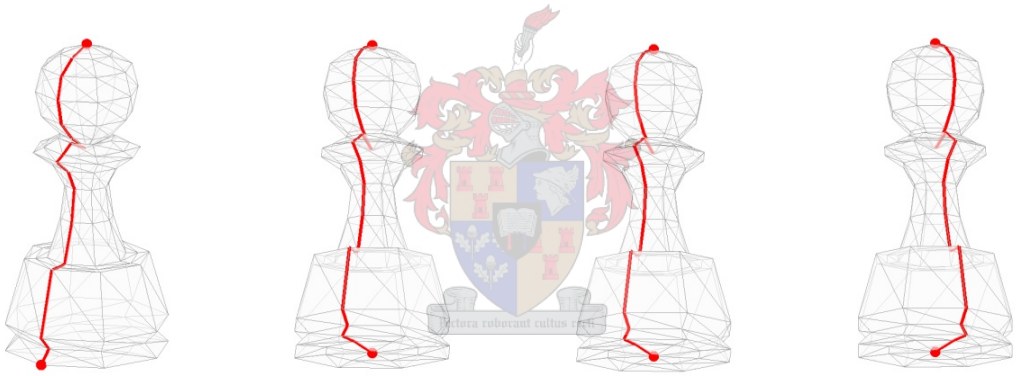
Figure 7.4 — Alexa's method

7.5 The θ–ϕ method

Results from applying the θ–ϕ method (from Chapter 6) are given in this section. Recall that the method selects two vertices and a path between them and then cuts the mesh open along this path to produce a mesh in $\mathcal{M}_B$. This "open" mesh is then embedded in the 2D θ–ϕ rectangle and folded onto the surface of the sphere.

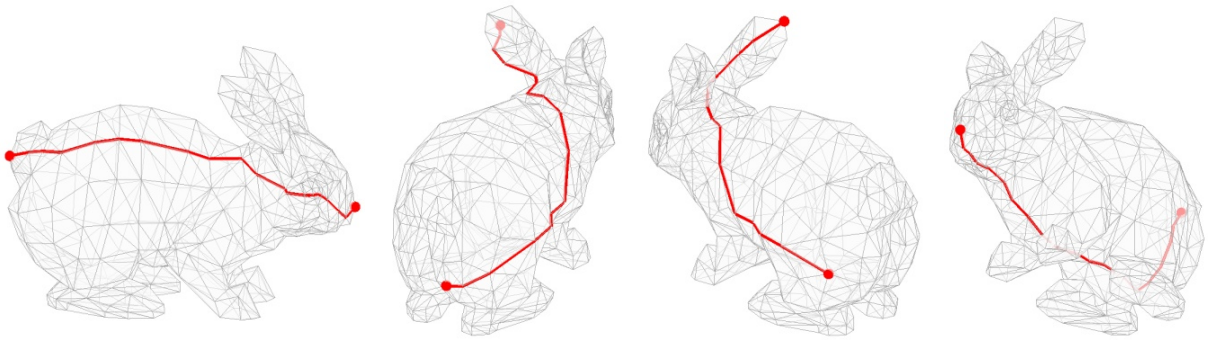
7.5.1 Pole selection

This section shows the results from implementing the four different pole selection schemes discussed in section 6.2. Figures 7.5 to 7.12 show the different models, and in each case the different cut paths resulting from different pole selections.



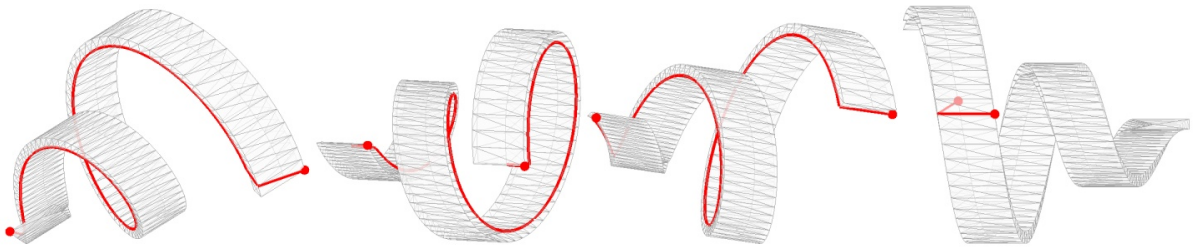
(a) physical (b) graph (c) fast-graph (d) symmetry

Figure 7.5



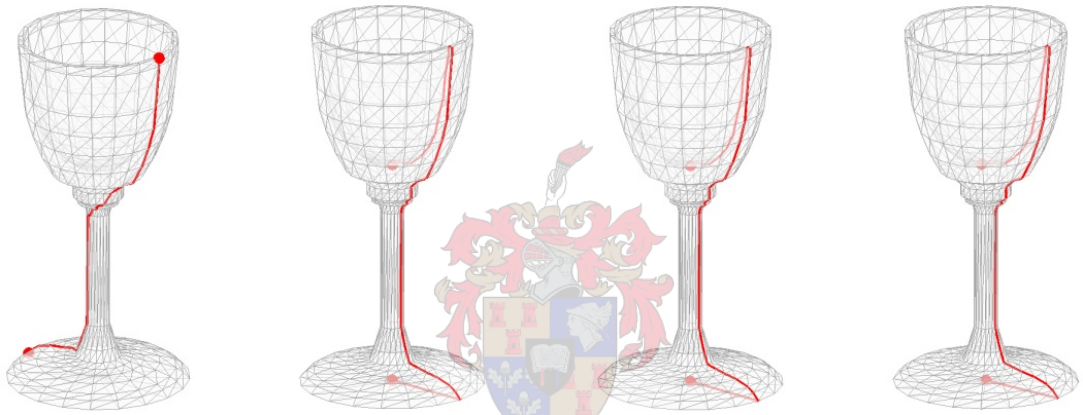
(a) physical (b) graph (c) fast-graph (d) symmetry

Figure 7.6



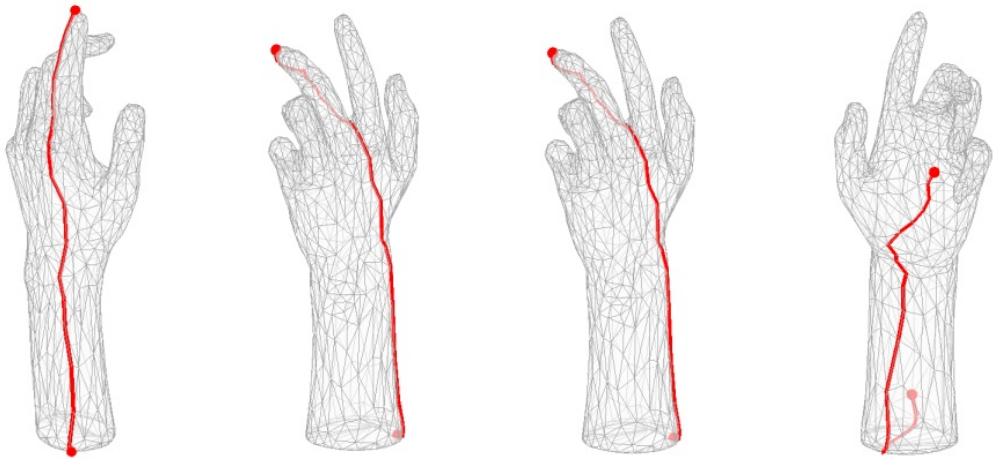
(a) physical (b) graph (c) fast-graph (d) symmetry

Figure 7.7



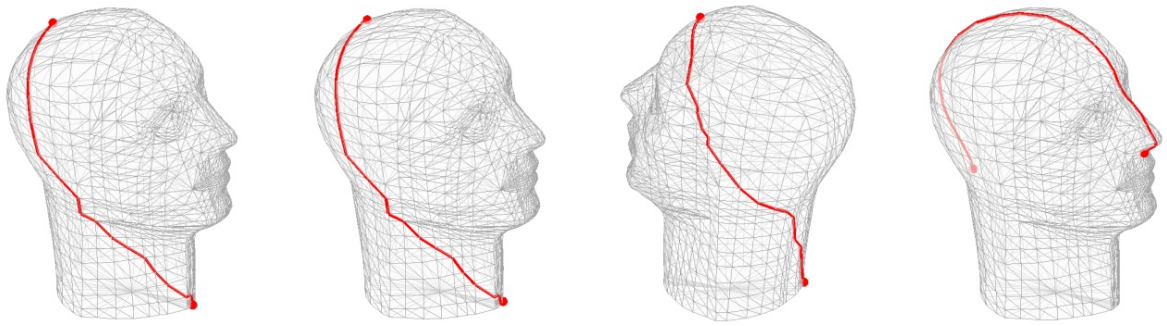
(a) physical (b) graph (c) fast-graph (d) symmetry

Figure 7.8



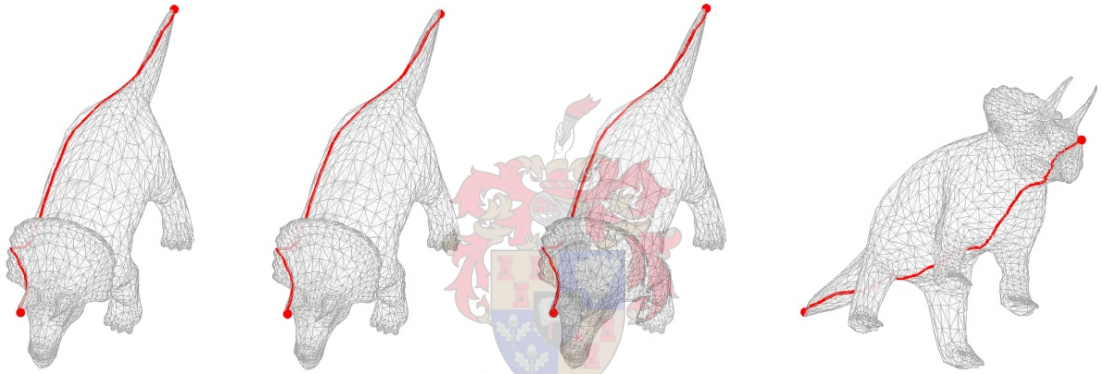
(a) physical (b) graph (c) fast-graph (d) symmetry

