The Polyhedral Gauss Map and discrete curvature measures in geometric modelling

Gilberto Echeverria

A thesis submitted in partial fulfilment of the requirements of
Sheffield Hallam University
for the degree of Doctor of Philosophy

05 July 2007

Sheffield Hallam University

Materials & Engineering Research Institute

The undersigned hereby certify that they have read and recommend to the Faculty of Arts, Computing, Engineering and Sciences for acceptance a thesis entitled **"The Polyhedral Gauss Map and discrete curvature measures in geometric modelling"** by **Gilberto Echeverria** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Dated: 05 July 2007

Research Supervisors: ______________________
Professor Marcos Rodrigues

____
Dr Lyuba Alboul

Examining Committee: ______________________
Professor Ruud Van Damme (University of Twente)

____
Professor Keith Berrington (MERI, SHU)

To my best friend, Maria del Rocío Leon, for all the great moments spent together.

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

The work in this thesis is concentrated on the study of discrete curvature as an important geometric property of objects, useful in describing their shape. The main focus is on the study of the methods to measure the discrete curvature on polyhedral surfaces. The curvatures associated with a polyhedral surface are concentrated around its vertices and along its edges. An existing method to evaluate the curvature at a vertex is the Angle Deficit, which also characterises vertices into flat, convex or saddle. In discrete surfaces other kinds of vertices are possible which this method cannot identify. The concept of Total Absolute Curvature (TAC) has been established to overcome this limitation, as a measure of curvature independent of the orientation of local geometry. However no correct implementation of the TAC exists for polyhedral surfaces, besides very simple cases.

For two–dimensional discrete surfaces in space, represented as polygonal meshes, the TAC is measured by means of the Polyhedral Gauss Map (PGM) of vertices. This is a representation of the curvature of a vertex as an area on the surface of a sphere. Positive and negative components of the curvature of a vertex are distinguished as spherical polygons on the PGM. Core contributions of this thesis are the methods to identify these polygons and give a sign to them. The PGM provides a correct characterisation of vertices of any type, from basic convex and saddle types to complex mixed vertices, which have both positive and negative curvature in them.

Another contribution is a visualisation program developed to show the PGM using 3D computer graphics. This program helps in the understanding and analysis of the results provided by the numerical computations of curvature. It also provides interactive tools to show the detailed information about the curvature of vertices.

Finally a polygon simplification application is used to compare the curvature measures provided by the Angle Deficit and PGM methods. Various sample meshes are decimated using both methods and the simplified results compared with the original meshes. These experiments show how the TAC can be used to more effectively preserve the shape of an object. Several other applications that benefit in a similar way with the use of the TAC as a curvature measure are also proposed.

Acknowledgements

I would like to thank my supervisor, Dr. Lyuba Alboul, for her patience and guidance in a field of knowledge which was new to me. I would also like to give thanks to my director of studies, Professor Marcos Rodrigues, for providing me with the opportunity to undertake this research program. Also would like to acknowledge all other members of the research team: Jacques Penders, Georgios Chliveros, Alan Robinson and Willie Brink, who shared their time and knowledge with me, and provided useful advice.

I want to thank my parents, whose love and devotion has allowed me to come this far. I owe everything to them. I would also like to mention my friends in Sheffield: Celine Guennebaud, Mentxu Aiertza, Orlando Velecela, Rodolfo Madrid, Enrique Muñoz, Hector Marín, Christian López and many others whom it would be impossible to list here. I am grateful to them for all the very enjoyable moments spent together during this time. Finally, very special thanks to Maria del Rocío Leon who constantly provided me with support and motivation to carry on working during all of my time in Sheffield.

List of Publications

- Echeverria, G. and L. Alboul (2007). Shape–preserving mesh simplification based on curvature measures from the Polyhedral Gauss Map. *International Journal on Computational Vision and Biomechanics. Received:* 26/02/2007. *Accepted:* 17/05/2007.

- Echeverria, G. and L. Alboul (2006, October 20–21). Decimation and smoothing of triangular meshes based on curvature from the polyhedral gauss map. In *Proc. Of CompIMAGE Conference. Coimbra, Portugal*, pp. 175–180.

- Echeverria, G., L. Alboul, and M. Rodrigues (2005, November 24–25). Enhancing game physics using gauss map computation. In *Proc. Of Game-On Conference. Leicester, UK*, pp. 47–51.

- Alboul, L. and G. Echeverria (2005, September 5–7). Polyhedral Gauss maps and curvature characterisation of triangle meshes. In R. R. Martin, H. E. Bez, and M. A. Sabin (Eds.), *Proc. of Mathematics of Surfaces XI, 11th IMA International Conference, Loughborough, UK*, Volume 3604 of *Lecture Notes in Computer Science*, pp. 14–33. Springer.

- Alboul, L., G. Echeverria, and M. Rodrigues (2005, March 9–11). Discrete curvatures and gauss map for polyhedral surfaces. In *Proc. 21st European Workshop in Computational Geometry, Eindhoven, Netherlands*, pp. 69–72.

- Alboul, L., G. Echeverria, and M. A. Rodrigues (2004, June/July). Total absolute curvature as a tool for modelling curves and surfaces. In *Proc. of Geomgrid, Moscow, Russia*, pp. 220–231.

- Alboul, L., G. Echeverria, and M. A. Rodrigues (2004, March 24–26). Curvature criteria to fit curves to discrete data. In *Proc. 20th European Workshop in Computational Geometry, Sevilla, Spain.*

Chapter 1

Introduction and problem statement

The shape of an object is an important geometrical property to differentiate it from other objects. The human eye is trained to recognise shapes, but for some computational applications it is necessary to give a numerical value to the shape of objects. The *curvature* is a measure that describes the shape of curves and surfaces. Loosely speaking, it describes how a curve differs from a straight line and how a surface in space differs from a plane. If a curve is nearly straight, its curvature will be close to zero, while if it has a pronounced bending, the curvature will be larger. In the most general terms, curvature can be seen as a function of the angles between tangent lines to a curve at different points.

The theory of curvature for smooth surfaces is already well established, in the realm of differential geometry. It involves definitions of curvature based on the 1^{st} and 2^{nd} derivatives of the underlying surface, which can be represented in a parametric or functional form. This requires a certain degree of continuity (at least C^2) of the surfaces. However, many natural shapes are non–regular, and in particular the computer representations of objects very often lack the required smoothness.

Various methods to extract the curvature of discrete surfaces have been in development in recent times. The initial attempts tried to approximate the discrete data by a smooth surface, and then use partial differential equations to extract geometrical properties. This approach, however, is complex and requires conversions between smooth and discrete input and output. Methods that apply directly to discrete surfaces are needed. In the last decades several attempts have been made to develop discrete methods that would be capable of dealing with non–regular objects. Among these methods are adaptations of the concepts of the theory of polyhedral metrics and non–regular surfaces (Aleksandrov and Zahlgaller 1967), (Banchoff 1970), (Burago 1970), (Brehm and Kühnel 1982) and (Aleksandrov and Reshetnyak 1989). A new method to evaluate (discrete) Mean curvature has been proposed in (Bobenko 2005).

This thesis concentrates on the study of the discrete analogue of *Total Absolute Curvature*. It deals with the theoretical background and computational algorithms to compute and evaluate the curvature of discrete objects. Based on the concept of *orientation* a sign can be given to the curvature at every point. However, a drawback of the signed curvatures is that positive and negative curvatures will cancel each other out, thus hiding important features of the object. Under this consideration all closed surfaces of the same genus possess the same global curvature measure, regardless of their geometric complexity (*i.e.* how are they embedded in space).

The concept of Total Absolute Curvature refers to the consideration of the changes in the turn of a curve at a point or in the walk around a point on a surface, which can be expressed by using the concept of orientation. This concept has been studied before in (Brehm and Kühnel 1982) and (Kuiper 1970). The absolute value of the angles provides a more complete representation of the underlying geometry. The Total Absolute Curvature of a curve is a global property, which can describe how non–convex an object is. It is an important fact that it reaches its minimum value with convex curves, while non–convex curves will have larger curvature values.

In computer science, objects are represented as discrete approximations of real surfaces. These are obtained from sampled data that represent the real object and only consist of a reduced and limited amount of interconnected points (known as *vertices*). While vertices have a direct correspondence to points on the actual surface, the structures that link them are simplified approximations of the shape. Discrete approximations of curves are composed of vertices and the *edges* that connect them. Closed curves in the plane are analogous to planar polygons. Surfaces are represented in discrete form as polygonal meshes, consisting of vertices, edges, and *faces* that make up the surface delimited between edges and vertices. These discrete representations do not possess the same degree of continuity as their smooth counterparts. This affects how the curvature can be measured, and demands the implementation of analogous methods that provide equivalent results to those for smooth objects.

The concept of curvature for non–regular curves and surfaces has been established in (van Rooij 1965) and (Aleksandrov and Zahlgaller 1967). With the prevalence of computers, the study of geometry for discrete surfaces has gained importance. Research on discrete curvatures is of growing interest in geometric modelling (an overview is given in (Alboul 2003)).

However there are particular considerations to make during the transition from smooth to discrete methods. In the smooth case the curvature measures create a characterisation of the surface into two types, according to the sign of the local curvature, either positive or negative. The regions of positive and negative curvature are clearly identified and never overlap. In contrast, discrete surfaces have features which are not found in their smooth counterparts, for instance, vertices that incorporate both positive and negative curvature. We will refer to such vertices as *mixed vertices*. Because of these cases, methods that are direct analogues of those for smooth surfaces are inadequate in dealing with all kinds of polyhedral surfaces. This is the case with the Angle Deficit method for measuring the analogue of the (integral) Gaussian curvature around a vertex. It can provide a characterisation of the vertices by positive or negative curvature, in accordance with smooth surfaces. But to correctly characterise different vertex types it is necessary to develop methods aimed directly at discrete surfaces. This will present a wider array of vertex types, all of which will be analysable.

The research question that arises is how to recognise these vertices, and to obtain reliable computations of their curvature. This can be presented as our main research topic:

- To find a procedure that correctly measures the curvature for the different types of vertices in a discrete polyhedral surface.

In order to fulfil this task we propose a discrete version of the Gauss Map, called the *Polyhedral Gauss Map*. We study the structure of the Polyhedral Gauss Map in all detail, as an extension of the method of Total Absolute Curvature. This will permit us to measure curvature and characterise the geometry of vertices of any type, including complex mixed vertices, vertices with self-intersections in their neighbourhoods and some kinds of degenerate vertices. The proposed method considers the connectivity of the neighbourhood of the vertex. It measures not only positive and negative curvature parts incorporated at the vertex, but also separates these parts in atomic subparts that provide a complete characterisation of the complex geometry around the vertex. A polyhedral Gauss map has been used by some authors to underline the curvature of a vertex (Kilian 2004), (Maltret and Daniel 2002), but its application has been limited to simple cases, and without the detailed study of its structure.

The vertex characterisation is also important in determining a *region segmentation* of a surface, where vertices of the same curvature sign are grouped together in representative regions. This is used in various applications to identify the different parts of an object, such as the nose in a human face or the wings of an aeroplane. The considerations previously presented for the Total Absolute Curvature allow a more complete region segmentation that includes the areas where positive and negative curvature are both present.

The research questions investigated in this thesis can be extracted from the previous discussions as:

1. To what extent is a discrete curve or surface characterised by its Total Absolute Curvature?
2. How can the Total Absolute Curvature of a discrete surface be measured with a discrete version of the Gauss Map?
   This problem can be subdivided as follows:
   (a) How to recognise all the curvature components of a vertex.
   (b) How can the correct curvature measure for all possible vertices, including non–manifold ones, be obtained?
   (c) Can the curvature measures of individual vertices provide a classification for regions of the mesh?
3. How can practical applications benefit from the additional measurement accuracy provided by Total Absolute Curvature? We explore the following applications as test environments:
   (a) Curve reconstruction (Amenta *et al.* 1998), (Althaus and Mehlhorn 2000)
   (b) Surface characterisation (Li and Gu 2004)
   (c) Face recognition (Grodon 1991), (Tanaka *et al.* 1998)
   (d) Terrain navigation (Falcidieno and Spagnuolo 1991), (Lee *et al.* 2001)
   (e) Decimation: (Schroeder *et al.* 1992), (Kim *et al.* 2002)

The research presented in this thesis is considered from a Computer Science point of view, while presenting all necessary mathematical background. Most proofs given are of pure geometric nature. Visualisation and graphical illustration of results represent an important part of the thesis. The computer and advanced computer graphics are central components in this work.

1.1 Aims of the research project

From a Computer Science point of view, the analysis of curvature presents various new challenges: complex objects are composed of thousands or even millions of vertices, each of which has to be analysed individually. The practical aim of the project is to obtain algorithms that can perform curvature measures of polyhedral surfaces, based on simple geometrical properties and calculations. The term polyhedral surface is used in a generalised sense, which includes self–intersecting and non–manifold surfaces. The curvature of any kind of vertex should be correctly evaluated.

A second aim for the project is to develop computer graphic techniques into a tool to visualise an abstract concept, such as curvature, in order to better understand its behaviour. The elements that affect the curvature of a vertex are difficult to imagine, or to draw in the plane. A three–dimensional visualisation permits a proper understanding of the properties of the vertices and surfaces. The visualisation can reflect immediate changes in the object and its curvature and provides interactive tools to aid in analysing the geometric properties.

We first study the concept of curvature on closed planar polygonal curves (polygons) and later on polyhedral surfaces. Initial analysis of curvature is done on curves lying in the plane. This research provides insight on how the shape and the curvature are related, and outline the properties of the Total Absolute Curvature. Analysis of planar polygons also provides a starting point for the concept of the Gauss Map, as a graphical representation of the measure of Total Absolute Curvature.

The research on planar curves also provides a good starting point for studying the properties of curvature in the case of polyhedral surfaces in three dimensions. However in this case the ordering and linking of the data are not unique, even for the simplest cases. While for a planar polygon the segments that join the vertices can only have one orientation and direction, this is not true for faces in 3D. This generates several more cases to consider when stepping up to a higher dimension. Some of the curvature properties observed on planar polygons can be immediately translated to three–dimensional surfaces. Others require a certain extension to function properly. The work on this thesis will explain the relationship between the two cases.

The main part of the research deals with the computation of the *Polyhedral Gauss Map* of vertices in a polyhedral surface. The Gauss Map of surfaces is well known, and has received some attention in the past (Banchoff *et al.* 1982), (Rodríguez and Rosenberg 2000) and (Grinspun and Schröder 2001), however it has never been studied extensively for all cases of polyhedral vertices. The curvature of a polygonal mesh is obtained from the spherical representation of the features of each individual vertex. This spherical representation is the Gauss Map, and is based on the normal vectors of the faces around each vertex. However, previous attempts at finding the Gauss Map of vertices have been limited to the simplest configurations available. Other more complex vertices are generally ignored or considered impossible to analyse (Lowekamp *et al.* 2002), (Yamauchi *et al.* 2005). The methods presented in this research seek to obtain an accurate measurement of curvature for all possible kinds of polyhedral surfaces, including non–convex and non–manifold surfaces.

An innovation of the Polyhedral Gauss Map method is that it allows the 'separation' of positive and negative curvature 'parts' incorporated at a polyhedral vertex of a complex geometry, and thus provides a complete characterisation of the shape of a vertex. Positive and negative curvature 'parts' are represented as spherical polygons on the surface of the sphere. The difficulty lies in separating the spherical polygons and determining their corresponding sign. The signed curvatures allow for clear identification of vertices that would not be valid according to other methods, or that would not be possible to correctly distinguish. Also, because of the use of the novel methods, more features of a vertex can be recognised.

The final objective is to indicate practical applications where the curvature information is useful to identify, classify or modify the polyhedral representation of an object. The computations required for the Polyhedral Gauss Map are relatively simple and fast, allowing for the method to be used in a variety of applications. Some examples are feature recognition, face recognition, computer vision, terrain navigation, mesh optimisation, and others.

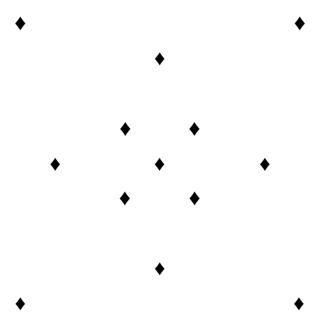
We implement an application on mesh simplification using the Polyhedral Gauss Map to measure curvatures and assign an importance weight to vertices. Those with a small weight are removed from the polygonal mesh, in a way that will preserve the original shape of the object. This experiment provides a numerical proof that the Polyhedral Gauss Map method provides better curvature measures with respect to the Angle Deficit method.

The main research objectives of this thesis can be summarised as follows:

- Study the properties of the Total Absolute Curvature on planar polygons
- Obtain accurate measurements of the curvature of vertices in a polygonal mesh
- Characterise the vertices according to their curvature signs
- Visualise the Polyhedral Gauss Map of vertices using computer graphics
- Apply the curvature measures and characterisation for practical applications and further processing of the data, such as optimisation, simplification or subdivision

1.2 Experimental methodology

The experiments presented in this research were performed using two main programs, developed to test the theoretical principles and to have a visual point of reference for the results. The applications were created using the *C programming language* and the *OpenGL* libraries for graphical display. The first such program is designed to fit curves over point clouds on the plane, and is called `reconstructor`. Its purpose is to study how curvature could be used to determine the usefulness of curve-fitting algorithms. An example of the results produced by this program is shown in Figure 1.1. Several curves are generated from each dataset by minimisation of various parameters. The curvature of the generated curves is used at later stages to evaluate the results.



(a) Point cloud **(b) Reconstructed curve**

Figure 1.1: Example of curve-fitting using the `reconstructor` **program.**

The second program is focused on finding the curvature of vertices in a polyhedral mesh by means of the Polyhedral Gauss Map and is called `gaussMap`. Figure 1.2 exhibits a screenshot of this program showing the Polyhedral Gauss Map of a vertex in a simple mesh. It incorporates the concepts of Total Absolute Curvature to obtain correct measurements for every vertex and for whole surfaces.

The `gaussMap` program is later extended to perform mesh simplification, using the curvature measures to select vertices to be removed from the mesh. This second version of the program is called `decimator`, and retains the same basic functions of `gaussMap`. Figure 1.3 shows the results of this implementation of polygon simplification.

Various weighted combinations of Total Absolute Curvature and areas are used to measure the importance of vertices to the shape of the object. The decimated meshes

Figure 1.2: Example of Polyhedral Gauss Map of a vertex from the `gaus sMap` **program.**

(a) Original (16,944 vertices)

(b) 97.1 % decimation (494 vertices)

Figure 1.3: **Example of polygon simplification with the** `decimator` **program.**

obtained with various parameters are compared using the `Metro` program (Cignoni *et al.* 1998), to evaluate their usefulness and quality.

Additional programs developed include scripts in *Perl* that generate source data for testing, and the `objViewer` program, created to visualise polyhedral meshes stored in the *OBJ* format. This file format is also employed by `gaussMap` and `decimator` programs.

1.3 Main research contributions

In general terms, the most important contribution of this research consists of providing a pure geometric background and developing computer algorithms to measure and visualise the curvature of discrete objects. These new methods are simple to compute and do not require the evaluation of computationally expensive differential geometry procedures.

A list of contributions is presented here and all of them will be explained extensively in later chapters.

- Computation of the Polyhedral Gauss Map for vertices in a polyhedral mesh

  - Algorithms to detect the intersection of arcs on the surface of a sphere
  - Algorithm to split a series of ordered arcs on a sphere into non-intersecting spherical polygons
  - Methods to determine the orientation of a spherical polygon related to a vertex
  - Identification of positive and negative components of the curvature of a vertex
  - Classification of vertices into the following types: flat, convex, saddle and mixed
  - Computation of complete curvature for any kind of vertex, including non–convex and non–manifold vertices

- Visual display of the Gauss Map using OpenGL
  - Methods to draw the spherical indicatrix of vertices
  - Algorithm to draw spherical triangles in OpenGL using clipping planes
  - Algorithm to triangulate spherical polygons for display in OpenGL

- Use of Total Absolute Curvature for polygon simplification
  - Proposal of new vertex weight parameters, based on curvature and areas, to determine the importance of a vertex
  - Use of Total Absolute Curvature to optimise local triangulations by edge flipping

- Algorithms for curve reconstruction from a disorganised point cloud in the plane
  - Algorithm to construct curves with one 'deviation' from the convex hull
  - Algorithm to construct curves with multiple 'deviations' from the convex hull
  - Evaluation of multiple minimisation criteria for vertex insertion in reconstructed curves. Criteria based on geometric parameters, such as local or global curvature, distances and combinations of them
  - Methods to detect and correct self–intersecting polygonal curves

1.4 Organisation of the thesis

The following is the description of the contents of each chapter. Chapter 2 is an introduction to the main concepts of Total Absolute Curvature. The main contributions of this thesis are included in Chapter 3, Chapter 4 and Chapter 5. A relevant literature review is given at the beginning of each corresponding chapter.

The structure of the thesis is the following:

Chapter 2 presents some basic geometrical concepts and theoretical background used during the thesis. Afterwards it introduces the concept of Total Absolute Curvature for polygonal curves in the plane, demonstrates its validity and its relation with the spherical image of a curve, as a precursor of the Gauss Map.

Chapter 3 presents the curvature measures for smooth surfaces, including the Gauss Map, and explains analogous methods for discrete surfaces. It presents a description of the Total Absolute Curvature of vertices, and the characterisation of vertices into four basic types. The method to compute the Polyhedral Gauss Map is introduced, and its validity in evaluating the Total Absolute Curvature is demonstrated. Finally this new method is compared with the Angle Deficit method.

Chapter 4 presents the computational algorithms developed to generate the Polyhedral Gauss Map of a vertex. It describes the process of constructing the spherical polygons and extracting the curvature information from them, including all the special cases and particular considerations that need to be observed. The chapter presents the results of the algorithm on various types of vertices, and a comparison of the characterisation obtained with this and the Angle Deficit method.

Chapter 5 describes the techniques used to display the Polyhedral Gauss Map in 3D using OpenGL.

Chapter 6 shows a practical application of the Polyhedral Gauss Map used in polygon simplification. A vertex decimation program is used as a test platform to produce simplified meshes with various curvature parameters. The results are compared to determine the effectiveness of Total Absolute Curvature to guide simplification algorithms.

Chapter 7 contains the conclusions and further research directions.

Finally, in Appendix A we present additional experiments on the measures of Total Absolute Curvature for planar polygons. This section deals with a problem that is loosely related to the main research. It describes in detail an application developed for curve–fitting, which incorporates two different algorithms. The curves generated by the minimisation of various parameters are evaluated and compared by means of their Total Absolute Curvature.

Chapter 2

Geometric concepts and discrete curvature in two dimensions

2.1 Introduction

Curves are fundamental objects in geometry, that can be used to build more complex objects of any shape. In this chapter we consider curves that lie only in the plane $\mathbb{R}^2$. The concept of *curvature* is a common geometric property of curves that describes how a line in the plane bends at any given point. Research about the properties of curvature has been very extensive, especially in differential geometry (van Rooij 1965), (Kuiper 1970), (Ujiie and Matsuoka 2003), (Sullivan 2006). For smooth curves, the curvature can be very accurately represented by the rate of change of the angle between tangent lines at different locations of a curve. Obtaining the curvature of a single point p of the curve requires the tangent lines at two other points at either side of p. Making the distance between these points and p as small as possible we get a close approximation of the curvature at p. Thus the curvature is obtained as a limit, involving the second order derivatives of the curve.

An additional property associated to curvature is the direction of the change in the tangents, used to give a sign to the curvature at any given point, classifying it as positive or negative. This direction is referred to as the *orientation* of the curve at a vertex.

We focus on the representation of curves most commonly used in computer science, as a *piecewise linear approximation* composed of vertices and edges. We refer to this representation as a *discrete curve*. *Vertices* are points sampled from a smooth curve or other data source, and are defined by their coordinates. The sampling frequency of these points depends on the origin of the information, which can provide vertices at regular intervals or at important feature points of the original object. *Edges* are straight line segments that connect the vertices in a certain order, and provide the shape of the information represented by the isolated points.

The concepts of curvature can be translated from the smooth to the discrete case with a few considerations. In the case of a discrete curve the changes in curvature occur exclusively at the vertices, since all the points of a single edge have the same curvature, equal to zero. We can determine that, for a polygonal curve, the curvature at a vertex is measured by the angle between the two edges incident on it. Integral curvature in the case of a smooth curve can be expressed as a *turn*, *i.e.* the angle between tangent lines at two points on the curve. Whereas in the case of a polygonal curve is just a turn of the curve, manifested by the angle at a vertex (Aleksandrov and Reshetnyak 1989).

In this research the study of the two–dimensional version of curvature provides useful insights for research on triangulations in three–dimensional space: the requirements for computing curvature, its relationship with the shape of the object, and identification of the special cases that require more particular treatments. Note also that the study of polygonal curves with respect to their total absolute curvature is also interesting for its own sake, as the total absolute curvature is related to the global properties of the curve and might be used to characterise diverse curve profiles (Ujiie and Matsuoka 2003).

This chapter begins by presenting the basic theoretical concepts, (mainly of geometrical nature) to be used in the consequent chapters. Secondly we deal with the analysis of curvature properties and derive the discrete method to measure the curvature of a polygon. Finally the concepts described are explained in the context of curve reconstruction.

2.2 Basic geometric concepts and computational implementations

This section describes several basic geometric concepts and their implementation in the computer programs developed. We will refer to a piecewise linear approximation of a smooth curve as a *discrete curve*, that is composed of points or *vertices* and line segments or *edges* connecting vertices together. Let a vertex be a tuple $\nu = (x, y) \in \mathbb{R}^2$. Given a collection of vertices $\mathbf{V} = \{\nu_i; i = 1, 2, \ldots, n\}$, we define a curve $\mathbf{P}(\mathbf{V})$ as a planar polygon that connects all vertices $\nu_i \in \mathbf{V}$ in a specific order. An edge can be represented as a line segment connecting two vertices, for example the edge between vertices ν_1 and ν_2 is denoted as $\overline{\nu_1\nu_2}$. The two vertices are known as the *endpoints* of the edge.

We will additionally provide the definitions for operations on vertices and vectors in three dimensions, where a vertex is denoted with $\nu = (x, y, z) \in \mathbb{R}^3$. The vector operations can also be carried out for vertices in the plane, by setting the third coordinate equal to zero, in order to maintain the consistency of all results.

Vector from two vertices

Given two vertices $\nu_1 = (x_1, y_1, z_1)$ and $\nu_2 = (x_2, y_2, z_2)$ we can define a vector $\mathbf{v} = (u, v, w)$ with origin at ν_1 and endpoint at ν_2 as:

$$\mathbf{v} = \nu_2 - \nu_1 = (x_2 - x_1, y_2 - y_1, z_2 - z_1). \tag{2.2.1}$$

The magnitude of a vector $\mathbf{v} = (u, v, w)$ is obtained as:

$$|\mathbf{v}| = \sqrt{u^2 + v^2 + w^2}. \tag{2.2.2}$$

A vector is said to be *normalised* if its magnitude is equal to 1. This is also called a *unit vector*. The method to normalise a vector is to divide each of its components by the magnitude of the vector, as such:

$$\overline{\mathbf{v}} = \left(\frac{u}{|\mathbf{v}|}, \frac{v}{|\mathbf{v}|}, \frac{w}{|\mathbf{v}|} \right). \tag{2.2.3}$$

Dot product of two vectors

The dot product is a binary operation defined over two vectors that returns a scalar value. Given two vectors $\mathbf{v_1} = (u_1, v_1, w_1)$ and $\mathbf{v_2} = (u_2, v_2, w_2)$, their dot product is given by:

$$\mathbf{v_1} \cdot \mathbf{v_2} = u_1 u_2 + v_1 v_2 + w_1 w_2. \tag{2.2.4}$$

An alternate definition of the dot product is:

$$\mathbf{v_1} \cdot \mathbf{v_2} = |\mathbf{v_1}||\mathbf{v_2}| \cos\theta, \tag{2.2.5}$$

and from this we can find the angle between the two vectors as:

$$\theta = \arccos\left(\frac{\mathbf{v_1} \cdot \mathbf{v_2}}{|\mathbf{v_1}||\mathbf{v_2}|} \right). \tag{2.2.6}$$

The angle θ is limited to a range between 0 and π radians. It can be used to obtain the relative direction of the vectors with respect to each other.

Cross product of two vectors

The cross product is a binary operation defined over two vectors that returns another vector. The resulting vector will be *normal* to both original vectors, that is, it will be at right angles to both. Given the two vectors $\mathbf{v_1}$ and $\mathbf{v_2}$, their cross product is given by:

$$\mathbf{v_1} \times \mathbf{v_2} = ((v_1 w_2 - w_1 v_2), (w_1 u_2 - u_1 w_2), (u_1 v_2 - v_1 u_2)). \tag{2.2.7}$$

Angle at a vertex

We measure the angle at a single vertex based on the dot product of two vectors. To know the angle θ at vertex ν_i, we need its two neighbour vertices ν_{i-1} and ν_{i+1}. We define two vectors $\mathbf{v_1} = \nu_i - \nu_{i-1}$, and $\mathbf{v_2} = \nu_{i+1} - \nu_i$. Then the cosine of the angle between the vectors is obtained from Equation 2.2.6:

$$\cos\theta = \frac{\mathbf{v_1} \cdot \mathbf{v_2}}{|\mathbf{v_1}||\mathbf{v_2}|}. \tag{2.2.8}$$

The resulting $\cos\theta$ will have a value between -1 and 1, and can be used to determine the relative positions of the points ν_{i-1} and ν_{i+1} with respect to ν_i. If $\cos\theta = 0$ then the vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ are at right angles. If $\cos\theta = 1$ the vectors are collinear and point in the same direction. If $\cos\theta = -1$ the vectors are collinear but point in opposite directions. In general if $\cos\theta < 0$ the vectors point in opposing directions with respect to ν_i.

Distance from a vertex to a line segment

We establish a particular constraint for computing the distance from a vertex to a segment as a special case of measuring the distance from a vertex to a line. Following the notation for the Euclidean distance between two vertices as $d(\nu_1, \nu_2)$, we denote the distance from a vertex ν to a segment $\overline{\nu_i\nu_{i+1}}$ as $d(\nu, \overline{\nu_i\nu_{i+1}})$.

Definition 2.2.1. Given an edge $e = \overline{\nu_i\nu_{i+1}}$, we can sweep e along a line perpendicular to itself. We define the *sweep region* R as the area in 2D space that is covered during the sweep.

Using this definition we can consider two different cases to measure the distance from a point ν to a segment e:

- If ν lies inside of the sweep region R, then the distance is measured as the length of a segment of a line perpendicular to e that passes through the vertex ν. This line will intersect e at a new point ν_x. The distance is measured from ν to ν_x.
- If ν lies outside of R, then the distance is measured from ν to the closest endpoint of e.

In Figure 2.1 vertex ν_1 is inside of the sweep region of the segment $e = \overline{\nu_3\nu_4}$, while vertex ν_2 is not.

For the computational implementation of this distance two vectors are created to identify whether the point ν is within the sweep region R of the segment e. One vector is parallel to e and the second vector goes from any one of the endpoints of e to the point ν.

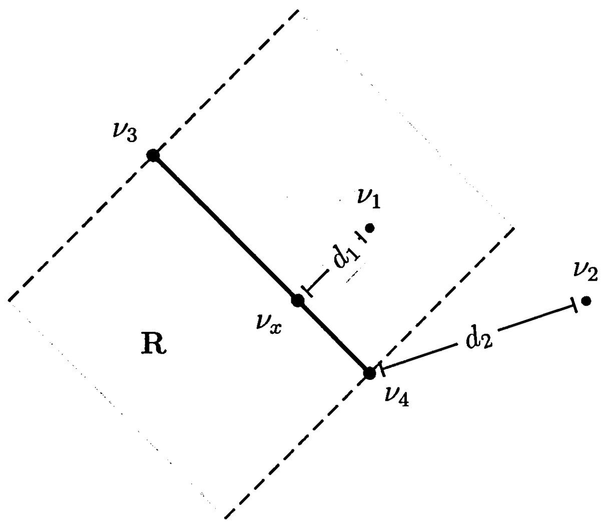


Figure 2.1: Distance from a point to a segment. Sweep region R shown in grey.

This is explained using the vertices in the example of Figure 2.1, the two vectors would be: $\mathbf{v_1} = \nu_4 - \nu_3$, and $\mathbf{v_2} = \nu_1 - \nu_3$. We obtain the dot product of these two vectors as:

$$a = \mathbf{v_1} \cdot \mathbf{v_2}. \tag{2.2.9}$$

If $a < 0$ then ν lies outside of R. If $a = 0$ then ν lies on a line perpendicular to e that passes over the endpoint ν_3. In both of these cases the distance from ν to e is measured as the distance from ν to ν_3.

Next we compute the dot product of the vector $\mathbf{v_1}$ with itself:

$$b = \mathbf{v_1} \cdot \mathbf{v_1}. \tag{2.2.10}$$

If $b < a$ then ν is outside of R. If $b = a$ then ν lies on a line perpendicular to e that passes over the endpoint ν_4. In both of these cases the distance from ν to e is measured as the distance from ν to ν_4.

Otherwise, if $b > a$ then ν is located inside of R in a line perpendicular to e that intersects the segment at a new point ν_x. The coordinates of this new point are obtained by:

$$\nu_x = \nu_3 - \left(\frac{a}{b}\right) \mathbf{v_1}. \tag{2.2.11}$$

The distance from the point to the segment is measured as the Euclidean distance from point ν to the new point ν_x, as shown in Figure 2.1.

Orientation

Any simple closed curve can be assigned an orientation, depending on the direction the curve is traversed. In general, if a walk along the curve keeps the interior of the curve to the left, the orientation is positive, and negative if the interior is to the right. This can also be expressed in terms of the *right hand rule* for a planar graph. Imagine that, using the right hand, one follows the direction of the curve with the index finger, while the middle finger points to the interior of the curve; if the thumb points upwards, the orientation is positive; if the thumb points downwards the orientation is negative.

In a smooth curve, the orientation at any given point is defined as the direction in which the curve turns at that point. According to (Borowski and Borwein 2002), this orientation can be either *Clockwise* (CW), *Counter–Clockwise* (CCW) or *Collinear* (COL). The same definition can be applied in the case of a planar polygon, using the angle formed by the two converging segments at that point. This concept of orientation can also be assigned to a pair of vectors that originate at the same point.

In any planar polygon $\mathbf{P}(\mathbf{V})$, the orientation at point ν_i is denoted with $\rho(\nu_i)$ and it is determined by using the two neighbouring points: ν_{i-1} and ν_{i+1}. The three points have coordinates: $\nu_{i-1} = (x_{i-1}, y_{i-1})$, $\nu_i = (x_i, y_i)$ and $\nu_{i+1} = (x_{i+1}, y_{i+1})$, as shown in Figure 2.2. At vertex ν_i the orientation $\rho(\nu_i)$ is defined as:

$$\rho(\nu_i) = \begin{cases} CW, & \text{if } \ r(\nu_i) < 0, \\ COL, & \text{if } \ r(\nu_i) = 0, \\ CCW, & \text{if } \ r(\nu_i) > 0, \end{cases} \tag{2.2.12}$$

where $r(\nu_i)$ is given by the formula (Crépeau 2004):

$$r(\nu_i) = ((y_i - y_{i-1})(x_{i+1} - x_i)) - ((y_{i+1} - y_i)(x_i - x_{i-1})). \tag{2.2.13}$$

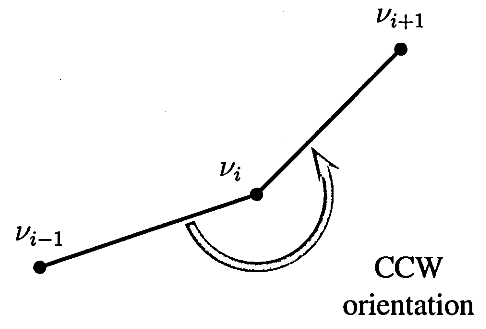


Figure 2.2: Finding the orientation of a vertex.

The whole polygonal curve $\mathbf{P}(\mathbf{V})$ also has an orientation, determined by the individual orientations found for each of the vertices. This is obtained by adding the signed angles at the vertices, where the sign will be determined by the orientation at the corresponding vertex.

For the implementation of the computation of orientation there is an issue that must be considered. The rounding of floating point numbers done by the computer may lead to problems when calculating the orientation of three points, since it may erroneously find that two points are collinear when they are not, or *vice versa*. This problem may severely impair the results, because many operations and comparisons used in the implementation are based on orientation. For this reason, a rounding tolerance is introduced, to accept very small values to be considered equal to zero, and thus allowing the program to have more consistent results, at the expense of precision in the measurements.

Intersection of segments on a plane

The edges of a curve can intersect, and it is necessary for our purposes to identify when these intersections occur. In the current implementation the exact location of the intersection of two segments is of no importance. Orientation can be used to detect segment intersections, by comparing the orientation of the points of one segment with respect to the other, and *vice versa*. Since the vertices of two segments may not be consecutive, we can refer to the orientation of an arbitrary vertex ν_a as $\rho(\nu_b, \nu_a, \nu_c)$, where the orientation is determined by the two segments $\overline{\nu_b\nu_a}$ and $\overline{\nu_a\nu_c}$, in that order. There are two cases to consider:

General Case: Given two segments $\overline{\nu_1\nu_2}$ and $\overline{\nu_3\nu_4}$, they will intersect if

$$\rho(\nu_1, \nu_2, \nu_3) \neq \rho(\nu_1, \nu_2, \nu_4) \tag{2.2.14}$$

and

$$\rho(\nu_3, \nu_4, \nu_1) \neq \rho(\nu_3, \nu_4, \nu_2). \tag{2.2.15}$$

The segments shown in Figure 2.3 intersect each other, while those in Figure 2.4 do not intersect, since the orientations $\rho(\nu_3, \nu_4, \nu_1)$ and $\rho(\nu_3, \nu_4, \nu_2)$ are equal.

Special Case: If the orientation for all the points is collinear, then both segments lie in the same line. To test if there is an intersection of the segments we measure the Euclidean distances from ν_1 to the other vertices ν_2, ν_3 and ν_4. If any of the points ν_3 or ν_4 is closer to ν_1 than ν_2 and in the same direction, then the segments overlap and there is an intersection, as shown in Figure 2.5, where $d(\nu_1, \nu_3) < d(\nu_1, \nu_2)$.

In some situations, it is necessary to compute the intersection of two segments including the endpoints of the segments, while sometimes the intersection at the endpoints is not considered. In these latter cases an extra comparison is done to verify that none of the endpoints of a segment is equal to those on the other segment. There are thus two functions to test for an intersection of two segments, one which includes the endpoints and another which does not.

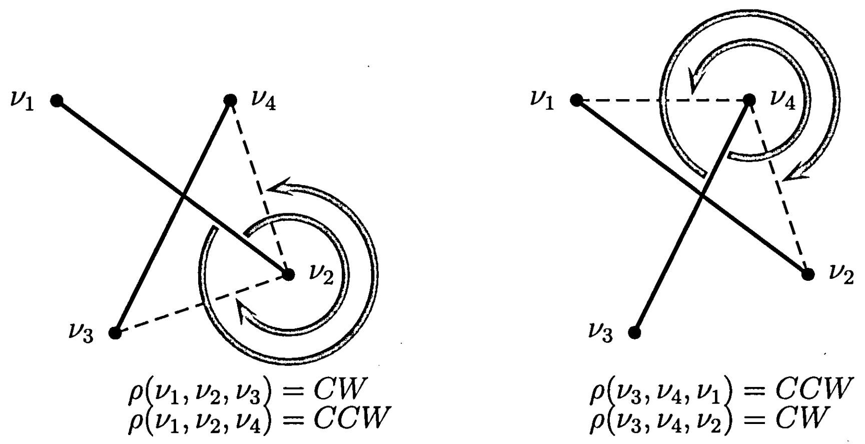

ν_1
ν_4
ν_2
ν_3
$\rho(\nu_1,\nu_2,\nu_3) = CW$
$\rho(\nu_1,\nu_2,\nu_4) = CCW$
ν_1
ν_4
ν_2
ν_3
$\rho(\nu_3,\nu_4,\nu_1) = CCW$
$\rho(\nu_3,\nu_4,\nu_2) = CW$

Figure 2.3: Determining self–intersections based on orientation.

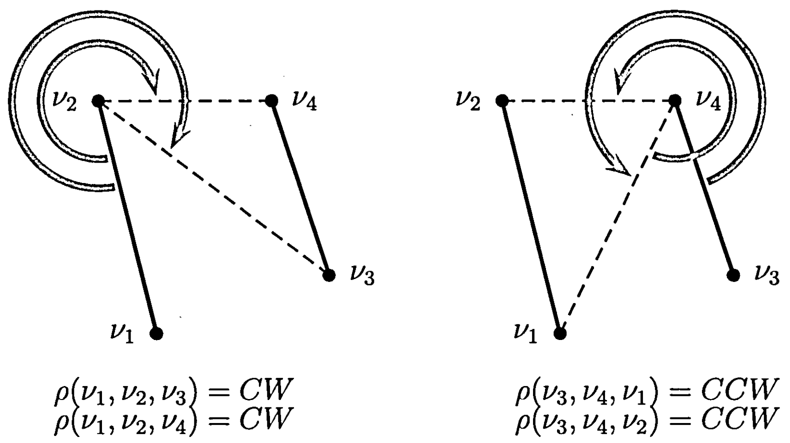

ν_2
ν_4
ν_3
ν_1
$\rho(\nu_1,\nu_2,\nu_3) = CW$
$\rho(\nu_1,\nu_2,\nu_4) = CW$
ν_2
ν_4
ν_3
ν_1
$\rho(\nu_3,\nu_4,\nu_1) = CCW$
$\rho(\nu_3,\nu_4,\nu_2) = CCW$

Figure 2.4: Non–intersecting segments.

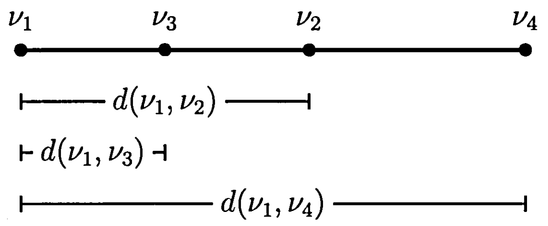


Figure 2.5: Self–intersection of collinear segments.

Convex hull

The *convex hull* of a dataset $\mathbf{V}$ is a curve that joins a certain subset $\mathbf{V}_{\mathrm{CH}}$ of the vertices in $\mathbf{V}$. We call this the *CH–curve*. Its requirement is that any line segment joining two points in the dataset will lie in the interior of the convex hull (de Berg *et al.* 1997) (see

Figure 2.6).

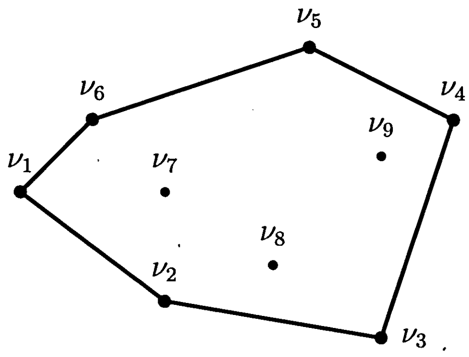
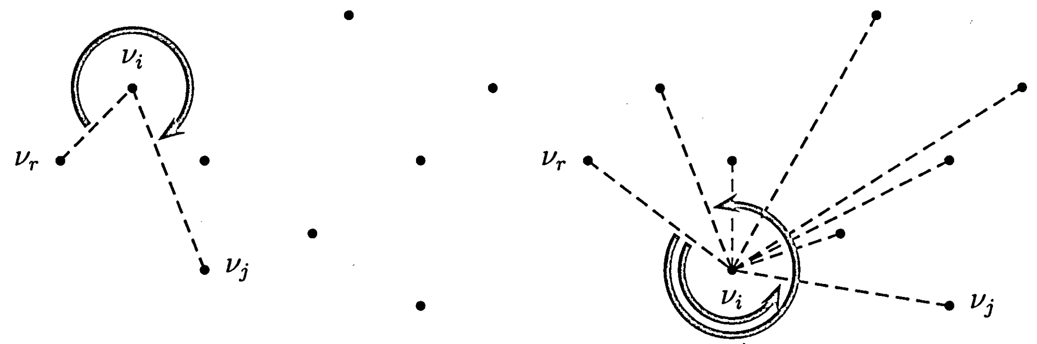


Figure 2.6: Convex hull of a point cloud.

