Anisotropic Mesh Generation with Particles

Frank Bossen

May 13, 1996

CMU-CS-96-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Email: bossen@cs.cmu.edu
bossen@ltssg7.epfl.ch

WWW: http://www.cs.cmu.edu/~bossen
http://ltswww.epfl.ch/~bossen

This document is a revised version of the author's master's thesis (Ingénieur EPF), Computer Science, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, March 1996.

Keywords: mesh generation, finite element analysis, CFD, anisotropy, Riemannian geometry, Delaunay triangulation, mesh smoothing, mesh relaxation

Abstract

Many important real-world problems require meshing, that is the approximation of a given geometry by a set of simpler elements such as triangles or quadrilaterals in two dimensions, and tetrahedra or hexahedra in three dimensions. Applications include finite element analysis and computer graphics. This work focuses on the former.
A physically-based model of interacting "particles" is introduced to uniformly spread points over a 2-dimensional polygonal domain. The set of points is triangulated to form a triangle mesh. Delaunay triangulation is used because it guarantees a low computational cost and reasonably well-shaped elements. Several particle interaction (repulsion and attraction) models are investigated ranging from Gaussian energy potentials to Laplacian smoothing. Particle population control mechanisms are introduced to make the size of the mesh elements converge to the desired size.
In most applications spatial mesh adaptivity is desirable. Triangles should not only adapt in size but also in shape, to better fit the function to approximate. Computational fluid dynamics simulations typically require triangles stretched in the direction of the flow. A metric tensor is introduced to quantify the stretching. The triangulation procedure is changed to generate "Delaunay" meshes in the Riemannian space defined by the metric.
This new approach to mesh generation appears quite promising.

Contents

Chapter 1

Introduction

Many important real-world problems require meshing, that is the approximation of a given geometry by a set of simpler elements such as triangles or quadrilaterals in two dimensions, and tetrahedra or hexahedra in three dimensions. Applications include finite element analysis [1] and computer graphics.

Many engineering simulations require the solution of partial differential or integral equations. Since most of these equations cannot be solved analytically, approximations must be used. If the domain has a simple shape, one can use finite difference methods with structured grids. For more complex domains, finite element methods are used on an unstructured grid, that is a mesh.

Any mesh generator should address the following concerns:

- functionality. Obviously it should work.
- robustness. It should work *all* the time, with *any* input.
- quality. The quality of the resulting mesh should be good, that is closely match the desires of the user.
- speed. The generation process should be fast.
- minimal user interaction. Everything that can be automated should be.
- controllability. The user should be able to influence the result in predictable ways.

In this work we limit ourselves to generate triangular meshes inside a two dimensional domain bounded by a polygon which may contain holes. The main application we focus on is finite element analysis.

A physically-based model of interacting "particles" is introduced to uniformly spread points over the domain. The set of points is triangulated to form a triangle mesh. Delaunay triangulation is used because it guarantees a low computational cost and reasonably well-shaped elements. Several particle interaction (repulsion and attraction) models are investigated ranging from Gaussian energy potentials to Laplacian smoothing. Particle population control mechanisms are introduced to make the size of the mesh elements converge to the desired size.

In most applications spatial mesh adaptivity is desirable. Triangles should not only adapt in size but also in shape, to better fit the function to be approximated. Computational fluid dynamics simulations typically require triangles stretched in the direction of the flow. A metric tensor is introduced to quantify the stretching. The triangulation procedure is changed to generate "Delaunay" meshes in the Riemannian space defined by the metric.

1.1 Previous Work

There are several surveys available on mesh generation [2, 15].

Most of present mesh generation algorithms are structured in the following way. First a mesh is build with methods such as advancing front [19, 21, 23], quadtree decomposition [35], or by greedy point insertion [3, 32]. The quality of the mesh is further improved with the use of smoothing. The most common method is Laplacian smoothing [8].

Advancing front methods start meshing at the boundaries of the domain. A list of nodes to be expanded (referred to as the front) is maintained. At each iteration, the front advances by expanding a node, and inserting a new node at the desired distance from the front. Badly shaped elements can appear in the middle of the domain, where the fronts collide.

Quadtree methods recursively split a square surrounding the domain into four smaller squares, until the desired size is reached. To obtain a triangular mesh, the squares are further divided into triangles.

Greedy insertion methods localize poorly shaped, or poorly sized elements, and split them by inserting a new node on an edge [3], at the center of a triangle [32], or at the center of the circle circumscribing a triangle [25]. Although the Delaunay criterion is the most common, other triangle quality criteria have been used [31] to determine the topology of the mesh.

Laplacian smoothing consists of moving each vertex to the centroid of its neighbors. This operation must generally be repeated several times for each node before the quality of the mesh is improved. Although Laplacian smoothing generally improves the shape of elements, it is not guaranteed to do so, especially if the domain is concave. Furthermore the degrees of the nodes, which are often the reason of poorly shaped elements, are not affected. Relaxations methods that combine Laplacian smoothing with local topological optimization have been proposed to remedy to this problem [10, 11].

Anisotropic mesh generation has not benefited from an extensive literature. One of the major fields of application for anisotropic meshes is computational fluid dynamics (CFD), where stretched triangles oriented in the direction of the flow are desirable. Mavriplis [22] proposed to stretch the plane by lifting it on a surface in three dimensions. The deformation of space is represented at each point by two values: an angle giving the direction of the stretching, and a value larger than 1 quantifying the stretching. Before that, Peraire [23] was using a similar representation to quantify the desired element size as a function of its position and orientation.

Later a metric tensor was introduced [3, 31]. The tensor representation has the advantage to be directly related to the Hessian of the function (speed, pressure, etc.) to be estimated [5]. It is thus a more natural representation to create adapted meshes. Quite impressive results have been produced by Castro-Diaz, Hecht, and Mohammadi [3]. In their method, which is of the greedy insertion type, edges that are too long are split, and edges that are too short are collapsed. When the desired number of elements is reached, the mesh is further relaxed to improve its quality.

Mesh generation has also benefited from other kinds of approaches. The idea of using physically-based simulations for mesh generation has been investigated by Shimada [26] and his bubble packing method. The bubble interaction (attraction/repulsion) model, which is inspired by the Lennard–Jones interaction model from molecular chemistry, generates triangulations that imitate Nature in her way of producing regular arrangements of points, such as in crystals. The main advantages of this method are good point placement, and intrinsic remeshing capabilities.

Physically-based models have also been used in computer graphics for sampling surfaces [29, 30, 33, 34]. In these models, particles spread over complex surfaces to form uniform sampling patterns. Szeliski [29] used attracting and repelling, oriented particles to interactively sculpt surfaces. Turk [30] considered resampling polygonal surfaces using repelling particles. Witkin and Heckbert [34] used repelling particles to sample implicit surfaces. Although many have thought of introducing anisotropy, and defining the sampling density based on surface curvature, few have done it [30]. Witkin and Heckbert [34] have noticed that such schemes can produce very regular patterns of points. Their work has been the starting point of the work presented in this document.

1.2 Approach Overview

In this work, we limit ourselves to generate triangular meshes over polygonal domains in the plane. The domain can contain holes, and constraints such as line segments and points. Constrained points define nodes that should appear in the mesh, and constrained line segments should not be crossed by any edge in the mesh. The domain can be represented by a planar straight line graph (PSLG), which is a set of points and non–crossing edges. An example of such a graph is given in Figure 1.1.

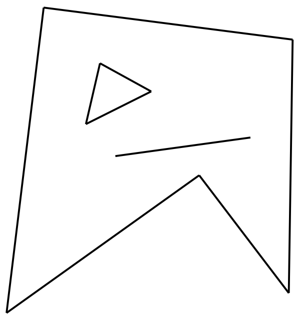


Figure 1.1: A planar straight line graph

Given such a domain, points are added inside it and triangulated to form a mesh such that the length of every edge matches as closely as possible a feature size function[1]. This function is given in input to the mesher, and is defined in terms of the position of the vertices of an edge. Different classes of such functions generate different kinds of meshes such as (see Figure 1.2):

constrained where no feature size function is specified

[1] We also refer to the feature size function as the desired edge length

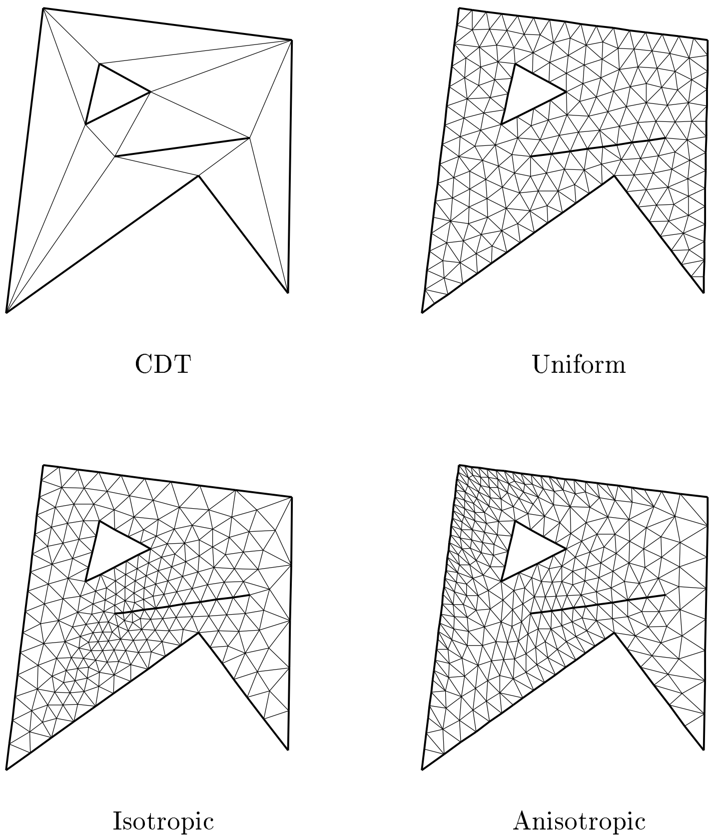


Figure 1.2: Different kinds of meshes

uniform where the feature size function is a constant

isotropic[2] where the desired edge length depends on the position of the edge

anisotropic where the desired edge length depends on the position and the orientation of the edge

To distribute the nodes inside the domain, a physically-based model of interacting "particles" is introduced. The functionality of the model can be expressed as:

1. Input: polygonal domain and feature size function
2. build an initial triangulation of the domain (constrained Delaunay), and create a particle on each vertex of the domain
3. create, move, and annihilate particles inside the domain until equilibrium is reached. The triangulation of the set of particles is maintained Delaunay at all times, that is

[2]These kind of meshes are also referred to as graded meshes

the topology of the mesh is locally optimized after each particle movement, creation, and annihilation.

4. Output: a nice mesh

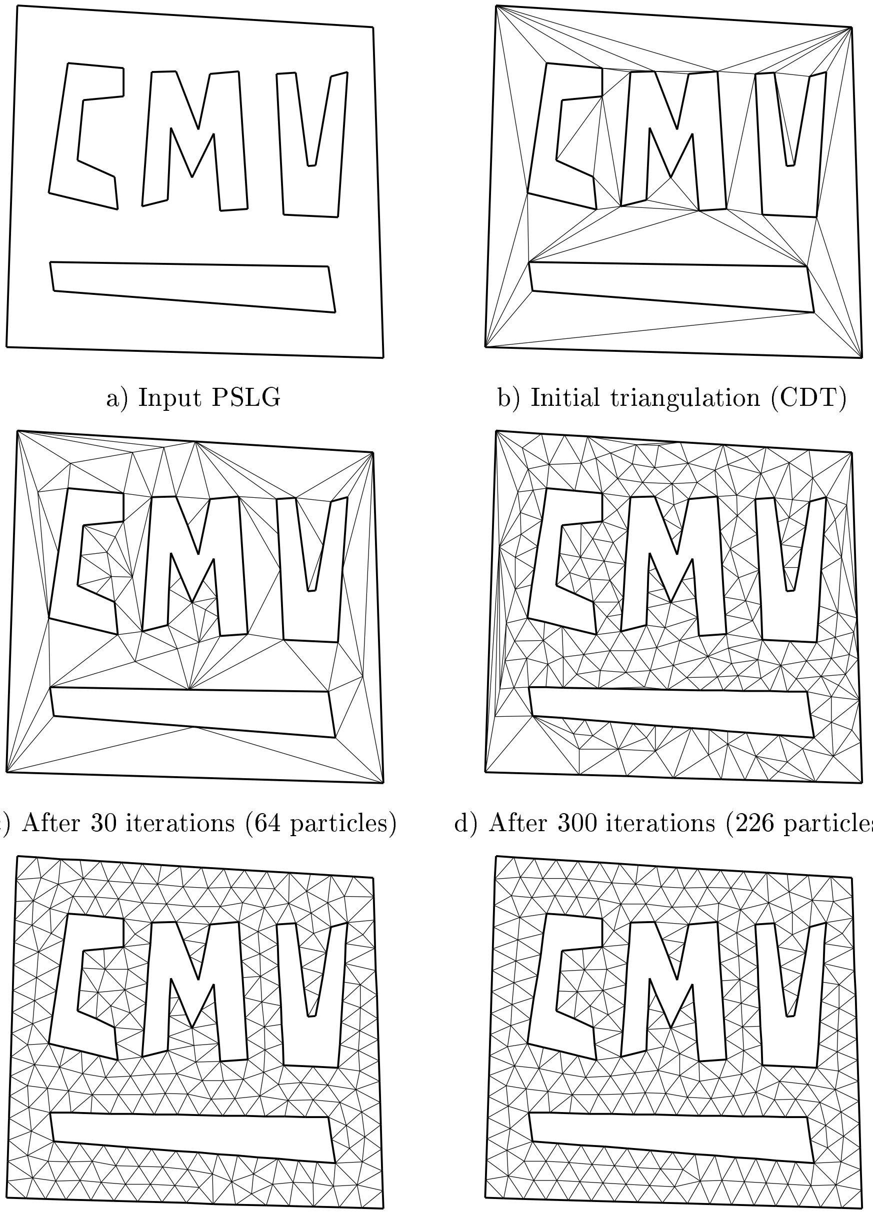
To illustrate this algorithm, consider the domain depicted in Figure 1.3a. First its constrained Delaunay triangulation is built (Figure 1.3b).

