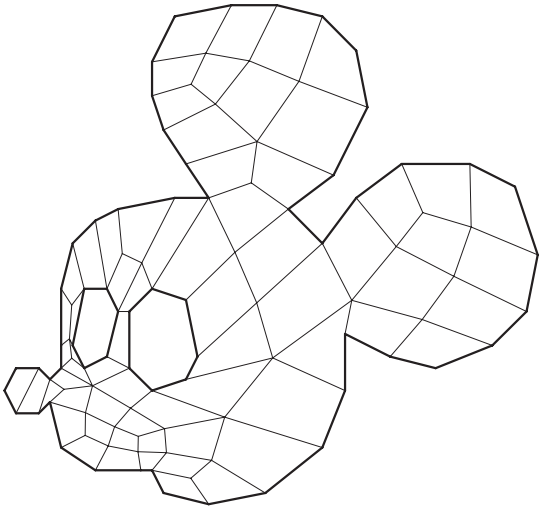
Q-Morph -
Implementing a Quadrilateral Meshing Algorithm

Karl Erik Levik
karll@ifi.uio.no



The thesis is submitted in partial fulfillment of the requirements for the degree of siv.ing. at Department of Informatics, University of Oslo.

Abstract

This thesis treats the implementational and some theoretical aspects of the Q-Morph algorithm for 2D domains. The main application areas are within FE methods. Q-Morph uses an advancing front method for generating unstructured, almost all-quadrilateral meshes containing at most one triangle, and few irregular nodes. The main algorithm is described in [16], while the post-processing methods are covered in [9, 4].

In addition to an introduction to the Q-Morph algorithm, the thesis also consists of some general background material for FEM meshing, discussions of many issues concerning the implementation, a presentation of important results, and a discussion of possible improvements. To ensure that the implementation conforms to the specifications of the algorithm, it has been tested on a number of different cases.

Preface

Chapter organization

For readers already familiar with the finite element method (FEM), I would suggest skipping some of the introductory material in the first chapter. If you know the basics of FEM, but perhaps not so much about meshing, then a good place to start is section 1.4. This section attempts to give a brief justification for the employment of quadrilateral elements, and presents some well-known arguments for why sometimes a better result can be obtained with quadrilateral rather than triangular elements. The next section might also be of interest. Here I discuss the properties of an optimal mesh.

For a more visually stimulating approach, one could have a glance at the figures and accompanying text in chapter 5.

Although it would be an advantage to know a little about the Q-Morph algorithm beforehand, one might hopefully find some of the necessary information in chapter 2. This chapter is devoted entirely to explaining some issues regarding the algorithm, or rather, the algorithms: the Q-Morph algorithm and the algorithms which it employs for post-processing. For topological cleanup, I have chosen the CleanUp algorithm [9], and for global smoothing the optimization-based smoothing algorithm [4].

Almost everything concerning the actual implementation of the algorithms is found in chapter 3. I suggest some possible improvements to the implementation in chapter 6.

Various results are presented in chapter 4: approximate values for the program constants and the results from some simple experiments.

Among other things, the appendix has a short glossary of some terms used in the text.

Acknowledgments

All scientific achievements, however minuscule and insignificant, are indebted to the efforts of the scientists who paved the way in the past. (E.g. the authors of [2].) This thesis is no exception in this regard.

In particular, I would like to thank the following people for their efforts, good advice, inspiration, and encouragement. Obviously, I must thank prof. Hans Petter Langtangen, my tutor and lecturer, who initially got me started in the world of meshing. Also obviously, I acknowledge the authors of the Q-Morph algorithm. The implementation of this algorithm is really what this thesis is all about.

Next, Øyvind Hjelle and Michael Floater, who were lecturing the course INF-TT in the autumn semester of 2001. Several of the topics taught in this course proved useful as background material. Equally important was the book "Automatic Mesh Generation - Application to Finite Element Methods", written by Paul-Louis George, which I spent several months studying. Although some illustrations in the thesis were created using my own software (MeshDitor), I also produced some illustrations in the applications dia and jPicEdt, and my thanks go to their respective authors, Alexander Larsson and Sylvain Reynal.

Most important, however, was the support from my friends and family. My parents have always been my most eager supporters, and have kindly provided for me during my studies. Likewise, the company of my good friends has been a great relief and inspiration. You all mean so much to me: Paul, John Einar, Jan Arve, Arne, Vanja, Ulrik.

Thank you!

Oslo, 28th October 2002

Karl Erik Levik

Contents

Chapter 1

Introduction

1.1 An analogous problem for the uninitiated reader

Throughout the entire course of writing this thesis, I have been struggling with finding a truly simple explanation to what it is actually about. For a long time it seemed impossible to give a short and intelligible explanation to people outside of the mathematical community.

Eventually, I found an analogous problem in garden architecture that I believe even non-scholars will understand quite easily.

Suppose that you want to build a slate paving in your garden and that the outline of the paving consists of straight lines that do not necessarily form any simple geometrical figure. Furthermore, suppose also that you want this covering-up of slabs to be somewhat pleasing to the eye. You have decided that this can be accomplished by fulfilling two criteria:

1. No corner of any slab should meet with the side of another slab, only with the corners of adjacent slabs.
2. Each slab must be a quadrilateral and should as closely as possible resemble a square.

So how exactly is the covering-up to be accomplished? By simplifying only slightly, one can express the purpose of my thesis as to write (and document) a computer program which automatically solves this problem. It computes the number of slabs needed, and the exact shape and position

of each slab.

The real advantage of this analogy is perhaps that almost anyone can relate to the problem of covering-up a surface by slate slabs. Although I find this to be a striking analogy, I do not purport having discovered a new and lucrative application area for quad-meshing!

1.2 Preliminaries

In the context of mesh generation, or *meshing*, there are some basic terms that must be learned before one can fully take advantage of the literature of the field. Firstly, there are *structured meshes* (or *grids*) as opposed to *unstructured meshes*, see figure 1.1. A structured mesh has a uniform topological structure, while an unstructured mesh has not. That is, every node in a structured mesh has the same number of neighbours (except for the boundary nodes). Because of this, assuming that the nodes are kept in some list-like structure, one can refer to any neighbour of a node by means of simple addition. In an unstructured mesh each node must maintain a list of pointers to its neighbours.

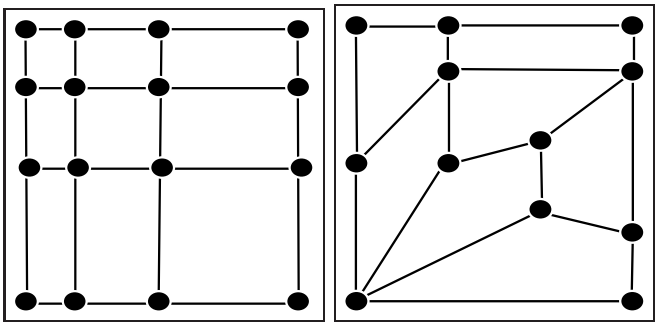


Figure 1.1: *Left:* A structured mesh. *Right:* An unstructured mesh.

Still, unstructured meshes are often preferred. Many problems have domains that just can not be adequately represented by structured meshes. The physics of yet other problems may require a fine distribution of nodes in one particular area of the domain, but allow for a much coarser distribution in the remaining area. When employing a structured mesh in such a problem, the need for a fine mesh in the one area will lead to the creation of redundant nodes in the remaining area. Thus, the number of nodes required by a structured mesh would probably be somewhat greater

than the number required by an unstructured mesh. The inevitable result is higher memory requirements and excessive computations.

There is also a third, perhaps less evident, advantage of using unstructured meshes. A comparison of relatively coarse structured and unstructured meshes would reveal that unstructured meshes can more closely follow any interior contours than structured meshes can. This is important in problems where different layers possess different physical characteristics.

The accuracy of the solutions to some problems greatly depends on that the meshes closely fit the domain boundaries. This can be achieved by allowing for *higher-order elements*; elements with curved sides. Commonly used higher-order elements are e.g. quadratic and cubic triangles and quadrilaterals. The sides of these elements have the shapes of second-order and third-order functions, respectively.

Furthermore, within quadrilateral meshing, there is a distinction between *direct methods* and *indirect methods*. Indirect methods require an initial triangle mesh as input, while direct methods do not. Traditionally, the direct methods have provided higher quality elements with fewer irregular nodes[1]. The indirect methods are considered to be fast, since they can utilize topological information from the initial triangle mesh. One famous direct method is the paving algorithm [2]. The Q-Morph algorithm [16] is an example of an indirect method.

This particular algorithm, the Q-Morph algorithm as described in [16], is the focus of this thesis. Q-Morph is an indirect method for generating unstructured quadrilateral meshes. The application area for meshes with quadrilateral elements are mainly restricted to FEM, whereas triangle meshes are also used within several other fields of science:

- map-making/terrain modeling
- geological modeling
- 3D maps
- flight simulators
- visualization
- medical modeling
- CAD/CAM - computer aided design/computer aided modeling

[1] See page 62 for a definition of regular and irregular nodes.

1.3 FEM, The Finite Element Method

FEM is a numerical method for solving partial differential equations (PDEs) on some given 1D or multidimensional domain. PDEs arise in a wide range of physical problems, such as heat conduction, diffusion, waves, fluid flow, solid and fluid mechanics. It is not the purpose of this thesis to offer any sort of thorough introduction to FEM, and nor do I possess skills even close to what would be required for such a grand task. Nevertheless, from a pedagogical and motivational point of view, it seems wise to try and present a subject (meshing in this case) within the context to which it belongs.

Therefore I will try to outline the very basic ideas of FEM, just to show how elements such as triangles and quadrilaterals come into play. A much more comprehensive introduction is given in [10], from which this section borrows heavily.

The main idea of FEM is to seek an approximation

$$\hat{u} = \sum_{j=1}^{M} u_j N_j(\mathbf{x})$$

to the unknown function $u(\mathbf{x})$. The $N_j(\mathbf{x})$-functions (often referred to as *basis functions*) are prescribed, and the coefficients u_j are unknown. Now, the goal is to minimize the error $u - \hat{u}$. In general this error is unknown. Therefore, within the so-called *weighted residual method*, WRM, of which FEM is a special case, the goal is to minimize the *residual*, R, arising when replacing u with $\hat{u}$ in the PDE. We hope that a small residual indicates a good approximation to u. In WRM, the weighted mean of R over Ω must vanish for M linearly independent and prescribed *weighting functions*, W_i:

$$\int_{\Omega} R\, W_i \, d\Omega = 0, \qquad i = 1, \ldots, M$$

Writing this out for some particular problem, we would see that it was simply a linear algebraic system for $u_1, \ldots, u_M$, and it could be written

$$\sum_{j=1}^{M} A_{i,j} u_i = b_i, \qquad i = 1, \ldots, M \tag{1.1}$$

where the matrix entries $A_{i,j}$ and vector entries b_i were integrals evaluated over the domain Ω.

In matrix form it would read $Au = b$, and we can now solve for u, selecting from a well of solution methods, ranging from the simple and slow

Gaussian elimination to the sophisticated and fast multigrid methods. We refer to $Au = b$ as the *global system*.

1.3.1 Basis functions and weighting functions

However, I have not yet said anything about how to choose the basis functions and the weighting functions. The most common choice of weighting functions in WRM is $W_i = N_i$. This is called Galerkin's method.

The *essential boundary conditions* impose some restrictions on the basis functions. For $\hat{u}$ to fulfill the ess. boundary conditions, it is a good idea to require basis functions that are zero on the boundary, $\partial\Omega$. Now, if the ess. boundary conditions are given as $u = \psi(\mathbf{x})$ (on the boundary, $\partial\Omega$, obviously), then we can simply rewrite the expression for $\hat{u}$ as:

$$\hat{u} = \psi(\mathbf{x}) + \sum_{j=1}^{M} u_j N_j(\mathbf{x})$$

Note that elsewhere in the domain we still use the old expression for $\hat{u}$, so here the value of $\psi(\mathbf{x})$ is irrelevant.

Now, FEM further consists in decomposing the domain into non-overlapping elements. In 1D, the only possible element type is the line segment, and the basis functions, N_i, are simple polynomials over each line segment. In 2D, common FEM elements are triangles and quadrilaterals. The appropriate basis functions for each of those element types are surface patches composed of triangles or quadrilaterals, respectively.

Globally, each basis function is a piecewise polynomial (1D) or a surface patch (2D) that is zero on every node except for node i, where it is 1:

$$N_i(\mathbf{x}) = \delta_{ij},$$

where δ_{ij} is the Kronecker delta:

$$\delta_{ij} = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}$$

This means that we get $N_1 = 1$ on node 1 and zero on all other nodes, $N_2 = 1$ on node 2 and zero on all other nodes, and so on, see fig. 1.2.

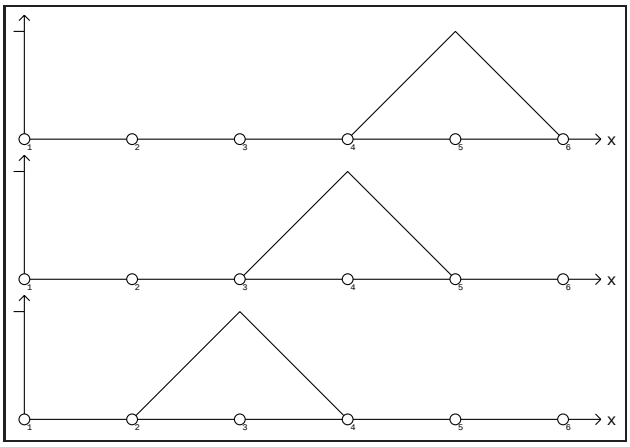


Figure 1.2: 1D linear basis functions over some elements. The lower function shows $N_3(x)$, the middle function shows $N_4(x)$, and the top one shows $N_5(x)$.

1.3.2 Local coordinates, the element matrix and vector

Recall that the matrix entries $A_{i,j}$ and vector entries b_i in equation 1.1 are integrals over the entire domain Ω. Instead of integrating over Ω, we can integrate over each element e, and take the sum:

$$A_{i,j} = \sum_{e=1}^{m} A_{i,j}^{(e)},$$
$$b_i = \sum_{e=1}^{m} b_i^{(e)}$$

where e is an index running over all the elements, and $A_{i,j}^{(e)}$ and $b_i^{(e)}$ are integrals over element no. e.

We can disregard all nodes that are not part of element no. e when calculating $A_{i,j}^{(e)}$ and $b_i^{(e)}$. The reasons for this are that the basis functions are zero for all nodes in the domain except for those that are part of element no. e, and the entries $A_{i,j}^{(e)}$ involve basis functions N_i and N_j as factors. Also the entries $b_i^{(e)}$ involve N_i as a factor. Thus, only when both nodes no. i and j are part of element no. e will $A_{i,j}^{(e)}$ be non-zero, and only when node no. i is part of element no. e will $b_i^{(e)}$ be non-zero.