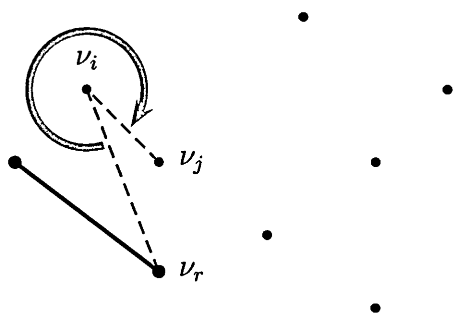
The convex hull of $\mathbf{V}$ can be computed by using the orientation of the points $\nu_i \in \mathbf{V}$. An important property of the convex hull is that all of its points have the same orientation, both with respect to the rest of the convex hull, and to the points inside (Crépeau 2004). Under these considerations, the computational implementation of the algorithm to obtain the convex hull of a dataset is described as follows:

- Initially all points will be in a list λ_r of remaining points, and will be deleted from it if they are identified to belong to the convex hull. Since the points in the list are already ordered by increasing value of the X coordinate, the first point will always belong to the convex hull, being the vertex at the left–bottom. This point is used as the first reference point ν_r.
- To locate the next point to be added to the convex hull, all points in λ_r are tested against each other, by using orientation. The value for the orientation of each point ν_i is obtained from two line segments, one going from ν_r to ν_i, and another segment from ν_i to any other point ν_j. See Figure 2.7(a).
- The point ν_i will be considered to belong to the convex hull if the orientation with respect to all other points in λ_r is Counter–Clockwise, as shown in Figure 2.7(b), or when it is Collinear with other points that also belong to the convex hull. In the case of collinear points, the one closest to the current reference point is considered first. In this way, all of the points that lie on the convex hull will be included, regardless of whether the points are vertices of the polygon or not.
- Each time a new point is added to the convex hull curve, it becomes the new reference point ν_r, and it is used to test the remaining points, as in Figure 2.7(c). It is also removed from the list λ_r and inserted at the end of the list λ_{ch} that corresponds to the convex hull.

- This process continues until the next point found to be inserted into the convex hull is equal to the first point used as a reference point. At that moment the curve has been closed, and all the remaining points are inside of the convex hull. The program stops evaluating any more points when this condition is reached, otherwise it would continue and produce spiral–like open curves that include all the points in the sample.



(a) Vertex ν_i rejected (b) Vertex ν_i accepted



(c) Continue with new ν_r

Figure 2.7: Selection of vertices belonging to the convex hull of a dataset, based on orientation.

2.3 Notions related to the concept of curvature

2.3.1 Formulae and definitions

The definition of the curvature of a curve is in general given under the assumption that the curve is of C^2–class, so that the 1^{st} and 2^{nd} derivatives are continuous. However, the notion of curvature can be generalised to the class of curves which possess continuous second derivatives except at a finite number of points where a jump discontinuity in the first derivative can occur. Such a curve can be seen as the sum of a finite number of

C^2–curves, joined together at n discontinuity points $s_0, s_1, \ldots, s_n$. Figure 2.8 shows a diagram of a curve of this kind.

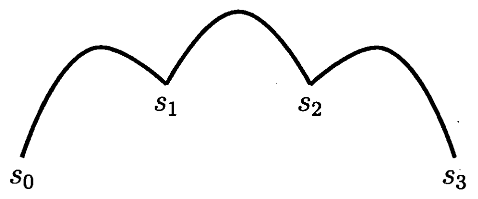


Figure 2.8: A curve with discontinuity points s_1 and s_2.

We can introduce the *exterior angle* $\alpha(s_i)$ formed by the right and left tangents at a point of discontinuity s_i (van Rooij 1965). The total curvature σ of the curve is then defined as the sum of the integral curvatures $k(s)$ of the smooth segments, plus the sum of the exterior angles at all discontinuity points. We then have the following formula for the curvature of an open curve:

$$\int_{s_0}^{s_n} d\sigma = \sum_{i=0}^{n-1} \int_{s_i}^{s_{i+1}} k(s)ds + \sum_{i=1}^{n-1} \alpha(s_i) \tag{2.3.1}$$

where $\sum \alpha(s_i)$ is the sum of all exterior angles enclosed by the right and left tangents at points of discontinuities between s_0 and s_n.

In the case of a discrete curve the points of discontinuity are its vertices, while the smooth curves are straight line segments that have curvature equal to zero. (Sullivan 2006) gives a description of some properties of curvature and the relationship between the smooth and discrete cases. We then have that for the discrete case the curvature is reduced to the sum of the angles at the vertices. Let us denote with $\alpha(\nu_i)$ the exterior angle at the vertex ν_i of a curve. Then the expression

$$\omega = \sum_{i=1}^{n} \alpha(\nu_i) \tag{2.3.2}$$

represents the total curvature of a closed polygonal curve. If this curve is the boundary of a closed simple polygon, ω is always equal to 2π (an elementary case of the Gauss-Bonnet theorem). The following expression:

$$\hat{\omega} = \sum_{i=1}^{n} |\alpha(\nu_i)| \tag{2.3.3}$$

is called the *Total Absolute Curvature* of a curve, abbreviated as the TAC. An important fact is that $\hat{\omega}$ reaches its minimum value 2π on convex curves (polygons). We can show this by using a rectangle, as depicted on Figure 2.9. The angles in each of its four vertices are equal to $\pi/2$, making the total sum of the four angles equal to 2π.

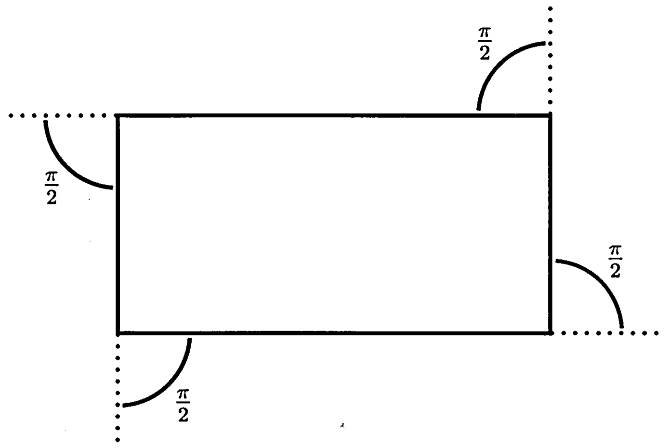


Figure 2.9: The TAC of a convex curve is always equal to 2π.

The convex hull of any curve is by definition a convex curve, and thus has a curvature equal to 2π. Any concave vertex in the curve will increase the TAC and make it larger than 2π. So we have that for any given simple closed curve the following hold:

$$\omega = \sum_{i=1}^{n} \alpha(\nu_i) = 2\pi \tag{2.3.4}$$

and:

$$\hat{\omega} = \sum_{i=1}^{n} |\alpha(\nu_i)| \geq 2\pi. \tag{2.3.5}$$

Therefore for a non–convex curve the excess in the Total Absolute Curvature with respect to 2π can be used as a measure of deviation from the convex curve. Given the dataset $\mathbf{V}$, we take as a *convex curve of reference* the boundary of the convex hull of the data, (referred to as the *CH–curve*).

Definition 2.3.1. We call a *deviation region* or simply a *deviation* the connected vertices and edges of a curve $\mathbf{P}(\mathbf{V})$ that do not belong to the convex hull of the dataset $\mathbf{V}$. Every deviation is connected to the convex hull at both of its ends, at what we call *deviation vertices.*

We want to determine the measure of curvature added to a curve $\mathbf{P}(\mathbf{V})$ by its deviations with respect to the CH–curve. For the problem of curve reconstruction, if our aim is to generate curves of minimum curvature, it is clear that we should minimise the amount of curvature contributed by *reflex vertices*. Let us make this assumption more precise.

It is possible for a discrete curve to have self–intersections, but in this chapter we presume that polygonal curves represent boundaries of simple polygons, *i.e.* are not self–intersecting. Furthermore we consider a simple polygonal curve $\mathbf{P}(\mathbf{V})$ as a set of line segments $\overline{\nu_i \nu_{i+1}}$ that join the vertices $\nu_1, \nu_2, \ldots, \nu_{n-1}, \nu_n$ in a specific order. We consider only closed curves, therefore the line segment $\overline{\nu_n \nu_1}$ belongs to the curve $\mathbf{P(V)}$. For each individual vertex we define:

Definition 2.3.2. The *star* of a vertex ν_i in a curve is the union of the vertex and its two adjacent line segments $\overline{\nu_{i-1}\nu_i}$ and $\overline{\nu_i \nu_{i+1}}$.

In the following explanations of the measurement of the Total Absolute Curvature we will consider the absolute values of all the angles, both convex and concave (reflex). We designate the vertices that lie on the CH–curve with $\mathbf{V_{CH}}$, and with $\gamma_i, i = 1, \ldots, q$ the exterior angles at these vertices with respect to the CH–curve. It is clear that

$$\sum_{i=1}^{q} \gamma_i = 2\pi. \qquad (2.3.6)$$

Let us denote with $\mathbf{V_{conv}}$ the convex vertices of $\mathbf{P(V)}$ and with $\alpha_j, j = 1, \ldots, r$ the exterior angles at these vertices with respect to $\mathbf{P(V)}$, and with $\mathbf{V_{reflex}}$ the reflex vertices of $\mathbf{P(V)}$ and with $\beta_k, k = 1, \ldots, s$, the corresponding exterior angles at these vertices with respect to $\mathbf{P(V)}$; where $r + s = n$. Figure 2.10 shows the location of these vertices in a polygon.

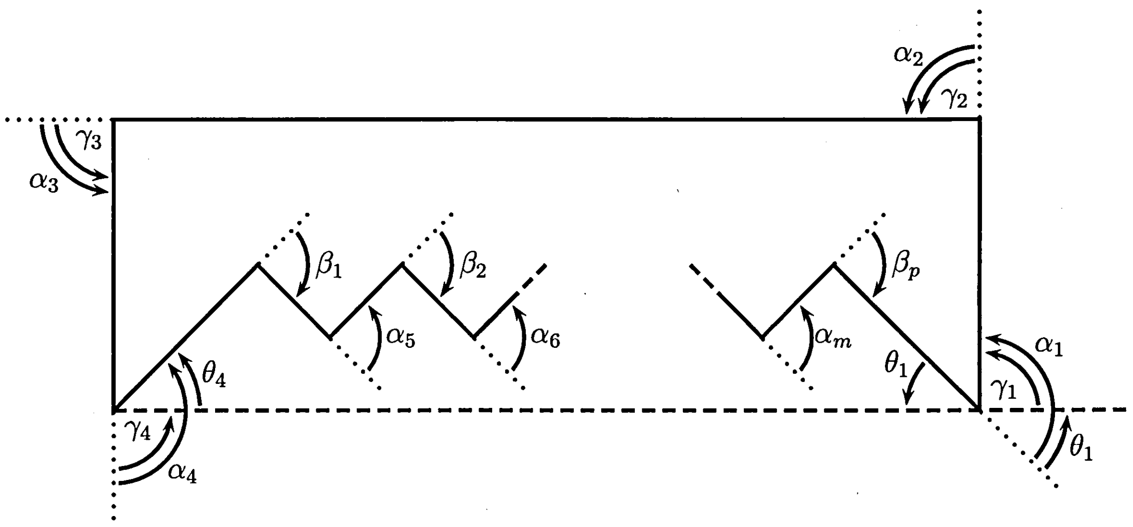


Figure 2.10: General case with several convex and concave angles.

For a simple closed polygon the following equality holds:

$$\sum_{j=1}^{r} \alpha_j - \sum_{k=1}^{s} \beta_k = 2\pi. \qquad (2.3.7)$$

Since $\hat{\omega} = \sum \alpha_j + \sum \beta_k$, the total absolute curvature of a polygonal closed simple curve can be represented as

$$\hat{\omega} = 2\pi + 2\sum_{k=1}^{s} \beta_k. \qquad (2.3.8)$$

The set of convex vertices $\mathbf{V_{conv}}$ can be further split into three disjoint subsets, namely, the subset $\mathbf{V_{conv-CH}}$ of vertices that lie on the CH–curve and such that their stars also belong to the CH–curve; the subset $\mathbf{V_{conv-int}}$ of vertices that are convex but do not belong to the CH–curve; and the subset $\mathbf{V_{conv-dev}}$ of vertices that lie on the CH–curve but their stars do not belong to the CH–curve. Vertices of the last type are called *deviation vertices* as at these vertices the curve is deviated from the CH–curve. The corresponding exterior angles are denoted as $\alpha_l^{CH}, l = 1, \ldots, t$, $\alpha_m^{int}, m = 1, \ldots, u$ and $\alpha_p^{dev}, p = 1, \ldots, v$, respectively. We also have that $l + m + p = r$. Obviously, any part of the curve that is deviated from the CH–curve starts and ends at the neighbouring vertices of $\mathbf{V_{conv-dev}}$, since our curve has no self–intersections. Segments that have a deviation vertex as an end–vertex, but do not belong to the CH–curve, are called *deviating segments*.

Therefore, Equation 2.3.7 can be rewritten as:

$$\sum_{l=1}^{t} \alpha_l^{CH} + \sum_{m=1}^{u} \alpha_m^{int} + \sum_{p=1}^{v} \alpha_p^{dev} - \sum_{k=1}^{s} \beta_k = 2\pi. \tag{2.3.9}$$

Each α_p^{dev} is equal to $\gamma_p + \theta_p$; where by θ_p we denote the angle at a deviation vertex of the curve $\mathbf{P}(\mathbf{V})$ with respect to the CH–curve, or in other words, the angle between the deviating segment of $\mathbf{P}(\mathbf{V})$ that has this deviation vertex as one of the end–vertices and the (imaginary) segment of the CH–curve, that would have the same deviation vertex as one of the end–vertices. We call such an angle a *deviating angle*, as illustrated in Figure 2.11.

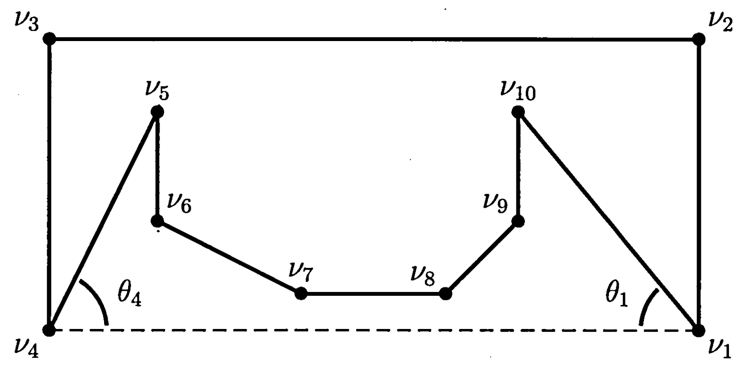


Figure 2.11: Deviation angles in a non-convex curve. The deviating angle at vertex ν_4 is formed by the deviating segment $\overline{\nu_4\nu_5}$ and the (imaginary) segment $\overline{\nu_4\nu_1}$ belonging to the CH–curve. At vertex ν_1 the deviation angle is formed by segments $\overline{\nu_4\nu_1}$ and $\overline{\nu_1\nu_{10}}$.

The above–mentioned considerations transform Equation 2.3.9 into the following one:

$$\sum_{l=1}^{t} \alpha_l^{CH} + \sum_{m=1}^{u} \alpha_m^{int} + \sum_{p=1}^{v} \gamma_p + \sum_{p=1}^{v} \theta_p - \sum_{k=1}^{s} \beta_k = 2\pi. \tag{2.3.10}$$

The sum $\sum_{l=1}^{t} \alpha_l^{CH} + \sum_{p=1}^{v} \gamma_p$ can be rewritten as $\sum_{i=1}^{q} \gamma_i$ and, as shown in Equation 2.3.6, is equal to 2π. Taking this into consideration, we obtain the following expression:

$$\sum_{m=1}^{u} \alpha_m^{int} + \sum_{p=1}^{v} \theta_p - \sum_{k=1}^{s} \beta_k = 0. \tag{2.3.11}$$

This expression can be rewritten as:

$$\sum_{m=1}^{u} \alpha_m^{int} + \sum_{p=1}^{v} \theta_p = \sum_{k=1}^{s} \beta_k. \tag{2.3.12}$$

Therefore, to find the curve (or curves) of minimum total absolute curvature among all curves that span the given data, it is sufficient to minimise either the sum of exterior angles at reflex vertices or the sum of exterior angles at the internal convex vertices and the deviating angles.

From Equation 2.3.12 it follows that if the curve does not have internal convex vertices then the amount of deviation of the curve from the CH–curve is concentrated in deviating angles, and we can extract the following theorem:

Theorem 2.3.1. *The* total absolute curvature *of a non–convex polygon without self–intersections is equal to the curvature of its convex hull plus the absolute value of the curvature of the deviation regions.*

Proof: We will use a certain property of the sums of the angles around a triangle. From basic trigonometry we know that in the triangle of Figure 2.12:

$$\beta = \pi - \beta' \tag{2.3.13}$$

and

$$\beta' + \theta_1 + \theta_2 = \pi. \tag{2.3.14}$$

Substituting to solve for β we have that:

$$\beta = \theta_1 + \theta_2. \tag{2.3.15}$$

These concepts are used to measure the curvature of the polygon in Figure 2.13. We observe that: $\alpha_1 = \frac{\pi}{2} + \theta_1$, $\alpha_2 = \frac{\pi}{2} + \theta_2$, $\alpha_3 = \frac{\pi}{2}$ and $\alpha_4 = \frac{\pi}{2}$. Computing the absolute curvature of the polygon we have:

$$\begin{aligned} \hat{\omega} &= \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 + \beta \\ &= \frac{\pi}{2} + \frac{\pi}{2} + \frac{\pi}{2} + \theta_1 + \frac{\pi}{2} + \theta_2 + \beta. \end{aligned} \tag{2.3.16}$$

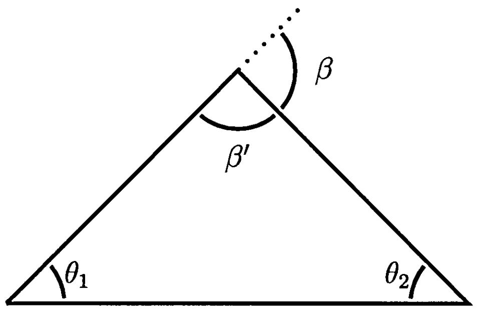


Figure 2.12: Relationship between internal and external angles in a triangle.

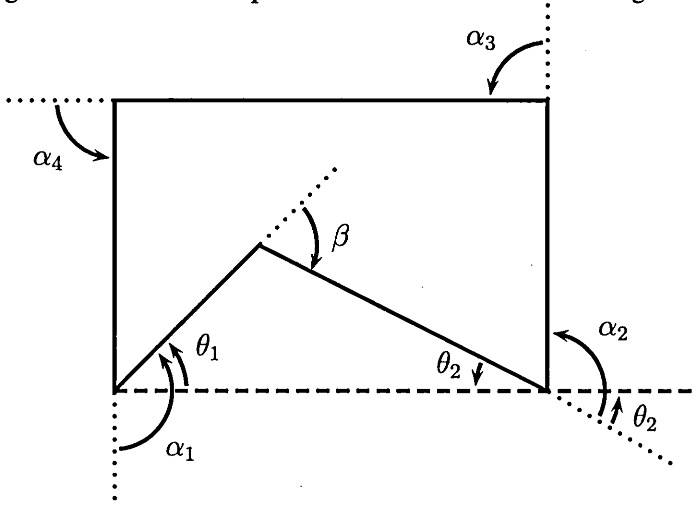


Figure 2.13: Sum of angles of a polygon with one deviation and no internal convex vertices.

Applying Equation 2.3.15 to the sum of the angles, we get:

$$\begin{aligned}\hat{\omega} &= 2\pi + \beta + \theta_1 + \theta_2 \\ &= 2\pi + \beta + \beta \\ &= 2\pi + 2\beta, \end{aligned} \tag{2.3.17}$$

which verifies the result presented in Equation 2.3.8. ■

We must note that deviating angles always appear in pairs and each pair indicates a deviation of the curve from the reference convex curve. Each deviated part $\mathbf{P(V)}_d$, $d = 1, \dots, f$ contains in its turn some convex internal vertices $\mathbf{V^d_{conv}}$ (its number may be equal to zero) and reflex vertices $\mathbf{V^d_{reflex}}$.

Taking into consideration that the reflex vertices may be split into the disjoint subsets, corresponding to the deviated parts of the curve, we can transform Equation 2.3.8 into:

$$\hat{\omega} = 2\pi + 2\sum_{d=1}^{f}\sum_{k^d=1}^{s^d}\beta_{k^d}. \tag{2.3.18}$$

Equation 2.3.12 holds for each deviated part:

$$\sum_{m^d=1}^{u^d} \alpha_{m^d}^{int} + \sum_{p^d=1}^{v^d} \theta_{p^d} = \sum_{k^d=1}^{s^d} \beta_{k^d}. \tag{2.3.19}$$

We now consider the case when extra vertices are inserted to the polygonal curve. A vertex ν_x will be inserted into the segment $\overline{\nu_i \nu_{i+1}}$, creating new segments $\overline{\nu_i \nu_x}$ and $\overline{\nu_x \nu_{i+1}}$. This will affect the curve by changing the angles at vertices ν_i and ν_{i+1}, and adding a new angle at vertex ν_x. This is shown in Figure 2.14.

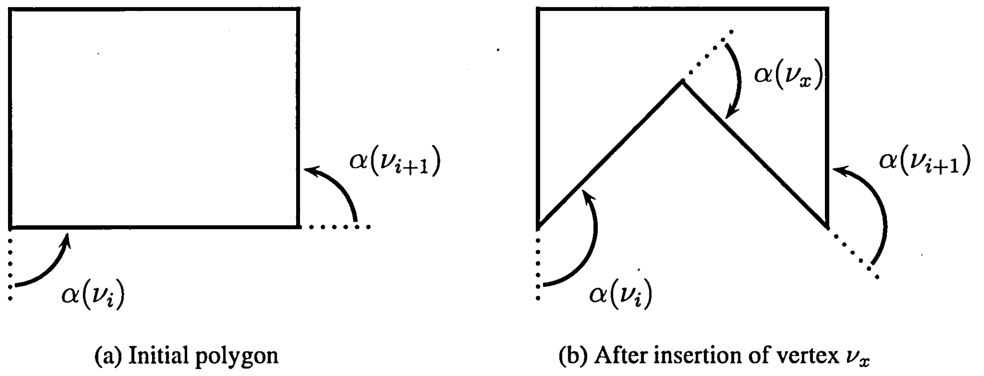


Figure 2.14: Changes in the angles of a polygon due to the insertion of a new vertex.

Let us define a convex region R_{convex} as a part $\nu_i, \nu_{i+1}, \ldots, \nu_{i+j}$ of a curve $\mathrm{P(V)}$ that satisfies the condition that each vertex that belongs to R_{convex} is convex, but vertices ν_{i-1} and ν_{i+j+1} are reflex. A concave region $R_{concave}$ is defined analogously. Both are illustrated in Figure 2.15.

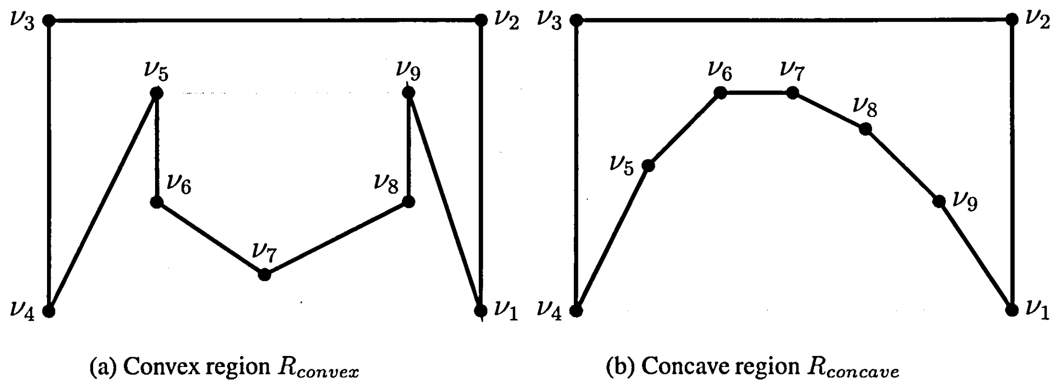


Figure 2.15: Convex and concave regions, shown in grey.

Those segments of a curve whose vertices are of different types (*i.e.* one is reflex, and another convex) are called *separating segments*. For any region we can extend the delimiting separating non–parallel segments until they intersect. The curvature of the region will be equal to the angle at the intersection point, regardless of the number of vertices in the region. We call this the *incorporated curvature* of the region, shown in Figure 2.16(a).

From the above mentioned we get the following statement:

Lemma 2.3.2. *Suppose we construct a curve* $\mathbf{P}_1(\mathbf{V})$ *which has g convex regions and h concave ones. If we add new vertices to these regions in such a way that the number and type of regions of each kind are preserved, then the new curve* $\mathbf{P}_2(\mathbf{V})$ *will possess the same total absolute curvature as* $\mathbf{P}_1(\mathbf{V})$.

Proof: Consider the convex region in Figure 2.16(a). For a vertex to be inserted in this region without creating new regions, it must be inside of the grey area shown in the figure. The curvature ω_R of the original region is:

$$\omega_R = \alpha_1 + \alpha_2. \tag{2.3.20}$$

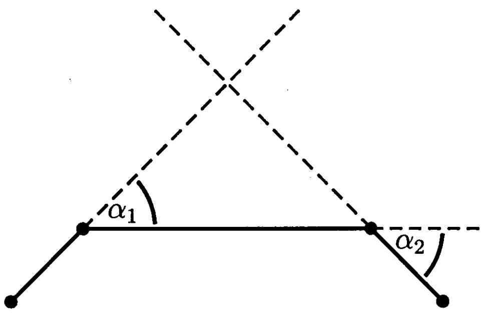
The insertion of a new vertex changes two of the angles in the existing curve (α_1 and α_2), and adds a new angle (α_3). The new curvature is

$$\omega'_R = \alpha'_1 + \alpha'_2 + \alpha_3, \tag{2.3.21}$$

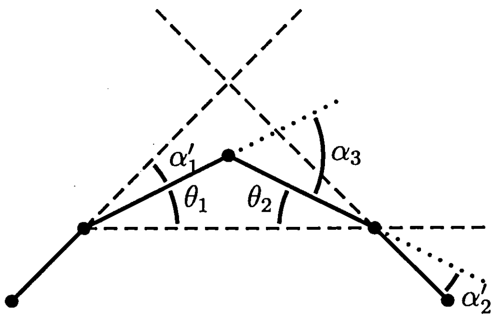
where $\alpha_1 = \alpha'_1 + \theta_1$ and $\alpha_2 = \alpha'_2 + \theta_2$. From Equation 2.3.15 we know that $\alpha_3 = \theta_1 + \theta_2$, thus the curvature of the region becomes

$$\begin{aligned} \omega'_R &= \alpha'_1 + \alpha'_2 + \theta_1 + \theta_2 \\ &= \alpha_1 + \alpha_2. \end{aligned} \tag{2.3.22}$$

The same applies for any number of insertions, both in convex or concave regions. ■



(a) Convex region (shown in grey)



(b) After insertion of a vertex

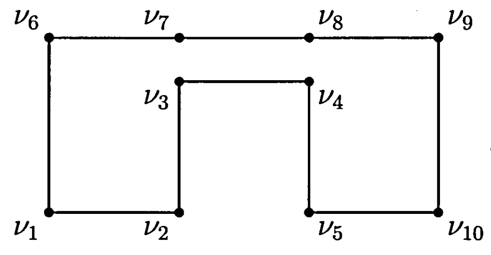
Figure 2.16: Insertion of a vertex into a convex region.

2.3.2 Spherical image of a curve and curvature identical curves

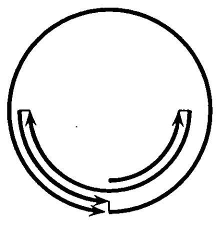
The *spherical image* of a curve illustrates graphically the concept of Total Absolute Curvature. The spherical image for a polygonal curve is constructed by means of outwards unit normals to the line segments of the curve. All of them are translated to the same origin such that their endpoints will lie on the unit circle. Let us suppose that we walk around the boundary of a polygon, for example, in Counter–Clockwise direction starting from vertex ν_1, passing through all the vertices according to their order until we arrive again at the vertex ν_1. From this walk, a corresponding walk on the circle is generated, where some parts of the circle are traversed several times for a non–convex curve. The length of this walk is equal to the Total Absolute Curvature of a curve.

The spherical image provides a vertex classification of the curve as for a reflex vertex the direction will be opposite to the chosen orientation. The spherical images can be put in one–to–one correspondence for two curves of the same dataset if the numbers of concavities/convexities and corresponding incorporated curvatures for both curves are the same. We say in this case that two curves are *curvature identical.* If the total absolute curvatures are equal for two curves, but they are not curvature identical, their spherical images will be different; because in this case either the numbers of concavities (*i.e.* concave regions of a curve) will be different or the incorporated curvatures will be different. Therefore, the spherical image can be used as a representative of any subset of curvature identical curves that span the same dataset.

Examples of two *curvature identical* curves and their corresponding spherical images are given in Figure 2.17 and Figure 2.18. The spherical images are represented as starting from the normal vector of the segment $\overline{\nu_1\nu_2}$. For the sake of clarity, the normal vectors are shown growing outwards in the circle.



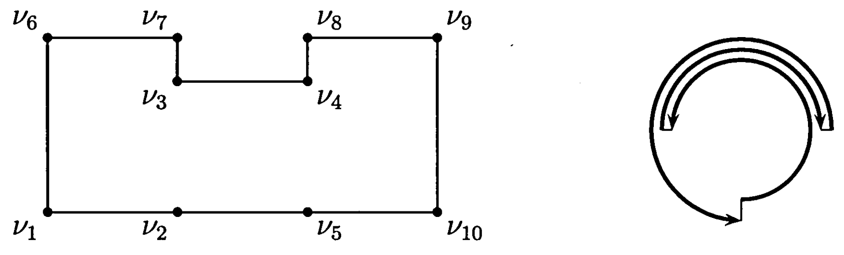
(a) Fitted curve



(b) Spherical image

Figure 2.17: Curve fitted over a ten–point dataset, with its spherical image.

If we change the starting point of our walk in one of the polygons, the spherical images of both polygons can be put in one–to–one correspondence. From the definition



(a) Fitted curve (b) Spherical image

Figure 2.18: Different curve fitted over the ten–point dataset, with its spherical image.

of the spherical image it follows that in order to determine the excess of total absolute curvature of a given curve with respect to the convex curve, it is sufficient to reconstruct the outwards normals only to separating and deviating segments. Then the length of the walk on a circle that the endpoints of these normals generate, by observing the proper ordering, will be equal to this excess.

We are interested in determining the general properties of discrete curves of minimum Total Absolute Curvature. As we saw in the previous subsection, a curve of minimum TAC spanning the data may not be unique. An open problem is, given a curve of minimum Total Absolute Curvature that spans a given dataset, determine other curves also of minimum TAC spanning the same data. This problem can be reformulated in a more general form as determining the subset of curves that span the data and also have the same Total Absolute Curvature, equal to a certain value Ω_α.

In general, we can assume that the set $\mathrm{P}(\mathrm{V})_n$ of all admissible curves that span the same finite discrete data is divided into a finite number of disjoint subsets, each subset $\mathrm{P}(\mathrm{V})_n^i$ containing curves with the same value of Total Absolute Curvature Ω_{α_i}.

Curves with the Total Absolute Curvature equal to Ω_{α_i} might be not curvature identical. It might be the case that the subset of curves with the same $\hat{\omega}$ are further divided into several disjoint sub–subsets of curvature identical curves. The curve that has the minimum Total Absolute Curvature with respect to all admissible curves, is denoted with $\mathrm{P}(\mathrm{V})_{min}$.

2.4 Potential applications of TAC

This section presents two practical applications where the concepts previously described for the Total Absolute Curvature of discrete curves can be used.

2.4.1 Curve reconstruction from point clouds

From the results in Section 2.3, we can conclude that in order to obtain a curve of minimum Total Absolute Curvature that spans a given data set V, three parameters may be minimised: the *number of deviations* of a curve from the CH–curve, the *number of convex regions in a deviation*, and the *amount of curvature contributed by reflex vertices in each deviation*. The dependence among these parameters is not straightforward. We need to decrease the amount of curvature contributed by all the reflex vertices in a curve, and a curve with only one deviation does not guarantee to be a curve of minimum TAC. However, the above remarks give us an indication on how to approach the study of properties of discrete curves with respect to their Total Absolute Curvature and how to design appropriate algorithms.

In order to minimise Total Absolute Curvature on curves that span the given data, we designed several algorithms. The algorithms serve also to understand better the properties of curves of minimum TAC, given the data. They can be divided in two groups, algorithms of the first group search for a curve with minimum TAC among curves with only one deviation, and in the second group among curves that allow multiple deviations. We first describe the properties of curves with one deviation.

If we have the data set V in a non–convex position, then we can determine the *layers* of this set by repeatedly removing all convex hull elements and considering the convex hull of the remaining set. The set V has k layers if this process terminates after precisely k steps. So the first layer is the convex hull of the whole dataset, and subsequent layers are *nested convex hulls* (see Figure 2.19).

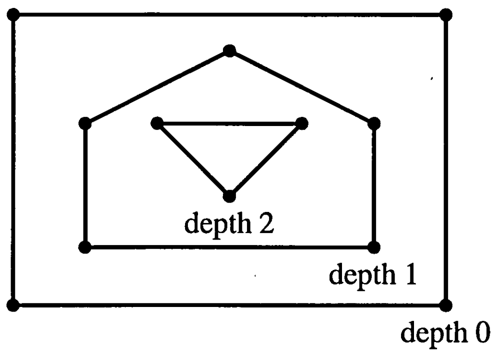


Figure 2.19: Nested convex hulls of a dataset.

Given a curve that covers a dataset with k layers, a segment of the curve whose end–points belong to two different layers is known as a *link segment*. Two link segments form a *bridge* if they belong to the same deviation and both segments are connected to at least one layer in common. Figure 2.20 illustrates link segments and bridges.

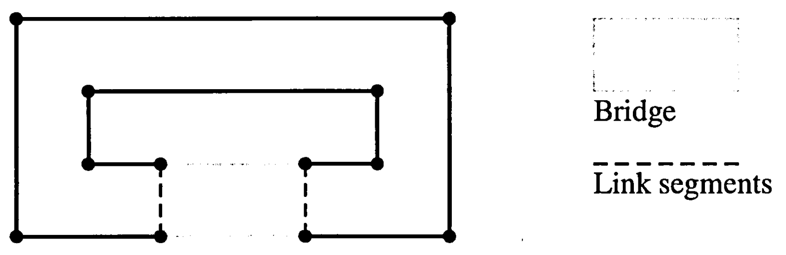


Figure 2.20: A bridge delimited by two link segments, in a dataset with two layers.

If both link segments of a bridge connect successive layers, the bridge is called *short.* To preserve a simple closed curve, no two bridges can have endpoints that belong to the same layers. The following statements are valid:

Lemma 2.4.1. *Given a dataset with k layers, the limits on the number of link segments n necessary to join all layers is as follows: If $k = 1$ the dataset is convex, and there is no need for link segments, $n = 0$. For non–convex curves, $k > 1$ and $n \geq k$.*

Proof: To join all layers with one segment between each pair we need $k-1$ link segments. In the minimum case, we need one additional link segment that joins the innermost layer with the outermost one, thus making the minimum number of link segments $n = k$. A curve with this configuration is shown in Figure 2.21 ■

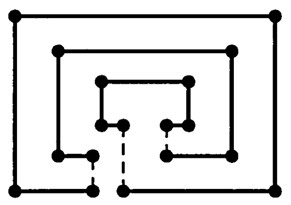


Figure 2.21: Curve with minimum amount of link segments. In the data $k = 3$, and there are 3 link segments.

Lemma 2.4.2. *The maximal number of bridges in a curve with one deviation does not exceed $k - 1$ where k is the number of layers. The maximal number of link segments in a curve with one deviation is equal to $2(k - 1)$.*

Proof: The maximum number of bridges is necessary when all of them are short, as in the curve on Figure 2.22. Each bridge completely joins two layers. In a dataset of k layers there are only $k - 1$ pairs of successive layers. ■

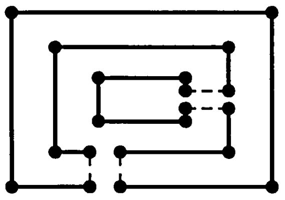


Figure 2.22: Curve with maximum amount of link segments. In the data $k = 3$, and there are 4 link segments.

Lemma 2.4.3. *The total curvature of a curve with one deviation does not exceed* $2\pi + 4\pi(k-1)$*, where* k *is the number of layers. This bound is exact.*

Proof: The curvature of the convex hull is 2π for the whole dataset. Every convex layer inserted increases the curvature by an additional 2π. Each link segment contributes to the curvature with a maximum of π at its endpoints, and in the worst case two link segments are necessary to join every pair of layers. Thus for every layer added, the curvature increases by a maximum of 4π. ■

From the previous proof we can conclude that having less link segments can produce curves with smaller curvature. From Lemma 2.4.1 we know that k is the minimum number of link segments in a dataset with k layers.

We next investigate how multiple deviations influence the TAC. This difficult problem is still under study, but we can make several observations. First is that any new deviation may contribute to TAC up to $2\pi + 4\pi(k-1)$. Adding new bridges may also contribute to curvature. So, a heuristic idea is to keep deviations as simple as possible, and deviating angles as small as possible. Appendix A describes two algorithms to reconstruct curves from a point cloud. One of them builds curves with a single deviation, and the other generates curves with multiple deviations. Below we briefly illustrate how these ideas work in practice.

Two examples are offered, in which the data consist of two layers. For simplicity, we assume that the data lie on two concentric circumferences, of radii r_{outer} and r_{inner} correspondingly. We denote a curve of minimum TAC with $\mathrm{P(V)}_{min}$, and with $\mathrm{P(V)}_{min}^{o}$ the curve with one deviation that has the minimum TAC with respect to all curves with one deviation that span the given data.

Example 1: In this example the data is situated so that if we connect the centre point O with a vertex ν_o on the outer circumference, there is another vertex ν_i that lies on the inner circumference and that belongs to the segment $\overline{O\nu_o}$. Each circumference is composed of 10 points. This example clearly shows that the concept of ‘nearness’ between two sample points is not essential for changing the Total Absolute Curvature of a curve. We can show that the curvature *Qa* of a curve that spans this data with multiple deviations is always larger than the P (V)^ in curve for the same data. Moreover, keeps the same value no matter how small is the distance *ST* *Touter* *Tinner* (soe Figure 2.23).

(a) Curve P (V)Y_{mm}n that is also P (V)7_{mm}

(b) Curve of minimum TAC among curves with multiple deviations

Figure 2.23: Concentric circles with vertices aligned.

This example also shows that a global minimum to a discrete optimisation problem might be very far indeed from the input configuration. For example, we can assume that sample points were originally taken from the curve P (V) sau; presented in Figure 2.24.

Figure 2.24: Original source of the data: curve P (V)

In these points high values of curvature are ‘incorporated’, and therefore they are considered as significant for shape description. The curve in Figure 2.23(b) might be served as an initial approximation of P (V) sau;, but not the curve in Figure 2.23(a).

Example 2: In this example also each circumference has 10 points. The points on the inner circumference are slightly rotated with respect to the outer circle, so that no point on the inner circumference belongs to the segment *0 u o*. In this case we can show that by decreasing the distance *Sr* = *router* —*rinner* at a certain moment the curve of minimum TAC among curves with multiple deviations becomes P (V) min and its TAC tends to *2n.*

The curve in Figure 2.25(a) has one deviation and is P (V)^in and also P (V) min, while Figure 2.25(b) is the curve of minimum TAC among curves with multiple deviations. Both of them are generated using the same dataset.

A new dataset, where the radius of the inner circle has been increased, produces the curves in Figure 2.26. Here Figure 2.26(a) is P(V)JCn but it is no longer P (V) min, and the curve in Figure 2.26(b) is now P (V) min.

(a) TAC = 16.3287 **(b) TAC = 24.1451**

Figure 2.25: Concentric circles with inner circle rotated.

(a) TAC = 16.3167 **(b) TAC = 16.2223**

Figure 2.26: Rotated concentric circles with smaller *Sr.*

Appendix A shows the experiments done using various algorithms to fit a curve over a dataset. The algorithms presented generate curves with one or various deviations, as the one shown in the previous examples. The principles presented in this chapter are used as a parameter to evaluate the curves generated.

2.4.2 Face recognition by curvature of face profiles

A practical application of curvature measures is their use in the field of *face recognition.* Such an idea has been previously explored in (Thodberg and Olafsdottir 2003), where curvature measures are used to determine feature points on curves representing the profile of faces. This section presents the experiments carried on using measurements of Total Absolute Curvature of polygonal curves to perform face recognition.

Face data was obtained from a *structured light scanner* developed in–house at Sheffield Hallam University (Robinson *et al.* 2004), (Robinson 2005). The scanner generates an approximate three–dimensional model of a human face from a single picture, when a pattern of structured light (multiple stripes) is being projected on a target. In theory, the 3D object can be recovered from the captured data in all cases. But in practice, due to varying lighting conditions, occlusions, and pixel noise in the 2D image, the generated 3D data can have inconsistencies.

For the purpose of comparing acquired 3D faces, a transversal section of the face is obtained from the single vertical stripe that covers the tip of the nose. This yields a number of points on the profile of the face. A curve is then fitted over the vertices, and the curvature measures of the discrete curve obtained are used to differentiate the faces. Initially the data is simplified by removing vertices with a curvature less than a certain threshold, keeping only important features of the face. The curve is closed to make a consistent ordering of the vertices and to enable measurements of the area of the polygons. This is done by adding a vertex at the back of the profile, in a position at the middle of the distance between the first and last vertices in the dataset, and that same distance perpendicularly to the back of the profile. As an example, Figure 2.27 shows the closed polygons for two face profiles. The measures obtained for the curve in Figure 2.27(a) are:

```
Curvature =               19.250550
Curvature squared =       16.741678
Length =                  26.719916
Area =                    36.689424
```

and for the curve in Figure 2.27(b), the following results were obtained:

```
Curvature =               19.531384
Curvature squared =       17.754017
Length =                  26.344608
Area =                    37.216353
```

The results of these partial experiments are of limited use because of the lack of a large enough sample population. However further work on this area is suggested, since curvature measures of the faces can be consistent even for different scans of the same person under various conditions.

(a) Male face profile

(b) Female face profile

Figure 2.27: Example profiles used for face recognition by curvature measures.

2.5 Conclusions

The discrete analogues of curvature for a piecewise linear approximation of a curve have been presented in this chapter as the sum of the angles at the vertices of a polygonal curve. By considering the absolute values of these angles, regardless of the direction of the turn, we get the Total Absolute Curvature of a polygon. We classify the vertices of a discrete curve according to their location with respect to the convex hull. A distinction is made between vertices that lie on the convex hull of the dataset, and interior vertices, that lie in what we call *deviations.* The relationship between the angles of each type of vertex and their curvature is developed to demonstrate that a convex curve has the TAC equal to 27t, and non-convex curves will have curvatures larger than 27t dependent on the inner angles at the deviations. From these results we can conclude that the curvature of a curve is highly dependent on the curvature of the deviations. An extension of these concepts will be used in the next chapter to measure the curvature of vertices in a polyhedral surface.

The spherical image of a curve has been presented as an alternative method to measure the Total Absolute Curvature of a polygon, based on the normal vectors to the edges of the curve. We have shown how this method can be used to clearly identify curves that have the same curvature but different geometries. This is the basis of the Polyhedral Gauss Map method that will be presented in Chapters 3 and 4.

The concepts of deviations and layers presented in this chapter can be applied to curve reconstruction algorithms. Some initial experiments on how this can be done are explained in Appendix A. Based on these concepts we have shown that, given the data, a curve of minimum TAC may not be unique, and that a solution to a discrete optimisation problem may lead to an unexpected curve and might be very far from the source of the data samples. The research on the discrete optimisation problem provides the foundation for the research on the curvature of surfaces in three-dimensions.

Chapter 3

Discrete curvature in three dimensions

3.1 Introduction

In many applications a physical object is represented by discrete data, commonly obtained by some measurement system to record the coordinates of sample points on its surface. Triangular or polygonal meshes (*i.e.* piecewise linear surfaces) are commonly used in modern computer–related applications to represent surfaces in three–dimensional space, which cover the sampled points. Therefore, there is a substantial need for accurate estimates of geometric attributes that are directly computed from a mesh, such as surface area, normal vectors, and curvatures. In recent years significant efforts have been made to define the analogues of differential geometry concepts on meshes, which imitate those of a smooth surface (Dyn *et al.* 2001), (Meyer *et al.* 2002), (Borrelli *et al.* 2003) and (Eastlick 2006). Among these concepts surface curvatures are particularly important, as they are basic measures to describe the local shape of a smooth surface. However, the surface of a triangle mesh is not smooth, and there is still no consensus about the most appropriate way of estimating geometric quantities such as curvature. Additionally, various methods are being developed to capture curvature information without referring to higher–order formulae of differential geometry. These methods are based on the discrete curvature concepts and are of growing interest for geometric modelling. They permit the discrete curvatures to be computed directly from triangle meshes using their intrinsic information. The principal difference between a polyhedral and a smooth surface is that the discrete curvatures in a polyhedral surface are concentrated around the vertices and along the edges.

Measures of curvature in a piecewise linear setting should be analogues of *integral formulae* for curvature in a smooth setting and should preserve the integral relations for curvature, such as the Gauss–Bonnet theorem (Polya 1954), (Banchoff 1970), (Brehm and Kühnel 1982). Such analogues exist and were introduced long ago in relation to the theory of non–regular surfaces (see an overview in (Alboul and van Damme 1994)). These analogues were discussed in detail in (Brehm and Kühnel 1982), where the authors also compare discrete curvatures with their smooth counterparts.