Figure 7.9



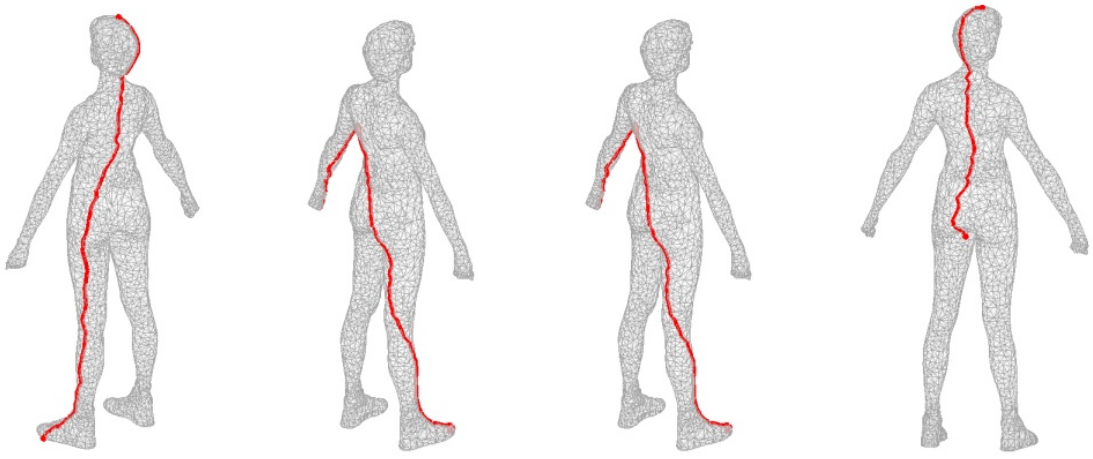
(a) physical (b) graph (c) fast-graph (d) symmetry

Figure 7.10



(a) physical (b) graph (c) fast-graph (d) symmetry

Figure 7.11



(a) physical (b) graph (c) fast-graph (d) symmetry

Figure 7.12

From the preceding examples it is evident that the graph distance method and the fast-graph method produce nearly the same poles in most cases. RABBIT and HEAD are the only cases where these two pole selections differ significantly.

The symmetry method seems to produce cut paths fairly close to the actual symmetry planes of the different models. Of course, this method fails to yield a sensible cut path for HELIX. There are also no symmetry planes in HAND, but the symmetry method does produce a usable cut path.

In section 7.6 the execution times of the the four pole selection methods are compared.

7.5.2 Embedding on the sphere

This section shows the TP-embeddings resulting from applying the θ–ϕ method on the different test models. In each case, the physical distance method was used to select the poles. For the 2D embedding we implemented Tutte weights (section 4.4.1).

Figure 7.13 shows the results obtained for all the test models. Recall from Proposition 6.2 that valid embeddings are guaranteed for all the models.

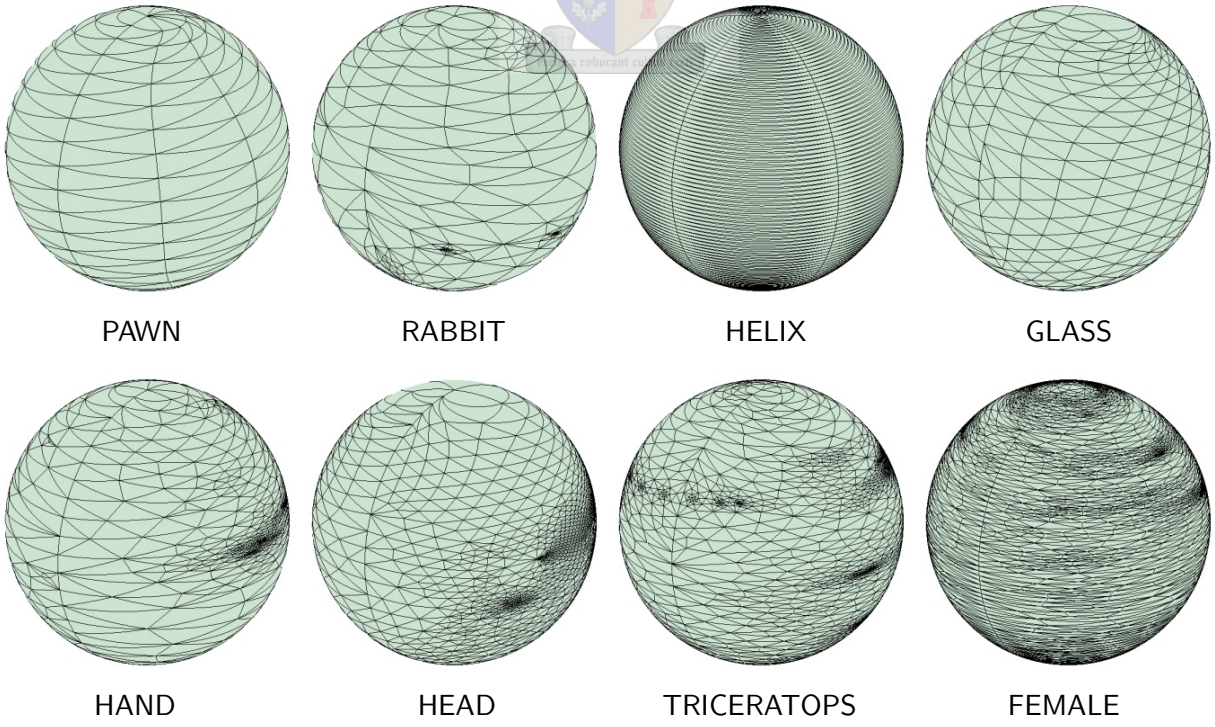


Figure 7.13 — The θ–ϕ method

Note the edges of the cut path visible in each of the embeddings in Figure 7.13. The θ–ϕ method forces these edges to lie on the line $\phi = 0$.

It seems that the embeddings obtained by the iterative methods preserve more of the geometric properties of the original model. For example, the face of HEAD is still clearly visible in the embedding obtained by Alexa's method, while it is not distinguishable in the case of the θ–ϕ method. However, where the iterative methods fail, the θ–ϕ method produces valid results.

7.6 Execution times

In this section the matter of execution time is considered. The execution times given here were all obtained with our implementation of the algorithms and methods.

Table 7.1 lists the execution times of Gotsman's method, for the two models where valid embeddings were obtained. Clearly, this method is not desirable.

Model	Gotsman's method
pawn	15.5 sec
rabbit	17 min 18 sec
helix	-
glass	-
hand	-
head	-
triceratops	-
female	-

Table 7.1

Recall from Chapter 5 that the implementation of the iterative methods involves a test to check the validity of an embedding every R iterations. Two tests were described in this thesis, namely the orientation test (section 3.4.1) and the area test (section 3.4.2).

Table 7.2 lists the execution times for performing one such test, on an embedding of each of the test models. It is clear from the figure next to the table, that the execution times of both these tests increase linearly as the number of vertices increase. Also, the orientation test is computationally more efficient.

Model	Orientation test	Area test
pawn	0.09 sec	0.29 sec
rabbit	0.22 sec	0.77 sec
helix	0.25 sec	0.86 sec
glass	0.33 sec	1.10 sec
hand	0.49 sec	1.67 sec
head	0.73 sec	2.48 sec
triceratops	1.38 sec	4.75 sec
female	1.96 sec	6.76 sec

Table 7.2

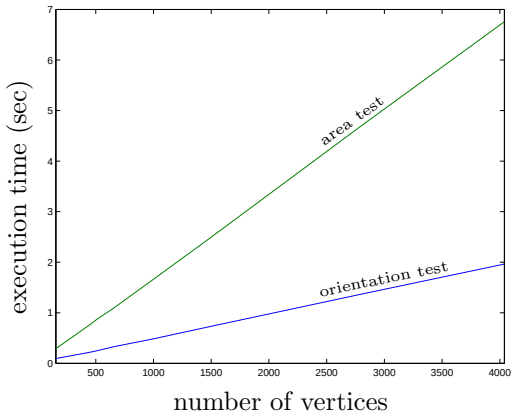


Table 7.3 shows the execution times of the method of iterative relaxation and Alexa's method for the different test models. In each case the number of iterations to a valid solution is also shown. The execution times shown were measured with no tests of validity (refer to Table 7.2 for the execution times of validity tests).

The cases of failure (graph collapsing) are indicated by "-". The cases where it is uncertain whether the method will ever reach a valid solution are indicated by "?".

Model	Iterative relaxation		Alexa's method	
	iterations	time	iterations	time
pawn	17	0.01 sec	20	0.19 sec
rabbit	-	-	104	2.74 sec
helix	-	-	?	?
glass	24	0.19 sec	37	1.50 sec
hand	700	11.19 sec	338	23.68 sec
head	1	0.36 sec	1	0.28 sec
triceratops	5,852	16 min 51 sec	610	6 min 40 sec
female	-	-	?	?

Table 7.3

Clearly, the number of iterations needed for a valid embedding does not depend on the number of vertices in the model. It does, however, depend on the geometry of the model. For the models that are "nearly convex", such as PAWN and HEAD, few iterations are required. For the models with more discernible concavity, such as HAND and TRICERATOPS, a large number of iterations are needed to smooth out these concavities. As already mentioned,

these two methods fail on "highly concave" models, such as HELIX and FEMALE.

Note also that on average, one iteration of the iterative relaxation method tends to execute faster than one iteration of Alexa's method.

Next, the θ–ϕ method is examined. Table 7.4 lists the execution times for selecting poles and a corresponding cut path, based on the four schemes from section 6.2.

