EikMesh

A Java library for 2D unstructured triangular meshes

Benedikt Zönnchen

June 18, 2019

Contents

1 Introduction

Computational modeling and simulation is a critical aspect of modern science and industrial design, underpinning a diverse range of application - from numerical simulations of physical phenomena, such as fluid dynamics to crowd simulation to computer graphics and animation. The basis of these simulations is a discrete representation of the underlying geometry, i. e. spatial domain. Modern mesh generation is concerned with the development of efficient and automatic algorithms to construct and maintain high-quality meshes for complex spatial domains.

EikMesh focuses on unstructured meshes which are highly flexible and therefore, require less elements, compared to structured meshes. Since unstructured meshes can be highly irregular, they require more memory per element (vertex, face, edge) because the connectivity of elements can not be saved implicitly. Regardless of the higher memory consumption per element, the overall memory is often significantly lower. Addressing and changing the mesh requires more complicated algorithms but, due to the reduced number of mesh elements, the overall computation time is often lower.

EikMesh is a 2D mesh generator for unstructured meshes. Its design goal is to provide a fast, light and user-friendly meshing tool with parametric input and advanced visualization capabilities. EikMesh generates exact Delaunay triangulations (DT), constrained Delaunay triangulations (CDT), conforming Delaunay triangulations (CCDT), Voronoi diagrams, and high-quality unstructured and conforming triangular meshes. The user can introduce new data types. Mesh elements work like nodes of a data collection, i. e. mesh elements can carry data which can be addressed in $\mathcal{O}(1)$ for a given mesh element. Therefore, a mesh is also an abstract data type like a `List`.

The goal of this document is to give the user a basic understanding of how to use EikMesh. We will also describe implementation details, algorithms and concepts introduced in the field of computational geometry which are not necessarily required for using the library. Feel free to skip those parts.

1.1 Domain description

Mesh generation begins with the domain to be meshed. In 2D this domain $\Omega_{in} \subset \mathbb{R}^2$ is the Euclidean space. We define the set of points outside the mesh domain to be

$$\Omega_{out} = \mathbb{R}^2 \setminus \Omega_{in}.$$

EIKMESH supports two descriptions of the domain which we introduce in the following sections.

1.1.1 Signed distance functions

One way is to define the domain is by analytical *signed distance functions* d such that

$$d(x) \leq 0 \iff x \in \Omega_{in}. \tag{1.1}$$

For example,

$$d_{circ}(x) = \|x\| - 1 \tag{1.2}$$

defines a circle to be the spatial domain. Especially for curved geometries where ∇d is unique, geometries defined by a distance function is a powerful technique. For more information we refer to [14].

Using distance functions can lead to some disadvantages. Especially for domains containing sharp boundaries, it can be difficult to enforce geometrical conformity at positions x at which $\nabla d(x)$ is not unique. One solution is to add additional fix points at those positions. Another problem might be the computational expensive evaluation of more complex distance functions. This can be solved by an approximated distance function which is prior-computed on a background mesh. Using this technique d is computed at points of the coarse background mesh and is interpolated elsewhere.

1.1.2 Planar straight line graphs

Another well-known domain representation are *planar straight line graphs* (PSLG)s. A PSLG consist of vertices and segments. The generated mesh has to contain all segments and points of the PSLG. We distinguish between *segments* of the PSLG and edges of the mesh. A segment can be represented by multiple edges of the generated mesh. The PSLG is planar i. e. segments only intersect at a shared vertex. Note that we can split intersecting segments to enforce this criterion.

For mesh generation, a PSLG must be *segment-bounded*. In 2D this means that there exist a set of segments in the PSLG which form a simple polygon which contains all points and therefore all other segments. This special polygon separates the mesh domain Ω_{in} from the *exterior domain* Ω_{out}. Furthermore, the mesh domain can contain holes which have to be segment-bounded too. PSLGs are especially useful for describing non-curved domains but they can lead to a high number of small segments if a curve has to be approximated.

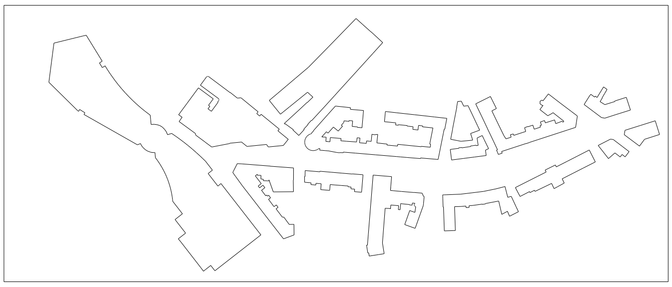


Figure 1.1: Illustration of a segment-bounded PSLG containing 15 holes.

Remark: Note that we can transform any simple polygon into a distance function thus we can transform the PSLG description into a distance function description but we lose all segments not part of a polygon (the bounding polygon or any hole).

2 Mesh data structure

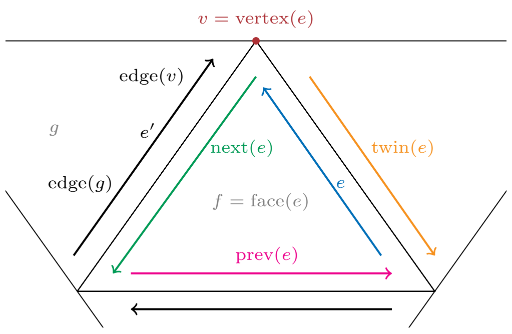


Figure 2.1: Illustration of the doubly connected edge list (DCEL).

2.1 Streams, monads and lambdas

Java 8 introduced some concept known from functional programming to the Java programming language. For example the concept of lazy evaluation and monads such as the `Optional` monad. Additionally, the `Stream` library is a powerful extension to write shorter and more readable code especially for iterating, filtering or reducing a collection of elements (in parallel). To use a `Stream` one also has to get used to the new lambda expressions, like `p -> p.x + p.y`, which is syntactical sugar for writing anonymous classes. EIKMESH excessively uses these language features to establish a user-friendly notation.

2.2 Implementation

The `IMesh` data structure contains and manages all elements of the actual mesh, that is edges, faces, vertices and all data which is stored on those geometrical elements. Therefore, to create, delete, change any element the user has to call methods offered by `IMesh`. For example, to create a new vertex we can call `mesh.createVertex(x, y)`. `IMesh` is an implementation of the edge based half-edge data structure also called doubly connected edge list (DCEL). A DCEL is able to manage planar straight-line graphs (PSLG). Note that a triangulation is a special case of a PSLG. Each edge of the mesh is represented by two counter clockwise (CCW) oriented half-edges (one for each face of the edge). Figure 2.1 shows how mesh elements (vertices, edges / half-edges and faces) are connected and how one can access neighbouring elements in constant time.

We decided to use the DCEL since it is very flexible and all local operations are rather fast. For example to compute the degree of a vertex the following code suffice:

```
E e = getEdge(v);
E n = edge;
int degree = 0;

do {
    n = getTwin(getNext(n))
    degree++;
} while(n != e)
```

Creating a mesh by hand requires a lot of code since all relations, e. g. a half-edge and its next, have to be set. However, this is not necessary to use our algorithms like the computation of the Delaunay triangulation or EikMesh or some other mesh generation method contained in EIKMESH.

2.2.1 Pointer- and array-based implementation

We implemented two versions of the DCEL. For the pointer-based implementation each relation between two objects in `IMesh` such as a `IHalfEdge` and the `IVertex` in which the `IHalfEdge` ends, is realized via Java references i. e. the object of type `IHalfEdge` possesses a reference pointing to the object of type `IVertex` representing its end point. Classes in this category start with a **P** e. g. `PMesh`, `PHalfEdge`, `PFace`, `PVertex`. This version is easier to debug and follows strongly the object-oriented paradigm. For examples in this document, we use these classes, but they can be exchange effortlessly.

The array-based version replaces pointers by indices. Elements of the same type are contained in an array and can be identified by their index i. e. their position in the array. Classes in this category start with an **A** e. g. `AMesh`, `AHalfEdge`, `AFace`, `AVertex`. The advantage of this approach is that those arrays can be sorted with respect to some spatial criterion. Consequently, elements geometrically close can be sorted in such a way that they are close in memory which can improve the performance of algorithms iterating over geometrically local elements.