Then the physically-based process is started. At each step, a particle is randomly picked, and its position updated according to the positions of its neighbors. Then the local particle density is estimated, and a particle is created/annihilated if the density is too low/high. Figure 1.3c–f shows the evolution of the mesh. In a first phase (approximatively first 1000 steps in this case), particles are created until the population reaches the desired level which depends on the feature size function. The growth is regulated by an adaptive population control scheme. In a second phase, the mesh is regularized, that is node placement is improved. For this example, after 8410 steps, equilibrium is reached and the algorithm halts.

During the whole simulation the triangulation is maintained Delaunay using procedures for point insertion, motion and removal.

1.2.1 Document Outline

The outline of this document is as follows. Chapter 2 introduces Delaunay triangulations and related algorithms. Chapter 3 describes the physically-based model of interacting particles. Chapter 4 is an *interlude* in which the use of Java for programming an interactive Delaunay triangulator on the web is presented. Chapter 5 generalizes the model described in Chapter 3 to the anisotropic case, and presents some results. Finally conclusions are drawn in Chapter 6.



a) Input PSLG

b) Initial triangulation (CDT)

c) After 30 iterations (64 particles)

d) After 300 iterations (226 particles)

e) After 3000 iterations (295 particles)

f) After 8410 iterations (295 particles)

Figure 1.3: Mesh Evolution

Chapter 2

Delaunay Triangulations

The Voronoi diagram (VD) of a set $S = \{s_1, s_2, \ldots, s_n\}$ of points in the plane, called *sites*, is a partition of the plane into n convex regions, one per site. Each Voronoi cell V_i contains all the points in the plane closer to s_i than to any other site. The planar dual of the Voronoi diagram, obtained by adding a line segment between each pair of sites of S whose Voronoi regions share an edge, is called the Delaunay triangulation (DT).

More practical definitions of a Delaunay triangulation are (all of the following statements are equivalent):

- if a and b are input points, the DT contains the edge $\{a, b\}$ if and only if there is a circle through a and b that intersects no other input points and contains no input points in its interior

- the circumscribing circle of each triangle contains no input points in its interior.

There is also a nice relationship between Delaunay triangulations and 3-dimensional convex hulls. Lift each point of the input to a paraboloid by mapping the point (x, y) to $(x, y, x^2 + y^2)$[1]. It can be proved [7] that the DT of the input points is the projection of the lower convex hull onto the xy-plane.

The Delaunay triangulation features other interesting properties. Indeed it maximizes of the minimum angle, minimizes the maximum circumcircle as many other measures.

The outline of the chapter is as follows. Section 2.1 defines data structures for representing triangulations. Associated topological and geometrical operators are defined in Sections 2.2 and 2.3. Section 2.4 presents some methods for building Delaunay triangulations. Section 2.5 addresses the problem of dynamically maintaining a triangulation (incremental site insertion and removal). Section 2.6 presents constrained Delaunay triangulations (CDT).

[1] This property also holds for any paraboloid $z = \alpha \cdot \left((x-a)^2 + (y-b)^2\right)$ where α, a, and b are constants. This trivially follows from the fact that a DT is invariant to translations of the input set of sites. The value of α doesn't matter either since it neither affects the topology of the convex hull, nor its projection on the plane

2.1 The Quadedge Data Structure

The quadedge data structure [13] was designed for representing general subdivisions of orientable manifolds. It simultaneously represents both the subdivision and its dual. Alternatives are the double-connected-edge-list (DCEL) [24] and the winged-edge data structures, which do not hold the dual. The quadedge structure has been preferred because an implementation was readily available [20].

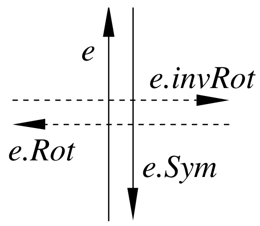


Figure 2.1: The four directed edges of a quadedge

Each quadedge record holds four directed edges corresponding to a single undirected edge in the subdivision and to its dual edge. Each directed edge has three references: *Org*, which points to the site at its origin, *Rot*, which points to its dual edge, and *Onext*, which points to the counterclockwise next edge in the subdivision. All other topological operators (see Figures 2.1 and 2.2) can be defined in terms of these primitives, as summarized below.

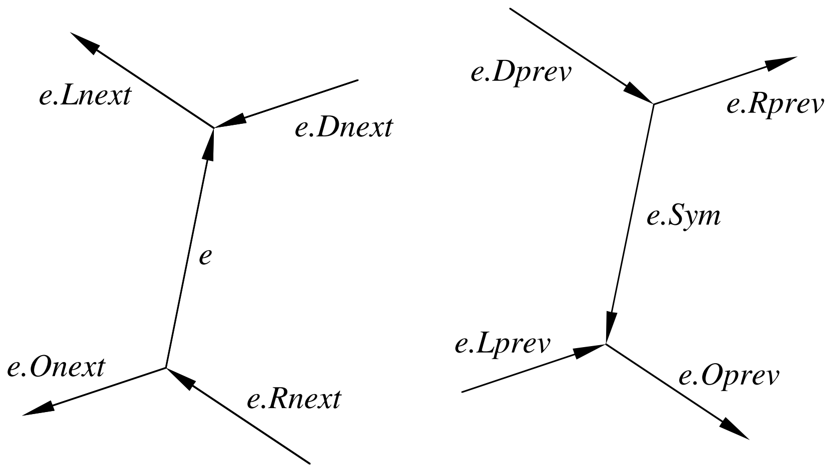


Figure 2.2: Edge navigation operators

Navigation inside a quadedge is achieved with the use of the identity, *Rot*, *Sym*, and

Rot^{-1} operators, all of which can be expressed in terms of Rot operations:

$$\begin{array}{lcl} e & = & e \\ e.Rot & = & e.Rot \\ e.Sym & = & e.Rot^2 \\ e.Rot^{-1} & = & e.Rot^3 \end{array} \tag{2.1}$$

where $e.Rot$ means that operator Rot is applied to edge e, and $e.Rot^n$ that Rot is applied n times.

The position of each site in the quadedge can be retrieved with the Org operator:

$$\begin{array}{lclcl} e.Org & = & e.Org & = & e.Org \\ e.Left & = & e.Rot^{-1}.Org & = & e.Rot^3.Org \\ e.Dest & = & e.Sym.Org & = & e.Rot^2.Org \\ e.Right & = & e.Rot.Org & = & e.Rot.Org \end{array} \tag{2.2}$$

where the middle column represents the definition of the operator, and the right column its expression in terms of primitives.

$Onext$ allows navigation through the subdivision. Movements to neighboring edges are represented in figure 2.2. They can be subdivided into two classes, namely counter-clockwise ones:

$$\begin{array}{lclcl} e.Onext & = & e.Onext & = & e.Onext \\ e.Lnext & = & e.Rot^{-1}.Onext.Rot & = & e.Rot^3.Onext.Rot \\ e.Rnext & = & e.Rot.Onext.Rot^{-1} & = & e.Rot.Onext.Rot^3 \\ e.Dnext & = & e.Sym.Onext.Sym & = & e.Rot^2.Onext.Rot^2 \end{array} \tag{2.3}$$

and clockwise ones:

$$\begin{array}{lclcl} e.Oprev & = & e.Rot.Onext.Rot & = & e.Rot.Onext.Rot \\ e.Lprev & = & e.Onext.Sym & = & e.Onext.Rot^2 \\ e.Rprev & = & e.Sym.Onext & = & e.Rot^2.Onext \\ e.Dprev & = & e.Rot^{-1}.Onext.Rot^{-1} & = & e.Rot^3.Onext.Rot^3 \end{array} \tag{2.4}$$

When assuming that the subdivision is a triangulation, it is possible to simplify some of the operations:

$$\begin{array}{lcl} e.Dprev & = & e.Onext.Rot^2.Onext \\ e.Dnext & = & e.Rot.Onext^2.Rot \end{array} \tag{2.5}$$

The next section presents higher level operators.

2.2 Topological Operators

Starting with the operator set proposed by Guibas and Stolfi [13], we have modified it to make it more symmetrical, and also to eliminate superfluous edge navigation. The parameters of the $Connect$ operator have been changed and its inverse operator $Disconnect$ is introduced. The latter accommodates some operations that were previously part of $DeleteEdge$. As a consequence $DeleteEdge$ has also been changed. Each operator is described in the next paragraphs.

2.2.1 MakeEdge

MakeEdge creates a new quadedge and initializes all of its four directed edges. It takes no argument and returns the first edge of the quadedge. The quadedge is a subdivision by itself, namely the one of a sphere [13]. The details of the procedure are described below. The *Create* operator creates a new directed edge and initializes all its pointers to *nil*.

begin $Edge.MakeEdge$
$this.Create,\quad e_2.Create,\quad e_3.Create,\quad e_4.Create$
$this.Onext \leftarrow this, e_2.Onext \leftarrow e_4,\quad e_3.Onext \leftarrow e_3,\quad e_4.Onext \leftarrow e_2$
$this.Rot \leftarrow e_2,\quad e_2.Rot \leftarrow e_3,\quad e_3.Rot \leftarrow e_4,\quad e_4.Rot \leftarrow this$
end

2.2.2 Splice

Splice is the basic operator used to attach and detach edges from each other (see figure 2.3). It takes an edge b as parameter and returns no value. It is its own inverse. Its description is given below.

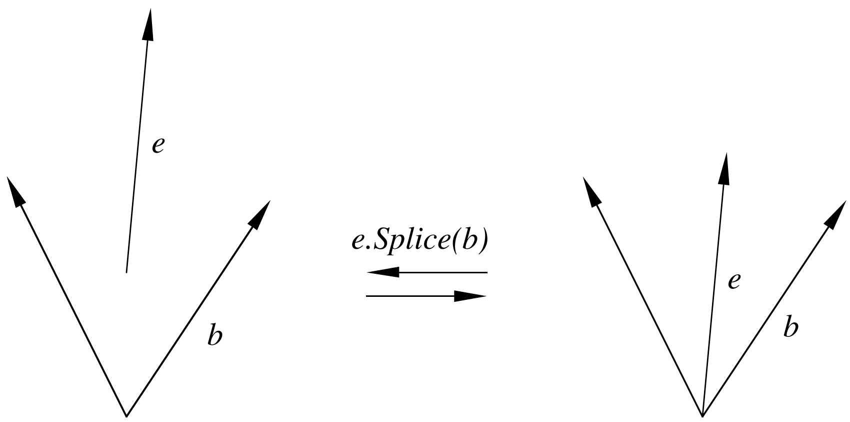


Figure 2.3: The effect of the *Splice* operator

begin $Edge.Splice(b)$
$e_1 \leftarrow this.Onext,\quad e_2 \leftarrow b.Onext$
$\alpha \leftarrow e_1.Rot,\quad \beta \leftarrow e_2.Rot$
$e_3 \leftarrow \alpha.Onext,\quad e_4 \leftarrow \beta.Onext$
$this.Onext \leftarrow e_2,\quad b.Onext \leftarrow e_1$
$\alpha.Onext \leftarrow e_4,\quad \beta.Onext \leftarrow e_3$
end

2.2.3 Connect

The *Connect* operator connects the two vertices of an edge e to respectively two edges a and b. It takes as argument the two edges a and b, and returns no value. The *Connect* operator is basically a succession of two *Splice* operations followed by an update of the *Org* fields. A description is given below.

begin $Edge.Connect(a, b)$
  $this.Splice(a)$
  $this.Org \leftarrow a.Org$
  $this.Sym.Splice(b)$
  $this.Dest \leftarrow b.Org$
end

This new definition of the *Connect* operator changes in two ways from its original [13] form: it does not create a new edge any more, and the arguments represent different edges so that they can directly be applied to both of the *Splice* operations.