Model	Physical	Graph	Fast-graph	Symmetry
pawn	0.09 sec	3.07 sec	0.09 sec	0.06 sec
rabbit	0.61 sec	31.77 sec	0.29 sec	0.23 sec
helix	0.75 sec	39.82 sec	0.31 sec	0.25 sec
glass	1.24 sec	1 min 21 sec	0.46 sec	0.44 sec
hand	2.88 sec	4 min 8 sec	1.01 sec	0.83 sec
head	6.16 sec	13 min 9 sec	1.98 sec	1.74 sec
triceratops	22.24 sec	1 h 29 min	7.03 sec	34.01 sec
female	46.08 sec	4 h 7 min	14.33 sec	7 min 11 sec

Table 7.4

Considering the selections of poles (depicted in Figures 7.5 to 7.12), it is evident that the fast-graph method should be used rather than the graph distance method. The computational cost for the graph distance method quickly becomes undesirably large as the number of vertices increase.

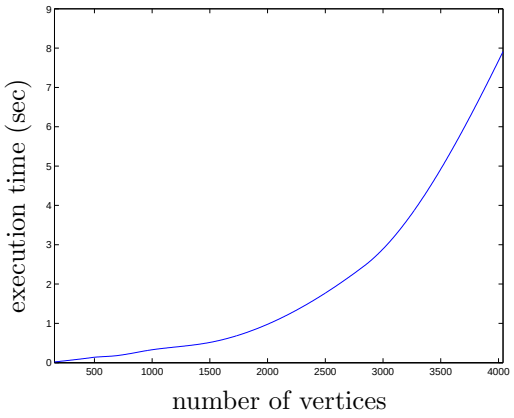
The symmetry method seems to be a good choice for the relatively small models, but interestingly, the execution time of this method also increases dramatically for the larger models.

From a purely computational viewpoint, it would seem that the fast-graph method is the best method to use. The poles resulting from this method also seem to be "good" choices (i.e., would probably cause the final embeddings to be fairly evenly distributed over the surface of the sphere).

Table 7.5 shows the execution times of applying the θ–ϕ method to the different models. Selection of poles and a cut path was disregarded in measuring these times. Any of the pole selection methods may be implemented, and the corresponding execution times from Table 7.4 may be added to those in Table 7.5. The figure next to the table is a graphical respresentation of the data from that table.

Model	The θ–ϕ method
pawn	0.02 sec
rabbit	0.12 sec
helix	0.14 sec
glass	0.17 sec
hand	0.33 sec
head	0.51 sec
triceratops	2.45 sec
female	7.91 sec

Table 7.5



Note that the θ–ϕ method was implemented to solve the linear system of Tutte 2D embedding directly. An iterative scheme such as the one discussed in section 4.5 may improve the efficiency, but because of the powerful matrix tools of MATLAB it does not seem to be necessary for these examples. For FEMALE, less than a second is required to solve the linear system. For models with tens of thousands of vertices an iterative scheme should become more efficient.

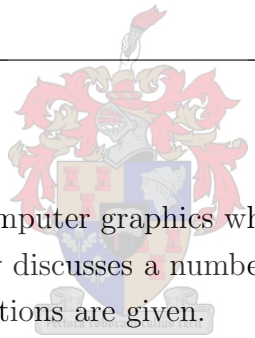
Clearly, the execution time of the θ–ϕ method increases monotonically as the number of vertices increase. This behaviour is due to the fact that the execution time is independent of the geometry of the model (contrary to the iterative relaxation method and Alexa's method).

Adding an efficient pole selection scheme such as the fast-graph method, yields execution times that are great improvements to those of the previous methods. This, and the fact that a solution is guaranteed, is a major advantage of the θ–ϕ method. The advantages and disadvantages of the θ–ϕ method are further explored in Chapter 9.

Finally we attempted to acquire GC-embeddings from the spherical embeddings obtained by the θ–ϕ method. The technique explained in section 6.7 yields valid GC-embeddings for every model except HELIX and FEMALE. For these two models edge-intersections occur when the edges are drawn as minor arcs, instead of straight line segments transformed with the mapping m defined in (6.35). These intersections occur mostly in the regions close to the poles on the sphere, where the difference in the two types of curves is most apparent.

CHAPTER 8

Applications



There are various applications in computer graphics where surface parameterisation plays an integral role. This chapter briefly discusses a number of these applications. References to papers dealing with these applications are given.

A number of the applications that may be implemented with the aid of surface parameterisation are also illustrated by means of examples.

8.1 Remeshing

One of the fundamental applications of parameterisation is that of remeshing. Remeshing is concerned with replacing an arbitrary mesh with a structured mesh.

Many algorithms in the analysis of 3D models require special structure in the meshes. Alternatively, some algorithms perform more efficiently and robustly if there is a certain structure in the given mesh. A structured mesh is typically a mesh generated by iteratively subdividing a coarse base mesh.

Hormann et. al. [21] and Kobbelt et. al. [23] proposed different techniques for remeshing a triangular mesh by first parameterising the surface.

8.2 Mesh compression

An application where parameterisation, and particularly remeshing, is used is mesh compression. This involves finding methods that compactly represent the data describing an arbitrary mesh. Gotsman et. al. [15] gives a survey of developments in this field.

8.3 Smoothing and filtering

Another widely used application is that of filtering the surface of a mesh to obtain, for example, a smoother surface.

In the case of a genus 0 surface, spherical parameterisation is applied to the mesh to obtain a spherical function or signal. This signal is then transformed to the frequency domain, using the so-called spherical harmonic transform (see for example [26]). A filter is applied to the transformed signal yielding, for example, a smoother surface in the spatial domain.

We implemented a crude version of this technique. Figure 8.1 shows the results of applying a low-pass filter with different cut-offs to the HAND model from section 7.1. The smoothing effect is clearly visible.

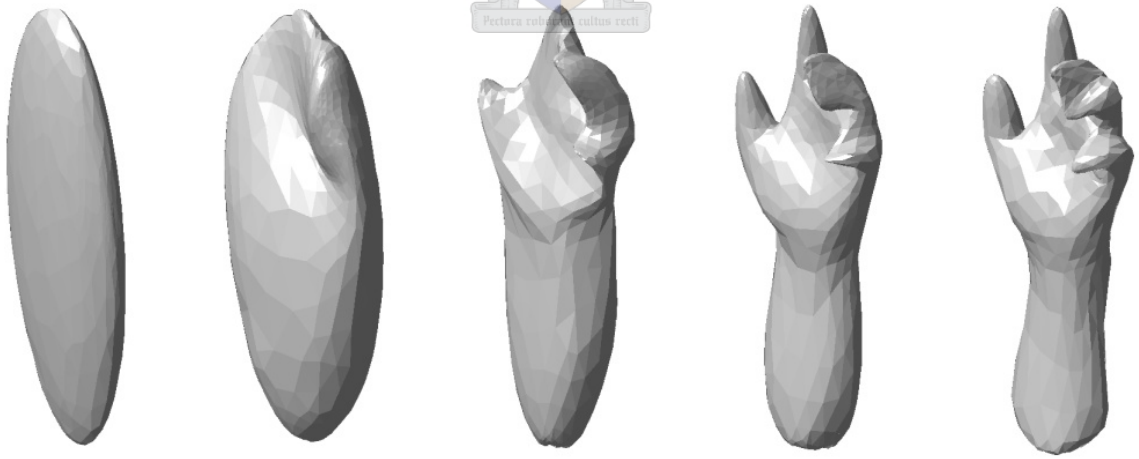


Figure 8.1

Brechbühler et. al. [3] also apply smoothing filters in the frequency domain, and obtain similar results to the ones shown above. Zhou et. al. [35] demonstrate the use of high-pass (or sharpening) filters. Kazhdan et. al. [22] explain how the spherical harmonic transform can be used to obtain a rotation invariant represention of a 3D model.

8.4 Morphing

Morphing has particular applications in animation. It involves finding a smooth transition between the surfaces of two 3D models.

Such a transition may be accomplished by parameterising the two surfaces. Points on the surface of the one model are mapped to the sphere (using the parameterisation of that model), and then mapped to the surface of the other model (using the parameterisation of that model). An interpolation scheme is then performed between the sets of points on the two surfaces.

We implemented this algorithm with linear interpolation. Figure 8.2 shows some intermediate steps of RABBIT morphing into HEAD.

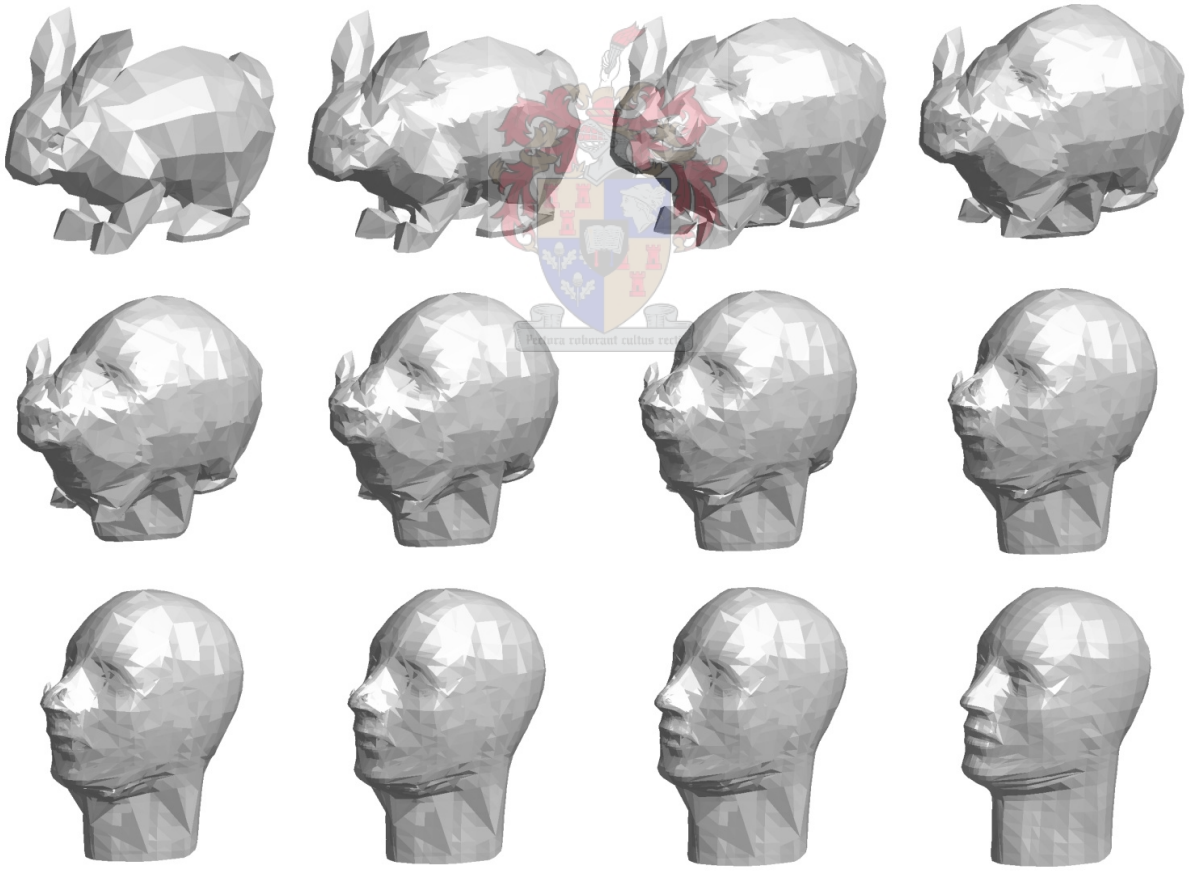


Figure 8.2

Shapiro et. al. [29] discuss a similar technique, and also show how the meshes of the two models can be merged to a single mesh. Gotsman et. al. [14] propose a method which