2.2.2 Properties

Each mesh element (vertex, half-edge, face) can have different user-defined properties. Each property has to have a unique String name. Properties are managed by the mesh, i.e. defining, inserting and accessing properties can be done via the mesh object by the following methods:

- `setData(V vertex, String name, CV value)`, where `CV` is a generic type
- `setData(E edge, String name, CE value)`, where `CE` is a generic type
- `setData(F edge, String name, CF value)`, where `CF` is a generic type
- `CV getData(V vertex, String name, Class)`, where `CV` is a generic type

- `CE getData(E edge, String name, Class)`, where `CE` is a generic type
- `CF getData(F face, String name, Class)`, where `CF` is a generic type

Remark: The user is responsible for the correctness of the types that is inserting the objects / primitives of the same type for one key / name and using the correct Class¡¿. The following code, for example, would cause an Exception:

```
mesh.setData(vertex, "velocity", 3.0);
mesh.getData(vertex, "velocity", Boolean.class);
```

In Java there is nothing like `typedef` thus the programmer has always to fully specify the data type which might lead to long lines of code like:

```
IMesh<PVertex, PHalfEdge, PFace> mesh = ...
```

However, we wanted to make it possible that the user can integrate his own `XMesh` implementation and at the same time, it should be easy to use the meshing library. Therefore, we introduce some classes which predefine some types like `PMesh`. We also encourage the user to use the local-variable type inference introduced in Java 10 by replacing long type definitions by the `var` keyword if possible. The code above and below are semantically the same.

```
PMesh mesh = ...
```

2.2.3 Implicit assumptions

Many operations make assumptions about the mesh it is operating on. For example the operation splitting a triangle into 3 triangles assumes that the split point lies inside the triangle and that the triangle is in fact a valid triangle. We insert assertions to test many of those assumptions but these assertions should only be active for testing because the performance overhead can be huge. The assumptions are listed in the source code documentation (JavaDoc). One very important one is that the mesh is always **counter-clockwise (CCW) oriented**.

2.3 Working with meshes

2.3.1 Building a mesh

The following code example creates a mesh consisting of one square defined by the point set

$$\{(0,0),(1,0),(1,1),(0,1)\}$$

saving Double values on its half-edges and faces. Note that the order of points matter, i. e. they have to form a **simple counter-clockwise (CCW) oriented polygon**. Some operations offered by `IMesh` require the construction of new points `P`, therefore it requires a `IPointConstructor`, here defined by the lambda expression `(x,y) -> new VPoint(x,y)`:

```
var mesh = new PMesh();
mesh.toFace(new VPoint(0,0),new VPoint(1,0),
            new VPoint(1,1),new VPoint(0,1));
```

To use the array-based implementation instead, we only have to exchange `PMesh` by `AMesh`:

```
var mesh = new AMesh();
mesh.toFace(new VPoint(0,0),new VPoint(1,0),
            new VPoint(1,1),new VPoint(0,1));
```

Result

2.3.2 Transforming a PSLG into a mesh

EIKMESH is able to transform a feasible segment-bounded PSLG file[1] into a `IMesh` or internal geometry object like `VPolygon` and `VLine`. The following code reads the `A.poly` file from an `InputStream`:

```
InputStream inputStream = ...
PSLG pslg = PolyGenerator.toVShapes(inputStream);
```

To transform a PSLG file into a `IMesh` the following code suffice:

```
IMesh<...> mesh = ...
InputStream inputStream = ...
PolyGenerator.toPMesh(inputStream, mesh);
```

Here we assume that the `mesh` is empty. All segments of the PSLG will be inserted.

2.3.3 Accessing mesh elements

Direct access

Given any mesh element (vertex, half-edge or face), to access other adjacent mesh elements, the `IMesh` object has to be used. Read the following code from right to left.

```
mesh.getFace(mesh.getTwin(mesh.getEdge(mesh.getFace())));
```

First we get access to some (arbitrary) face f of the mesh. Via the second call we access some (arbitrary) edge e of the face f. Then we get the twin half-edge e_t of this edge. And finally we access the face f_t of the twin half-edge e_t. Therefore, we access a neighboring face of f.

[1]the file format is identical to the format used by TRIANGLE which can be found here

Iterators & streams

To simplify certain access pattern such as accessing all edges of a specific face the library offers different Iterators and Streams to go over

- all half-edges, vertices / points, neighbouring faces of a specific face
- all half-edges ending at a vertex / point
- all faces surrounding a specific vertex / point
- all faces, edges, vertices of a mesh
- ...

The following code iterates over all points of the mesh which are at the border, i. e. those points are connected to at least one half-edge which has only one neighboring face:

```
for(VPoint point : mesh.getPointIt(mesh.getBorder())) {
    ...
}
```

The following gives a parallel stream of the same elements:

```
mesh.streamPoints(mesh.getBorder()).parallel();
```

Remark: Note that those iterators and streams rely on the connectivity of the mesh itself. Therefore, changing the connectivity while iterating will cause unpredictable behaviors and will possibly destroy the validity of the mesh!

2.3.4 Containers

As already mentioned each mesh element (vertex, half-edge, face) can carry some data of the type defined by the use.

```
for(PFace face : dt.getMesh().getFaces()) {
    dt.getMesh().setData(face, "area", dt.getMesh().toTriangle(face).getArea());
}
```

We can, for example, compute the area which is triangulated:

```
double areaSum = dt.getMesh()
    .streamFaces()
    .mapToDouble(f -> dt.getMesh().getData(f, "area", Double.class)
    .get())
    .sum();
double averageArea = areaSum / dt.getMesh().getNumberOfFaces();
double triangulatedArea = (100 * (areaSum / (width * height)));
System.out.println("Triangulated area = " + areaSum);
System.out.println("Average triangle area = " + averageArea);
System.out.println("Area triangulated = " + triangulatedArea + " %");
```

Result (Output)

```
Triangulated area = 97.10080319694295
Average triangle area = 0.04939003214493538
Area triangulated = 97.10080319694295 %
```

We can also exchange `VPoint` by any point container extending `IPoint`.

2.3.5 Visualization

During the development of EIKMESH we created some useful utility classes to accelerate the development process. Visualizing the mesh helped us to understand and improve certain algorithms and to write documents like this.

Tikz generator

The first utility enables the user to convert any valid `IMesh` into a Tikz file to generate high quality vector graphic figures. The following code generates Tikz code where each non acute triangle of a Delaunay triangulation is painted in red:

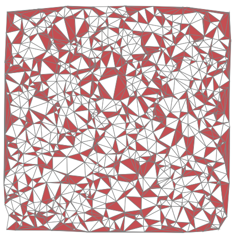
```java
TexGraphGenerator.toTikz(
    dt.getMesh(),
    f -> dt.getMesh().toTriangle(f).isNonAcute() ? red : Color.WHITE,
    1.0f)
```

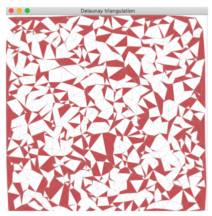
Live visualization

Furthermore, we can display the same mesh on a Java `Swing` canvas:

```java
var panel = new PMeshPanel(
        dt.getMesh(),
        500,
        500,
        f -> dt.getMesh().toTriangle(f).isNonAcute() ? red : Color.WHITE);
panel.display("Delaunay triangulation");
```

Result


(a) Tikz generator



(b) Live visualization

Figure 2.2: Illustration of mesh visualization.

3 Preliminary algorithms

3.1 The point location problem

Given a triangular unstructured 2D mesh and a point p, a very important operation in computational geometry is to find the face f which contains it. Every following algorithm uses this basic operation multiple times. A fast implementation is key for fast unstructured mesh generation. To solve the point location problem different walking strategies, described in [7], are implemented:

- straight walk (default), see Fig. 3.1
- orthogonal walk
- probabilistic walk

Furthermore we implemented different point localization algorithms and data structures:

- Jump and Walk (default) [8]
- Delaunay-Tree [10]
- Delaunay-Hierarchy [5]
- plain walk (no additional strategy),

For a random insertion order of n points, using the Delaunay-Tree or the Delaunay-Hierarchy leads to a time complexity of $\mathcal{O}(\log(n))$ for each point location [10, 5] and using the Jump and Walk strategy requires $\mathcal{O}(n^{1/4})$ time [8]. However, surprisingly the Jump and Walk algorithm does not require any additional data structure thus offers the most flexibility. Additionally, it performs very well in practice, often better than its two alternatives. Basically a `ITriangulation` is the combination of a triangular mesh `IMesh` and a point location algorithm `IPointLocator`.