2.2.4 Disconnect

The *Disconnect* operator is the inverse of the *Connect* operator. Thus[2]

$$e.Connect(a, b).Disconnect(a, b) \equiv e$$

Its description is given below. Since the *Splice* operator is its own inverse the *Disconnect* operator is quite the same as *Connect*. The difference lies in the update of the *Org* fields (which are reset to *nil* by *Disconnect*).

begin $Edge.Disconnect(a \leftarrow this.Oprev, b \leftarrow this.Lnext)$
  $this.Splice(a)$
  $this.Org \leftarrow nil$
  $this.Sym.Splice(b)$
  $this.Dest \leftarrow nil$
end

2.2.5 Swap

The *Swap* operator is at the core of the edge swapping algorithm described in section 2.4.1. To swap an edge is to replace it by the other diagonal of the quadrilateral in which it is inscribed. *Swap* is its own inverse and thus $e.Swap.Swap \equiv e$.

begin $Edge.Swap$
  $a \leftarrow this.Lprev$
  $b \leftarrow this.Sym.Lprev$
  $c \leftarrow a.Lprev$
  $d \leftarrow b.Lprev$
  $this.Disconnect(d, c)$
  $this.Connect(b, a)$
end

[2]The relation only holds if $e.Org = e.Dest = nil$

2.2.6 DeleteEdge

The *DeleteEdge* operator undoes everything the *MakeEdge* operators does, as described below.

begin *Edge.DeleteEdge*

$e_2 \leftarrow this.Rot$,	$e_3 \leftarrow e_2.Rot$,	$e_4 \leftarrow e_3.Rot$	
$this.Rot \leftarrow nil$,	$e_2.Rot \leftarrow nil$,	$e_3.Rot \leftarrow nil$,	$e_4.Rot \leftarrow nil$
$this.Onext \leftarrow nil$,	$e_2.Onext \leftarrow nil$,	$e_3.Onext \leftarrow nil$,	$e_4.Onext \leftarrow nil$
$this.Destroy$,	$e_2.Destroy$,	$e_3.Destroy$,	$e_4.Destroy$

end

Putting references to *nil* and then *Destroy*-ing edges clearly is redundant. The reason both appear in the pseudocode it that languages such as C++ require the latter, whereas languages featuring automatic garbage collection require the former.

2.3 Geometrical Operators

Geometrical operators are defined to test the relative positions of points and edges. There is also an *InCircle* operator which tests whether a point lies inside the circle defined by three other points, and which is one of the most important operators for constructing Delaunay triangulations.

2.3.1 OnRight, OnLeft and OnEdge

The *OnRight*, *OnLeft* and *OnEdge* operators tell if a given point is respectively to the right of, to the left of, or on an edge. The decision is based on the sign of the signed area of the triangle defined by the two vertices of the edge and the point to test.

begin $Edge.OnRight(p)$
  return $(p - this.Org) \times (this.Dest - this.Org) > 0$
end

begin $Edge.OnLeft(p)$
  return $(p - this.Org) \times (this.Dest - this.Org) < 0$
end

begin $Edge.OnEdge(p)$
  return $(p - this.Org) \times (this.Dest - this.Org) = 0$
end

where $\times$ is the cross product between two vectors.

2.3.2 InCircle

Given an edge e and two points i and j, the *InCircle* operator returns true if and only if the circle defined by i and the two vertices of e includes j. It tells whether an edge should be swapped or not. This test is at the heart of most Delaunay triangulation algorithms, and is floating point intensive. In practice it is important to optimize it. Three different ways of performing the test are presented, and their speed is evaluated.

Circle Test

The naive way is to compute the position of the center and the radius of the circle, and then check if the distance from its center to j is less than its radius. The center of the circle lies at the intersection of the two lines that respectively bisect $e.Org$ and i, and $e.Dest$ and i. These two lines can be expressed as

$$\begin{aligned} n_1 \cdot x - c_1 &= 0 \\ n_2 \cdot x - c_2 &= 0 \end{aligned}$$

where $n_1 = e.Org - i$ is a normal to the first line and $c_1 = \frac{1}{2}(e.Org + i) \cdot n_1$ is a constant. n_2 and c_2 are similarly defined for the second line. The two equations can be combined into a single matrix equation $Nx - c = 0$. The solution of which is $x = N^{-1}c$, where x is the intersection of the two lines, and thus the center of the circle.

The *InCircle* test then summarizes to

$$|x - j| < |x - i|$$

To compute the center of the circle, 11 multiplications, 13 additions, and 2 divisions are required. The distance test requires additional 6 additions and 4 multiplications. The total cost is thus 15 multiplications, 19 additions, and 2 divisions. It is possible to avoid the divisions at the cost of four extra multiplications (by multiplying everything by the determinant of N).

Convex Hull Test

A better way might be to consider some other properties of Delaunay triangulations. The convex hull test is based on the nice relation between Delaunay triangulations and 3-dimensional convex hulls. The test is based on the sign of the signed volume of the tetrahedron defined by the projection of the four points on a paraboloid:

$$\begin{vmatrix} x_1 & y_1 & x_1^2 + y_1^2 & 1 \\ x_2 & y_2 & x_2^2 + y_2^2 & 1 \\ x_3 & y_3 & x_3^2 + y_3^2 & 1 \\ x_4 & y_4 & x_4^2 + y_4^2 & 1 \end{vmatrix} > 0 \tag{2.6}$$

where $(x_1, y_1) = e.Org$, $(x_2, y_2) = e.Dest$, $(x_3, y_3) = i$, and $(x_4, y_4) = j$. When naively computed, this determinant requires 48 multiplications and 27 additions. A good idea is to consider that the last column is filled with ones to reduce the cost to 20 multiplications and 24 additions, as in the code proposed by Lischinski [20].

It is possible to improve this result by translating all the points by a same amount, so that one of the points lies at the origin $(0, 0)$:

$$\begin{vmatrix} 0 & 0 & 0 & 1 \\ \tilde{x}_2 & \tilde{y}_2 & \tilde{x}_2^2 + \tilde{y}_2^2 & 1 \\ \tilde{x}_3 & \tilde{y}_3 & \tilde{x}_3^2 + \tilde{y}_3^2 & 1 \\ \tilde{x}_4 & \tilde{y}_4 & \tilde{x}_4^2 + \tilde{y}_4^2 & 1 \end{vmatrix} > 0 \tag{2.7}$$

where $\tilde{x}_i = x_i - x_1$, $\tilde{y}_i = y_i - y_1$ and the paraboloid is $\tilde{z}_i = (x_i - x_1)^2 + (y_i - y_1)^2$. The test is thus equivalent to:

$$\begin{vmatrix} \tilde{x}_2 & \tilde{y}_2 & \tilde{x}_2^2 + \tilde{y}_2^2 \\ \tilde{x}_3 & \tilde{y}_3 & \tilde{x}_3^2 + \tilde{y}_3^2 \\ \tilde{x}_4 & \tilde{y}_4 & \tilde{x}_4^2 + \tilde{y}_4^2 \end{vmatrix} > 0 \tag{2.8}$$

which requires only 15 multiplications and 14 additions.

Angle test

It also possible to define a test based on angles[3]: if the sum of angles α and β (see figure 2.4) is smaller than 180 degrees, then the edge should be swapped. Another way to put it is $\sin(\alpha + \beta) > 0$? This expression can be simplified in the following way:

$$\begin{aligned} \sin(\alpha+\beta) &= \sin\alpha\cos\beta + \sin\beta\cos\alpha \\ &= \frac{a \times b}{|a| \cdot |b|} \cdot \frac{c \cdot d}{|c| \cdot |d|} + \frac{c \times d}{|c| \cdot |d|} \cdot \frac{a \cdot b}{|a| \cdot |b|} \\ &\propto (a \times b)(c \cdot d) + (c \times d)(a \cdot b) \end{aligned} \tag{2.9}$$

This test only requires 10 multiplications and 13 additions.

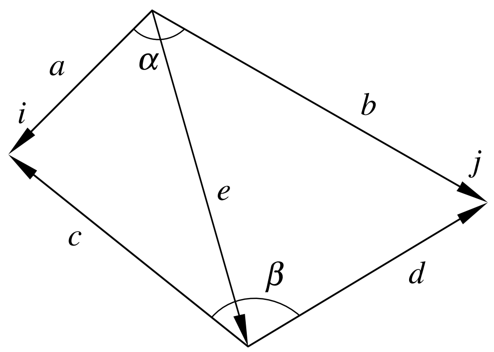


Figure 2.4: The *InCircle* test

Summary

From the several ways of performing the *InCircle* test that have been examined, the angle test is fastest, followed by the convex hull test, and the circle test. The operator is thus defined using the angle test:

[3]Thanks to Marshall Bern for pointing it out

begin $Edge.InCircle(p \leftarrow e.Onext.Dest, q \leftarrow e.Oprev.Dest)$
  $a \leftarrow q - this.Org$
  $b \leftarrow p - this.Org$
  $c \leftarrow p - this.Dest$
  $d \leftarrow q - this.Dest$
  return $(a \times b)(c \cdot d) + (c \times d)(a \cdot b) > 0$
end

2.4 Triangulation Algorithms

Several methods are known for triangulating the convex hull of a set of points, namely the flip algorithm, the incremental algorithm, the divide-and-conquer algorithm and the sweepline algorithm. The first two are the topic of the next sections. The divide-and-conquer and sweepline algorithms are only briefly described because they haven't been used in the frame of this work.

In the divide-and-conquer algorithm, the input site set is first sorted by x-coordinate and split vertically in two subsets of equal size. The Delaunay triangulation is computed recursively for each subset. Subsets are then merged using a sweeping circle algorithm. A complete description is given in [13]. The time complexity of the algorithm is $O(n \log n)$.

The sweepline algorithm is due to Fortune [9]. In a comparison with divide-and-conquer, Leach [18] has optimized both algorithms and measured execution speed. According to Leach, the sweepline algorithm is the fastest, but also appears to be the least robust.

2.4.1 The Edge Swapping Algorithm

Given an initial triangulation, the edge swapping algorithm maintains a queue of edges that might fail the circumcircle test. An edge e is said to pass the circumcircle test if the circle through $\{e.Org, e.Dest, e.Onext.Dest\}$ does not include the point $e.Oprev.Dest$. In the initial triangulation, any edge might fail the test, so the queue initially contains all edges. Then, the first edge e is repeatedly removed from the queue. If e does pass the circumcircle test, the next edge is dequeued. But if e does fail the test, it is swapped. As this operation might change the status of some of the four edges of the quadrilateral of which e is a diagonal, those edges are added into the queue if not already there. The algorithm stops when the queue is empty.

The algorithm always terminates after $O(n^2)$ flips. Combined with an $O(n^2)$ algorithm for constructing the initial triangulation, the Delaunay triangulation of a set of points can be found in $O(n^2)$ time with this method.

2.4.2 The Incremental Algorithm

The incremental algorithm was first proposed by Lawson [17]. It starts with a triangle large enough to contain all the points of the input set (ideally the three vertices are at infinity). Points are added into the triangulation one by one, maintaining the invariant that the triangulation is Delaunay. First the triangle T containing the new point p is located. New edges are created to connect p to the vertices of T. The edges of T are inspected to verify that they still satisfy the circumscribing circle condition. If the condition is satisfied the edge remains unchanged. If it is violated the offending edge is swapped. In this case two more edges become candidates for inspection. The process continues until no more candidates remain.

If the order of insertion is randomized, and with use of appropriate data structures for point-in-triangle location, it can be shown [14] that the expected running time of the algorithm is $O(n \log n)$. However, without any additional data structure, jump-and-walk methods can be used to reach an expected time of $O(n^{4/3})$ [6].

2.5 Dynamic Maintenance of a Triangulation

The problem of dynamically maintaining the triangulation can be divided into two sub-problems: insertion and removal of a site. Moving a site can be seen as a removal followed by an insertion.

2.5.1 Site Insertion

The procedure for inserting a point i runs as follows:

1. locate the triangle T in which i lies
2. add an edge from i to each vertex of T
3. put the three edges of T on a queue
4. while the queue is non-empty, remove the first edge e from the queue. If e does pass the circumcircle test, the next edge is dequeued. If e fails the test, swap it, and insert in the queue the two edges a and b adjacent to e which do not contain i (see Figure 2.5).

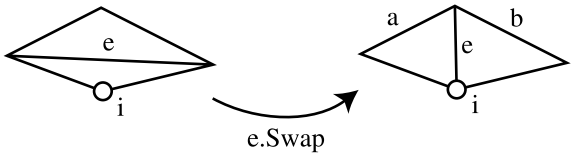


Figure 2.5: Edge swapping during point insertion

It is usually assumed that points are in general position, that is that no three points are collinear or four points cocircular. In practice however this is not the case. As a result the insertion of a site i on an edge e (three colinear point) has to be treated in a special way.

1. remove the edge e. i then lies inside a quadrilateral Q
2. add an edge from the i to each vertex of Q which is the containing quadrilateral
3. put the four edges of Q on a queue

Step 4 is identical to the one for general position point insertion. A more detailed description of the point insertion routine is given in figure 2.6.

The number of edge flips is equal to the degree of i after its insertion. As the average degree of a vertex in a planar graph is less than six, the average cost of step 4 is $O(1)$. Worst

```
begin Insert(p)
   e ← Locate(p)

   if e.OnEdge(p) then
      e ← e.Oprev
      e.Onext.DeleteEdge

   base ← MakeEdge
   base.Splice(e)
   base.Org ← p
   base.Dest ← p

   b ← base.Sym
   loop
      a ← base.Lprev
      if a = e then
         exit loop
      base ← MakeEdge
      base.Connect(a, b)

   e ← base
   do
      a ← e.Onext
      b ← a.Onext
      while a.InCircle(b.Dest, p) do
         a.Swap
         a ← b
         b ← a.Onext
      e ← e.Dprev
   while e ≠ base
end
```

Figure 2.6: Point insertion procedure

case is $O(n)$ and happens, for example, when i is inserted at the center of n points lying on a circle.

The point location cost (step 1) can be reduced to $O(\log n)$ time with appropriate $O(n)$-space data structures [14]. Without additional data structures a simple walking method can be used to achieve an expected $O(\sqrt{n})$ time performance. A fast walking method is described in the next section.

The total cost of an insertion is thus dominated by the cost of the point location procedure.