guarantees that this new merged mesh never intersects itself during the morphing process. Alexa [1] gives a method for aligning certain features of the two models (if, for example, we want to morph the eyes of RABBIT to the eyes of HEAD).

8.5 Texture mapping

The last application discussed in this chapter is texture mapping. This involves the mapping of a two-dimensional image onto the surface of a 3D model. Over the last few years there has been a surge of interest in the study of different techniques for mapping texture. This is due to the success in the entertainment industry, of animated films and video games in particular.

Mapping texture may be accomplished by first parameterising the surface of the given mesh. The texture is then mapped to this parameter domain, and then inversely mapped onto the surface of the mesh.

Figure 8.3 gives an example of mapping a 2D image of the earth onto the surface of HEAD. First, the surface of the model is parameterised. This was accomplished by 200 iterations using Alexa's method. The 2D image is then mapped onto the surface of the unit sphere. We now have a one-to-one mapping between the surface of the sphere and the surface of HEAD. The position of every pixel of the image on the surface of the sphere is mapped to the surface of HEAD. This yields the image shown at the bottom of the figure.

Haker et. al. [17], Praun et. al. [27] and Zhang et. al. [34] are some examples of papers concerned with different techniques for texture mapping. An important aspect of these techniques is to somehow minimise the amount by which the 2D image is stretched, or deformed, when it is mapped to the surface of a 3D model.

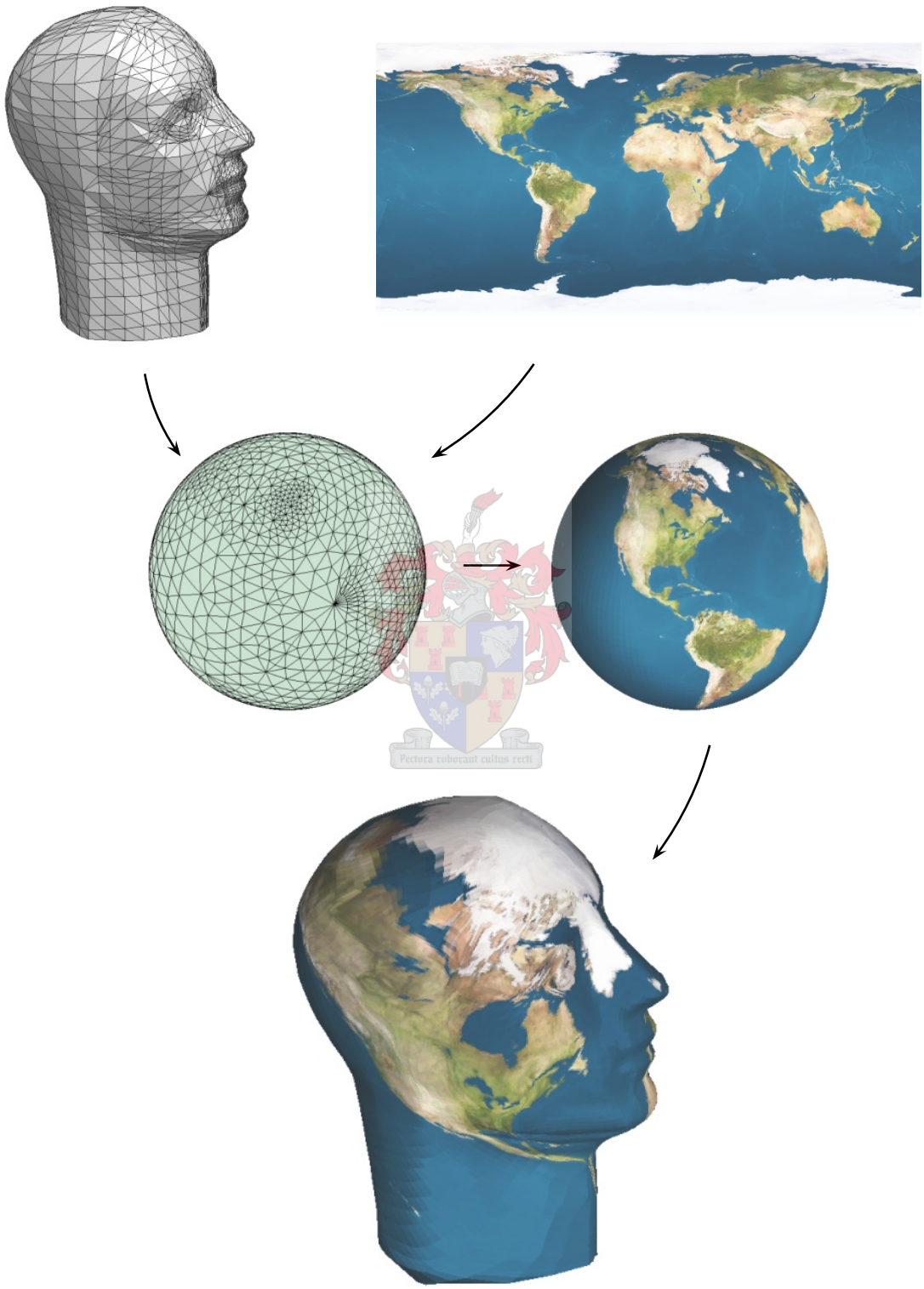
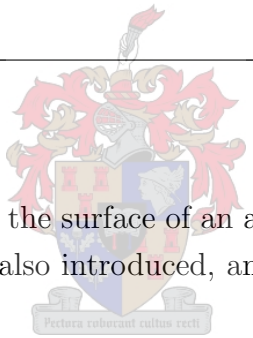


Figure 8.3

CHAPTER 9

Conclusions



Existing methods for parameterising the surface of an arbitrary mesh in the class $\mathcal{M}$ have been studied. The θ–ϕ method was also introduced, and proven to yield a valid spherical parameterisation.

In this chapter some advantages and disadvantages of the θ–ϕ method are presented. The method is also compared to the existing methods. Section 9.3 gives some recommendations for further research.

9.1 Advantages of the θ–ϕ method

Possibly the greatest advantage of the θ–ϕ method is that it guarantees a valid spherical parameterisation of any given mesh in $\mathcal{M}$.

Saba et. al. [28] claim that Gotsman's method, as explained in section 5.1.3, is the first method to produce a provably valid parameterisation. Although Gotsman's method is theoretically sound, it is impractical. Solving a huge system of non-linear equations with a generic algorithm is by no means efficient. We have presented a method that is theoretically sound, and numerically efficient compared to the existing methods.

Saba et. al. [28] also present an efficient and provable method. Their method involves an

initial embedding, followed by an iterative scheme to get rid of edge intersections. The initial embedding is obtained by cutting the mesh open along a closed curve, yielding two meshes in $\mathcal{M}_B$. These two meshes are embedded in the plane with Tutte's algorithm, and then projected to the two hemispheres of the unit sphere. Only the vertices are projected, and the edges are then drawn as minor arcs. This may result in edge intersections, and they then proceed to iteratively smooth out these intersections.

While their method does generate valid parameterisations, the θ–ϕ method immediately yields a valid parameterisation, and does not require any iteration thereafter. For this reason, the θ–ϕ method is probably more efficient than the method of Saba et. al.

Another important advantage of the θ–ϕ method is that the execution time required does not depend on the geometry (or shape) of the given mesh.

As mentioned in Chapter 7, the efficiency of the method of iterative relaxation and Alexa's method both depend heavily on the geometry of the model. Also, for models that are "extremely concave" these two methods either collapse, or fail to terminate and become impractical.

The θ–ϕ method, on the other hand, is executed in roughly the same length of time for any mesh with a fixed number of vertices, regardless of the geometry.

9.2 Disadvantages of the θ–ϕ method

A disadvantage of the θ–ϕ method is that the cut line is always visible on the spherical embedding. This may be undesirable for some applications.

Another possible disadvantage of the θ–ϕ method is that it may not always produce valid GC-embeddings. Although it was shown that the problem of parameterisation is still solved with a TP-embedding, some existing applications may specifically require a GC-embedding.

In the next section some possibilities to overcome these disadvantages, are discussed.

9.3 Recommendations

This section provides some possibilities for future research, specifically for the iterative methods and the θ–ϕ method.

9.3.1 Iterative methods

An issue that needs further investigation is that of obtaining an initial embedding for the iterative methods. Recall from Chapter 5 that an interior point **m** is required, and the nodes of the mesh are then normalised with respect to **m**.

In some cases, choosing **m** as the arithmetic mean of the nodes in the mesh yields a valid interior point. However, models exist for which this choice does not lie inside the mesh (see for example HELIX from Chapter 7). For these models, a more sophisticated method for choosing **m** is needed.

Another possible topic would be a comprehensive analysis of the iterative methods. It is not clear exactly why, and under what conditions, the method of iterative relaxation collapses for some models.