The first call of the following code example uses the fact that our mesh is a triangulation. The call starts the so called Jump & Walk algorithm which is fast. The second one checks each face in a brute force manner until it finds the face containing the point, which is slow:

```
dt.locateFace(5,5);
dt.getMesh().locate(5,5);
```

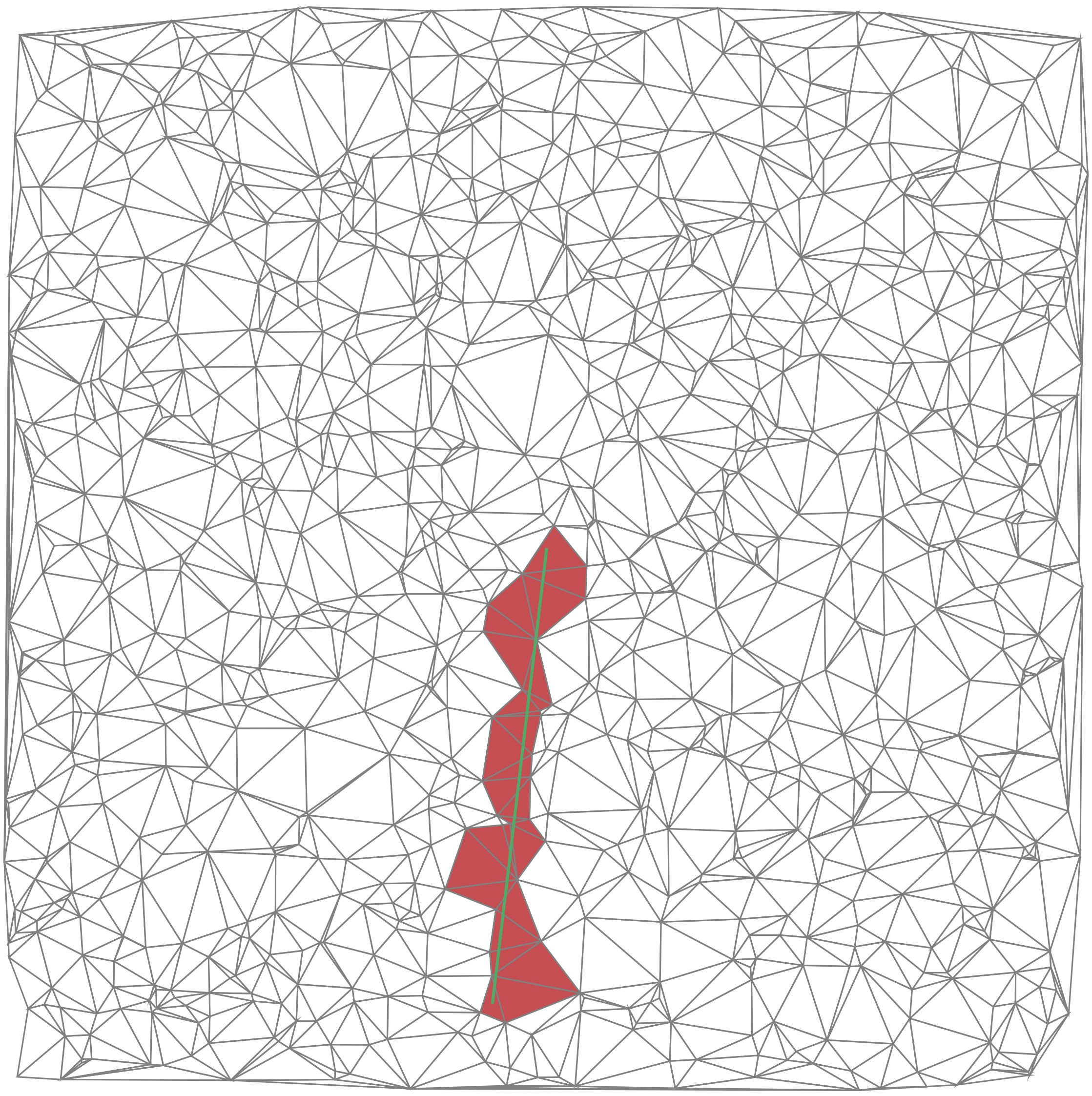


Figure 3.1: Illustration of a straight walk through a Delaunay triangulation of a random point set.

Remark: For bookkeeping reasons, EikMesh does not support point removal if the Delaunay-Tree or the Delaunay-Hierarchy is used.

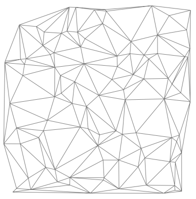
3.2 The Delaunay triangulation

The Delaunay triangulation (DT) is one of the most important structures in computational geometry and the bases for a whole set of algorithms that generate unstructured meshes such as [13, 15, 16, 2, 14, 3, 4]. The so called Delaunay criterion states that a valid triangulation is Delaunay if and only if there is no point contained in any circumcircle of any triangle. We will refer to $\mathrm{DT}(V)$ as the Delaunay triangulation of the vertex set V. DistMesh computes the Delaunay triangulation multiple times to construct the connectifity of the mesh while EikMesh avoids this repetitive computation. To be flexible, i. e. to insert and remove points after the initial triangulation has finished we implemented the incremental method of Lawson [12] also presented in [9]. The following code constructs a Delaunay triangulation of 100 random points uniformly distributed in a 10×10 square:

```
// (1) generate a point set
Random random = new Random(0);
int width = 10;
int height = 10;
int numberOfPoints = 100;
var supply = () -> new VPoint(
                            random.nextDouble()*width,
                            random.nextDouble()*height);
Stream<VPoint> randomPoints = Stream.generate(supply);
List<VPoint> points = randomPoints
                              .limit(numberOfPoints)
                              .collect(Collectors.toList());

// (2) compute the Delaunay triangulation
var dT = new PDelaunayTriangulator(points);
var triangulation = dT.generate();
```

Result

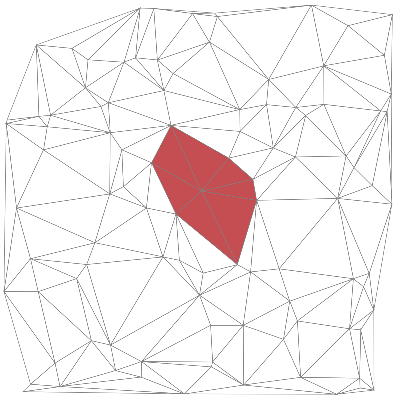


3.2.1 Removing points from a DT

The next code snippet removes the first point p of the face located at $q = (5.0, 5.0)$. We implemented the algorithm presented in [6].

```
var mesh = triangulation.getMesh();
var face = triangulation.locateFace(new VPoint(5,5)).get();
var deletePoints = mesh.getPoints(face);
triangulation.remove(deletePoints.get(0));
```

Result


(a) Before the removal of p.

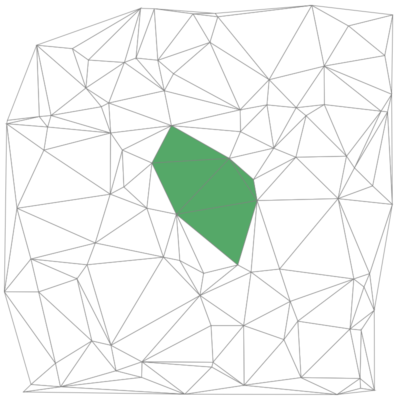

(b) After the removal of p.

Figure 3.2: Two Delaunay triangulations. The connectivity of colored faces changes.

3.3 The constrained Delaunay triangulation

Another well-known triangulation is the so called constrained Delaunay triangulation which is often used instead of the Delauny trianuglaiton if certain segments, for example defined by a planar straight line graph (PSLG), have to be part of the triangulation. We call $\mathrm{CDT}(V)$ a constrained Delaunay triangulation of V. The algorithm we implemented was presented by Sloan in [20]: In the first step we compute the Delaunay triangulation for the PSLG G. In the second step we restore edges of G by edge flips. For more details we refer to [20].