2.5.2 Walking Method for Point Location

Guibas and Stolfi [13] have proposed a simple walking method for point location, which Lischinski has implemented [20]. An improved version where redundant tests have been removed is described by a finite state automaton in Figure 2.8. At any time the point to locate is on the left of the current edge. Figure 2.7 shows the two possible movements at each step. Performance is compared in Table 2.1 in terms of function calls. It turns out that the new algorithm is more than twice as fast as the original one.

	10000 locations			100000 locations		
Procedure	Guibas	Bossen	ratio	Guibas	Bossen	ratio
$Edge.RightOf$	207.8	90.7	2.29	657	277	2.37
$Point.=$	167.7	78.1	2.15	531	243	2.18
$Edge.Onext$	126.1	44.1	2.86	403	137	2.94
$Edge.Dprev$	80.1	43.6	1.84	251	137	1.83
$Edge.Org$	83.9	39.1	2.15	265	122	2.17
$Edge.Dest$	83.9	39.1	2.15	265	122	2.17
$Edge.Sym$	0.52	0.62	0.84	0.40	0.62	0.65
Total[4]	374.5	166.5	2.25	1183	519	2.27

Table 2.1: Guibas versus Bossen point location performance (listing the number of calls to each procedure, per insertion)

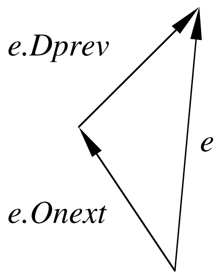


Figure 2.7: The two possible walking directions

2.5.3 Site Removal

Unfortunately removing a site i is not as easy as inserting one. Although it can be simple when using appropriate data structures and removing the last inserted point [16], the general case is more difficult to handle. The procedure description is still very short:

1. remove all the edges e such that $e.Org = i$
2. delete i
3. triangulate the simple polygon which contained i

[4] Sum of all topological operations calls

```
begin Locate(p)
s0:  e ← some edge f such that ¬f.RightOf(p)
     goto if e.Org = p ∨ e.Dest = p then s7 else s1

s1:  f ← e.Onext
     goto if f.RightOf(p) then s4 else s6

s2:  f ← e.Dprev
     goto if f.RightOf(p) then s3 else s5

s3:  f ← e.Onext
     goto if f.RightOf(p) then s7 else s6

s4:  f ← e.Dprev
     goto if f.RightOf(p) then s7 else s5

s5:  e ← f
     goto if e.Org = p then s7 else s1

s6:  e ← f
     goto if e.Dest = p then s7 else s2

s7:  return e
end
```

Figure 2.8: Triangle location procedure

The cost of a deletion is $O(n^2)$ (see next section on simple polygon triangulation), where n is the degree of site i. As the degree i is expected to be less than six on average, the asymptotic time is not relevant in practice.

2.5.4 Triangulation of a Simple Polygon

We consider here the case of a *hole* (simple polygon) inside a triangulation that needs to be retriangulated. Triangulating the simple polygon is not as easy as one would like it to be since the polygon cannot be assumed to be convex. A simple solution is to build a triangulation of the polygon by successively cutting off ears[5], and then run the edge swapping algorithm.

The time cost of the retriangulation is $O(n^2)$, where n is the number of sites of the hole.

2.6 Constrained Delaunay Triangulations

A constrained Delaunay triangulation (CDT) is very similar to a Delaunay triangulation. The difference is that some edges are fixed (constrained). They cannot be swapped even if they do not pass the *InCircle* test.

[5]An ear is a triangle which shares two edges with the polygon to be triangulated

The methods for generating a constrained triangulation are quite similar to the ones used for unconstrained triangulations. The edge swapping algorithm for instance starts with a triangulation where all constrained edges are present, then runs as usual but a constrained edge is never swapped.

There are also $O(n \log n)$ algorithms for generating constrained Delaunay triangulation such as divide-and-conquer [4], but they are usually difficult to implement. The next section presents an asymptotically slower incremental algorithm which is easier to implement.

2.6.1 An Incremental CDT Algorithm

The present algorithm for building constrained Delaunay triangulations can be split in two phases. In the first one, the constraints are ignored, and a triangulation is built using the incremental algorithm described in section 2.4.2. In the second phase, the constrained edges are inserted one by one, if not already present. First all the edges crossing the constrained edge are removed, then the constrained edge is inserted, and finally the simple polygons on each side of the constrained edge are retriangulated. The procedure for removing the crossing edges and inserting the constrained edge is described in Figure 2.9.

begin $InsertConstraint(c)$
  find an edge e such that $e.Org = c.Org$
  while $\neg(e.OnLeft(c.Dest) \vee \neg e.Onext.OnLeft(c.Dest))$
    $e \leftarrow e.Onext$
  $e_0 \leftarrow e$
  do
    $f \leftarrow e.Lnext$
    while $c.OnLeft(f.Dest)$ **do**
      $f.Remove$
      $f \leftarrow e.Lnext$
    $e \leftarrow f$
  while $e.Dest \neq c.Dest$
  $e \leftarrow e.Lnext$
  $c.Splice(e_0)$
  $c.Sym.Splice(e)$
  retriangulate the hole on the left of c
  retriangulate the hole on the right of c
end

Figure 2.9: The $InsertConstraint$ procedure

Chapter 3

A World of Particles

This chapter describes a system of interacting particles that is used to uniformly spread points over a domain. At any time the set of points defined by the positions of the particles is triangulated using the Delaunay criterion. The goal is to generate a particle distribution such that every edge in the mesh has a length equal to $\hat{\sigma}$, which is a constant. As this optimality criterion is usually not achievable because of constraints at the boundaries of the domain, we try to get as close to it as possible. A particle interaction scheme is shaped according to following objectives:

- Simplicity. The system should be governed by a small set of simple rules.
- Speed. The computational cost for performing one time step should be low ($O(1)$). Interactions should be local, so that each particle directly interacts with only a small neighborhood. This neighborhood can be defined based on the topology of the triangulation. Also the convergence should be fast. A minimal amounts of time steps should be taken before the system reaches equilibrium.
- Robustness. The algorithm should always converge.
- Quality. The resulting mesh should locally match the desired edge length. For now we will consider this length equal to $\hat{\sigma}$, which is defined by the user. In a more sophisticated scheme, the implicit geometric feature size [25] could also be considered.

The chapter is structured as follows. Section 3.1 introduces the general dynamics of the interaction model. Section 3.2 defines the interaction neighborhood. Section 3.3 presents different potential functions. Section 3.4 defines a model for controlling the particle population. Section 3.5 presents some improvements regarding speed. Finally, results are shown in Section 3.6.

3.1 Interaction Model

Whereas in Nature and in some previously implemented models [27, 29], the motion of the particles is defined by a second order differential equation ($m\ddot{x} = \sum_i F^i(x, \dot{x})$), we for now consider a simpler, first order model, as used in [34]. Although first order models are more likely to get stuck in local minima, they are generally numerically more stable. The motion of the particles can be expressed as

$$\frac{dx}{dt} = \nabla_x E + C \tag{3.1}$$

where x is the $2n$-vector containing the 2-dimensional positions of all n particles, $\nabla_x E$ the gradient of the potential energy field, and C the set of forces imposed by domain boundary conditions.

The behavior of the system highly depends on the choice of the energy potential function E, which can be expressed as the sum of the potential energies between each pair of particles:

$$E = \sum_i \sum_j E^{ij} \tag{3.2}$$

where E^{ij} is the potential energy of particle i due to particle j. $E^{ij} = \hat{\sigma}^2 \Phi(d^{ij}/\hat{\sigma})$, where Φ is the potential function, depends only on the distance d^{ij} between particles i and j, and the desired edge length $\hat{\sigma}$, thus

$$\nabla_{x^k} E^{ij} \equiv 0 \text{ if } i \neq k \text{ and } j \neq k \tag{3.3}$$

and

$$\nabla_{x^i} E^{ij} = -\frac{\hat{\sigma}}{d^{ij}} \Phi' \left(\frac{d^{ij}}{\hat{\sigma}} \right) r^{ij} \tag{3.4}$$

where $r^{ij} = x^j - x^i$ is defined as the displacement vector between particles i and j, x^k being the position of particle k.

Symmetry is also a desirable feature, thus

$$E^{ij} = E^{ji} \tag{3.5}$$

Finally, interactions are local. Particle i only interacts with a neighborhood N^i, thus

$$E^{ij} \equiv 0 \text{ if } j \notin N^i \tag{3.6}$$

Now that the properties of the energy functions are better defined, it is possible to say more about its gradient:

$$\begin{aligned}
\nabla_{x^i} E &= \nabla_{x^i} \sum_j \sum_k E^{jk} \\
&= \sum_j \sum_k \nabla_{x^i} E^{jk} \\
&= \sum_j \nabla_{x^i} E^{ji} + \sum_k \nabla_{x^i} E^{ik} \\
&= \sum_j (\nabla_{x^i} E^{ji} + \nabla_{x^i} E^{ij})
\end{aligned}$$

and since $E^{ij} = E^{ji}$:

$$\nabla_{x^i} E = 2 \cdot \sum_j \nabla_{x^i} E^{ij} \tag{3.7}$$

Also, particle i only interacts with particles within a neighborhood N^i:

$$\nabla_{x^i} E = 2 \cdot \sum_{j \in N^i} \nabla_{x^i} E^{ij} = -2 \sum_{j \in N^i} \frac{\hat{\sigma}}{d^{ij}} \Phi' \left(\frac{d^{ij}}{\hat{\sigma}} \right) r^{ij} \tag{3.8}$$

To simplify the above expression, we define $\lambda^{ij} = d^{ij}/\hat{\sigma}$ as the "normalized" distance between particles i and j:

$$\nabla_{x^i} E = -2 \sum_{j \in N^i} \frac{\Phi'(\lambda^{ij})}{\lambda^{ij}} \cdot r^{ij} \tag{3.9}$$

Several potential functions Φ are discussed in Section 3.3.

3.1.1 Boundary Conditions

Boundary condition forces are necessary to make sure that no particle moves outside the domain. When a particle i is on a constrained vertex, its motion should be zero, thus $C^i = -\nabla_{x^i} E$. If it lies on a domain boundary, motion should be constrained to the edge it lies on, thus $C^i = -(n \cdot \nabla_{x^i} E) \cdot n$ where n is a unit vector normal to the boundary. For all other particles $C^i = 0$.

3.1.2 Numerical Resolution

To solve the motion equation 3.1, numerical integration can be achieved with Euler's method:

$$x(t + \Delta t) = x(t) + \Delta t \cdot (\nabla_x E + C) \tag{3.10}$$

The notion of time can be eliminated and the process expressed in a more algorithmic fashion as

$$x \leftarrow x + \delta \cdot (\nabla_x E + C) \tag{3.11}$$

Thus for each particle i we have

$$x^i \leftarrow x^i + \delta \cdot (\nabla_{x^i} E + C^i) \tag{3.12}$$

3.1.3 Asynchronous Updating

Although it is common usage to update the positions of the particles synchronously, we proceed in an asynchronous manner:

loop
  randomly pick a particle i
  compute $\nabla_{x^i} E$ and C^i
  $x^i \leftarrow x^i + \delta \cdot (\nabla_{x^i} E + C^i)$
  update the triangulation

where the triangulation update step is performed by first removing site i and then reinserting it at its new position (see Section 2.5 on moving sites). Most of the time the topology of the mesh remains unchanged, sometimes the motion of the particle can result in the swapping of one or several edges in the triangulation.

This asynchronous scheme has several advantages over a synchronous one, namely

- it introduces some noise into the system and reduces the chances of reaching a local minimum
- it opens the possibility of biasing the pick and accelerating the convergence (more on this in Section 3.5)

3.1.4 Second Order Model

Although first order integration is nice and simple, its performance can be poor, as the system often converges to a not so good local minimum. To overcome this problem, a second order model is introduced. Its formulation is rather non-standard but it is easy to implement, and doesn't seem to suffer from numerical instabilities.

$$\begin{aligned} v^i &\leftarrow \beta \cdot v^i + \nabla_x E + C \\ x^i &\leftarrow x^i + \delta \cdot v^i \end{aligned} \tag{3.13}$$

The velocity vector v^i can be seen as a buffer which accumulates the forces, and which decays exponentially. The differential equation associated with this model is of the form

$$\ddot{x} + k \cdot \dot{x} = \nabla_x E + C$$

3.2 Interaction Neighborhoods

In previous work [34, 27, 30, 29] the interaction neighborhood of a particle i was defined based on geometrical properties. The neighborhood was defined by a sphere with a radius equal to some constant times the desired edge length. The set of particles N^i interacting with a particle i was expressed as

$$N^i_{c\hat{\sigma}} = \{j \mid i \neq j,\ d^{ij} < c\hat{\sigma}\}$$

where d^{ij} is the distance between particles i and j, and c is a constant, typically between 1.5 and 2. For fast location of the particles within $N^i_{c\hat{\sigma}}$, $k-d$ trees [29], buckets or some other additional data structures are used. In the present case, however, a triangulation is already provided, and it is thus easy to define a neighborhood based on topology instead of geometry. The distance between two particles i and j is then defined by the number of edges in the shortest path between i and j in the triangulation. The set of particles N^i interacting with a particle i becomes

$$N^i_k = \{j \mid i \neq j,\ \exists \text{path between } i \text{ and } j \text{ of length} \leq k\}$$

In practice small neighborhoods are desirable, so only path lengths of $k = 1, 2$ are considered. Using the quadedge terminology from Chapter 2, these neighborhoods are (also see Figure 3.1):

$$N^i_1 = \{j \mid \exists e : e.Org = i,\ e.Dest = j\} \tag{3.14}$$
$$N^i_2 = \{j \mid i \neq j, \exists e, f : e.Org = i,\ e.Dest = f.Org,\ f.Dest = j\} \tag{3.15}$$

The size of the neighborhoods are typically 6 for N_1, and 18 for N_2. Since they are both $O(1)$ in size, an update operation as described in Section 3.1.3 requires only $O(1)$ time, whereas it would take $O(n)$ if all particles interacted with all others.