It is now convenient to work at a local level and apply local coordinates ξ, η and node numbers r, s instead of working at the global level with global coordinates x, y and node numbers i, j. For each element e, we collect the

nonzero contributions to each matrix $A_{i,j}^{(e)}$ and vector $b_i^{(e)}$ into an *element matrix* $\tilde{A}_{r,s}^{(e)}$ and *element vector* $\tilde{b}_r^{(e)}$, respectively.

Furthermore, we will of course need to transform the basis functions, their derivatives, and the integrals, into local coordinates. Each physical element[2] is mapped onto a reference element with local coordinates. To evaluate the integrals now deduced, numerical integration is applied.

1.3.3 Essential boundary conditions

The essential boundary conditions assign values to some of the nodes in vector u, e.g. $u_1 = u_B$. Before we assemble the global system from the element matrices and vectors, it is now a good idea to make them fulfill the ess. boundary conditions. If we are to accomplish this on the element level, we must manipulate the corresponding element matrix and vector accordingly. Node no. 1 is part of element no. 1, so its corresponding matrix and vector are the ones that we want. If we use linear 1D elements, then this is a 2×2 linear system:

$$\begin{pmatrix} \tilde{a}_{11} & \tilde{a}_{12} \\ \tilde{a}_{21} & \tilde{a}_{22} \end{pmatrix} \begin{pmatrix} \tilde{u}_1 \\ \tilde{u}_2 \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \end{pmatrix}$$

We want $u_1 = u_B$, so we insert this as the first equation, and get:

$$\begin{pmatrix} 1 & 0 \\ \tilde{a}_{21} & \tilde{a}_{22} \end{pmatrix} \begin{pmatrix} \tilde{u}_1 \\ \tilde{u}_2 \end{pmatrix} = \begin{pmatrix} u_B \\ \tilde{b}_2 \end{pmatrix}$$

To make the system fulfill other ess. boundary conditions, similar manipulations can be performed. However, this was only one possible way of fulfilling the ess. boundary condition. If the element matrix was initially symmetric, and we wanted to preserve this property, a little more sophisticated manipulations would be required. Symmetric element matrices yield a symmetric global matrix, and this allows for faster linear solvers. Hence, symmetry is a rather desirable property.

1.3.4 Assembly of the global system

Previously, the global coefficient matrix was written as a sum of element matrices. Now we need to put these back into a global system. This is accomplished by means of a function mapping from local into global node numbers: $i = q(e, r)$ and $j = q(e, s)$. The entry (r, s) in element matrix no.

[2] For 2D domains, physical elements are e.g. triangles and quadrilaterals.

e is a contribution to the entry (i, j) in the global coefficient matrix, so we can obtain the global coeff. matrix by applying the update formula

$$A_{q(e,r),\, q(e,s)} := A_{q(e,r),\, q(e,s)} + A^{(e)}_{r,s}$$

to an an initially empty matrix, for all entries in each element matrix.

1.4 Briefly on triangular versus quadrilateral elements

The reason for not always sticking to meshes composed solely of linear triangles, is that it has been shown that if the grading and size of the quadrilateral elements are carefully controlled, and thereby generating well-shaped quads, an increase in efficiency over pure triangle meshes can be achieved [11, 13].

Besides, according to [20]:

> Some analyses require all-quad meshes. However, even if the solver allows triangles, triangles typically do not perform as well as quadrilaterals. This is especially true for linear analyses (Zienkiewicz, 1977).

The reference is presumably for [21], in which the performance of linear triangles is compared to that of linear and higher-order quadrilaterals.

Owen also writes favourably of quadrilateral meshes in [14], where he voices the position that quadrilaterals have superior performance to triangles when comparing an equivalent number of degrees of freedom.

That being said, the optimal choice of element type still greatly depends on the specific problem at hand. Sometimes the physical characteristics of the problem favours one particular element shape. For example, quadrilaterals should be preferred in situations where alignment of elements is important.

When efficiency is the main concern, linear elements are generally superior. Other times accuracy is more important, and in general one then has to resort to higher-order elements.

However, it would be a strenuous task to obtain a truly clear conception of all the issues concerning element type, and it is beyond the scope of this thesis to elaborate any further. Nevertheless, a glance at the referenced

literature above leaves the impression that there is certainly no reason to doubt that quadrilateral meshes do indeed have an important role to play in many FEM applications, and that it is worthwhile to pursue the quest of developing and implementing algorithms for high quality quad meshing.

1.5 The optimal mesh

In the context of FEM, the ideally shaped element is equiangular, that is, an element whose angles are equal in size. Elements with large or small angles can degrade the quality of the numerical solution, and due to the shapes of the input geometries, one will in practice to some extent have to be content with such lesser attractive angles.

Still, there is a limit to how poorly shaped an element can be. A triangle must not contain an interior angle which is greater than 180°, while a quadrilateral can not have more than one interior angle which is greater than 180° [4]. Elements not conforming to these criteria are termed *inverted*.

On the other hand, small angles can cause the coupled systems of algebraic equations produced by FEM (or other numerical methods) to be ill-conditioned [19]. When employing direct methods in such a case for solving the system of equations, the accuracy of the solution is degraded by roundoff errors. Conversely, iterative solution methods will suffer with slow convergence.

Large angles pose two other problems: *Discretization errors* may occur when applying FEM (or other numerical methods) to a mesh containing elements with large angles. The solution may be far less accurate than under more favourable circumstances. In theory, the solution yielded from FEM should approach the exact solution as the element size decreases. Nevertheless, it has been shown [1] that such convergence may fail to occur when the mesh contains angles approaching 180°.

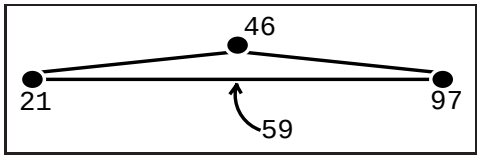


Figure 1.3: The vertical directional derivative approaches infinity as the top angle approaches 180°.

The second problem concerning large angles is errors in the derivatives of the solution. In many FEM applications, the main subject of interest is not the solution itself, but derivatives thereof. This applies to e.g. elasticity problems. Consider fig. 1.3: The values at the nodes in the figure represent the discrete solution at those points. Now, assume that the solution along the lower edge is estimated by linear interpolation, and that the interpolated values depend on the values of the lower nodes only. It is now trivial to see that as the angle at the top node approaches 180°, the directional derivative in the vertical direction becomes arbitrarily large.

Of course, for triangular and quadrilateral elements, large and small angles are somewhat intertwined; an element with a small angle usually also has a large angle and vice versa.

On the basis of this information, we understand that the best solution possible is a mesh that has few poorly shaped elements and no inverted elements.

For triangle meshes, it can be shown that the Delaunay triangulation (DT) in some sense yields optimal results: Of all possible triangulations of a set of points, the DT has the largest minimal angle, the smallest maximal circumcircle, and smallest maximal min-containment circle [19]. (The min-containment circle of a triangle is the smallest circle that contains it.) DTs have been the subject of decades of research, and is by now very well understood. Several classes of algorithms for generating DTs have been developed. A good introductory, to-the-point example of a theoretically optimal DT algorithm is [17].

For quadrilateral meshes, the case is somewhat less clear. The Q-Morph algorithm may be seen as an effort in the direction of developing an optimal quad mesher. However, due to its employment of heuristics, it can not be guaranteed that the resulting mesh is *the* best mesh out of all possible quad meshes, albeit probably pretty close.

It follows from the preference to equiangular elements, that the *node valence* (that is, the number of edges incident with the node[3]) is 4 for optimal quad meshes. Thus, one possible approach for generating an optimal mesh, would be trying to maximize the number of 4-valent nodes.

Another approach is attempting to maximize the distortion metric of the elements by smoothing the nodal positions. See section 2.2.6 for details on the method used in Q-Morph.

[3]See the glossary on page 62 for a more precise definition.

Now, to measure and compare the quality of entire meshes of elements, it is customary to consider both the average metrics and the minimum metrics of the meshes. In [4] it is claimed that analyses are more affected by minimum metrics than by low average metrics. A similar claim is also found in [20], where the papers [1, 7] are referred to on this subject.

In [13], mesh quality is measured with a formula based on the geometric mean of the metric, rather than the average metric.

Chapter 2

The algorithm

2.1 What it does

Q-Morph is an indirect algorithm that takes a triangle mesh as input, and generates an unstructured, almost all-quadrilateral mesh containing at most one singular triangle, and few irregular nodes. An advancing front sweeps through the triangles, selecting which triangles to be transformed next. The authors claim that it has the following desirable features:

1. It is *boundary sensitive*, in that it will generate a mesh with contours that closely follows the contours of the boundary.
2. It is *orientation insensitive* in that it will generate a mesh with the same topology for any two triangle meshes with equal topologies.
3. It generates a mesh with *few irregular nodes*.
4. It is an indirect method, and thus it avoids many expensive intersection calculations and poor element quality resulting from colliding fronts. This is because it can, contrary to direct methods, utilize topological information in the existing triangle mesh.

The algorithm is developed by Steve J. Owen, Matthew L. Staten, Scott A. Canann, and Sunil Saigal. Their implementation of the algorithm is part of a commercial release of ANSYS.

The algorithm promises to generate an all-quadrilateral mesh provided that the input geometry has boundaries consisting of an even number of nodes. If the boundary has an odd number of nodes, then the algorithm will have

to generate one singular triangle, usually somewhere towards the interior of the mesh.

2.2 How it is done

There is not much point in rephrasing the paper by Owen et al [16], but for the sake of completeness, a short-version of the main algorithm is outlined in table 2.1 on page 14. Some of the steps found there are elaborated in greater detail in the next paragraphs.

2.2.1 Constructing the initial triangle mesh

The initial triangle mesh should be some kind of regular triangulation[1], or a regular triangulation with holes. Any meshing method goes that meets those requirements. Note however that the local sizing of the final quad mesh roughly will follow that of the triangle mesh.

2.2.2 Edge state

The state of a front edge is determined by the angles between it and the two adjacent edges on the front. There are four different states: 0-0, 0-1, 1-0, and 1-1. If the angle at one of the nodes is less than $3\pi/4$, then that node bit is set to 1, else it is 0. So let us say that the left node bit is 1 and the right node bit is 0. The state of this front edge is then 1-0.

2.2.3 Edge level and front loops

Each edge of the initial front belongs to level 0. If there are no holes in the triangulation, then all the level 0 edges also belong to the same *loop*. A loop is a consecutive ring of edges that, in any given instant during the process, constitutes or is part of the current front. The current front may consist of one or more such loops, depending on whether the triangulation has holes in it. When holes are present, the front will advance both from the hull of the triangulation and from the loops around its holes.

Level 1 front edges are those edges that replace level 0 edges as the front advances. Level 2 edges replace level 1 edges and so on.

[1] See page 62 for a definition of a regular triangulation.

Create the initial triangle mesh
Define the initial front
Using the triangle mesh, it is straightforward to define the initial front: Any edge adjacent to only one triangle becomes part of the front.
Classify front edges
Front edges are classified according to *state* (see 2.2.2).
Front edge processing
An edge is selected to form the base of the quad to be built. The selection procedure considers three attributes: 1) state, 2) level (see 2.2.3), and sometimes 3) length. Length is only considered when large transitions are required. Next, the selected edge is given the following treatment:

- **Check for special cases**
  Very small angles or large transitions local to the current front are given special treatment.
- **Side edge definition**
  Choose from existing edges, or create new ones, depending on angles etc.
- **Top edge recovery**
  Delete edges intersecting a line between the top nodes of the side edges, and create a new edge that connects these nodes.
- **Quadrilateral formation**
  Delete all elements, nodes, and edges inside the new quad.
- **Local smoothing**
  Smooth the positions of the four nodes of the quad, and all the nodes that are adjacent to these (that is, connected with one edge).
- **Local front reclassification**
  Affected front edges need to be reclassified after smoothing and other operations on the topology.

Topological cleanup
This step tries to optimize all the node valences.
Global smoothing
Laplacian and optimization-based smoothing is performed in order to improve element quality even further.

Table 2.1: Overview of the Q-Morph algorithm

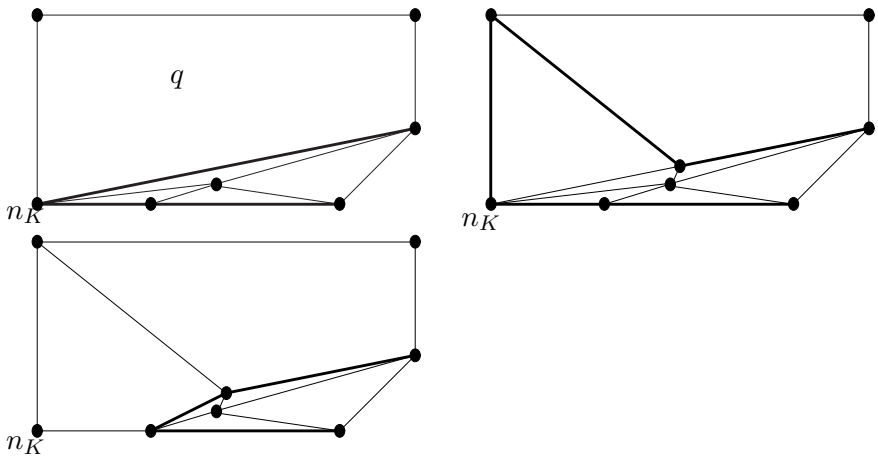


Figure 2.1: The transition seam operation.

2.2.4 Special cases

Special cases are subjected to one of the following operations: Seaming, transition seaming or transition splitting.

The values of the constants ϵ_1 and ϵ_2 are used when deciding whether one of the two seaming operations is to be performed. A seam or transition seam operation can be called whenever one of the following criteria is fulfilled:

1. The angle between two adjacent front edges is less than ϵ_1 and the number of quads adjacent to their common node is greater than 4.
2. The angle between two adjacent front edges is less than ϵ_2.

Note that $\epsilon_1 < \epsilon_2$.

If the transition from one of these fronts to the other is large (the length ratio is larger than 2.5), then instead a transition seam is performed. If the transition is large, but the angle is not small enough, then the transition split operation is called.

In the following, n_K denotes the common node of these two front edges. q denotes the quad adjacent the split edge.

A transition seam operation splits the longer of the two front edges at the midpoint. A new edge connects the node at the split to the vertex node that is connected to n_K. This effectively divides q into a quad and a

triangle. The triangle that was opposite q at the longer front edge, is also divided at the split, creating two new triangles. Then follows a recovery of the edge between the node at the split and the node opposite n_K on the shorter of the two front edges. The operation is completed by local smoothing. The transition seam operation is illustrated in figure 2.1.

Note that if the largest angle of q were greater than 180°, and located at node n_K, then the procedure for transition seam would fail, as it would create a non-conformal mesh. There is not given any solution for this case in the algorithm.

Also the transition split operation splits the longer of the two front edges at the midpoint. A new node is inserted at the centroid of q. The quad is divided into two quads and a triangle. Lastly, local smoothing is performed. The transition split operation can be seen in the two first frames of figure 3.3.

2.2.5 Topological cleanup

A crucial step in the Q-Morph algorithm is topological cleanup. Here, the number of irregular nodes is reduced by identifying and improving cases (which are identified among others by their valence patterns[2],) through a series of local operations. This yields a mesh with internal contours that more closely follow the contours of the boundary.