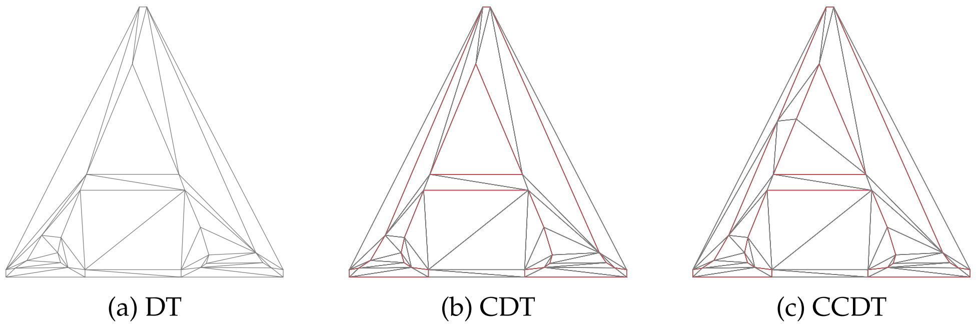
3.4 The conforming Delaunay triangulation

The conforming Delaunay triangulation CCDT is constrained as well as a Delaunay triangulation. That is, each constrained segments is the union of edges in the CCDT and the triangulation is in fact a Delaunay triangulation as well.

The following code generates the CDT of the PSLG `A.poly`. If we replace `conforming = false` by `conforming = true` the CCDT will be computed.

```
final InputStream inputStream = ...
boolean conforming = false;
PSLG pslg = PolyGenerator.toPSLGtoVShapes(inputStream);
var cdt = new PContrainedDelaunayTriangulator(
                        pslg,
                        conforming);
var triangulation = cdt.generate();
```

Result



(a) DT (b) CDT (c) CCDT

Figure 3.3: Illustration of the Delaunay triangulation (DT), the constrained Delaunay triangulation (CDT) and the conforming Delaunay triangulation (CCDT) of the PSLG `A.poly`. Constrains are highlighted (red). The code above constructs the CDT.

4 Mesh generation

4.1 The edge length function

The goal of unstructured mesh generation is to produce high quality meshes containing as less elements as possible. To control the size of the elements, the user has to define an edge length function. More precisely,

$$h : \Omega_{in} \to \mathbb{R}^+$$

is the edge length function defined on Ω_{in}. In the following examples you will find different edge length functions.

4.2 Rebay's algorithms

EIKMESH offers the implementation of two early meshing algorithm introduced by Rebay [15]. The first one, the so called "Voronoi-Vertex point insertion method", is a *Delaunay-refinement* technique, i. e. additional *Steiner vertices* are inserted at the circumcenter of some Delaunay triangle. The second one, called "Voronoi-Segment point insertion method", is a *Frontal-Delaunay algorithm*. Frontal-Delaunay algorithms are a hybridization of advancing-front and Delaunay-refinement techniques, in which a Delaunay triangulation is used to define the topology of a mesh while new Steiner vertices are inserted in a manner consistent with *advancing-front techniques*. Both algorithms have a time complexity of $\mathcal{O}(n \log(n))$, where n is the number of vertices of the generated triangulation. For both the user can control the element size by

$$h : \mathbb{R}^2 \to \mathbb{R}. \tag{4.1}$$

Let t_j be a triangle of a triangulation then

$$\alpha_j = \frac{\rho_j}{\sqrt{3}h(x_j)} \tag{4.2}$$

gives a ratio, where x_j is the circumcenter and ρ_j the circumcircle radius of t_k. We multiply by $\sqrt{3}$ to get the circumradius of a equilateral triangle of side length h. If

$$\max_j \alpha_j \leq 1, \tag{4.3}$$

the triangulation is "fine enough".

Given a segment-bounded PSLG G, our algorithms start by constructing the conforming Delaunay triangulation of G. In the second step we split all boundary edges $e = \{v_1, v_2\}$ at their midpoint x_e until

$$\|v_1 - v_2\| \leq h(x_e). \tag{4.4}$$

Note that Rebay uses the Boywer-Watson algorithm, that is, triangles are removed and the emerging convex polygon is triangulated. EIKMESH applies the flipping technique described by Lawson in [12] instead. After computing the CCDT and the splitting of boundary edges, additional Steiner vertices are inserted. In each iteration a new Steiner vertex v is inserted according to h and the CDT $\mathcal{T}_{k+1} = \text{CDT}(p(\mathcal{T}_k) \cup \{v\})$ is established on the basis of $\mathcal{T}_k$. While refining, the properties of the CDT is retained, i. e. $\mathcal{T}_k$ might violate the Delaunay criterion but is a CDT. We achieve this by avoiding flips of edges belonging to G. After vertex insertion, triangles outside the domain are removed.

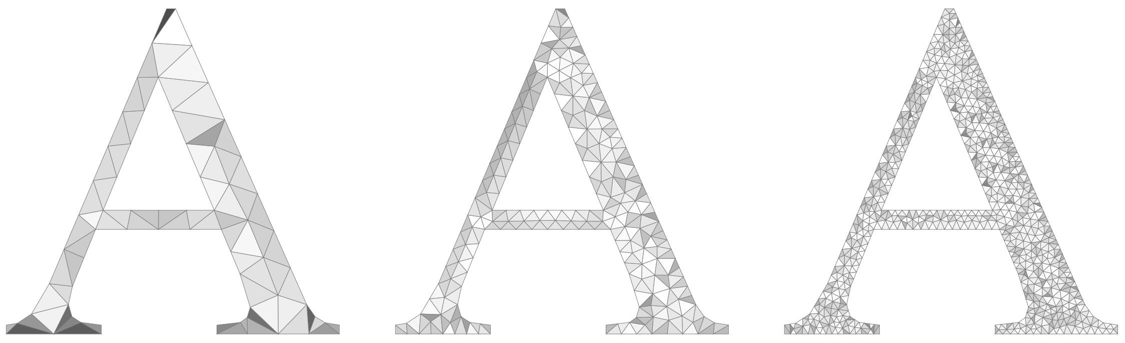
4.2.1 Voronoi-Vertex point insertion method

In each iteration k, v is the circumcenter of the triangle $t_j \in \mathcal{T}_{k-1}$ with the largest circumcenter radius ρ_j. Note that if $\mathcal{T}_{k-1}$ is a Delaunay triangulation, v is in fact a vertex of the Voronoi diagram of the current vertex set. It might be the case that v lies outside of the segment-bound of G. In this case we ignore the vertex, i. e. it will not be inserted.

The following code generates a mesh using $h(x_j) = 0.05$:

```java
PSLG pslg = ...
double h0 = 0.05;
var vviMethod = new PVoronoiVertexInsertion(
                pslg,
                p -> h0);
var triangulation vviMethod.generate();
```

Result



(a) $h(x_j) = 0.05$ (b) $h(x_j) = 0.02$ (c) $h(x_j) = 0.01$

Figure 4.1: Triangulation using the Voronoi-Vertex point insertion method of the PSLG defined by `A.poly`. The color code indicate triangle qualities.

4.2.2 Voronoi-Segment point insertion method

In each iteration k, v lies on the segment of the Voronoi diagram of $p(\mathcal{T}_{k-1})$. Additional each inserted vertex is an attempt to generate a new triangle t_j such that

$$\frac{\rho_j}{\sqrt{3}h(x_j)} = \alpha_j \leq 1. \tag{4.5}$$

is satisfied, which is not always possible. At the beginning all *external* triangles are *accepted*. A triangle is *active* if it is not *accepted* and there is a neighboring *accepted* triangle. The triangle which will be considered for refinement is the *active* triangle t_j with the largest circumradius ρ_j, i. e. for which

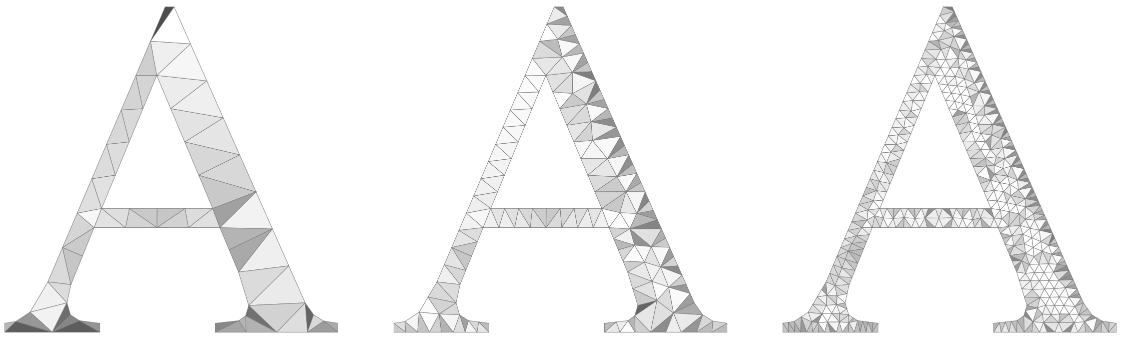
$$\max_j \rho_j \tag{4.6}$$

holds. Like before, v might be outside of the segment-bound of G. In this case we ignore the vertex, i. e. it will not be inserted. For more information we refer to [15].