It may be possible to improve on the efficiency of the iterative methods by localising the iterative updating of vertex positions. Consider for example TRICERATOPS in Chapter 7. After about 500 iterations of the method of iterative relaxation the edge intersections were mostly smoothed out, but the embedding required an additional 5,000 iterations for very small segments in the graph where edges still intersected. Performing this much work for such a small part of the graph seems wasteful.

An important aspect not covered in this thesis is that of comparing the quality of different parameterisations for a fixed model. For such a comparison a parameter may be implemented that measures some predefined amount of distortion between the surface of the original model, and the parameterisation.

Suppose that for a specific application the edge-lengths in the spherical embedding need to be more or less equal. This would probably then lead to an even distribution of the embedding over the surface of the unit sphere. A parameter that measures, for example, the standard deviation in edge-lengths may be implemented. A spherical embedding for which this parameter is relatively low would then yield a "good" parameterisation.

9.3.2 The θ–ϕ method

As mentioned in section 9.2, one of the disadvantages of the θ–ϕ method is the fact that the cut line is always visible in the final spherical embedding. One possible solution is to implement a localised iteration scheme, to perturb only the vertices on and adjacent to the cut path. A carefully constructed updating technique may keep the embedding free from

edge intersections. Further research into this area may prove to be interesting and useful.

Another disadvantage of the θ–ϕ method is the fact that it does not always yield valid GC-embeddings. It would be interesting to find conditions under which the θ–ϕ method would give valid GC-embeddings, and possibly, to steer the method towards meeting these conditions (selecting the weights for the 2D embedding in a specific manner, for example). A local updating scheme similar to the one described above may possibly be implemented here as well, to locally correct areas of the spherical embedding where edge intersections occur due to edges being drawn as minor arcs.

9.3.3 Other methods for solving the problem of parameterisation

There are other methods for solving the problem of parameterisation which are not covered in this thesis.

The most notable of these is the so-called method of progressive meshes, due to Hoppe [19, 20]. This method involves the iterative removal of vertices and edges from the mesh until a convex mesh remains. This convex mesh is then projected onto the sphere, and the vertices are iteratively replaced in the opposite order in which they were removed. Lindstrom and Turk [24] as well as Shapiro and Tal [29] presented similar methods.

It would be interesting to see how the θ–ϕ method compares to these methods from a computational viewpoint, and also the quality of solutions obtained.

9.4 Concluding remark

The θ–ϕ method seems to compare very well with existing iterative methods. Considering all the advantages, we believe that the θ–ϕ method is a major contribution to the research of surface parameterisation.

APPENDIX A

Dijkstra's algorithm

A method for finding the distance between two vertices in a graph is discussed in this appendix. Recall from section 2.1 that the distance from a vertex u to a vertex v is the length of a shortest u–v path. The length of a path is defined in (2.5), and a shortest path between two vertices u and v is a (not necessarily unique) path with minimum length among all possible paths connecting u and v.

Note that every edge (i, j) in a graph must be assigned a weight $\rho_{ij} > 0$. If no weights are assigned, it is assumed that $\rho_{ij} = 1$ for every edge (i, j).

A popular algorithm to obtain distances between vertices is discussed next.

Dijkstra's algorithm [7] determines the distances from a single source vertex to every other vertex in the graph. The algorithm can be repeated for each vertex in the graph to obtain distances between every pair of vertices in the graph.

Consider a graph G with vertex set $\{1, 2, \ldots, p\}$ and a source vertex v. We wish to calculate distances from v to every other vertex in the graph. Dijkstra's algorithm is given below.

The algorithm implements a priority queue Q to store values of the vertices. The choice of a priority queue optimises execution time [13, p. 586]. A priority queue is a list of elements, each of which has a priority in the list, depending on the value of a key. The function removeMin(Q) removes an element with smallest key value from the queue Q, and

returns the label of that vertex.

Algorithm A.1 : (Dijkstra's algorithm)
INPUT: A graph G with p vertices and a source vertex $v \in V(G)$
OUTPUT: Distances from v to every vertex in G

1. $d(v) \leftarrow 0$, *and* $d(u) \leftarrow \infty$, *for every* $u \neq v$
2. *Let Q contain all the vertices of G, with the $d(\cdot)$ labels as keys*
3. **while** *Q is not empty* **do**
   $u \leftarrow \text{removeMin}(Q)$
   for *each vertex* $z \in N(u) \cap Q$ **do**
   if $d(u) + \rho_{uz} < d(z)$ **then**
   $d(z) \leftarrow d(u) + \rho_{uz}$
   update key of z to $d(z)$ in Q
   end
   end
   end
4. **return** $d(u)$ *for every* $u \in V(G)$

During the execution of the algorithm, a value $d(u)$ is stored for every $u \in V(G)$, which denotes the length of the shortest v–u path found so far. Initially, $d(v) = 0$ and $d(u)$ is set to ∞ for every vertex $u \in V(G) \setminus \{v\}$. The set $Q \subset V(G)$ is defined to the set of vertices of which the correct distances from v have not yet been determined. Initially, $Q = V(G)$.

At each iteration of the algorithm a vertex $u \in Q$ with smallest $d(u)$ is selected, and removed from Q. For the first iteration v is removed from Q. Once a new vertex u has been removed from Q, the value of $d(z)$ is updated for every vertex z adjacent to u and inside Q, to compensate for the fact that there may be a shorter path from v passing through u. For a vertex z adjacent to u, the updating is performed as follows,

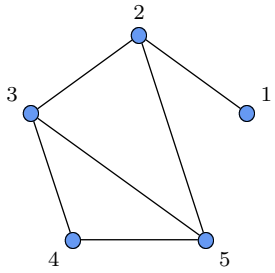
$$\textbf{if } \; d(u) + \rho_{uz} < d(z) \; \textbf{ then } \; d(z) \leftarrow d(u) + \rho_{uz}, \tag{A.1}$$

with ρ_{uz} the weight of edge (u, z). The operation (A.1) is sometimes refered to as edge relaxation. The process is repeated until Q is empty, so that the distance from v to every other vertex is known.

The algorithm returns values $d(u)$ that denote the distance from v to every vertex u in the graph. With an added labelling scheme it is possible to recover shortest paths, rather than

just the lengths.

Figure A.1 shows an example of a small graph and results in tabular form when Dijkstra's algorithm is applied to vertex 1.

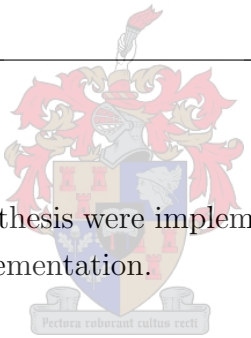


u	$d(1)$	$d(2)$	$d(3)$	$d(4)$	$d(5)$	C
-	0	∞	∞	∞	∞	$\emptyset$
1	0	1	∞	∞	∞	$\{1\}$
2	0	1	2	∞	2	$\{1,2\}$
3	0	1	2	3	2	$\{1,2,3\}$
5	0	1	2	3	2	$\{1,2,3,5\}$
4	0	1	2	3	2	$\{1,2,3,4,5\}$

Figure A.1

APPENDIX B

MATLAB implementation



The methods and algorithms in this thesis were implemented in MATLAB. This appendix gives a short description of the implementation.

B.1 Graphs and meshes

The data describing a triangular mesh with n vertices and m faces is stored in two matrices: a vertex list `VL` and a face list `FL`. The vertex list is an $n \times 3$ matrix, such that the ith row gives the coordinates of vertex i. The face list is an $m \times 3$ matrix, such that the ith row gives the three indices of the vertices of face i. The function `drawmesh` draws such a mesh, in the colour specified by `col`.

```
function drawmesh(VL,FL,col,edges)
f = size(FL,1);
X = VL(:,1); Y = VL(:,2); Z = VL(:,3);
for j = 1:f,
    face = FL(j,:);
    if edges, patch(X(face),Y(face),Z(face),col);
    else patch(X(face),Y(face),Z(face),col,'EdgeColor','none'); end
end
```

```
view(50,28)
axis image, axis tight, axis off, axis vis3d
```

A graph G is described by means of an edge list `EL`. This is an $q \times 2$ matrix, where q denotes the number of edges in G. The ith row of this matrix gives the indices of the two vertices of edge i. The function `edgelist` may be used to obtain the edge list of the underlying graph of a given mesh (see Definition 2.1).

```
function EL = edgelist(FL)
v = max(max(FL)); f = size(FL,1);
A = zeros(v,v);
for j = 1:f,
    f = FL(j,:);
    A(f(1),f(2)) = 1; A(f(1),f(3)) = 1; A(f(2),f(3)) = 1;
    A(f(2),f(1)) = 1; A(f(3),f(1)) = 1; A(f(3),f(2)) = 1;
end
A = triu(A); [I,J] = find(A);
EL = [I J];
```

The function `checkmesh` implements Proposition 2.8. The three conditions of the propsition are tested, and if all three test positive, the mesh belongs to the class $\mathcal{M}$.

```
function checkmesh(VL,FL);
EL = edgelist(FL);
v = size(VL,1); e = size(EL,1); f = size(FL,1);

% 2-manifoldness
A = zeros(v,v);
I = EL(:,1); J = EL(:,2);
A(v*(I-1) + J) = 1;
A(v*(J-1) + I) = 1;
E = [];
for j = 1:v, if sum(sum(FL == j)) ~= sum(A(j,:)), E = [E; j]; end; end
test1a = isempty(E);
if test1a,
    E = [];
    for i = 1:v,
        [I,J] = find(FL == i);
        face = I(1);
        ijk = FL(face,:); ijk = ijk(find(ijk ~= i));
        j = ijk(1); k = ijk(2);
        first = j;
```

```
        cyclelength = 1;
        while k ~= first,
            [I1,J1] = find(sum(FL == i,2).*sum(FL == k,2));
            g = I1; g = g(find(g ~= face));
            ijk = FL(g,:);
            ijk = ijk(find(ijk ~= i)); ijk = ijk(find(ijk ~= k));
            j = k; k = ijk; face = g;
            cyclelength = cyclelength + 1;
        end
        if cyclelength ~= length(I), E = [E; j]; end
    end
    test1b = isempty(E);
else
    test1b = 0;
end
test1 = test1a & test1b;

% connectivity
i = 1; D = zeros(1,v); D(i) = 1;
C = ones(1,v); j = i; inserted = 1;
while inserted ~= 0,
    nb = find(A(j,:));
    D(nb) = 1; C(j) = 0;
    if sum(D & C) == 0,
        inserted = 0;
    else
        inserted = 1;
        j = find(D & C); j = j(1);
    end
end
test2 = (sum(D) == v);

% Euler's formula
test3 = (v-e+f == 2);

if test1 & test2 & test3, disp('Valid mesh');
else
    if ~test1, disp('Mesh is not 2-manifold !'); end
    if ~test2, disp('Mesh is disconnected !'); end
    if ~test3, disp('Euler''s formula does not hold !'); end
end
```