Owen's paper [16] does not treat this subject in any detail, but refers to three other papers. It is not clear which of these that has been used for Owen's implementation. I decided to use the paper by Kinney [9] for my own implementation, although I did not do a complete implementation of his algorithm.

Now, let us take a closer look at the particular steps that were in fact implemented:

In the first action required, the elimination of all chevrons[3], each chevron is removed along with a neighbouring quad. The hole is filled with a replacement pattern. This is one of the combine with neighbour operations.

Then follow three passes over the main cleanup processes. These are:

[2]See the appendix on page 62 for an explanation of the term valence pattern.

[3]In the paper by Kinney[9] a chevron is defined as a quad in which one angle is greater than 200°. This convention is also used here.

1. **Connectivity cleanup** - The mesh is scanned for cases, which are identified among others by their valence patterns. When a match is made, the pattern found is modified so that it becomes the appropriate replacement pattern.

2. **Boundary cleanup** - This process is in principle similar to connectivity cleanup, but with the twist that the cases are located on the boundary. Furthermore, efforts are made to remove boundary diamonds and triangularly shaped quads from the boundary.

3. **Shape cleanup** - Quads with large angles, chevrons and bowties are corrected in combine with neighbour operations.

After each of the cleanups, the mesh should be smoothed. Local smoothing should also be applied after each individual action within the three cleanups.

2.2.6 Global smoothing and distortion metrics

For global smoothing, Owen's paper [16] refers to an algorithm developed by Canann, Tristano, and Staten [4]. Here, *Constrained Laplacian smoothing* (CLS) is used in combination with *optimization-based smoothing* (OBS). The particular CLS algorithm described uses basic Laplacian smoothing to find a new position for each node. If the new position is not acceptable according to some certain criteria, a more moderate move is attempted. Among others, the acceptance criteria is based on *distortion metrics*, and moves resulting in inverted elements are denied. The distortion metric used here are explained in detail later in this section.

OBS is particularly expensive in terms of CPU time, so CLS is used in the first iterations. Each node that has not yet been moved to an acceptable position will be smoothed by OBS. OBS aims at increasing the minimum metric of the elements adjacent to each node. This is accomplished by means of a method named steepest descent.

In short, the steepest descent method regards the distortion metric of an element i, μ_i, as a function of an adjacent node's position, $\mathbf{x}$. This way, a higher value for the distortion metric can be found by adding some vector to the position. Because the values of μ_i for non-inverted elements generally varies between -1 and 1, the direction of this vector, the gradient vector $\vec{g_i}$, can be found by perturbing x_i by δ in each of of the component directions, and measuring the distortion metric for each direction: $\mu^+_{i,x}$, $\mu^+_{i,y}$ and alternatively $\mu^+_{i,z}$. The components of $\vec{g_i}$ are now a function of the new

and old values of the distortion metric: $g_{i,x} = (\mu^+_{i,x} - \mu_i)/\delta$ and similarly for $g_{i,y}$ and $g_{i,z}$.

Of all the vectors $\vec{g_i}$ obtained from the elements adjacent to the node, the $\vec{g}$-value of the element with the smallest initial distortion metric, μ_{min}, is selected as this node's gradient vector $\vec{g}$. $\vec{g}$ and the other $\vec{g_i}$'s are used in the calculation of a value γ, so that the new position of the node is $\mathbf{x} + \gamma\vec{g}$.

The metrics used can be considered as an enhancement of that used in Lee and Lo's algorithm [12]. Q-Morph employs the following metrics for triangles:

$$\alpha(ABC) = I \cdot 2\sqrt{3} \cdot \frac{\vec{CA} \times \vec{CB}}{CA^2 + AB^2 + BC^2} \tag{2.1}$$

where the factor I indicates inversion of the triangle:

$$I = \begin{cases} 1 & \text{if } (\vec{CA} \times \vec{CB}) \cdot \vec{N}_s > 0 \\ -1 & \text{if } (\vec{CA} \times \vec{CB}) \cdot \vec{N}_s < 0 \end{cases}$$

Here, $\vec{N}_s$ is the surface normal evaluated at the center of the triangle, and the other quantities are (3D) vectors between the triangle vertices, and their corresponding lengths.

When considering quadrilaterals, they are divided into four triangles along each of their two diagonals. Then the distortion metric for each triangle, α_i, is computed. The distortion metric for the quadrilateral is then defined as:

$$\beta = \{min(\alpha_1, \alpha_2, \alpha_3, \alpha_4)\} - negval \tag{2.2}$$

where:

$$negval = \begin{cases} 1 & \text{if any of the corner angles of the quad are } < 6^\circ, \\ & \text{any two of the nodes are coincident within a tolerance,} \\ & \text{or two of the triangles are inverted} \\ 2 & \text{if three of the triangles are inverted} \\ 3 & \text{if all four triangles are inverted} \\ 0 & \text{otherwise} \end{cases}$$

Note that the negval quantity used here has merely a heuristic foundation.

Chapter 3

The implementation

3.1 Limitations to the original algorithm

Due to time constraints, I could not implement the full Q-Morph algorithm. My implementation is restricted to 2D domains.

Also, the triangulation method is quite crudely implemented: It has no means for defining holes, and it does not support constraints. That is, the triangulator will generate a triangle mesh with a convex boundary and without holes. And finally, the triangulator does not generate any internal nodes. Although some internal nodes may be inserted (and removed) later as part of seaming, transition seaming and splitting, and topological cleanup, the most reliable way to introduce internal nodes, is to supply them along with the boundary points.

I should also mention that there is no support for constraints such as hardpoints and hardlines in any part of the implementation. However, neither is there any mention of hardpoints and hardlines in the Q-Morph paper [16].

As mentioned briefly in the previous chapter, the implementation of the CleanUp algorithm is not complete. There are several reasons to why I was satisfied with, or rather had to content myself with, an incomplete implementation: 1) Q-Morph does not provide parameters like the size function that is required in the size cleanup step of Kinney's algorithm. 2) Some of the steps in Kinney's algorithm are not needed. The reasons for this are that we know that the previous steps of Q-Morph have generated a conformal mesh, and the CleanUp implementation will also try to maintain a conformal mesh at all times. 3) Not all details of Kinney's algorithm are

revealed in his paper.

In this matter there is some comfort in reading the quotation [3] found in the beginning of Kinney's paper:

> A cleanup implementation does not have to be complete to be useful. Even if only a few of the cases are implemented, the mesh will be better than it was before. The implementer can concentrate on the cases determined to be most important, saving additional cases for later.

3.2 Choosing a suitable programming language

Initially being most familiar with `C` and `C++`, I first considered these languages. `C` is a widespread language with compilers for almost any computer, and `C` compilers also generate very fast code. However, the lack of OO[1] support makes it difficult to write readable code when working with very large and complicated algorithms. With a yet somewhat vague idea that this was actually a pretty large and nasty algorithm, I decided that the `C` language did not qualify.

`C++` on the other hand, has both OO support and many of the pros of `C`, like fast code. The syntax is also very similar, and `C++` is, as the name suggests, an extended `C` language.

However, from my experience with `C` and `C++`, it seems that one spends a lot of time debugging, at least compared to some newer programming languages, like Java. I am not quite sure why this is so, but I suspect that it is related to the way pointers are implemented in `C` and `C++`.

One might also add that sometimes the compact syntax of `C` and `C++` source code degrades the readability.

It is also worth mentioning Borland's Delphi and the Linux version, Kylix. The language is based on traditional Pascal, but extended with OO support. The compilers generate native code, and according to Borland, source code written for Kylix can be recompiled with Delphi, and vice versa, without or almost without any modifications. Unfortunately, the GPL version of Kylix was not yet available upon initiation of this project. Delphi, running on MS Windows OSes, was ruled out due to personal convictions and preferences regarding OSes.

[1] Object-Oriented

I had been told many wonderful things about Java, and I was also hoping that there would be some useful Java classes for mathematics and geometry that I could utilize in this project. Java has not got the pointer problem which sometimes troubled me as a `C++` programmer. The syntax is not quite as compact as that of `C++`, so the code is easier to read. Apart from that, the syntax is pretty similar. If one understands `C++` code, it is easy to learn Java.

One contest that Java certainly seems to loose, is code speed. The usual way of compiling a Java program, is first compiling it into platform independent bytecode, which in turn must be executed by a virtual machine. The virtual machine does an on-the-fly translation of the bytecode into native code that it feeds to the processor. Of course, platform independent code will not run as fast as the fastest native code, although it allegedly can come pretty close.

Another drawback with using Java is that most scientific applications and libraries are written in `C`, `C++`, and `Fortran`, and interfacing between that code and mine is not straightforward. So, for interfacing and demonstration purposes, I have included some `C++` code that creates a Java virtual machine and runs the Java code. Section 3.3 has more on this.

A great advantage of Java over `C` and `C++`, is that graphics support is standard and incorporated as a part of the language itself. With `C` or `C++` one would have to get a multiplatform graphical library for this task, if the code was to run on more than one platform.

Now, after having spent quite some time writing Java code, I might add that I have gradually grown somewhat disappointed with the allegedly wonderful Java classes for mathematics and geometry. Three very essential methods that I needed for the Delaunay mesh generation class were not supported at all, or a proper implementation was missing. (See the methods `inCircle`, `inHalfplane`, and `intersectsAt` in section 3.5.1.) Thus, I had to write my own code for these methods. A somewhat tedious, but still very instructive task.

On the other hand, I have become quite found of the informative runtime error messages generated by Java code crashes: The line number of the code that triggered the crash, and a list of calls leading up to the fatal line. This way faulty code is instantly located, so that valuable time can be spent on correcting the actual error, rather than scanning thousands of lines with no clues whatsoever to what caused the crash. This feature is also supported in Delphi/Kylix and by `C++` debugger applications.

3.3 Interfacing with Java from C++

Java comes with an interface, the Java Native Interface (JNI), for interfacing with some native code: `C`, `C++`, and `Fortran`. This means that Java code can be called from native code, and native code can be called from Java code. Moreover, native code can manipulate data fields in Java classes and so forth. More information can be found on the web[8]. The source code that was developed as part of this thesis has got some `C++` source code for starting up the Java implementation, i.e. the MeshDitor application.

3.4 Program code organization

Of course, all of the code is divided into classes, since that is the way to do it in Java. Some classes represent the different parts of a mesh, such as nodes, edges, triangles, and quadrilaterals. Others are devoted to the specific meshers. The following lists each class with a comment explaining its purpose:

`class AboutDialog:` A class which opens an "about" dialog window.

`class Constants:` This class holds the program "constants". That is, they are given as parameters to the Q-Morph implementation.

`class Dart:` A very simple implementation of a dart.

`class DelaunayMeshGen:` This class offers methods for incrementally constructing Delaunay triangle meshes.

`class Edge:` This class has methods and fields for edges.

`class Element:` This class declares methods and fields that are common to quads and triangles.

`class ExportToLaTeXOptionsDialog:` This class supports exporting of meshes to LaTeX format. Make sure you include include both packages epic and eepic in the header of your LaTeX document.

`class GCanvas:` The Canvas class which paints the background grid, the nodes, the edges etc.

`class GControls:` The Panel class with step button, zoom menu, and axes and grid toggle buttons etc.

`class GUI`: This class implements the graphical user interface.

`class GeomBasics`: This is a basic geometry class with methods for reading and writing meshes, sorting Node lists, printing lists, topology inspection, etc.

`class GlobalSmooth`: This class is an implementation of the algorithm described in [4].

`class HelpDialog`: A class which opens a help dialog window.

`class MeshDitor`: This is the executable class. It has methods for outputting version and help information, and for processing user command line options.

`class Msg`: This class outputs messages to the user.

`class MyFilterOutputStream`: This is a class for capturing Java error messages.

`class MyLine`: This class has methods and fields for for lines. The purpose of this class is solely to determine the intersection point between two lines. The length of a line is, of course, infinite.

`class MyVector`: This class holds information for vectors, and has methods for dealing with vector-related issues.

`class Node`: This class holds information for nodes, and has methods for the management of issues regarding nodes.

`class QMorph`: This is the main class, implementing the triangle to quad conversion process.

`class QMorphOptionsDialog`: A class for an options dialog for supplying parameters to the Q-Morph implementation.

`class Quad`: A class holding information for quadrilaterals, and with methods for the handling of issues regarding quads.

`class Ray`: This class holds information for rays, and has methods for dealing with ray-related issues. The purpose of this class is solely to determine the intersection point between a ray (origin and direction) and a vector (origin and x,y giving the direction, the length of the ray is considered to be infinite).

`class TopoCleanup`: This class constitutes a simple implementation of the cleanup process as outlined by Paul Kinney in [9].

`class Triangle`: A class holding information for triangles, and with methods for the handling of issues regarding triangles.

3.5 Problems, strategies and solutions

As I gradually became aware of the immensity of the task ahead, I tried to work out a good solution strategy. Firstly, I found that a reasonable way to organize the work, would be a divide and conquer-strategy: divide the work into bulks of lesser proportions, and solve each task separately. Next, it seemed that a good choice of such bulks would be the main steps of the algorithm, with each step having one or more dedicated Java methods.

Of course, initially I spent quite some time trying to get a good overview of the algorithm, so that I early on could spot potential problems related to different approaches. Still, I could not possibly foresee every little obstacle on the way.

On the occasions of facing seemingly unsolvable problems, poorly explained or simply undefined steps, I often resorted to a strategy of temporarily evasion. It seemed entirely possible, often even probable, that a solution sooner or later would reveal itself from the context of its surroundings. Or possibly, when the problem at hand was choosing between several seemingly equivalent approaches, I could implement each one, and then let the actual code performance help me select the best.

Considering that I did have the algorithms with papers dedicated to explaining them, it might seem that the implementation now would be fairly straightforward. Naturally, any such misconceived ideas that might initially have deluded me, were quickly exposed as the work progressed. Although several of the algorithm steps are indeed explained in great detail, others remained utterly in the dark. This section is devoted to describing these problems and the corresponding solutions applied in the implementation.

3.5.1 Constructing the initial triangle mesh

The specific approach used here is an incremental Delaunay algorithm, but as stated in the previous chapter, any triangle meshing strategy works, provided that it generates a regular mesh with or without holes. In the first step, the program creates an initial mesh consisting of only the two triangles spanned by the four nodes that are farthest away in each direction from the center of the node set. The other nodes are then inserted one at a time, and after each insertion, the mesh is updated to remain Delaunay compliant. When a node exterior to the current triangulation is encountered, the program first determines the influence region of that

particular node. The mesh in this region is then retriangulated by simply deleting all the triangles, and then connecting each of the nodes in the influence polygon to the inserted node.

However, please note that the implemented mesher has no support for defining holes in the domain. Still, in addition to node sets, the program can receive triangle meshes as input. This way, triangle meshes with holes can still be Q-Morphed.