In the last decade the number of papers that explore discrete curvatures within certain contexts has increased significantly. Much attention is paid to the discrete Gaussian curvature, known also as the *Angle Deficit*. It has also been referred to as *angular defect* and abbreviated in this thesis as the AD. The concept was brought to the attention of the geometric modelling community in (Calladine 1986), where the author listed several applications of the angle deficit in surface modelling, mostly in the the context of the mechanics of thin–shell structures. Nowadays, the angle deficit is used to evaluate curvature information directly from a mesh, as well as to estimate the Gaussian curvature and derive principal curvatures of the underlying smooth surface, assuming that the mesh samples the surface in a certain way (Peng *et al.* 2003), (Meyer *et al.* 2002).

In (Borrelli *et al.* 2003) the problem of the correct estimation of the Gaussian curvature is investigated in detail, and they argue, on the basis of several approximation results, that approaches based on the use of normalised angular deficits are often erroneous, and can be applied correctly only if the geometry of meshes is precisely controlled. We advocate these conclusions, and in the following chapters we highlight why the angular deficit is sufficient neither to estimate the Gaussian curvature of the underlying smooth surface nor to capture the curvature information of a polyhedral surface. Loosely speaking, the reason is that there are more curvatures for polyhedral surfaces than for smooth ones. This fact is still not fully acknowledged, but without addressing it, it is impossible to develop correct curvature estimates.

In the smooth case, the *Gauss Map* and related *shape operator* completely determine the shape of the original surface (Kühnel 2002). Therefore, efforts have been made to use analogues of the Gauss Map to explore shape characteristics of a complex polyhedral surfaces. For example, an analogue such as the *extended Gaussian image* is used in Computer Graphics and Vision to compare objects and to illuminate the structure of surface shape (Little 1985), (Lowekamp *et al.* 2002). However, the extended Gaussian image and its generalisations construct only normals to the faces of the polyhedral surface without indicating their connectivity. There exist few attempts to create the Gauss map directly from the mesh, but the results are still scarce and ambiguous for non–convex objects (Lowekamp, Rheingans, and Yoo 2002).

Our method constructs a polyhedral analogue of the Gauss map directly from a polygonal mesh and uses this map to characterise surface shape. We believe that such an algorithm is developed here for the first time. We also give a definition of the *Polyhedral Gauss Map* (PGM) and show its conformity with the smooth case. The resulting PGM provides a description of the surface by determining its *curvature domains*, that is, flat, convex or saddle regions, with respect to incorporated curvatures. These domains are often only implicitly present in a polyhedral surface, and cannot be determined by the sign of the angle deficit only. Each domain can be split up into uniquely determined sub–domains; therefore each surface can be associated with the collection of these sub–domains. The method provides also a better insight into the geometric structure of complex triangle meshes, by describing various vertex types, some of them with a very complex PGM.

A good understanding of the geometry of meshes is a step towards more robust mesh manipulation algorithms. The PGM method besides shape recognition and description can be used for optimisation of the underlying model or for developing subdivision schemes. Finally, the proposed PGM is computationally viable, can be viewed dynamically, and is effective in visualising curvature features of complex polyhedral surfaces. The theory and algorithms in this and the next chapter have been partially presented in (Alboul *et al.* 2005) and (Alboul and Echeverria 2005).

3.1.1 Previous research on Gauss Maps

The papers (Banchoff 1967) and (Banchoff 1970) present the analogous to the Gauss–Bonnet theorem for the curvature of polyhedral surfaces. Banchoff draws directly from the method described by Gauss to find the curvature of a surface, and presents the Gauss Map of vertices in a polyhedral surface, using the normal vectors of the faces around a vertex. He proves how the curvature of a vertex can be measured in an analogous way to the methods for a smooth surface. He points out the relationship between the critical points of a surface and the types of vertices that can be identified with the Gauss Map. This is also the case in (Brehm and Kühnel 1982), where curvature measures are expressed in terms of the number of critical points.

One of the first studies on using Gaussian Images to identify objects is presented in (Horn 1983). The method used is described as *Extended Gaussian Images* (EGI) and is based, for the discrete polyhedral case, on projecting the normal vectors of the face of the polyhedron into a sphere, and assigning to each of these normals a density directly related to the corresponding face area. The resulting Gaussian image is considered as a weighed mass, with a centre of mass always located at the centre of the sphere. In this report, Horn also demonstrates the relationship between the computation of curvature for the discrete and smooth cases. Further analysis is made there for the best possible tesselation of the sphere in order to group the Gaussian image in cells. Useful tesselations need to have cells of similar area and shape, have regular shapes and provide good resolution on the surface of the sphere. The best tesselations found are obtained from the projection of regular or semi–regular polyhedra onto the sphere, and then further subdividing them into triangles. In another section, Horn deals with curvature measures of solids of revolution, including a torus, which is a non–convex object but can have its curvature measured using EGI.

(Little 1985) uses a slightly different definition of the EGI, where each of the normal vectors has a length proportional to the area of its corresponding face. He investigates how this form of EGI, being unique for every convex polyhedron, can be used to reconstruct the original object, according to the *Minkowski theorem* (Lyusternik 1963). This approach requires a definition of the orientation of the faces of the object, known as a *combinatorial type*, which describes the adjacency relationship between faces and edges. The technique can be directly used to reconstruct an object in 2D thanks to the inherent definition of orientation, but in the case of 3D it requires an iterative process to approximate the target shape. This is because of the disconnected nature of the normals. Reconstructing a 3D surface requires extra data structures that relate to the connectivity of normal vectors of adjacent faces.

Another estimation of the curvature of a polygonal surface is obtained in (Cohen-Steiner and Morvan 2003) based on the normal cycle at vertices, edges and triangles. An error bound is proven from the curvature of a discrete surface obtained as a restricted Delaunay triangulation of the smooth surface.

We will present a novel approach to measuring curvature that has not been explored before. The understanding of positive and negative components of curvature permits the correct characterisation of vertices, impossible using previous methods. Additionally the Polyhedral Gauss Map proposed makes use of the correlation of the normal vectors directly from the polygonal mesh. It reflects more accurately the geometry of the vertices and their local neighbourhoods on the surface.

3.1.2 Existing applications using Gauss Maps

The Gauss Map has been used in the past to aid in the solution to several different problems. The way to compute it varies as well. Here we present some applications and explain the methods used to find it.

An analysis on the treatment of spherical polygons is found in (Chen *et al.* 1993) for an application on automatic machining of mechanical parts. The problem tackled is identifying optimal orientations of pieces in a cutting machine to perform the most piece cuts in one single setup. They use the Gauss Map of the pieces, where the aim of the algorithms is to identify the areas of the sphere where most of the spherical polygons are situated, as the optimal orientations of the pieces in the cutting machine. They define operations of the surface of the sphere to determine the intersections of spherical polygons with a great circle, and then find the hemisphere with the most number of spherical polygons.

The Gauss Map of a vertex is computed and used in (Rodríguez and Rosenberg 2000) to extend the Cauchy theorem related to the rigidity of convex polyhedra in $\mathbb{R}^3$. By using only face orientations they are able to define the 'Gaussian image' in such a way that for certain non–convex vertices this image represents a convex spherical polygon. They explore the different vertex configurations, such as: convex cone, non–convex cone, saddle and figure–eight. In their approach the Gauss Map for a given vertex is obtained from the normal vectors of the incident faces, which are projected to the surface of a sphere and joined with geodesic arcs. The direction of these vectors can be changed so as to get a convex projection on the surface of the sphere. If such a convex geodesic polygon can be found for all the vertices in the polyhedron, then it is qualified as rigid, including polyhedra with figure–eight vertices. The authors do not intend their version of Gauss Map to be used as a measurement of curvature. The algorithm used for the Gauss Map avoids intersecting geodesic arcs by changing the direction of the normal vectors, and hence it does not really reflect the amount of curvature incorporated in a vertex.

The work of (Grinspun and Schröder 2001) makes use of the Gauss Map of subdivision surfaces to detect self–intersections of the surface when it is subjected to deformation. The test implemented here requires placing a plane between the centre of the sphere and the Gauss Map area on the surface of the sphere. If such a plane exists, then it is assumed that there is no intersection. This approach uses a different version of the Gauss Map for whole patches of the surface, while each vertex contributes only with one vector to the Gauss Map, in a manner similar to the smooth case.

In (Yamauchi *et al.* 2005) the Gauss Map is used to determine a more adequate mesh segmentation to do texture mapping without deformations. They define a segmentation that distributes curvature evenly among every patch. The iterative growing process for the patches places constraints on how fast they can develop, so that they remain balanced in curvature. The curvature measure is used both to decide when a vertex, edge or face is incorporated into a patch, and to balance the growth of the larger patches with respect to their Gauss area. Their choice of using Gauss Map to obtain the curvature is based on mathematical robustness. In their implementation, the Gauss Map provides curvatures in the range $[0, 4\pi]$. The Gauss Map of vertices is obtained using the normal vectors of adjacent faces, while the Gauss Map for edges and faces is computed using the normal vectors of the incident vertices. Again this approach does not consider the case of non–convex vertices.

3.2 Basic concepts and definitions

For the purpose of this thesis we are interested only in discrete curvatures related to the integral Gaussian curvature, *i.e.* those that are supported on the vertices. In what follows we give a brief comparative analysis between the known integral relations for curvature for smooth surfaces and their discrete counterparts.

3.2.1 Polyhedral surface

By a polyhedral surface we understand a triangulated polyhedral surface. Let a vertex be a triple: $\nu = (x, y, z) \in \mathbb{R}^3$. We design $\mathbf{V}$ as a finite point set in three–dimensional space, $\mathbf{V} = \{\nu_i; i = 1, 2, \ldots, n\}$, we denote by $\mathbf{P}(\mathbf{V})$ a polyhedral surface with the vertex set $\mathbf{V}$. The term *polyhedron* refers to a closed polyhedral surface. In such a setting a polyhedron is bounded, but might be non–homeomorphic to a sphere. Also it might be multi–connected and self–intersecting, and its interior volume is not necessarily part of the polyhedron. Therefore, a polyhedron is not necessarily a solid body. A triangle mesh is a particular case of a polyhedral surface. Therefore, all properties of a polyhedral surface discussed below are applicable to a mesh.

Given a polyhedron $\mathbf{P}(\mathbf{V})$, the set of its vertices is denoted by $\mathbf{V}$, the edges by $\mathbf{E}$, and the faces by $\mathbf{F}$.

Definition 3.2.1. The *star of a vertex* ν, denoted as $star(\nu)$, is the union of all the faces and edges that contain the vertex. The *link of the vertex* (the boundary of the star) is the union of all those edges of the faces of the $star(\nu)$ that are not incident to ν. It is denoted as $link(\nu)$.

For simplicity we will make an initial distinction between vertices.

Definition 3.2.2. We refer as *manifold vertices* to those vertices for which their star maps one–to–one to an open disc. Vertices for which such relationship cannot be found will be called *non–manifold vertices*. These are vertices with self–intersections of the faces in their star.

3.2.2 Integral Gaussian curvature and the angle deficit

For a domain U of a smooth surface S the Gauss map $N(U)$ is the map assigning to each point $p \in U$ the point on the unit 2–sphere $S^2 \in \mathbb{R}^3$, by translating the unit normal vector $N(p)$ to the origin (Kühnel 2002). The endpoints of normals, therefore, will cover a certain region on S^2 (see Figure 3.1). This method is an extension of the one presented in Section 2.3.2 for the spherical image of a planar curve.

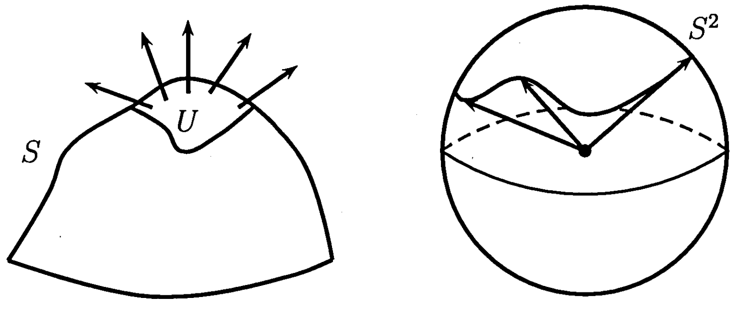


Figure 3.1: Gauss Map of a smooth surface.

Given a neighbourhood $U(p)$ on S, the ratio of the area $N(U(p))$ to the area of $U(p)$ can be considered as a measure of the amount of curvature of the surface S near the point p. Then the Gaussian curvature $K(p)$ is defined by setting

$$K(p) = \lim_{U(p)\downarrow p} \frac{area(N(U(p)))}{area(U(p))} \tag{3.2.1}$$

where the limit is taken as the neighbourhood $U(p)$ contracts down to the point p (do Carmo 1976).

If a neighbourhood $U(p)$ is sufficiently small such that the map $N(U(p))$ is one–to–one and orientation–preserving (outward normals at corresponding points on S and S^2 correspond), then the area $N(U(p))$ is considered positive, and the corresponding region $U(p)$ is said to be strictly *convex* and $K(p) > 0$. If the map $N(U(p))$ is one–to–one but orientation reversing, then the area $N(U(p))$ is considered to be negative, p is a saddle point and $K(p) < 0$. Of course, different regions of S can be mapped to the same region on the unit sphere, which results in multiplicities of the Gauss map.

Therefore for a region $U(p)$, for which the map $N(U(p))$ might not be one–to–one, the integral Gaussian curvature (K_{int}) is understood as the integral of the area of the image of $U(p)$ under the Gauss mapping:

$$K_{int} = \int_U K dA. \tag{3.2.2}$$

For an entire closed smooth surface the previous formula turns into the mathematical expression of the Gauss–Bonnet theorem (Bloch 1998):

$$\int_S K dA = 2\pi\chi(S), \tag{3.2.3}$$

where $\chi(S)$ is the Euler characteristic of S. The discrete analogue of Equation 3.2.2 is known as the *Angle Deficit* (shown in Figure 3.2), first introduced by Descartes, and which measures the *discrete Gaussian curvature* ω around vertex ν:

$$\omega = 2\pi - \theta, \tag{3.2.4}$$

where $\theta = \sum \alpha_i$ is the total angle around vertex ν, and α_i are those angles of the faces in $star(\nu)$ that are incident on ν. This is a polygonal analogue of the Gauss–Bonnet theorem (Banchoff 1970).

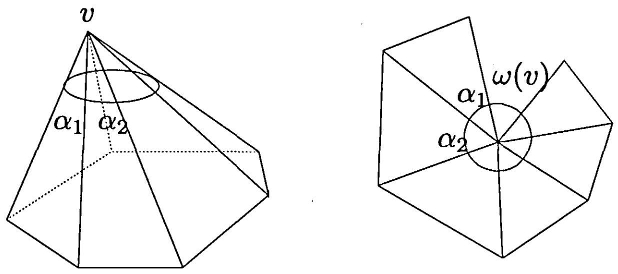


Figure 3.2: Computation of the Gaussian curvature (ω) around vertex ν.

For any non–vertex point in the surface $p \in \mathrm{P}(\mathrm{V})$, the curvature ω is identically equal to zero. Hence, for a domain $U \subseteq \mathrm{P}(\mathrm{V})$ the total curvature Ω_U is determined as the sum of the curvature of every vertex:

$$\Omega_U = \sum_{\nu \in U} \omega(\nu). \tag{3.2.5}$$

For an oriented closed polyhedral surface $\mathrm{P}(\mathrm{V})$ of genus g the total curvature $\Omega_{P(V)}$ is equal to $(1-g)4\pi$, so the analogue of the integral relation for the Gaussian curvature is preserved (Alboul and van Damme 1994), (Banchoff and Kühnel 1998).

3.2.3 Integral absolute curvature and its discrete analogue

The following measure which we determine is an analogue of the integral absolute curvature for a polyhedral domain. The most obvious candidate for this measure seems to be the sum of absolute values of the angle deficits around the vertices in the domain.

However, in Figure 3.3 we can see that in both polyhedra all curvatures $\omega(\nu_i)$ are positive. In the depicted polyhedra the curvatures $\omega(\nu_i)$ are actually equal for every corresponding vertex, *i.e.* for all corresponding vertices $\nu_i \in P_1$ and $\nu_i \in P_2$, the curvature $\omega(\nu_i)$ is the same. Therefore, we have:

$$\Omega(P_1) = \Omega(P_2) = \sum_{\nu \in P_1} |\omega(\nu)| = \sum_{\nu \in P_2} |\omega(\nu)| = 4\pi. \tag{3.2.6}$$

The left polyhedron in Figure 3.3 is non–convex, but Equation 3.2.6 does not reflect this fact.

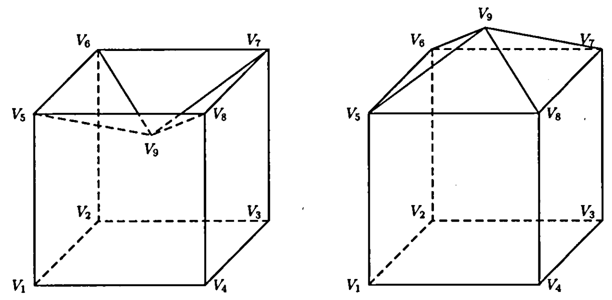


Figure 3.3: Two polyhedra with similar Gaussian curvatures.

The *total absolute curvature* $K_{abs} = \int_S |K| dA$ for a closed non–convex smooth surface S is greater than 4π; therefore, $\sum_{\nu \in P} |\omega(\nu)|$ is not an appropriate analogue of K_{abs}. The problem is that the curvature ω around a vertex may consist of positive and negative *components* that are 'glued' together, and the task is to separate them.

For any given vertex ν we can imagine the convex hull of the subset of vertices included in $star(\nu)$. If the vertex ν belongs to the convex hull of its star, then we can define another star, denominated $star^+(\nu)$, which contains the vertex ν and the faces and edges of the convex hull that are incident on the vertex. We refer to $star^+(\nu)$ as the *convex cone* of vertex ν. Then, using a variation of Equation 3.2.4, the *positive (extrinsic) curvature* ω^+ is defined as:

$$\omega^+ = 2\pi - \theta^+ \tag{3.2.7}$$

where θ^+ is the sum of angles of the faces incident on ν in $star^+(\nu)$. The positive curvature ω^+ is equal to zero if the vertex ν and all the vertices in $link(\nu)$ lie in the same plane. If the convex cone around ν does not exist, *i.e.* ν lies inside the convex hull of $star(\nu)$, then ω^+ is, by definition, equal to zero.

The *negative (extrinsic) curvature* ω^- of ω is determined as the complement of the positive curvature:

$$\omega^- = \omega^+ - \omega. \tag{3.2.8}$$

The *absolute (extrinsic) curvature* $\hat{\omega}$ is defined as:

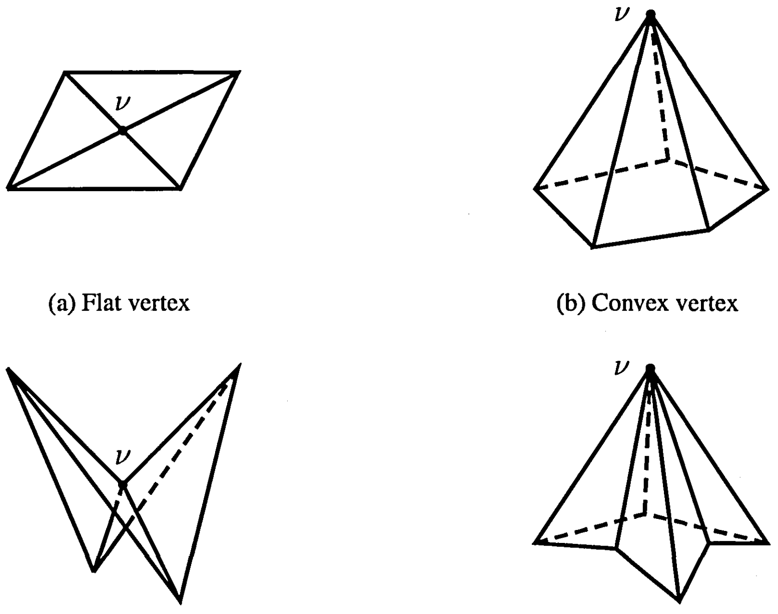
$$\hat{\omega} = \omega^+ + \omega^-. \tag{3.2.9}$$

The word 'extrinsic' in the introduced curvatures is used due to the fact that they reflect how a vertex is embedded in space, while the angle deficit, computed only by using the angles around a vertex remains an intrinsic measure.

Four basic types of vertices for an embedded polyhedral surface are then distinguished, using the positive and negative components of curvature ω^+ and ω^-. We call a vertex:

- *Flat* if $\omega = 0$;
- *Convex* if $\omega^+ = \omega$;
- *Saddle* if $\omega^- = -\omega$;
- *Mixed* if $\omega^+ > 0$ and $\omega^+ \neq \omega$.

In other words, convex vertices have a curvature which is entirely positive, saddle vertices have entirely negative curvature, and mixed vertices have both kinds of curvature. Examples of these kinds of vertices are shown in Figure 3.4.



(a) Flat vertex (b) Convex vertex

(c) Saddle vertex (d) Mixed vertex

Figure 3.4: Vertex types identified using the Total Absolute Curvature.

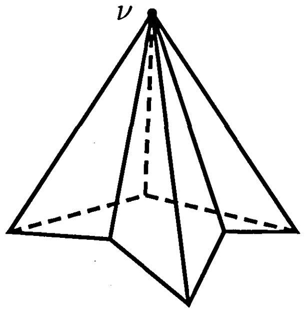
We now establish an analogy between the measure of the curvature of a planar polygon, as presented in Section 2.3.1, and the curvature of a vertex in a two–dimensional triangulated polyhedral surface. The curvature of a planar polygon is obtained from the sum of angles at the vertices. For the curvature of a vertex in space we measure the sum of the angles of the incident faces.

For a vertex ν, we can split the set F of faces in $star(\nu)$ into two disjoint subsets: The faces that belong to the convex cone $star^{+}(\nu)$ are placed in the set F^{+}, and the angles of these faces at the vertex ν are denoted with $\alpha_j, j = 1, \ldots, r$. The faces that do not belong to $star^{+}(\nu)$ are placed in $\mathrm{F}^{\mathrm{dev}}$, and the angles of these faces at the vertex ν are denoted with $\beta_k, k = 1, \ldots, s$. If n is the number of faces in $star(\nu)$, then $r + s = n$. We can then expand Equation 3.2.4 into:

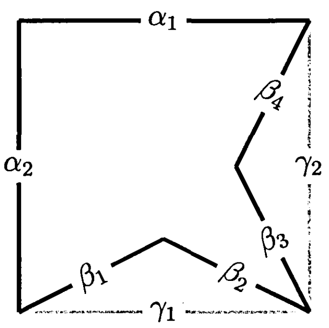
$$\hat{\omega} = 2\pi - \sum_{j=1}^{r} \alpha_j - \sum_{k=1}^{s} \beta_k. \tag{3.2.10}$$

Note that all the edges of $star^{+}(\nu)$ are also edges of $star(\nu)$, and if $star(\nu)$ is not convex, then its deviation from the convex star occurs at an edge of $star^{+}(\nu)$. We call such an edge a *deviation edge* (analogous to the *deviation vertex* presented in Section 2.3 for planar polygons). The deviation edges occur always in pairs. The dihedral angle at the *deviation edge* in $star(\nu)$ is sharper than the corresponding dihedral angle in $star^{+}(\nu)$.

First we analyse the case of vertices where the faces in $\mathrm{F}^{\mathrm{dev}}$ all have one edge that belongs to $star^{+}(\nu)$, as is the case of the vertex in Figure 3.5(a). For simplicity we draw the $link(\nu)$ from a top–down perspective, as in Figure 3.5(b). In it every edge corresponds to a face in $star(\nu)$, and edges are labelled with the angle of the face at ν.



(a) Mixed vertex



(b) Top–down view of the link

Figure 3.5: Mixed vertex where all faces have at least one *deviation edge*.

We can designate with Γ the faces of the convex hull that do not belong to $star(\nu)$. The angles of these faces are designated $\gamma_i, i = 1, \ldots, q$. Equation 3.2.10 is rewritten as:

$$\hat{\omega} = 2\pi - \sum_{j=1}^{r} \alpha_j - \sum_{i=1}^{q} \gamma_i + \sum_{i=1}^{q} \gamma_i - \sum_{k=1}^{s} \beta_k. \tag{3.2.11}$$

We can then split Equation 3.2.11 into two parts:

$$\hat{\omega} = 2\pi - \left(\sum_{j=1}^{r} \alpha_j + \sum_{i=1}^{q} \gamma_i\right) + \left(\sum_{i=1}^{q} \gamma_i - \sum_{k=1}^{s} \beta_k\right). \tag{3.2.12}$$

The first part corresponds to the positive curvature:

$$\omega^+ = 2\pi - \left(\sum_{j=1}^{r} \alpha_j + \sum_{i=1}^{q} \gamma_i\right), \tag{3.2.13}$$

and then the remaining part is the negative curvature:

$$-\omega^- = \left(\sum_{i=1}^{q} \gamma_i - \sum_{k=1}^{s} \beta_k\right). \tag{3.2.14}$$

The value of $(-\omega^-)$ will always be negative, since the sum of β will always be larger than the sum of γ. The minus sign before ω^- is used in order to keep its value positive. If the sums were equal, then the faces would be part of the convex hull.

As was shown in Section 2.3.1 for the curvature of planar polygons, the curvature of a vertex can be determined as the sum of the convex curvature plus the curvature of the deviations, which are convex regions themselves, but with opposite orientation. Using this principle we can deconstruct the stars of mixed vertices where not every face in $star(\nu)$ has deviation edges.

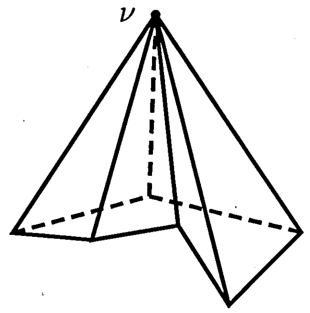
For each 'deviation' from the convex hull, we can define a second cone, consisting of the faces that belong to that single deviation (a subset of $\mathrm{F}^{\mathrm{dev}}$), and closed by one face that belongs to the $star^+(\nu)$ but not to $star(\nu)$. The new cone corresponds to a negative part of the curvature of ν. This new cone is not necessarily convex, but can be 'separated' into a number of convex cones, by inserting 'imaginary' faces inside of it. The set of imaginary faces is denoted with Γ', and the faces have angles $\gamma'_l, l = 1, \ldots, t$ incident on ν. The curvature of the deviation is measured by the sum of angles of the faces of each of the individual convex cones thus obtained. Such a vertex is illustrated in Figure 3.6. In this example, the negative curvature is measured as:

$$\omega^- = (\gamma'_1 - \beta_2 - \beta_3) + (\gamma_1 - \beta_1 - \gamma'_1). \tag{3.2.15}$$

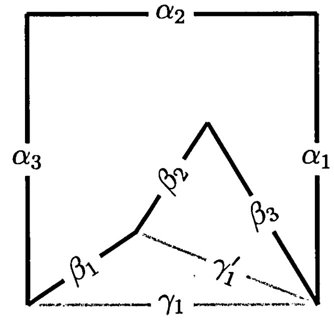
The *total absolute extrinsic curvature* $\hat{\Omega}$ is defined then as the sum of absolute extrinsic curvatures of all the n vertices of a polyhedral surface $\mathrm{P(V)}$:

$$\hat{\Omega} = \sum_{i=1}^{n} \hat{\omega}(\nu_i) = \sum_{i=1}^{n} \left(\omega^+(\nu_i) + \omega^-(\nu_i)\right) \tag{3.2.16}$$

The total absolute extrinsic curvature $\hat{\Omega}$ produces different values on the polyhedra that are depicted in Figure 3.3. It is equal to 4π on the right polyhedron (as it represents



(a) Mixed vertex



(b) Top–down view of the link

Figure 3.6: Mixed vertex where some faces do not have any *deviation edges*.

a convex body), and is greater than 4π on the left polyhedron, which is not convex. This example illustrates the fact that the total absolute extrinsic curvature of a polyhedral surface is an adequate analogue of Total Absolute Curvature of a smooth surface. The Total Absolute Curvature identifies hidden folds in the stars of the vertices that Angle Deficit ignores.

3.3 The Polyhedral Gauss Map

Separation of the positive and negative parts of the curvature for a mixed vertex can also be carried out using an analogue of the Gauss map for a polyhedral surface, which we call the *Polyhedral Gauss Map*. For a polyhedral surface curvatures are concentrated around vertices, so we need to be able to construct the PGM for an individual vertex, and the union of the Polyhedral Gauss Maps for all vertices determines the Total Absolute Curvature of a polyhedral surface. Assuming that the surface is oriented, we can construct an outward unit normal to any point of a face except at vertices and edges, and all these normals are parallel to one another. By translating them to the same origin, we get a unique unit vector. Applying the same procedure to each face of $star(\nu)$ we get a bundle of unit vectors. The endpoints of these vectors lie on the unit sphere. Without loss of generality, we assume that no two neighbouring faces lie in the same plane, then each endpoint corresponds to a face. By analogy with the smooth case seen in Section 3.2.2 we can make the following definitions:

Definition 3.3.1. The bundle of unit normals, corresponding to $star(\nu)$ is called the *normal star* of ν. The endpoints of the vectors in the normal star are joined, in order around the vertex, by geodesic arcs of the unit sphere. We will refer to the collection of these geodesic arcs as the *spherical indicatrix* of a vertex ν.

The construction of the spherical indicatrix from the normal vectors of a convex vertex is illustrated in Figure 3.7.

(a) Normal vectors of each face **(b) Vectors joined by geodesic arcs**

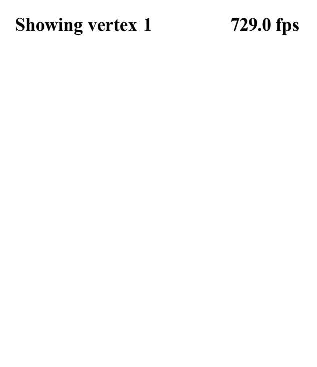
Figure 3.7: Construction of the spherical indicatrix of a convex vertex.

The spherical indicatrix can be also used to determine the Mean curvature H of a polyhedral surface P(V), which is a discrete analogue of the integral mean curvature for smooth surfaces. The Mean curvature H is determined along the edges, and for an edge *e, H(e)* is equal to (half) the oriented exterior angle *(3(e)* between the faces adjacent to **e.** The absolute value of *(3(e)* is equal to the length of the geodesic arc that connects two normals to the faces adjacent to **e.** This is true since the arcs lie on a unit sphere. In this research we are interested only in analogues of the Gaussian curvature, and therefore we only consider the Mean curvature H in special cases.

In the simplest cases the arcs of the PGM will not intersect and draw a single spherical polygon on the surface of the sphere. This is the case for flat, convex or simple saddle vertices, as defined in Section 3.2.3, which have only positive or negative curvature components, but not both. In the case of a mixed vertex, there will be self-intersections in the indicatrix. Figure 3.8 shows the spherical indicatrices with and without self-intersections.

The spherical indicatrix also delimits the area of one or more *spherical polygons.* The ordering of the vectors with respect to their corresponding faces is used to determine the sign of the spherical polygons. Figure 3.9 exemplifies this idea, showing a convex and a saddle vertex from a top-down view. When the order of the faces and normal vectors is the same (Figure 3.9(a)), we say the spherical polygon is positive. When the order of the faces and normals is reversed (Figure 3.9(b)), the curvature of the spherical polygon is negative. In the case of mixed vertices (Figure 3.9(c)), the concavities causes the normals of the faces to switch directions temporarily and turn Clockwise with respect to the vertex, opposite to the direction of their corresponding faces. When this occurs, two of the arcs intersect, signifying a change in the sign of the curvature. The intersection point is where

(a) Convex vertex



(b) Mixed vertex

Figure 3.8: Two kinds of vertices with their spherical indicatrices.

two separate spherical polygons of opposite sign meet. In mixed vertices this produces what we call 'hidden' saddles, or negative curvature components.

The sum of the areas of the spherical polygons is equal to the absolute curvature of the vertex. Using this sign we are not only able to separate *u(y)* for a vertex *v* into positive and negative components *u +(v)* and *co~(u),* but can also recognise individual parts of each sign, represented by individual spherical polygons. The *Polyhedral Gauss Map* of a vertex represents, therefore, a set of spherical polygons equipped with a positive or negative sign. The sum of the signed areas of the polygons is equal to the *discrete Gaussian curvature,* or *Angle Deficit,* of the vertex. The sum of the absolute areas represents the measure of *Total Absolute Curvature* associated with this vertex. Each spherical polygon of the negative sign represents a potential (hidden) saddle region.

The PGM visualisation of the basic vertex types is shown in Figure 3.10. The images present two different views, or scenes: the left scene shows the model of the vertex star and the right scene shows the spherical polygons of the Gauss Map. Positive areas in the PGM are shown in red, while negative areas are displayed in blue. The vertices in the meshes are coloured according to their vertex type as such: flat vertices are white, convex vertices are red, saddle vertices are blue and mixed vertices are green.

3.4 Analysis of the validity of the Polyhedral Gauss Map

This section will present the demonstration of the soundness of the PGM method for measurements of curvature. First a few concepts have to be introduced in order to give a proof of the validity of the computations obtained with the PGM. Then each different case of vertex is analysed in analogy with the concepts of the Total Absolute Curvature.

For every vertex *v,* we can compute the *vertex normal,* N„, as an average of the

Showing vertex 1
Curvature (PGM): 0.445364

(a) Faces and normals in the same direction (convex vertex)

(b) Faces and normals in reverse direction (saddle vertex)

(c) Normals changing direction at intersection point (mixed vertex)

Figure 3.9: Ordering of the faces and corresponding normal vectors around a vertex.

product of the normals of the faces in *star(is*) and their corresponding areas. If we call N, the normal vectors to the triangles G *Tv,* then Nj, is obtained as (Schroeder *et al.* 1992):

$$N_{,} = \frac{\sum N_i\, area(t_i)}{\sum_{i=1} area(t_i)}. \tag{3.4.1}$$

The vertices in *link(is)* are projected onto the plane *P0* defined by N„ that passes through *u.* The projectionwill draw a planar polygon calledthe *projected link* of *v.* For every vertex *iSj* G *link (is),* the procedure to project it on *P0,* following a line parallel to the vector N„ (see Figure 3.11 for reference), is as follows:

We can obtain the projected vertex *is'-* as a displacement of *ISJ* along the direction of the normal vector N^, adjusted by a scalar *L,* thus:

$$is' = iSj - LN_{,,}. \tag{3.4.2}$$

The value of *L* is determined as the projection of the vector along the edge (*isj — is)* on

» S e V? ® j : 0.000000

M « f ^ 0 .« > 3 6 4

(b) Convex

0556705

(c) Saddle

(d) Mixed

Figure 3.10: PGM of the basic types of vertices.

Figure 3.11: Projection of a neighbour vertex on a plane.

the vector N^. This is given by the dot product:

$$L = N_{,,} \bullet (isj - is). \tag{3.4.3}$$

Finally we can substitute Equation 3.4.3 in Equation 3.4.2 to get:

$$\nu_j' = \nu_j - (\mathrm{N}_\nu \cdot (\nu_j - \nu))\, \mathrm{N}_\nu. \tag{3.4.4}$$

When all the vertices in $link(\nu)$ have been projected to a plane, they can be joined by line segments that correspond to the edges in $link(\nu)$.

Definition 3.4.1. For a vertex ν we define its *projected link* as the planar polygon generated by the projection of the vertices and edges in $link(\nu)$, into the plane determined by ν and its normal vector N_ν.

For the analysis of the PGM we will make a further distinction of two types of vertices, depending on the location of a vertex ν within its convex star $star^+(\nu)$.

Definition 3.4.2. If ν belongs to the convex hull of $star(\nu)$, then it is said to be a *cone–star vertex*. Otherwise, if ν lies inside of the convex hull, then it will be a *saddle–star vertex*. This classification is know as the *star type* of the vertex.

For the following discussion we will consider that vertices of the basic types: flat, convex and mixed fall into the *cone–star* type. The various possible saddle vertices are of the *saddle–star* type.

We now have the necessary tools to present the main theorem of this chapter, regarding the validity of the Polyhedral Gauss Map as a measure of discrete curvature.

Theorem 3.4.1. *The* absolute curvature $\hat{\omega}(\nu)$ *of a vertex is determined from the Polyhedral Gauss Map as the sum of the total area of the closed spherical polygon(s) inscribed on the surface of the sphere by the arcs of the indicatrix.*

Proof: Given that the curvature is computed as the sum of separate areas, we can say that it is an additive property, meaning that it can be seen as the sum of the curvatures of individual features of a vertex star. This property will be used to construct the proofs of Theorem 3.4.1 for increasingly complex vertices. We split the proof of the PGM for cone–star vertices (case 1) and saddle–star vertices (case 2), as follows:

1. **PGM of cone–star vertices**

   (a) **Flat vertices:** All the faces around the vertex have the same normal vector, which maps to a single point on the sphere, as shown in Figure 3.10(a). This results in an area, and a curvature, equal to zero.

   Proof: When all the faces in $star(\nu)$ lie on the same plane, the sum of their angles $\sum \alpha$ is equal to 2π, making the curvature $\omega(\nu) = 2\pi - \sum \alpha = 0$. ■

(b) **Convex vertices:** If $star(\nu)$ is convex then the area of the spherical polygon (cut out by the normal star of ν in the unit sphere, as in Figure 3.10(b)) gives the measure of curvature around ν, equal to $\omega(\nu)$.

Proof: This is known from the theory of convex polyhedra (Aleksandrov 2005). ■

(c) **Mixed vertices:** If $star(\nu)$ is the star of a mixed vertex then the PGM of ν is not a convex spherical polygon. Moreover the spherical indicatrix has self–intersections, partitioning the PGM into several simple polygons, possibly overlapping. However, the *convex normal star*, *i.e.* the normal star of $star^+(\nu)$, will give us the measure of the positive curvature ω^+. This is in conformity with the smooth case.

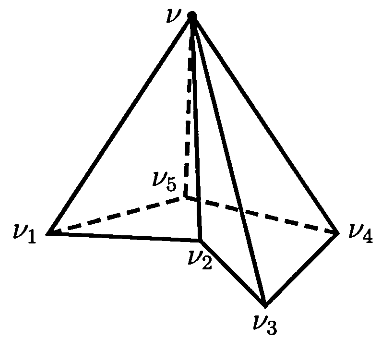
Indeed, if ν is a simple mixed vertex then the areas of the PGM that represent the parts of negative curvature do not overlap with the area of positive curvature. This can be seen in Figure 3.10(d). The area of the positive curvature in this case represents a convex simple polygon, which is unique.

Therefore in the PGM, for all the faces that do not belong to $star^+(\nu)$ but are incident on a deviation edge, their normals will lie outside the spherical polygon formed by the convex normal star. The intersection points of the spherical indicatrix, which separate the positive area of the PGM from negative parts, correspond to those faces of $star^+(\nu)$ that are not faces of $star(\nu)$.

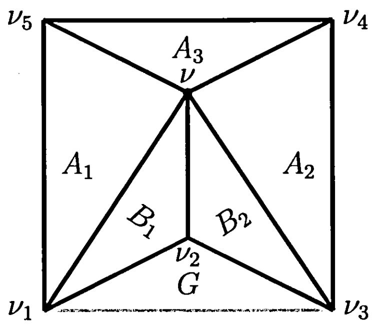
Proof: We will use the vertex from Figure 3.12(a). The method for this example can be used to analyse more complex mixed vertices. In Figure 3.12(b) the faces are labelled around the vertex. Faces $A_1, A_2, A_3 \in \mathbf{F}^+$ have normals $\mathbf{n_{A_1}}$, $\mathbf{n_{A_2}}$ and $\mathbf{n_{A_3}}$; $B_1, B_2 \in \mathbf{F}^{\mathrm{dev}}$ have normals $\mathbf{n_{B_1}}$ and $\mathbf{n_{B_2}}$; and $G \in \mathbf{\Gamma}$ with normal $\mathbf{n_G}$. In the figure, the face G is delimited by vertices ν, ν_1 and ν_3. The deviation edges are $\overline{\nu\nu_1}$ and $\overline{\nu\nu_3}$.

We can imagine a rotation of the face B_1 around the edge $\overline{\nu\nu_1}$. The rotation goes from 0 radians, when of $\mathbf{n_{A_1}}$ and $\mathbf{n_{B_1}}$ are pointing in opposite directions, until a maximum rotation of π when the normals of both faces coincide and are equal. Throughout this rotation $\mathbf{n_{A_1}}$ and $\mathbf{n_{B_1}}$ will always lie in the same plane, defining an arc $(\mathbf{n_{A_1}}, \mathbf{n_{B_1}})$. The normal vector $\mathbf{n_G}$ also lies in this plane, at the point where the rotation of B_1 makes it coincide with G.

The same rotation can be imagined for B_2 around the edge $\overline{\nu\nu_3}$. Again the normal vector $\mathbf{n_G}$ will lie in the arc defined by $\mathbf{n_{A_2}}$ and $\mathbf{n_{B_2}}$. Thus the vector $\mathbf{n_G}$ belongs to the intersection of the arcs $(\mathbf{n_{A_1}}, \mathbf{n_{B_1}})$ and $(\mathbf{n_{A_2}}, \mathbf{n_{B_2}})$.



(a) Mixed vertex



(b) Top–down view of the vertex

Figure 3.12: Mixed vertex and top–down view showing the faces and vertices.

So we find that the indicatrix of the vertex includes the normal vectors of $star^+(\nu)$, thus providing $\omega^+(\nu)$ from Equation 3.2.13.

Next we demonstrate that the remaining vectors draw a spherical polygon whose area is equal to $\omega^-(\nu)$. We imagine an extension of the edge $\overline{\nu\nu_2}$ in the direction opposite to ν. Let us insert a new vertex ν_2' in this new edge. We can create two new imaginary faces $B_1' = (\nu, \nu_2', \nu_1)$ and $B_2' = (\nu, \nu_3, \nu_2')$, where $\mathbf{n_{B_1}} = \mathbf{n_{B_1'}}$ and $\mathbf{n_{B_2}} = \mathbf{n_{B_2'}}$, as in Figure 3.13. The faces B_1', B_2' and G form a new convex cone, with correctly oriented normals that are ordered Clockwise around ν. We call this cone the *dual convex cone of a saddle.*

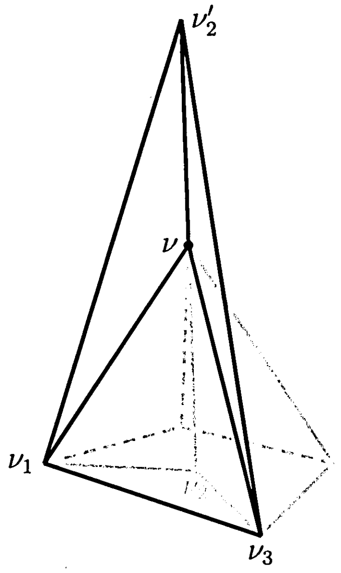


Figure 3.13: Convex cone representing the negative part of the curvature of a mixed vertex.

The curvature of this cone is:

$$\omega = 2\pi - \beta_1' - \beta_2' - \gamma. \tag{3.4.5}$$

But in this case $\beta_1' = \pi - \beta_1$ and similarly $\beta_2' = \pi - \beta_2$. This gives us:

$$\begin{aligned} \omega &= 2\pi - \pi + \beta_1 - \pi + \beta_2 - \gamma \\ &= \beta_1 + \beta_2 - \gamma. \end{aligned} \tag{3.4.6}$$

Since the order of the normals around the cone is Clockwise, the curvature is identified as negative, resulting in:

$$-\omega = \gamma - \beta_1 - \beta_2. \tag{3.4.7}$$

This is equal to the result obtained in Equation 3.2.14.

Some mixed vertices will yield negative spherical polygons that are non–convex (for example, the vertex in Figure 3.6). These spherical polygons can be subdivided into convex spherical polygons (not necessarily triangles), where each of them corresponds to a 'hidden' saddle part of the vertex. In this case the total negative curvature is the sum of the smaller parts, similar to the example in Equation 3.2.15. ■

(d) **Cone–star vertices with self–intersections:** We will sketch a proof of the curvature of non–manifold vertices with self intersections. This can be justified by the following considerations of possible vertex links $link(\nu)$: Imagine the polygon generated by the projected link of ν. The projected link of a mixed vertex with no self–intersections has 'concavities' at the vertices that do not belong to the convex hull of the $star(\nu)$. If the vertices in the concavity are translated, a self–intersection can be produced, but the orientation of the normals is maintained with respect to the vertex. For example, the vertex ν_2 in the link of the cone in Figure 3.12 can be moved in such a way that it will take a new position ν_2' on the other side of the edge $\overline{\nu_4\nu_5}$ but all other vertices will keep their positions. We then get the vertex in Figure 3.14. This will cause two face intersections, defining two *intersection lines*. We call such type of self–intersections *removable*.

Vertices with self–intersections can be considered in the opposite way. If their star presents two intersection lines, they can be transformed into vertices without intersections by moving the vertices in their link. If the transformed part of the link of the vertex remains isotopic to the original part, then this operation is valid, and the computation of the curvature is the same as for mixed vertices. The result is a larger negative component to the curvature, as shown in Figure 3.15.