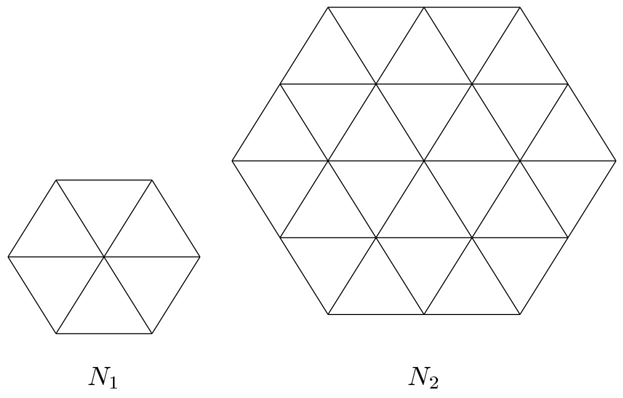


Figure 3.1: Two different sizes of neighborhoods

It is of course also possible to define a neighborhood based on geometrical *and* topological properties.

3.3 Potential Functions

This section presents several potential functions that have been used in the past, but also introduces new ones. All potential functions should fulfill the following desirable properties:

- C^1 continuity. The function and its first derivative should be continuous functions
- no singularities. The function should have finite values for any finite inter-particle distance.

3.3.1 Requirements at Equilibrium

The optimal particle arrangement is a regular array of equilateral triangles with edge length of $\hat{\sigma}$. The forces between particles should be defined so that this pattern is an equilibrium. Moreover the equilibrium should be stable, that is, if the positions of the particles are slightly altered, they should move back to the optimum.

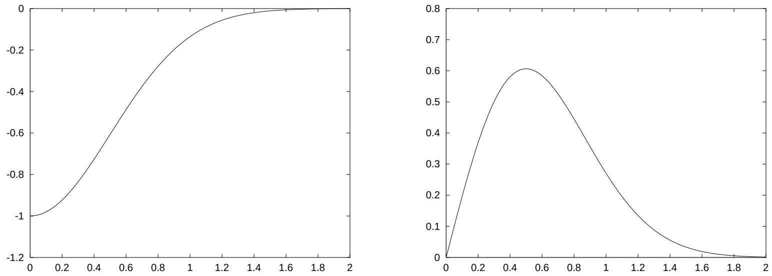
These considerations should also hold for particles near or on the boundaries of the domain. If particles on the boundaries are constrained to stay on it, no problem arises. But particles which are one edge length away from the boundary are not at equilibrium if the neighborhood is N_2 or larger: if the particles repel each other at distances larger than $\hat{\sigma}$, they tend to drift towards the boundary, and in the reverse direction if the interaction is attractive. Thus, to guarantee optimality near the boundaries the interaction neighborhood should be N_1.

On the other hand N_2 neighborhoods are more powerful at organizing particles away from the boundaries. A tradeoff must thus be found between quality near the boundaries and elsewhere. As the effects depend a lot on the choice of the potential energy function, we empirically search for the best solution for each function.

3.3.2 Gaussian Potential

Witkin and Heckbert [34] used a Gaussian potential function to spread particles on implicit surfaces (Figure 3.2):

$$\Phi(\lambda^{ij}) = \exp\left(-2(\lambda^{ij})^2\right)$$



Potential $\Phi(\lambda^{ij})$ Derivative $\Phi'(\lambda^{ij})$

Figure 3.2: Gaussian potential function

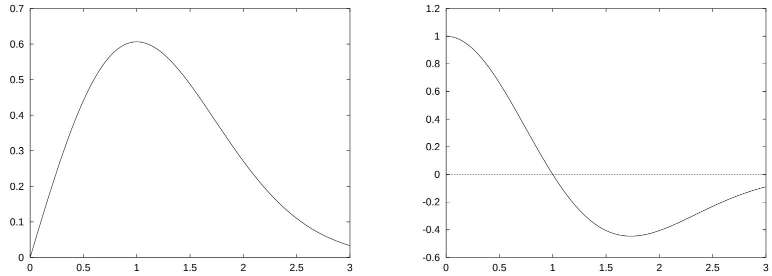
This model works well with N_2 neighborhoods but fails to arrange particles into the desired pattern with N_1. Originally Witkin and Heckbert used $N_{1.5\hat{\sigma}}$ neighborhoods.

Instead of having the potential be a Gaussian function, it is possible to have it be the derivative of a Gaussian (Figure 3.3):

$$\Phi(\lambda^{ij}) = \lambda^{ij} \exp\left(-\frac{(\lambda^{ij})^2}{2}\right) \tag{3.16}$$

of which the derivative is

$$\Phi'(\lambda^{ij}) = (1 - (\lambda^{ij})^2) \exp\left(-\frac{(\lambda^{ij})^2}{2}\right) \tag{3.17}$$



Potential $\Phi(\lambda^{ij})$ Derivative $\Phi'(\lambda^{ij})$

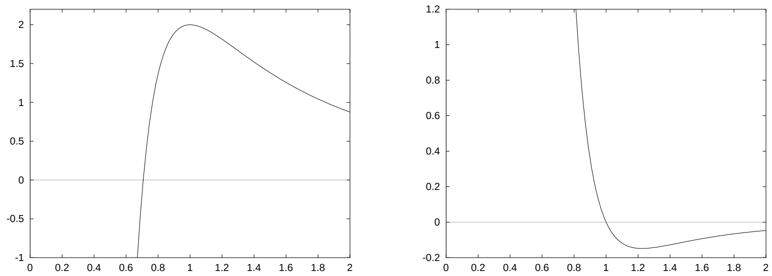
Figure 3.3: Gaussian derivative potential function

This new scheme generates regular patterns for N_1 and N_2 neighborhoods.

3.3.3 Lennard-Jones Potential

One of the most widely used interaction model [29, 27] is based on the Lennard-Jones potential, or van der Waals force which is inspired from molecular chemistry. The potential is defined as (Figure 3.4)

$$\Phi(\lambda^{ij}) = 12(\lambda^{ij})^6 - 6(\lambda^{ij})^{12} \tag{3.18}$$



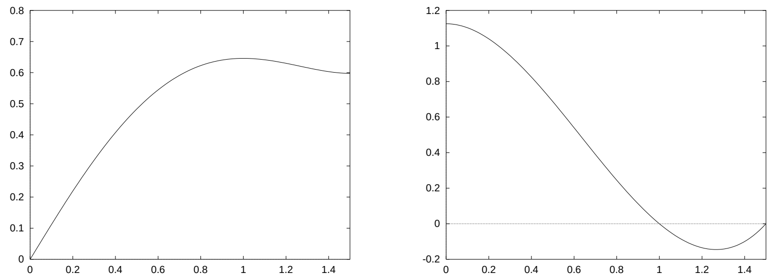
Potential $\Phi(\lambda^{ij})$ Derivative $\Phi'(\lambda^{ij})$

Figure 3.4: Lennard-Jones potential function

The problem with such a model is the singularity at $\lambda^{ij} = 0$ ($\Phi'(0^+) \to \infty$). Numerical integration can thus be disastrous.

Shimada [27] has defined a potential function with a similar shape but which overcomes the singularity problem (Figure 3.5):

$$\Phi'(\lambda^{ij}) = \frac{5}{4}\left(\lambda^{ij}\right)^3 - \frac{19}{8}\left(\lambda^{ij}\right)^2 + \frac{9}{8} \tag{3.19}$$



Potential $\Phi(\lambda^{ij})$ Derivative $\Phi'(\lambda^{ij})$

Figure 3.5: Shimada's potential function

Shimada used $N_{1.5\hat{\sigma}}$, but this scheme also works with N_1 and N_2.

Other approximations of the Lennard-Jones potential function are possible, such as (Figure 3.6)

$$\Phi'(\lambda^{ij}) = (1 - (\lambda^{ij})^4)\exp(-d(\lambda^{ij})^4) \tag{3.20}$$

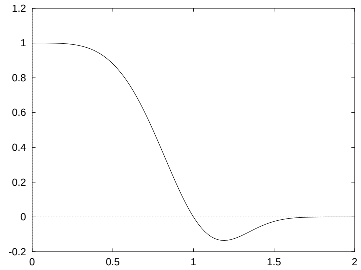


Figure 3.6: New Lennard-Jones approximation

3.3.4 Laplacian Smoothing

Laplacian smoothing [8] is a widely used technique for improving the shape of triangles in a mesh. In particle terminology, Laplacian smoothing consists of moving a particle i to the centroid of its neighbors. Its displacement can be expressed as

$$\Delta x^i = \left(\frac{1}{|N_1^i|} \sum_{j \in N_1^i} x^j \right) - x^i = \frac{1}{|N_1^i|} \sum_{j \in N_1^i} (x^j - x^i) = \frac{1}{|N_1^i|} \sum_{j \in N_1^i} r^{ij}$$

As the displacement is proportional to the derivative of the potential, the latter can be defined as (Figure 3.7)

$$\Phi'(\lambda^{ij}) = -\frac{1}{2} \tag{3.21}$$

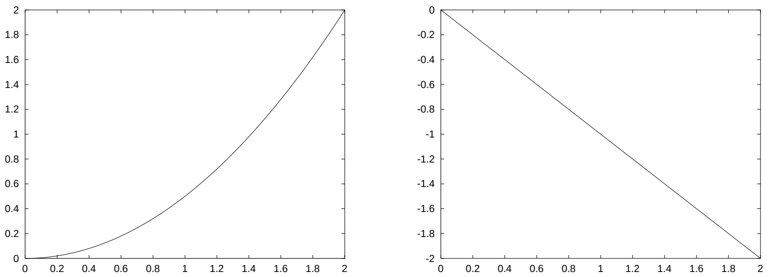


Figure 3.7: Laplacian potential function

3.3.5 Error Potentials

Another way of putting the problem is to define a quality measure of the mesh and consider the particle interaction as an optimization process by gradient descent.

We here define two quality measures. The first one is based on the difference between the actual and the optimal distances between particles (Figure 3.8):

$$\Phi(\lambda^{ij}) = -\frac{1}{2}(\lambda^{ij} - 1)^2 \tag{3.22}$$

of which the derivative is

$$\Phi'(\lambda^{ij}) = 1 - \lambda^{ij} \tag{3.23}$$

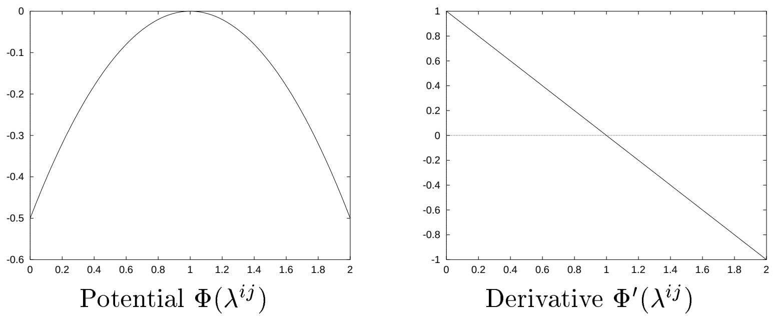


Figure 3.8: Error potential function

The second quality measure is based on the ratio of the distances (Figure 3.9):

$$\Phi(\lambda^{ij}) = -\frac{1}{2}\log^2 \lambda^{ij} \tag{3.24}$$

Although this definition can seem obscure at first, one can easily verify that error is the same for $d^{ij} = \beta \cdot \hat{\sigma}$ and $d^{ij} = \frac{1}{\beta} \cdot \hat{\sigma}$, that is, edges that are too long or too short by a factor β are equally penalized. The derivative of the potential is:

$$\Phi'(\lambda^{ij}) = \frac{\log \lambda^{ij}}{\lambda^{ij}} \tag{3.25}$$

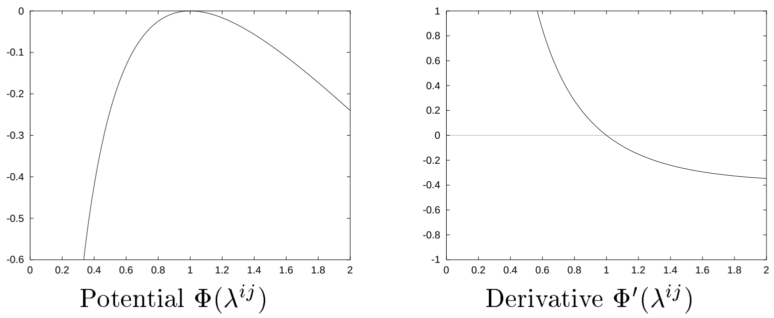


Figure 3.9: Error potential function

The natural interaction neighborhood of these potentials is N_1.

3.3.6 Which is best?

Empirical testing has led to the following conclusions:

- models that are attractive and repulsive perform better than the ones that are attractive only or repulsive only
- the N_2 neighborhood does a better job at arranging particles to create equilateral triangle away from the boundaries...
- ...but performs rather poorly near the boundaries

In practice a Lennard-Jones like scheme, such as the proposed approximations, seems best.