Although I do not consider the triangle mesher to be a part of the core Q-Morph algorithm, I still feel that a further elaboration of this topic is justified by some non-trivial issues introduced by the implemented triangle mesher. Also, the surprisingly vast amount of time that I spent working on the triangle mesher speaks for a further elaboration. Thus, I have authored some brief paragraphs on these topics:

The `inHalfplane` method

The purpose of this method is to determine whether a point, $p = (x_p, y_p)$, is located to the left, to the right, or on a line defined by the two points (x_1, y_1) and (x_2, y_2). In the literature this method is often referred to as the orientation method. This task can seemingly easily be solved by evaluating the determinant

$$\begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_p & y_p & 1 \end{pmatrix}$$

If the determinant evaluates to a negative number, then the point is located to the left of the line, whereas a positive number indicates a location to the right of the line, and zero is the number we get for points located on the line itself.

However, because of the limited precision of computers, round-off errors arise when this determinant is to be evaluated in the implementation. This would not be such a big deal if the points where not located in the vicinity of the line. It is only when a point is located close to the line that problems arise. After all, the only thing that we need to know is whether the value of the determinant is less than, equal to, or greater than zero. We need not know the exact value of the determinant, unless it is zero.

In the Java implementation, I have utilized the methods of the `java.math.BigDecimal` class to calculate the value of the determinant. Hopefully, this gives a value that is sufficiently close to the exact solution.

The unitNormalAt method

The need for this method arose during the development of another method, the `inBoundedPlane` method, which is used to test whether a node is located within a given bounded plane. `unitNormalAt` will create a unit normal at one of the endpoints of a given edge.

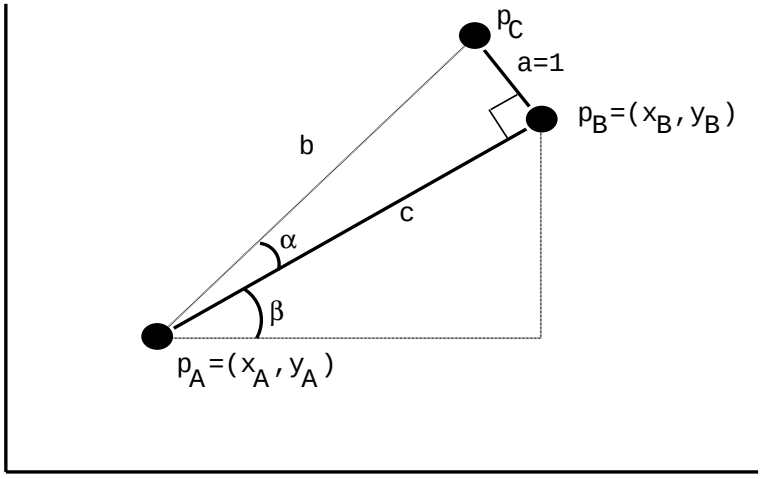


Figure 3.1: How can we find the point p_C?

Consider figure 3.1: With only the information regarding the location of the endpoints p_A and p_B, how can we find the point $p_C = (x_C, y_C)$, which is the endpoint of the unit normal at point p_B?

A pretty obvious answer is given by the following coordinates for p_C:

$$\begin{aligned} x_C &= x_A + b \cdot cos(\alpha + \beta) \\ y_C &= y_A + b \cdot sin(\alpha + \beta) \end{aligned} \tag{3.1}$$

However, I first need to know the values of α and β. To solve this problem, I employ some well known results from trigonometry:

$$cos(u + v) = cos(u) \cdot cos(v) - sin(u) \cdot sin(v)$$
and
$$sin(u + v) = sin(u) \cdot cos(v) + cos(u) \cdot sin(v)$$

Furthermore, instead of first finding the angles α and β, and then evaluating the sine and cosine of α and β directly, I found a much less computing intensive approach using standard trigonometry: $cos(\alpha) = \frac{c}{b}$, $cos(\beta) = \frac{x_B - x_A}{c}$, $sin(\alpha) = \frac{a}{b}$, and $sin(\beta) = \frac{y_B - y_A}{c}$.

Now, using this information in equation 3.1, and skipping some intermediate calculations, we arrive at the final answer:

$$x_C = x_B - \frac{y_B - y_A}{c}$$
$$y_C = y_B + \frac{x_B - x_A}{c}$$

The `inCircle` method

This method determines the location of a point relative to a circle: outside, inside, or on the circumcircle. Much like in the `inHalfPlane` method, this task could be solved using extended precision and evaluating a determinant. However, a faster, less accurate, but yet numerically stable approach has been developed [5].

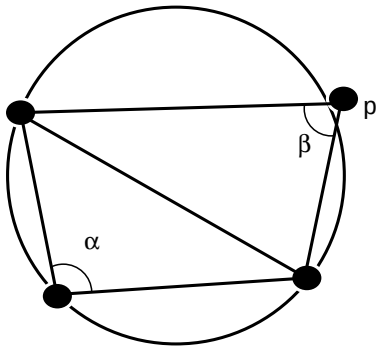


Figure 3.2: The `inCircle` method: is point p located inside, outside, or on the circumcircle?

As in the `unitNormalAt` method, the sine and cosine values of the two angles can be found without ever knowing the values of the angles themselves.

Now, if the cosine values of each angle are both less than 0, this implies that the angles are obtuse, and therefore the point must be inside the circumcircle. The opposite, that both cosine values are greater than 0, implies that the angles are acute, and that the point is outside the circumcircle.

If none of these cases match, the matter is settled by calculating $cos\alpha\, sin\beta + sin\alpha\, cos\beta$. If this value is less than 0, the point is inside the circumcircle, it is outside if the value is greater than zero, and on the circumcircle if the value is zero.

The intersection methods

Some steps of the algorithm require the calculation of points of intersection. This is the case in the particular triangulation method used, but also in several steps of the Q-Morph algorithm itself. Intersection calculations are expensive, so it is important to employ efficient methods for this.

I have developed several different intersection methods. Some just determine whether two line segments intersect, some also seek a point of intersection. The methods are all built on the same principles, which I have mainly taken from [6]:

Two line segments are either non-intersecting, intersecting in one singular point, or intersecting in an interval. The two line segments in question can be parameterized as $\vec{P_0} + s\vec{D_0}$ and $\vec{P_1} + t\vec{D_1}$, with $s, t \in [0, 1]$. Furthermore, we need the vector $\vec{\Delta} = \vec{P_1} - \vec{P_0}$, and we define $\vec{D_0} \times \vec{D_1} \equiv x_0 y_1 - x_1 y_0$.

We put the two line segments into an equation:

$$\begin{aligned} \vec{P_0} + s\vec{D_0} &= \vec{P_1} + t\vec{D_1} \\ s\vec{D_0} - t\vec{D_1} &= \vec{P_1} - \vec{P_0} \\ s\vec{D_0} - t\vec{D_1} &= \vec{\Delta} \end{aligned}$$

Because this is a vector equation in two variables, we get two scalar equations. We solve these for s and t, and get:

$$\begin{aligned} (\vec{D_0} \times \vec{D_1})s &= \vec{\Delta} \times \vec{D_1} \\ (\vec{D_0} \times \vec{D_1})t &= \vec{\Delta} \times \vec{D_0} \end{aligned} \tag{3.2}$$

Now, if $\vec{D_0} \times \vec{D_1} = 0$, then the two lines, to which the actual *line segments* belong, are parallel. A singular point of intersection is then only possible on the endpoints of the line segments (s and t are 0 or 1), assuming that the two *lines* are actually the same *line*. The equations 3.2 now reduce to one equation $0 = \vec{\Delta} \times \vec{D_0}$ (because $\vec{D_0}$ is a scalar multiple of $\vec{D_1}$). The lines are the same if and only if this equation holds. A singular point of intersection can now be sought at P_0 and $P_0 + \vec{D_0}$; the point being unveiled by either one of the following four equations that holds: $\vec{P_0} = \vec{P_1}$ or $\vec{P_0} = \vec{P_1} + \vec{D_1}$ or $\vec{P_0} + \vec{D_0} = \vec{P_1}$ or $\vec{P_0} + \vec{D_0} = \vec{P_1} + \vec{D_1}$.

On the other hand, if $\vec{D_0} \times \vec{D_1} \neq 0$, then the two *lines* intersect in a singular point somewhere, but we do not yet know if this also holds for the particular line segments in question. This can easily be answered by assuring that the values found for s and t are in the correct range:

$s, t \in [0, 1]$. Of course, the point of intersection is now given by $\vec{P_0} + s\vec{D_0}$ or $\vec{P_1} + t\vec{D_1}$.

The algorithm also requires some calculation of points of intersection between lines and between line segments and rays. It is straightforward to find these by trivial changes to the approach explained above.

3.5.2 Selecting the next front edge

It is a little unclear exactly in which cases the length of a front edge is to be considered more important than its state and level. In the end, I decided to use the following algorithm for selecting the next front edge:

1. *Find the first state list that contains a selectable edge. Start parsing the state lists in the state 1-1 list, then parse the state 1-0 and 0-1 lists, and finally look in the 0-0 list.*

2. *Parse this state list to find the front edge at the lowest level. If there are more than one edge at this level, select the shortest.*

3. *If the candidate edge is not in state 1-1, and the transition to one of the front neighbour edges is large, and this neighbouring front edge is marked as selectable and it is shorter than the candidate, then the neighbour is selected. Otherwise the candidate is selected.*

The reason for excluding state 1-1 edges in the last step, is that the implementation might otherwise fail irreversibly, plunging into a non-terminating loop. Consider the case in figure 3.3. Suppose that edge e_B is created and elevated to state 1-1 by a transition split operation[2], so that e_B in the next iteration of the main loop can form the base edge of a new quadrilateral. Unless edges in state 1-1 are specifically disregarded as candidates for the transition split operation, then we risk creating a situation in which e_B itself will be subjected to a transition split operation. A new edge e_C is created and elevated to state 1-1, and in the next iteration of the main loop, edge e_C is the victim of this reoccurring situation. The program will eventually crash in iteration N when the centroid of the quad adjacent edge e_N is placed outside of the quad due to limited precision.

[2]Note that the transition split operation is briefly described in 2.2.4.

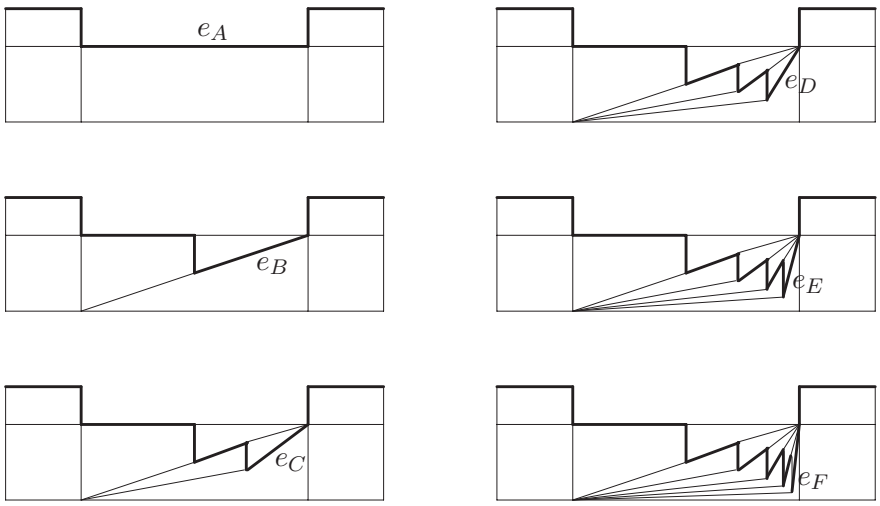


Figure 3.3: The code would fail unless special precautions are taken at the programming level: The transition split operation is called repeatedly, creating the pattern depicted above, and eventually the code would crash. (For simplicity, the initial triangles ahead of the front is not drawn in the figure.)

3.5.3 Recovering an edge

The algorithms for recovering an edge (algorithm 1 and 2 in [16]) will fail if the edge is intersecting a quad, and there are no clues as how to handle such a situation. I pounded over this problem for quite some time, considering extensions to the algorithms that would allow for the edge to intersect both triangles and quads.

In the end I decided that the only sensible thing to do, would be to abort the current quadrilateral formation process, temporarily mark the current base edge as unselectable, and initiate a new process with a different base edge. This solution seems to work fine, and I also believe this to be more in the spirit of the original algorithm.

3.5.4 Quadrilateral formation

When the base, side, and top edges have been found, the quadrilateral formation process is almost complete. However, the process may still fail. The reason for this is that the quadrilateral may contain holes, and we can

not allow the quad formation process to "eat" them. It would be difficult to detect the presence of holes at an earlier stage. If a hole is present, the process must abort, marking the current base edge as unselectable until another quadrilateral has successfully been formed. If no holes are found, then the process may continue as normal, deleting all triangles contained in the area inside of the four edges, and at last the new quadrilateral is formed.

3.5.5 Local smoothing

Although this step is explained in great detail, some issues are left undefined in [16]. These are:

- How to smooth a triangle. The implementation uses an approach similar to that used for quads.
- What to do with a node that is part of the front and at the same time belongs to more than two quads. For these cases, I chose to adjust the location of the nodes using isoparametric smoothing only, as proposed in [2].
- Consider employing length weighted Laplacian smoothing, as described in [2]. This is one of two options for smoothing nodes that are not part of the front. Now, for those of the contribution vectors pointing to a node on the boundary, one should adjust for angular smoothness. However, the algorithm for angular smoothness is defined for front nodes only. Naturally, it is trivial to extend this algorithm to also include boundary nodes.
- Another problem concerning angular smoothness arises in situations like the one illustrated in figure 3.4. The extension of vector P_{B2} will not intersect the line segment between N_{i-1} and N_{i+1}, as required by the algorithm.

  This issue is still unresolved in the implementation.

3.5.6 Constants

No clues are given to which values to assign to different constants, such as ϵ_1 and ϵ_2 for the seaming operations. Well-working values for the different constants are the topic of section 4.1.

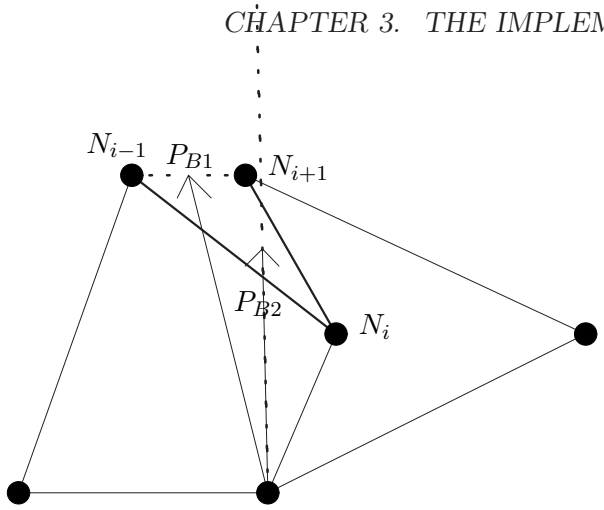


Figure 3.4: Problematic adjustment for angular smoothness

In swapping and splitting operations where N_m is on an opposing front, no indications are given to how large values one should allow for ϵ, or what to do if such a limit is exceeded. In the implementation all angles are allowed.

3.5.7 Intersection

See the paragraph on intersection in section 3.5.1.