^3

(a) Normal mixed vertex

(b) Mixed vertex with self-intersection

Figure 3.14: Self-intersecting mixed vertex and top-down view showing the vertices.

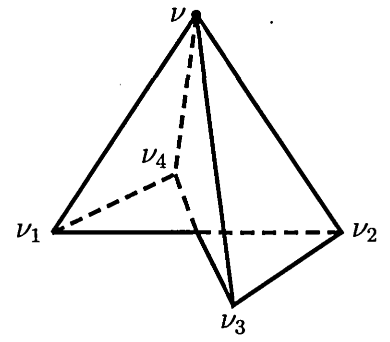
l: 3.237744

€

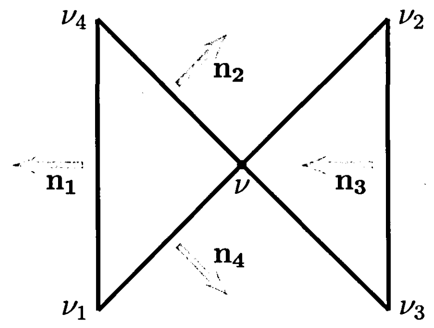
Figure 3.15: Mixed vertex with self-intersection.

Nevertheless, when there is only one single intersection line it is impossible to move the vertices in the link without producing a singularity, known as a 'beak', where the property of isotopy is not preserved. Such types of self-intersections are called *non-removcible.* This fact is known from the theory of homotopy, but the exact proof for it is outside of the scope of this thesis. This condition occurs in the vertices known as *'figure-8'*. They receive that name because of the shape of their projected link, illustrated in Figure 3.16. In this example, if vertex *u2* were to be moved below the edge *VfiTl* to remove the self-intersection, the normals would loose consistency.

Imagine a walk around the link of a 'figure-8' vertex. The orientation of the normals is always consistent with that of the faces, suggesting a positive curvature. When it is no longer possible to remove intersections by moving vertices, the cone of the vertex can be separated into two simple cones along the single intersection line. Both of the convex cones will have positive curvature, and the sum of both curvatures is equal to the one for the 'figure-8' vertex.



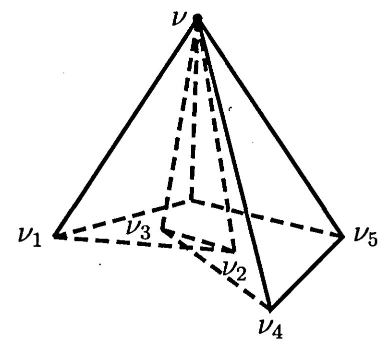
(a) *Figure–8* vertex



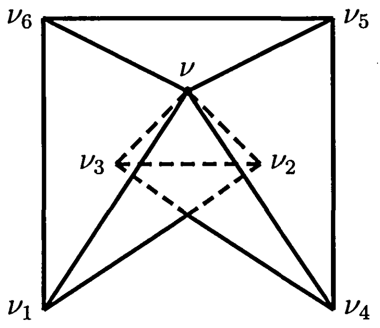
(b) Top–down view showing the face normals

Figure 3.16: A 'figure–8' vertex.

Another case when this occurs is what we call a *'pinch vertex'*, shown in Figure 3.17. The projected link of the vertex turns twice around the vertex, generating two loops. In the PGM each loop will produce a spherical polygon of positive orientation, and the two polygons will overlap.



(a) *Figure–8* vertex



(b) Top–down view showing the face normals

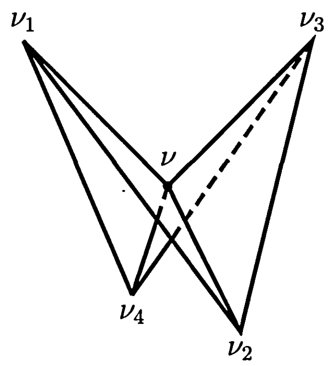
Figure 3.17: A 'pinch' vertex.

Any mixed vertex with an arbitrary number of self–intersections can be seen as a combination of the previous cases. Theoretically a mixed vertex ν could be simplified, by moving the vertices in $link(\nu)$, to remove the self–intersections that cause two intersection lines, until the link of the vertex is shaped as a 'figure–8', a 'pinch', a circle (the case of a vertex without self–intersections) or the union of these. The PGM of any cone–star vertex with self–intersections can be produced by decomposing the vertex in this way.

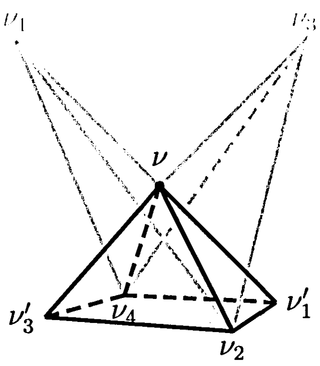
2. **PGM of saddle–star vertices**

   (a) **Basic saddle vertices:** The curvature of basic saddle vertices is found in a similar way as for convex vertices. For basic saddles the spherical polygon will also be convex, as is the case in Figure 3.10(c), but in this case the orientation of the normals will be inverted, producing a negative curvature.

Proof: The same case as for the negative part of mixed vertices can be applied to saddle vertices. Figure 3.18(a) shows a basic saddle. In Figure 3.18 the edges $\overline{\nu\nu_1}$ and $\overline{\nu\nu_3}$ have been extended to generate the vertices ν_1' and ν_3'. As before, the new faces form the dual convex cone of the saddle. The normal vectors of the faces remain the same as in the saddle, but the direction of the turn of the normals around the vertex is opposite to that of the faces, thus giving a negative sign to the curvature. ■



(a) Saddle vertex



(b) 'Dual' convex vertex

Figure 3.18: Extension of the faces in a saddle vertex to create its *dual convex cone.*

For more complex saddle vertices with non–convex spherical polygons, the same procedure can be followed to group and extend the faces of $star(\nu)$, so that the PGM is split into convex spherical polygons. Again the total negative curvature is the sum of the areas of the convex spherical polygons, in a similar way as in the example of Equation 3.2.15.

Some saddles–star vertices will also present self intersections in their indicatrix, either removable or not. Vertices with non–removable self–intersections have a positive curvature component, and can be treated the same way as mixed vertices.

(b) **Non–basic saddle vertices:** More complex saddle–star vertices, such as the monkey–saddles, present several overlapping areas of negative curvature. These can be completely or only partially overlapping. When a monkey–saddle has pairs of non–adjacent coplanar faces, their normals will produce a loop in the spherical indicatrix. These loops need to be identified to extract the two independent and overlapping spherical polygons. The area of each of these polygons can be evaluated in the same manner as for basic saddles.

(c) **Saddle–star vertices with non–removable self–intersections:** In these vertices the complexity of the PGM may cause several spherical polygons to overlap. If the signs of these polygons are opposite, the areas may cancel each other and disappear from the PGM. As in the case of cone–star vertex, these vertices can be split along the line of the self intersection. The sum of the curvatures of the new split vertices will be equal to the total curvature of the original vertex.

This is the end of the general proof of Theorem 3.4.1. ■

3.5 Conclusions

This chapter has presented the methods to measure the curvature of discrete surfaces. The method of Angle Deficit has been shown as an analogue of the Gaussian curvature ω at a vertex. We have further expanded the concepts of Total Absolute Curvature for a vertex by defining the positive and negative components of curvature. The positive curvature ω^+ of a vertex is measured from the convex cone around its star, and then the total negative curvature ω^- is the difference between ω and ω^+.

Using the concepts of positive and negative curvature we can characterise four basic types of vertices: Flat vertices have curvature equal to zero. Convex vertices have only positive curvature. Saddle vertices have only negative curvature. An additional type of vertices, denominated 'mixed', combine both components of curvature. Identification of these vertices is not possible using existing methods based on the discrete case, due to the fact that mixed vertices do not exist in smooth surfaces.

The discrete version of the Gauss Map has been shown as a method to measure both the positive and negative components of the curvature of a vertex, thus providing a complete representation of the Total Absolute Curvature of the vertex. The main contribution of this chapter is the demonstration of the validity of the curvature computed by the Polyhedral Gauss Map, demonstrated under the different types of vertices previously mentioned. Using the Polyhedral Gauss Map we can clearly characterise all types of vertices, not possible with other methods to measure discrete curvatures, such as the Angle Deficit. This includes vertices with self–intersections.

From the experiments performed with a variety of vertices it is possible draw the following conclusions on the properties of the Polyhedral Gauss Map for discrete surfaces:

- The positive spherical polygons should be always convex. In the special case of a 'figure–8' vertex, its PGM seems to have one large non–convex spherical polygon. This however can be described as two convex spherical polygons that appear to be glued together. This is demonstrated by splitting the 'figure–8' vertex along the intersection of the faces to form two convex cones. The sum of the PGM of both cones will be equal to the PGM of the single 'figure–8' vertex. Considering these cases, all positive spherical polygons will have zero or two concave angles.

- The cone–star vertices can be segregated according to the shape of their projected link as: *circle*, *'pinch'* or *'figure–8'* shapes. The projected link of all cone–star vertices can be developed into one of these kinds by unwinding them. However, it is impossible to transform a 'figure–8' or a 'pinch' vertex into a vertex without self–intersections, without producing a 'beak' type singularity.
- The Polyhedral Gauss Map of a vertex with self–intersecting faces should be equal to the sum of the Gauss Maps of the individual cones obtained by splitting the star of the vertex along the intersections.

From the results obtained in this chapter we can conclude that the method of the Polyhedral Gauss Map is a valid and correct measure of the curvature of a vertex in a discrete surface. We have also shown that it can identify hidden features in the shape of the star of the vertex.

Chapter 4

Computational implementation of the Polyhedral Gauss Map

4.1 Introduction

This chapter presents the computational algorithms and methods developed to determine the Polyhedral Gauss Maps of vertices in a polygonal mesh. The method is based on the analysis of each individual vertex of the mesh, and the normal vectors of the faces in its star. The Total Absolute Curvature of the object can be determined as the sum of the curvature values of all vertices.

Existing applications that make use of the Gauss Map to compute the curvature of a discrete object are limited to dealing with vertices that lie on a cone or a saddle (Grinspun and Schröder 2001), (Yamauchi *et al.* 2005). In these papers, other types of vertices are either assumed to be impossible to correctly analyse or simply ignored. This excludes the vast majority of the possible vertex configurations, and is ultimately not more useful than the traditional method to find the curvature via Angle Deficit. There are also some theoretical works dealing with more complex vertices such as 'figure–8' (Rodríguez and Rosenberg 2000), but they do not use the Gauss map in a classical way, but transforms it in order to get certain results in the rigidity theory of polyhedra.

The procedure to compute the Polyhedral Gauss Map proposed here aims to analyse any kind of vertex and to construct unambiguous Gauss maps for polyhedral surfaces of complex shape, of various genera, self–intersecting and even non–manifold surfaces. We construct the Gauss Map in the classical sense, by correctly identifying each normal with respect to a chosen orientation, and preserving the adjacency of faces by connecting the corresponding normals by a geodesic on the sphere. The algorithm is relatively straightforward for simple embedded vertices, but a sound computation of more complex vertices requires thorough analysis of the geometry of a vertex as well as intricate computational techniques. These techniques constitute the main novel contributions brought forward in this chapter. They deal mostly with the projection of the normal vectors onto a sphere, and the correct identification of the individual areas that compose the Polyhedral Gauss Map of a vertex.

The contributions are mentioned here for reference, and will be explained later in the chapter:

- Preservation of the ordering of the normal vectors: The order of the faces and corresponding vectors around a vertex provide the orientation for the Polyhedral Gauss Map. The sequence of faces around a vertex is represented in the PGM by connecting vectors with geodesic arcs.
- Detection of arc intersections: Several geodesic segments on the surface of the sphere must be checked for intersections with the other arcs. These intersections can happen at various points along the arcs, and all cases must be dealt with accordingly.
- Splitting of the spherical polygons: Determining which of the arc intersections are actually the meeting point of two spherical polygons, or whether they are points where spherical polygons overlap. Determining where the overlapping may cause positive and negative areas to cancel each other.
- Identification of the orientation of the polygons: Each spherical polygon obtained splits the sphere in two regions, and can be interpreted in two ways, either a positive area, or its negative complement. Several parameters are tested to determine the correct orientation, based on the properties observed for each kind of area. This is complicated by positive and negative regions that overlap and can even cancel each other.
- Classification of vertices according to the signed components of their PGM, into flat, convex, mixed or saddle.

From this point onward the focus is on the procedures and applications of the Polyhedral Gauss Map on discrete polygonal meshes. To simplify the explanations, the term Gauss Map will refer to the Polyhedral Gauss Map, abbreviated as the PGM, unless stated otherwise. The term Gaussian curvature will be used to represent the discrete analogue of the integral Gaussian curvature.

4.2 Overview of the `gaussMap` program

The `gaussMap` program was developed for the purpose of studying the Polyhedral Gauss Map of various types of vertices. It receives as an input a polygonal mesh, and produces a Polyhedral Gauss Map for all vertices. It receives as a parameter the name of the file that stores the mesh information. A second optional parameter is a vertex number. If present, the program will analyse that single vertex and ignore the rest.

The polygonal mesh used represents an orientable surface in three–dimensional space, and therefore we can determine a coherent orientation on the whole mesh (Alboul 2003). We have chosen to use meshes with a Counter–Clockwise orientation. That is, the vertices that define each face are ordered in the Counter–Clockwise direction around the face. Such faces will have a normal vector that points outwards from the mesh. This direction is considered to be the positive direction of the normal vector.

4.2.1 Data input

The information about the polygonal mesh is read from files in the *Wavefront OBJ* format (Murray and VanRyper 1996). This format contains the description of an object in three–dimensional space in terms of vertices and faces. Optional details such as normal vectors, texture coordinates and object materials can also be included in the files but are disregarded by our implementation.

The *OBJ* file format requires no header information. In its most basic form the description of an object is done in two sections:

- The first section specifies the vertex coordinates. Each line has the descriptor character "v" to signify a vertex, followed by the 3 coordinates of the vertex.
- Faces are indicated with the descriptor "f", and the indices of the vertices that compose the face. The vertex indices are numbered according to the order in which they are defined before, beginning at index number 1.

The following code listing shows a sample *OBJ* file describing a cube made up of 12 triangles:

```
# 8 vertices
v -1.0 -1.0  1.0
v  1.0 -1.0  1.0
v -1.0  1.0  1.0
v  1.0  1.0  1.0
v -1.0  1.0 -1.0
v  1.0  1.0 -1.0
v -1.0 -1.0 -1.0
v  1.0 -1.0 -1.0

# 12 faces
f 1 2 4
f 1 4 3
f 3 4 6
```

```
f 3 6 5
f 5 6 8
f 5 8 7
f 7 8 2
f 7 2 1
f 2 8 6
f 2 6 4
f 7 1 3
f 7 3 5
```

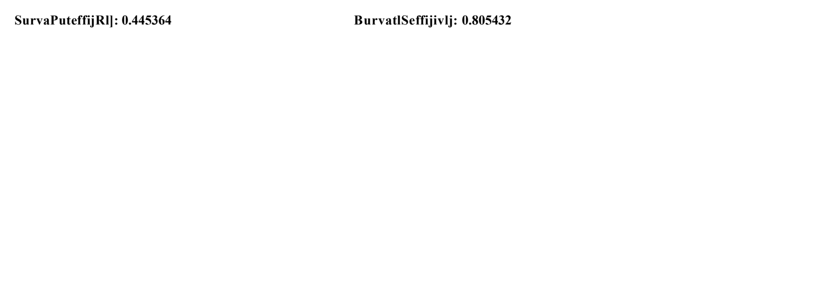
Vertex coordinates are read from the input file and stored temporarily in a linked list. After reading all the vertices from the file, the list is converted into an array for storage in the object structure. Similarly the faces of the object are read and stored in a list during the parsing of the file and then transferred into an array. The following information is also extracted during the parsing of the input file: For every vertex a linked list is created with the faces incident on it. The number of incident faces on each vertex is also stored in an array. This effectively represents the star of each vertex. Finally, the normal vector for every face is computed as the cross product of two of its edges. After parsing the input file the procedure to compute the Gauss Map is carried out individually for every vertex in the polygonal mesh. The whole process is done in stages, obtaining and storing new data at every step.

4.2.2 Program output

The `gaussMap` program generates the PGM of all the vertices in the mesh or of the single selected vertex. The results are shown in a graphical interface, and the numerical computations of the curvature are also printed on the screen.

The curvature measures of a vertex are obtained as the areas of the spherical polygons in its PGM. In the simplest case, the spherical indicatrix presents no self–intersections, and a single spherical polygon determines the curvature of a vertex, as in the case of convex (Figure 4.1(a)) and simple saddle (Figure 4.1(b)) vertices. Mixed vertices present intersection of the arcs in the indicatrix, producing more than one spherical polygon, as shown in Figure 4.1(c).

The numerical measures of the curvature are extracted from the areas of all the spherical polygons and printed on screen. The absolute curvature is obtained as the sum of the positive and negative components. Gaussian curvature is obtained using the Angle Deficit method, and is equal to the positive curvature minus the negative one. These computations are printed on the console. The absolute curvature of the vertex is also shown on the graphical visualisation of the PGM. Figure 4.2 shows the example of a vertex whose PGM



(a) Convex **(b) Saddle** **(c) Mixed**

Figure 4.1: Simple vertices and their spherical indicatrices.

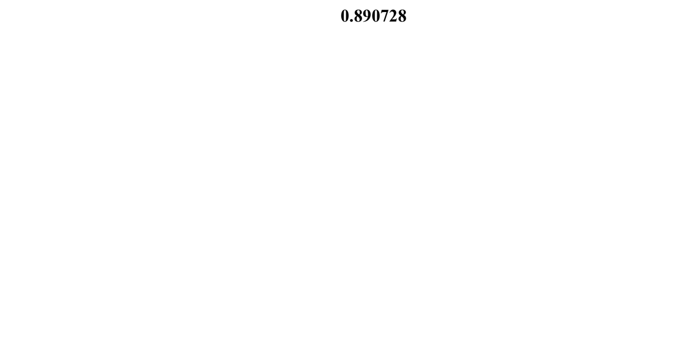


Figure 4.2: A vertex with positive and negative areas in its Gauss Map.

has various areas of both positive and negative sign. The `gaussMap` program presents the following measures of curvature for the selected vertex:

```
ABSOLUTE CURVATURE      0.890728
POSITIVE CURVATURE      0.445364
NEGATIVE CURVATURE      0.445364
GAUSSIAN CURVATURE      0.000000
```

This particular example showcases the advantage of the PGM over the Angle Deficit methods. According to the AD computation the Gaussian curvature of the vertex is zero, and hence the vertex is classified as flat. The `gaussMap` program gives a complete characterisation of any vertex v. It identifies all folds in $slar\{y)$, thus emphasising 'hidden' curvature regions. It can analyse non-manifold vertices with intersecting faces.

4.2.3 Algorithm to compute the Polyhedral Gauss Map

The process of obtaining the Polyhedral Gauss Map for a single vertex v is summarised as follows:

- The faces in $star(v)$ are visited in the Counter-Clockwise direction around v. Adjacent normal vectors that are equal are discarded. The remaining normal vectors are stored in a list, in the same order as the faces are visited.

- All the normals are translated to the same origin, forming the *normal star* of the vertex. Their endpoints will lie on the surface of a unit sphere.
- The endpoints of the normals are joined together, respecting their order in the list. The shortest line between two endpoints is a segment (arc) of a great circle of the sphere.
- The arcs thus defined form the *spherical indicatrix*, and will delimit an area on the surface of the sphere, which is composed of one or more spherical polygons.
- If the spherical indicatrix is self–intersecting, then there are more than one spherical polygon. In this case, new vectors must be added, that will have their end–points at the intersection points of the arcs. The set of the face normal vectors plus these new intersection vectors is called the *extended normal star* of the vertex.
- The new vectors identify points where curvature components of different sign meet on the Polyhedral Gauss Map. These intersection vectors are shared by two or more spherical polygons.
- The orientation of each of the spherical polygons is determined by following the corresponding loop of the spherical indicatrix, in the order already established for the normals (by visiting the faces around the vertex in Counter–Clockwise order). The orientation will be positive if the walk along the loop is Counter–Clockwise (CCW), and negative if the loop is Clockwise (CW). Accordingly, the area of a polygon of the positive orientation is considered as positive, and of the negative orientation as negative.
- The PGM of a single vertex can have all of its areas positive, or all negative, or have spherical polygons with both orientations. The vertex is then classified as positive, negative or mixed, respectively. A flat vertex will present a PGM with only one point on the sphere, and area equal to zero.
- The area of the Gauss Map is computed as a sum of the areas of the individual spherical polygons. For simple manifold vertices, the area of each individual spherical polygon will be less than 2π. On non–manifold vertices with self–intersecting faces the polygons are more complex and their areas can be larger than 2π.

Algorithm 4.1 shows the pseudo–code representation of the main program to compute the PGM of a mesh.

Algorithm 4.1 Polyhedral Gauss Map.

```
For each vertex $\nu \in$ V
{
    face_list ← Get list of faces in $star(\nu)$.
    Order face_list in CCW direction around $\nu$.
    normal_list ← List of the normals from face_list.
    extended_normal_list ← Find arc intersections in normal_list.
    If there are any arc intersections, then
    {
        polygon_list ← Divide into areas using extended_normal_list.
    }
    Else check for the special case of the 'monkey saddle'
    {
        polygon_list ← Check for duplicate normals in normal_list.
    }
    polygon_count = number of polygons in polygon_list.
    Determine the sign of the polygons in polygon_list.
    For each spherical polygon $P_i \in$ polygon_list
    {
        If $P_i$ is positive, then
        {
            Positive_Curvature ← Positive_Curvature + $area(P_i)$.
            positive_count ← positive_count + 1.
        }
        If $P_i$ is negative, then
        {
            Negative_Curvature ← Negative_Curvature + $area(P_i)$.
            negative_count ← negative_count + 1.
        }
    }
    Classify the type of vertex:
        If polygon_count = 0, then
            Vertex $\nu$ is flat.
        Else If polygon_count = positive_count, then
            Vertex $\nu$ is positive.
        Else If polygon_count = negative_count, then
            Vertex $\nu$ is negative.
        Else
            Vertex $\nu$ is mixed.
    Absolute_Curvature ← Positive_Curvature + Negative_Curvature.
    Gaussian_Curvature ← Positive_Curvature - Negative_Curvature.
}
```

4.3 Step–by–step explanation of the algorithm

This section describes in detail the main steps of the Polyhedral Gauss Map algorithm, and the techniques used for its programming implementation in the *C programming language*.

4.3.1 Face ordering around ν

For every vertex ν we define T_ν as the list of the n triangles in $star(\nu)$, with $\mathrm{T}_\nu = \{t_i; i = 1, 2, \ldots, n\}$ The initial step in analysing ν is sorting the faces in T_ν in the Counter–Clockwise order. The faces are ordered so that the vertices in $link(\nu)$ are in a Counter–Clockwise sequence with respect to ν. This step is of paramount importance to obtain the correct representation of the Gauss Map of the vertex, because the algorithm assumes a Counter–Clockwise ordering to determine the orientation of the spherical polygons. When all faces are in order, $star(\nu)$ is updated to reflect the changes. Simultaneously with the ordering of the faces, we compute the Gaussian curvature of the vertex, by adding together the angles α_i incident on ν of every face $t_i \in \mathrm{T}_\nu$, and using the Angle Deficit formula:

$$\omega = 2\pi - \sum_{i=1}^{n} \alpha_i. \tag{4.3.1}$$

In the case of a vertex lying on the boundary of an open mesh, the formula is modified to be (Brehm and Kühnel 1982):

$$\omega = \pi - \sum_{i=1}^{n} \alpha_i. \tag{4.3.2}$$

4.3.2 Normal star and spherical indicatrix

The *normal star* of a vertex ν consists of all the normal vectors N_i of the faces in $star(\nu)$. We create a list λ_N to store these normal vectors, respecting the order of the corresponding faces. The *spherical indicatrix* is constructed from the list of vectors. However it is necessary to eliminate some of the normal vectors in order to obtain a correct PGM. The vectors are discarded on the following conditions:

- If two adjacent faces are nearly coplanar with nearly equal normals, one of the normals will be eliminated, within a certain tolerance. The selection of this threshold is of importance since vectors that are very close together may produce ambiguity at later stages when detecting arc intersections. We define an allowed difference ϵ between two vectors. If all three of the coordinates of the first vector are within a range $[-\epsilon, \epsilon]$ from the coordinates of the second vector, both of them are considered to be equal. This reduces slightly the precision of the Gauss Map measurements, but makes the computation more robust in the presence of noise.

- Another case where normal vectors are dropped from the list λ_N is when three or more adjacent vectors lie in the same plane. This would result in two or more arcs lying along the same great circle, complicating the detection of arc intersections. Figure 4.3 shows this case, using a planar representation of the spherical indicatrix. In the figure, the vertices of the spherical indicatrix are the endpoints of the face normal vectors, and are denoted as the corresponding normal vectors (*i.e.* $\mathbf{n}_1$).

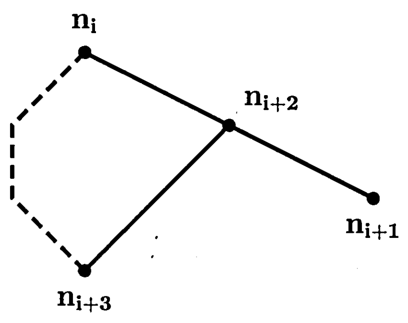


Figure 4.3: Special case of three vectors forming a line in the spherical indicatrix.

The *spherical indicatrix* of ν is constructed by joining the ordered normal vectors by geodesic arcs. Every pair of vectors lies along a great circle of the sphere, so there will be two possible geodesic arcs to join them. It is not possible to have an angle larger than π between two faces. Because of this, in all cases the smallest of the two arcs between the normal vectors is chosen.

A special case occurs if $star(\nu)$ includes two adjacent faces that are coplanar but opposite to each other, that is, the normal vectors of the faces are separated by an angle of π radians, and the endpoints of the normals are antipodal to each other. In this case, there is an infinite number of arcs of the same length that can connect the two vectors, but only one correctly represents the shape of the vertex. To clarify the direction of the arc, we add an extra vector, normal to the edge that connects the two faces, and that lies in the same plane as these faces. This extra vector will single out the direction of the arc that joins the two normal vectors. The example shown in Figure 4.4 requires two extra vectors, one for each edge of this degenerated star, to determine the arcs to join the face normals.

4.3.3 Detection of arc intersections

Except in the cases of perfect cone or basic saddle vertices, there will be intersections of some of the arcs in the spherical indicatrix of the vertex. The procedure to determine whether two arcs intersect is the following:

Every arc is a segment of a great circle, and in turn these great circles are defined

3.141593

Figure 4.4: Gauss map of a vertex with two coplanar faces.

by the intersection of the sphere with a plane that passes through the origin of the sphere. Thus two arcs will only intersect at the location where their defining planes and the sphere intersect. The intersection of the planes determines an *intersection line*, and the two *antipodal points* where this line crosses the sphere are the only places where an intersection of the arcs can occur.We have to note that, since the arcsare limited to a length of 7r, two of them canonly intersectonce on the sphere, at one of the antipodal points.

Given two arcs defined by the vectors at its endpoints: $A = (n_k, n_{k+i})$ and $B = (n_{k+h}, n_{k+h+i})$ (as shown in Figure 4.5), we obtain the vectors perpendicular (normal) to each arc, by computing the cross product of the two endpoint vectors:

$$n_A = n_k \times n_{k+i} \tag{4.3.3}$$

and

$$n_B = n_{k+h} \times n_{k+h+i} - \tag{4.3.4}$$

Figure 4.5: Intersection of two arcs defined by the normal vectors at their endpoints.

The cross product of the arc normals determines the *intersection vector.* We assign one such vector to each arc, thus having:

$$I_A = I_B = n_A \times n_B, \tag{4.3.5}$$

where iA and iB are the intersection vectors for arcs *A* and *B,* respectively. These will be used to determine the correct antipodal point. Figure 4.6 shows a diagram with the planes defined by the arcs, and the line of intersection that determines the two antipodal points on the sphere.

Figure 4.6: Intersection of two arcs. The planes defined by the arcs are shown. The vectors, the arc and the plane use the same colour. The *intersection line* **is shown in red.**

If the intersection vectors are equal to zero, then the arcs lie in the same plane. There should be no intersections caused by coplanar arcs, since this case is eliminated when creating the list of vectors.

If the intersection vector of an arc is equal to any of the endpoint vectors of the corresponding arc, then it must be handled as a special case. The binary variable `endpoint` is used to keep track of which endpoints are intersecting. The `endpoint` variable is treated as a binary number of four bits, all of them initially equal to zero. Each of the bits represents one of the vectors of the arcs, as represented in Figure 4.7. If the intersection vector is equal to any of the vectors, the corresponding bit is set to 1. At most two of the bits will be equal to 1, since the intersection vector cannot be equal to the two vectors in the same arc.

23	22	21	2°
nk+h+l	Hk+h	nk+l	nk

Figure 4.7: **The** `endpoint` **variable as a four bit number, representing the four vectors involved in an arc intersection.**

If `endpoint` is still equal to zero, then we need to determine if the intersection vectors lie within the arcs. If the intersection vectors lie outside of the arcs, then the arcs are not intersecting each other.See Figure 4.8 for reference.For each arc,we compute the cross products of the endpointvectors and the intersection vector. That is, for arc *A* we compute the vectors:

$$Ci — nk \times iA \tag{4.3.6}$$

and

$$c2 = iA \times nk{+}i. \tag{4.3.7}$$

If ci = c2 then the intersection vector iA lies within the segment of the great circle covered by the arc *A*.

Figure 4.8: Arcs that do not intersect. In this case the intersection line is outside of at least one of the arcs.

It is possible that the intersection vector is pointing in the opposite direction of the arc. We use the dot products of the intersection vector and the endpoint vectors to identify if iA points in the correct direction. The integer value *d,* computed as:

$$d = (iA \bullet nk) + (iA \bullet nk{+}i), \tag{4.3.8}$$

will determine the direction of the intersection vector with respect to the arc. If *d* is negative, then iA must be inverted. Otherwise the intersection vector is already in the correct direction.

The same procedure is done for arc *B* to determine if its intersection vector is valid. Finally, if all previous tests are passed and the correct direction is determined for both intersection vectors, then iA and iB must be equal in order to conclude that the two arcs do intersect each other.

Both the normal vectors and the new intersection vectors represent the *extended normal star* of the vertex, and are stored in a list, represented as Aa . The order of the vectors in this list is still the same as in the original *XN,* plus the intersection vectors in their corresponding position.

4.3.4 Matching of intersection vectors

Every arc in the spherical indicatrix will be tested for intersections against all other arcs. While this may not seem like an optimal solution, it has a side effect than can be effectively exploited when identifying the spherical polygons of the Gauss Map: Each intersection is detected once for each of the arcs involved. For the process of splitting the spherical polygons, at the location of arc intersections we need to have a vector for each of the spherical polygons that meet at that point.

Let n be the number of vectors in λ_N. If $n < 4$ there can be no intersections, so no checks are performed. Each arc is represented by its two endpoint vectors: $(\mathbf{n_k}, \mathbf{n_{k+1}})$, which are normal vectors of adjacent faces.

New intersection vectors are added to the normal list as they are detected. Consider the arcs $A = (\mathbf{n_k}, \mathbf{n_{k+1}})$ and $B = (\mathbf{n_{k+h}}, \mathbf{n_{k+h+1}})$, where $h > 1$. If arcs A and B intersect at the vector $\mathbf{i_1}$, then this new vector will be inserted in the list of the *extended normal star* (λ_X), after vector $\mathbf{n_k}$. An equal intersection vector $\mathbf{i_2}$ will be identified later on when testing for the intersection of B with A, and then the vector $\mathbf{i_2}$ will be inserted after vector $\mathbf{n_{k+h}}$.

This is best explained with an example. We will use a planar representation of the spherical indicatrix of the vertex shown in Figure 4.1(c). With the vector names in Figure 4.9 the algorithm would start by testing arc $A = (\mathbf{n_1}, \mathbf{n_2})$ for intersection against all other arcs. The first intersection would happen with arc $C = (\mathbf{n_3}, \mathbf{n_4})$ at the new vector $\mathbf{i_1}$. The same intersection point will be found when comparing C with A. At the end, the list of the *extended normal star* would have the vectors in the following order: $\mathbf{n_1}, \mathbf{i_1}, \mathbf{n_2}, \mathbf{n_3}, \mathbf{i_2}, \mathbf{n_4}, \mathbf{n_5}$.

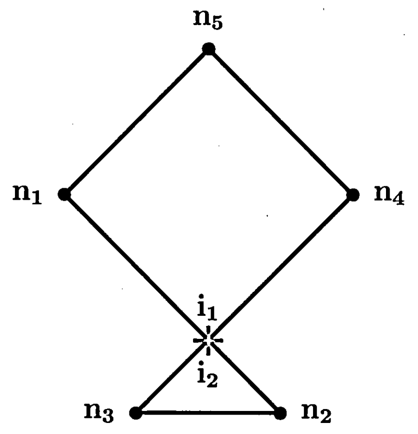


Figure 4.9: Planar representation of the spherical indicatrix, showing the intersection vectors.

The intersection vectors are determined by the four normal vectors corresponding to the endpoints of the intersecting arcs, in the given example both $\mathbf{i_1}$ and $\mathbf{i_2}$ are determined by the vectors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$ and $\mathbf{n_4}$. These vectors are used to match the intersections into pairs. These pairs of intersections are the ones used to split the spherical polygons.

The list of vector pairs λ_{pairs} stores the indices of the matching intersection vectors, as well as the vectors before and after the intersection for both arcs. In this case there will be an entry in λ_{pairs} containing the reference indices to the intersection vectors $\mathbf{i_1}$ and $\mathbf{i_2}$, and also the endpoint vectors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$ and $\mathbf{n_4}$.

It is possible for an arc to be intersected more than twice. The endpoint vectors stored in λ_{pairs} are used to identify these cases. In this situation the intersection vectors must be ordered to follow the same sequence as the rest of the vectors in the extended list. An extreme case of this is when an arc is intersected by several other arcs at the same point. This is a point where more than two spherical polygons meet in the PGM. To correctly split these polygons, each of the intersection vectors found has a variable `num_intersections` that serves as a counter of how many pairs of arcs intersect at its locations.

If the intersection of the arcs occurs precisely at the endpoint of one or both of the arcs, then a special treatment is necessary to maintain the consistency of the lists. Normally such intersections would be detected 3 or four times by the algorithm, creating too many intersection vectors at the same location. The possible intersections are shown in Figure 4.10:

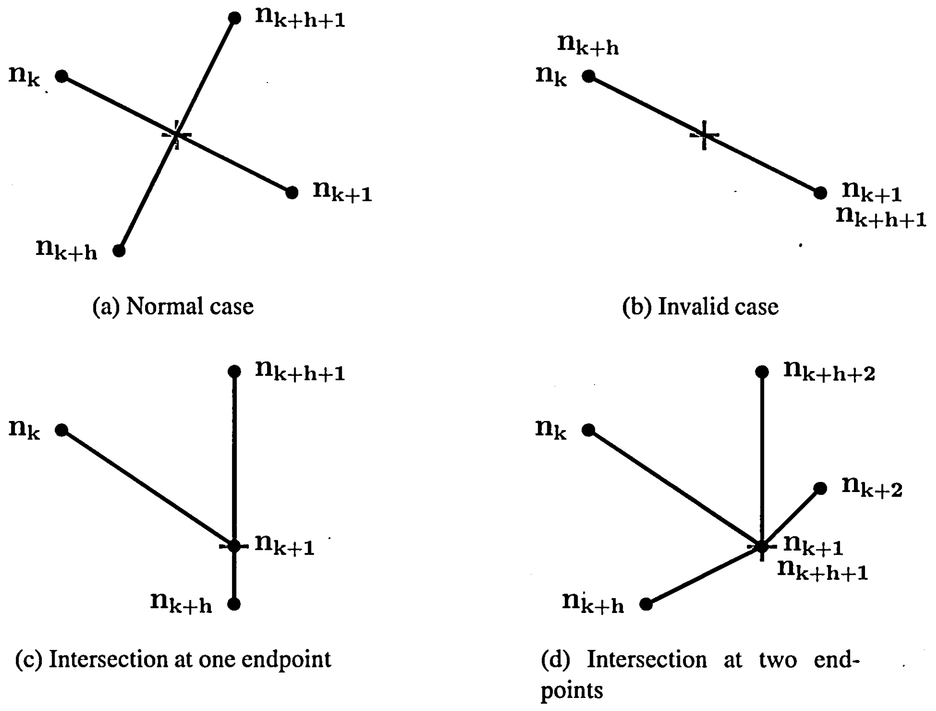


Figure 4.10: Different cases of arc intersections.

The integer value of the `endpoint` variable is used to determine which of these cases occurs. Only three possible values of `endpoint` need to be considered, while the others are simply ignored:

- `endpoint` = 0: This is the normal case explained before. Intersections of this kind are matched in pairs with other intersection vectors of the same type.
- `endpoint` = 1: Intersection occurs at the first endpoint of the first arc ($\mathbf{n_k}$). It has to be matched to an intersection with `endpoint` = 3.
- `endpoint` = 3: Intersection occurs at the first endpoint of the second arc ($\mathbf{n_{k+h}}$). It has to be matched to an intersection with `endpoint` = 1.
- `endpoint` = 5: This represents the case of an intersection at the first endpoint of both arcs ($\mathbf{n_k}$ and $\mathbf{n_{k+h}}$). Intersections of this kind are matched in pairs with other intersection vectors also with `endpoint` = 5.

As new intersection vectors are detected, they are compared to previously found intersections to match them in pairs. The list of vector pairs λ_{pairs} will contain the previously found vectors. When a new vector is found that matches one already in the list, it is added as the matching pair of that other vector.

For the cases when an intersection occurs at the first endpoint of the first arc, the insertion of the intersection vectors into the extended list λ_X will only be done when the intersection is matched for the first endpoint of the second arc. That is, an intersection with `endpoint` = 1 can only match an intersection with `endpoint` = 3 and *vice versa*. A similar treatment is done for the cases of an intersection at the first endpoint of the two arcs. While the intersection will be detected by all four of the involved arcs, it should only be stored twice. Only when `endpoint` is equal to 5 is it registered as a valid intersection.

All other possible values for `endpoint`: 2, 6, 8, 9, or 10, are always discarded, to avoid unnecessary duplication of intersection vectors. All intersections are correctly handled with the cases explained above.

When the intersection happens at one of the endpoints, then that vector becomes an intersection vector, and no new vectors are inserted into λ_X. The vector which becomes an intersection also gets a counter `num_intersections`. In this case the two adjacent vectors become the endpoints of the arc it intersects. This information is stored in the list λ_{pairs}. In the example shown in Figure 4.10(c), the entry on the list would have the intersection vectors $\mathbf{n_{k+1}}$ and $\mathbf{i_2}$, and the endpoint vectors $\mathbf{n_k}$, $\mathbf{n_{k+2}}$, $\mathbf{n_{k+h}}$ and $\mathbf{n_{k+h+1}}$. This is necessary to match the vector pairs in the case of multiple intersections at the same location.

4.3.5 Splitting into spherical polygons

After building the extended normal list, it is necessary to separate the individual spherical polygons that meet at the intersection points.

While there may be many intersecting arcs in the spherical indicatrix of a vertex, not all of the arc intersections found correspond to the meeting point of two spherical polygons. Many of the intersections detected mark only the place where two spherical polygons overlap. Such is the case of the vertex shown in Figure 4.11.

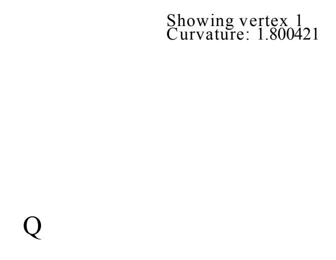


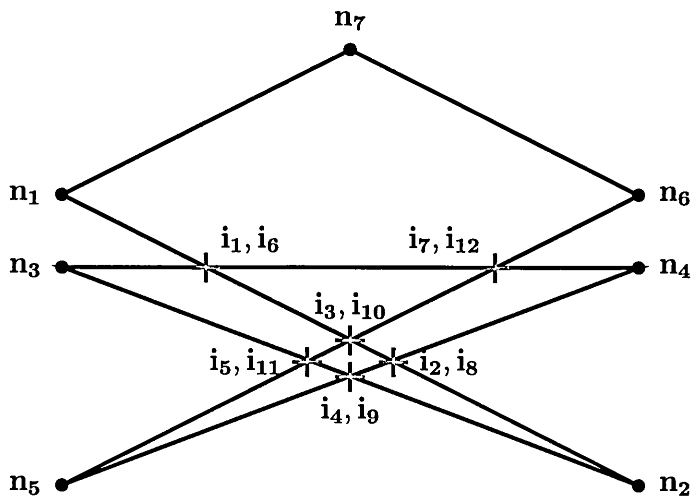
Figure 4.11: Gauss Map of a cone with several intersections on the spherical indicatrix and overlapping spherical polygons.

The procedure to identify the spherical polygons determined by the indicatrix consists on making a walk around the vectors in Ax in accordance to their order. All the intersection vectors found are stored in a list, and every time a new intersection vector is met, it is checked against the previously visited intersections. If there is a match, all the vectors visited in between are removed from Ax and stored into a new list *Xsp,* that represents a new spherical polygon.

For all the intersection vectors moved to *Xsp,* their matching vectors are located in the Ax and marked as visited, so that they will not trigger another match. All of these pairs and the one that triggered the match have their `num_inter sect ions` variable decreased by one. A vector is removed from *Xpairs* when its `num_inter sect ions` counter reaches zero. This procedure continues until the list *Xpairs* is empty. At that point, the reminder of Ax is also inserted as the final spherical polygon.

The cone-star vertex in Figure 4.11 will be used to exemplify this. Again using a planar representation, the indicatrix of the cone is shown in Figure 4.12, together with the normal vectors and the intersection vectors named in the order they are found.

A walk around the indicatrix would start at vector n i. The first intersection found is ii. It is stored in a temporary list. Next i3 is found. Since it is not the pair of il5 it is also stored in the temporary list, and the same occurs for intersection vectors i2, *14* and



$\mathbf{n}_7$
$\mathbf{n}_1$
$\mathbf{n}_6$
$\mathbf{i}_1, \mathbf{i}_6$
$\mathbf{i}_7, \mathbf{i}_{12}$
$\mathbf{n}_3$
$\mathbf{n}_4$
$\mathbf{i}_3, \mathbf{i}_{10}$
$\mathbf{i}_5, \mathbf{i}_{11}$
$\mathbf{i}_2, \mathbf{i}_8$
$\mathbf{i}_4, \mathbf{i}_9$
$\mathbf{n}_5$
$\mathbf{n}_2$

Figure 4.12: Planar representation of the spherical indicatrix, showing the intersection vectors.

$\mathbf{i}_5$. When $\mathbf{i}_6$ is found, it matches $\mathbf{i}_1$, so a spherical polygon has been identified, defined by the vectors: $\mathbf{i}_1, \mathbf{i}_3, \mathbf{i}_2, \mathbf{n}_2, \mathbf{i}_4, \mathbf{i}_5, \mathbf{n}_3$. These are removed from λ_X and inserted into the list for the first spherical polygon λ_{sp_1}. The intersection vectors that match those removed are marked as visited, removed from the list λ_{pairs} and their intersection counts are reduced by one. In this case it is vectors: $\mathbf{i}_6, \mathbf{i}_{10}, \mathbf{i}_8, \mathbf{i}_9, \mathbf{i}_{11}$, and since they only have one intersection each, their intersection counter `num_intersections` is left equal to zero, meaning that they are no longer valid intersection vectors and only treated as normal vectors. The result is shown in Figure 4.13.

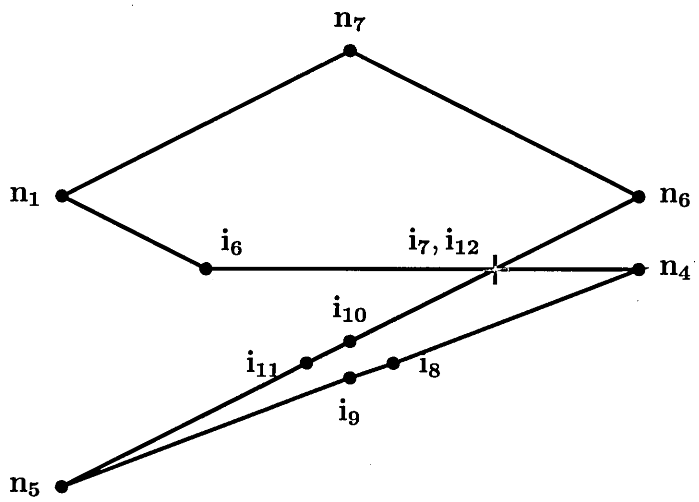


Figure 4.13: Indicatrix after the removal of the first spherical polygon.