3.4 Adaptive Population Control

Generally it is not known a priori how many particles are needed for the edges to conform the desired size. The number could be determined from the area of the domain, but there are several disadvantages to this strategy:

- when generalizing to surfaces in 3-D or to a non-uniform desired edge length, it is difficult to correctly estimate the area.
- even if one knew how many particles are needed to optimally fit the domain, the question would remain of where to put them. If the initial configuration is poor, it can take a long time for the system to reach a reasonably good configuration.

An alternative strategy is an adaptive scheme where new particles are created where their density is too low, and others are annihilated where too high. The problem thus becomes to evaluate particle density. In a first step we will consider the 1-dimensional case, and then generalize to two dimensions.

3.4.1 1-D Algorithm

Let S be a set of edges, and $L(S)$ the sum of the normalized lengths[1] of each segment in S. Ideally $L(S)$ should be equal to $\hat{L}(S) = |S|$.

The particle density of S is defined as $\delta(S) = \hat{L}(S)/L(S)$. The density error measure associated with S is defined to be

$$\varepsilon(S) = \begin{cases} \delta(S) & \text{if } \delta(S) > 1 \\ 1/\delta(S) & \text{otherwise} \end{cases} \tag{3.26}$$

Creation Rule

Let S be composed of one edge e, and S' the set of the two halves of e after a particle has been inserted at its middle point. A particle is inserted if $\varepsilon(S') < \varepsilon(S)$. We know that $L(S) = L(S') = ||e||$, $|S| = 1$, and $|S'| = 2$. Thus a new particle should be inserted if $||e|| > \sqrt{2}$, where $||e||$ is the normalized length of edge e.

[1] the normalized length is defined to be the actual length divided by $\hat{\sigma}$

Annihilation Rule

Let S be the two edges e_1 and e_2 on each side of a particle p. Let S' be a set containing the concatenation e of e_1 and e_2 after removal of p. Particle p is annihilated if $\varepsilon(S') < \varepsilon(S)$. We know that $L(S) = L(S') = ||e_1|| + ||e_2||$, $|S| = 2$, $|S'| = 1$. Thus particle p should be annihilated if $||e_1|| + ||e_2|| < \sqrt{2}$, where $||e_i||$ is the normalized length of edge e_i.

3.4.2 2-D Algorithm

Let T be a set of triangles, and $A(T)$ the sum of the normalized areas[2] of each triangle in T. Ideally $A(T)$ should be equal to $\hat{A}(T) = |T|\triangle$, where $\triangle = \sqrt{3}/4$ is the area of a unit equilateral triangle.

The particle density of T is defined as $\delta(T) = \hat{A}(T)/A(T)$. The density error measure associated with T is defined to be

$$\varepsilon(T) = \begin{cases} \delta(T) & \text{if } \delta(T) > 1 \\ 1/\delta(T) & \text{otherwise} \end{cases} \tag{3.27}$$

Creation Rule

Let T be the set of the triangles on each side of an edge e, and T' the set of triangles incident to the two halves of e after a particle has been inserted at its middlepoint. A particle is inserted if $\varepsilon(T') < \varepsilon(T)$. We know that $A(T) = A(T')$, $|T| = 2$, and $|T'| = 4$. Thus a new particle should be inserted on e if $A(T) > \sqrt{3/2}$.

A more sophisticated rule takes into consideration the local optimization of the topology (Delaunay triangulation) after insertion. The average degree of a node inside a mesh is 6, and $|T'|$ should thus be 6. Since it would be costly to compute the areas of the triangles after the local optimization, we consider two virtual triangles which are affected by the edge swapping. Let U be the set T augmented by two virtual triangles of ideal size, and U' the set U augmented by the two same virtual triangles. We know $A(U) = A(U') = A(T) + 2\triangle$, $|U| = 4$, and $|U'| = 6$. Thus a new particle should be inserted on e if $A(T) > 3/\sqrt{2} - \sqrt{3}/2$.

Annihilation Rule

Let T be the set of triangles incident to a particle p. Let T' be set of triangles resulting from the retriangulation of the region covered by T. Clearly p should be annihilated if $\varepsilon(T') < \varepsilon(T)$. We know that $A(T') = A(T)$ and $|T'| = |T| - 2$. Thus particle p should be annihilated if $A(T) < \sqrt{3|T|(|T|-2)}/4$.

3.4.3 Combination of 1-D and 2-D rules

The rules that are actually used in our algorithm are a combination of 1-dimensional and 2-dimensional rules. Let i be the particle that is picked at the given step.

First, the annihilation rules are considered. If the particle i lies on a boundary, then the 1-D rule in applied. Otherwise the 2-D rule is applied.

If the particle is not annihilated, the creation rules are considered for all the edges that are connected to particle i. If several edges are candidates for insertion, the one that maximizes the improvement in particle density is selected for splitting.

[2] the normalized area is defined to be the actual area divided by $\hat{\sigma}^2$

3.4.4 Initial Population

The initial population corresponds to the sites of the PSLG defining the domain. These particles are never moved or annihilated so that the triangulation always remains conforming.

3.5 Speeding up the Process

Since the particles are moved one by one, it is possible to define a heuristic so that the ones that "need" to be moved first are indeed moved first. The introduction of an *alive* flag for each particle can lead to a simple heuristic:

- all the particles are initially alive
- when an alive particle p is picked and updated, if its normalized speed $|v^p|/\hat{\sigma}$ is larger than a threshold, all of its dead neighbors become alive. If the speed is too low, p is marked dead.
- when a particle p is annihilated, all of its dead neighbors become alive.
- when a particle p is created, p and all of its dead neighbors become alive.
- a dead particle is never picked. Consequently, particles don't move while they are dead.

More complex schemes have been tested such as assigning picking probabilities to each particle proportional to their speed or their total energy (kinetic and potential). Preliminary results have shown no improvement over the simple scheme. Moreover the computational cost of such a method is high (time cost for one step would be $O(\log n)$ instead of $O(1)$).

3.5.1 Ending the simulation

The heuristic defined in the previous section allows a very simple test to end the simulation: end when no more particles are alive.

3.6 Results

In this section we briefly present results for a simple mesh. The domain that is meshed is a unit square, and the desired edge length is set to $\hat{\sigma} = 0.02$. The mesh we have obtained with an approximated Lennard-Jones potential (Equation 3.20) is depicted in Figure 3.10.

The angle and normalized edge length histograms show that the elements are close to optimality (i.e. 60 degree angles, and unit normalized edge length).

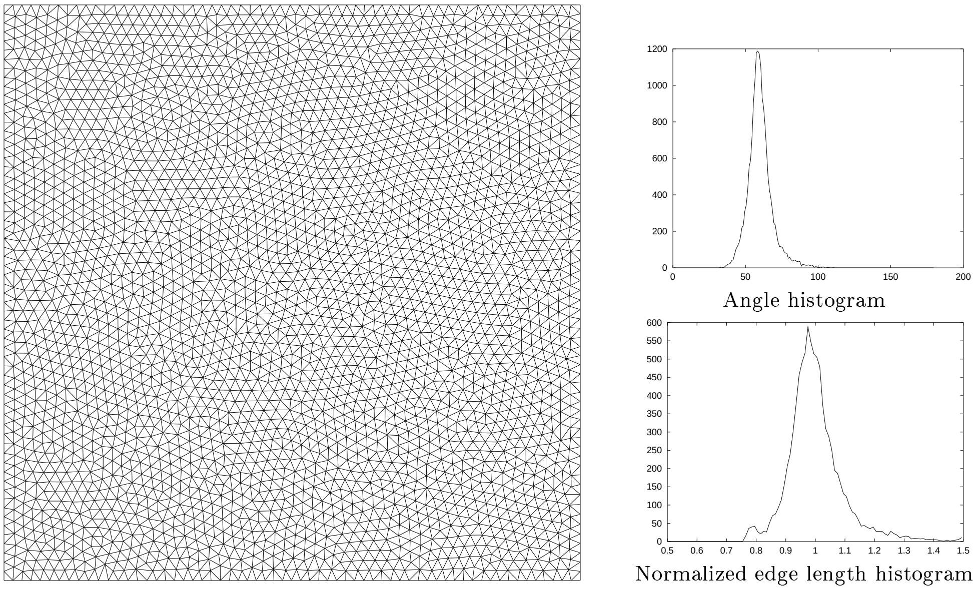


Figure 3.10: A simple mesh

Chapter 4

Interlude: Delaunay Triangulations with Java

Many people are unfamiliar with computational geometry and Delaunay triangulations. The World Wide Web appears to be a good tool to familiarize those people with these concepts. And as learning is often easier when playing games, we have designed an interactive Delaunay triangulator using the Java programming language developed by Sun [12]. It allows all people using a Java enabled browser such as Netscape Navigator™ 2.0 to build a triangulation by interactively inserting and removing sites. The triangulator can be found at:

`http://www.cs.cmu.edu/~bossen/triangulator.html`

Complete user instructions are given on the web page.

The algorithm that has been implemented is the incremental one. When the user adds a new point, it can simply be inserted into the triangulation. When a point is removed the whole triangulation is rebuilt[1]. The quadedge data structure has been used to represent the triangulation. This choice is justified by the fact that the Voronoi diagram is also computed and displayed.

[1]Although not very efficient, this solution was easier to implement

Chapter 5

Anisotropic Meshes

In this chapter we generalize the particle-based mesh generation method described in Chapter 3 to nonuniform, isotropic and anisotropic, meshes. The feature size is no longer a constant $\hat{\sigma}$, but is position–dependent and direction–dependent.

In strongly directional phenomena such as shocks and limit layers in fluid flows, the elements should not only adapt in size but also in shape. They should be stretched in the direction of the flow. The desired edge length depends on its orientation, hence the name anisotropy. A metric tensor is introduced to quantify the stretching of the triangles at every point in the domain. It reflects the curvature of the function that is approximated.

Relatively little work has been done on anisotropic meshes. D'Azevedo [5] has proposed the use of coordinate transformations to generate optimal triangulations. Although very elegant, his method is limited to metric tensors that represent a flat space (the Riemann-Christoffel tensor [28], that is the curvature of the space represented by the metric, should be zero). Another weakness of his method is poor meshing at boundaries.

Peraire et al. [23], and later Mavriplis [22] introduced stretching vectors to quantify the anisotropy, which really are eigenvectors of the metric tensor, but never formally introduced the latter. Greedy insertion algorithms have been proposed by researchers at INRIA [3, 31]. We believe that better node positioning can be achieved.

This chapter is structured as follows. Section 5.1 introduces some concepts of Riemannian geometry. Section 5.2 exposes the changes made to the triangulation procedure due to anisotropy. Section 5.3 presents an anisotropic version of the particle interaction model described earlier. Section 5.5 summarizes the meshing algorithm. Section 5.4 presents background meshes which are convenient for representing the metric. Section 5.6 shows how to derive the metric M for a given function f that the mesh should adapt to. Section 5.7 shows how the mesh can incrementally be adapted to a function when no *a priori* knowledge of the solution is available.

5.1 Riemannian Geometry

In Euclidean geometry, the definition of distances is isotropic and very simple. Riemannian geometry describes "warped" or curved spaces. In Riemannian geometry, the anisotropy of distance is defined by a metric tensor M. This tensor quantifies the desired stretching of the triangles in the mesh. M is a symmetric 2 by 2 matrix, and both of its eigenvalues λ_1 and λ_2 are positive:

$$M = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{5.1}$$

If both the eigenvalues are equal to 1, then $M = I$ and the space is Euclidean. If M is considered locally constant, the unit ball is the ellipse p$x^T M x = 1$. The first axis of the ellipse has length $r_1 = 1/\sqrt{\lambda_1}$, and makes an angle θ with any horizontal line. The second axis has length $r_2 = 1/\sqrt{\lambda_2}$ (Figure 5.1).

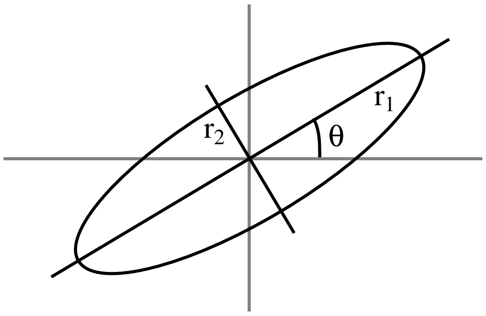


Figure 5.1: Elliptic unit ball

In Riemannian geometry, the basic geometrical operators are redefined. The dot product becomes

$$a^T M\ b \tag{5.2}$$

and the cross product

$$\sqrt{\det M}(a \times b) \tag{5.3}$$

where $a \times b = a_1 b_2 - a_2 b_1$ is the 2-dimensional cross product.