3.5.8 Testing for clockwise ordering of vectors

I needed a method to determine whether an edge was cw[3] to another. More specifically, the method should return true if the angle between the calling edge and the argument edge was $\in [0, 180°)$.

Rather than some cunning technical approach, the implementation of this method consists of a series of if-sentences that has proven to work well. Firstly, the method finds the quadrants in which the vectors are located. This is done by simply considering the sign of the vector components. If the vectors are not in the same quadrant, and also not in opposite quadrants, it is straightforward to tell whether or not they are cw ordered. If in fact they *are* in the same quadrant, or in opposite quadrants, the question can be answered by comparing the slopes of the vectors.

[3]"cw" is an abbreviation for "clockwise".

3.5.9 Counter-clockwise ordering of edges incident with a node

The purpose of algorithm 2 in [16] is to determine which edges that are intersected by a line between the start and end nodes, N_c and N_d respectively. To accomplish this, it first creates a vector $\vec{V}_s$ from N_c to N_d, and an ordered set of ccw triangles and quads adjacent N_c.

We know that every node must be connected to at least two edges. Furthermore, if a two-edge node lies on the boundary, then the two connected edges also reside on the boundary. However, if N_c is a boundary node with only two edges connected to it, it would be difficult to decide on which side of them to look for the end node, N_d, by use of the methods and data structures found elsewhere in the code[4]. The following outlines the solution employed in the code.

Instead of directly creating the ordered set of ccw triangles and quads found around N_c, the implementation first creates a corresponding set of edges and a vector representation for each edge. Now, the following scheme is based on the method `isCWto(MyVector)` in class `MyVector` that returns true if the calling vector is clockwise to the argument.

If the node resides on the boundary, then it is assumed that there are two and only two boundary edges incident with the node. The boundary edge which is cw to its neighbour edge (according to the `isCWto` method) is selected to be the first in the list, unless the element between them has a concavity at the node, in which case the other boundary edge is placed first in the list. The subsequent edges in the list are added in cw order by parsing the elements around the node, starting at the (only) element adjacent to the first edge in the list.

Provided that there are only two edges in the list, which implies that they are boundary edges, both orderings of their vector representations are *in principle* be legal ccw orderings. So why is it not indifferent which of them get selected? Here we must carefully consider the needs of the other methods in the code, i.e. the implementation of algorithm 2.

Let us now assume that we use the `isCWto` method and the ordering described above. In the case of only two edges in the list, we must require that the most cw edge according to the `isCWto` method, comes first in the list. When the implementation of algorithm 2 parses this list, it will test

[4]However, if I had employed edges with *direction*, and all boundary edges were directed so that e.g. the outside of the mesh was to the right and the inside was to the left of the edge, then this would not have been a problem.

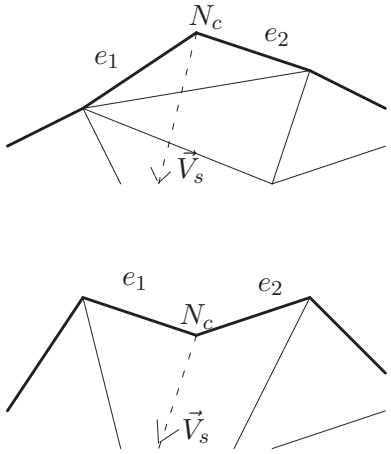


Figure 3.5: Top: The boundary edges at N_c form a convex boundary segment. e_1 is the most cw edge according to the `isCWto` method. Bottom: The boundary edges at N_c form a concave boundary segment. Their common element is a quad, or more precisely, a chevron. Of the two boundary edges, e_2 is the most cw one according to the `isCWto` method. However, the ordering of the edges must remain the same in both configurations: first edge e_1, then edge e_2.

each edge against $\vec{V_s}$ using the `isCWto` method. It will first test $\vec{V_s}$ against the first edge in the list, and then against the second edge, correctly detecting whether $\vec{V_s}$ does in fact intersect the triangle spanned (or quad partly spanned) by the two edges in the list. If the ordering in the list was reversed, then the loop would fail.

If the node does not reside on the boundary, then it does not matter which edge comes first in the list. A random edge is selected, and then the remaining edges are added to the list in ccw order by first finding the element which is ccw to the first edge (using the `isCWto` method), and then parsing all the elements around the node starting at the first edge and this element and continuing in the direction of the other edge that is also connected to the node and the same element.

3.6 Topological cleanup

Because of the somewhat undetailed nature of the paper describing topological cleanup[9], it was hard work trying to solve all the issues concerning the implementation of this algorithm. Eventually it still seems that I arrived at a decent solution, which is outlined below.

Note that illustrations of most of the different cases are found in [9].

3.6.1 Chevron elimination

Before a combine with neighbour operation is executed, the cleanup code attempts a much simpler solution: Unless the node at the greatest angle of the chevron is on the boundary, it can be relocated so that the chevron becomes an ordinary quad. A Laplacian smooth is used for this purpose.

If the Laplacian smooth fails to relocate the node so that the quad is no longer a chevron, or the node is on the boundary, then a suitable neighbour is found, both quads are removed, and the hole is filled with either fill_3 or fill_4. The difficulty here lies in selecting the most suitable neighbour.

As depicted in figure 3.6 (left), one can choose between two neighbours, q_1 and q_2. Relative to the node at the concavity, c, the first quad neighbour is located adjacent the edge between the next cw node from c in the chevron and the node opposite of c in the chevron. Conversely, the other neighbour is found adjacent the edge from c's opposite node and the next ccw node from c in the chevron.

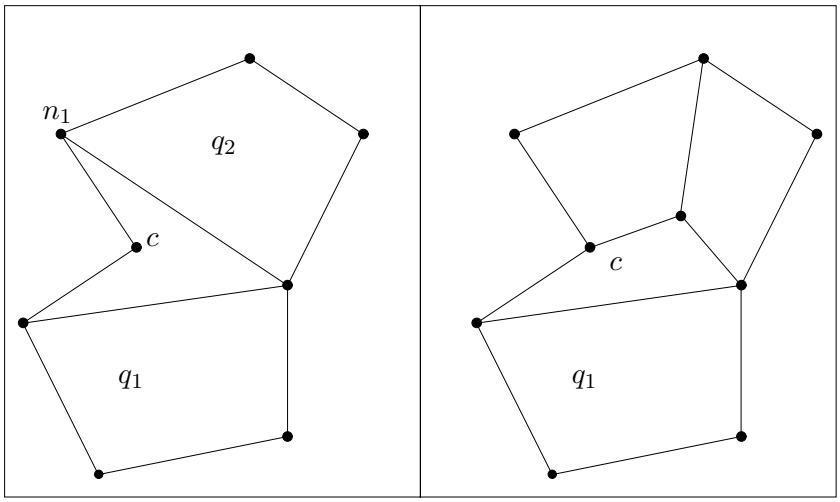


Figure 3.6: The elimination of a chevron by a combine with neighbour operation: fill_3.

The decision with which neighbour the chevron is to be combined, depends on the valence values resulting from each choice. The neighbour that gets chosen, is the one that, when combined with the chevron in a fill_3 operation, yields the valence values considered to be most optimal.

Then the fill_3 operation is performed as described in Kinney's paper: The two quads are removed, and the hole is filled with three new quads, as can be seen in fig.3.6 (right).

However, if the angles at node n_1 in the chevron and the selected neighbour in the figure sum up to more than 200°, then a fill_3 will not solve the problem, as it introduces another chevron. We must then use fill_4 instead. This situation is depicted in fig. 3.7.

3.6.2 Resolving cases by compositions

The code which I wrote for correcting cleanup cases, employed in both connectivity cleanup and partly in boundary cleanup, is perhaps *the* achievement with which I am most pleased in retrospect. Even so, it is probably not particularly original, and I would be surprised if it turned out that Kinney's implementation was essentially different in this respect.

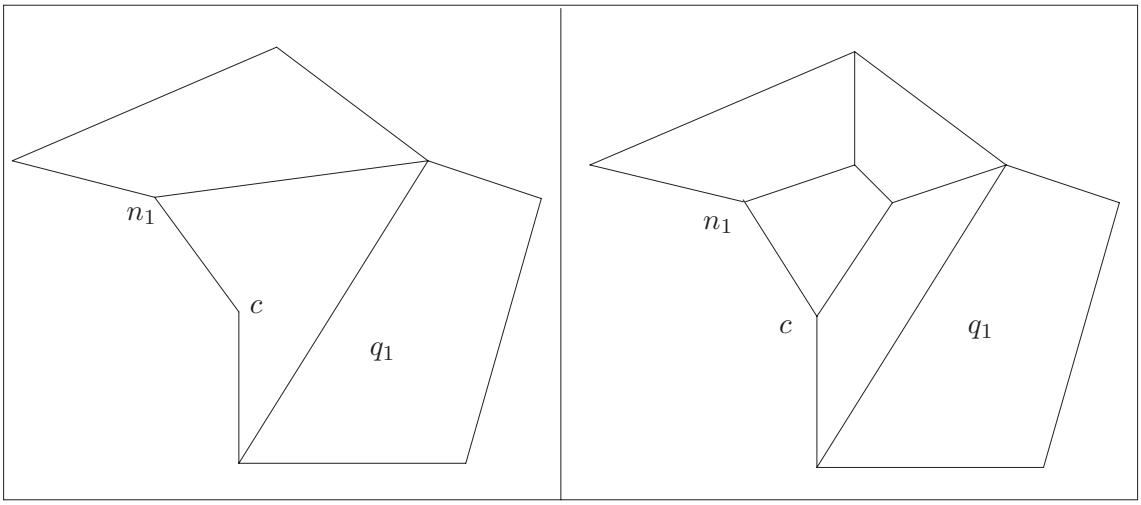


Figure 3.7: The elimination of a chevron by a combine with neighbour operation: fill_4.

Code	Description
3	Close the current quad, new pos of cur. node at the opposite node
4	Close the current quad, new pos of cur. node midway to oppos. node
5	Fill cur. quad and neighbour with fill_3
6	Fill cur. quad and neighbour with fill_4
7	Split cur. quad into two new quads along diagonal from cur. node
8	Switch cur. edge clockwise
9	Switch cur. edge counter-clockwise

Table 3.1: The mesh modification codes.

The approach employs the concepts of *darts*[5] and *α-iterators*[6].

I introduced 7 new codes in addition to the three α-iterators (0,1,2); codes which I have named *mesh modification codes*, that each signify a particular cleanup action to be performed at the current dart. (See table 3.1.) With appropriate *compositions* (or sequences) of these 10 codes, I was able to perform all the action routines described in Kinney's paper.

[5] A dart consists of one element (e.g. triangle or quad), one of its edges, and one of the nodes on the edge.

[6] α-iterators are operations performed on a dart for traversing the nodes, edges and elements of a mesh. Given a dart, d, the operation $\alpha_0(d)$ changes the current node to become the other node on the edge. $\alpha_1(d)$ changes the current edge to the neighbour edge at the current node inside the element. And finally, $\alpha_2(d)$ changes the current element to become the neighbour element at the current edge.

3.6.3 Connectivity cleanup

As noted in the previous chapter, the cases are identified among others by their valence patterns[7]. For the so-called standard cases in connectivity cleanup, a reliable identification must also take into account that such a case has 4 vertices, which are found among the nodes outlining the area of the mesh in question. If one of these nodes is to be regarded as a vertex, then the internal angle at the node must be smaller than the internal angle at every non-vertex node on the outline. A vertex pattern informs which nodes are to be regarded as vertices.

For some cases, an additional requirement is that certain nodes are internal nodes. Consequently, such cases must have a corresponding internal nodes pattern.

To resolve the cases, connectivity cleanup relies entirely on compositions. The complete list of implemented cleanup cases and their corresponding solutions, in terms of compositions, is found in the appendix on page 64.

3.6.4 Boundary cleanup

For reliable identification of boundary cases, the code must verify that certain nodes in the valence pattern are located on the boundary. Thus, each valence pattern should have a corresponding boundary pattern indicating which of the nodes that should be on the boundary.

Boundary cleanup now continues along the lines of connectivity cleanup. Those of the boundary cases that can be identified by their valence and boundary patterns, are listed in the appendix in table 3 on page 65.

After that, the code attempts to replace triangularly shaped quads having two boundary edges with the large angle at their common node. As illustrated in [9], the replacement depends on whether a one row transition or a two row transition can be used. To accomplish the one row transition, a fill_4 operation followed by a fill_3 operation will do the job. On the other hand, two consecutive fill_3 operations is needed for the two row transition. (Of course, the operations should be performed at some particular darts, and they should be followed by appropriate smoothing.)

In addition, attempts are made to remove boundary diamonds[8]. There is some hinting in [9] that not all boundary diamonds should be removed, but

[7] See the appendix on page 62 for an explanation of the term “valence pattern”.
[8] Boundary diamonds are quads with one and only one node on the boundary.

there is no good explanation to which ones should actually be removed. In my implementation, those boundary diamonds that conform to this inequality will be removed:

$$|4-(valence(n_a)+valence(n_b)-2)| \leq max(|4-valence(n_a)|, |4-valence(n_b)|)$$

where n_a and n_b are the two nodes in the diamond that are directly connected to the boundary node. Admittedly, the reason for choosing this particular criteria was more empirically based rather than based on any theoretical proof. What is actually compared here, is the deviation at the nodes n_a and n_b from the optimal node valence, 4. On the left side in the inequality we find the deviation of the node resulting from closing the diamond, and effectively merging n_a and n_b. On the right side is the larger of the deviations at n_a and n_b when the diamond remains open.

3.6.5 Shape cleanup

In shape cleanup, a total of four different cases are given in [9]. Firstly, there are the two cases with quads having angles greater than 160°. These cases are identified by simply comparing angles, and verifying factors like the number of incident edges at the central node, the existence of appropriate neighbour quads, and that certain nodes are on the boundary. The first case can be resolved by a fill_4 operation, and the second case is resolved by a split quad operation followed by two fill_3 operations. In the first case, the criteria for which neighbours to combine with, is based on the size of the angles both at the central node and others. More information can be found in the source code. The operations should be accompanied by appropriate smoothing.

Secondly, there are the chevron and bowtie cases. Chevrons are removed as described in section 3.6.1. Bowties are illegal quads that violates the topology of the mesh. I decided that the simplest way to deal with bowties was to avoid creating any altogether. Therefore efforts are made throughout the cleanup code to guarantee that bowties are never introduced into the mesh.

3.6.6 The full `CleanUp` implementation

To close this section, I give the pseudo-code for the CleaunUp implementation:

```
Chevron elimination
```

```
Loop 3 times:
{
  Connectivity cleanup
  Global smoothing
  Boundary cleanup
  Global smoothing
  Shape cleanup
  Global smoothing
}
```

Note that within each of the major cleanups, the mesh should be completely relieved of one particular case, before the next case is considered. The order in which cases are resolved is also important.

3.7 Global smoothing

The global smoothing algorithm requires the calculation of distortion metrics for each element. In this process, each quadrilateral is divided into four triangles along its two diagonals, and the distortion metric for the quadrilateral is computed according to equation 2.2. Now, this might not sound so difficult. However, I eventually learned that my first idea of how to divide the quadrilateral was erroneous.