The process begins again from the vertex where the last spherical polygon was split, in this case $\mathbf{i}_6$. Intersection vector $\mathbf{i}_7$ is found first. There are no more valid intersections, except for its pair, $\mathbf{i}_{12}$. This removes a second spherical polygon: $\mathbf{i}_7, \mathbf{n}_4, \mathbf{i}_8, \mathbf{i}_9, \mathbf{n}_5, \mathbf{i}_{11}, \mathbf{i}_{10}$, which is stored as λ_{sp_2}. After removing the pair just matched, the list λ_{pairs} is now empty. Thus all the vectors left in λ_X form the final spherical polygon λ_{sp_3}, shown in Figure 4.14, ending the polygon splitting process.

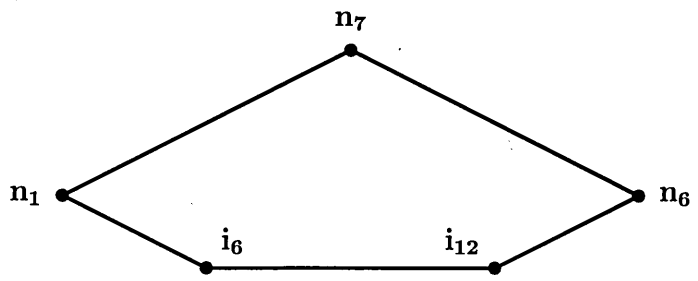


Figure 4.14: Indicatrix after the removal of the second spherical polygon. There are no more intersections, thus the indicatrix represents now the last spherical polygon.

4.3.6 Area of the spherical polygons

After splitting the indicatrix into individual spherical polygons, we compute the area of each polygon to find the curvature of the vertex. To do this we require a measurement of the angles between arcs on the surface of the sphere. The arcs cover segments of geodesics, or great circles, which correspond to the intersection between the sphere and a plane. The measure of an angle between two great circles is thus the same as the angle between two planes. In our implementation, we have arcs defined by two vectors, and we need to measure the angle between two adjacent arcs. Computing the cross product of the two vectors that define an arc we determine a vector normal to the plane of the arc. The angle between the normal vectors of the planes is the same as the angle between the planes.

The area of a spherical polygon is related to the number of vertices of the polygon and the angles at said vertices. The normal and intersection vectors used to build the indicatrix now become the vertices of the spherical polygons. Given a spherical polygon P_i with n vertices, its area is computed using the *spherical excess formula* (Weisstein 1999b) as such:

$$area(P_i) = \left(\sum_{j=1}^{n} \alpha_j - (n-2)\pi \right) radius^2, \tag{4.3.9}$$

where α_j is the angle at the vertex j of the spherical polygon. When using a unit sphere, as is our case, the square of the radius is equal to one. Thus the last multiplication can be safely ignored.

4.3.7 Determining the orientation of a spherical polygon

We need to know the orientation of a spherical polygon to compute the sum of its angles and find its area. Any spherical polygon will split the sphere into two areas, giving two possible ways to interpret it. One area will be oriented Counter–Clockwise and is positive, according to our implementation; the complementary area is negative and oriented Clockwise.

Only one of these areas correctly describes the curvature of a vertex in the polygon mesh. It is of great importance to select the correct one. We have determined that if the ordering of the faces and their normal vectors around the vertex is consistent, the area is positive; if the ordering of faces and normals is reversed, the area is negative. However, due to the potential complexity of the vertex stars, the identification of these orderings represents a non–trivial task. The approach taken to determine the sign of spherical polygons is to initially consider them as being positive. After that some tests are performed to decide if the orientation of a polygon must be changed to be negative. The initial approach at identifying the appropriate area is based on the areas of the spherical polygons.

Lemma 4.3.1. *For any manifold vertex (without any self–intersections) the maximum area of any of its individual spherical polygons will always be less than 2π (half of a sphere).*

Proof: For a convex–star vertex, there are two extreme cases:

- A flat vertex, where the sum of the angles of the faces in $star(\nu)$ is 2π, and thus the curvature is:
$$\omega_{flat} = 2\pi - 2\pi = 0. \tag{4.3.10}$$
- A 'needle' like vertex, where for every face $t_i \in star(\nu)$ the angle α_i incident on ν is less than ϵ, and approximating zero. In this case the curvature is:
$$\omega_{needle} = 2\pi - \sum_{i=1}^{n} \epsilon_i < 2\pi. \tag{4.3.11}$$

Thus the curvature of every individual spherical polygon of a convex–star vertex is in the range $0 \leq \omega < 2\pi$. The same can be said of the dual convex cone of basic saddle–star vertices. ■

Based on this fact we compute the area of a spherical polygon with the assumption of it being oriented Counter–Clockwise. If the resulting area is larger than 2π, then the orientation of the polygon is switched to negative, and the area recomputed for the complement of the polygon. This approach will find the correct orientation of the spherical polygons in meshes without non-manifold vertices.

Experiments with *figure-8'* vertices with sharp cones show that it is possible to have a single spherical polygon with an area larger than ***2ty,*** as is the case of the vertex in Figure 4.15(a). The selection of the sign based on areas smaller than would select the inverse of the area, resulting in an incorrect negative curvature. The resulting Polyhedral Gauss Map is shown in Figure 4.15(b). For these complex vertices it becomes necessary to employ other geometrical properties of the vertex to determine the correct orientation of spherical polygons.

» r e V eja

ihowing verti
.urvature (PC

(a) ‘Figure-8’ vertex and PGM with area larger than *2K*

(b) Incorrect result using the inverse area

Figure 4.15: Gauss Map of a vertex with a self-intersection.

From the division of curvature into positive and negative components *UJ+* and *ω~,* we have determined in Section 3.2.3 that the positive components correspond to convex cones, and thus that they are themselves convex spherical polygons. We use this knowledge as a second discriminating parameter to determine the sign of spherical areas.

Lemma 4.3.2. *The maximum perimeter of a convex spherical polygon will be 2n, and this corresponds to a polygon covering half of the unit sphere, that is, with an area of 2n.*

Proof: The proof of for this follows from the proof of Lemma 4.3.1. ■

For the special case of a ‘figure-8’ vertex the area of a positive spherical polygon is greater than 27r, then its perimeter must also be greater than *2n,* otherwise we can assume the polygon should be inverted and turned into a negative one.

In the case when both the area and the perimeter are larger than *2n,* the sign of the polygon is still unclear. Additional information about the *star type* of the vertex *a,* described in Definition 3.4.2, can partially help clarify the orientation of the spherical polygons. Separate parameters must be compared to determine the orientation of spherical polygons for either *cone-star vertices* or *saddle-star vertices.*

As explained in the previous chapter, the ‘figure-8’ vertices can be considered as two independent convex cones sharing one edge, where the faces of the vertex star intersect. This results in a PGM consisting of two convex spherical polygons also glued together at one shared edge. The combined spherical polygon will have exactly two concave angles. Thus, if *a* is a *cone-star vertex,* then its spherical polygons with area larger than *2n* must have exactly two concave angles to remain positive. If there are more or less concave angles, then the spherical polygon must be inverted and become negative. However, if *a* is a *saddle-star vertex* this condition can not always be used. These vertices present the possibility of positive and negative polygons cancelling each other. In such cases, positive polygons become non-convex, and can have any number of concave angles. For saddle-star vertices with self-intersections the correct sign of the spherical polygons is not always found. These vertices can have spherical indicatrices with multiple arc intersections and overlapping spherical polygons. More work needs to be done to determine a correct parameters to identify the sign of spherical polygons for complex vertices, as the ones in Figure 4.16.

9.130837 l» r e V?^ 12.200121

Figure 4.16: Vertices with overlapping and cancelling spherical polygons.

Another method proposed to determine the correct polygons is to select those that include the normal vector of the vertex itself. However, for self-intersecting vertices this method does not work, and hence this method is not considered useful. Figure 4.17 shows a counter example, where the normal vectors of the vertices are shown in magenta. The normal of the selected vertex is shown in the PGM, and it lies outside of the spherical polygons that represent the curvature of the vertex.

Figure 4.17: PGM of a vertex, showing the normal of the vertex in magenta colour.

4.3.8 Special cases

There are several special cases, but they belong mostly to one of the types listed below, or a combination of these types; each case is dealt with in a different way:

- A vertex with pairwise coinciding normals, related to non-adjacent faces. An example is a vertex of the configuration commonly known as the *monkey saddle,* with its 6 faces equally sized. It has three pairs of pairwise coinciding normals. This gives us three pairs of identical vectors, and the PGM has, therefore, two overlapping identical areas. But in its spherical indicatrix there is no clear intersection of the arcs between the normal vectors. This case is solved by an initial step that will look for groups of normal vectors that begin and end with a vector at the same location. These are known as *vector loops.*
- A vertex with the spherical indicatrix self-intersecting several times at the same point. An example is a vertex that can be constructed by adding two more faces to a monkey saddle. The star of such a vertex has four convex edges and four concave edges. Let us also assume that all convex edges lie in the same plane. It means that the star of the vertex has a degenerated convex cone, i.e. it represents a plane. Therefore the positive curvature is equal to zero. We call such a vertex a *tulip.* The arcs of its spherical indicatrix intersect themselves four times precisely at the same point. This new point will be common to four different spherical polygons. Our approach at detecting arc intersections allows us to keep several equal intersection vectors at this point. They will later be distributed among the four spherical polygons.
- The area of a simple spherical polygon is greater than 27r. This case arises only if two simple spherical polygons are ‘glued’ together, i.e. have a common boundary. This is the case of vertices in the ‘figure-8’ configuration. This requires additional checks, as described before.

4.3.9 Computational complexity of the algorithm

The segmentation of the spherical indicatrix into individual polygons requires the comparison of all the face normals and intersection vectors against each other. This operation is the most computationally expensive and will be used to show an upper bound on the complexity of the algorithm.

For a closed triangular mesh of n vertices, it is known that the number of faces is $2n - 4$ and the number of edges is $3n - 6$. In all cases there will then be $2n - 4$ face normals, and additionally non–convex polyhedra will have intersection vectors because of the mixed vertices. Only a small number of the intersection vectors actually correspond to the point where two spherical polygons meet, as was shown in the example in Section 4.3.5. We will call these the *real intersection* vectors, denoted with R; the intersection vectors produced by the overlapping of two spherical polygons will be called *virtual intersection* vectors, and denoted with V. The total number of vectors in the indicatrix is denoted as $P = R + V$.

The real intersection vectors are generated by deviation edges in the mesh, *i.e.* where a convexity changes to a concavity or *vice versa*. If a concavity is formed by several concave edges, in this analysis it is equivalent to a concavity with only one concave edge. Taking this into consideration, we can estimate the upper bound of such deviations as half the number of edges in the mesh. But each such edge is incident on two vertices, and thus will be used twice. Assuming the worst case, in which every concave edge produces a real intersection vector, we have that $R = 3n - 6$.

Now the number of virtual intersections needs to be determined. Every pair of spherical polygons can intersect in at most 4 points. The number of spherical polygons is determined by the number of convex cones that can be extracted from the decomposition of mixed vertices. In the worst case (not achievable) where all the concavities are incident on a single mixed vertex, the negative part of the PGM is made up of a spherical polygon of $(3n - 6) - 1 + 2$ vertices. Any polygon with m vertices can be split into $m - 2$ triangles. Again using the worst case scenario, the maximum number of spherical triangles is $t = 3n - 5$.

Each pair of triangles can intersect up to 4 times. The number of possible triangle pairs is obtained by a permutation as:

$$\begin{pmatrix} t \\ 2 \end{pmatrix} = \frac{t(t-1)}{2!}. \tag{4.3.12}$$

Substituting the value of t to get V we have:

$$V = 4\frac{(3n-5)(3n-4)}{2}. \tag{4.3.13}$$

The total number of vectors in the indicatrix is then:

$$P = R + V = (3n-6) + 2(3n-5)(3n-4). \tag{4.3.14}$$

This is a highly pessimistic upper bound on the number of vectors that must be analysed, and so the program can be considered to be order $O(n^2)$.

In normal polyhedral meshes the connectivity of the vertices is much simpler, and these worst case scenarios are impossible to achieve. In a real application the complexity can be considered to be much lower. However the complexity is highly dependant on the connectivity of the mesh, and it is difficult to make a correct practical estimation.

4.4 Examples of curvature measures using PGM

In this subsection we present examples of PGM of several types of vertices. The images show two different views, or scenes: the left scene shows the model of the vertex star and the right scene shows the spherical polygons of the Gauss Map. Positive areas in the PGM are shown in red, while negative areas are displayed in blue. The vertices in the meshes are coloured according to their vertex type as such: flat vertices are white, convex vertices are red, saddle vertices are blue and mixed vertices are green.

Figure 4.18 shows the PGM of a complex mixed vertex. In the left picture the vertex is presented with its spherical indicatrix, and in the right picture the spherical polygons representing positive and negative curvature can be seen. The positive area is clearly separated from the negative ones, which are overlapping. Two simple spherical polygons of negative area represent the 'concavities' in the cone of the vertex. The curvature measures of this vertex, as provided by the program, are the following:

```
=== ABSOLUTE CURVATURE:      1.619293 ===
--- POSITIVE CURVATURE:      1.359348 ---
--- NEGATIVE CURVATURE:      0.259945 ---
--- GAUSSIAN CURVATURE:      1.099403 ---
```

In Figure 4.19 the PGM of the monkey saddle is presented. The PGM looks similar to the one for a simple saddle vertex, but consists eventually of two completely overlapping spherical polygons. The indicatrix of the monkey saddle has no additional intersection vectors, so the face normals must be used as intersection vectors as well. The curvature of this vertex is measured as:

Figure 4.18: A complex mixed vertex and its spherical indicatrix (left) and PGM (right).

```
=== ABSOLUTE CURVATURE:      4.657674 ===
    POSITIVE CURVATURE:      0.0 00000
    NEGATIVE CURVATURE:       4.657674 ——
    GAUSSIAN CURVATURE:       -4.657674-——
```

Figure 4.19: PGM of the monkey saddle.

PGM visualisations of a generalised monkey saddle are shown in Figure 4.20. This kind of vertex also has two loops of completely overlapping spherical polygons, in a more complex configuration. The curvature measures obtained by the program for this vertex are:

```
=== ABSOLUTE CURVATURE       6.686595 ===
    POSITIVE CURVATURE       0.000000 -----
    NEGATIVE CURVATURE       6.686595 ——
    GAUSSIAN CURVATURE       -6.686595 ——
```

In Figure 4.21, PGM of the 'tulip' vertex is presented. Its PGM consists of four equal spherical polygons, pairwise adjacent to each other, which are revealed by the spherical indicatrix. However the coloured PGM looks like one spherical polygon with four sides. The curvature measures for the 'tulip' vertex are:

```
=== ABSOLUTE CURVATURE:      0.805432 ===
    POSITIVE CURVATURE:      0.000000----
```

Figure 4.20: PGM of a generalised monkey saddle.

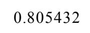


Figure 4.21: PGM of the tulip vertex.

```
NEGATIVE CURVATURE:      0.805432-——
GAUSSIAN CURVATURE:     -0.805432 ——
```

The vertices presented in Figures 4.18^4.21 have no self-intersecting faces, and their Gauss maps are in conformity with curvature characterisations for embedded polyhedral surfaces given in (Banchoff 1970). In the next few examples we present PGM visualisations of vertices with self-intersections. This provides curvature characterisation of more complex surfaces, which are neither embedded nor immersed. It can be observed that, for these vertices, the Gaussian curvature measures are no longer equal to the positive minus the negative curvature components, as the AD has problems with such vertices.

Figure 4.22 and Figure 4.23 show the PGM of two vertices, which we call a *pinch vertex* and a *reverse pinch vertex,* respectively. In order to understand the difference between a pinch vertex and a reverse pinch vertex, imagine a walk along the link of the star of a vertex *v.* In the case of the pinch vertex the walk makes two full turns around the vertex, both turns having the same orientation (for example, Counter-Clockwise). In the case of the reverse pinch point, the walk makes also two full turns, one in the Clockwise-Direction, and the second one in the inverse direction (i.e. Clockwise). The Gauss map of the pinch vertex, presented in Figure 4.22, has two overlapping areas, each of positive sign, which is a case not previously met. One area is equal to the curvature of the convex star of the pinch vertex.

```
=== ABSOLUTE CURVATURE       6.358589 ===
    POSITIVE CURVATURE       6.358589 —
    NEGATIVE CURVATURE       0.000000 -----
    GAUSSIAN CURVATURE       0.075404 —
```

Figure 4.22: Pinch vertex with the corresponding Gauss Map.

The Gauss map of the reverse pinch vertex, shown in Figure 4.23, has also two areas of positive curvature, but they are non-overlapping and separated by an area of negative curvature.

```
=== ABSOLUTE CURVATURE       5.992379 ===
    POSITIVE CURVATURE       5.940217 —
    NEGATIVE CURVATURE       0.052161 —
    GAUSSIAN CURVATURE       -0.395130 —
```

Figure 4.23: Reverse pinch vertex with the corresponding Gauss Map.

Finally, we show the Gauss map visualisation of the*figure-8 vertex.* Such a vertex can also be considered as the ‘extreme’ case of the reverse pinch vertex, when the negative curvature part has disappeared. The two remaining positive parts are ‘glued’ together. Figure 4.24 presents the PGM of three*figure-8* vertices with a cone of increasing height. We can see that as the height grows, so does the area of the PGM. The curvature of the first vertex is:

```
    ABSOLUTE CURVATURE      4.723622 =
    POSITIVE CURVATURE      4.723622 -
    NEGATIVE CURVATURE      0.000000 -
    GAUSSIAN CURVATURE      -1.559563
```

In the middle image the area is 2π (half the sphere):

```
=== ABSOLUTE CURVATURE      6 .283185
    POSITIVE CURVATURE      6.283185
    NEGATIVE CURVATURE      0 .000000
    GAUSSIAN CURVATURE      0.000000
```

In the third image the PGM covers a large part of the surface of the sphere:

```
=== ABSOLUTE CURVATURE      10.988695 ===
    POSITIVE CURVATURE      10.988695 -
    NEGATIVE CURVATURE      0.000000 -----
    GAUSSIAN CURVATURE      4.705510 --
```

8 & G U V & I:6.283185

Figure 4.24: Cone-eight vertices of increasing height.

4.5 Results and potential applications

This section presents the advantages of the method of Polyhedral Gauss Map to measure Total Absolute Curvature of discrete surfaces, as opposed to the Gaussian curvature measured by the Angle Deficit method. A number of potential applications that can benefit from the new method are outlined as well.

4.5.1 Comparison of the Angle Deficit and the Polyhedral Gauss Map

The Gaussian curvature measured by the AD method is capable of clearly identifying three kinds of vertices: flat, convex or saddle. However, vertices which do not fit into these categories will present problems to the algorithm. This was illustrated previously with the polyhedra in Figure 3.3. The left polyhedron has vertices that are non-convex. However the AD method will recognise them as convex, and will measure the same curvature for each corresponding vertex in both objects.

The Total Absolute Curvature provides a better understanding of the geometry of an object than the Gaussian curvature alone. The PGM method allows for clear distinction of the vertices into four basic types: flat, convex, saddle and mixed. This information can be used to completely characterise an object. The vertex characterisation can be refined, by identifying convex vertices or the cones of mixed vertices that are concave with respect to the object, increasing the vertex classes to six, thus including convex–concave vertices and splitting the mixed vertices into two types: mixed–convex and mixed–concave vertices. The curvature domains of a discrete surface can be recognised by colouring the vertices according to their type. In the following images we show polyhedral meshes with colour coding to represent the identified vertex curvatures, using both the AD and PGM methods. Vertex colours by type are: flat (white), convex (red), saddle (blue), mixed–convex (green), convex–concave (magenta) and mixed–concave (brown).

For the example of the two polyhedra, using the PGM method to determine the curvature immediately shows a difference in the shape of the vertices at the top corners of both objects. Figure 4.25 shows the vertex characterisation of the two objects using AD and PGM. The images highlight one of the vertices that have equal AD but distinct TAC. The curvature measure of the selected vertex is displayed in the images, as well as its corresponding Polyhedral Gauss Map.

The curvature visualisation of a discrete torus is presented in Figure 4.26. We see the wireframe of the mesh, the surface shaded according to the vertex characterisation provided by Angle Deficit, and the surface shading using the vertex characterisation obtained with the PGM method. Both visualisations are presented with the PGM of a selected vertex, and the numerical value of the curvature of the vertex, as determined by each method. It is clear that the curvature visualisation by means of the Angle Deficit does not reflect the irregularity on the surface of the torus. The vertex, classified as a vertex of positive curvature by means of the Angle Deficit, has two hidden domains of negative curvature, and is a mixed vertex, which is clearly seen in the curvature visualisation by means of the PGM method. The PGM visualisation indicates that the mesh may not be optimal.

By applying the optimisation method based on the minimisation of total absolute extrinsic curvature, introduced in (Alboul and van Damme 1994) and later further developed in (Dyn *et al.* 2001), (Alboul 2003) and (Netchaev 2004), we obtain an optimised mesh for this torus. The optimisation method preserves the location of all the vertices, and only flips the edges to reduce the curvature of the whole mesh. The visualisations of the optimised mesh are given in Figure 4.27. Again, as in Figure 4.26, we see the surface coloured based on the curvature characterisation. We can see that now the two curvature

Ihowing verti
lurvature (PC

(a) Convex object

1.682137

(b) Non-convex object

Figure 4.25: Vertex classification of the two polyheclra. Left: Angle Deficit classification. Right: Polyhedral Gauss Map classification.

visualisations are more similar. In the PGM view, mixed vertices form a thin circular domain (shown in green colour) that separates the domains of positive and negative curvature. This domain is not present in the AD view. The PGM of a vertex situated in this domain reflects the almost symmetrical distribution of negative and positive curvatures in this vertex.

Another optimisation of the torus can be obtained by minimisation of the *Willmore energy,* as presented in (Brink and Alboul 2006). The new optimised torus is shown in Figure 4.28. It provides a more regular distribution of the triangles that results in a smoother surface than the previous cases. The PGM of the vertices in the mixed domain are simpler, reflecting the regularity of the surface.

Figure 4.29 shows the visualisation of the curvature domains in a simplified *venns* mesh. Figure 4.30 has the same visualisations on a *triceratops* model. In both cases the curvature domains of mixed vertices can be seen on the PGM. Also the vertices selected show a difference in the curvature measured by these two methods.

(a) Wireframe (b) AD colouring (c) PGM colouring

Figure 4.26: Visualisations of the initial torus, coloured by curvature, highlighting the curvature of a selected vertex.

(a) Wireframe (b) AD colouring (c) PGM colouring

Figure 4.27: Visualisations of the optimised torus, coloured by curvature, highlighting the curvature of a selected vertex.

4.5.2 Object recognition based on curvature

Various applications can benefit from a more accurate measurement of curvature and vertex classification. In (Li and Gu 2004) the curvature is used to compare a design model of an object with a mesh scanned from the production object. The measurements of both Mean and Gaussian curvature are used to characterise the vertices of the meshes. Two distinct approaches are described, and then the third method is introduced that benefits from the other two. The first approach is to classify vertices by the measured value of the curvatures. The second one uses the sign of the curvatures, and the third uses both value and sign. If both curvature and sign agree, then the vertex is clearly identified. Otherwise a weight parameter is defined to choose the parameter that is less ambiguous. With this approach four types of curvature are found: convex, concave, saddle and flat. As well as in object identification, measurements of curvature have previously been used in face recognition (Grodon 1991), (Tanaka *et al.* 1998). However, these applications use the EGI and AD to compute curvature, and could be improved with the proposed PGM.

In Figure 4.31 we show meshes showing the vertex classification obtained using both the AD and the PGM. Regions of similar curvature are clearly seen from the images, and

001016 a i m

(a) Wireframe (b) AD colouring (c) PGM colouring

Figure 4.28: Another optimised torus, coloured by curvature, highlighting the curvature of a selected vertex.

Showing vertex:398
CurvaiUre (AD)::0.001498

ttM .033192

Figure 4.29: AD and PGM visualisations of a simplified *venus* model.

their identification can be of particular use in many applications of object identification or face recognition.

4.5.3 Terrain description and navigation

Another use of vertex characterisation using the PGM is the identification of terrain features. This information can be applied for the navigation of a discrete representation of a surface (Falcidieno and Spagnuolo 1991). A common representation of terrain is by the use of height maps, where a coordinate grid over the terrain references the altitude at

0.036340

Figure 4.30: AD and PGM visualisations of a *triceratops* model.

(a) *Venus* **model**

(b) *Face* **model**

(c) *Casting* **model**

Figure 4.31: Various meshes with vertices colour coded according to their characterisation using the two methods. Measures of curvature using Angle Deficit (centre) and Polyhedral Gauss Map (Right).

every corresponding point of the surface. From these maps it is straightforward to create a polyhedral mesh.

Measurements of curvature over a mesh are used in (Lee *et al.* 2001) to simulate the spreading of a fire that consumes the surface of an object. This application is useful in computer graphics and simulations. Information about the geometry of the terrain is also employed for terrain reasoning in 3D action games (Van der Sterren 2001), for example, for path finding on a terrain, which satisfies certain requirements. One requirement might be to minimise the energy that a game object needs to apply to navigate on the surface, or that the path chosen should go over the tops of mountains.

The use of the PGM for terrain navigation has been explored in (Echeverria *et al.* 2005). A terrain can be viewed as an irregular surface, and it is possible to determine areas of the surface with similar values of curvature, by grouping together vertices of the same type. Areas of positive curvature would represent mountains or valleys, while areas with mixed curvature are rugged terrain. The border between mountains and valleys would be composed of saddles, and thus have a negative curvature.

Some experiments have been carried out on the use of the PGM to characterise the terrain to be navigated. The aim is to find a path from an *origin vertex* to a *destination vertex*. Our approach is to generate paths that stay within regions of the same kind of vertices, preferably of saddle type (regions of negative curvature), to avoid obstacles and navigate around them, based on the features of the surface. Figure 4.32 shows the comparison of using a direct path approach against the use of curvature information to navigate around a mountain in the terrain. The examples shown make use of these algorithms to create two different paths over each surface:

- The direct path, where at each vertex the next step is to go directly to the vertex that is closest to the destination.
- A path that considers the curvature of the neighbour vertices and will give preference to vertices of negative curvature for the next step.

By keeping within regions of negative curvature the paths produced are not much longer than taking a direct route, however we avoid going over the peaks of the mountains. Thus there are less climbs and descents, resulting in a smaller change in height (Δh). This is important as a means of minimising the energy required to traverse the terrain. In the experiments, for each path we measure the length l and the height difference Δh. Figure 4.32 shows two examples of terrain and the paths generated using both approaches.

4.5.4 Polyhedral mesh subdivision or simplification

A possible application of the PGM is to use it in mesh subdivision, by modifying the existing vertices and adding new ones in a way that will remove all mixed vertices and leave only convex or saddle vertices. In this case the PGM will determine the point where a vertex has to be modified. A simple example of this is shown in Figure 4.33. The new saddle vertex must be located in a point that belongs to the convex hull of the original vertex. For more complex vertices more care is required to select the location of new vertices.

Another application for measures of curvature can be found in polygonal mesh simplification. We will elaborate on the use of the PGM for that purpose in Chapter 6.

(e) Length: 4.96, *Ah:* 0.91

(f) Length: 5.99, *Ah:* 2.25

Figure 4.32: Use of curvature for terrain navigation. Top: Terrain meshes. Middle: Direct path. Bottom: Path based on curvature.

(a) Original mixed vertex **(b) Modified vertex now convex** **(c) New saddle vertex**

Figure 4.33: Subdivision of a mixed vertex to generate two vertices, one positive and one negative.

4.6 Conclusions

We have presented a novel method to construct Gauss maps directly from a polyhedral surface. The method is theoretically valid and robust in implementation. It constructs complete curvature representations for a very large class of polyhedral surfaces, including certain non-manifold surfaces.

The Polyhedral Gauss Map enables us to reveal all existing domains of positive and negative curvature of a surface, which can not be correctly revealed by means of the Angle Deficit. An advantage of the PGM method is the capability of splitting the curvature domains of the same sign into sub-domains, each represented by a spherical polygon. This gives us a more complete description of geometric features of the surface.

The methods described in this chapter for the generation of the PGM include the detection of arc intersections in the spherical indicatrix, the splitting of the indicatrix into simple spherical polygons and the selection of the correct sign for each of the spherical polygons.

For cone-star vertices, the PGM algorithm can handle non-manifold surfaces, correctly identifying the curvature of vertices with self-intersecting faces. These vertices will produce larger than normal areas in the Gauss Map. The PGM method might be useful to check the validity of the mesh, since it is capable of identifying undesired self-intersections.

In the case of self-intersecting saddle-star vertices, positive and negative areas can overlap or cancel each other out in some cases. These cases may cause the program to confuse the areas and signs of the spherical polygons, because of the many intersecting arcs in the indicatrix. Further research is needed to determine the correct parameters to obtain the sign of spherical polygons.

Several potential applications for the new vertex characterisation have been presented.

The vertices of a polyhedral can be coloured according to their type. Grouping together vertices of the same type permits the visualisation of regions of equal curvature on a surface. This is useful in identification of manufactured pieces, face recognition, terrain description and operations on polygonal meshes. We demonstrated the advantages of PGM in these applications by comparing polygonal meshes that are characterised using either the Gaussian or the Total Absolute Curvature.

4.7 Future research directions

The method developed is very sensitive to noise in the data. Every feature of the mesh contributes to the total curvature, and noisy data can generate very large curvatures. This produces a large variation on the vertex types, and makes recognition of curvature domains difficult. An approach to solving this problem would be to expand the analysis of a vertex star to include also the stars of the neighbours. Thus for the PGM of a vertex, the normal vectors of the faces incident on it and its neighbours could be used. This would increase the tolerance to noise, but also reduce the accuracy of the measurements and increase the computational cost of the algorithm.

Another alternative solution to identify curvature domains is based on the results observed from the mesh simplification application, to be presented in Chapter 6. In a simplified mesh the vertices with small curvature are removed, leaving only those with the largest contribution to the shape of the object. Using our simplification method, the vertices that remain provide a clearly defined characterisation of the curvature domains.

With the current algorithm, the splitting of the indicatrix to find individual spherical polygons varies depending on the order of the triangles in the source data (*i.e.* the OBJ file). This may cause spherical polygons to be incorrectly identified, and in some cases the correct version can only be found from a very specific ordering of the faces. One solution implemented is to cycle through the possible orderings for each vertex, and finally choose one according to certain parameters. These parameters can be a minimisation of the area or the arc length of spherical polygons. However, the correct parameters have not been clearly defined, and the process of cycling the faces is time consuming. More research needs to be done to determine the correct parameters to find the correct representation of the PGM that can be obtained from all the possible vertex orderings.

Chapter 5

Visual display of the Polyhedral Gauss Map with 3D graphics

5.1 Introduction

When dealing with abstract mathematical concepts in three–dimensional space it is often very difficult to imagine the behaviour of all the elements involved and their spatial relationships. Even drawing the expected results on paper is often unclear and ambiguous. The best possible solution to these problems is visualisation in three dimensions. Modern computer graphics provide an ideal platform to represent the mathematical concepts in a graphical form that helps to understand these concepts better and facilitate proofs. The graphical representation also is capable of revealing 'hidden' associations and relations between these concepts, which may manifest in creating new techniques and concepts otherwise impossible.

For the research presented in this thesis, the visualisation of the Polyhedral Gauss Map in three dimensions is a reliable tool to properly evaluate the results obtained. This particular construction requires the projection of complex structures onto the surface of a sphere. A two–dimensional drawing would never suffice, since the entire surface of a sphere cannot be adequately mapped into a plane without distortion and without losing the connectivity of the objects on the sphere. The use of computer graphics to represent the sphere in 3D allows viewing from any angle, zooming into particular features, and filtering information to simplify the analysis of the data. Additionally, a 3D display can also show the results of the curvature measures directly on the source mesh being analysed. It permits visualising regions of similar curvature and selection of each individual vertex to analyse its Gauss Map.

The program for visualisation of the Gauss Map was developed exclusively for this project, dealing with very particular issues associated with this specific problem. Most of the programming used implements new ideas which comprise the contributions mentioned in this chapter, such as the display of spherical polygons by first triangulating them, and then drawing them using clipping planes in OpenGL.

OpenGL was chosen to implement the visual display of the PGM because of its flexibility and simplicity to program. It enables the creation of specialised functions required for the projection of structures on a sphere and dealing with the resulting spherical polygons. It also integrates well with the rest of the program and the data structures implemented, allowing for the visualisation to be used since the very early stages of the development of the algorithm. This is a valuable tool to simplify the understanding of the concepts behind the theory and provides a point of reference for comparison of the numerical results obtained. An additional advantage is the capability to visually identify problems and errors in the algorithm. For these reasons, it is desirable to have as accurate a representation of the results as possible.

The visual display is independent and not necessary for the measurement of the Gauss Map areas, since all the required curvature computations are performed before invoking any functions of OpenGL. The development of the 3D visualisation is a process that has been improved gradually, at each step adding new features and providing a better interface, as well as solving the problems found. This chapter will present the major milestones reached to obtain the current visualisation system. It describes the tasks performed to draw the Gauss Map for a single vertex in the polygon mesh, the same process is repeated for each of the vertices that compose the object.

5.2 Other visualisations of the Gauss Map

In (Horn 1983) the representation for the Gaussian Images is done using simple flat drawings to illustrate the presented concepts. In all cases the figures show simple objects and their corresponding Extended Gaussian Images (EGI). Since the graphics are done in two dimensions a circle is used to represent the sphere. The normal vectors of faces are roughly represented as points in the circle, with the same shape as that of the corresponding face. This initial representation is not very clear and ill suited to demonstrate more complex objects. The difficulties in dealing with non–convex objects are explained, but no solutions or illustration of these cases are given.

The work of (Banchoff *et al.* 1982) presents a graphical visualisation of the Gauss Map for smooth surfaces. Using computer graphics they produce the Gauss Map of various smooth surfaces, such as a handkerchief surface, a torus surface and a monkey saddle. The visualisation is created along whole patches of the surfaces, but not for the vertices in a polyhedral surface.

(Lowekamp *et al.* 2002) presents a visualisation of the Gauss Map using computer graphics and a modification of the EGI. In this application two views are presented to the user, one with the polygonal mesh and a second with the Gaussian Images. Both views are linked, so that they can be rotated simultaneously. The authors call this visualisation the Interactive Gaussian Image (IGI). The connection of the vertices in the mesh is translated into the IGI by connecting the normal vectors of the vertices in the same way as the vertices are connected in the mesh. Colour is used to relate the vertices on the mesh to their vectors on the Gaussian Image. When dealing with non–convex objects, the IGI presents overlapping patches on the sphere, which are difficult to tell apart. A partial solution provided is to permit the user to select patches of the mesh to visualise the IGI of a group of vertices simultaneously.

Our approach also makes use of two *viewports* to show both the mesh and the PGM at once, allowing the same transformations on both simultaneously. We also provide a better way to analyse the curvature of vertices individually, and can show the star of a vertex and its corresponding normals and spherical indicatrix directly on the view of the mesh. Our PGM visualisation program also provides other options to select the information to be displayed.

5.3 Display of the PGM with OpenGL

The display program used throughout this project is based on a simple graphic engine in OpenGL by Christophe Devine. This initial application already provides the basic tools required to draw objects in three dimensions. Rendering of a polygonal mesh requires the specification of faces as groups of 3 vertices. The *OBJ* format selected for the input files provides the required information about vertices and faces. A simple parser reads the *OBJ* file and creates the data structures that store the data.

The *display lists* of OpenGL are used to preprocess the objects to display, and speed up the visualisation. These lists are created immediately after the curvature computations. The first such list is created for the whole polyhedral mesh. For every vertex, additional display lists are created for the vertex star, normal star and spherical polygons on the PGM.

5.3.1 Visualisation of the spherical indicatrix

The first application to display the PGM used only one *viewport* to show both the polyhedral surface and the PGM. In the initial stages, only the PGM of the first vertex in the input file was generated. The normal vectors in *star(y)* were translated to the centre of the coordinate system. Being unit vectors, the endpoints lie on the surface of a sphere, as shown in Figure 5.1. This corresponds roughly to the Extended Gaussian Images approach. A sphere is drawn as a reference, with the same radius as the length of the normal vectors.

j

(a) Vertex with a non-convex neighbourhood

(b) Normal vectors to the faces, translated to the centre of a sphere

Figure 5.1: Visualisation of the *normal star* of a vertex.

The initial representation of the indicatrix was produced by joining together pairs of vectors with a straight line from the endpoint of the first vector to the endpoint of the second one. Repeating the same process for every pair of vectors in the appropriate order yields a crude representation of the indicatrix, as shown in Figure 5.2(a). This visualisation presents a very vague and deformed preview of the areas of the Polyhedral Gauss Map. The straight lines used to join the vectors do not always intersect each other, even if the arcs they represent would actually intersect on the surface of the sphere. Figure 5.2(b) evidences this difference when using a sphere as a point of reference.

Using the normal vectors linked this way, an approximation of the areas of the Gauss Map can be drawn as flat polygons in 3D space. Again this representation is ambiguous and does not show the real shape of the polygons on the sphere, as shown in Figure 5.3.

The next step was to substitute the straight lines for real arcs. To do this, the OpenGL function `gluPartialDisk` was the most adequate option. It draws a flat area delimited by two circles, and two angles, effectively a segment of a disc, on the XY plane. It takes as parameters a pointer to a `GLUquadric` structure, the radii of the outer and inner circles of the disc, the initial and final angles around the XY plane (specified in degrees), and two other parameters for the subdivision of the arcs (Figure 5.4 shows the arc and the related parameters). These will determine how many polygons compose the arcs. The function specification is the following (Schreiner *et al.* 1999):

(a) Normal vectors joined by straight segments

(b) Wireframe sphere for reference

Figure 5.2: Initial visualisation of the spherical indicatrix.

Figure 5.3: Initial visualisation of the areas of the Polyhedral Gauss Map.

```
void gluPartialDisk(
        GLUquadricObj *qobj,
        GLdouble innerRadius,
        GLdouble outerRadius,
        GLint slices,
        GLint loops,
        GLdouble startAngle,
        GLdouble sweepAngle )
```

Using this function to draw an arc between two vectors requires setting the parameters for the angles correctly. In our implementation, the `startAngle` will always be equal to 0, while `sweepAngle` is the angle between the two vectors. The `outerRadius` is equal to the length of the vectors, while the `innerRadius` is a slightly smaller, to make it possible to identify each individual arc. Each arc in the indicatrix is coloured differently to distinguish them and identify the sequence order. The subdivision parameters `slices` and `loops` are set according to the length of the arc to make them look smooth without using too many polygons.

The first problem before using this function effectively is defining the orientation of

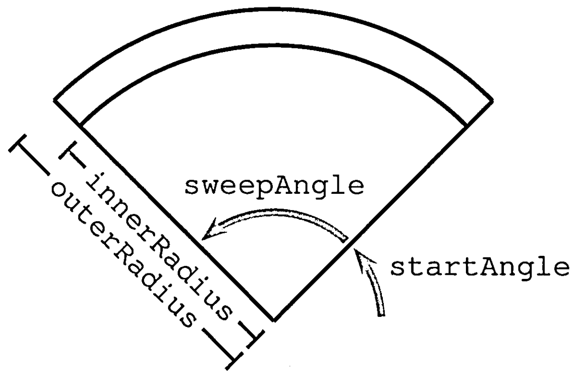


Figure 5.4: The parameters for the `gluPartialDisk` function.

the arc. The function `gluPartialDisk` will always generate the disc on the plane defined by the default X and Y axis in OpenGL. Placing the arc correctly requires calling the function once the appropriate transformations have been executed in the OpenGL scene to match the internal XY plane to the plane defined by the two vectors. This is done in OpenGL by using a rotation matrix. To do this we create a new function, called `rotateToXYplane`. It will also be useful later to draw the areas of the spherical polygons.

5.3.2 The `rotateToXYplane` function

The function takes as parameters the two vectors ($\mathbf{v_1}$ and $\mathbf{v_2}$) that define a plane. It will generate the rotation matrix $\mathbf{M_r}$ that will match the XY plane in OpenGL space with the plane specified by the two vectors.

Inside the function two vectors are defined to represent the Y and Z axis: $\mathbf{a_Y} = (0, 1, 0)$ and $\mathbf{a_Z} = (0, 0, 1)$. We compute a vector perpendicular to both $\mathbf{a_Y}$ and $\mathbf{v_1}$ as $\mathbf{n_1} = \mathbf{a_Y} \times \mathbf{v_1}$. The angle α_1 between $\mathbf{a_Y}$ and $\mathbf{v_1}$ is measured in radians. Next we compute a 3 by 3 *rotation matrix* $\mathbf{M_1}$ that specifies a rotation of α_1 radians around vector $\mathbf{n_1}$, where $\mathbf{n_1} = (u, v, w)$, $\psi = \sin\alpha_1$ and $\phi = \cos\alpha_1$ (Baker 2007):

$$\mathbf{M_1} = \begin{bmatrix} u^2 + (1 - u^2)\phi & -w\psi + (1 - \phi)uv & v\psi + (1 - \phi)uw \\ w\psi + (1 - \phi)uv & v^2 + (1 - v^2)\phi & -u\psi + (1 - \phi)vw \\ -v\psi + (1 - \phi)uw & u\psi + (1 - \phi)vw & w^2 + (1 - w^2)\phi \end{bmatrix}.$$

An *inverse rotation matrix* $\mathbf{M_1'}$ is computed analogous with a rotation of $-\alpha_1$ around the same vector. When $\mathbf{v_1}$ is equal to the negative Y axis, the angle α_1 will be equal to π, and there are infinitely many rotation matrices. However in this case it is not necessary to have a rotation around $\mathbf{n_1}$, as the vectors are already aligned. In this case $\mathbf{M_1}$ and $\mathbf{M_1'}$ are converted to the identity matrix.

The vector v2 must be readjusted to the new coordinate system. This is done by multiplying the vector by the inverse rotation matrix, as follows:

$$v'2 = M'lV2. \qquad (5.3.1)$$

After this we find n2 = *V2* x aY, and the angle *a2* between az and n2 is calculated. A second rotation matrix M2 is computed with a rotation of *a2* around ay- Finally the desired rotation matrix Mr is obtained from the multiplication of the previous two rotation matrices, as such:

$$Mr = MxM2 \qquad (5.3.2)$$

This process is done for every pair of vectors in the normal star of the vertex. The resulting rotation matrix Mr is used to transform coordinate system of OpenGL, so that the function `gluPartialDisk` will generate the arcs in the correct locations. The results can be seen in Figure 5.5.

Figure 5.5: Using arcs to join the normal vectors.

5.3.3 Spherical polygons with OpenGL

With the Indicatrix in place it is possible to visualise the spherical polygons on the surface on the sphere. However it is still necessary to fill in the appropriate areas, and assign to them a colour corresponding to the orientation of the curvature to completely represent the curvature of the mesh vertex being analysed.

As is the case for the computation of the PGM presented in Chapter 4, the vertices of the spherical polygons are in fact the normal vectors of the faces around a vertex in the mesh. Thus the vertices of spherical polygons are represented with the same notation as for the normal vectors (i.e. ni).

OpenGL has no predefined functions to draw a spherical polygon. The method that can be used to generate spherical polygons is to use *clipping planes* (Woo *et al.* 1999) to 'cut' them out from a sphere. A clipping plane splits space into two regions, where only the objects in one of these two regions will be drawn, and objects on the other side will be ignored during the rendering process. OpenGL relies normally on clipping planes to delimit an area in 3D space that will be rendered. In any scene there are six default clipping planes that define the *viewing volume* (Watt and Watt 1999). Besides these, additional clipping planes can be defined. These can be enabled temporarily before drawing an object, and disabled afterwards. An object drawn while the clipping planes are active will be 'cut' by the planes.

The function `glClipPlane` has the following specification:

```
void glClipPlane(
        GLenum plane,
        const GLdouble *equation )
```

It receives as a parameter a plane,

$$Ax + By + Cz + D = 0, \tag{5.3.3}$$

as an array containing the four coefficients (A, B, C, D). The plane defines the half space that will be hidden. The new viewing volume will be specified by the intersection of the default viewing volume and the half space determined by the plane. By default the plane equation of all clipping planes is (0,0,1,0), equivalent to the XY plane. Using the function `rotateToXYplane`, the spherical polygons are drawn by aligning the XY plane with each of the arcs that delimit the sides of the polygon.

However, there are immediate drawbacks to drawing a full spherical polygon using clipping planes for each of its sides. Firstly, it is only possible to define a limited number of additional clipping planes, normally six, in OpenGL. This limits the number of sides a spherical polygon could have. Secondly, non–convex polygons are impossible to draw, since the clipping planes of the concavities would remove other parts of the polygon. Such non-convex spherical polygons are present in the PGM of some saddle vertices, such as the one in Figure 5.6.

The solution to both of these problems is to triangulate the spherical polygons before drawing, and then render them as several individual spherical triangles. This requires only 3 clipping planes per triangle, which can be reused for all triangles. Any spherical polygon of any complexity can be triangulated regardless of it being convex or not.

5.3.4 Triangulation of spherical polygons

The triangulation of spherical polygons is a modification of the ear–slicing triangulation algorithm for flat polygons in 2 dimensions (Fournier and Montuno 1984), (Elgindy *et al.*

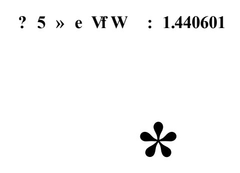


Figure 5.6: Truncated saddle vertex. Its PGM has a non-convex spherical polygon.