The function `checkfaces` implements Algorithm 2.9, to test and correct the orientation of faces in a given face list. We assume that face `fstar` is correctly orientated. The

algorithm returns a face list in which every face is orientated as `fstar`. The first two functions implement edges$[F, f]$ as defined in (2.24), and otherface$[F, e, f]$ as defined in (2.25).

```
function [a,b,c] = edges(FL,f);
f = FL(f,:); a = f([1,2]); b = f([2,3]); c = f([3,1]);


function f = otherface(FL,e,f_);
f = find(sum((FL == e(1)) | (FL == e(2)),2) == 2); f = f(find(f ~= f_));


function nFL = checkfaces(FL,fstar);
nFL = FL; FT = []; FAC = fstar;
[e1,e2,e3] = edges(nFL,fstar);
fp = otherface(FL,e1,fstar); if sum(FAC == fp) == 0, FT = [FT; fp e1]; end
fp = otherface(FL,e2,fstar); if sum(FAC == fp) == 0, FT = [FT; fp e2]; end
fp = otherface(FL,e3,fstar); if sum(FAC == fp) == 0, FT = [FT; fp e3]; end
while size(FAC,1) < size(FL,1),
    f = FT(end,1); e = FT(end,2:3); FT = FT(1:end-1,:);
    [e1,e2,e3] = edges(FL,f);
    if sum((e1 == e(1)) | (e1 == e(2))) == 2, ee = e1;
    elseif sum((e2 == e(1)) | (e2 == e(2))) == 2, ee = e2;
    else ee = e3;
    end
    if sum(ee == e) == 2,
        nFL(f,:) = fliplr(FL(f,:));
    end
    FAC = [FAC; f];
    [e1,e2,e3] = edges(nFL,f);
    fp = otherface(FL,e1,f); if sum(FAC == fp) == 0, FT = [FT; fp e1]; end
    fp = otherface(FL,e2,f); if sum(FAC == fp) == 0, FT = [FT; fp e2]; end
    fp = otherface(FL,e3,f); if sum(FAC == fp) == 0, FT = [FT; fp e3]; end
end
```

B.2 Surface parameterisation

The function `gcdrawing` may be used to draw a specific GC-drawing of a mesh, from a face list, and an $N \times 3$ matrix `V` which contains coordinates $\mathbf{v}_i \in S_0$, for every vertex i. The function `minorarc` draws a minor arc on the surface of the unit sphere between two

given points.

```
function minorarc(v1,v2);
lambda = 0:0.05:1;
g = lambda'*v1 + (1-lambda)'*v2;
normg = sqrt(sum(g.^2,2)); g = g./[normg normg normg];
plot3(g(:,1),g(:,2),g(:,3),'k');


function gcdrawing(V,FL)
EL = edgelist(FL); e = size(EL,1);
figure, hold on
for j = 1:e,
    edge = EL(j,:);
    minorarc(V(edge(1),:),V(edge(2),:));
end
[x,y,z] = sphere(30); r = 0.999;
surf(r*x,r*y,r*z,'EdgeColor','none','FaceColor',[0.804, 0.894, 0.824]);
view(46,18)
axis image, axis tight, axis off, axis vis3d
```

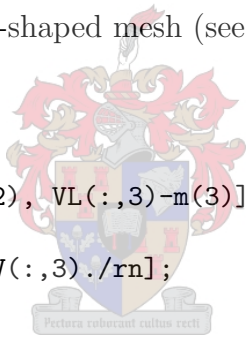
The next two functions, `gctestorientation` and `gctestarea`, implement the orientation test from section 3.4.1, and the area test from section 3.4.2. Both of these functions test a given GC-drawing to determine whether it is in fact a valid GC-embedding. The function `areasphertria` is used in `gctestarea`, and computes the area of a spherical triangle with vertices at `v1`, `v2` and `v3`.

```
function valid = gctestorientation(V,FL)
f = size(FL,1);
S = zeros(f,1);
for j = 1:f,
    S(j) = sign(dot(cross(V(FL(j,1),:),V(FL(j,2),:)),V(FL(j,3),:)));
end
valid = 0;
if min(S) == 1, valid = 1; end


function area = areasphertri(v1,v2,v3)
na = v2 - dot(v1,v2)*v1; nb = v3 - dot(v1,v3)*v1;
a = acos(dot(na,nb)/norm(na)/norm(nb));
na = v1 - dot(v2,v1)*v2; nb = v3 - dot(v2,v3)*v2;
b = acos(dot(na,nb)/norm(na)/norm(nb));
```

```
na = v2 - dot(v3,v2)*v3; nb = v1 - dot(v3,v1)*v3;
c = acos(dot(na,nb)/norm(na)/norm(nb));
area = a + b + c - pi;


function valid = gctestarea(V,FL)
f = size(FL,1);
A = 0;
for j = 1:f,
    A = A + areasphertri(V(FL(j,1),:),V(FL(j,3),:),V(FL(j,2),:));
end
valid = abs(4*pi - A) < 1e-12;
```

The function `normvl` implements equation (3.17) for every vertex in a mesh. Thus, every vertex in `VL` is centred and normalised. This function may, for example, be used to obtain a GC-embedding of a convex or star-shaped mesh (see section 3.5).

```
function V = normvl(VL)
m = mean(VL);
V = [VL(:,1)-m(1), VL(:,2)-m(2), VL(:,3)-m(3)];
rn = sqrt(sum(V.^2,2));
V = [V(:,1)./rn, V(:,2)./rn, V(:,3)./rn];
```



B.3 2D planar embeddings

For a given mesh in $\mathcal{M}_B$ the function `boundary` returns a vector `B` containing indices of the vertices in the boundary cycle, in the order of the cycle.

```
function B = boundary(FL);
v = max(max(FL));
EL = edgelist(FL);
A = zeros(v,v);
I = EL(:,1); J = EL(:,2);
A(v*(I-1) + J) = 1; A(v*(J-1) + I) = 1;
E = zeros(1,v);
for j = 1:v,
    if sum(sum(FL == j)) == sum(A(j,:)) - 1, E(j) = 1; end
end
i = find(E); i = i(1);
```

```
nb = find(A(i,:) & E); j = nb(1); last = nb(2);
B = [i j];
while j ~= last,
    nb = find(A(j,:) & E);
    oldj = j; j = nb(find(nb ~= i)); i = oldj;
    B = [B j];
end
```

The function `weightmatrix2d` sets up the matrix `A` for the generalised 2D Tutte embedding algorithm (see section 4.3). A mesh must be supplied, with its corresponding boundary cycle `B`. Chord length weights are used, depending on the value of `rho`.

```
function A = weightmatrix2d(VL,FL,B,rho)
EL = edgelist(FL);
v = size(VL,1); e = size(EL,1);
A = zeros(v,v);
for j = 1:e,
    edge = EL(j,:); omega = norm(VL(edge(1),:) - VL(edge(2),:));
    A(edge(1),edge(2)) = omega^rho;
    A(edge(2),edge(1)) = omega^rho;
end
for j = 1:v, A(j,:) = -A(j,:)/sum(A(j,:)); A(j,j) = 1; end
for j = B, A(j,:) = zeros(1,v); A(j,j) = 1; end
```

The function `tutte2d` performs generalised 2D Tutte embedding for a given mesh in $\mathcal{M}_B$. The boundary cycle `B` must be supplied, together with an $n \times 2$ matrix `U`, that gives the 2D coordinates of the boundary vertices. Chord length weights are used for a given `rho`. If `iter` is 1, then `n_iter` iterations are used to approximate the solution of the linear system. If not, the linear system is solved directly.

```
function [x,y] = tutte2d(VL,FL,B,U,rho,iter,n_iter)
v = size(VL,1);
px = U(:,1); py = U(:,2);
A = weightmatrix2d(VL,FL,B,rho);
bx = zeros(v,1); by = zeros(v,1);
bx(B) = px; by(B) = py;
if iter == 1,
    x = zeros(v,1); x(B) = px;
    y = zeros(v,1); y(B) = py;
    A = A - diag(ones(v,1));
    for r = 1:n_iter,
```

```
        for j = 1:v,
            x(j) = bx(j) - A(j,:)*x;
            y(j) = by(j) - A(j,:)*y;
        end
    end
else
    x = sparse(A)\bx; y = sparse(A)\by;
end
```

The function `draw2dembedding` plots the 2D planar embedding obtained with `tutte2d`. The boundary edges are drawn in thick red lines.

```
function draw2dembedding(FL,B,x,y)
figure, hold on
EL = edgelist(FL);
for j = 1:size(EL,1),
    e1 = EL(j,1); e2 = EL(j,2);
    plot([x(e1) x(e2)],[y(e1) y(e2)],'k');
end
for j = 1:length(B)-1,
    plot([x(B(j)) x(B(j+1))],[y(B(j)) y(B(j+1))],'r','LineWidth',2);
end
plot([x(B(1)) x(B(end))],[y(B(1)) y(B(end))],'r','LineWidth',2);
plot(x,y,'k.','MarkerSize',10);
axis([min(x)-.2, max(x)+.2, min(y)-.2, max(y)+.2]);
```