5.1.1 Computing Distances

In Riemannian geometry, the length of a parametric curve $\Gamma(t)$ between points i and j, where $t \in [0,1]$, $\Gamma(0) = x^i$, and $\Gamma(1) = x^j$ is defined as

$$\ell(\Gamma) = \int_0^1 \sqrt{\dot{\Gamma}(t)^T M(\Gamma(t))\dot{\Gamma}(t)}\ dt \tag{5.4}$$

where $\dot{\Gamma} = \frac{d\Gamma}{dt}$. The distance between two points is the length of the shortest parametric curve Γ. Such a curve is called a geodesic. Computing the geodesic for an arbitrary metric is not trivial. It is simpler to approximate the distance by computing the length of a simple, non-guaranteed shortest, path. Integrating along the straight line $\Gamma(t) = x^i + t \cdot (x^j - x^i)$ seems reasonable. If M varies linearly between i and j then the distance between the two points can be defined as

$$\begin{aligned} d^{ij} &= \int_0^1 \sqrt{r^{ijT}(M^i + t(M^j - M^i))r^{ij}}\ dt \\ &= \int_0^1 \sqrt{(d_i^{ij})^2 + t((d_j^{ij})^2 - (d_i^{ij})^2)}\ dt \end{aligned} \tag{5.5}$$

where $d_k^{ij} = \sqrt{r^{ijT} M^k r^{ij}}$ is the distance as "seen" from point k. With appropriate variable changes, the integral solves to

$$d^{ij} = \frac{2}{3} \frac{(d_i^{ij})^2 + d_i^{ij} d_j^{ij} + (d_j^{ij})^2}{d_i^{ij} + d_j^{ij}} \tag{5.6}$$

Assuming that particles i and j are close to each other, and that the metric is slowly changing, cruder approximations of the distance are also possible, namely by simply averaging the distances as "seen" from i and j

$$d^{ij} \approx \frac{d_i^{ij} + d_j^{ij}}{2} \tag{5.7}$$

or by averaging the metrics as

$$d^{ij} \approx \sqrt{r^{ijT} M^{ij} r^{ij}} \text{ , with } M^{ij} = \frac{M^i + M^j}{2} \tag{5.8}$$

From the three proposed approximations the last is the fastest to compute since it requires only one square root operation. Also finding the squared distance doesn't require any square root operation at all.

5.1.2 Computing Areas

The area of a domain Ω is defined as

$$A(\Omega) = \iint_\Omega \sqrt{\det M(x)}\; dx_1 dx_2 \tag{5.9}$$

Again it would be possible to solve the integral for a triangle $\triangle_{abc}$, knowing the metric at each vertex, and linearly interpolating it in between.

We will not bother to do so, and simply give a cruder approximation:

$$A(\triangle_{abc}) = \frac{1}{2} \sqrt{\det \left(\frac{M^a + M^b + M^c}{3} \right)} \cdot (b-a) \times (c-a) \tag{5.10}$$

5.2 Anisotropic Triangulation

Although it is possible to compute the Delaunay triangulation of an anisotropically distributed point set, it is reasonable to consider anisotropicity in the triangulation process. It has been shown by D'Azevedo [5] that it is better to Delaunay triangulate in a transformed space to minimize errors for function interpolation.

Only simple cases (for example when the metric tensor is constant) allow a global map of the domain into a 2-dimensional Euclidean space by coordinate transformation. In the general case, spaces of higher dimensions are needed. To solve this problem, a simple approximation is used. It is considered that the metric is locally constant when performing the circumcircle test. This local constant metric is computed by averaging the metric at the four points considered for the test:

$$M = \frac{1}{4} \cdot \sum_{i=1}^{4} M^i \tag{5.11}$$

In Euclidean geometry the *InCircle* test is defined by equation 2.9. Its Riemannian equivalent is:

$$\sqrt{\det M}(a \times b)(c^T M\ d) + (a^T M\ b)\sqrt{\det M}(c \times d) > 0 \tag{5.12}$$

and since $\sqrt{\det M}$ is positive:

$$(a \times b)(c^T M\ d) + (a^T M\ b)(c \times d) > 0 \tag{5.13}$$

5.3 Anisotropic Energy Potential

All the energy potential functions defined in Chapter 3 are in the form

$$E^{ij} = \hat{\sigma}^2 \cdot \Phi\left(\lambda^{ij}\right)$$

and the gradient of which is

$$\nabla_{x^i} E^{ij} = \frac{\Phi'\left(\lambda^{ij}\right)}{\lambda^{ij}} \cdot r^{ij} \tag{5.14}$$

In an anisotropic context the normalized distance λ^{ij} is replaced by d^{ij} as defined in Section 5.1.1. Finding an equivalent for $\hat{\sigma}^2$ is more difficult. In E^{ij} it could be replaced by $1/\sqrt{\det M^i}$, but that would break the symmetry between E^{ij} and E^{ji}. To solve this, we can average $1/\sqrt{\det M^i}$ and $1/\sqrt{\det M^j}$ to get $\frac{\sqrt{\det M^i}+\sqrt{\det M^j}}{2\sqrt{\det M^i}\cdot\sqrt{\det M^j}}$. As this expression is rather complex, in practice we will approximate it by $\kappa = 1/\sqrt{\det M^{ij}}$, where M^{ij} is the average between M^i and M^j.

The potential energy function becomes

$$E^{ij} = \kappa \cdot \Phi(d^{ij})$$

of which the gradient is

$$\nabla_{x^i} E^{ij} = \kappa \cdot \Phi'(d^{ij}) \cdot \nabla_{x^i} d^{ij} + \Phi(d^{ij}) \cdot \nabla_{x^i} \kappa$$

where $\nabla_{x^i} d^{ij}$ is quite a complex expression, which depends on the derivative of the position of particle i, but also on the derivatives of M. For the sake of simplicity, we will consider that $d^{ij} = \sqrt{r^{ijT} M^{ij} r^{ij}}$, as defined is equation 5.8, and that M^{ij} is locally constant. Following this simplification we have

$$\nabla_{x^i} d^{ij} \approx \frac{M^{ij} r^{ij}}{d^{ij}}$$

Furthermore $\nabla_{x^i} \kappa$ is zero since M^{ij} is considered locally constant. The gradient can thus be rewritten as

$$\nabla_{x^i} E^{ij} \approx \frac{\kappa \cdot \Phi'(d^{ij})}{d^{ij}} \cdot M^{ij} r^{ij} \tag{5.15}$$

It is also possible to simplify the expression of the gradient in a more radical way, by substituting d^{ij} for λ^{ij} in the gradient expression 5.14:

$$\nabla_{x^i} E^{ij} \approx \frac{\Phi'(d^{ij})}{d^{ij}} r^{ij} \tag{5.16}$$

Equations 5.15 and 5.16 are quite similar, but whereas in the latter the gradient of E^{ij} is aligned with r^{ij}, it is usually not the case in the former, where r^{ij} is multiplied by the matrix κM^{ij}. This multiplication has for consequence a precession movement that will tend to align r^{ij} with one of the eigenvectors of M^{ij}. When r^{ij} is parallel to one of the eigenvectors of M^{ij} then the gradient of E^{ij} is aligned with r^{ij}.

In practice, the simpler scheme (Equation 5.16) is used.

5.4 Background Mesh

Although it is possible to explicitly define the metric tensor as a set of functions, it is often preferable to define the metric at given points, and then interpolate in between. A background mesh is used to do so. At each node of the background mesh a metric matrix is defined. The metric at the other points of the domain can be computed by linear interpolation. If M^a, M^b, and M^c are the metric matrices at the vertices of a triangle $\triangle_{abc}$ in which a point p lies, then the metric matrix at p is given by:

$$M^p = \frac{w^{ap} \cdot M^a + w^{bp} \cdot M^b + w^{cp} \cdot M^c}{w^{ap} + w^{bp} + w^{cp}} \tag{5.17}$$

where

$$\begin{aligned} w^{ap} &= (x^p - x^b) \times (x^c - x^b) \\ w^{bp} &= (x^p - x^c) \times (x^a - x^c) \\ w^{cp} &= (x^p - x^a) \times (x^b - x^a) \end{aligned}$$

The triangulation of the set of samples defining the metric can either be given by the user or computed. In the latter case, Delaunay triangulation is used.

To compute the metric at a given point, the triangle it lies in first has to be determined. To do so, it is possible to use data structures such as the history tree proposed in [14]. Although the cost of one location is only $O(\log m)$, where m is the number of samples defining the metric, this method is suboptimal. Since the metric is only evaluated at positions where particles are located, and that particles rarely move across more than one triangle in the background mesh during one time step, it is better to cache the triangle a particle lies in. When a particle is moved, the eventually new triangle it moves into, can quickly be determined by a simple walking method. The metric evaluation cost can thus be reduced to $O(1)$.

A potential drawback of only piecewise linearly defining the metric is that the feature size function only has C^0 continuity. In the present scheme, however, this is not a problem.

5.5 Method Summary

Section 3.1.3 introduced the main loop of the mesh generator. We will now extend it to the general, anisotropic case:

build the constrained Delaunay triangulation of the domain
loop
  randomly pick a particle i that is marked alive
  compute $\nabla_{x^i} E$ and C^i

$x^i \leftarrow x^i + \delta \cdot (\nabla_{x^i} E + C^i)$
compute the areas of triangles incident to i
create/annihilate a particle if areas are too high/low
update the triangulation

where equations 5.15 and 3.20 are used to compute $\nabla_{x^i} E$, and equation 5.10 to compute triangle areas. For distances, either equation 5.8 or 5.6 can be used.

This algorithm has been implemented in C++ on Silicon Graphics workstations. The source code spans over roughly 5000 lines. The OpenGL graphical library was used to display the mesh evolution in real time. All the meshes inserted in this document have been generated by the program, which can directly output encapsulated Postscript files.

5.6 Function Interpolation

Given a function f, the metric that generates a mesh for piecewise linear interpolation such that the RMS error is minimum, is given by the "curvature" of the function. This curvature is equivalent to the Hessian matrix of the function defined as

$$H^f = \begin{pmatrix} \partial^2 f/\partial x_1^2 & \partial^2 f/\partial x_1 \partial x_2 \\ \partial^2 f/\partial x_2 \partial x_1 & \partial^2 f/\partial x_2^2 \end{pmatrix} \tag{5.18}$$

Since H^f is symmetric, it can be represented as

$$H^f = R^T \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} R \tag{5.19}$$

where R is a rotation matrix. Given this decomposition, it is trivial to define a matrix M^f such that both its eigenvalues are positive[1]:

$$M^f = R^T \begin{pmatrix} |\lambda_1| & 0 \\ 0 & |\lambda_2| \end{pmatrix} R \tag{5.20}$$

In practice it is necessary to limit the values of the eigenvalues to a given range. Otherwise the mesh elements could get arbitrary small or large. It has also been proposed [3] to align one of the eigenvectors of the metric matrix with the boundary, when close to one.

5.6.1 Example: a Gaussian function

The function we approximate is

$$f(x, y) = \exp\left(-\frac{x^2 + y^2}{2}\right)$$

of which the Hessian is

$$H^f = \begin{pmatrix} x^2 - 1 & xy \\ xy & y^2 - 1 \end{pmatrix} \cdot f$$

The eigenvalues of H^f are $\lambda_1 = -f$ and $\lambda_2 = (x^2 + y^2 - 1) \cdot f$. The corresponding eigenvectors are $e_1 = \begin{pmatrix} -y/r & x/r \end{pmatrix}^T$ and $e_2 = \begin{pmatrix} x/r & y/r \end{pmatrix}^T$, where $r = \sqrt{x^2 + y^2}$.

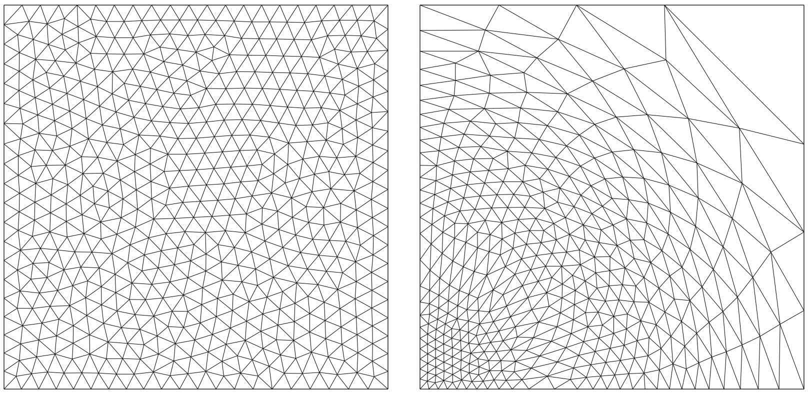
[1] The eigenvalues need to be positive to ensure that the squared distance between any two points is always positive.

Following the rules previously described, we obtain the metric tensor

$$M^f = \begin{pmatrix} y^2 + cx^2 & (c-1)xy \\ (c-1)xy & x^2 + cy^2 \end{pmatrix} \cdot \frac{|f|}{x^2 + y^2}$$

where $c = |x^2 + y^2 - 1|$.

Figure 5.2 shows the mesh obtained with the M^f over the domain $\Omega = [0, 3] \times [0, 3]$.



Unadapted: 572 particles   Adapted: 409 particles

Figure 5.2: Meshes for Gaussian function approximation

Experiments have shown that, for an identical number of particles, the RMS error of the piece-wise linear approximated function is about 8 times smaller for adapted meshes than for unadapted ones.

5.7 Incremental Mesh Adaptation

To generate a good mesh for finite element analysis, one has to have a good knowledge of what the solution is. But what if one doesn't? A solution is to generate a metric by successive refinements. The incremental adaptation loop is

1. start with a uniform feature size function
2. build/update the mesh
3. run the finite element solver with the current mesh
4. estimate the error. If the error is smaller than some amount, then exit, else adapt the metric according to the current solution, and go back to step 2

While the initial mesh is uniform as in Figure 5.3, after several iterations, the metric becomes better adapted to the problem and the quality of the mesh is improved, similarly to Figure 5.4[2].

