The PSurface Library

Oliver Sander*†

July 12, 2010

Abstract

We describe PSurface, a C++ library that allows to store and access piecewise linear mappings between simplicial surfaces in $\mathbb{R}^2$ and $\mathbb{R}^3$. These mappings are stored in a graph data structure and can be constructed explicitly, by projection, or by surface simplification. Piecewise linear maps can be used, e.g., to construct boundary approximations for finite element grids, and grid intersections for domain decomposition methods. In computer graphics the mappings allow to build level-of-detail representations as well as texture- and bump maps. We document the data structures and algorithms used and show how PSurface is used in the numerical analysis framework Dune and the visualization software Amira.

1 Introduction

Let S_1 and S_2 be d-dimensional simplicial surfaces in $\mathbb{R}^{d+1}$, where $d \in \{1, 2\}$. A map $\Xi : D \subseteq S_1 \to S_2$ will be called *piecewise linear*, if for each pair of simplices $T_1 \in S_1$ and $T_2 \in S_2$ the restriction of Ξ to $T_1 \cap \Xi^{-1}(T_2)$ is an affine function. We assume that all maps we consider are continuous, injective, and such that $\Xi(S_1)$ is a simplicial subsurface of S_2. However, we do not require that simplices of S_1 be mapped onto simplices of S_2. If, additionally, S_1 and S_2 are homeomorphic and $\Xi : S_1 \to S_2$ is a homeomorphism, we call Ξ a piecewise linear *parametrization* of S_2 over S_1.

It is occasionally useful to be able to handle such piecewise linear maps in a computer algorithm. In the numerical simulation of two-body contact problems in linear elasticity, e.g., regions of potential contact on the boundaries of the objects need to be identified with a homeomorphism. This homeomorphism needs to be evaluated repeatedly in the course of the simulation [19]. Also, when a partial differential equation is to be solved on a domain with a curved or highly detailed boundary, its approximation by a grid may discard important geometric information. To make this information available at the time of grid refinement, a parametrization of the domain boundary over the grid boundary is necessary. In computer graphics, parametrizations can be used, e.g., to generate texture- and bump maps [6, 17]. Atlas-based methods for the automatic segmentation of medical image data use parametrizations to make the different models in an atlas comparable [15].

*Institut für Mathematik, Freie Universität Berlin, Arnimallee 6, 14195 Berlin, sander@mi.fu-berlin.de

†Supported by the DFG research center Matheon

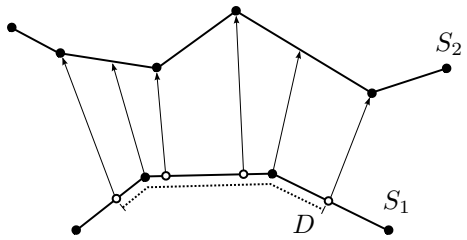

Figure 1: Piecewise linear map from a subset D of a one-dimensional simplicial surface S_1 to a second simplicial surface S_2.

PSurface is a C++ library to handle continuous piecewise linear maps between simplicial surfaces. Its core is a data structure that can hold the simplicial surfaces S_1, S_2 and a piecewise linear map $\Xi : D \subset S_1 \to S_2$. The surfaces can be one- or two-dimensional and are expected to be embedded in a Euclidean space of one dimension higher. S_1 and S_2 are not assumed to be manifolds but can, e.g., contain edges with more than two adjacent triangles. The mapping Ξ itself is stored as a graph $\mathcal{G}$ on the simplices of S_1. This graph is the image of the edges of $\Xi(S_1) \subset S_2$ under Ξ^{-1}. If $d = 2$ we obtain a planar graph on each two-dimensional simplex of S_1. If $d = 1$ the graph consists of nodes along the one-dimensional simplices of S_1 and graph edges between them (Figure 1). For any point $x \in S_1$, the image $\Xi(x) \in S_2$ can then be evaluated by a point-location in $\mathcal{G}$ and subsequent linear interpolation.

Several ways of constructing mappings Ξ have been implemented. The most simple way is by direct construction. A special factory class allows to set up maps by explicitly prescribing how vertices of S_1 are mapped onto S_2, what points of S_1 are mapped onto the vertices of S_2, and how the images of the edges of S_1 cross edges of S_2. Given two surfaces, a mapping can also be generated by projecting S_1 onto S_2 in the direction of a given vector field on S_1. Finally, a parametrization of a given fine surface S_2 over a coarser surface S_1 can be constructed by initially setting $S_1 = S_2$, $\Xi = \mathrm{Id}$, and successively removing points from S_1. A valid $\Xi : S_1 \to S_2$ is kept at each step, and hence this creates the coarse surface S_1 along with the parametrization function Ξ.

To the knowledge of the author, PSurface is the only library for the general handling of piecewise linear mappings between simplicial surfaces currently available. Very similar functionality exists in various places, e.g., the code used for atlas-based image segmentation in [15] or the LGM domain manager for piecewise linear boundary descriptions in the UG finite element software [3]. However, these implementations are all hardwired to their respective applications and cannot be used separately.