B.4 Iterative methods for spherical embedding

The function `weightmatrix3d` sets up the matrix `W` for spherical Tutte embedding (see section 5.1). Chord length weights are used, for a specific `rho`. If `normrows` is 1, then the rows of `W` are normalised such that they sum to 1.

```
function W = weightmatrix3d(VL,FL,rho,normrows)
EL = edgelist(FL);
v = size(VL,1); e = size(EL,1);
W = zeros(v,v);
for j = 1:e,
    edge = EL(j,:);
    omega = norm(VL(edge(1),:) - VL(edge(2),:));
```

```
    W(edge(1),edge(2)) = omega^rho;
    W(edge(2),edge(1)) = omega^rho;
end
if normrows == 1, for j = 1:v, W(j,:) = W(j,:)/sum(W(j,:)); end; end
```

The function `methodgotsman` implements Gotsman's method from section 5.1.3. Chord length weights are used for a specific `rho`. The maximum number of iterations must also be supplied. The function calls `gotsmanfun`, which returns a function value `F`. The built-in function `fsolve` attempts to find a solution where `F` is zero.

```
function F = gotsmanfun(X,W)
x = X(:,1); y = X(:,2); z = X(:,3); a = X(:,4);
F = [(x.^2 + y.^2 + z.^2 - 1), (a.*x - W*x), (a.*y - W*y), (a.*z - W*z)];
```

```
function V = methodgotsman(VL,FL,rho,maxiter);
v = size(VL,1);
W = weightmatrix3d(VL,FL,rho,1);
X0 = normvl(VL); a0 = 4*pi/sqrt(3*v) + zeros(v,1);
X0 = [X0 a0];
options = optimset('Display','iter','MaxFunEvals',Inf,'MaxIter',maxiter);
[X,fval,exitflag] = fsolve(@gotsmanfun,X0,options,W);
V = X(:,1:3);
```

The function `methoditerative` implements the method of iterative relaxation from section 5.2. Chord length weights are used for a given `rho`. The function checks the solution for validity every `R` iterations with the orientation test. The process is continued until a valid solution is obtained, or until `maxiter` iterations have been applied.

```
function [V,valid] = methoditerative(VL,FL,rho,maxiter,R)
v = size(VL,1);
W = weightmatrix3d(VL,FL,rho,1);
V = normvl(VL);
valid = 0; r = 0;
while ~valid & r < maxiter,
    r = r + 1;
    U = W*V;
    rn = sqrt(sum(U.^2,2));
    V = [U(:,1)./rn, U(:,2)./rn, U(:,3)./rn];
    if mod(r,R) == 0, valid = gctestorientation(V,FL); end
end
```

```
valid = gctestorientation(V,FL);
```

The function `methodalexa` implements Alexa's method from section 5.3. The function checks the solution for validity every `R` iterations with the orientation test. The process is continued until a valid solution is obtained, or until `maxiter` iterations have been applied.

```
function [V,valid] = methodalexa(VL,FL,maxiter,R)
EL = edgelist(FL);
v = size(VL,1); e = size(EL,1);
A = zeros(v,v);
for j = 1:e,
    A(EL(j,1),EL(j,2)) = 1;
    A(EL(j,2),EL(j,1)) = 1;
end
V = normvl(VL);
valid = 0; r = 0;
while ~valid & r < maxiter,
    r = r + 1; oldV = V;
    for j = 1:v,
        vi = oldV(j,:);
        nb = find(A(j,:));
        q = [0 0 0]; c = 0;
        for k = 1:length(nb),
            edge = vi - oldV(nb(k),:);
            q = q + edge*norm(edge);
            if norm(edge) > c, c = norm(edge); end
        end
        q = q/length(nb)/c;
        V(j,:) = (vi - q)/norm(vi - q);
    end
    if mod(r,R) == 0, valid = gctestorientation(V,FL); end
end
valid = gctestorientation(V,FL);
```

B.5 The θ–ϕ method

The function `dijkstra` implements Dijkstra's algorithm (see Appendix A). The input matrix `G` contains the weights of the edges of the graph, and `v` is the source vertex. This function will be used in some of the functions that follow.

```
function [d,pred] = dijkstra(G,v);
p = size(G,1);
pred = zeros(p,1);
d = zeros(p,1) + Inf; d(v) = 0;
Q = [[1:p]' d];
while ~isempty(Q),
    [a,b] = min(Q(:,2)); u = Q(b,1); Q = [Q(1:b-1,:); Q(b+1:end,:)];
    Gu = zeros(1,p); Gu(Q(:,1)) = 1;
    neighb = find(Gu.*G(u,:));
    for z = neighb,
        if d(u) + G(u,z) < d(z),
            d(z) = d(u) + G(u,z);
            Q(find(Q(:,1) == z),2) = d(z);
            pred(z) = u;
        end
    end
end
```

The following three functions, `tppolesphysical`, `tppolesgraph` and `tppolesfastgraph`, implement three of the pole selection methods discussed in section 6.2, for use in the θ–ϕ method. Each of these functions returns the two indices of the selected poles.

```
function [north,south] = tppolesphysical(VL);
v = size(VL,1);
D = zeros(v,v);
for j = 1:v, for k = 1:v, D(j,k) = norm(VL(j,:) - VL(k,:)); end; end
[a,b] = max(D); [c,d] = max(a);
north = b(d); south = d;
```

```
function [north,south] = tppolesgraph(VL,FL);
v = size(VL,1);
G = weightmatrix3d(VL,FL,1,0);
D = zeros(v,v);
for j = 1:v, D(j,:) = [dijkstra(G,j)]'; end
[a,b] = max(D); [c,d] = max(a);
north = b(d); south = d;
```

```
function [north,south] = tppolesfastgraph(VL,FL);
G = weightmatrix3d(VL,FL,1,0);
i = 1;
```

```
[a,north] = max(dijkstra(G,i));
[b,south] = max(dijkstra(G,north));
```

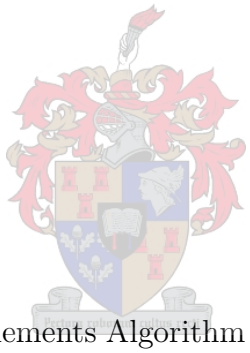
The function `tpcutpath` selects a shortest path through the underlying graph of the given mesh, between vertices `north` and `south`. This path `P` then serves as the cut path in the θ–ϕ method.

```
function P = tpcutpath(VL,FL,north,south)
G = weightmatrix3d(VL,FL,1,0);
[d,pred] = dijkstra(G,north);
P = south; v = south; ov = v;
while v ~= north, v = pred(ov); P = [P; v]; ov = v; end
P = P(end:-1:1);
```

The function `tppolessymmetry` implements the symmetry method from section 6.2.5 for selecting poles and a cut path. The function returns indices of the two poles, as well as a cut path between them.

```
function [north,south,P] = tppolessymmetry(VL,FL);
v = size(VL,1);
Y = (VL(FL(:,1),:) + VL(FL(:,2),:) + VL(FL(:,3),:))/3;
n = size(Y,1);
a = VL(FL(:,1),:); b = VL(FL(:,2),:); c = VL(FL(:,3),:);
w = cross(b-a,c-a,2); w = sqrt(sum(w.^2,2))/2;
W = diag(w)/sum(w);
C = Y'*W*Y;
[S,L] = eig(C);
s1 = S(:,1); s2 = S(:,2); s3 = S(:,3);
m = sum(W*Y)/trace(W);
d = zeros(n,1); s = s1/norm(s1);
for j = 1:n, d(j) = s'*(Y(j,:)'-m'); end; D1 = sum(d);
d = zeros(n,1); s = s2/norm(s2);
for j = 1:n, d(j) = s'*(Y(j,:)'-m'); end; D2 = sum(d);
d = zeros(n,1); s = s3/norm(s3);
for j = 1:n, d(j) = s'*(Y(j,:)'-m'); end; D3 = sum(d);
[a,b] = min(abs([D1 D2 D3]));
if b == 1, s = s1/norm(s1);
elseif b == 2, s = s2/norm(s2);
else s = s3/norm(s3);
end
d = zeros(v,1); for j = 1:v, d(j) = s'*(VL(j,:)' - m'); end
tol = 0.2*max(abs(d));
```

```
closepoints = find(abs(d) < tol);
EL = edgelist(FL);
EE = 0*EL;
for j = 1:length(closepoints), EE = EE | (EL == closepoints(j)); end
closeedges = find(sum(EE,2) == 2);
CEL = EL(closeedges,:);
G = zeros(v,v);
for j = 1:size(CEL,1),
    edge = CEL(j,:);
    omega = norm(VL(edge(1),:) - VL(edge(2),:));
    G(edge(1),edge(2)) = omega;
    G(edge(2),edge(1)) = omega;
end
i = closepoints(1); d = dijkstra(G,i); d(d == Inf) = 0; [a,north] = max(d);
[d,pred] = dijkstra(G,north); d(d == Inf) = 0; [b,south] = max(d);
P = south;
v = south; ov = v;
while v ~= north,
    v = pred(ov);
    P = [P; v];
    ov = v;
end
P = P(end:-1:1);
```



The function `tpfacestochange` implements Algorithm 6.1, which identifies all the faces in the given mesh that change when the mesh is cut open. The function uses `faces` which returns indices of the two faces sharing a given edge.

```
function [f1,f2] = faces(FL,edge)
[i,j] = find(sum(FL == edge(1) | FL == edge(2),2) == 2);
f1 = i(1); f2 = i(2);
```

```
function FC = tpfacestochange(FL,P)
n = P(1); s = P(end);
p = P(2:end); k = length(p) - 1;
FC = []; i = 1;
[f1,f2] = faces(FL,[n,p(1)]); f = f1;
face = FL(f,:); q = face(find(face ~= n & face ~= p(1)));
for j = 1:k,
    while p(j+1) ~= q(i),
        FC = [FC; f];
        [f1,f2] = faces(FL,[q(i),p(j)]);
```

```
            if f1 == f, f = f2; else f = f1; end
            face = FL(f,:);
            i = i + 1;
            q(i) = face(find(face ~= q(i-1) & face ~= p(j)));
        end
        i = i - 1;
    end
    FC = [FC; f];
```