The following code generates a mesh using $h(x_j) = 0.05$:

```
PSLG pslg = ...
double h0 = 0.05;
var vviMethod = new PVoronoiSegmentInsertion(
                pslg,
                p -> h0);
var triangulation vviMethod.generate();
```

Result



(a) $h(x_j) = 0.05$ (b) $h(x_j) = 0.02$ (c) $h(x_j) = 0.01$

Figure 4.2: Triangulation using the Voronoi-Segment point insertion method of the PSLG defined by `A.poly`. The color code indicate triangle qualities.

Remark: Since both algorithm ignore Steiner vertices outside of the segment-bound of G, the split of segments which are part of the segment-bound is essential. It is easy to construct a PSLG G for which the circumcenter of each triangle of the CCDT of G is not contained in the segment-bound!

4.3 Ruppert's algorithm

Ruppert's algorithm [16] for two-dimensional quality mesh generation is probably the first theoretically guaranteed meshing algorithm to be truly satisfactory in practice. The algorithm allows the density of triangles to vary quickly over short distance. It is quite similar to Rebay's Voronoi-Vertex point insertion method, but avoids Steiner vertices outside the segment-bound. Furthermore, it is also a Delaunay-refinement method. For an excellent and extensive description we refer to [19].

Remark: The termination of Ruppert's algorithm is only guaranteed for $\theta_{min} \leq 20.7°$ and if there is no angle smaller $60°$ in G. The extension in [19] resolves the issue of small input angles in G but at this moment EIKMESH only supports the algorithm proposed by Ruppert. A C-implementation is accessible via Triangle. See [18, 17, 19].

The following code generates a mesh using $h(x_j) = 0.05$ and $\theta = 20°$:

```
PSLG pslg = ...
double h0 = 0.02;
double theta = 20.0;
var ruppert = new PRuppertsTriangulator(
        pslg,
        p -> h0,
        theta);
var triangulation = ruppert.generate();
```

Result

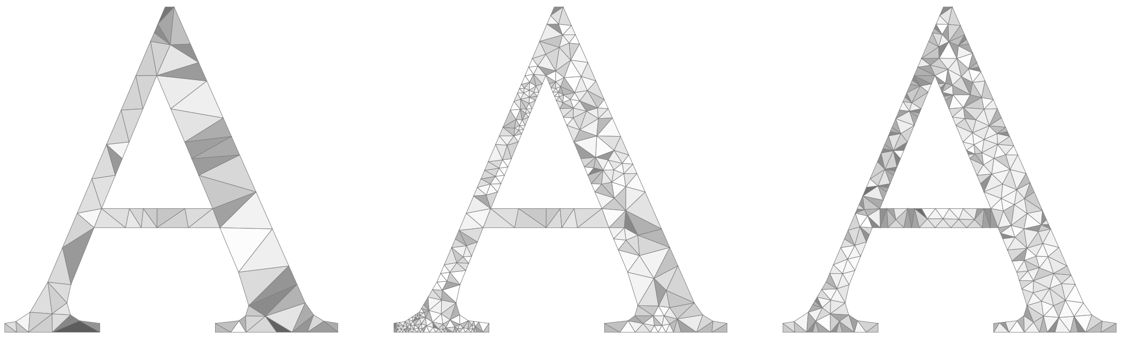
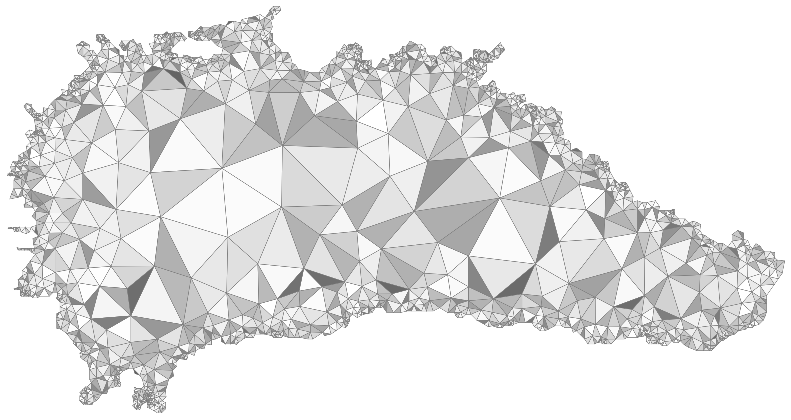
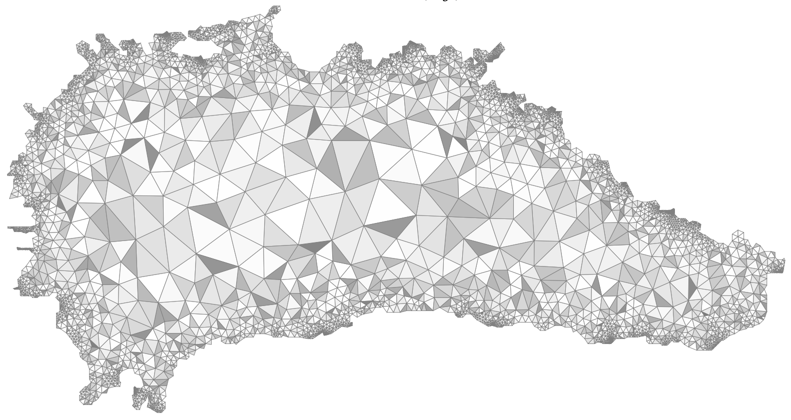

(a) $\theta = 20°, h(x_j) = \infty$ (b) $\theta = 30°, h(x_j) = \infty$ (c) $\theta = 20°, h(x_j) = 0.02$

Figure 4.3: Accomplished triangulation using Ruppert's algorithm of the PSLG defined by `A.poly`. The color code indicate triangle qualities. For $\theta > 32°$ Ruppert's algorithm does not terminate for this example.



(a) $\theta = 25^\circ, h(x_j) = \infty$



(b) $\theta = 30^\circ, h(x_j) = \infty$

Figure 4.4: Accomplished triangulation using Ruppert's algorithm of the PSLG of Greenland. The color code indicate triangle qualities.

5 EikMesh

EikMesh starts by constructing an initial triangulation $\mathcal{T}_0$ which satisfies the size constrains defined by the edge length function h. Based on the initial triangulation a smoothing process improves the mesh quality by moving vertices iteratively. The algorithm is an adaptation of the DistMesh algorithm introduced in [14]. In this document we left out an extensive explanation of DistMesh and refer instead to the publication of Persson and Strang. EikMesh is also described in [21].

5.1 Initial triangulation

5.1.1 Bisection according to a space filling curve

To avoid the creation of vertices, we replaced the rejection mechanism of DistMesh by another algorithm. We construct an initial high-quality mesh by applying the hierarchical mesh refinement strategy described in [1]. The starting point is a square with side length s containing Ω_{in}. The square is spit into two triangles as depicted in

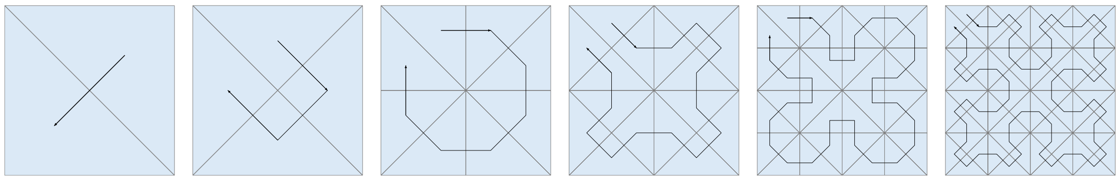


Figure 5.1: The initial mesh for the first 5 refinement steps and the corresponding space-filling Sierpinski curve.