1989). The main difference when applying this algorithm to a spherical polygon is the computation of the angles at the vertices. Our implementation consists of the two functions:

- `triangulateSphericalPolygon`
- `divideSphericalPolygon`

The former receives the list of vertices of the spherical polygon and returns a list of triangles that form the polygon. Each triangle is itself a list of three vertices. The latter function takes as a parameter the list of vertices of the spherical polygon and returns two smaller lists of vertices that separate the input polygon in two parts. This is done by finding three adjacent vertices ni, n2 and n3 that form a convex angle at n2. They represent a triangle and the algorithm next checks if there are any other vertices inside the triangle. The function `point InsideSphericalTriangle` checks if a vertex ni is inside the triangle by using the concept of orientation. If all three orientations p(rii, ni, n2), *p{*rii, n2, n3), p(rii, n3, ni) are equal then rij is inside. If any of them is different, then rii must be outside the triangle.

If there are no vertices inside, the triangle is cut from the polygon along a new edge n!n3. If there are any vertices inside, the one closest to n2 is chosen as n4 and the triangle is split along the edge n2n4. Every time a triangle is split from the polygon, the two vertices on the shared edge will be duplicated. Figure 5.7 shows both cases, using planar polygons for simplicity.

Initially the function `triangulateSphericalPolygon` makes a call to function `divideSphericalPolygon` and then recursively calls itself with each of the two smaller polygons returned. When `triangulateSphericalPolygon` receives a polygon with three vertices, it inserts the triangle into the final list of triangles and returns that list.

(a) No points inside initial triangle (b) Points inside

Figure 5.7: Two cases of ear-slicing for polygon triangulation.

With the final list of triangles the spherical polygon is drawn one triangle at a time. `rotateToXYplane` is called for each pair of vertices in in every triangle, followed by the enabling of a clipping plane. With the three clipping planes in place drawing a sphere will result in only the spherical triangle being rendered, as shown in Figure 5.8. In this manner all possible spherical polygons can be accurately visualised.

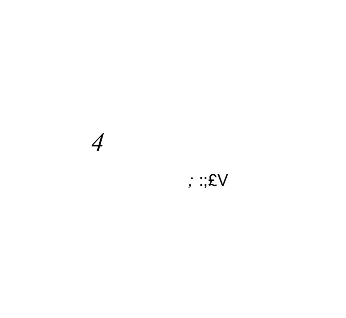


Figure 5.8: A triangle on a sphere, drawn using 3 clipping planes to segment a red sphere.

5.4 Interaction with the display

The visualisation program includes various functions to provide a better understanding of the PGM. Here we mention some of these features of the program.

- The user has the possibility to rotate, scale or translate the object being analysed, within the display space. This allows viewing particular details of the mesh or PGM.
- The screen space is split into two *viewports,* as allowed by OpenGL. The left viewport shows the polygonal mesh, and the vertex selected, if any. The right viewport

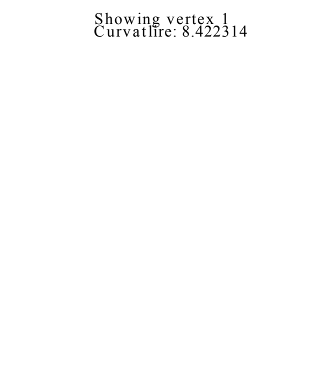
displays either the Polyhedral Gauss Map of the mesh, or of the selected vertex. Both viewports are subjected to the same transformations, thus rotation, translation or scaling affect both the mesh and the PGM.

- The vertices of the mesh are coloured according to the type of vertex as determined by the program. This permits the visualisation of the object in regions of curvature. The types of vertices detected are of positive, negative, mixed and zero curvature.
- Additionally the program can display the vertices coloured according to the vertex classification provided by the Angle Deficit method, which consists only of vertices of zero, positive and negative curvature.
- The user can select any vertex for individual analysis. When a vertex is selected, its measured TAC is printed on the screen, the star of the vertex is highlighted as wireframe, and the PGM is shown for that particular vertex.
- To better identify normal vectors and faces, the ID numbers for both can be shown on the screen. The mesh in the left viewport shows the ID numbers of the faces, and the PGM in the right shows the numbered normal and intersection vectors. This can be seen in Figure 5.9, where intersection vectors have a negative ID to distinguish them from face normals.

•howing vertex 1
lurvatUre: 1.497827

Figure 5.9: Showing the ID numbers of faces and vectors. Intersection vectors have negative ID numbers.

- Since the curvature components of different sign in a PGM can overlap, the program permits toggling on or off the display of spherical polygons depending on their sign. It is possible to show either only positive, only negative, both kinds of polygons or none. This reveals sections of curvature that may be hidden, as shown in Figure 5.10.
- The program can show the projection of a vertex link into a plane, obtained with the procedure explained for Definition 3.4.1. Figure 5.11 shows a vertex and its *projected link.*
- There are many other options of data to display. It is possible to toggle the display



(a) Both curvature components

(b) Only positive polygons

(c) Only negative polygons

Figure 5.10: Toggling of the display of spherical polygons according to their sign.

(a) Vertex star

(b) Projected link

Figure 5.11: Display of the *projected link* of a vertex.

of normal vectors on the mesh, the display of the indicatrix for a vertex in the left viewport, or the PGM in the right viewport. With some modifications the program can show only one viewport with either the mesh or the PGM.

5.5 Solving problems of the Gauss Map using the visualisation

The ability to visually analyse the Polyhedral Gauss Map of vertices is very valuable to evaluate the numerical results obtained from the algorithm. It was also used during the development of the algorithms to show problems and errors in the computations. Since the graphics generated are based on the data structures, they are an entirely reliable representation of the results obtained. Errors in the display can be traced back to incorrect computations.

The first problems detected with the visualisation were the cases when various face normals around a vertex would lie in the same plane. This creates degenerate spherical polygons, which in the display appear in the opposite side of the sphere as the normal vectors.

The multiple cases of arc intersections would also produce visual errors. If the intersections at the arc endpoints are not correctly handled, the PGM would show missing spherical polygons that were being delimited in the indicatrix.

Initial problems with the splitting of the spherical polygons were identified by visually inspecting the results presented by the program. In many cases as well, the spherical polygons were drawn on the opposite side of the sphere as the spherical indicatrix. This problem occurred when the areas had been given an incorrect sign, and the indicatrix was analysed in the inverse order. In this way both the detection of arc intersections and the separation of the individual polygons were improved after analysing the images produced.

The solutions to the special cases mentioned in the previous chapter were determined based on the visual observations. Such is the case for the 'figure–8' vertices and the various types of arcs intersections at the location of multiple vectors. The visualisation also made clear the condition when positive and negative areas cancel each other, thus reducing the total curvature computed for a vertex.

For the application of the curvature for mesh simplification that will be introduced in Chapter 6 the visual display of the PGM immediately reflects the changes on the properties of a vertex affected by the simplification. The information of the mesh is recomputed after each step and graphically reflects the changes made. The visualisation permits identification of the problem if the decimated mesh is not correctly reconnected. Through this procedure, the need for an inverse edge flip function (explained in Section 6.4.1) was identified and implemented.

5.6 Conclusions

We have presented the development of a visualisation program for the Polyhedral Gauss Map. The program has been progressively improved to give an accurate representation of the spherical polygons on the PGM. The difficulties to generate the spherical polygons in OpenGL have been explained, and also the techniques employed to solve the problem.

Spherical polygons are created by the use of clipping planes and a function that generates a rotation matrix to match the XY plane in OpenGL space with any plane defined by two vectors. Additionally the spherical polygons are triangulated to cope with the limitations of the clipping planes. These methods prove to be very effective at reproducing all possible spherical polygons correctly.

The interactive capabilities of the program have been presented. The visualisation permits selecting the information to be displayed. It shows both the object and the Polyhedral Gauss Map of selected vertices. Colours are used to distinguish vertices according to their type classification.

Graphical visualisation simplifies the understanding of complex concepts, such as the Polyhedral Gauss Map. In this particular case even a 2D representation is difficult to achieve and is generally incomplete. A 3D display allows viewing the projection of the PGM on a sphere from any angle, which gives the user a full scope of the indicatrix and the spherical polygons. The Polyhedral Gauss Map visualisation helps to correctly identify not only the curvature of a vertex but also to analyse the structure of its star, as in the case, for example, of 'figure–8' vertices.

The visualisation presented in this chapter is effective and accurate, and allows the validation of the numerical results computed by the `gaussMap` program. The curvature measures for any vertex are immediately reflected in the visualisation. This also helps with the identification and correction of errors in the algorithm.

5.7 Future research directions

While the program currently allows the user to select the display of positive, negative or both kinds of spherical polygons at once, at some points it is helpful to have it show individual areas identified for the curvature of a vertex. A minor modification to the program can be used to show all areas with a slight scaling to separate them from the surface of the sphere, so that it is possible to identify them. However this solution is not ideal, as the areas often overlap, or become so many that they cannot be shown simultaneously (see Figure 5.12).

ihowing verti
.urvature (PC

Figure 5.12: Separation of the spherical polygons by scaling them by a small factor.

A better solution would be to implement a mechanism to toggle the display of individual areas on or off. This implies having separate storage for each area, and generating individual display lists for each. This could possibly impact the performance of the program.

Chapter 6

Triangle mesh simplification using the Polyhedral Gauss Map

6.1 Introduction

Many applications of computer graphics rely heavily on polygonal meshes to represent objects. They provide great flexibility in representing a variety of shapes, from simple to very complex ones, and in varying degrees of detail. Modern methods for generating 3D meshes of objects involve high definition scanning of data, providing very detailed meshes of the source objects. Often this produces an enormous amount of data, which is difficult to handle by current computer systems for real time applications. Meanwhile, the data acquired with these methods may be redundant or excessive to portray the basic shape of the object.

For these reasons, it is desirable to simplify the polygonal geometry in a model. *Mesh simplification* is the process of decreasing the number of components from a polygonal mesh, reducing its overall complexity, while at the same time still providing a good visual representation of the original object. The necessity to simplify a model may come from limited resources: either storage, transmission bandwidth, processing power or the requirement to display in real time. All of these problems are solved or lessened by having a smaller mesh that still appropriately represents the original object.

Several different techniques already exist to simplify a polygonal mesh. The approaches taken vary greatly from one method to another, however a common trait to all of them is that they must identify which parts of the object are important to the shape, and which can be safely removed. This is generally done by assigning a relevance weight to the elements of the mesh, that is, to vertices, edges or faces. The curvature of an object is a good measure of the behaviour of the shape, and thus an important characteristic to consider when modifying a mesh model. Curvature can be calculated for the whole object, specific regions, or individual vertices or edges.

The most commonly used measure of curvature in existing simplification algorithms is the Angle Deficit method. However, as previously explained, it suffers from not fully reflecting the local shape structure around vertices and therefore is not adequately suited for decimation based on curvature. This chapter presents the use of the Polyhedral Gauss Map method as a better tool to guide mesh simplification algorithms. The Total Absolute Curvature thus measured is an ideal measure for correctly determining the importance of a vertex for the shape of the mesh. Figure 6.1 shows an example of mesh decimation using the TAC.

(a) Original (2,832 vertices)

(b) 88.3 %decimation (332 vertices)

Figure 6.1: *Triceratops* **model: shaded (left), wireframe (right).**

We present a set of weight values for the vertices of a polygonal mesh to specify how relevant each vertex is to the overall shape of the object. All of these weights are based on the TAC and in most cases also include the area around the vertex. In general we refer to this set of vertex weights as the *Weighted Total Absolute Curvature,* abbreviated as WTAC. These measures weigh the complexity of the vertex with the visual impact it has on the overall surface of the object. We present the reasoning behind the selection of four different weights, and compare the obtained results when using them to guide the simplification process.

We also use the TAC to optimise the corresponding region in the simplified mesh after a vertex removal by using edge flipping, as presented in (Alboul 2003). This procedure ensures a further optimisation of the mesh with respect to curvature, since the higher TAC yields more complex mesh geometry. As the TAC of a mesh region is computed by summing the TAC of the vertices belonging to this region, our method can be considered also as *shape–preserving*. It emphasises domains of the most prominent curvature of the region, that is either positive or negative, while decreasing or completely eliminating curvature domains of the opposite sign.

We also compare the results obtained using either the common Angle Deficit method or the new PGM. We show how using the TAC curvature improves the results when using only the Gaussian curvature obtained with AD. We produce simplified meshes using both measures, and the results are compared numerically against the original models to draw a conclusion.

The measurement of vertex weights presented here can be applied to various mesh simplification algorithms based on decimation or edge collapse to improve their selection of the vertices to simplify. To demonstrate the advantages of using the TAC in a simplification algorithm we have implemented a simple vertex decimation program to test and compare the obtained results on different models.

The main contributions are the application of the TAC measure to a mesh simplification algorithm, proving its advantages over decimation algorithms based on existing curvature measurement methods. The TAC is also used to optimise the simplified mesh by means of edge flips.

The structure of the chapter is as follows: Section 2 presents the existing research in the field of mesh simplification. Section 3 shows the selection of the vertex weight measures and compares them. The details of our decimation program are given in Section 4, followed by experimental results in Section 5. Section 6 deals with potential applications of the developed techniques, Section 7 presents the conclusions and future research directions are pointed at in Section 8.

6.2 Previous research on mesh simplification

Extensive research has already been carried out in this field. There are several different approaches to simplify a polygonal mesh. The papers (Cignoni *et al.* 1998) and (Luebke 2001) review the most important of these methods, and compare their advantages and shortcomings.

Cignoni *et al.* do a comparison of the error difference, from the original models to the simplified ones, and the time taken to obtain the results, using several different algorithms. In order to achieve this, they created the `Metro` tool (Cignoni *et al.* 1998), which has since been used by others to compare any new algorithms with existing ones.

As explained in these papers, simplification techniques vary in: *optimisation goal*, which can be size minimisation given an error bound, or error minimisation given a target object size; *local* or *global optimisation*; *preservation of the original object's topology*; *maintaining the original vertices* or *remeshing the model.* Some simplification methods also perform *view dependent* decimation. That is, a model is decimated more in areas far away or hidden from the current perspective. This permits a larger reduction of the mesh, but requires that the mesh be re–simplified if the view direction changes.

The algorithms tested in the aforementioned works include Mesh Decimation, Simplification Envelopes, Multiresolution Decimation, Mesh Optimization, Progressive Meshes and Quadric Error Metric Simplification. A detailed description of these methods can be found in the aforementioned papers. In the tests, Mesh Decimation produces the largest error with respect to the original model, although it is by far the fastest algorithm. Other techniques perform better in most respects, specifically Quadric Error Matrix (QEM) (Garland and Heckbert 1997), which is generally considered as a very effective simplification method and used as a parameter of comparison for newer techniques.

The first method for *decimation* was proposed by (Schroeder *et al.* 1992), and is based on progressive removal of specific vertices from a mesh. All vertices are classified according to their local topology, and are handled accordingly. Vertices are labelled as 'simple', 'complex' or 'boundary'. Complex ones are generally non–manifold vertices, and are left untouched. Simple and boundary vertices are treated differently. Simple vertices are further classified according to the number of feature edges connected to the vertex. Vertices are selected for decimation according to a distance metric. For simple vertices the parameter is the distance from an average plane; for boundary vertices, it is the distance to the line connecting the neighbours along the boundary. The vertex with the shortest distance is selected for removal, along with the triangles surrounding it. The hole produced in the mesh is filled using a recursive splitting of the remaining region into triangles. This method generates a subset of the original vertices, not adding any new ones. It also preserves the topology of the object.

A scheme to re–tile a polygonal mesh with less vertices was proposed by (Turk 1992). Here a new set of vertices is distributed over the original model, the new vertices will repel one another to adequately cover the whole surface. Next the surface is retriangulated using both new and old vertices to preserve the shape, and later the old vertices are removed.

An extra step is also included which uses an estimation of the curvature to aid in the distribution of the new vertices over the surface.

(Cohen *et al.* 1998) developed a method to preserve more accurately the appearance of a simplified model, by using texture and normal maps, in addition to the polygonal mesh. They initially compute the shading colours and the normals of the full model, and convert this information into maps that contain this important visual information. Then the mesh can be simplified to several levels of detail. Applying the maps with the original information will make the model look very similar to the full detail version. The main improvement for this method is the *texture deviation metric* which is used to assign texture coordinates to the remaining vertices in their corresponding position with respect to previous vertices. This metric can also be used to guide the simplification algorithm.

(Lindstrom and Turk 1998) implemented an algorithm using edge collapses where the position of new vertices is obtained from the optimisation of a few simple geometrical properties of the local neighbourhood of the edge. They use preservation and optimisation of the volume and boundary related to the edge. The same optimisations are used to give weights to the edges for the selection of the collapses.

A probabilistic approach is employed by (Wu and Kobbelt 2002) to reduce resource requirements of a simplification algorithm. They use a Multiple-Choice Algorithm to randomly select a few candidate edges and select from those the best option. This avoids having to keep an updated queue of the best possible option at all times. Doing so they significantly speed up the simplification process and reduce the memory requirements, while obtaining a good degree of simplification, comparable to the QEM method.

In (Kim *et al.* 2002) a measure based on curvature is used to assign a cost to the edges, and to select the ones to be collapsed. The authors make use of both the Gaussian and mean curvatures on the mesh. After deciding on an edge to collapse, a new vertex is generated in place of the two edge vertices. The location of this new vertex is found using a butterfly subdivision mask. Using curvature to decide on the geometry to eliminate, they prove that important features of the object are preserved after heavy simplification.

(Hussain *et al.* 2004) propose a simplification driven by half–edge collapses that keep at least one of the vertices of the edge removed. They use a metric based on the angle difference from the original faces to the ones that will be created after the collapse; effectively an approximation of the curvature of the region, although not very accurate, but useful again in preserving important geometry. This implementation competes in performance with QEM but claims to require less memory to store data.

Our method can be considered, to a certain extent, as a generalisation of the one proposed by Schroeder et al. We also present a novel contribution to the selection of vertices, based on curvature measures, used to drive decimation methods. This research has been partially presented in (Echeverria and Alboul 2006).

6.3 Curvature as decimation parameter

Most mesh simplification algorithms based on vertex decimation assign a weight to each individual vertex, further referred to as the *relevance weight*, that signifies its importance to the shape of the object. If the value is smaller than a certain threshold, then the vertex can be removed without significantly altering the mesh, while if the weight is larger than the threshold, the vertex must be kept.

The values used as weights for the vertices are different for each implementation, but generally are based on the geometrical properties of the surrounding region, such as distances, areas or measures of curvature. We present a set of new vertex weight measurements to guide the decimation procedure, based on the Total Absolute Curvature. We refer to these measurements in general as the *Weighted Total Absolute Curvature*, denoted as WTAC. The WTAC measures consist of multiplying the TAC by some factor that represents the geometrical area of each vertex. Four different weights are tested and compared in our decimation program. We first present definitions of the area computations used as factors for the WTAC.

Definition 6.3.1. Cone area: We define T_ν as the list of the n triangles in $star(\nu)$, so that $t_i \in T_\nu$, then the *cone area* of vertex ν is the sum of the areas of all the triangles in $star(\nu)$, and is denoted as $A_C(\nu)$. Thus,

$$A_C(\nu) = \sum_{i=1}^{n} area(t_i). \tag{6.3.1}$$

Definition 6.3.2. Projected area: We define a planar polygon P_ν as the projection of the link of ν into the plane defined by the vertex ν and its normal vector N_ν. The projection of the vertices in $link(\nu)$ is obtained according to the method described in Definition 3.4.1. The *projected area* of vertex ν is computed as the area of the planar polygon P_ν, and is denoted as $A_P(\nu)$.

It is possible, however, that the plane chosen for the projection will lead to self–intersections in the polygon P_ν, specially when dealing with very complex saddle type vertices. In these cases the area computed will be unreliable. In order to overcome the aforementioned difficulties and shortcomings related to the determination of the flat projection, we also use the well–known isoperimetric inequality (Osserman 1978) $L^2/4\pi \geq$

A (where L is the perimeter length of the polygon in the plane, and A is its area) in order to get an approximated area of the flat projection. In our case we use as L the length of the $link(\nu)$.

Definition 6.3.3. Length–area inequality: We define E_ν as the list of the n edges in $link(\nu)$, so that $e_i \in E_\nu$, then the *length–area* of vertex ν is obtained by using the formula:

$$A_L(\nu) = \frac{1}{4\pi}\left(\sum_{i=1}^{n} length(e_i)\right)^2 . \tag{6.3.2}$$

Using these definitions of the areas around a vertex, the four versions of WTAC are as follows:

6.3.1 TAC weight

This parameter uses exclusively the Total Absolute Curvature of the vertex ν as its relevance weight, denoted as $TAC(\nu)$. In this case other properties of the vertex are discarded, and only the curvature has an effect on its importance.

6.3.2 ATAC weight

Multiplication of the TAC by the cone area around vertex ν:

$$ATAC(\nu) = TAC(\nu) \cdot A_C(\nu). \tag{6.3.3}$$

This measure takes into account the complexity of the vertex and its visual importance in terms of the area and curvature simultaneously. It gives more priority to vertices with large incident triangles making them less susceptible to elimination, but those with small stars are more likely to be removed.

6.3.3 PTAC weight

Knowing the cone and projected areas we can use a *normalised area* of ν as the parameter to multiply TAC:

$$PTAC(\nu) = TAC(\nu) \cdot \frac{A_P(\nu)}{A_C(\nu)}. \tag{6.3.4}$$

An important feature of this parameter is that the values of the factor will always lie in the half–open interval $(0, 1]$. It is clear that if the vertex is in a flat region, both the cone and projected areas will be the same, making the factor 1. If the vertex has a very sharp cone, the projected area will be much smaller than the cone area, making the factor close to 0.

The PTAC weight therefore considers the local properties of the neighbourhood around the vertex, but disregards the size of the region. It ensures that vertices with similar stars at different scales will be treated in the same way. This may result in the removal of a vertex in a relatively flat region that leaves a very large hole in the mesh. Such a hole can be difficult to properly retriangulate and lead to a more expensive optimisation of the initial triangulation.

As mentioned before the computation of the projected area is not always reliable, depending on the plane chosen for the projection of the vertices.

6.3.4 LTAC weight

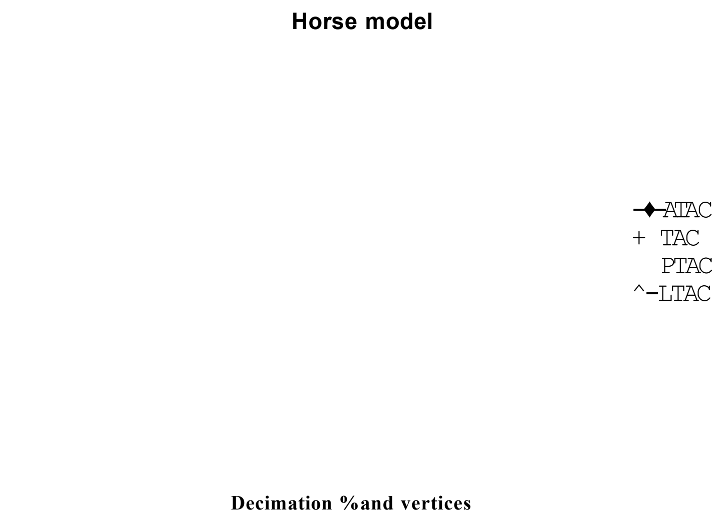
Using the projected area and the length–area inequality, we establish an alternative version of the normalised area:

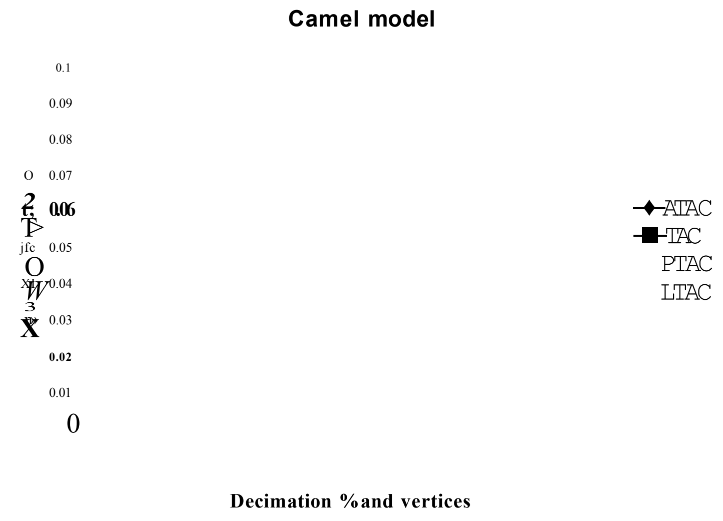
$$LTAC(\nu) = TAC(\nu) \cdot \frac{A_L(\nu)}{A_C(\nu)}. \tag{6.3.5}$$

This weight measure attempts to solve the shortcomings of the PTAC, by computing a planar area by means of the perimeter of the star instead of the projection into a plane. This avoids any self–intersections in the projection. In this case the factor is not guaranteed to lie in the interval $(0, 1]$. However this weight also disregards the scale of the neighbourhood of the vertex.

The `Metro` tool was also used to compare the different weights used to guide the decimation. Various polygonal meshes were increasingly simplified using the four vertex weights. Then each of the decimated meshes was compared to the original. The Hausdorff distance is measured as the maximum difference between the original and the simplified mesh. A smaller distance indicates a closer shape to the original. Figure 6.2 shows the `Metro` results for three different meshes using the various vertex weights. In the tables the X axis represents the number of vertices in the simplified mesh, and the Y axis shows the value of the Hausdorff distance. The *Decimation %* parameter shown refers throughout this chapter to the percentage of the vertices removed during the decimation.

The results from this comparison, both visually and numerically indicate that the best results are produced by using the ATAC weight. Using normalised areas, as in the PTAC and the LTAC, preserves better the finer detail in the mesh, but produces much larger errors in the areas of small curvature. This is due to the fact that areas of relatively small curvature will eventually be reduced to a single vertex with small curvature and several very large incident triangles. This single vertex represents the shape previously represented by the whole region. If it is removed, the appearance of the object is altered significantly. This is illustrated in Figure 6.3.





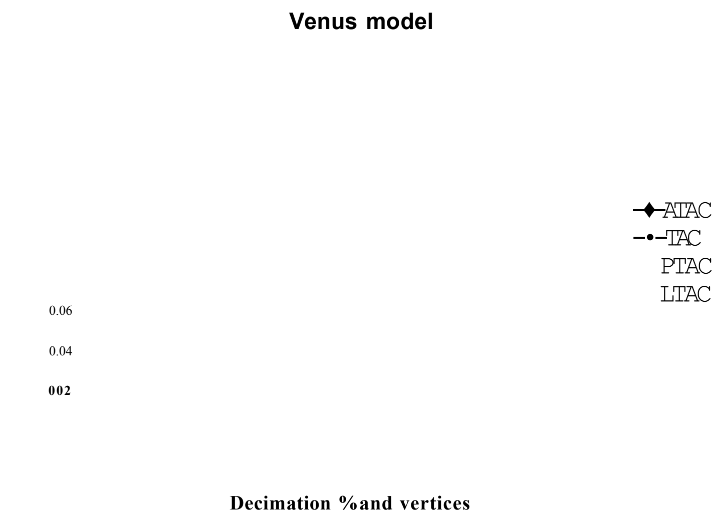


Figure 6.2: Comparison of the vertex weight parameters: TAC, ATAC, PTAC and LTAC.

(a) Vertex with many large incident faces

(b) Removal of the vertex changes the shape

Figure 6.3: *Triceratops* **model simplified with LTAC to 84.4%.**

6.4 Implementation of a vertex decimation algorithm

For the purpose of testing the curvature metric on simplification, a simple vertex decimation algorithm is used, similar to the one used by Schroeder *et al.*

The algorithm makes several passes over the whole dataset, removing each time the vertex with the smallest weight. After removing the vertex it updates the mesh by re-triangulating the 'hole' in the mesh. The retriangulation is done in such a way that the curvature of the mesh does not increase.

The process of vertex decimation is as follows:

- The polygonal mesh to be decimated is analysed as presented in Chapter 4. This will provide the arrays of vertices and faces, the TAC for each vertex and all the extra information needed: Angle Deficit, list of triangles in the star, list of neighbours, artificial vector normal to the vertex, projection of the neighbours and feature edges.
- After gathering all the above-mentioned information, an extra parameter is assigned to each vertex: the *relevance weight,* described in general as the WTAC. A vertex *v* will be selected for removal if it has the smallest WTAC value from all the remaining vertices in the mesh. The WTAC value is computed according to the criteria selected. For the special case of vertices on the boundary of an open object, the parameter used for selection also considers the Angle Deficit, since the Gauss Map area at these vertices can be equal to zero.
- All vertices are sorted by increasing value of the WTAC, to simplify the process of selecting the next vertex for decimation. To this end, two new arrays are created. One of them has the vertices ordered by WTAC, and includes the vertex ID number and the value assigned to it. The other array is ordered by vertex ID numbers, and

contains the index location of each vertex in the other array. This second array is used to simplify searching for other vertices.

Vertices to delete are taken from the top of the array. The removed vertices have their index changed to a negative sign, to indicate they are no longer valid for the next decimations. After each removal the WTAC array is updated with the new values for the affected neighbour vertices. For each of the affected vertices the new WTAC value is computed, then the displacement on the array is measured. All the entries in the array between the original and new positions are shifted one index, and finally the affected vertex is re–inserted in its new position.

- Two different approaches were taken to updating the geometry of the region around the vertex removed. The first method involves removing all the faces incident on the deleted vertex and retriangulating the hole left. The second technique is more similar to the half–edge collapse, where only two triangles are removed and the rest are modified to fill the gaps.

  For the hole retriangulation algorithm, when a vertex ν is selected for decimation, itself and all the triangles $t_i \in star(\nu)$ are marked as deleted. Then, for each $\nu_j \in link(\nu)$, the triangles t_i are removed from the lists of their corresponding stars.

  The hole left in the place of $star(\nu)$ is then retriangulated using the projection of the star. The ear–slicing algorithm for planar polygons is used to produce new triangles on the projection (Fournier and Montuno 1984), (Elgindy *et al.* 1989). The resulting list of planar triangles is then translated to the coordinates of the corresponding neighbours in 3D space, to create a new set of triangles in the mesh. Using the projected polygon, the problem of triangulating is significantly simplified over doing it directly in 3D space. However, the same drawback previously mentioned is found when the projection plane is not properly chosen and self–intersections occur on the projected polygon. In these cases the computation of the area of the polygon will be incorrect and the triangulation will consequently produce non–manifolds, which will require a few more tests to identify such occurrences and correct them, making the program more complex and error prone. This problem is recognised in (Lee *et al.* 1998), and an alternative projection into a conformal map is proposed.

  To avoid the retriangulation, a second approach was taken, consisting of performing half–edge collapses. Here another vertex ν_c is chosen from the neighbours of the deleted vertex ν. Only the two triangles that share the edge $\overline{\nu\nu_c}$ are deleted, and the rest of the triangles incident on ν are updated to be now incident on ν_c. This automatically takes care of closing the hole, and does not require the creation of new triangles in the program. This method is also likely to create some degeneracies, mainly in the form of adjacent triangles facing opposite directions but lying in the same plane. The program will check that two identical faces are not facing in opposite directions. There is no check in the case the triangles share only two vertices, however these cases are easily taken care of by optimising the curvature of the triangulation.

- Once the initial retriangulation is complete, the curvature data for all the neighbour vertices ν_j is recalculated using the new triangles.
- An additional step optimises the resulting triangulation using edge flips. The objective is to reduce the curvature of the proposed triangulation. A list of edges is created only for the triangles in the polygon that now replace the previous $star(\nu)$, the edges on the boundary of the polygon are then discarded and only the edges lying inside the polygon are kept. Each edge structure contains the two vertices that define it (its *endpoints* ν_{e1} and ν_{e2}), the two incident triangles (f_1 and f_2) and the two *opposite* vertices (ν_{op1} and ν_{op2}). These elements can be seen in Figure 6.4. For each edge, the current curvature is measured as the sum of the curvatures at four vertices: the two endpoints and the two opposite vertices. After a flip, the curvature of these four vertices is recomputed. The flip is accepted if the sum of curvatures is smaller than the original, otherwise the edge and incident triangles and vertices are returned to the previous state. This process is repeated until none of the edges will flip to a smaller curvature configuration. Further details of the edge flip procedure are given in Section 6.4.1.
- The mesh is now in a stable condition, and a new vertex can be selected for removal.

6.4.1 Optimisation of the triangulation using edge flips

We optimise the initial retriangulation of the hole by using edge flips to minimise the TAC of the mesh. This preserves topology as well as the shape up to extreme levels of decimation.

The technique of *edge flipping* was first presented in (Lawson 1972), has been extensively explored in (Choi *et al.* 1988), (Dyn *et al.* 1993), (Alboul *et al.* 1999), (Morris and Kanade 2000) and (Aichholzer *et al.* 2002). It has also been studied in (George and Borouchaki 2003) and (Aloupis *et al.* 2004). In this latter work it is proved that it is possible to go from any triangulation in the plane to any other using edge flips and point moves. (Brink and Alboul 2006) use edge flips to optimise the shape of a polygon mesh by minimisation of the Willmore energy, which is a function of the Gaussian and Mean curvatures that measures the deviation of a surface from a sphere.

In our implementation, when performing a flip, the opposite vertices are switched with the endpoints, and the incident triangles are updated, rotating their vertices in the Counter–Clockwise direction. If a flip is found to be invalid, then an inverse flip is performed that will restore the endpoints and opposite vertices to their original order. The vertices of the incident faces are rotated in clockwise direction to maintain consistency with the other edges. This will avoid producing non–manifold edges shared by more than two faces (Bærentzen 2006). If the flip is valid, the edges around the incident triangles must be updated as well, setting the correct opposite vertices. Also the indices of the incident triangles may change and have to be updated too. Figure 6.4 illustrates how the vertices and triangles of an edge are updated after a flip or inverse flip.

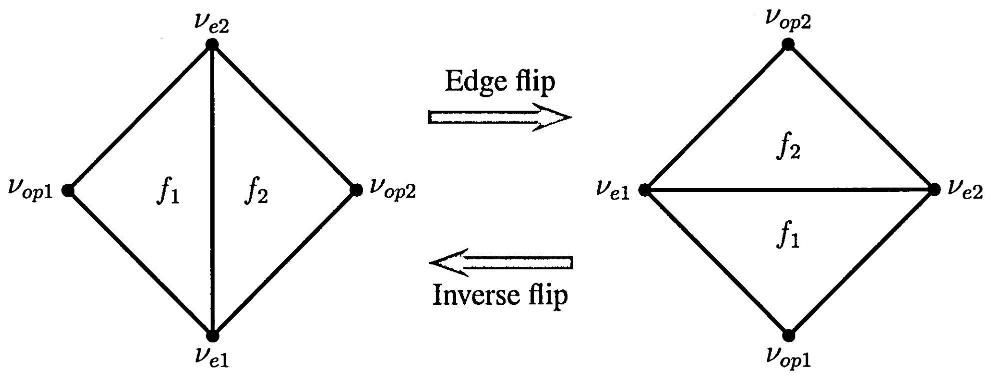


Figure 6.4: Edge flip and inverse edge flip of an edge.

Figure 6.5 shows an example of the removal of a vertex and how edge flipping can reduce the curvature of the resulting object. In the figure the original mesh is shown to the left, followed by two possible retriangulations after the removal of a vertex. Regardless of what the initial triangulation may be, both results are tested using edge flips, and ultimately the program will select the third mesh because of the smaller curvature.

Once a valid triangulation has been obtained, for every vertex ν_j the curvature and areas are updated according to the new mesh. The decimation procedure then continues until a termination condition is met. Otherwise, the algorithm continues until the shape of the object is completely lost. In the case of a genus 0 mesh the final 3D figure left is a tetrahedron. Termination condition for the decimation can be selected according to the requirements of the final mesh. It can be chosen as a maximum value of WTAC of the vertices to be deleted. Since all vertices are sorted by their WTAC value, once a certain threshold is reached, the decimation stops. Alternatively, the termination condition can be

(a) Original mesh
TAC = 13.817166

(b) Before flip
TAC = 14.177234

(c) After flip
TAC = 12.566371

Figure 6.5: Comparison of the triangulations obtained after removal of a saddle vertex.

a maximum error difference between original and decimated model, to avoid losing detail in the mesh. If the requirement is to reach a certain file size for the model, the limit can be set to a decimation percent or a set number of faces/vertices in the final model.

6.5 Experimental results

The resulting simplified meshes generated by the program are also stored as *OBJ* files that can later be compared with the original files, and also be used as input for the Gauss Map program to analyse their curvature. By changing the settings of the program, we determine what curvature measure will be used (AD or PGM) and also the WTAC for each simplification.

Figure 6.6 shows the results of using both curvature measures to decimate a polygonal mesh. The top row has the original mesh, while the centre row has the mesh decimated using PGM and the mesh in the bottom row was obtained using the AD. In the right column of images all vertices are colour coded according to the value of PGM, shown in red for positive curvature, blue for negative, and green for mixed (both positive and negative) curvature. Regions of similar curvature can be distinguished where several vertices of the same kind are linked together. It is noticeable how the model decimated using the AD presents more regions of mixed curvature, which represents irregular geometry. The mesh decimated based on the PGM presents more clearly defined areas of either positive or negative curvature that mimic those in the original mesh.

We evaluate numerically the benefit of using the PGM instead of the AD to guide a mesh simplification algorithm. The decimation algorithm was applied to various models using both curvature measures to assign the relevance weight to vertices and to select optimal edge flips. In both cases the formula to compute the weight for the vertices is the ATAC, but substituting the TAC with the Gaussian curvature and multiplying by the

(a) Original (11,362 vertices)

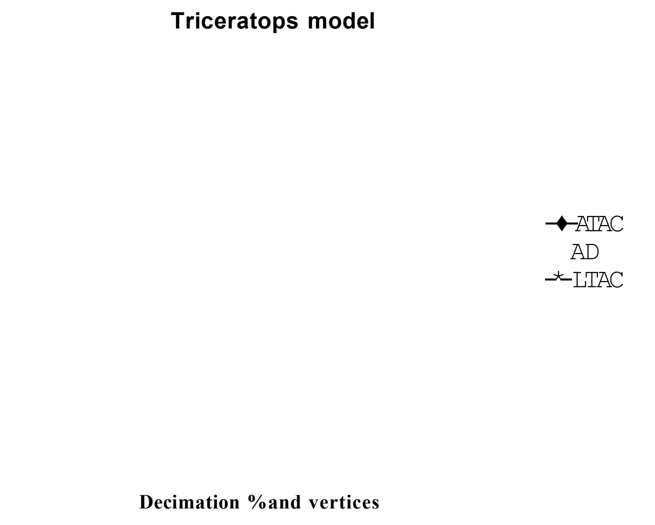
(b) 96.8 % decimation with PGM (362 vertices)

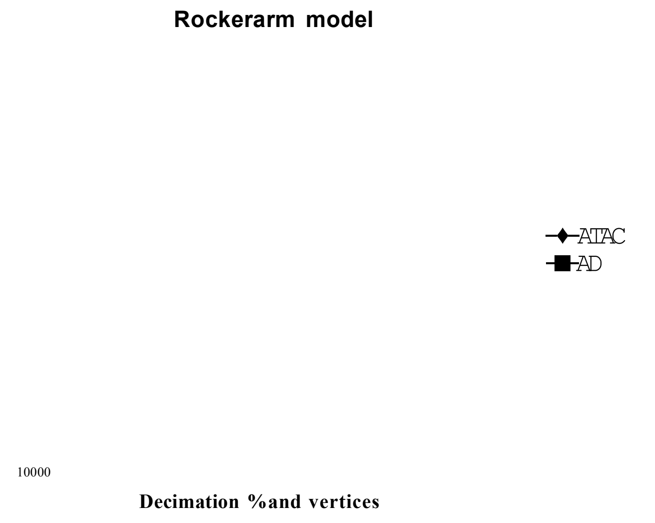
(c) 96.8 % decimation with AD (362 vertices)

Figure 6.6: *Venus* model: shaded (left), wireframe (centre), curvature coded (right).

cone area $Ac(v)$. The resulting simplified models were compared using the `Metro` tool. Figure 6.7 shows the results of using the decimation program on three different meshes.

As can be seen from the results, the use of the PGM instead of the AD produces a smaller error, and the difference in performance increases as the decimation progresses. We have not performed tests to compare the time difference to do similar levels of decimation using both measures. Constructing the PGM is more complex than the Angle Deficit computation. As we showed above the upper bound for its complexity is quadratic, however in practice the algorithm to produce the PGM is close to linear. Currently our application computes both the PGM and the AD simultaneously, and thus the time taken is the same in both cases.





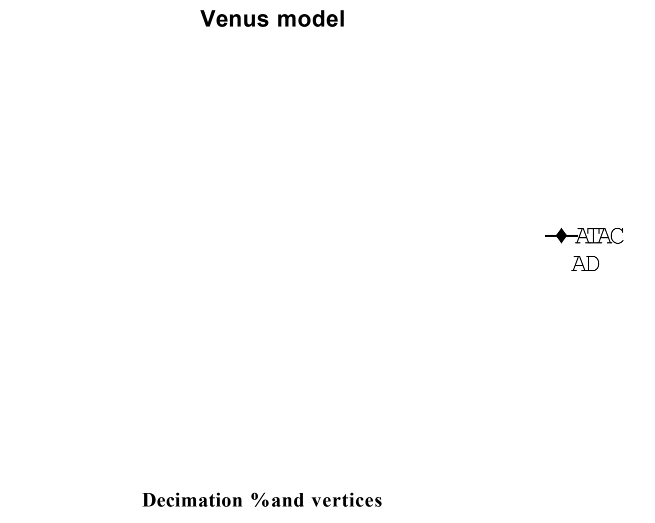


Figure 6.7: Comparison of the results obtained using PGM or AD as a measure for curvature in decimation.

During the decimation process the geometric properties of the mesh are altered in differing ways. Table 6.1 and Table 6.2 show the values of the curvature as computed with the PGM and the AD, total surface area of the object and Hausdorff distance to the original mesh. The TAC and the area both decrease with the loss of geometry, but the change in the TAC is more dramatic, because it is more sensitive to the changes in the geometry of the object. On the other hand, the Gaussian curvature of the whole mesh, as represented by the AD remains the same, at a value equal to 4π for closed meshes of genus 0.

Decimation	Vertices	TAC	AD	Surface Area	Hausdorff
0 %	11,362	77.045602	12.566380	30.136994	0
17.60 %	9,362	69.976640	12.566385	30.136968	0.002301
35.21 %	7,362	68.334118	12.566394	30.136956	0.003741
52.81 %	5,362	66.487840	12.566390	30.136331	0.004303
70.41 %	3,362	63.640547	12.566383	30.136502	0.007639
88.01 %	1,362	57.489905	12.566375	30.123395	0.021712
96.81 %	362	47.290207	12.566365	30.005538	0.076089

Table 6.1: Comparison of the curvature of the *venus* model with increasing decimation.

Decimation	Vertices	TAC	AD	Surface Area	Hausdorff
0 %	33,587	730.119954	12.566346	23505.330791	0
11.91 %	29,587	706.423968	12.566354	23505.398398	0.108923
23.82 %	25,587	703.362744	12.566344	23504.541003	0.153921
35.73 %	21,587	652.604403	12.566359	23503.028226	0.164336
47.64 %	17,587	587.653313	12.566370	23501.645275	0.164336
59.55 %	13,587	513.781304	12.566373	23500.950160	0.242121
71.46 %	9,587	442.974448	12.566368	23499.417764	0.247055
83.37 %	5,587	340.123710	12.566366	23498.766032	0.390726
95.27 %	1,587	175.379923	12.566375	23450.253691	1.292738
98.25 %	587	71.259997	12.566376	23343.561350	1.499396

Table 6.2: Comparison of the curvature of the *igea* model with increasing decimation.

Some of the existing mesh simplification algorithms produce a polygonal mesh with evenly distributed triangles around the final model, particularly those that re–sample the mesh and use new vertices. This presents some advantages, specially for more even texture mapping. However, some of the finer detail is lost from the original shape. The use of curvature to drive a simplification algorithm ensures that the reduced mesh will have more triangles where the shape changes more rapidly, and less triangles in flatter areas. As an example, we use the 'rockerarm' model in Figure 6.8. As can be seen in the close–ups of the decimated 'rockerarm' in Figure 6.9 the finer detail of the model is preserved even after several steps of decimation.

(a) Original (10,000 vertices)

(b) 95 % decimation with PGM (500 vertices)

Figure 6.8: *Rockerarm* **model: shaded (left), wireframe (right).**

6.6 Potential applications of a curvature guided simplification

The field of medical visualisation can greatly benefit from strong simplification algorithms that preserve the features of the object. It is common in medical imaging that the capture systems can obtain large amounts of data for an object. Mesh simplification can permit a real time visualisation of the data. (See Figure 6.10 for an example).

A simplification algorithm using curvature tends to maintain the shape of an object, in particular of curved, smooth surfaces. Most organic material presents this kind of shape, and will be accurately simplified using the techniques presented here.