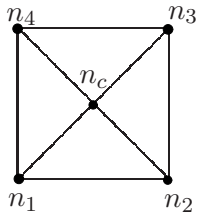


Figure 3.8: How I first thought a quad was to be partitioned.

I thought I understood perfectly well how this division was to be accomplished: We get four triangles defined by their vertices, see fig. 3.8: (n_1, n_2, n_c), (n_2, n_3, n_c), (n_3, n_4, n_c), and (n_4, n_1, n_c). Here, n_c is the point where the two diagonals intersect.

This did work to some extent, but for some quads, like chevrons and most inverted quads, the intersection point is located outside the quad. We then get triangles whose area is also located outside the quad. It might seem a little strange that the distortion metric of such artificial triangles should be considered in the computation of a distortion metric for the quad.

Furthermore, we might also imagine an inverted quad whose point n_c does not exist because the diagonals are parallel.

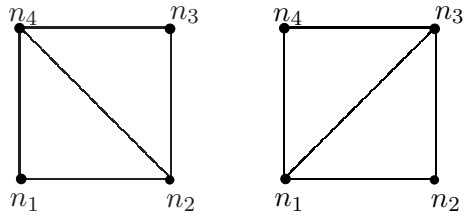


Figure 3.9: A better way to partition a quad.

Thus, my first conception of the partitioning for quads had to be discarded, and I then found the second conception, which I believe to be correct. We get the two first triangles by dividing the quadrilateral along the first diagonal, and then we get the two last ones by dividing along the second diagonal, see fig. 3.9.

In computing the quadrialteral distortion metrics, I found it necessary to use a slightly different formula for the four triangle distortion metrics: α_1, α_2, α_3, and α_4. As we do in fact not want equiangular triangles, but (90°, 45°, 45°) triangles so that two triangles form one square quadrilateral, the normalizing factor in formula 2.1 should now be 4 rather than $2\sqrt{3}$.

3.7.1 Detecting inverted elements

During the implementation of the global smoothing methods, I realized the need for robust and efficient metohds for detecting inversion of elements. These methods must work correctly for all possible situations, and yet they should not be too computing expensive.

We assume that each element is initially uninverted, but that it may become inverted at a later stage when the nodes are moved about.

For triangles, one can apply the method for calculating the "I" quantity that is briefly mentioned in section 2.2.6 and in [4]. But because the current implementation is restricted to 2D domains, there is no need to involve the surface normal. We can stick to 2D vectors, and the surface normal is quite unnecessary, as we shall see:

First note that I use the definition of a cross product for 2D vectors as given in section 3.5.1. Assuming that each triangle has a dedicated base edge and a top node opposite this base edge, we find the cross product of the two vectors originating from each of the endpoints of the base edge and

pointing to the top node. This cross product indicates which side of the base edge that the top node is located. Now, the triangle is inverted if the top node is no longer on the same side of the base edge as it was initially. That is, the triangle is inverted if the cross product becomes negative. However, the order of the factors matters because cross products are anticommutative: $\vec{a} \times \vec{b} = -(\vec{b} \times \vec{a})$. Thus, each triangle must have a field indicating the order of the factors.

For quadrilaterals, it was slightly harder to find methods for detecting inversion. After some time pondering, I came up with the following idea: With some minor extensions, we can apply the strategy for detecting inversion of triangles. By regarding each of the edges of the quad as the base edge of a triangle and each of the two opposite nodes as top nodes in triangles, inversion is detected if for one of the base edges, none of the opposite nodes is on the initial side.

This rather brute force strategy requires at most eight cross product calculations per quad. With a pen and some paper, and a little experimenting with the editor (see section 3.8), I found that this number can be reduced to four with the following optimization:

We check only two opposite edges in the quad, and demand that at least two cross products be negative to be certain that the quad is inverted. If three or all four cross products are positive, then the quad is not inverted. This means that a quad is inverted if more than one node is located on the "wrong" side of an opposite edge, or in other words, more than one interior angle is greater than 180°.

3.8 The interactive GUI

When working with any geometry code that reaches a certain level of complexity, I suppose the aid of some kind of interactive GUI for visualizing the actual geometry is a tremendous advantage over trying to work your way through hundreds and hundreds of debug messages. For this purpose I developed a very simple application, called MeshDitor.

Initially, the only features were visualization of meshes, manual construction of meshes by point and click, and loading and saving meshes. But as I grew increasingly tired of debugging aided only by debug messages and Java runtime error messages, I decided to add some more functionality.

With the click of menu buttons, the editor will execute the triangulation or Q-Morph code on the current mesh. Other convenient features included are

zooming and scrolling the mesh window, turning on and off debug messages, and a step mode that allows for executing the triangulation or quad-meshing methods one edge at a time.

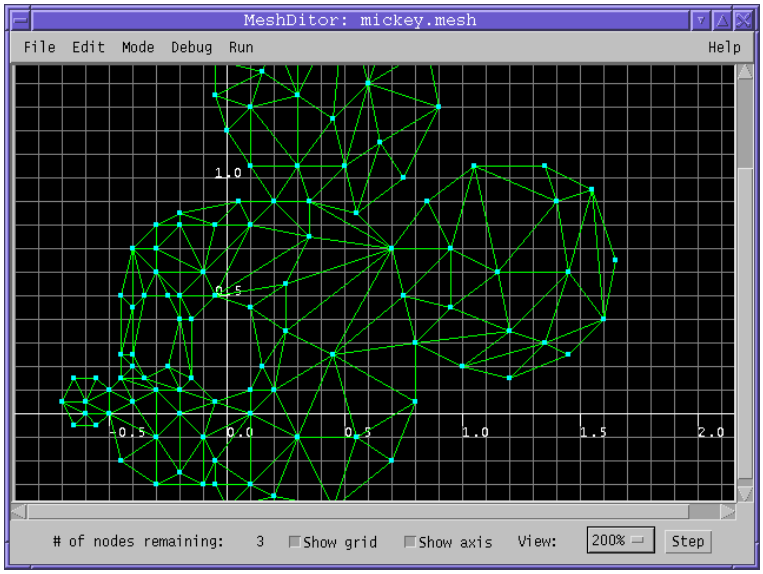


Figure 3.10: MeshDitor, the mesh editor with which I created most of the mesh cases.

Chapter 4

Results

4.1 Tuning the constant values

By running the Q-Morph code on several triangle meshes, and in each case trying out different values for the program constants, I have found, that the values given in table 4.1 seem to work pretty well. Still, I cannot guarantee that the values listed here will suite all implementations of the Q-Morph algorithm. Nevertheless, these values should at least give an idea to which range to look for optimal values.

Each of the constants are explained below.

4.1.1 The ϵ_1 and ϵ_2 constants

These are explained in section 2.2.4

4.1.2 The COINCTOL constant

This constant is used in the quadrilateral distortion metric to determine whether two of the quad's nodes are too close. Preferably, I think that the value of this constant should depend on the spacing of the boundary nodes. (I.e. it would not really be a constant at all.)

Imagine a scenario in which the value of COINCTOL is higher than the length of any of the boundary edges. In effect, the element quality of the quads at these edges would suffer. In my opinion, this is undesirable. Thus,

Constant	Value
ϵ_1	$0.04 \cdot \pi$
ϵ_2	$0.09 \cdot \pi$
COINCTOL	see section 4.1.2
MOVETOLERANCE	see section 4.1.3
OBSTOL	0.1
DELTAFACTOR	10^{-5}
MYMIN	0.05
θ_{max} (THETAMAX)	200°
TOL	10^{-5}
γ (GAMMA)	$[0.8, 0.9]$
MAXITER	∞ ?

Table 4.1: Well-working values for the program constants. Keep in mind that these are merely suggestions.

a better solution is perhaps that the value of COINCTOL becomes a constant factor times the length of the smallest boundary edge in the mesh.

4.1.3 The MOVETOLERANCE constant

A node movement that is proposed by the CLS[1] and whose length is less than the MOVETOLERANCE constant is denied. Ideally, and as with the previous constant, the value should be a constant factor times the length of some feature in the mesh rather than simply a constant factor.

4.1.4 The DELTAFACTOR constant

The δ value is used for perturbing the position of the nodes in the steepest descent method, which is briefly described in section 2.2.6. The value of δ is set to DELTAFACTOR·maxModDim, where maxModDim is the maximum model dimension[2].

4.1.5 The MYMIN constant

This should be a value in the range 0 to 1. (A value of 0.05 is proposed in [4].) In the CLS it is the minimum element quality which is considered to

[1] Constrained Laplacian Smoothing

[2] My interpretation of that is the length of the longest edge in the mesh.

be acceptable for a node move.

4.1.6 The θ_{max} (THETAMAX) constant

When performing the CLS on a node, θ_{max} is the maximum angle that is allowed for the angles in the surrounding elements.

4.1.7 The OBSTOL constant

In the global smoothing process, if the Constrained Laplacian Smoothing (CLS) has been performed twice on a particular node, and the smallest distortion metric of the elements surrounding the node is still less than OBSTOL, then the Optimization-Based Smoothing (OBS) should be performed.

4.1.8 The γ (GAMMA) constant

If the steepest descent method, briefly described in section 2.2.6, fails in finding a suitable γ value, then the program reverts to the GAMMA constant.

4.1.9 The TOL constant

In the final step of the OBS, the proposed move of the node is accepted provided that the lowest distortion metric, μ^+_{min}, measured for the surrounding elements is greater than or equal to the lowest distortion metric for the same elements before the move plus TOL:

$$\mu^+_{min} \geq \mu_{min} + TOL$$

4.1.10 The MAXITER constant

At least during the development of the implementation, it was sometimes the case that the global smoothing process did not terminate. Thus, I had to enforce a limit for the maximum number of iterations allowed. It may well be that it is now obsolete, as some serious flaws have been corrected recently.

4.1.11 Definition of a chevron

However, what seems to have the greatest impact on the element quality is the definition of the chevron, which is used by the CleanUp implementation. If the lower limit for the largest angle is set to 180° instead of 200°, this leads to an improvement in element quality in most cases. Especially the minimum distortion metric benefits from this. Practical results of the two different definitions can be studied by comparing the results in table 4.3.

Still, in what appears to be the minority of the cases, the 180° limit results in lower element quality. In some cases the program might simply fail due to new chevrons continually being introduced to the mesh during chevron elimination. 200° seems safe for the current implementation, and this is also the limit given in [9].

4.2 Robustness

The implemented code has been tested on a number of different mesh cases. Whenever a program crash or other fail situation occurred, the bug was located and corrected. It seems that the code now runs smoothly on all the tested mesh cases without crashing or generating inverted elements.

However, I cannot promise that the implementation now will run perfectly, by any means. The complexity of the program is far too great, and needless to say, it is impossible to test all possible cases. Bugs have been discovered and corrected up until the very last weeks before the deadline, and alas it is likely that more bugs will continue to surface also in the future. Still, there is some comfort in the fact that the number of bugs in a program is finite. As this number already has been dropping over quite some time, it is my hope that the number of bugs in the implementation is now so close to zero that the implementation actually can perform some useful tasks.

I am aware that the program may produce low-quality quad-meshes for some rare cases. Also, the user interface has some annoying flaws, and additional efforts are required before it can be brought to a satisfactory state.

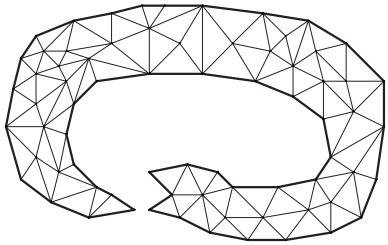


Figure 4.1: Model snake-t-1: A relatively high quality triangle mesh.

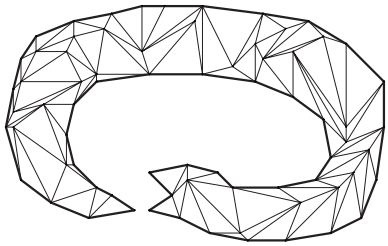


Figure 4.2: Model snake-t-2: The internal nodes have been move about in an effort to significantly reduce the mesh quality.

4.3 The impact of the triangle mesh on the result

Q-Morph puts no restrictions on the quality of the initial triangle mesh, promising to create an overall high quality quad mesh no matter what. I have conducted some simple experiments just to see if this is a reliable promise.

The first test investigates the "snake cases". In both of these, the boundary edges and boundary nodes remain fixed. The first snake case has triangles of relatively high quality, whereas in the other case the internal nodes have been moved about, so that most of the triangles have become severely distorted. See figures 4.1 and 4.2.

Considering the data for the snake cases in table 4.2, it seems that the distortion of triangles does not necessarily have a negative impact on the final element quality. The average element quality is actually a little higher for the quad mesh created from the distorted triangles (snake-q-2, see figure 4.4), than for the quad mesh created from the high quality triangle mesh (snake-q-1, see figure 4.3). The minimum element quality remains unchanged.

In the second experiment, the "ball cases", the boundary nodes and edges

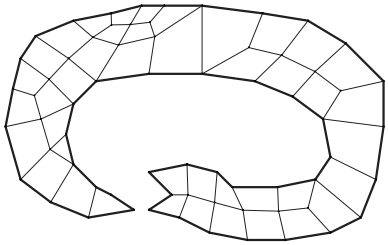


Figure 4.3: Model snake-q-1: The Q-Morphed version of snake-t-1.

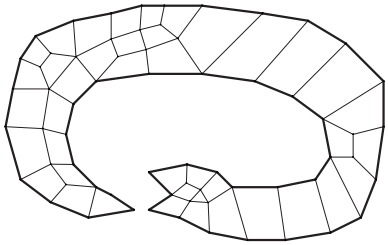


Figure 4.4: Model snake-q-2: The Q-Morphed version of snake-t-2.

are still fixed in both cases, and the internal nodes of the second case have been moved about, but here also additional internal nodes have been inserted. Consequently, the number of triangles is increased. See figure 4.5.

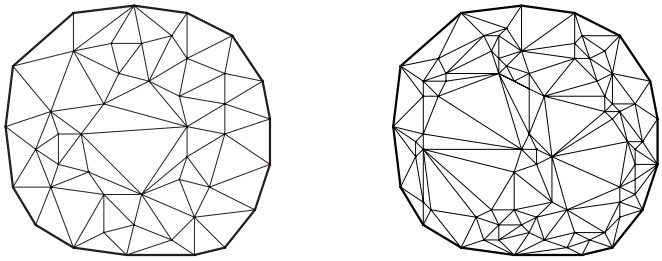


Figure 4.5: Models ball-t-1 (left) and ball-t-2 (right). ball-t-1 has a low number of internal nodes, whereas ball-t-2 has a higher number of internal nodes.

Looking at the ball case data in table 4.2, it appears that the increase in number of nodes does not have any negative impact on final element quality. On the contrary, the resulting element quality in the quad mesh is significantly higher when a high number of nodes is present in the triangle mesh. The resulting quad meshes can be seen in figures 4.6.