Edge e of length l_e are split until

$$l_e \leq h(e). \tag{5.1}$$

hods. Afterwards, all triangles outside of Ω_{in} are removed. By using this bisection strategy the triangle quality defined in [14] is

$$2\sqrt{2} - 2 \approx 0.83. \tag{5.2}$$

for each triangle.

5.2 The smoothing algorithm

5.2.1 Edge flips

The MatLab code of DistMesh presented [14] is short and simple due to the use of the build-in Delaunay triangulator. By computing the Delaunay triangulation, DistMesh avoids code which changes the connectivity, i. e. the set of edges E. Furthermore, even if the final result of DistMesh depends on the initial vertex set V_0, it can handle any initial vertex set especially sets for which $\mathcal{T}_0 = \text{DT}(V_0)$ contains a lot poor quality triangles. However, this comes at a cost since the time complexity of computing the Delaunay trioangulation of n point is $\mathcal{O}(n \log(n))$. EikMesh avoids the computation of the Delaunay triangulation whenever possible. Let $\mathcal{T}_i$ be some **valid** triangulation at iteration i then the following flip algorithm suffices to construct a valid Delaunay triangulation $\mathcal{T}_{i+1}$ based on $\mathcal{T}_i$:

Algorithm 1: edge-flipping [12]

```
while $\exists e \in E_i$ : ¬isDelaunay(e) do
    flip($e$);
end
```

In worst case this algorithm requires $\mathcal{O}(n^2)$ time. However, if only a few changes to the connectivity are necessary, which is the case if the overall triangle quality is above some threshold, the complexity of Algorithm 1 is linear. Additionally, exploiting massive parallelism for Algorithm 1 is straightforward. Flipping edges for each iteration removes all jumps in the triangle quality and improves the convergence towards an equilibrium [21].

5.2.2 Long boundary edges

EikMesh changes the way in which boundary elements (vertices, edges, triangles) are treated. Why are these elements special with respect to the DistMesh algorithm? First of all, **boundary edges will never be flipped**, i. e. successive Delaunay triangulations computed by DistMesh will contain the same boundary edge. The only way to get rid of a boundary edge is when its triangle is removed, which is only the case if its triangle centroid lies outside the domain. Additionally, **the movement of boundary vertices is more restricted** (by external forces and fix points). For the remaining section, let v_o be the opposite vertex of the boundary edge e_b and t_b the boundary triangle of e_b.

Let us assume that e_b is the longest edge of t_b and all incident edges of v_o have reached almost their desired length. In this situation

$$F(v_o) \approx 0 \tag{5.3}$$

holds. See Fig. 5.2a. More general, there is not necessarily a force acting towards $\partial\Omega_{in}$. We observed that those situations appear frequently during a DistMesh run, which leads to v_o moving very slowly towards the domain boundary while the quality $\tau(t_b)$ drops. This can lead to low quality triangles in the final mesh also observed by Jonas Koko in [11] and worsens the convergence rate of DistMesh.

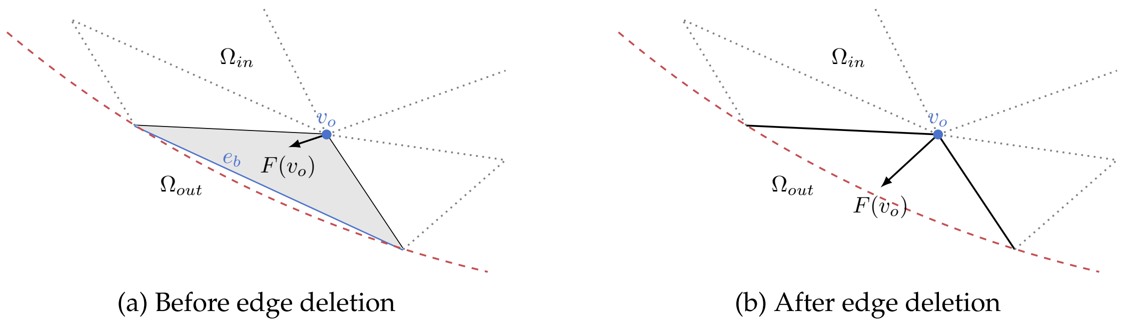


Figure 5.2: Illustration of a long boundary edge and its treatment. Left: The force $F(v_0)$ acting on the opposite vertex of a long boundary edge e_b (blue) at the boundary domain $\partial\Omega_{in}$ (red) tends to be very small. Therefore it takes many iterations until the bad quality triangle t_b (gray) will be removed. Right: Inside projection of v_o (blue) onto $\partial\Omega_{in}$ (red) after the deletion of the boundary triangle t_b.

One solution we introduced in [21] is to remove t_b (and e_b) and project v_o towards the domain boundary with respect to ∇d. More precisely, EikMesh removes low quality triangles at the boundary and projects all boundary points towards $\partial\Omega_{in}$. This is different from DistMesh which does only project points outside of Ω_{in}. To do so one has to identify boundary vertices to be at the boundary of the triangulation $\mathcal{T}$ which is trivial using the DCEL data structure. By adjusting boundary vertices during the mesh generation, make sure that if a vertex is at the boundary `mesh.getEdge(v)` returns always an edge which is at the boundary in $\mathcal{O}(1)$.

5.3 Examples

5.3.1 Uniform ring

The following code snippet generates a mesh using the standard EikMesh algorithm for a simple curved geometry. The geometry is defined by the distance function

$$d_{ring}(x) = \text{abs}(\|x - c\| - 0.5) - 0.4$$

with $c = (1, 1)$ to be the center of the ring. The inner radius is 0.2 and the outer radius 1.0. The edge length is $h(x) = 0.1$ and the `bound` has to contain Ω_{in}. The result is illustrated in Fig. 5.3.

```
VRectangle bound = new VRectangle(-0.1, -0.1, 2.2, 2.2);
IDistanceFunction d_r = IDistanceFunction.createRing(1, 1, 0.2, 1.0);
double h0 = 0.1;
PEikMesh meshImprover = new PEikMesh(d_r,h0,bound);
meshImprover.generate();
```

5.3.2 Disc subtracted by a rectangle

Distance functions can be combined. The result is illustrated in Fig. 5.3.

```
 1 VRectangle bound = ...
 2 VRectangle rect = new VRectangle(0.5, 0.5, 1, 1);
 3 IDistanceFunction d_c = IDistanceFunction.createDisc(0.5, 0.5, 0.5);
 4 IDistanceFunction d_r = IDistanceFunction.create(rect);
 5 IDistanceFunction d = IDistanceFunction.substract(d_c, d_r);
 6 double edgeLength = 0.03;
 7 var meshImprover = new PEikMesh(
 8         d,
 9         p -> edgeLength + 0.5 * Math.abs(d.apply(p)),
10         edgeLength,
11         bound,
12         Arrays.asList(rect));
```

Line 1 defines a bounding box containing Ω_{in}. The second line defines the hole and is transformed into a distance function d_{rec} in line 4. The disc is defined by the distance function d_{disc} constructed in line 3. By subtracting d_{rect} from d_{disc} we construct

$$d(x) = d_{comb}(x) = \max\{d_{disc}, -d_{rec}\}$$

Note that even if the geometry is defined by d_{comb}, the rectangle `rect` is an argument of EikMesh. This is optional and automatically inserts fix points at the corners of the rectangle which improves the mesh quality. Furthermore, in line 9 we use a non-constant edge length function

$$h(x) = h_{min} + 0.3 \cdot \text{abs}(d_{comb}(x)).$$

5.3.3 Combining distance functions

We can also construct geometries by the union and intersection of distance functions. For example, Fig. 5.3j illustrate the result of the following code.

```
 1 // inner rectangle
 2 VRectangle rect = new VRectangle(-0.5, -0.5, 1, 1);
 3 
 4 // outer rectangle
 5 VRectangle boundary = new VRectangle(-2,-0.7,4,1.4);
 6 
 7 // construction of the distance function, define the 2 discs
 8 IDistanceFunction d1_c = IDistanceFunction.createDisc(-0.5, 0, 0.5);
 9 IDistanceFunction d2_c = IDistanceFunction.createDisc(0.5, 0, 0.5);
10 
11 // define the two rectangles
12 IDistanceFunction d_r = IDistanceFunction.create(rect);
13 IDistanceFunction d_b = IDistanceFunction.create(boundary);
14 
15 // combine distance functions
16 IDistanceFunction d_unionTmp = IDistanceFunction.union(d1_c, d_r)
17 IDistanceFunction d_union = IDistanceFunction.union(d_unionTmp, d2_c);
18 IDistanceFunction d = IDistanceFunction.substract(d_b,d_union);
19 
20 // h_min
21 double edgeLength = 0.03;
22 
23 var meshImprover = new PEikMesh(
24         d,
25         p -> edgeLength + 0.5 * Math.abs(d.apply(p)),
26         edgeLength,
27         GeometryUtils.boundRelative(boundary.getPath()),
28         Arrays.asList(rect));
29 
30 // generate the mesh
31 var triangulation = meshImprover.generate();
```