Another field where mesh simplification is extensively used is on video games. These require very fast visualisation of objects in motion, and in increasingly complex environments. To make processing faster, it is common to have varying *levels of detail* of the meshes, which are selected depending on the distance of the objects from the camera. Our simplification method can be used to generate several intermediate meshes with increasing decimation. Figure 6.11 shows a textured *triceratops* at three different levels of detail.

Figure 6.9: Detail of the decimated *rockerarm* model at 95 % decimation.

In (Hubeli and Gross 2000) the focus of the research is on fairing of collections of non-manifold meshes. They propose the use of non-manifolds as a more intuitive modelling tool for design. They also use mesh simplification to better determine the intersections of very complex meshes, and later perform fairing on them. We believe that in this case a simplification algorithm that preserves the shape of the separate elements will produce better results.

6.7 Conclusions

We have implemented a vertex decimation program that bases its selection of vertices for removal on the curvature and area of the vertex. The program progressively removes vertices from the mesh, choosing the next vertex with the smallest relevance weight. The changes made to the local geometry are optimised using again the curvature measures of the affected vertices.

Experimental results have shown how the use of the Total Absolute Curvature of a vertex provides significantly improved results in polygonal mesh simplification over the use of the more common Gaussian curvature computation. This is due to the fact that the TAC provides a more detailed description of the neighbourhood of a vertex and can distinguish features that Gaussian curvature would not. Both curvature measures are employed to assign relevance weights to vertices and to optimise the triangulation of affected areas using edge flips.

We have experimented with different parameters to determine the importance of a vertex in the mesh. A set of vertex weights based on the TAC were presented, generally referred to as WTAC. These involve various measures of the area of the neighbourhood of the vertex. From the results obtained, the best overall factor is the one known as the ATAC, where the TAC is multiplied by the surface area of the triangles incident on the vertex. Other weights are better at preserving the smaller details of the object, but alter

(a) Original (2,154 vertices)

(b) 85.4 % decimation with PGM (314 vertices)

Figure 6.10: *Skeletcilfoot* **model: shaded (left), wireframe (right).**

Figure 6.11: Textured *triceratops* **at three levels of detail. Mesh simplified, from front to back, at 0, 70.6 and 88.3 % decimation, respectively.**

significantly the shape of the object in the regions where vertices have smaller curvature.

6.8 Future research directions

At the moment the algorithm works progressively vertex by vertex. The next step is to move from vertices to regions, by grouping together regions of similar curvature and removing all vertices within that region if it has minimal WTAC. The Gauss Map allows us to identify such regions. This however would imply a graph theoretical problem to navigate around the mesh analysing the regions.

After the removal of a vertex, the current policy for the optimisation of the modified region focuses only on minimisation of the local curvature. In some cases this may actually alter the shape of the object, since important feature edges may be flipped for the sake of a smaller curvature. It is necessary to identify these edges and ensure they are not flipped during the optimisation phase. One way to address this problem is to also use as a parameter to select vertices for decimation the difference between the curvature of the region before and after the removal of the vertex. This requires extra computational load to compute the cost of removing every single vertex, but may produce better results.

Currently the proposed ATAC weight gives the best results, but at the loss of some detail in the mesh, while the other three weights tested are better at preserving such detail. An implementation that can keep the shape of the object on both large and small features would be desirable. This could be possible by selecting an appropriate weight measure for each individual vertex according to the properties of its neighbourhood.

An interesting side effect of the decimation algorithm is that the curvature domains of an object become better defined after simplification, even in the case of fairly noisy data or uneven surfaces. The reduction of complexity in the mesh translates in a reduction in the number of curvature domains, and the removal of mixed vertices with small positive or negative curvature components. According to our observations, it would be possible to use decimation as a preparation step before doing feature identification: the mesh is first simplified, then the features regions are recognised on the surface, and finally they are related back to the original mesh.

Chapter 7

Conclusions and further research

7.1 Conclusions

This thesis has presented a new method to compute the Total Absolute Curvature of polyhedral meshes from the Polyhedral Gauss Map. This new technique is analogous to the methods to compute curvature for smooth surfaces. As a main contribution, the approach introduced here can distinguish between separate signed components of the curvature for each vertex. This distinction permits a better characterisation of vertices into representative types, and their correct handling in diverse applications.

Chapter 2 presented the basic geometric concepts used in the computational implementations. It also described the discrete analogues of curvature for the two–dimensional case, and exposed the relationship between the curvature and the direction of the angles. From these concepts, the definition of Total Absolute Curvature is extracted. It is also demonstrated how the curvature of a planar polygon is dependent on the local curvature of the *deviations* from the convex hull.

Chapter 3 introduced the main concepts behind the discrete methods to measure the curvature of a surface in space. The concept of Total Absolute Curvature is presented as an extension of the Gaussian curvature, using the definitions of positive and negative components. The Polyhedral Gauss Map is introduced as a method to extract all components of positive and negative curvatures from the vertices of a polyhedral mesh. This method can characterise any kind of vertex in a discrete surface, including self–intersecting objects. The validity of the curvature computations using the Gauss Map is demonstrated for the various types of vertices.

Chapter 4 presented in detail the algorithm used to construct the Polyhedral Gauss Map of vertices. The methods and algorithms presented comprise the central contributions of this thesis. The methods developed are capable of correctly estimating the curvature of a polyhedral vertex of any kind, including non–convex and non–manifold vertices. The main innovations are:

- The methods to construct the spherical indicatrix of a vertex, by considering the connection of the faces around a vertex.
- The detection of arc intersections in the various cases that occur.
- The splitting of the spherical indicatrix into individual spherical polygons.
- The selection of the sign corresponding to each spherical polygon.
- The measure of the absolute curvature of a vertex as the sum of the areas of the spherical polygons in its PGM.

A comparison is made between the use of Total Absolute Curvature versus Gaussian Curvature in several practical applications. The PGM presents advantages over AD in better identifying features of the shape of an object. This can be used to more adequately deal with differently shaped polyhedra.

Chapter 5 describes the techniques developed to visualise the PGM of individual vertices and whole surfaces. The main development here is the techniques used to display the spherical polygons of complex Gauss Maps. Three–dimensional computer graphics are used to generate a practical and interactive tool to study the Polyhedral Gauss Map. This can be used to corroborate the numerical curvature measures and to improve the understanding of the properties of complex objects.

Finally, Chapter 6 used the example of polygonal mesh simplification to demonstrate the advantages of the Polyhedral Gauss Map method over the Angle Deficit on a practical application. A vertex decimation algorithm is implemented, which makes use of curvature measures to determine the vertices to remove from the mesh. New parameters to assign a relevance weight to vertices are introduced. They use combinations of curvature and area to give valid weights. Various sample meshes are simplified, and then numerically compared with the un–simplified versions. The results obtained show that the use of the PGM to select vertices for decimation can preserve better the details of an object without altering its shape.

We note that in Appendix A we will report two algorithms developed to fit curves over disorganised point clouds. These algorithms are guided by minimisation of curvature and other geometric parameters to generate adequate curves where no or limited information is available about the source of the data. The aim of these algorithms is the search and identification of the parameters that contribute the most to the curvature of a polygon, when used as energy functionals for curve reconstruction. These experiments need to be extended, to evaluate the usefulness of the algorithms presented therein.

7.2 Future research directions

The majority of the programs developed for this research have useful potential applications, and need to be extended. This is the case for curve reconstruction (Section 2.4.1), face recognition (Section 2.4.2), terrain description (Section 4.5.3) and so on.

Furthermore the computational implementations are not optimal, and would require extensive work to interface with other programs. We would suggest the use of optimised functions available in a computational geometry repository, such as the CGAL project (CGAL Editorial Board 2006), to improve the proposed methods.

Finally, the new measures of curvature using the Polyhedral Gauss Map can be used to extend research on optimisation of a polygonal mesh based on curvatures. Measures of Gaussian curvature have been used in (Alboul 2003) and (Netchaev 2004) to optimise a mesh using edge flips. Using this method, it is common to find vertices with self intersecting faces through the middle stages of the optimisation. The better computation of TAC should improve the obtained results, even in the case of complex vertices.

Bibliography

Aichholzer, O., L. S. Alboul, and F. Hurtado (2002). On flips in polyhedral surfaces. *International Journal of Foundations of Computer Science 13*(2), 303–311.

Alboul, L. (2003). Optimising triangulated polyhedral surfaces with self-intersections. In M. J. Wilson and R. R. Martin (Eds.), *IMA Conference on the Mathematics of Surfaces*, Volume 2768 of *Lecture Notes in Computer Science*, pp. 48–72. Springer.

Alboul, L. and G. Echeverria (2005, September 5–7). Polyhedral Gauss maps and curvature characterisation of triangle meshes. In R. R. Martin, H. E. Bez, and M. A. Sabin (Eds.), *Proc. of Mathematics of Surfaces XI, 11th IMA International Conference, Loughborough, UK*, Volume 3604 of *Lecture Notes in Computer Science*, pp. 14–33. Springer.

Alboul, L., G. Echeverria, and M. Rodrigues (2005, March 9–11). Discrete curvatures and Gauss map for polyhedral surfaces. In *Proc. of 21st European Workshop in Computational Geometry, Eindhoven, Netherlands*, pp. 69–72.

Alboul, L., G. Echeverria, and M. A. Rodrigues (2004a, March 24–26). Curvature criteria to fit curves to discrete data. In *Proc. 20th European Workshop in Computational Geometry, Sevilla, Spain.*

Alboul, L., G. Echeverria, and M. A. Rodrigues (2004b, June/July). Total absolute curvature as a tool for modelling curves and surfaces. In *Proc. of Geomgrid, Moscow, Russia*, pp. 220–231.

Alboul, L., G. Kloosterman, C. Traas, and R. Van Damme (1999, July 20). Best data-dependent triangulations.

Alboul, L. and R. van Damme (1994). Polyhedral metrics in surface reconstruction. In G. Mullineux (Ed.), *IMA Conference on the Mathematics of Surfaces*, pp. 171–200. Clarendon Press.

Aleksandrov, A. D. (2005). *Convex Polyhedra*. Springer.

Aleksandrov, A. D. and Y. Reshetnyak (1989). *General Theory of Irregular Curves.*

Kluwer Academic Publishers.

Aleksandrov, A. D. and V. A. Zahlgaller (1967). *Intrinsic Geometry of surfaces*. Providence, Rhode Island: AMS.

Aloupis, G., P. Bose, and P. Morin (2004). Reconfiguring triangulations with edge flips and point moves. In J. Pach (Ed.), *Graph Drawing, 12th International Symposium, GD 2004, New York, NY, USA, September 29 - October 2, 2004, Revised Selected Papers*, Volume 3383 of *Lecture Notes in Computer Science*, pp. 1–11. Springer.

Althaus, E. and K. Mehlhorn (2000, January 9–11). TSP-based curve reconstruction in polynomial time. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, N.Y., pp. 686–695. ACM Press.

Amenta, N., M. W. Bern, and D. Eppstein (1998). The crust and the beta-skeleton: Combinatorial curve reconstruction. *Graphical Models and Image Processing 60*(2), 125–135.

Bærentzen, J. A. (2006, June 29). Optimizing 3D triangulations to recapture sharp edges. Technical report.

Baker, M. J. (2007). Maths – rotation matrices. In `http://www.euclideanspace.com/maths/algebra/matrix/orthogonal/rotation/%`.

Banchoff, T. F. (1967). Critical points and curvature for embedded polyhedra. *Journal of Differential Geometry 1*, 245–256.

Banchoff, T. F. (1970). Critical points and curvature for embedded polyhedral surfaces. *American Mathematical Monthly 7*, 475–485.

Banchoff, T. F., T. Gaffney, and C. McCrory (1982). Cusps of Gauss mappings. *Research Notes in Mathematics 55.*

Banchoff, T. F. and W. Kühnel (1998, June 24). Tight and taut submanifolds.

Bloch, E. D. (1998). The angle deficit for arbitrary polyhedra. *Contributions to Algebra and Geometry 39*(2), 379–393.

Bobenko, A. I. (2005). A conformal energy for simplicial surfaces. *Combinatorial and computational geometry 52*, 133–143.

Borowski, E. J. and J. M. Borwein (2002). *Mathematics Dictionary*. Glasgow, UK: Harper–Collins.

Borrelli, V., F. Cazals, and J.-M. Morvan (2003). On the angular defect of triangulations and the pointwise approximation of curvatures. *Computer Aided Geometric Design 20*(6), 319–341.

Brehm, U. and W. Kühnel (1982). Smooth approximation of polyhedral surfaces regarding curvatures. *Geom. Dedicata 12*, 435–461.

Brink, W. and L. Alboul (2006, July 4–7). Willmore energy as a tool for mesh optimisation. In *Numerical Geometry, Grid Generation and High Performance Computing*, pp. 6–12.

Burago, Y. D. (1970). *Isoperimetric inequalities in the theory of surfaces of bounded external curvature*. Consultants Bureau.

Calladine, C. (1986). Gaussian curvature and shell structures. In J. A. Gregory (Ed.), *The Mathematics of Surfaces*, pp. 179–196. Clarendon Press.

CGAL Editorial Board (2006). Cgal-3.2 user and reference manual. In `http://www.cgal.org/Manual/3.2/doc_html/cgal_manual/index.html`.

Chen, L., S. Chou, and T. C. Woo (1993, October). Separating and intersecting spherical polygons: Computing machinability on three-, four-, and five-axis numerically controlled machines. *ACM Transactions on Graphics 12*(4), 305–326.

Choi, B. K., H. Y. Shin, Y. I. Yoon, and J. W. Lee (1988). Triangulations of scattered data in $3d$ space. *Comp. Aided Design 20*(5), 239–248.

Cignoni, P., C. Montani, and R. Scopigno (1998, February). A comparison of mesh simplification algorithms. *Computers & Graphics 22*(1), 37–54. ISSN 0097-8493.

Cignoni, P., C. Rocchini, and R. Scopigno (1998). Metro: Measuring error on simplified surfaces. *Computer Graphics Forum 17*(2), 167–174. ISSN 1067-7055.

Cohen, J., M. Olano, and D. Manocha (1998, August). Appearance-preserving simplification. *Computer Graphics 32*(Annual Conference Series), 115–122.

Cohen-Steiner, D. and J.-M. Morvan (2003, June 8–10). Restricted Delaunay triangulations and normal cycle. In *Proceedings of the nineteenth Conference on Computational Geometry (SCG-03)*, New York, pp. 312–321. ACM Press.

Crépeau, C. (2004, April). Geometric algorithms. In `http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/30.Geometry.pdf`.

de Berg, M., M. van Kreveld, M. Overmars, and O. Schwarzkopf (1997). *Computational Geometry Algorithms and Applications*. Berlin Heidelberg: Springer-Verlag.

Dey, T. K. and R. Wenger (2000). Reconstruction curves with sharp corners. In *Symposium on Computational Geometry*, pp. 233–241.

do Carmo, M. P. (1976). *Differential Geometry of Curves and Surfaces*. New Jersey: Prentice–Hall Inc.

Dyn, N., I. Goren, and A. Rippa (1993). Transforming triangulations in polygonal domains. *Computer Aided Geometric Design 10*, 531–536.

Dyn, N., K. Hormann, S.-J. Kim, and D. Levin (2001). Optimizing 3D triangulations using discrete curvature analysis. In T. Lyche and L. Schumaker (Eds.), *Mathematical Methods for Curves and Surfaces: Oslo 2000*, Innovations in Applied Mathematics, pp. 135–146. Nashville, TN: Vanderbilt University Press.

Eastlick, M. T. (2006, July). *Discrete Differential Geometry and an Application in Multiresolution Analysis*. Ph. D. thesis, University of Sheffield.

Echeverria, G. and L. Alboul (2006, October 20–21). Decimation and smoothing of triangular meshes based on curvature from the polyhedral Gauss map. In *Proc. Of CompIMAGE Conference. Coimbra, Portugal*, pp. 175–180.

Echeverria, G., L. Alboul, and M. Rodrigues (2005, November 24–25). Enhancing game physics using Gauss map computation. In *Proc. Of Game-On Conference. Leicester, UK*, pp. 47–51.

Elgindy, H., H. Everett, and G. Toussaint (1989, November 01). Slicing an ear in linear time.

Falcidieno, B. and M. Spagnuolo (1991). A new method for the characterization of topographic surfaces. *International Journal of Geographical Information Systems 5*(4), 397–412.

Fournier, A. and D. Y. Montuno (1984). Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics 3*(2), 153–174.

Friedel, I., P. Mullen, and P. Schröder (2003). Data-dependent fairing of subdivision surfaces. In *Symposium on Solid Modeling and Applications*, pp. 185–195. ACM.

Garland, M. and P. S. Heckbert (1997, August). Surface simplification using quadric error metrics. *Computer Graphics 31*(Annual Conference Series), 209–216.

Geomview (1996). Geomview interactive 3d viewing program for unix. In `http://www.geomview.org/`.

George, P.-L. and H. Borouchaki (2003). Back to edge flips in 3 dimensions. In *IMR*, pp. 393–402.

Giesen, J. (2000). *Curve Reconstruction*. Ph. D. thesis, Swiss Federal Institute of Technology, ETH Zürich.

Grinspun, E. and P. Schröder (2001). Normal bounds for subdivison-surface interference detection. In T. Ertl, K. Joy, and A. Varshney (Eds.), *Proceedings Visualization 2001*, pp. 333–340. IEEE Computer Society Technical Committee on Visualization and Graphics Executive Committee.

Grodon, G. G. (1991). Face recognition based on depth maps and surface curvature. In *SPIE Geometric Methods in Computer Vision*, Volume 1570, pp. 234–247.

Horn, B. K. P. (1983, July). Extended Gaussian images. Technical Report AIM-740, MIT Artificial Intelligence Laboratory.

Hubeli, A. and M. Gross (2000, October 8–13). Fairing of non-manifolds for visualization. In T. Ertl, B. Hamann, and A. Varshney (Eds.), *Proceedings of the Conference on Visualization 2000 (VIS-00)*, Piscataway, NJ, pp. 407–414. IEEE Computer Society.

Hussain, M., Y. Okada, and K. Niijima (2004, February). Efficient and feature-preserving triangular mesh decimation. In V. Skala (Ed.), *Journal of WSCG*, Volume 12, Plzen, Czech Republic. UNION Agency - Science Press.

Kilian, M. (2004). Differentialgeometrische konzepte für dreiecksnetze. Master's thesis, University of Karlsruche.

Kim, S.-J., C.-H. Kim, and D. Levin (2002, October). Surface simplification using a discrete curvature norm. *Computers and Graphics 26*(5), 657–663.

Kindratenko, V. (1997). *Development of Applications of Image Analysis. Techniques for Identification and Classification of Miscroscopic Particles*. Ph. D. thesis, University of Antwerp.

Kirkpatrick, D. G. and J. D. Radke (1985, June 5–7). A framework for computational morphology. In *Proceedings of the Symposium on Computational Geometry (Baltimore)*, pp. 217–248. ACM: ACM Press.

Kühnel, W. (2002). *Differential geometry. Curves–Surfaces–Manifolds*. Amer Math Society.

Kuiper, N. H. (1970). Minimal total absolute curvature for immersions. *Inventiones Mathematicae 10*, 209–238.

Lawson, C. (1972). Transforming triangulations. *Discrete mathematics 3*, 365–372.

Lee, A. W. F., W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin (1998, July 19–24). MAPS: Multiresolution adaptive parameterization of surfaces. In *Proceedings of the ACM Conference on Computer Graphics (SIGGRAPH-98)*, New York, pp. 95–104. ACM Press.

Lee, H., L. Kim, M. Meyer, and M. Desbrun (2001, September 2–3). Meshes on fire. In N. Magnenat-Thalmann and D. Thalmann (Eds.), *Computer Animation and Simulation '01*, Eurographics, pp. 75–84. Springer-Verlag Wien New York. Proceedings of the Eurographics Workshop, Manchester, UK.

Li, Y. and P. Gu (2004). Free-form surface inspection techniques state of the art review. *Computer-Aided Design 36*(13), 1395–1417.

Lindstrom, P. and G. Turk (1998). Fast and memory efficient polygonal simplification. In *IEEE Visualization*, pp. 279–286.

Little, J. J. (1985, June). Extended Gaussian images, mixed volumes, shape reconstruction. In J. O'Rourke (Ed.), *Proceedings of the Symposium on Computational Geometry*, Baltimore, MD, pp. 15–23. ACM Press.

Lowekamp, B., P. Rheingans, and T. S. Yoo (2002, October). Exploring surface characteristics with interactive Gaussian images (A case study). In R. J. Moorhead, M. Gross, and K. I. Joy (Eds.), *Proceedings of IEEE Visualization 2002*, pp. 553–556. IEEE Computer Society: IEEE Computer Society Press.

Luebke, D. P. (2001, May/June). A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications 21*(3), 24–35.

Lyusternik, L. A. (1963). *Convex Figures and Polyhedra.* New York: Dover Publications.

Maltret, J. L. and M. Daniel (2002, March). Discrete curvatures and applications: a survey. Technical report.

Meyer, M., M. Desbrun, P. Schröder, and A. H. Barr (2002, October 04). Discrete differential-geometry operators for triangulated 2-manifolds.

Morris, D. D. and T. Kanade (2000). Image-consistent surface triangualtion. In *19th Conf. Computer Vision and Pattern Recognition, IEEE*, Volume 1, pp. 332–338.

Murray, J. D. and W. VanRyper (1996, April). *Encyclopedia of Graphics File Formats.* O'Reilly Media, Incorporated.

Netchaev, A. I. (2004, September). *Triangulations and their Application in Surface Reconstruction.* Ph. D. thesis, University of Twente.

Osserman, R. (1978). The isoperimetric inequality. *Bull. Amer. Math. Soc. 84*, 1182–1238.

Peng, J., Q. Li, C. C. Ja Kuo, and M. Zhou (2003). Estimating Gaussian curvatures from 3d meshes. In B. E. Rogowitz and T. N. Pappas (Eds.), *Human Vision and Electronic Imaging VIII, Proc. of SPIE 5008*, pp. 270–280.

Polya, G. (1954). An elementary analogue to the Gauss-Bonnet theorem. *American Mathematical Monthly 61*, 601–603.

Robinson, A. (2005, January). *Surface Scanning with Uncoded Structured Light Sources*. Ph. D. thesis, Sheffield Hallam University.

Robinson, A., L. Alboul, and M. Rodrigues (2004). Methods for indexing stripes in uncoded structured light scanning systems. *Journal of WSCG 12*(1–3), 371–378.

Rodríguez, L. and H. Rosenberg (2000). Rigidity of certain polyhedra in $\mathbb{R}^3$. *Comentarii Mathematici Helvetici 75*(3), 478–503.

Sabin, M. A. and N. A. Dogson (2005). A circle–preserving variant of the four–point subdivision scheme. *Mathematical Methods for Curves and Surfaces:*, 1–12.

Schreiner, D., D. E. Schreiner, and D. Shreiner (1999). *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Schroeder, W. J., J. A. Zarge, and W. E. Lorensen (1992, July). Decimation of triangle meshes. In E. E. Catmull (Ed.), *Computer Graphics (SIGGRAPH '92 Proceedings)*, Volume 26, pp. 65–70.

Sullivan, J. M. (2006, June 01). Curves of finite total curvature. Comment: 25 pages, 4 figures; notes from a lecture at the Oberwolfach Seminar "Discrete Differential Geometry" (June 2004).

Tanaka, H. T., M. Ikeda, and H. Chiaki (1998). Curvature-based face surface recognition using spherical correlation-principal directions for curved object recognition. In *International Conference on Automatic Face and Gesture Recognition*, pp. 372–377. IEEE Computer Society.

Thodberg, H. H. and H. Ólafsdóttir (2003, September). Adding curvature to minimum description length shape models. In *British Machine Vision Conference, BMVC.*

Turk, G. (1992). Re-tiling polygon surfaces. *Proceedings of SIGGRAPH'92*, 55–64.

Ujiie, Y. and Y. Matsuoka (2003). Total absolute curvature to represent the complexity of diverse curved profiles. In *6th Asian Design International Conference, Tsukuba.*

Van der Sterren, W. (2001). Terrain reasoning for 3d action games. In *GDC Proceedings*.

van Rooij, A. C. M. (1965). The total curvature of curves. *Duke Mathematical Journal 32*(2), 313–324.

Watt, A. H. and A. Watt (1999). *3d Computer Graphics with Cdrom*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Weisstein, E. W. (1999a). Hypotrochoid. from *MathWorld*—-a wolfram web resource. In `http://mathworld.wolfram.com/Hypotrochoid.html`.

Weisstein, E. W. (1999b). Spherical polygon. from *MathWorld*—-a wolfram web resource. In `http://mathworld.wolfram.com/SphericalPolygon.html`.

Woo, M., Davis, and M. B. Sheridan (1999). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Wu, J. and L. Kobbelt (2002, November 20–22). Fast mesh decimation by multiple-choice techniques. In G. Greiner (Ed.), *Proceedings of the Vision, Modeling, and Visualization Conference (VMV 2002), Erlangen, Germany*, pp. 241–248. Aka GmbH.

Yamauchi, H., S. Gumhold, R. Zayer, and H.-P. Seidel (2005). Mesh segmentation driven by Gaussian curvature. *The Visual Computer 21*(8-10), 659–668.

Appendix A

Curve–fitting over datasets on a plane

A.1 Introduction

This chapter deviates from the main research topic of the thesis, but is based on the curvature principles studied in it. It presents the experiments performed to analyse the relationship between the curvature and the shape of a polygon in $\mathbb{R}^2$. The concepts of Total Absolute Curvature presented in Chapter 2 are applied to solve the problem of curve reconstruction from a unorganised point cloud: Given a set of vertex coordinates as input, the objective is to generate a simple closed curve that joins all points. By simple closed curve we mean it must be possible to go from any point in the data to another by traversing the curve, and there should be no self–intersections.

The problem of reconstructing curves arises in various domains such as Computer Vision, Computer Graphics, Reverse Engineering, Image Processing, Mathematics and Chemistry, among others. The vertex information can originate from various sources, such as image processing, medical scanning, physical data or others. In many applications the data are presumed to be taken from a known, often smooth, curve and the task is to reconstruct this curve from the given data. In general, the first step of the curve reconstruction process, is to build up a polygonal curve that spans the given data points, in such a way that it approximates the original curve in the best possible way. The accuracy of this approximation can be measured, for example, by the least square method. The idea behind the approximation is that if further processing is made, for example by applying a subdivision, the original curve may be completely recovered (Sabin and Dogson 2005).

Our particular focus is on the discrete case of curves represented as polygons, composed of vertices and edges. In this chapter, two new algorithms are introduced for the generation of polygonal curves. Both are based on the convex hull of the dataset, but differ in the number of *deviations* permitted. The implementation presented here reconstructs curves by minimisation of various combinations of curvature and distances as energy functionals. The curves reconstructed are evaluated by means of the Total Absolute Curvature, presented in the Chapter 2. The objective is to determine which algorithms and parameters are better suited to reconstruct particular curves, for the cases when some information is known about the source of the data.

The idea behind minimisation of a certain energy is to obtain a fair curve, *i.e.* a correct representation of the underlying real continuous curve. However, the 'correct' curve is often not defined. Another problem is that energy functionals can have multiple local minima. In general, any local minimum of energy is considered as a solution to the optimisation problem, mostly because the global minimum is hard to reach. However, the global minimum might also be not unique, moreover, it might be far away from the input configuration (Friedel *et al.* 2003). A local minimum, which is as close as possible to the initial configuration, might be more meaningful.

Three algorithms are used to produce curves with different parameters and restrictions:

- Simple closed curves: Created with a basic algorithm using the principle of *orientation*. This method permits the creation of several different curves from the same data. It does not incorporate any optimisation, and may produce polygons of large curvature.
- Curves with only one *deviation* from the convex hull: This is done by recursively creating nested convex hulls and later joining them together in a single curve, using different parameters to identify where to join the hulls. The aim of this approach is to minimise the number of *deviation vertices* and find out how this affects TAC. This algorithm represents a new contribution on curve–fitting algorithms.
- Curves with several deviations from the convex hull: The CH–curve is obtained first, using the method described in Chapter 2, and the remaining vertices are added to the curve one by one by minimising an *insertion cost*. The cost is assigned to vertices depending on where they will be inserted. The insertion cost is obtained from various combinations of geometrical parameters of the curve. This method of adding vertices to the convex hull is also considered as a new contribution.

All three algorithms presented are susceptible to certain kinds of self–intersections during the reconstruction process. This chapter describes the methods developed to detect such intersections and correct the curve. These methods vary according to the algorithm used.

The research in this chapter has been partially presented in (Alboul *et al.* 2004a) and (Alboul *et al.* 2004b).

A.2 Previous research on curve reconstruction

An important characteristic used to classify curve reconstruction algorithms is the sampling rate required to produce an adequate result. Some algorithms take data sampled at regular intervals from the original source. Others require only *non–uniform sampling*, that is, few points have to be taken from straight parts of the original curve, and more points are needed from sharp corners.

Many algorithms are based on the computations of the Voronoi diagram and the Delaunay triangulation. For example the algorithms of *Crust* and *β–Skeleton*, described in (Amenta *et al.* 1998) as:

- Crust algorithm: Given a set S of points in the plane and let S' be the union of S with the vertices of its Voronoi diagram. An edge in the Delaunay triangulation of S' belongs to the crust of S if both of its endpoints belong to S.
- β–Skeleton algorithm: Let $\beta \geq 1$ be a constant. Let p and q be a pair of points in the plane at distance $|p - q|$. The edge $\overline{pq}$ is in the β-skeleton if the union of the two balls of diameter $\beta|p - q|$ tangent to p and q is empty. It was first described in (Kirkpatrick and Radke 1985), and here an adequate value for β was chosen to obtain correct reconstructions.

The authors also define the concepts of *medial axis* and *feature size*. The medial axis of a curve is the set of points on the plane which have more than one closest point in the curve. The feature size is defined for each point in the curve as the distance to the closest point in the medial axis. These algorithms depend on a certain sampling rate of the data to produce a good reconstruction. The sampling rate is chosen as a value dependent on the feature size.

In his PhD thesis, (Giesen 2000) does a review of the existing algorithms for the reconstruction of a curve in 2D and presents a new approach that is a mixture of other methods. He gives mathematical proofs of the most important geometric properties of curves: the measurements of length and curvature. He classifies curve reconstruction algorithms into filter or optimisation techniques. *Filter techniques* are those which construct a curve out of a subset of the Delaunay graph, such as the *Crust* and *β–Skeleton* algorithms. Giesen proves that these algorithms are not reconstruction schemes for polygons, that is, they will not correctly reconstruct curves which have sharp corners at some vertices. This is because at this points the medial axis crosses the curve, requiring near infinite sampling near corner vertices. *Optimisation techniques* include *Minimum Spanning Tree* and *Travelling Salesman Path*, and these are proven to be reconstruction schemes for curves, but not for polygons. He proposes a new filtering technique called λ*–Filter* to be used together with an optimisation technique to drive a more robust curve reconstruction algorithm, where the filtering stage would identify independent clusters of points and then the optimisation algorithm will do the reconstruction for each of those clusters.

(Kindratenko 1997) studies image analysis for the purpose of identification of microscopic particles. The Part 2: Shape analysis, does an extensive review of the methods used to obtain further information from the boundary of an object. The contour of the object is treated as a 2D continuous curve. The author describes several different approaches to shape analysis. The *functional approach* consists of identifying a function that represents the shape of the object, with the advantage that the correct function can simplify the data representation of the object, and be used to obtain further information about the object. The *set theory approach* is based on common geometric properties of an object, combined together for further insight on the properties of a curve. The basic parameters used are area, perimeter, the diameters of circles with the same area or perimeter as the curve, orthogonal projections of the shape on the X, Y or other significant directions, diameters of the figure, the convex hull, radial distances from an arbitrary centre of the figure, lengths of the semi–axes the ellipse with the same area and perimeter as the original curve, and several other measures. All these are also combined to obtain aspect ratios with different applications.

(Dey and Wenger 2000) propose a method that is capable of reconstructing curves with very sharp corners. The sampling rate they define is dependent on corner vertices. Additionally a normal for each vertex is computed, and vertices are connected with the two neighbours at either side of their normal. Some heuristic restrictions make this approach robust even at sharp corners.

The Travelling Salesman Path (TSP) is another common algorithm for curve reconstruction, where all vertices are inserted by selecting the next closest point to the partial curve under construction. Normally TSP algorithms have two main drawbacks:

- They treat the whole dataset as a single object and do not identify separate components.
- They are very sensitive to noise in the data, since they blindly incorporate all points into the reconstructed curve.

The paper (Althaus and Mehlhorn 2000) shows that under certain sampling conditions the Travelling Salesman Path can be obtained in polynomial time. The methods proposed in the following sections are all designed to generate a simple closed curve that includes all vertices in the dataset. They can be seen as a specific class of TSP algorithms, with the main difference being that our methods begin the reconstruction from the convex hull of the dataset.

A.3 Details of the implementation

For the purpose of testing the curve–fitting algorithms, a computer program was developed, called `reconstructor`. The program will receive a set of vertex coordinates, and produce as output a curve that fits over the dataset received. The program allows the user to select among various algorithms and minimisation parameters to generate different curves. The resulting polygons are evaluated in terms of their curvature, perimeter and area. This section describes the data structures, input and output files used in the `reconstructor` program.

A.3.1 Data input

The program takes as input a text file with the *NODE* format, which specifies the number of vertices, and the coordinates of each of them. It does not provide other information on the structure of the data. The format of the file is:

- The first line contains an integer indicating the number of vertices in the sample, and optionally another integer representing the number of coordinates of each vertex.
- Each line after that contains the floating point numbers, representing the X, Y and Z coordinates of the vertex. Since the program currently only works with vertices on the plane, the values for Z can be omitted, and will be assumed to be equal to 0.0

Example:

```
10 2
-4.5 -2.0 0.0
-1.5 -2.0 0.0
-1.5  1.0 0.0
 1.5  1.0 0.0
 1.5 -2.0 0.0
-4.5  2.0 0.0
-1.5  2.0 0.0
 1.5  2.0 0.0
```

```
4.5  2.0 0.0
4.5 -2.0 0.0
```

After reading the input file the vertices are stored in a list, ordered according to their X coordinate in ascending order; that is, vertices are sorted from left to right. If more than one vertex has the same value for their X coordinate, then they are ordered by increasing value of Y. This ordering step facilitates the construction of curves, guaranteeing that the first vertex in the list will be located in one of the corners and will belong to the convex hull (de Berg *et al.* 1997). Also as the list is traversed the vertices will go from left to right on the plane, simplifying the process of finding the next vertex to add to a curve when using certain algorithms.

It is always assumed that no two vertices will have the same coordinates. In its current state, the program does no check for duplicated vertices. In such a case the program will treat the vertices separately, which will lead to the curve having self–intersections.

A.3.2 Program output

The program generates various curves, according to the algorithm chosen. All curves are represented internally as an ordered list of vertices, with Counter–Clockwise orientation. The `reconstructor` program can display the resulting curves using OpenGL, showing the vertices and the segments that join them. It will also show the measurements obtained for each curve.

Additionally, the best curve generated will be saved as an *OFF* file that can be read using the *Geomview* program (Geomview 1996). This file contains the coordinates of the vertices, plus the edges that make up the curve represented as pairs of vertices. A sample output file looks like this:

```
OFF
8 8 0
4.000000        0.000000        0.000000
0.000000        4.000000        0.000000
-4.000000       0.000000        0.000000
-0.000000       -4.000000       0.000000
1.131371        1.131371        0.000000
-1.131371       1.131371        0.000000
-1.131371       -1.131371       0.000000
1.131371        -1.131371       0.000000
2 2 6
2 6 3
2 3 7
2 7 0
2 0 4
```

```
2 4 1
2 1 5
2 5 2
```

The format of these files is the following:

- The first line is a header, indicating the file type as an "OFF" file.
- The second line contains the number of vertices, faces, and edges in the object.
- The first data section contains the X, Y and Z coordinates of the vertex.
- The second section has the faces, indicating first the number of vertices of each face, followed by the indices of the corresponding vertices.

The OFF format is generally used for 3D geometry, specifying the faces as triangles or quadrilaterals delimited by three or four vertices. In our application, however, we delimit the faces by two vertices and *Geomview* will display the faces as line segments and appropriately show the result as a discrete curve.

The following sections present the several curve–fitting algorithms. We use the data samples shown in Figure A.1 to demonstrate the curves produced by each algorithm. The TAC of all the generated curves are shown for reference. TAC is measured as an angle in radians, and shown as a decimal number. The datasets consist of the following:

(a) Five independent circles. Figure A.1(a)

(b) Two non–smooth curves. A semicircle on top and a curve with a sharp angle in the shape of the letter "v". Figure A.1(b)

(c) Points sampled from an hypotrochoid (Weisstein 1999a). The formulas used to create the data are:

$$x = (r_2 - r_1)\cos\theta + c\cos\left(\frac{r_2}{r_1 - 1}\theta\right) \tag{A.3.1}$$

and

$$y = (r_2 - r_1)\cos\theta - c\cos\left(\frac{r_2}{r_1 - 1}\theta\right), \tag{A.3.2}$$

with $r_1 = 0.5$, $r_2 = 1.5$ and $c = 0.7$. Figure A.1(c)

(d) The vertices from two triangles and one hexagon nested within each other. Figure A.1(d)

A.4 Simple closed curves by orientation

The first curve–fitting algorithm is used as a reference point to compare the new algorithms. The vertices in the dataset are added to the curve in a specific order, determined

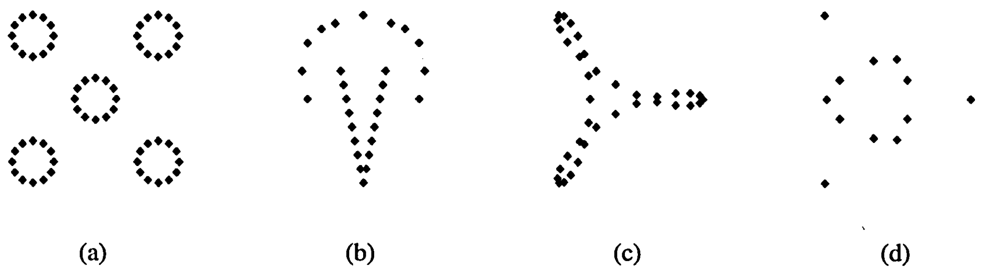


Figure A.1: The four datasets used for the curve reconstruction algorithms.

by sweeping a line over the point cloud in Counter–Clockwise direction, and inserting the vertices as they are covered by the line. The implementation of this algorithm is very similar to that of the convex hull, explained in Section 2.2. However in this case the reference point ν_r remains the same during the construction of the whole curve.

Given a dataset $\mathbf{V}$ consisting of n vertices, the algorithm selects a vertex $\nu_r \in \mathbf{V}$ as the *reference point*, and then finds a line that passes through ν_r and has all of the remaining points in the dataset on one side. This line is chosen as the *reference axis*, and represented as $\mathbf{A_r}$. The order in which new vertices are inserted into the curve is determined by the angle formed between the reference axis and the line segment $\overline{\nu_r\nu_i}$ joining each vertex ν_i with the reference point ν_r. The point with the smallest angle is the next one added to the curve (see Figure A.2). This is equivalent to rotating the reference axis 360 degrees around the reference point in a Counter–Clockwise direction and adding vertices to the curve in the order in which they are found as the line sweeps the plane.

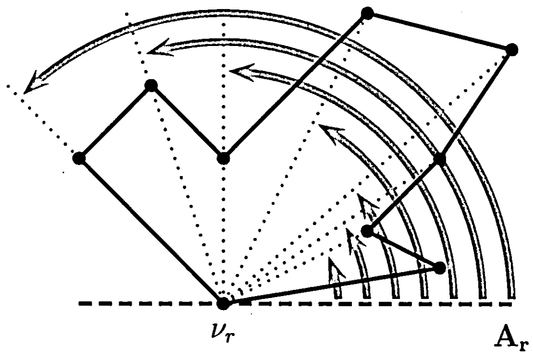


Figure A.2: Generation of a simple closed curve based on orientation.

Instead of computing the angles for each vertex, the concept of *orientation* is used to find the next point to add. At each moment considering only the points not yet allocated into the curve, the vertex with the smallest angle will be the one whose orientation with respect to all other vertices is Counter–Clockwise. A line joining this point and the reference point will always have all the remaining vertices on one side.

Since the reference point remains the same during the whole process, given a dataset $\mathbf{V}$ consisting of n vertices, the algorithm can generate n different curves $\mathbf{P_r}(\mathbf{V})$, by selecting each individual vertex as the starting reference point ν_r. Each of these curves is stored in an array. After all the n curves have been generated, the program will compare the curvatures and present the polygon with the smallest curvature. All the other curves can also be displayed for reference.

The program makes use of two lists of vertices: λ_p contains all the *pending vertices* not added yet to the curve, ordered by increasing values of the X coordinate, and λ_c has the ordered list of vertices that already belong to the *new curve* $\mathbf{P_r}(\mathbf{V})$. Initially λ_p contains all vertices in $\mathbf{V}$, and λ_c is empty. As the algorithm progresses, vertices are removed from λ_p and inserted into λ_c in the corresponding order. Algorithm A.1 shows the pseudo–code of the method to select the order of insertion of the vertices.

Algorithm A.1 Curve–fitting based on orientation.

```
For each vertex $\nu_r \in \mathbf{V}$
{
    Create list $\lambda_p$ with all vertices in $\mathbf{V}$.
    *failed* $\leftarrow$ FALSE.
    While $\lambda_p$ not empty
    {
        Get next *evaluation vertex* $\nu_e \in \lambda_p$.
        For each vertex $\nu_i \in \lambda_p$
        {
            Get orientation $\rho(\nu_e)$ from the vertices $(\nu_r, \nu_e, \nu_i)$.
            If $\rho = CW$, then
                *failed* $\leftarrow$ TRUE.
            If $\rho = COL$ and $d(\nu_r, \nu_i) < d(\nu_r, \nu_e)$, then
                *failed* $\leftarrow$ TRUE.
        }
        If not *failed*, then
        {
                Insert $\nu_e$ in $\lambda_c$.
                Remove $\nu_e$ from $\lambda_p$.
        }
    }
}
```

For the computational implementation two techniques are used to simplify the algorithm and reduce the number of comparisons made:

1. When evaluating a point ν_e to determine whether it should be added to the curve next, all the points that have already been discarded as the new insertion vertex are no longer considered when testing other vertices ν_i. This permits the algorithm to obtain curves with no self–intersections even for points that are inside the convex hull of the dataset, since it will ignore some of the points, facilitating the identification of a reference axis to begin the creation of the curve.
2. Since this is a greedy algorithm, the first point that satisfies the condition of having CCW orientation with respect to all other vertices in λ_p will be inserted into the curve, and the rest of the points are not tested, even if they would be a better option. This last condition, coupled with the previous assumption, can produce special cases in which self–intersections may occur, when ν_r does not belong to the convex hull. These cases are dealt with in a special way.

This algorithm is simple to implement and also very fast even for large datasets. It will produce a valid closed curve for each of the points in the sample. However, it includes no restrictions for the shape of the curve produced. In many cases the generated curve will be very irregular, especially in the vicinity of the vertices most distant from ν_r. Because of this, the curves constructed are generally not optimal, and their curvature may be very large.

There are several cases in which self–intersections can be produced when more than one vertex in the dataset are collinear with the reference point, and also when the reference point does not belong to the convex hull of the sample. For some of these vertices it will be impossible to find a reference axis that contains all of the remaining vertices on one side. This is solved by ignoring some of the other vertices before establishing a reference axis.

A.4.1 Self–intersections

There are four different cases for self–intersections caused by collinearity, and each is handled independently:

- Groups of vertices collinear with ν_r are situated in the middle of the dataset: These create an ambiguity in the basic algorithm, because all the collinear points will have the same angle with respect to the reference axis. In these cases the point that is closest to ν_r is inserted first. This case is exemplified in Figure A.3
- All the vertices at the *end of the dataset* are collinear: This happens when more than one vertex are left in λ_p and all of them are collinear with ν_r, as shown in

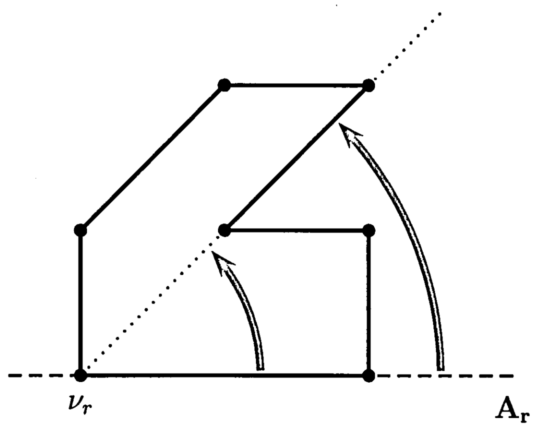


Figure A.3: Special case of vertices collinear with ν_r.

Figure A.4. In these cases the order of the vertices in λ_p must be reversed, thus giving preference to the points that are farther away from ν_r. When these cases are identified, and after the algorithm has proposed a point ν_e to be added next, another function validates whether ν_e is the one farthest from ν_r, and if not then the point with the largest distance to ν_r is returned instead.