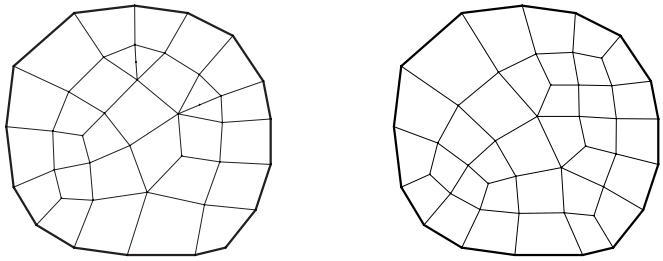


Figure 4.6: Models ball-q-1 (left) and ball-q-2. ball-q-1 is created from a triangle mesh with a low number of internal nodes, whereas ball-q-2 is created from a triangle mesh with a higher number of internal nodes.

4.4 Element quality: some statistics

Since my main priority was getting the code right, and my thesis is already long overdue, I was prevented from performing any proper study of element quality in the Q-Morphed meshes. This would require a significant number of mesh cases, and ideally the cases should also have relevance to real-world simulations. Nevertheless, during the program development, I created a moderate number of strictly artificial mesh cases, and in table 4.2 I have collected the statistics for the largest of these. I suspect that the low minimum metrics for most of the resulting quad meshes should be attributed to the inadequate number of cleanup cases. Of course, the average metric would also benefit from this.

Several other mesh cases were also used for testing purposes. Some of these are very small, i.e. they have very few nodes. Others have extremely difficult geometries. These cases are not representative for real-world domains, and thus they are not presented here.

	Before Q-Morph			After Q-Morph			
Model	**Num. tris**	**Min. metric**	**Avg. metric**	**Num. quads**	**Num. tris**	**Min. metric**	**Avg. metric**
snake-t-1	97	0.520	0.846	41	1	0.0769	0.635
snake-t-2	97	0.126	0.551	40	1	0.0769	0.670
ball-t-1	72	0.429	0.828	30	0	-0.00355	0.655
ball-t-2	150	0.0	0.697	35	0	0.519	0.766
concave	107	0.223	0.760	49	1	$-7.63 \cdot 10^{-17}$	0.595
difficult	206	0.371	0.810	95	0	-0.00456	0.495
donut	63	0.465	0.832	24	1	0.327	0.593
sqrhole	28	0.277	0.672	27	0	-1.11	0.192
mask	68	0.433	0.779	46	0	-1.03	0.344
onehole	41	0.611	0.864	31	1	-0.0102	0.382
cry-head	82	0.234	0.670	50	0	-0.0672	0.542
mickey	156	0.295	0.771	66	0	-0.121	0.565

Table 4.2: The statistics for the most relevant mesh cases. The constant values were $\epsilon_1 = 0.04 \cdot \pi$, $\epsilon_2 = 0.09 \cdot \pi$, COINCTOL=0.01, MOVETOLERANCE=0.01, OBSTOL=0.1, DELTAFACTOR=10^{-5}, MYMIN=0.05, $\theta_{max} = 200°$, TOL=10^{-5}, $\gamma = 0.8$ and MAXITER=5.

	200° limit				180° limit			
	Number of		Metric:		Number of		Metric:	
Model	**quads**	**tris**	**min.**	**avg.**	**quads**	**tris**	**min.**	**avg.**
snake-t-1	41	1	0.0769	0.635	40	1	0.0769	0.646
snake-t-2	40	1	0.0769	0.670	46	1	0.0769	0.644
ball-t-1	30	0	-0.00355	0.655	31	0	0.405	0.772
ball-t-2	35	0	0.519	0.766	-	-	-	-
concave	49	1	$-7.63 \cdot 10^{-17}$	0.595	50	1	0.150	0.618
difficult	95	0	-0.00456	0.495	99	0	0.0	0.491
donut	24	1	0.327	0.593	24	1	0.215	0.545
sqrhole	27	0	-1.11	0.192	22	0	0.106	0.338
mask	46	0	-1.03	0.344	68	0	$3.25 \cdot 10^{-12}$	0.373
onehole	31	1	-0.0102	0.382	50	1	-0.968	0.273
cry-head	50	0	-0.0672	0.542	57	0	0.0433	0.563
mickey	66	0	-0.121	0.565	68	0	0.160	0.577

Table 4.3: A comparison of the results obtained with the original definition of chevrons (left) and the results obtained with a new definition (right). The constant values are otherwise identical to those used in table 4.2. However, the program is no longer stable, as it breaks down during chevron elimination in model ball-t-2.

Chapter 5

Example problems

As I did not initially have any cases/models/data sets to test the program with, I had no option but to create my own. In this task I was aided by the editor functionality in MeshDitor. However, it is a tiresome task to construct large mesh cases by this method, and alas I have never tested the implementation on meshes of any significant size.

The triangle meshes were constructed for the sole purpose of serving as test cases for the implementation, i.e. they do not have a basis in real-world simulations.

5.1 Some general cases

Firstly, let us look at some snapshots of the Q-Morph process. In figure 5.1 the front advances through the triangle mesh, and one by one the quads are formed. In the final snapshot, topological cleanup and smoothing have been performed. One singular triangle remains in the mesh due to an odd number of boundary intervals.

The advantage of algorithms like Q-Morph is their ability to obtain a satisfactory result even on the most bizarrely shaped domains. In the next figure (5.2) we see my triangle mesh interpretation of the well-known Disney character, and the result after subjecting it to the Q-Morph implementation.

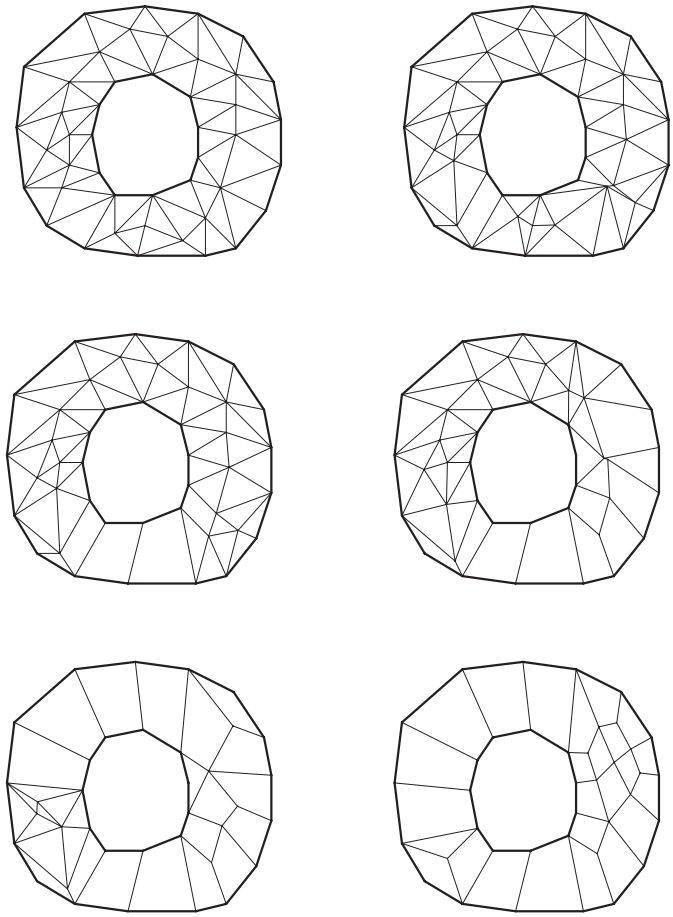


Figure 5.1: The donut case: A series of snapshots of the Q-Morph implementation at work in a triangle mesh.

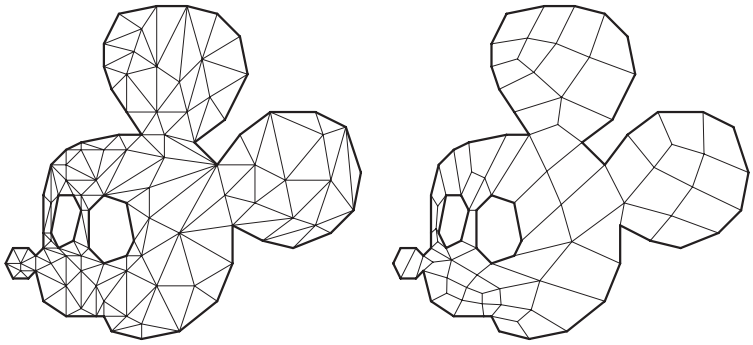


Figure 5.2: Models mickey-t (left) and mickey-q (right): The triangle meshed model and its Q-Morphed counterpart.

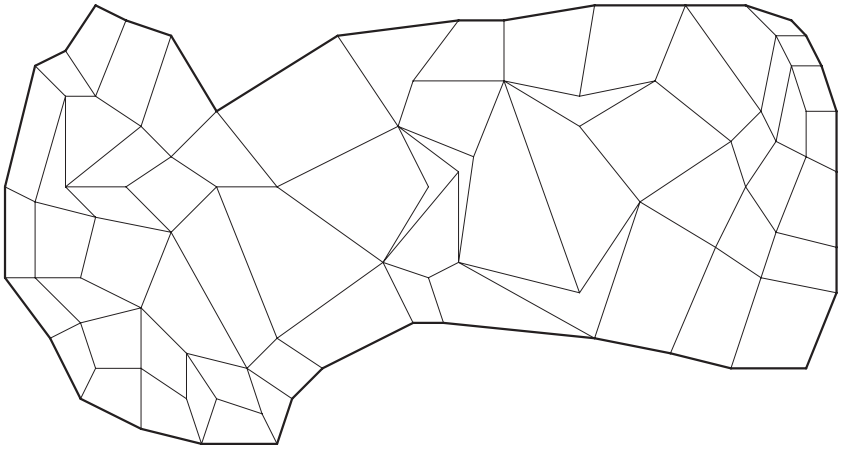


Figure 5.3: A quad mesh before topological cleanup.

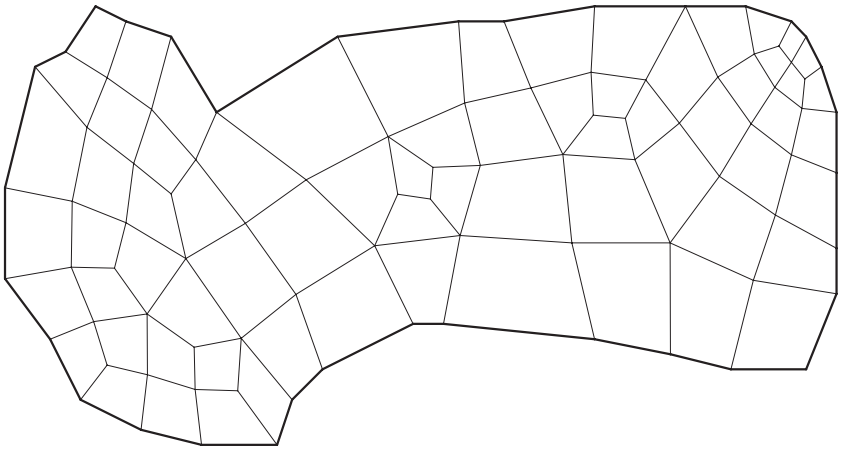


Figure 5.4: The quad mesh in figure 5.3 after topological cleanup.

5.2 Case illustrating topological cleanup

Topological cleanup can dramatically improve mesh quality. In figure 5.3, the minimal element quality is -0.800 and the average element quality is 0.455. The mesh has 64 quads, 1 2-valent node, 15 3-valent nodes, 53 4-valent nodes, 10 5-valent nodes and 3 6-valent nodes. After the CleanUp code has completed, see figure 5.4, the minimal element quality is 0.386 and the average element quality is 0.701. The mesh has now 68 quads, no 2-valent or 6-valent nodes, 11 3-valent nodes, 65 4-valent nodes and 10 5-valent nodes.

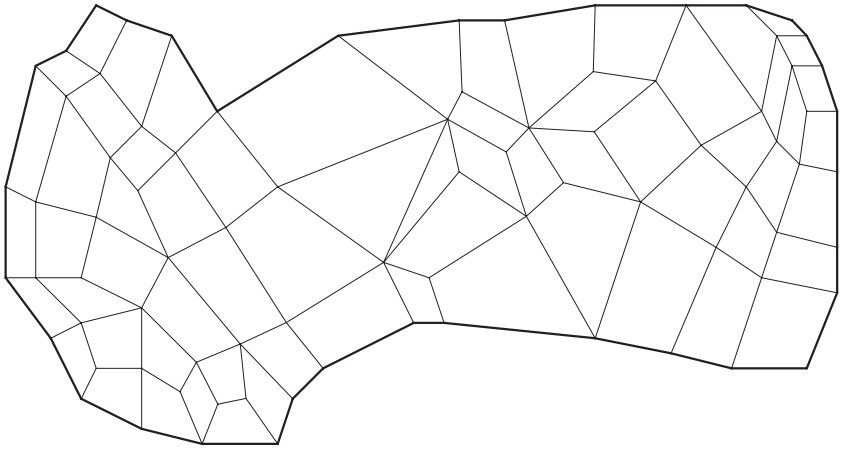


Figure 5.5: The quad mesh in figure 5.3 after optimization-based smoothing. Note that the triangularly shaped elements are actually degenerated quads.

5.3 Same case subjected to opt.-based smoothing

In the Q-Morph algorithm, optimization-based smoothing should be performed after topological cleanup. However, in the following example the optimization-based smoothing is executed on the mesh in figure 5.3 without a preceding topological cleanup. The resulting mesh is found in figure 5.5. Smoothing does not alter the valency of nodes, but element quality can still be improved. The minimal element quality is now $-5.93 \cdot 10^{-9}$, while the average element quality is 0.612.

Chapter 6

Improvements

As the main focus of the development has been on correctly implementing the algorithm rather than finding the optimal data structure and writing optimal code, there is certainly room for many improvements in this respect.

6.1 A better data structure

When I began the implementation process, I had no clear idea what demands the algorithm would force upon the data structure. I therefore chose one which was flexible and extensive. It became increasingly evident that a more optimized and carefully planned data structure would have spared me from a lot of hard work.

However, having based everything on this particular structure, it appeared to be a much too arduous task to convert the code to work with a different one. Thus, I was stuck with the initial data structure which is depicted in figure 6.1.

If I was to re-implement the algorithm, or if I was to offer advice to others in this respect, the data structure in figure 6.2 might be a starting-point.

The `Edge` class does not any longer need to worry about which of the nodes are to the left and to the right. Instead, each edge is now directed, having a start and end node. The `Node` class is kept just for convenience, so that the `x` and `y` fields can be transferred in a singular operation between methods. Operations on nodes should be carried out from the `Edge` class rather than from the `Node` class itself. E.g. to obtain the list of edges connected to a

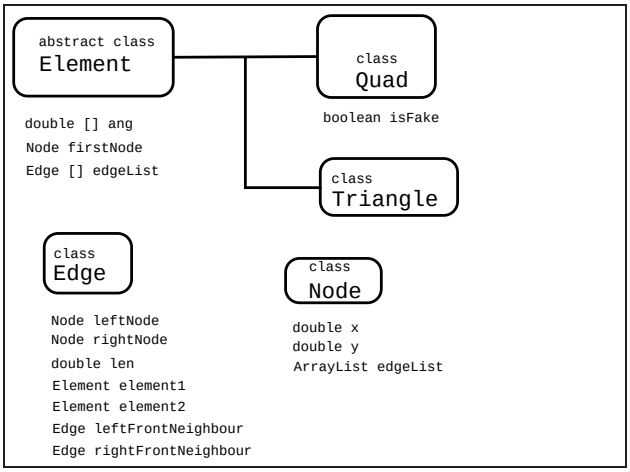


Figure 6.1: The implemented data structure, somewhat simplified. The Quad and Triangle classes are subclasses of the Element class. Only the most significant data fields in each class are shown here.