The function `tpcutmeshopen` cuts the given mesh open (as explained in section 6.3) along a given cut path `P`. The list of faces `FC`, obtained with `tpfacestochange`, must also be supplied. The function returns a vertex list `oVL` and a face list `oFL` of the "open" mesh.

```
function [oVL,oFL] = tpcutmesh(VL,FL,P,FC)
oVL = [VL; VL(P(end-1:-1:2),:)];
oFL = FL;
N = size(VL,1); p = P(2:end-1); k = length(p);
for j = 1:length(FC),
    face = FL(FC(j),:);
    for l = 1:3,
        if sum(p == face(l)) > 0,
            face(l) = N + k - find(p == face(l)) + 1;
        end
    end
    oFL(FC(j),:) = face;
end
```

The function `methodthetaphi` implements the θ–ϕ method from Chapter 6. Note that the physical distance pole-selection method is implemented, but it may be changed to any of the other methods. Also, the function `tutte2d` is used to embed the graph of the "open" mesh in the θ–ϕ rectangle. The function returns the θ and ϕ coordinates of every vertex.

```
function [theta,phi] = methodthetaphi(VL,FL,rho);
N = size(VL,1);

% Select poles and cut path
[north,south] = tppolesphysical(VL);
P = tpcutpath(VL,FL,north,south);

% Cut mesh open
FC = tpfacestochange(FL,P);
```

```
[oVL,oFL] = tpcutmeshopen(VL,FL,P,FC);

% Embed open mesh in theta-phi rectangle
k = length(P) - 2;
B = [P', N+1:N+k];
h = pi/(k+1);
U = [pi 0; zeros(k,1) [1:k]'*h; pi pi; zeros(k,1)+2*pi [k:-1:1]'*h];
[phi,theta] = tutte2d(oVL,oFL,B,U,rho,0);
draw2dembedding(oFL,B,phi,theta);
```

The function `tpdrawing` may be used to draw the TP-drawing that results from the θ–ϕ method. The vectors `theta` and `phi`, obtained by `methodthetaphi`, must be supplied. The function `thetaphiline` draws the straight line segment between points $(\texttt{t1}, \texttt{p1})$ and $(\texttt{t2}, \texttt{p2})$, mapped with m (as defined in (6.35)), on the surface of the sphere.

```
function thetaphiline(t1,p1,t2,p2);
lambda = 0:0.05:1;
g = lambda'*[t1 p1] + (1-lambda)'*[t2 p2];
t = g(:,1); p = g(:,2);
x = sin(t).*cos(p); y = sin(t).*sin(p); z = cos(t);
plot3(x,y,z,'k');


function tpdrawing(theta,phi,FL,north,south);
EL = edgelist(FL); e = size(EL,1);
figure, hold on
for j = 1:e,
    edge = EL(j,:);
    t1 = theta(edge(1)); p1 = phi(edge(1));
    t2 = theta(edge(2)); p2 = phi(edge(2));
    if edge(1) == north | edge(1) == south, p1 = p2; end
    if edge(2) == north | edge(2) == south, p2 = p1; end
    thetaphiline(t1,p1,t2,p2);
end
[x,y,z] = sphere(30); r = 0.999;
surf(x,y,z,'EdgeColor','none','FaceColor',[0.804 0.894 0.824]);
view(129,18)
axis image, axis tight, axis off, axis vis3d
```

The function `tp2gc` attempts to obtain a GC-embedding from the θ–ϕ method (see section 6.7) by drawing the edges as minor arcs on the sphere between endpoints. It should be noted that this may not always result in a valid GC-embedding.

```
function V = tp2gc(theta,phi,FL)
V = [sin(theta).*cos(phi), sin(theta).*sin(phi), cos(theta)];
gcdrawing(V,FL); view(129,18)
```

Bibliography

[1] Alexa, Marc, *Merging polyhedral shapes with scattered features*, The Visual Computer, Vol. 16, No. 1, pp. 26–37, 2000.

[2] Anton, Howard, *Elementary Linear Algebra (7th ed)*, John Wiley & Sons, Inc., New York, 1994.

[3] Brechbühler, C., G. Gerig & O. Kübler, *Parameterization of closed surfaces for 3-D shape description*, Computer Vision and Image Understanding, Vol. 61, No. 2, pp. 154–170, 1995.

[4] Colin de Verdière, Éric, Michel Pocchiola & Gert Vegter, *Tutte's barycenter method applied to isotopies*, Computational Geometry: Theory and Applications, Vol. 26, Issue 1, pp. 81–97, 2003.

[5] Cromwell, Peter R., *Polyhedra*, Cambridge University Press, Cambridge, 1997.

[6] De Berg, M., M. van Kreveld, M. Overmars & O. Schwarzkopf, *Computational Geometry, Algorithms and Applications*, Springer-Verlag, Berlin, 1997.

[7] Dijkstra, E.W., *A note on two problems in connection with graphs*, Numeriske Math, Vol. 1, pp. 269–271, 1959.

[8] Fáry, I., *On straight line representations of planar graphs*, Acta scientiarum mathematicarum (Szeged), Vol. 11, pp. 229–233, 1948.

[9] Floater, Michael S., *Parameterization and smooth approximation of surface triangulations*, Computer Aided Geometric Design, Vol. 14, pp. 231–250, 1997.

[10] Floater, Michael S., *Mean value coordinates*, Computer Aided Geometric Design, Vol. 20, pp. 19–27, 2003.

[11] Gerschgorin, S., *Über die Abgrenzung der Eigenwerte einer Matrix*, Izv. Akad. Nauk. USSR Otd. Fiz.-Mat. Nauk 7, pp. 749–754, 1931.

[12] Golub, Gene H. & Charles F. van Loan, *Matrix Computations (3rd ed)*, Johns Hopkins University Press, Baltimore, 1996.

[13] Goodrich, Michael T. & Robert Tamassia, *Data Structures and Algorithms in Java (2nd ed)*, John Wiley & Sons, Inc., New York, 2001.

[14] Gotsman, Craig & Vitaly Surazhsky, *Guaranteed intersection-free polygon morphing*, Computers & Graphics, Vol. 25, pp. 67–75, 2001.

[15] Gotsman, Craig, Stefan Gumhold & Leif Kobbelt, *Simplification and compression of 3D meshes*, in Tutorials on Multiresolution in Geometric Modelling, pp. 319–361, Springer-Verlag, Heidelberg, 2002.

[16] Gotsman, Craig, Xianfeng Gu & Alla Sheffer, *Fundamentals of spherical parameterization for 3D meshes*, ACM Transactions on Graphics, Vol. 22, No. 3, p. 358–363, 2003.

[17] Haker, Steven, Sigurd Angenent, Allen Tannenbaum, Ron Kikinis, Guillermo Sapiro & Michael Halle, *Conformal surface parameterization for texture mapping*, IEEE Transactions on Visualization and Computer Graphics, Vol. 6, No. 2, 2000.

[18] Hill, F.S. Jr., *Computer Graphics Using OpenGL (2nd ed)*, Prentice Hall, Inc., New Jersey, 2001.

[19] Hoppe, Hugues, *Progressive meshes*, ACM SIGGRAPH, pp. 99–108, 1996.

[20] Hoppe, Hugues, *Efficient implementation of progressive meshes*, Computers & Graphics, Vol. 22, No. 1, pp. 27–36, 1998.

[21] Hormann, K., U. Labsik, G. Greiner, *Remeshing triangulated surfaces with optimal parameterizations*, Computer-Aided Design, Vol. 33, pp. 779–788, 2001.

[22] Kazhdan, Michael, Thomas Funkhouser & Szymon Rusinkiewicz, *Rotation invariant spherical harmonic representation of 3D shape descriptors*, Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing, pp. 156–164, 2003.

[23] Kobbelt, Leif P., Jens Vorsatz, Ulf Labsik & Hans-Peter Seidel, *A shrink wrapping approach to remeshing polygonal surfaces*, EUROGRAPHICS '99, Vol. 18, No. 3, pp. 119 –130, 1999.

[24] Lindstrom, Peter & Greg Turk, *Fast and efficient polygonal simplification*, Proceedings of the Conference on Visualization, North Carolina, pp. 279–286, 1998.

[25] McKay, Brendan, *Sequence A049337*, in The On-Line Encyclopedia of Integer Sequences, `http://www.research.att.com/~njas/sequences/`.

[26] Mohlenkamp, Martin J., *A fast transform for spherical harmonics*, The Journal of Fourier Analysis and Applications, Vol. 5, pp. 159–184, 1999.

[27] Praun, Emil, Adam Finkelstein & Hugues Hoppe, *Lapped textures*, ACM SIGGRAPH Proceedings, pp. 465–470, 2000.

[28] Saba, Shadi, Irad Yavneh, Craig Gotsman & Alla Sheffer, *Practical spherical embedding of manifold triangle meshes*, Proceedings on Shape Modelling International (SMI), 2005 (TO APPEAR).

[29] Shapiro, Avner & Ayellet Tal, *Polyhedron realization for shape transformation*, The Visual Computer, Vol. 14, No. 8/9, pp. 429–444, 1998.

[30] Spiegel, Murray R. & John Liu, *Mathematical Handbook of Formulas and Tables (2nd ed)*, McGraw-Hill, New York, 1999.

[31] Steinitz, E. & H. Rademacher, *Vorlesung über die Theorie der Polyeder*, Springer, Berlin, 1934.

[32] Tutte, W.T., *Convex representations of graphs*, Proceedings London Mathematics Society, Vol. 10, pp. 304–320, 1960.

[33] Tutte, W.T., *How to draw a graph*, Proceedings London Mathematics Society, Vol. 13, No. 3, pp. 743–768, 1963.

[34] Zhang, Eugene, Konstantin Mischaikow & Greg Turk, *Feature-based surface parameterization and texture mapping*, ACM Transactions on Graphics, Vol. 24, Issue 1, pp. 1–27, 2005.

[35] Zhou, Kun, Hujun Bao & Jiaoying Shi, *3D surface filtering using spherical harmonics*, Computer-Aided Design, Vol. 36, pp. 363–375, 2004.

[36] Ziegler, Günter M., *Lectures on Polytopes*, Springer-Verlag, New York, 1995.

Index