[2]This figure has been obtained with a hand-defined background mesh. No solver has been used.

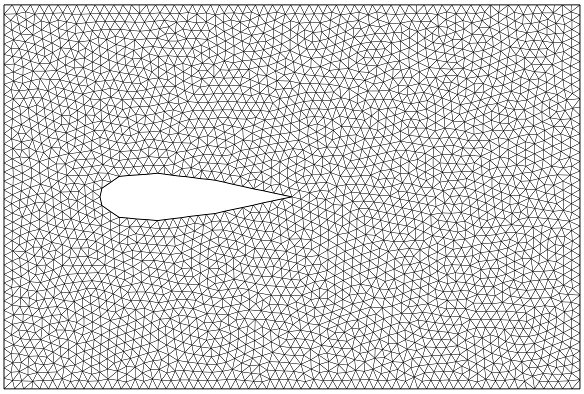


Figure 5.3: Without background mesh: uniform mesh with 2865 particles

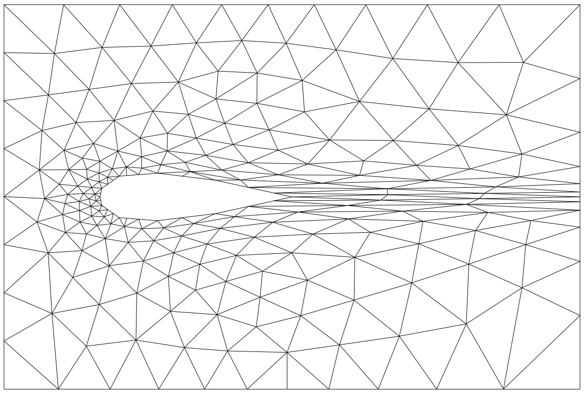


Figure 5.4: With background mesh: anisotropic mesh with 229 particles

5.8 Running Time

To empirically determine the asymptotic running time of the algorithm, we have generated meshes for Gaussian function approximation with different numbers of particles.

Figure 5.6a shows the CPU time[3] required to generate a mesh containing n particles. The running time increases slightly faster than linearly with the number of particles. To find out why this is, we have looked at the growth of the number of steps versus the number of particles. Figure 5.6b shows the number of steps grows less than linearly with the number of particles. On the other hand, the number of steps executed within a second decreases as the number of particles grows. This is in contradiction with the theory, which predicts that the time complexity of one step is $O(1)$. Our guess is that speed performance decreases because the number of cache misses grows. Using profiling, we have found that the subroutine which gets more and more time consuming proportionally to the number of steps, is a quite simple subroutine which does actually only scan through all the edges adjacent to a newly picked

[3]On a 250Mhz Mips R4400 chip

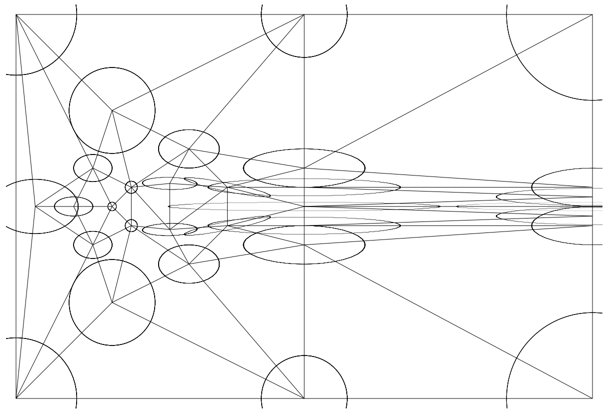
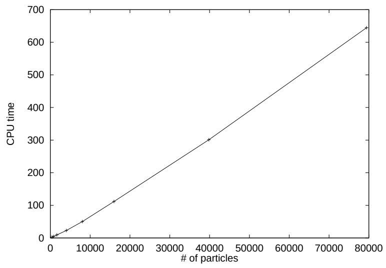
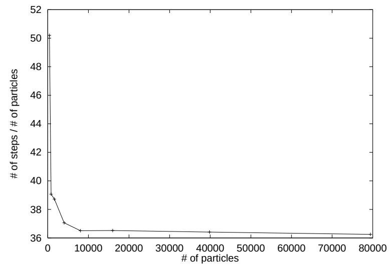


Figure 5.5: Background mesh



a)



b)

Figure 5.6: Running time results

particle i. The reason for apparent sup-linear time complexity is thus cache misses.

Our conclusion is that the asymptotic time complexity of the algorithm is $O(n)$.

Chapter 6

Conclusion

We have presented a new method for generating triangular meshes in a 2-dimensional domain bounded by straight lines. The following considerations have led to the formulation we have given:

- the particle approach seemed promising because it allowed good node placement.
- whereas in previous methods the particle interaction neighborhood was defined by geometrical distances, we have opted for topological distances. This allows for faster neighborhood computation at zero data structure space cost.
- although it is possible to define particle interaction as purely attractive or repulsive, it turned out that mixed attractive/repulsive schemes work better.
- finally anisotropy has been introduced because the previously defined scheme seemed well suited, and the generalization straightforward.

A list of desirable features has been presented in the introduction. We now review it:

- functionality. The new approach works, as the meshes in this document testify.
- robustness. Unfortunately the scheme is not as robust as it could be. Problems can arise when constraints and the feature size function are "incompatible", that is if the distance between two constrained nodes is much shorter than the desired edge length. The system oscillates as particles are continuously created and annihilated. The system can suffer from the same pathology when changes in the metric tensor are too abrupt.
- quality. The quality of the results is good, as the angle and edge length histograms show. The presented approach can also be seen as a mesh postprocessing tool. In practice it has proven superior to Laplacian smoothing. The reasons seem to be twofold: Laplacian smoothing doesn't have any knowledge of what the inter-point distance should be, and maintenance of the Delaunay characteristics after point displacement improves the shape of the mesh elements.
- speed. It is probably not a strength of the presented approach, although the asymptotic running time seems to be $O(n)$, where n is the number of nodes in the mesh.

However, in applications where constant remeshing is required the new approach should perform well.

- minimal user interaction. Several variables controlling the particle interaction model need to be set. Although an experienced user could decide on what values to use, it is possible to come up with a standard set of values, in which case no user input is needed at this point.
- controllability. The introduction of a background mesh seems to give the user sufficient control. If he miscalculates the effect of the linear interpolation, he can always add a new sample to have the mesh better fit his desires.

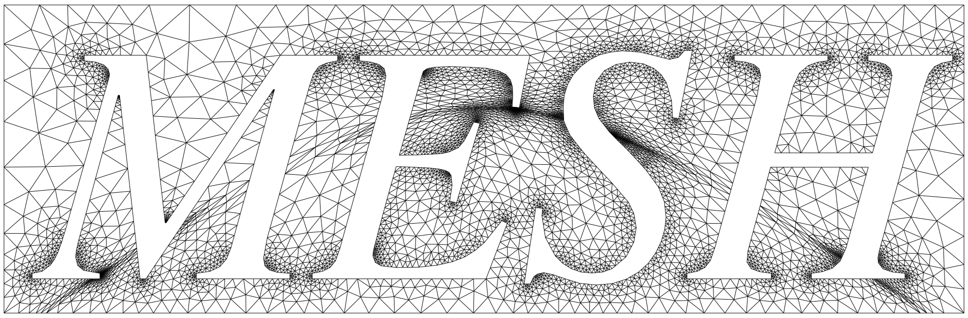


Figure 6.1: A nice anisotropic mesh

6.1 Future work

Directions for future work are many:

- apply the method to solve real problems. The meshes that have been presented here were all generated with hand-made domains and hand-made background meshes. The addition of a finite element solver would help to demonstrate the abilities of this new meshing method to generate adapted meshes.
- generalize to non-polygonal domains. Most objects in the real world cannot be described by a set of line segments. A more powerful scheme, which may include splines, is thus needed.
- generalize to higher dimensions. 3-dimensional meshing is not as well understood as 2-dimensional meshing. Maybe a physically–based approach could solve the sliver problem[1].
- find even better particle interaction schemes. We believe that the proposed interaction scheme is good, but could still be improved, especially on the particle creation/annihilation side. A scheme where particles are inserted at centers of circumcircles could maybe generate better meshes that need less smoothing. On the annihilation side, collapsing edges might be better than removing nodes.

[1]A sliver an ill-shaped tetrahedron of which the four vertices are almost co-planar

- introduce an *a priori* feature size function based on geometry when none is provided. This would probably make the scheme more robust (see previous remark on robustness).

- find out how much remeshing from a previous solution is better (same mesh quality at lower computational cost) than remeshing from scratch.

6.2 Acknowledgements

I would like to thank Paul Heckbert for inviting me to Carnegie Mellon, and for being a wonderful advisor to work with; Marshall Bern, Michael Garland, Michael Erdmann, Andy Witkin, Jim Winget, Scott Canann, and Paul Heckbert for their helpful comments and suggestions; and my father who has been supporting me for more than 23 years now.

Bibliography

[1] Eric B. Becker, Graham F. Cary, and J. Tinsley Oden. *Finite Elements: An Introduction*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[2] Marshall Bern and David Eppstein. Mesh generation and optimal triangulation. Technical Report P92-00047, Xerox PARC, 1992.

[3] M.J. Castro-Diaz, F. Hecht, and B. Mohammadi. New progress in anisotropic grid adaptation for inviscid and viscous flows simulations. In *4th Annual International Meshing Roundtable*, 1995.

[4] L. Paul Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[5] E.F. D'Azevedo. Optimal triangular mesh generation by coordinate transformation. *J. Sci. Stat. Comput.*, 12(4):755–786, July 1991.

[6] L. Devroye, E.P. Mücke, and B. Zhu. A note on point location in Delaunay triangulations of random points. Submited for publication, 1995.

[7] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Disc. and Comp. Geom.*, 8(1):25–44, 1986.

[8] D.A. Field. Laplacian smoothing and Delaunay triangulations. *Comm. Appl. Num. Methods*, 4:709–712, 1988.

[9] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[10] William H. Frey and David A. Field. Mesh relaxation: A new technique for improving triangulations. *International Journal for Numerical Methods in Engineering*, 31:1121–1133, 1991.

[11] N.A. Golias and T.D. Tsiboukis. An approach to refining three-dimensional tetrahedral meshes based on delaunay transformations. *International Journal for Numerical Methods in Engineering*, 37:793–812, 1994.

[12] James Gosling and Henry McGilton. *The Java Language Environment: a white paper.* Sun Microsystems, October 1995.

[13] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[14] L.J. Guibas, D.E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. Technical Report STAN-CS-90-1300, Stanford University, 1990.

[15] K. Ho-Le. Finite element mesh generation methods: a review and classification. *Computer–Aided Design*, 20(1):27–38, Jan/Feb 1988.

[16] Thomas Kao and David M. Mount. Dynamic maintenance of Delaunay triangulations. Technical Report CS-TR-2585, University of Maryland, 1991.

[17] C. L. Lawson. Software for c^1 surface interpolation. In John R. Rice, editor, *Mathematical Software III*, pages 161–194. Academic Press, 1977.

[18] Geoff Leach. Improving worst-case optimal Delaunay triangulation algorithms. In *4th Canadian Conference on Computational Geometry*, 1992.

[19] D.A. Lindholm. Automatic triangular mesh generation on surfaces of polyhedra. *IEEE Trans. Magnetics*, 19:2539–2542, 1983.

[20] Dani Lischinski. Incremental Delaunay triangulations. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 47–59. Academic Press, 1994.

[21] S.H. Lo. A new mesh generation scheme for arbitrary planar domains. *International Jounral for Numerical Methods in Engineering*, 21:1403–1426, 1985.

[22] Dimitri J. Mavriplis. Adaptive mesh generation for viscous flows using Delaunay triangulation. *Journal of Computational Physics*, 90(2):271–291, October 1990.

[23] J. Peraire, M. Vahdati, K. Morgan, and O.C. Zienkiewicz. Adaptive remeshing for compressible flow computations. *Journal of Computational Physics*, 72:449–466, 1987.

[24] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, 1985.

[25] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *4th ACM-SIAM Symp. on Disc. Algorithms*, pages 83–92, 1993.

[26] Kenji Shimada. *Physically-Based Mesh Generation: Automated Triangulation of Surfaces and Volumes via Bubble Packing.* PhD thesis, MIT, 1993.

[27] Kenji Shimada and David C. Gossard. Computational methods for physically-based fe mesh generation. In *PROLAMAT.* IFIP TC5/WG5.3, 1992.

[28] I.S. Sokolnikoff. *Tensor Analysis, Theory and Applications to Geometry and Mechanics of Continua.* John Wiley, New York, 2nd edition, 1964.

[29] Richard Szeliski and David Tonnesen. Surface modeling with oriented particle systems. In *SIGGRAPH'92 Proceedings*, pages 185–194, 1992.

[30] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. In *SIGGRAPH'91 Proceedings*, pages 289–298, 1991.

[31] Marie-Gabrielle Vallet. Generation de maillages anisotropes adaptes - application a la capture de couches limites. Technical Report 1360, INRIA, 1990.

[32] N.P. Weatherill and O. Hassan. Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints. *International Journal for Numerical Methods in Engineering*, 37:2005–2039, 1994.

[33] William Welch. *Serious Putty: Topological Design for Variational Curves and Surfaces.* PhD thesis, Carnegie Mellon University, 1995.

[34] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH'94 Proceedings*, 1994.

[35] M.A. Yerry and M.S. Shepard. A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics and Applications*, 3:39–46, January/February 1983.