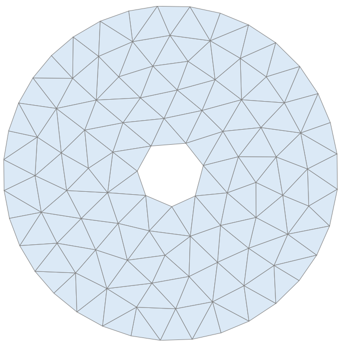
We define two circles and a rectangle and compute the union d_{union} of these three distance functions in line 7. Finally we subtract d_{union} from a larger rectangle.

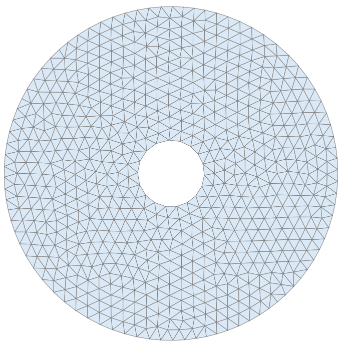
Remark: The definition of the distance function d, the edge length function h, the minimum edge length h_{min} and the bounding box have to be carefully defined by the user. For example, if h_{min} is too large EikMesh might fail to construct a mesh.

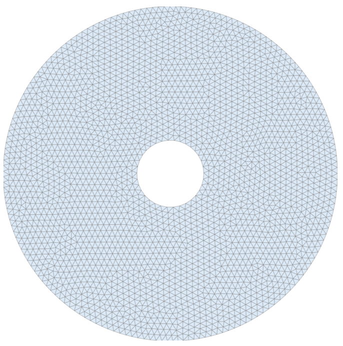
5.3.4 Meshing a PSLG

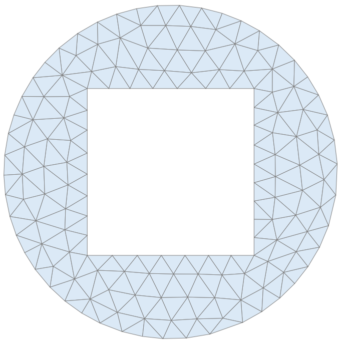
The following code sniped produces the mesh illustrated in Fig. 5.4 by constructing a distance functions based on the given PSLG:

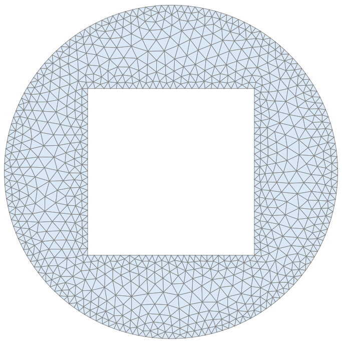
```
PSLG pslg = ...
PEikMesh meshImprover = new PEikMesh(
          pslg.getSegmentBound(), 0.02, pslg.getHoles());
```

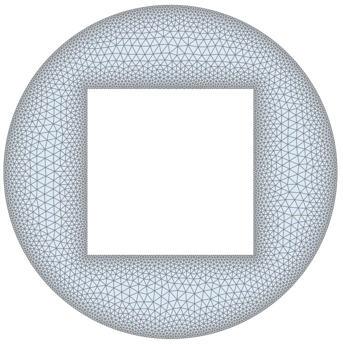

(a) $h(x) = 0.3$.

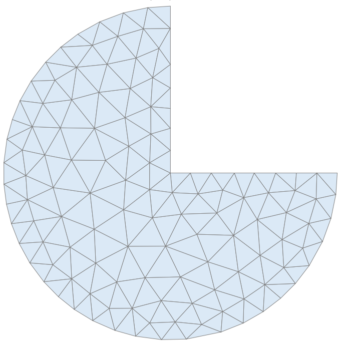

(b) $h(x) = 0.1$.

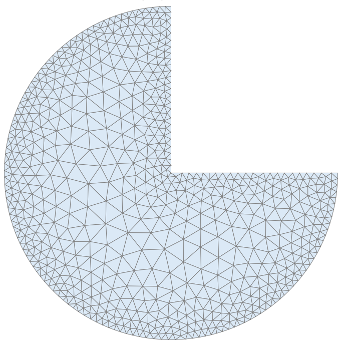

(c) $h(x) = 0.05$.

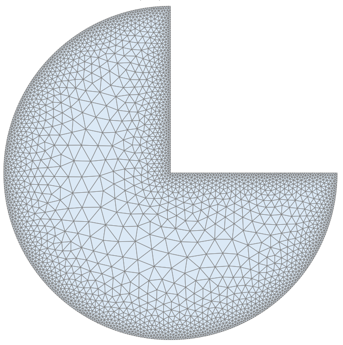

(d) $h(x) = 0.1$.


(e) $h(x) = 0.03$.


(f) $h(x) = 0.01$.


(g) $h(x) = 0.1$.


(h) $h(x) = 0.03$.


(i) $h(x) = 0.01$.

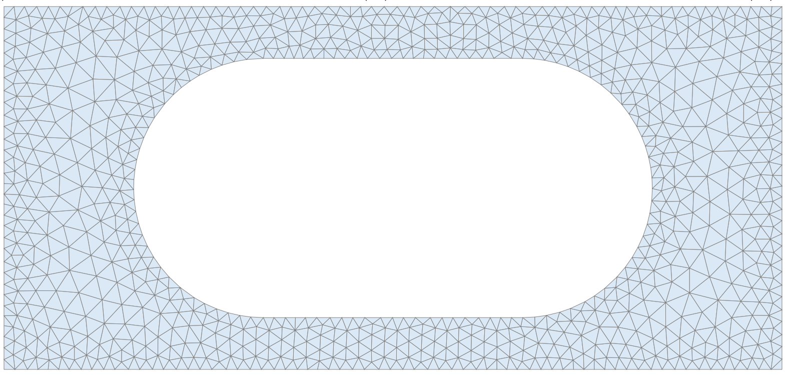
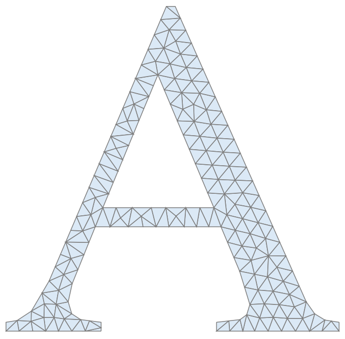
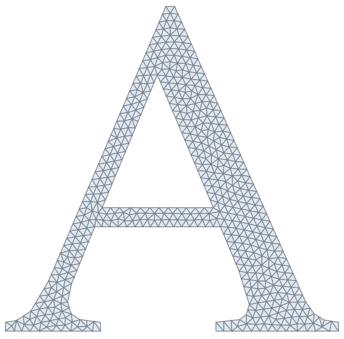

(j) $h(x) = 0.03$.

Figure 5.3: Meshes generated by EikMesh using different distance functions d and edge length functions h.


(a) $h(x) = 0.05$.


(b) $h(x) = 0.02$.