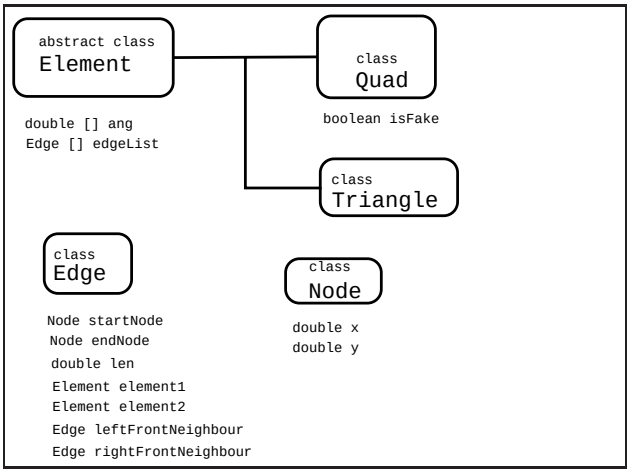


Figure 6.2: Perhaps a better data structure.

specified node, this can be accomplished by starting at the edge which we already know, and then traversing the elements around the node. Thus, the `edgeList` field in the `Node` class can also be removed.

We demand that the edges connected to a particular element all have the same direction, i.e. they all have this element on a specified side, e.g. their left side. The `firstNode` field in the `Element` class was only needed to help determine if the element had become inverted. Since the edges now are directed, detecting inversion is easier, and the `firstNode` field is no longer needed.

6.2 Topological cleanup

The topological cleanup process has a significant impact on mesh quality, even with the modest number of cleanup cases currently implemented. It is evident from some of the test cases that the implementation would benefit from implementing even more cleanup cases.

6.3 Code optimizations

The `ArrayList` class is used extensively throughout the implementation, although in many cases, more efficient array classes could have been chosen.

Although it is stated specifically in [16] that there should be four state lists, one for each state, Q-Morph does not discriminate between edges in states 1-0 and 0-1. Thus, a simplification would be to put those edges in a common list. (This improvement has been carried out in the current implementation.)

Sometimes the removal of a boundary diamond actually degrades the quality of the mesh. It is likely that a better result would be obtained by using distortion metrics to determine whether to close a boundary diamond or not.

Chapter 7

Conclusion

7.1 Summary of results

We have seen that some of the promises made in the paper by Owen et al. [16] appears to hold also for this implementation: At most one triangle is generated, and the mesh quality is generally high, even though it cannot always compare to the quality of the initial triangle mesh. The reason for this is presumably that more cleanup cases are needed for the CleanUp implementation, but perhaps also that triangles are in fact better suited in cases with relatively few boundary intervals and highly non-trivial boundaries, i.e. boundaries with extreme concavities and unevenly spaced nodes.

With respect to the initial triangle mesh, it appears that its quality and positioning of internal nodes have little influence on final element quality. It seems that perhaps a high number of internal nodes is beneficial.

In some of the cases, a better mesh quality especially in terms of minimal distortion metric, can be achieved by defining a chevron as a quad whose greatest internal angle is greater than 180° rather than 200°. However, the current implementation becomes unstable with this value (180°).

7.2 Further work

7.2.1 Performance

As mentioned in the previous chapter, the mesh quality can be increased by implementing more cleanup cases. (Especially, there seems to be a need for cleanup cases that remove 2-valent nodes.)

The triangle mesher will not really be useful before it can handle node sets with constraints. This way it can mesh domains with holes and concave boundaries.

Presumably, the current Java implementation is somewhat slow and not well suited for large datasets. Neither is interfacing with other FEM tools. To improve on this situation, some countermeasures could be taken: Firstly, the program would benefit greatly from implementation of filter methods for reading and writing common mesh file formats. Secondly, the optimizations suggested in the previous chapter could be implemented.

As a last resort, and a rather drastic enterprise, the code could be ported into `C++` and thereby contributing to the speed-up. As the implementation is not heavily based on exotic Java classes, this would perhaps not be as difficult as it might sound. A port to `C++` would also ease the interfacing with existing scientific computing code.

One important task that probably remains incomplete, is the identification and correction of all bugs. Countless hours of labour have already been spent on this goal, but I still fear that it has not been entirely successful.

7.2.2 A comparison with other quad meshing methods

There seems to be considerable activity going on in the meshing method research community, with new papers being published all the time, and all over the world. Common topics are e.g. tetrahedral and hexahedral mesh generation, mesh improvement, and triangular and quadrilateral mesh generation. In the context of this thesis, the interesting papers are predominantly the ones dedicated to quadrilateral mesh generation.

The ultimate goal in this research is to find the optimal method. However, this might not be an achievable goal, at least not for years to come. Thus, it would be interesting to conduct really extensive experiments to at least single out the best method available at the present time. Here, another

problem is posed by all the different metrics appearing in the papers. It is impossible to pick a winner when they all use their own metrics, and their own sets of cases. Someone really ought to enforce a standard by which all methods could be fairly judged. Properties that should be evaluated are e.g. element quality, robustness, efficiency and application to non-planar surfaces. An extensive set of test cases should also be part of the standard.

Other quad-meshers are e.g. LayTracks [18], the Paving algorithm [2], and Lee and Lo's algorithm [12].

However, perhaps the best result can be achieved independently of the quad generation method. Perhaps are the best results obtained by applying clever mesh improvement methods.

Considering the complexity of the Q-Morph algorithm, and the associated difficulties in implementing it, it would be fair to expect a relatively huge leap in performance over the more straightforward and naive quad meshing algorithms. The simplest way to generate an all quadrilateral mesh from a triangle mesh, is splitting each triangle into three quads, as mentioned in [15]. (Add one internal node and one node on each edge of the triangle.) The drawback with this method is that it introduces a huge number of irregular nodes. Another approach is the merging method, which is described in [13, 15]. In short, one can explain this method simply as merging pairs of adjacent triangles in some intelligent fashion. The drawback here is the risk of not achieving an all-quad mesh, i.e. more than one triangle might appear in the final mesh.

Furthermore, the post-processing methods used by Q-Morph can easily be applied to the naive methods, presumably giving them quite a boost in mesh quality. It would be interesting to see whether this improvement could close the expected mesh quality gap between the methods.

Glossary

A triangle is ***degenerate*** if its nodes are collinear. A quad is ***degenerate*** if three or all four of its nodes are collinear.

A ***regular node*** is a node with four incident edges, in the case of a quad mesh, and a node with six incident edges, in the case of a triangle mesh. Thus, all other nodes are ***irregular*** nodes.

A ***regular triangulation*** Δ defined on a domain Ω must meet the following requirements:

1. No triangles are degenerate.
2. No interiors of any two triangles are intersecting.
3. No vertices intersect any edge except for at its endpoints.
4. The union of all triangles in Δ equals Ω.
5. The domain is connected.
6. The triangulation has no holes.
7. Two boundary edges are incident with each boundary vertex.

The ***valence*** of a node (or valency or degree of a node) is the number of edges incident with it. However, on the boundaries this is not entirely true, at least not by the definition of "valence" found in [9]. This definition explains how to compute the valence of boundary nodes, but for practical purposes the following interpretation will suffice:

If the internal angle at a boundary node is less than or equal to 135°, then the valence is the number of incident edges plus 2. If the angle is in the range $< 135°, 225° >$, then the valence is the number of incident edges plus 1. Otherwise the valence is equal to the number of incident edges.

A ***valence pattern*** is a list of valences. The first number is the valence of the central node. Then comes a dash, and lastly, in ccw order around the central node, follow the valences of the neighbour nodes. An example of a valence pattern is 5-4433534444.

Valence pattern	Vertex pattern	Composition
5-4+34+3404043	1 0 1 0 0 1 0 1 0 0	5,1,9
6-4-4+4344+4-34+34+3	0 1 0 0 0 1 0 0 1 0 1 0	2,1,5,2,1,8,2,1, 0,2,1,0,5,0,2,1, 2,0,1,9,5
6-4+3434+3404043	1 0 0 0 1 0 0 1 0 1 0 0	5,1,2,1,5,1,0,5
5-4+340404043	1 0 0 1 0 1 0 1 0 0	5
5-4+0434+34+34+3	0 1 0 0 1 0 1 0 1 0	1,2,1,0,8,5,0,1, 2,0,1,9,2,1,0,1,9

Table 1: The standard cases in connectivity cleanup and their solutions.

Valence pattern	Internal nodes pattern	Composition
5-3443000000	-	1,9
5-4430000003	-	8
5-344+4300000	-	1,0,5
3-4+34+000	1 1 1 0 0 0	4
3-354544	1 0 0 0 0 0	0,3,0,3
4-34434445	1 1 1 1 1 0 0 0	4,0,3,2,0,1,0,2, 1,4,0,3

Table 2: The normal cases in connectivity cleanup and their solutions. Note that the case in row 4 can only be reliably identified by using its vertex pattern in addition to the valence and internal nodes patterns. The vertex pattern for row 4 is: 0 0 0 1 0 1.

Compositions

The following tables contain the compositions of α-iterators and mesh modification codes that are used in the implementation for resolving the different cleanup cases. Illustrations of most of these cases are found in [9].

Valence pattern	Boundary pattern	Composition
5-4354344	1 1 0 1 1 0 0 1	1,8
5-44+34534	1 1 0 0 1 1 0 1	1,2,1,9
4-43544	1 1 0 1 1 1	1,5
4-44534	1 1 1 1 0 1	1,5
4-54345434	0 1 0 0 1 1 0 0 1	8,2,0,1,0,2,3,1,0,2,3
5-4353534	1 1 0 1 1 1 0 1	1,2,1,5,1,8

Table 3: The boundary cases and their solutions. The first value in each of the boundary patterns indicates the boundary state of the central node.

Bibliography

[1] Ivo Babuška and A.K. Aziz. On the Angle Condition in the Finite Element Method. *SIAM Journal on Numerical Analysis*, 13(2):214–226, 4 1976.

[2] Ted D. Blacker and Michael B. Stephenson. Paving: A New Approach to Automated Quadrilateral Mesh Generation. *International Journal for Numerical Methods in Engineering*, 32:811–847, 1991.

[4] Scott A. Canann, Joseph. R. Tristano, and Matthew L. Staten. An Approach to Combined Laplacian and Optimization-Based Smoothing for Triangular, Quadrilateral and Quad-dominant Meshes. In *Proceedings of the 7th International Meshing Roundtable*, pages 479–494. October 1998.
URL http://www.imr.sandia.gov/papers/imr7/canann98.ps.gz

[3] Scott Canann, Sella Mutukrishnan, and Bob Phillips. Topological Improvement Procedures for Quadrilateral and Triangular Finite Element Meshes. In *Proceedings of the 3rd International Meshing Roundtable*, pages 559–588. 1994.

[5] A.K. Cline and R.J. Renka. A storage efficient method for construction of a thiessen triangulation. *Rocky Mountain J. Math.*, 14:119–140, 1984.

[6] David Eberly. Intersection of Linear and Circular Components in 2D, 2002. Documentation to Magic Software's online source code.
URL
http://www.magic-software.com/Documentation/IntersectionLin2Cir2.pdf

[7] L.N. Gifford. More on Distorted Isoparametric Elements. *Int. J. Numer. Method Eng.*, 14(2):290–291, 1976.

[8] Sun Microsystems Inc. *Java Native Interface*, 2002.
URL http://java.sun.com/j2se/1.3/docs/guide/jni/

[9] Paul Kinney. CleanUp: Improving Quadrilateral Finite Element Meshes. In *Proceedings of the 6th International Meshing Roundtable*, pages 449–467. Ford Motor Company, 1997.
URL http://www.imr.sandia.gov/papers/imr6/kinney97.ps.gz

[10] Hans Petter Langtangen. *Computational Partial Differential Equations*. Springer, 1999.

[11] D.A. Lavender and D.R. Hayhurst. An Assessment of Higher-Order Isoparametric Elements for Solving an Elastic Problem. *Computer Methods in Applied Mechanics and Engineering*, 56:139–165, 1986.

[12] C. K. Lee and S. H. Lo. A new scheme for the generation of a graded quadrilateral mesh. *Computers and Structures*, 52(5):847–857, 1994.

[13] S. H. Lo and C. K. Lee. On using meshes of mixed element types in adaptive finite element analysis. *Finite Element Anal. Design*, 11:307–336, 1992.

[16] Steven J. Owen, Matthew L. Staten, Scott A. Canann, and Sunil Saigal. Advancing Front Quadrilateral Meshing Using Triangle Transformations. In *Proceedings of the 7th International Meshing Roundtable*. October 1998.
URL http://www.imr.sandia.gov/papers/imr7/owen98.ps.gz

[15] Steven J. Owen. A Survey of Unstructured Mesh Generation Technology. In *Proceedings of the 7th International Meshing Roundtable*, pages 239–267. October 1998.
URL
http://www.imr.sandia.gov/papers/imr7/owen_meshtech98.ps.gz

[14] Steve J. Owen. Mesh Generation: A Quick Introduction, 2002. Introduction given at Steve Owen's Meshing Research Corner.
URL http://www.andrew.cmu.edu/user/sowen/mintro.html

[17] Samuel Peterson. Computing Constrained Delaunay Triangulations in the Plane, 1998. Part of Minnesota Center for Industrial Mathematics Undergraduate Industrial Mathematics Project.
URL http://www.geom.umn.edu/~samuelp/del_project.html

[18] W. R. Quadros, K. Ramaswami, F. B. Prinz, and B.Gurumoorthy. Laytracks: A new approach to automated quadrilateral mesh generation using MAT. In *Proceedings of the 9th International Meshing Roundtable*, pages 239–250. 2000.
URL http://www.imr.sandia.gov/papers/imr9/quadros00.ps.gz

[19] Jonathan Richard Shewchuk. *Lecture Notes on Delaunay Mesh Generation*, September 1999. Lecture notes to Shewchuk's course, CS 294-5 Meshing and Triangulation in Graphics, Engineering, and Modeling.
URL
http://www.cs.berkeley.edu/~jrs/meshpapers/delnotes.ps.gz

[20] Matthew L. Staten and Scott A. Canann. Post refinement element shape improvement for quadrilateral meshes. *Trends in Unstructured Mesh Generation*, 220:9–16, 1997.
URL http://www.imr.sandia.gov/papers/mcnu/staten97.ps.gz

[21] Olgierd Cecil Zienkiewicz. *The finite element method*, chapter 9. McGraw-Hill Book Company (UK) Limited, third edition, 1977.