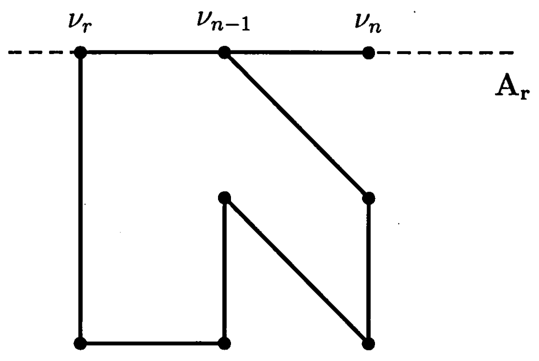


Figure A.4: Self–intersection caused by collinear points at the *end* of the dataset.

- Collinear vertices parallel to the X axis are at the rightmost of the dataset: Since the vertices are ordered by increasing X value, all of the points with the same X coordinate will be grouped together in λ_p, and the last of these points will also have the highest value for Y. When there are more than two collinear vertices at the end of the dataset and ν_r is neither the first nor the last of these points, there will be a self–intersection. The normal algorithm will discard all of the vertices before the final group of collinear points, and then tests these vertices against themselves. The result is that these points will be added to the curve before any other, and since the reference point is in the middle, the new segments will eventually overlap, as shown in Figure A.5.

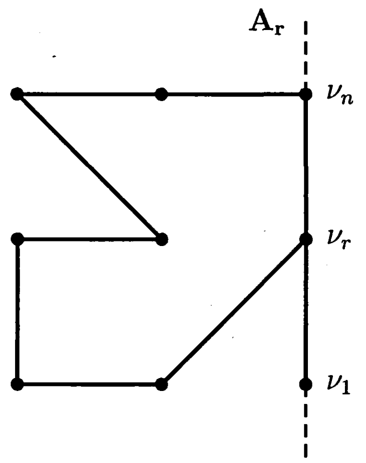


Figure A.5: Self–intersection caused by collinear points at the *right* of the dataset.

To solve this problem, the X coordinate of the reference point is compared to that of the last vertex. If they are the same, then all the points also having the same X value are moved from the end of the list to the front, except for the last point. This will cause the algorithm to test those vertices against more points, and discard them early on, so they will not be inserted into the curve until the end. The drawback of this technique is that these points will be tested each time, only to be rejected again.

- Reference point does not belong to the convex hull: This special case was mentioned earlier. The cause of this problem is that the algorithm will always insert the first vertex that satisfies the required conditions, regardless of whether other vertices are better options. This will produce a self–intersection if the first point chosen for insertion is to the left of the reference point, as shown in Figure A.6.

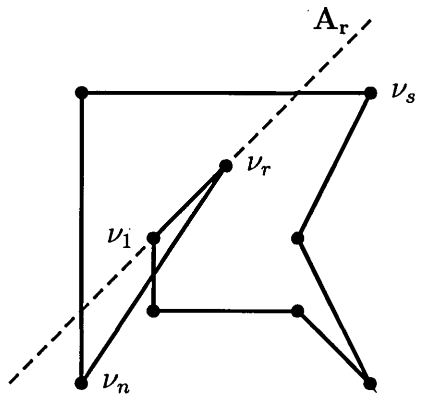


Figure A.6: Self–intersections in the curve created.

To solve these cases, it is necessary to force the algorithm to always choose the first point to be either below or to the right of the reference point. This check will be enforced until the next vertex *us* inserted is to the right and above *vr*. After inserting *va* it is safe to add points to the left of *vr* without producing self-intersections. The points that are discarded because of this condition are also moved to the beginning of *Xp,* so that they will not be eligible again until the end of the algorithm.

Figure A.7 shows the results of using this algorithm on the datasets presented before. For each dataset a curve was generated for every vertex. In the figure only the curve with the smallest curvature is shown.

(a) TAC = 105.495 **(b) TAC = 14.555** **(c) TAC = 49.998** **(d) TAC = 16.543**

Figure A.7: Curves obtained using the orientation algorithm.

A.5 Curves with one deviation from the convex hull

An algorithm proposed for creating curves with small curvature is to reconstruct them starting from the convex hull and allowing only one *deviation* with respect to the convex hull. In a curve with one deviation, only one of the segments of the convex hull will be open and all other points inside will be connected to the convex hull through that single gap. This would minimise the amount of *deviation vertices.*

The approach taken to link all internal points in a single deviation is to iteratively generate convex hull curves with the remaining points within the previous convex hull. This will produce a number of nested convex curves as shown in Figure A.8. Each new inner hull is identified by an index, referred to as the *layer depth* of the hull, where the outermost layer is number 0, the next one has depth 1, and so on until the final layer *k.* We will refer to these nested hulls as such: $CH_0, CH_i, \ldots, CH_k$. Figure A.9 shows the layers in the sample datasets being used. Inner layers have a lighter colour.

All of these layers are joined together to form a single curve by inserting the whole list of vertices of an inner layer into the outer layer. To obtain a correct ordering for the vertices in a single curve, the vertices in the list of an inner convex hull must be in the inverse order as the vertices in the immediate outer hull. In this implementation the even numbered layers will have a Counter-Clockwise orientation, while the odd numbered

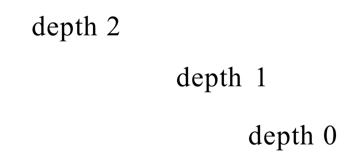


Figure A.8: Nested convex hulls of a dataset.

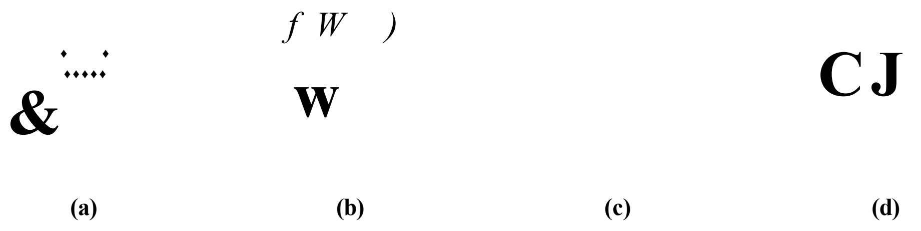


Figure A.9: The four test datasets showing the layers with different colours.

ones will be oriented Clockwise.

New inner convex hulls are inserted between segments of the outer hulls. The two segments where the insertion is done are removed, one from the outer and one from the inner hull. These are called *blank segments.* Afterwards two new segments are created to join the now open curves, and these are called *link segments.*

The curve being generated has an associated list of all the blank segments that have already been removed. Each time a new inner hull is inserted in the curve, there are two more entries in the list of blank segments, one for each of the hulls involved in the linking (see Figure A. 10). When all the layers have been added there will be two blank segments for each layer, except for the first and the last one, which only have one blank segment.

The link segments used to join outer hulls can be eligible to insert the new inner layers on them and become in turn blank segments. In this way a curve can be generated with the minimum number of link segments, thus generating curves with smaller curvature (see Lemma 2.4.3 in Chapter 2). However inserting new vertices at previous link segments can produce self-intersections in the curve. These cases must be verified before continuing adding more inner hulls. We explain later how to deal with these self-intersections.

To keep track of the segments that have been removed from each convex hull, there is

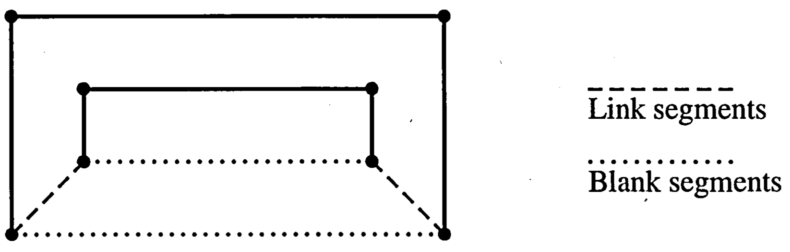


Figure A.10: Breaking the hulls at *blank segments* and joining them with *link segments*.

another list that contains segment structures for each of the depth levels. As the convex hulls are iteratively created, all the vertices are registered with the depth level to which they belong. This is used to identify when a new link segment is going over one or more intermediate hulls, when validating the curves generated. The method to build curves with one deviation is outlined in Algorithm A.2.

An important feature of this approach is that every time an inner layer is inserted, several possible curves can be generated, depending on the insertion points. Even by using varying parameters for selection, it is possible to find various points that produce the same initial results. As new hulls are inserted, the number of curves available branches out in a tree–like manner. Consequently, a large amount of curves can be generated from the same data. The program can find several of these curves, but is limited by the increasing memory requirements as the data grows. Some solutions have been proposed to limit the number of options available and are described later. The drawback is that the minimisation of parameters will be local, and the final solutions found may not be the optimal ones.

We implement two different methods to identify where to insert the inner layers into the outer curve: by measuring the distances between the layers or the angles formed by the new link segments.

A.5.1 By distances

For this method we pick the vertices closest to each other in contiguous layers, and insert the inner hull after the closest vertex of the outer layer. To compare the distances, all the vertices in either of the two convex hulls can be used as the reference. By using the hull with the most vertices we can measure all the distances in a single pass, and have the most options available for the insertions.

Using the sample data from Figure 2.6, we can determine two layers as shown in Figure A.11, $\text{CH}_0 = \{\nu_1, \nu_2, \nu_3, \nu_4, \nu_5, \nu_6\}$ and $\text{CH}_1 = \{\nu_7, \nu_8, \nu_9\}$.

Algorithm A.2 Curve–fitting with one deviation.

```
$\lambda_p \leftarrow \nu \in \mathbf{V}$.
$k \leftarrow 0$.
While $\lambda_c$ not empty
{
    $\mathrm{CH_k} \leftarrow \nu \in CH(\lambda_p)$.
    $\lambda_p \leftarrow \nu \notin CH(\lambda_p)$.
    $k \leftarrow k+1$.
}
$\lambda_c \leftarrow \mathrm{CH_0}$.
For each pair of convex layers $\mathrm{CH_k}$ and $\mathrm{CH_{k+1}}$
{
    $w_{min} \leftarrow \infty$.
    $index \leftarrow 1$.
    $i \leftarrow 1$.
    For each segment $\overline{\nu_i\nu_{i+1}} \in \mathrm{CH_k}$
    {
        Compute cost $f(i)$ of inserting $\mathrm{CH_{k+1}}$ in $\overline{\nu_i\nu_{i+1}}$.
        If $f(i) < w_{min}$, then
        {
            $w_{min} \leftarrow f(i)$.
            $index \leftarrow i$.
        }
        $i \leftarrow i+1$.
    }
    Insert $\mathrm{CH_{k+1}}$ in $\lambda_c$ after vertex $index$.
}
```

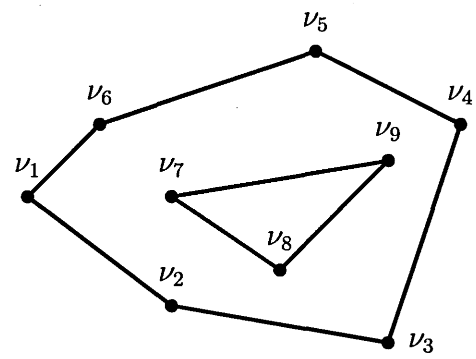


Figure A.11: Dataset with two layers of nested convex hulls.

The smallest distance between vertices of different layers is $d(\nu_4, \nu_9)$, so the points of $\mathrm{CH_1}$ are inserted in the segment $\overline{\nu_4\nu_5}$. When inserting them into the outer hull, the resulting closed curve will consist of the vertices in the following order: $\mathrm{P(V)} =$

{^i, ^2, ^3j ^4 ? *Vi,* ^55^6>^i}- Note that the order of the vertices in the inner layer was changed, to maintain the general orientation of the curve. Figure A. 12 shows the resulting polygon.

z's

Figure A. 12: Convex hull layers linked to form a single curve.

In this example, there would be two new entries into the blank segments list, which would be segments F4F5 of the outer hull, and Fpv °f the inner one. The new link segments are i/4z9 and *v5is7.* While the first of these segments has the minimal length, the length of the latter one is not considered.

There is a variation of this method, where the distances of both new link segments are added and used to do the selection of the insertion point. Figure A. 13 shows the curves obtained using only one distance to join the layers, while the curves in Figure A. 14 make use of two distances to determine where to join the hulls.

(a) TAC = 47.647 **(b) TAC = 13.356** **(c) TAC = 40.387** **(d) TAC = 15.577**

Figure A. 13: Curves with one deviation. The layers are joined using one distance.

Since this method relies on the distance between vertices of different hulls, it will not obtain all possible curves with one deviation for some datasets. It is possible for pairs of vertices in two layers to be so far away that they will always be discarded, even if joining the hulls at those vertices may prove to generate a polygon with smaller curvature, as in the example shown in Figure A. 15.

9 ^ 1

(a) TAC = 56.259 (b) TAC = 13.356 (c) TAC = 44.693 (d) TAC = 15.577

Figure A. 14: Curves with one deviation. The layers are joined using two distances.

Point too far

Figure A. 15: Curve not found by the algorithm.

There are three possible scenarios for self-intersections, which must be considered for this algorithm. There can be self-intersections between the new link segments joining two convex hulls, or the link segments could intersect either the inner hull or, when joining two non-contiguous hulls, any of the intermediate hulls.

A.5.2 By angles

The other method for joining the concentric convex hulls is to measure the angles in the interior of each hull, always looking for the ordering of the vertices that will produce the least curvature. This will ensure a small curvature overall for the whole curve, once all the inner hulls have been integrated into the curve.

The vertices in a closed curve are considered as an open curve, with one segment missing from the hull. The sum of the angles for these curves is computed, each time removing a different segment from the hull. Finally, the curve with the smallest curvature is chosen to be inserted into the outer hull (see Figure A. 16).

Other two angles are of importance when inserting the minimal curvature curve into the outer hull. Those are the angles at the joining vertices. They must also be minimised in order to obtain the curve with the least curvature.

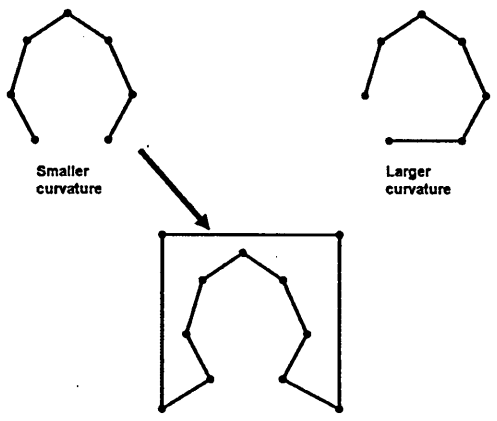


Figure A.16: Joining hulls by angles, the curve with the smallest curvature is chosen.

This latter condition can also lead to curves which have self–intersections, since the segment which minimises the angle of the new linking segments may be in the opposite side of the outer hull, creating self–intersections, as in Figure A.17.

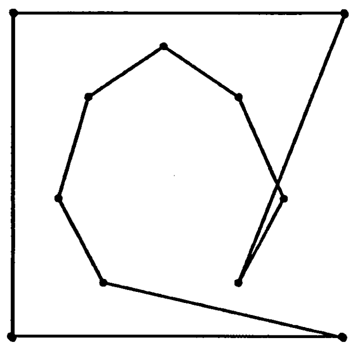


Figure A.17: Self–intersection when joining two hulls.

The algorithm creates other curves as well to have more options in the case the optimal segment to join the hulls produces such a self intersection. These other curves may not be locally optimal but have no self–intersections.

Since this approach relies on minimisation of the angles, it is capable of finding curves with curvature smaller than using distances to join the layers. However only the angles forming the curvature of the inner hull are minimised, while the angles at the link segments may be non–optimal, and can increase the curvature of the final polygon. Examples of obtained curves using this algorithm are shown in Figure A.18.

It is possible to reduce the number of curves generated with this algorithm by also considering the angle of the vertices that join two hulls. This will find the best curve that

(a) TAC = 50.548 **(b) TAC = 13.093** **(c) TAC = 48.337** **(d) TAC = 15.577**

Figure A. 18: Curves with one deviation. The layers are joined using two distances.

can be produced at each level depth, but will ignore other alternatives. Doing this it is possible that in future iterations a self-intersection will be found, caused by a previous choice. Unless previous choices of curves are stored, the algorithm has no means of back-tracking, meaning some datasets would not produce a valid curve at all when performing this kind of simplification.

A.5.3 Self-intersections

For the single deviation technique, three cases were found which could produce self-intersections in the generated curves:

- Segments linking the same hulls
  When joining two contiguous convex hulls, the two segments linking them could intersect each other in some special cases. This is solved by testing for an intersection of the two segments, and discarding the resulting curve.
  In the example of Figure A. 19, C H 0 is formed by the sequence of vertices from *Ui* to *z/6*, while CHx is made up of vertices and i/8. The segments that link both hulls are and i/8i/4, and they create a self-intersection.

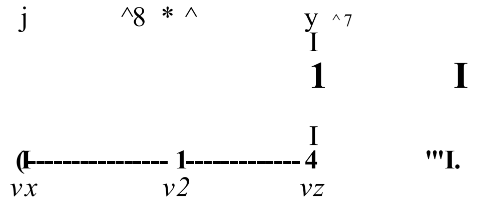


Figure A. 19: Intersection of the link segments joining two layers.

- Intersection of the inner hull

In some cases, the link segment may come across another segment of the inner hull. If the inner hull will be inserted after vertex ν_{o1} of the outer hull, then $\overline{\nu_{o1}\nu_{o2}}$ is the blank segment that will be removed from the outer hull when inserting the inner layer, and the inner hull will be inserted between the vertices ν_{o1} and ν_{o2} by new link segments. For example, in Figure A.20 $\mathrm{CH}_1 = \{\nu_1, \nu_2, \ldots, \nu_{n-1}, \nu_n\}$ and the new link segments are $\overline{\nu_{o1}\nu_1}$ and $\overline{\nu_n\nu_{o2}}$. Since ν_{o1} and ν_1 are the closest points, their segment will not intersect any other, however the segment $\overline{\nu_n\nu_{o2}}$ can have intersections.

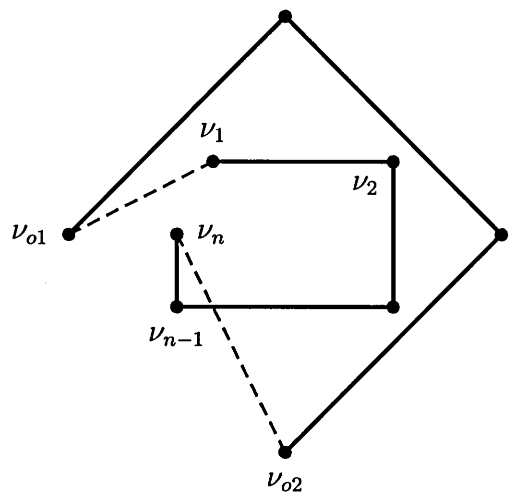


Figure A.20: Intersection of the inner convex hull with a new link segment.

To detect this kind of intersections, a triangle is formed using vertices: ν_{o1}, ν_{o2} and ν_n (see Figure A.21). It is then tested whether ν_{n-1} lies inside or outside of this triangle. If it is inside there will be a self–intersection at some point in the inner convex hull.

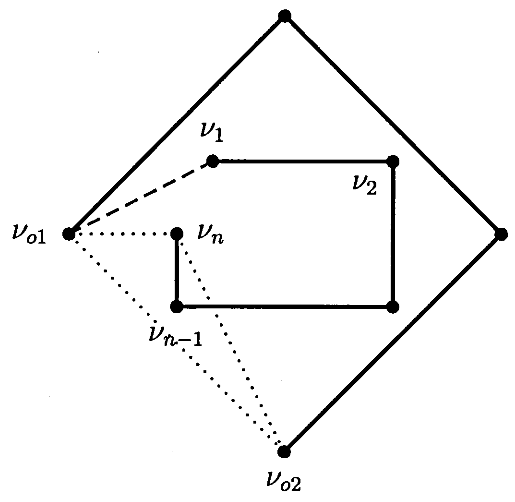


Figure A.21: Test triangle to identify self–intersections.

There are also some special cases when some of the vertices involved are collinear. In these cases, the order of the vertices is obtained from their distances. If the vertices lie in an incorrect order, then the linking segments may touch each other, producing an invalid curve.

- **Intersection with intermediate convex hulls**

  When there are three or more layers, it is possible that the endpoints of new link segments do not belong to contiguous layers. This happens when the segment where an inner hull is inserted into the curve is itself a link segment of outer layers. In such cases, a new link segment may go across one or more of the intermediate layers and produce an intersection.

  Because of the ordering of the vertices and hulls, this kind of intersection occurs only for the link that goes back from the inner hull to the outer one. The first point in the inner hull is always connected to a point in the immediately outer hull (see Figure A.22).

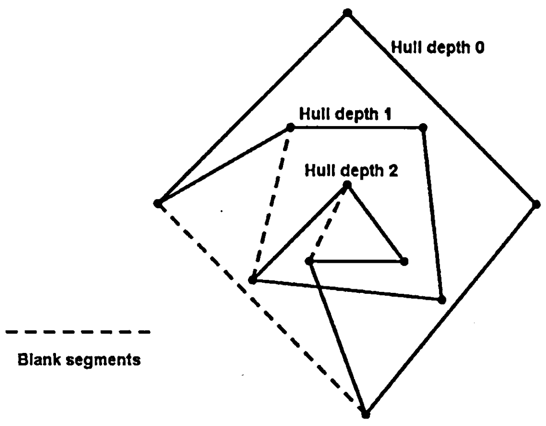


Figure A.22: Intersection of an intermediate convex hull.

There are valid cases where the link segment goes across several layers without producing self–intersections. This happens when the link goes through a blank segment that was removed when linking the previous layers. Thus the list of blank segments is used to identify this kind of self–intersections. When two hulls need to be linked across several intermediate layers, the new link is tested for intersection with the blank segments of the outer layers. If they intersect, then the new link will pass through an empty space, and thus will not produce a self intersection on the curve. Otherwise, if the new link does not intersect with the empty segments of all the convex hulls it crosses, then we conclude there will be a self–intersection at some point.

For each depth level, there are two entries in the blank segment list, but only the segment that corresponds to the link with the outer hull is tested, since the space left by this removed segment is the one that a new link can safely use without producing a self–intersection.

The endpoints of the blank segments are not considered when testing for intersection with the segments, since those vertices belong not only to the blank segment, but also to some other segment. If the new link passes over such point, then it will create a self–intersecting curve, as in Figure A.23.

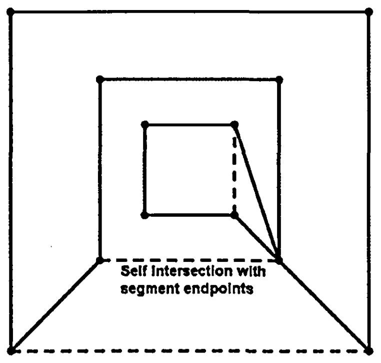


Figure A.23: Intersection of an intermediate hull at one of the vertices.

A.6 Curves with multiple deviations

The second new algorithm generates curves with multiple deviations using the convex hull as the starting point, and subsequently adding all of the remaining vertices individually. We use several different parameters to assign an *insertion cost* $f(\nu)$ to every vertex with respect to each segment of the curve where it can be inserted. Vertices with the smallest cost are inserted first.

The algorithm begins from the convex hull of the dataset, as explained in Section 2.2. The vertices that do not belong to the CH–curve are kept in the list λ_p. Initially the list for the new curve λ_c is a copy of the CH–curve. Then each one of the remaining vertices is tested for insertion at every segment of the current curve λ_c. Algorithm A.3 presents this curve–fitting method. The combination of a vertex ν_e and a segment $\overline{\nu_i \nu_{i+1}}$ with the smallest cost identifies the next vertex to be inserted and the location in the curve where it will be added. The vertex is then deleted from the list λ_p, and inserted into the final curve list λ_c. This continues until all the vertices in the dataset have been added to the final curve.

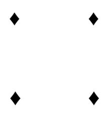
Algorithm A.3 Curve–fitting with multiple deviations.

```
$\lambda_c \leftarrow \nu \in \mathrm{CH(V)}$.
$\lambda_p \leftarrow \nu \notin \mathrm{CH(V)}$.
For each vertex $\nu_e \in \lambda_p$
{
    $w_{min} \leftarrow \infty$.
    $index \leftarrow 1$.
    $i \leftarrow 1$.
    For each segment $\overline{\nu_i \nu_{i+1}} \in \lambda_c$
    {
        Compute insertion cost $f(\nu_e)$ for $\overline{\nu_i \nu_{i+1}}$.
        If $f(\nu_e) < w_{min}$, then
        {
            $w_{min} \leftarrow f(\nu_e)$.
            $index \leftarrow i$.
        }
        $i \leftarrow i + 1$.
    }
    Insert $\nu_e$ in $\lambda_c$ after vertex $index$.
    Remove $\nu_e$ from $\lambda_p$.
}
```

Every time a new vertex ν_e is added to the current curve at a location i, the *insertion segment* $\overline{\nu_i \nu_{i+1}}$ is removed. It is replaced by two new link segments, joining the vertices $\overline{\nu_i \nu_e}$ and $\overline{\nu_e \nu_{i+1}}$. Both of the new segments must be tested for intersections, either with the rest of the segments of the existing curve, or with the remaining vertices in λ_p. The way to deal with these cases is explained in Section A.6.1. Figure A.24 shows an example of a curve constructed using this method, starting from the convex hull and gradually adding the rest of the vertices.

In general, the disadvantage of this approach is that it looks for the local minimum, but does not check for globally better alternatives. Each point chosen for insertion is the best given the current curve generated at the moment, but it is not possible to determine if the vertex chosen will be the best overall option after the insertion of all the vertices.

This technique will only generate one curve for each dataset by inserting at each moment the vertex with minimal cost. It does not keep track of the decisions taken before and cannot do backtracking. The number of different possibilities available at each point impose a very large requirement of computer resources to track the operations performed, in a manner that would permit backtracking.



(a) **(b)**

(C) **(d)**

(e) **(f)**

Figure A.24: Adding vertices to the convex hull to fit a curve over the whole dataset. Point cloud (a), convex hull (b), points added in sequence (c-e) and final curve (f).

The vertex cost parameters are based on local geometric properties of the vertices and edges affected during an insertion. All the different measures used are shown in Figure A.25. The vertex costs presented later refer to the names shown in these figures.

The following are the various parameters used to assign the insertion weight *f(i/)* to a vertex *v:*

Total curvature: *cu:* This is a brute force technique, it involves actually creating all the

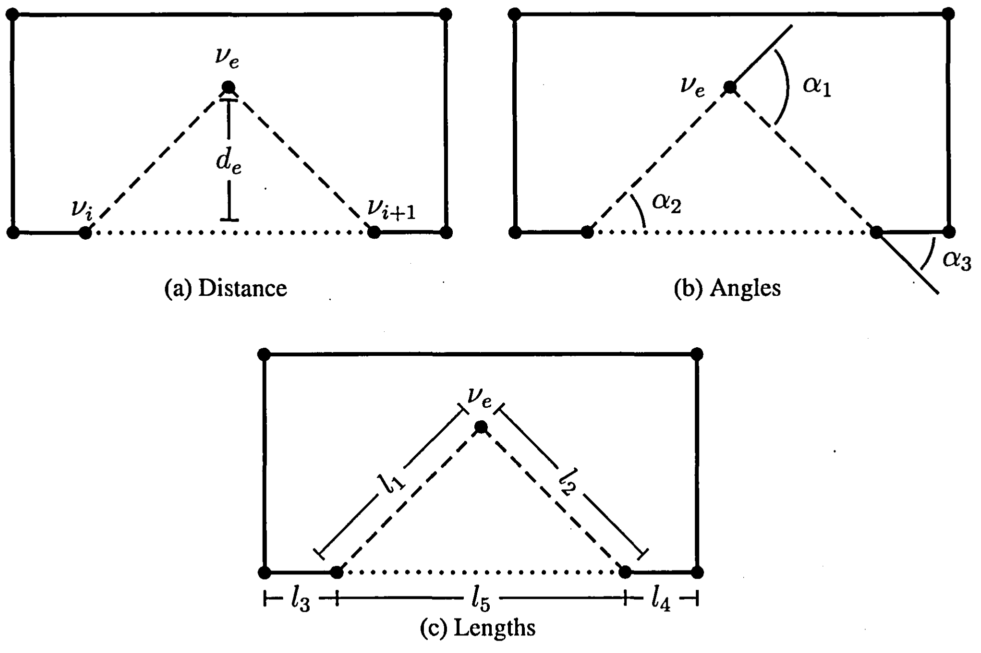


Figure A.25: Measurements used as minimisation parameters for the selection of vertices.

possible curves by comparing the difference in curvature resulting from inserting every possible vertex into every possible segment. At every step the insertion that produces the smallest curvature is kept and the rest are discarded.

Each unallocated vertex $\nu_e \in \lambda_p$ is inserted in every segment $\overline{\nu_i \nu_{i+1}}$ of the current curve λ_c. We measure the TAC of the resulting curve and remove the vertex. After all the tests, the option with minimal curvature is kept and the vertex chosen is inserted again in the corresponding segment. The results of applying this algorithm to the test datasets can be seen in Figure A.26.

Since this method directly minimises the curvature of the whole curve being created it will generally produce curves with a small TAC. However it is not guaranteed to find the curve with minimum curvature $\mathrm{P(V)}_{min}$, since it still focuses on local minima. The disadvantage of this test is that it is slower than the methods that will be presented next. To reduce the computation time, and avoid measuring all of the angles in the curve each time it is changed, we make use of the results presented in Section 2.3.1, and update the curvature of a new polygon by subtracting the value of the two angles that are modified by the insertion, and adding the value of the three new angles formed.

Distance from the insertion segment: d_e**:** In this method, the vertex selected to for insertion is the one with the smallest distance to any of the segments in the current

(a) TAC = 39.794 **(b) TAC = 14.556** **(c) TAC = 14.722** **(d) TAC = 16.070**

Figure A.26: Curves with multiple deviations using /(^ e) = *Ɑ*.

curve. The distance from a vertex to a segment is measured using the method shown in Section 2.2. Using the distances shown in Figure A.25 we define the vertex insertion cost of a vertex *ve* as:

$$f(v\,e) = de. \tag{A.6.1}$$

Figure A.27 shows the polygons generated with this algorithm. The use of a single parameter as the cost makes it simpler and faster than the previous method. This approach does not minimise the curvature directly, but does produce polygons with small curvature because of the small triangles formed by the inserted vertex *ve* and the endpoints of the insertion segment, and *ui+i*.

(a) TAC = 52.738 **(b) TAC = 14.556** **(c) TAC = 30.226** **(d) TAC = 16.070**

Figure A.27: Curves with multiple deviations using /(i'e) = *de*.

Distance from segment *de* and curvature [illegible] This is an extension of the previous method that uses the distance *de* to assign vertex insertion costs. Only in the cases where more than one vertex produce the smallest distance to any segment, then the curvature of the candidate insertions is measured and the vertex that produces the curve with smaller TAC will be selected. Using this method the TAC does not need to be computed at every step, but only as a deciding factor in the case of a ‘draw’ of two or more vertices with the same distance. The resulting curves can be seen in Figure A.28.

(a) TAC = 43.459 (b) TAC = 14.556 (c) TAC = 33.266 (d) TAC = 16.070

Figure A.28: Curves with multiple deviations using /(t'e) = *de* **and** *Ca*

Ratio of distance and the sum of both link segments: The vertex insertion cost in this method is the division of the distance from the vertex to a segment, over the sum of the lengths of the new segments produced by the insertion of the vertex in the selected location. The ratio to be minimised is obtained with the formula:

$$/K\ ) = \frac{r\ z\ r}{h\ T^{*}\ h} \qquad (A\text{-}6\text{-}2)$$

Figure A.29 shows the polygons generated with this algorithm.

(a) TAC = 43.982 (b) TAC = 14.556 (c) TAC = 19.738 (d) TAC = 16.070

Figure A.29: Curves with multiple deviations using /(^ e) =

Ratio of distance and length of blank segment: A similar approach is to use the length of the blank segment instead of the new link segments. In this case the cost is computed as:

$$f(u\,e) \quad \frac{f}{`5}\ . \qquad (A.6.3)$$

The curves generated are shown in Figure A.30.

New angle cvi: The vertex selected for insertion will be the one that adds the smallest concave angle when inserted in the curve. From Figure A.25 we use the angle *a* \ as the cost. This angle is obtained from three vertices: the two endpoints of the insertion segment, ^ and *vi+i,* and the new vertex *ve* being tested.

This algorithm is very simple, since it only requires the computation of one angle. It is very fast and directly optimises curvature. However, it does not consider the

(a) TAC = 43.982 **(b) TAC = 14.556** **(c) TAC = 19.738** **(d) TAC = 16.070**

Figure A.30: Curves with multiple deviations using /(^ e) = $\frac{de}{h}$ *

9

(a) TAC = 39.793 **(b) TAC = 14.556** **(c) TAC = 14.722** **(d) TAC = 16.070**

Figure A.31: Curves with multiple deviations using *f(ve)* —

orientation of the angles produced. Figure A.31 shows the polygons generated with this algorithm.

When a new vertex is inserted, there are a total of three angles that are modified from the curve. The previous approach only uses the angle at the inserted vertex *ve,* however also the angles at the vertices and *vi+i* are modified and also affect the resulting curve. To address this problem we can extend this method to include the other two angles in the cost function, to make it:

$$/(\hat{}e) - a_l \mathrm{T} \alpha_2 + a_3. \quad \text{(A.6.4)}$$

The curves generated using this variation are presented in Figure A.32.

(a) TAC = 40.841 **(b) TAC = 18.343** **(c) TAC = 14.722** **(d) TAC = 16.070**

Figure A.32: Curves with multiple deviations using /(^ e) = + &2 + ^3-

Product (ai(Zi + *h))'** This method uses the angle at the new inserted vertex and the lengths of the new link segments. The angle is multiplied by the sum of lengths of both link segments:

/(^e) = *a i{h + h)-* (A.6.5)

Figure A.33 shows the polygons generated with this algorithm.

(a) TAC = 67.021 **(b) TAC = 30.489** **(c) TAC = 42.839** **(d) TAC = 16.070**

Figure A.33: Curves with multiple deviations using /(z'e) = o i(/i + ***h)-***

This algorithm can also be extended to make use of the three angles modified and including also the length of the segments adjacent to the insertion segment. In this case the cost function is:

J iye) —(a i{h + h)) + {a 2{h + ^3)) + (<23(^2 + *U))-* (A.6.6)

The curves generated using this variation are presented in Figure A.34.

(a) TAC = 48.496 **(b) TAC = 28.085** **(c) TAC = 22.246** **(d) TAC = 15.577**

Figure A.34: Curves with multiple deviations using /(^ e) = ^ 1(^1 + ***h)*** **for three angles.**

Product (or(M2)): This is obtained by multiplying the angle at the new vertex with the product of the lengths of the link segments. The product to be minimised is obtained with the formula:

f(v e) = a,(A.6.7)

Figure A.35 shows the polygons generated with this algorithm.

(a) TAC = 67.021 **(b) TAC = 20.676** **(c) TAC = 33.957** **(d) TAC = 16.070**

Figure A.35: Curves with multiple deviations using *f(ve) = v-iihh)-*

Again this method can be extended to include the two adjacent angles and segments, making the cost function:

/(^e) —*{ai{hh))* + (^2(M3)) + *{a3{hU))-* (A.6.8)

The curves generated using this variation are presented in Figure A.36.

(a) TAC = 49.826 **(b) TAC = 25.377** **(c) TAC = 33.266** **(d) TAC = 15.577**

Figure A.36: Curves with multiple deviations using *f{ise)* = ®*i{hh)* **for three angles.**

Ratio ($_{l1+l2}$ I: The ratio to be minimised is obtained with the formula:

$$\frac{a\backslash}{h + h} \quad (A.6.9)$$

Figure A.37 shows the polygons generated with this algorithm.

(a) TAC = 59.143 **(b) TAC = 22.246** **(c) TAC = 14.722** **(d) TAC = 16.070**

Figure A.37: Curves with multiple deviations using *f(ve)* —

Considering the three angles:

- (iT t) +(ilTi;) +(s r s) ■ (AA10)

The curves generated using this variation are presented in Figure A.38.

(a) TAC = 80.693 **(b) TAC = 15.840** **(c) TAC = 30.390** **(d) TAC = 18.408**

Figure A.38: Curves with multiple deviations using *f(ve)* — **f°r three angles.**

Ratio (±*aix*- *h* +*h*) *:* The ratio to be minimised is obtained with the formula:

f(m)\ = x x x - *h* "I~ *h* (A.6.11)

Figure A.39 shows the polygons generated with this algorithm.

(a) TAC = 45.719 **(b) TAC = 14.556** **(c) TAC = 33.266** **(d) TAC = 16.070**

Figure A.39: Curves with multiple deviations using *f{ye)* — **X t •**

Considering the three angles:

M = **x f r**) + **x f r** + X T X **V*h*** ^ ***h J*** \ ***h*** ^ ***h J*** \ ***h*** ^ ***k*** (A.6.12)

The curves generated using this variation are presented in Figure A.40.

Product **("(x))**: The product to be minimised is obtained with the formula:

/ (^ e) = « 1 *(rPj-* (A -6 1 3 >

(a) TAC = 46.567 **(b) TAC = 25.377** **(c) TAC = 24.367** **(d) TAC = 15.577**

Figure A.40: Curves with multiple deviations using /(^ e) = $\bar{i}\ i^{4}\bar{h}$ for three angles.

(a) TAC = 73.490 **(b) TAC = 21.641** **(c) TAC = 14.722** **(d) TAC = 16.070**

Figure A.41: Curves with multiple deviations using *f(ise) = an*

Figure A.41 shows the polygons generated with this algorithm.

Considering the three angles:

f M ~ a (^)+(2Hi)+(3hi (A.6.14)

The curves generated using this variation are presented in Figure A.42.

(a) TAC = 80.723 **(b) TAC = 36.063** **(c) TAC = 39.619** **(d) TAC = 19.455**

Figure A.42: Curves with multiple deviations using *f(ve) =* **f°r three angles.**

Square sum of three angles: The product to be minimised is obtained with the formula:

f(ve) = a\ + a\ + a\. (A.6.15)

Figure A.43 shows the polygons generated with this algorithm.

(a) TAC = 39.793 (b) TAC = 27.115 (c) TAC = 14.722 (d) TAC = 16.070

Figure A.43: Curves with multiple deviations using *f{u*e) = *af* + + a 3 -

Square angles over sum of lengths: The product to be minimised is obtained with the formula:

/("«) = *a* I j1 + ^ 2 / *fix'* V ^ l + ^ 3 / + *OL*\ ^4 (A.6.16)

Figure A.44 shows the polygons generated with this algorithm.

(a) TAC = 39.793 (b) TAC = 16.574 (c) TAC = 14.722 (d) TAC = 18.340

Figure A.44: Curves with multiple deviations using /(^ e) = f°r three angles.

Square angles over sum of inverse lengths: The product to be minimised is obtained with the formula:

/("•) = T ^ T *h* ^ *h* + T ^ T + h (A.6.17)

Figure A.45 shows the polygons generated with this algorithm.

Product of square angles and product of inverse lengths: The product to be minimised is obtained with the formula:

1 1 1 1 1 1 (A.6.18)

Figure A.46 shows the polygons generated with this algorithm.

Product of square angles and product of lengths: The product to be minimised is obtained with the formula:

f(ye) —{a l(,hh)) + (<*2 (^ 3)) + *{a l{hU))-* (A.6.19)

(a) TAC = 44.115 **(b) TAC = 25.047** **(c) TAC = 14.722** **(d) TAC = 15.577**

Figure A.45: Curves with multiple deviations using $f(ve) =$ **(XTX) f°r three angles.**
ll l2

(a) TAC = 39.793 **(b) TAC = 26.617** **(c) TAC = 14.722** **(d) TAC = 18.408**

Figure A.46: Curves with multiple deviations using $f(ae) =$ **f°r three angles.**

Figure A.47 shows the polygons generated with this algorithm.

(a) TAC = 40.841 **(b) TAC = 30.749** **(c) TAC = 14.722** **(d) TAC = 15.577**

Figure A.47: Curves with multiple deviations using $f(ae) =$ **(<*1 (^ 2)) for three angles.**

Sum of total curvature and curve length over diameter: This measure makes use of the main features of the curve: the curvature and the total length. The length of the curve is normalised by dividing it over the *diameter* of the polygon. Here the diameter *d* is defined as the longest distance between any two vertices in the convex hull of the curve.

The minimisation parameter is obtained as:

n / *sr^m* *ı*

/(„ e) = 5 > + iff (A.6.20)

i=i \

where n is the number of vertices already inserted in the curve and m is the number of line segments in the curve. Figure A.48 shows the polygons generated with this algorithm.

(a) TAC = 39.793 **(b) TAC = 14.556** **(c) TAC = 14.722** **(d) TAC = 16.070**

—)•

A.6.1 Self-intersections

To avoid self-intersections, each new segment added to the curve is tested for two cases:

- Intersection of existing segments
  New segments added to the curve have the risk of crossing over the current curve. The likelihood of this happening depends on the parameter being used for minimisation. There is no simple way to detect these cases, since vertices can be inserted anywhere and in any order. The only way to detect self-intersections is by testing each of the segments.
  The amount of tests is reduced by first comparing the projections on the Y axis of the segments involved. The actual testing for self-intersections will be carried out only when these projections overlap at some point for the segments being tested (de Berg *et al.* 1997).
- Overlapping of unallocated vertices
  A new segment may go over an unallocated vertex *Vj* E *Ap*. This would create a problem when finally inserting *Vj,* since it could create overlapping segments or intersections.
  To identify these cases, new segments are tested for collinearity with the vertices in *Ap*, and if they are, then another evaluation is done to check if the vertex lies inside of the segment.
  When an intersection of this kind occurs, the point that is closest to the original curve is inserted instead of the new point proposed by the normal algorithm. This

will eliminate the problem but, depending on the choice of the minimisation parameter, may not keep the optimisation done by the algorithm.

A.7 Results and evaluation of the algorithms

All the curves generated by our algorithms are closed polygons with n vertices and n edges. They are connected in Counter–Clockwise direction. The main parameter used to evaluate and discriminate curves is their Total Absolute Curvature, measured from the external angles at each vertex. The best curve is considered to be the one with the smallest TAC.

The algorithm of reconstruction based on orientation has limited capability to correctly reconstruct the source curve from the point cloud. Only for very particular cases it will get the correct result, as shown in Figure A.7. In general the best curves are those generated by choosing a reference point ν_r on the convex hull of the dataset.

In general, the behaviour of the algorithm based on one deviation is very dependant on the data. It is capable of finding curves with very small curvature, specially when the data can be described with two layers, or when the lengths of the blank segments is small. However, when there are more than two layers and the length of blank segments is very large, then the curvature of the polygons generated is much bigger than what can be obtained with the multiple deviation algorithms.

For the algorithm to generate curves with multiple deviations, several insertion cost functions were presented. The best results are obtained by minimisation of the weights $f(\nu_e) = \alpha_1$ and $f(\nu_e) = \sum_{i=1}^{n} \alpha_i + \left(\frac{\sum_{j=1}^{m} l_i}{d}\right)$. This last parameter is computationally more demanding, but can give better results in a wider range of data. Minimisation of the single angle, on the other hand, is fast and gives a good approximation of the data in most cases.

The source for the data in Figure A.1(c) is an *hypotrochoid.* None of the algorithms was able to exactly recreate the original curve, since it contains self–intersections. However, a good approximation is obtained using many of the parameters presented. For the five point dataset in Figure A.1(a), since the algorithms are based on the convex hull of the point cloud, in all cases the result is a single closed curve, and it is not possible to identify independent shapes.

The algorithms developed do not require a specific sampling rate to correctly estimate the curve. A closed curve will always be generated, regardless of the point density in the dataset. This contrasts with other methods, like the *Crust* and *β–Skeleton*, that demand denser sampling at sharp corners. However, our algorithms are unable to distinguish separate elements in a point cloud, and will treat all vertices as belonging to the same object.

A.8 Conclusions

This chapter has presented two new algorithms for curve reconstruction, based on the convex hull. The first algorithm generates curves with one deviation from the convex hull, by successively generating nested convex hulls and joining them into a single curve. The second algorithm progressively adds vertices to the convex hull until all vertices in the dataset are included. The insertion of the vertices is made by minimisation of various criteria of angles and distances in the affected neighbourhood.

The results of the reconstruction algorithms are compared by means of the Total Absolute Curvature. From the results we observe that smoother curves have smaller TAC. However a curve with minimum TAC is not necessarily the most adequate for all datasets.

Using the methods presented, it is possible to obtain a large amount of different curves for every dataset. The results described here are not final yet, as this is an ongoing research project. Further evaluation of the reconstruction algorithms is necessary to determine their usefulness in various applications, where the vertex coordinates come from different sources. This requires a measure of the similarity of the curves generated with the original data, in terms of curvature and other shape parameters, like area and perimeter.

The methods described here for reconstruction of planar curves could be extended to perform reconstruction of surfaces in three–dimensional space. Several techniques exist for the computation of the convex hull of a three–dimensional object. The main requirement for an extension of the algorithms presented here is a measure of curvature at the vertices in space.