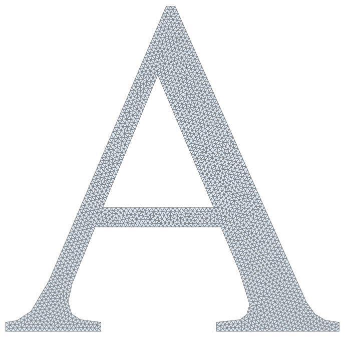
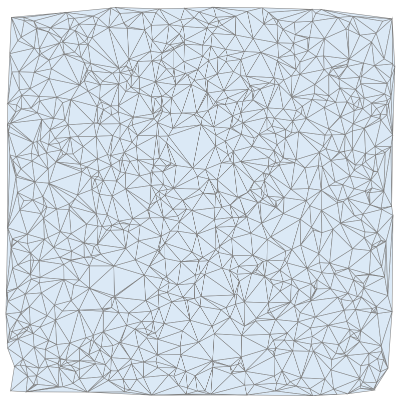

(c) $h(x) = 0.01$.

Figure 5.4: Meshes generated by EikMesh using different edge length functions h.


(a) $\mathrm{DT}(V)$ of 1000 uniformly random distributed points.

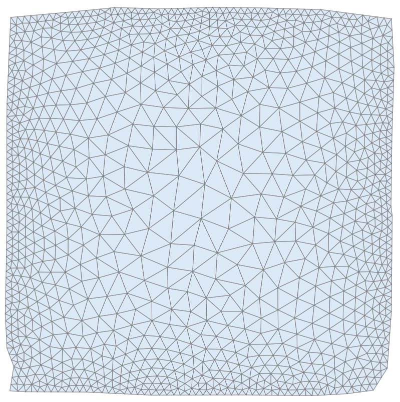

(b) The resulting triangulation generated by EikMesh.

Figure 5.5: Meshes generated by EikMesh using a given Delaunay triangulation as input. The boundary of the mesh does not change.

5.3.5 Random Delaunay triangulation

The following code uses a Delaunay triangulation of 1000 random points as the initial triangulation, also depicted in Fig. 5.5.

```
var dt = ... // the Delaunay triangulation
var eikMesh = new PEikMesh(
        p -> 1.0 + Math.abs(bound.distance(p)),
        dt.getTriangulation());
var triangulation = eikMesh.generate();
```

The quality of the resulting triangulation is approximately 0.96. This example shows that EIKMESH can be used to improve even a very low quality initial triangulation.

5.3.6 Urban environment

In the last example we mesh an urban environment ($600 \times 250[m^2]$) which is in fact a geometry used for pedestrian simulation. The result is depicted in Fig. 5.6.

```java
// (1) read the PSLG from a file / an input stream
final InputStream inputStream = MeshExamples.class.getResourceAsStream("/poly/
    kaiserslautern.poly");
PSLG pslg = PolyGenerator.toPSLGtoVShapes(inputStream);

// (2) construct a distance function d out of the PSLG
Collection<VPolygon> holes = pslg.getHoles();
VPolygon segmentBound = pslg.getSegmentBound();
IDistanceFunction d = IDistanceFunction.create(segmentBound, holes);

// (3) use EikMesh to construct the mesh
double h0 = 5.0;
var meshImprover = new PEikMesh(
        d,
        p -> h0 + 0.3 * Math.abs(d.apply(p)),
        h0,
        new VRectangle(segmentBound.getBounds2D()),
        pslg.getHoles(),
);
meshImprover.generate();
```

Remark: Since the distance function d is complicated the mesh generation is quite slow because d has to be evaluated for each iteration of the smoothing process for almost any mesh element. To reduce the complexity of the evaluation one can construct a background mesh using, for example, Ruppert's algorithm and approximate d via the interpolation on the background mesh.

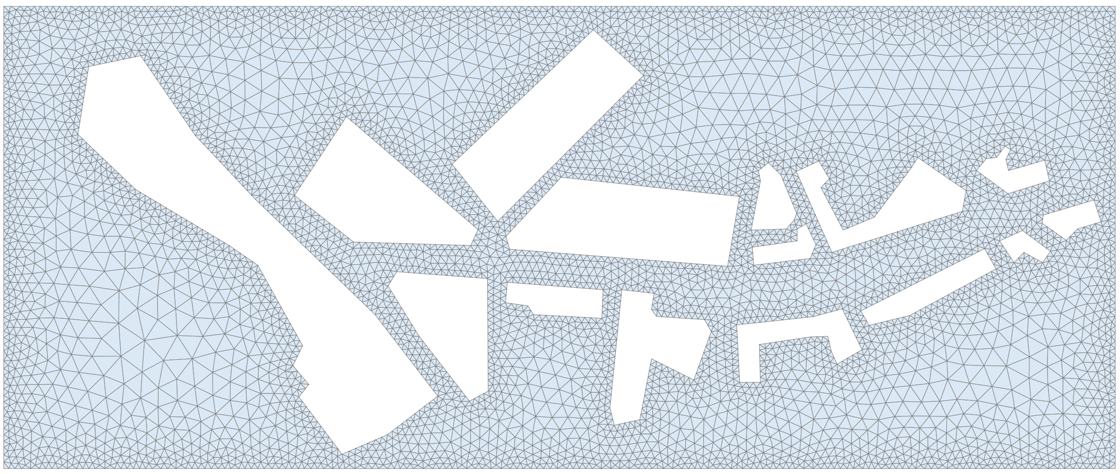


Figure 5.6: A high quality mesh of an urban environment.

Bibliography

[1] Jörn Behrens and Michael Bader. Efficiency considerations in triangular adaptive mesh refinement. *Philosophical Transactions of the Royal Society A*, 367:4577–4589, October 2009. Theme Issue 'Mesh generation and mesh adaptation for large-scale Earth-system modelling'.

[2] Siu-Wing Cheng and Tamal K. Dey. Quality meshing with weighted delaunay refinement. *SIAM Journal on Computing*, 33(1):69–93, 2003.

[3] Siu-Wing Cheng, Tamal K. Dey, and Joshua A. Levine. A practical delaunay meshing algorithm for alarge class of domains*. In Michael L. Brewer and David Marcum, editors, *Proceedings of the 16th International Meshing Roundtable*, pages 477–494, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] Siu-Wing Cheng, Tamal K. Dey, and Edgar A. Ramos. Delaunay refinement for piecewise smooth complexes. *Discrete & Computational Geometry*, 43(1):121–166, Jan 2010.

[5] Olivier Devillers. The delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13(02):163–180, 2002.

[6] Olivier Devillers. On deletion in delaunay triangulations. *International Journal of Computational Geometry &amp; Applications*, 12(03):193–205, 2002.

[7] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. In *17th Annual ACM Symposium on Computational Geometry (SCG)*, pages 106–114, Boston, United States, June 2001.

[8] Luc Devroye, Christophe Lemaire, and Jean-Michel Moreau. Expected time analysis for delaunay point location. *Computational Geometry*, 29(2):61 – 89, 2004.

[9] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15(3):223–241, Mar 1996.

[10] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, 7(1):381–413, Jun 1992.

[11] Jonas Koko. A matlab mesh generator for the two-dimensional finite element method. *Applied Mathematics and Computation*, 250:650 – 664, 2015.

[12] C.L. Lawson. Software for c1 surface interpolation. In John R. Rice, editor, *Mathematical Software*, pages 161 – 194. Academic Press, 1977.

[13] Rainald Löhner and Paresh Parikh. Generation of three-dimensional unstructured grids by the advancing-front method. *International Journal for Numerical Methods in Fluids*, 8(10):1135–1149, 1988.

[14] Per-Olof Persson and Gilbert Strang. A simple mesh generator in matlab. *SIAM Review*, 46(2):329–345, 2004.

[15] S. Rebay. Efficient unstructured mesh generation by means of delaunay triangulation and bowyer-watson algorithm. *Journal of Computational Physics*, 106(1):125–138, 1993.

[16] Jim Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 83–92, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[17] Johnathan Richard Shewchuk. Robust adaptive floating-point geometric predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, SCG '96, pages 141–150, New York, NY, USA, 1996. ACM.

[18] Jonathan Richard Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry Towards Geometric Engineering*, pages 203–222, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[19] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry*, 22(1):21 – 74, 2002. 16th ACM Symposium on Computational Geometry.

[20] S. W. Sloan. A fast algorithm for generating constrained delaunay triangulations. *Computers & Structures*, 47(3):441–450, 1993.

[21] Benedikt Zönnchen and Gerta Köster. A parallel generator for sparse unstructured meshes to solve the eikonal equation. *Journal of Computational Science*, 2018.