This article is intended to describe and document the data structures and algorithms in PSurface. Information about PSurface has appeared implicitly in various publications [13, 14, 18, 19], but this is the first document that describes PSurface explicitly, exclusively, and exhaustively. It is not a reference manual. As PSurface is still under development the actual method names and signatures can still be subject to change, and mention of them here would outdate quickly. Up-to-date information is available in form of a doxygen-generated documentation distributed with the library. The library itself is

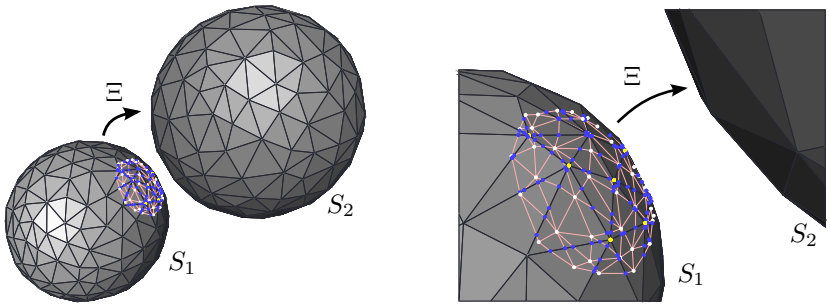

Figure 2: Piecewise linear map from a subset of a sphere S_1 to another sphere S_2. The preimages of the edges of S_2 form a graph on S_1.

currently available under a free license at `http://numerik.mi.fu-berlin.de/dune/psurface`.

2 The data structure

This section describes the data structure for continuous piecewise linear maps between simplicial surfaces, which is the heart of PSurface. We concentrate on the case $d = 2$, because it is considerably more complicated than $d = 1$. Remarks concerning maps between one-dimensional surfaces will be given where appropriate.

2.1 Simplicial surfaces

To fix ideas and notation we briefly define what we mean by 'simplicial surface'.

Definition 2.1. *A simplicial k-complex $\mathcal{K}$ is a set of closed simplices of dimension less than or equal to k, that satisfies the following conditions:*

1. *Any face of a simplex from $\mathcal{K}$ is also in $\mathcal{K}$.*
2. *The intersection of any two simplices σ_1, $\sigma_2 \in \mathcal{K}$ is a face of both σ_1 and σ_2, or empty.*

A simplicial k-complex $\mathcal{K}$ is called homogeneous if every simplex of dimension less than k is the face of some simplex $\sigma \in \mathcal{K}$ of dimension exactly k.

We call *d-dimensional simplicial surface* a homogeneous simplicial d-complex in a $d+1$-dimensional Euclidean space. Note that this definition does not restrict the setting to discrete approximations of manifolds. Indeed, situations like three triangles meeting at a common edge are easily handled by PSurface.

Let S be a d-dimensional simplicial surface with $d \geq 1$. We call the 0-dimensional simplices of S the *vertices*, and the 1-dimensional simplices the *edges* of S. The d-dimensional simplices are called *elements*. We denote the set of vertices by $\mathcal{V}$, the set of edges by $\mathcal{E}$, and the set of elements by $\mathcal{T}$.

2.2 A data structure for piecewise linear maps

The foundation of the data structure for piecewise linear maps $\Xi : D \subset S_1 \to S_2$ are the data structures for the two surfaces S_1 and S_2. There are separate data structures for each surface. These are conceptually very similar, but not identical, because more functionality is needed for S_1. Both data structures store an array with the vertex positions and, for each k, $1 \leq k \leq d$, an array of $k+1$-tuples of vertex indices to store the k-dimensional simplices.

Before describing the data structure for piecewise linear maps between simplicial surfaces we define these maps formally.

Definition 2.2. *Let S_1, S_2 be simplicial surfaces. A function $\Xi : D \subset S_1 \to S_2$ is called piecewise linear if for each pair of triangles $T_1 \in S_1$ and $T_2 \in S_2$ the restriction of Ξ to $T_1 \cap \Xi^{-1}(T_2)$ is an affine function.*

We make the following additional assumption.

Assumption 2.1. *For each element T of S_2, the set $T \cap \Xi(S_1)$ is a simplex of S_2 or empty.*

By restricting our attention to a subsurface of S_2 we even assume that Ξ is surjective, i.e., the image surface $\Xi(S_1) \subset S_2$ is equal to S_2. However, this restriction is not imposed in the implementation.

The following special case will appear frequently.

Definition 2.3. *If S_1 and S_2 are homeomorphic, and $\Xi : S_1 \to S_2$ is defined on all of S_1 and is a homeomorphism, then Ξ is called a parametrization of S_2 over S_1.*

The edges and vertices of S_2 form a graph with a geometric realization, which is mapped by the inverse function Ξ^{-1} onto S_1. By linearity, it is enough to know how Ξ acts on this graph image, and this is how PSurface is implemented. Remember that a *graph* $\mathcal{G}$ is a finite set of vertices V together with a set E of unordered pairs of vertices which are called *edges*. A graph can be given a geometry in $\mathbb{R}^d$ by associating each vertex $v \in V$ with a position $p(v) \in \mathbb{R}^d$, and each edge $e = (v_0, v_1)$ with the line segment from $p(v_0)$ to $p(v_1)$. If $d \in \{1, 2\}$ and none of these segments intersect except at vertices then the graph together with the embedding into $\mathbb{R}^d$ is called a straight-line plane graph [7]. Each plane graph divides $\mathbb{R}^d$ into a set of d-dimensional *regions*. If each region except for the unbounded one is a simplex, $\mathcal{G}$ is called a *triangulation*. More generally we can define embeddings of graphs into simplicial surfaces.

Definition 2.4. *Let $\mathcal{G} = (V, E)$ be a graph and S a simplicial surface. With each vertex $v \in V$ associate a position $p(v) \in S$, and with each edge $e = (v_0, v_1) \in E$ associate a set of open line segments $\eta_e = \{(e_0, f_0), \ldots, (e_{n_e}, f_{n_e})\}$ with $(e_i, f_i) \subset S$ for all $i = 0 \ldots, n_e$. If*

- *$p(v_0) = e_0$, $p(v_1) = f_{n_e}$, $f_i = e_{i+1}$,*
- *for each (e_i, f_i) there exists an element T of S such that $(e_i, f_i) \subset T$,*
- *for any two segments (e_i, f_i) and (e'_j, f'_j) we have $(e_i, f_i) \cap (e'_j, f'_j) = \emptyset$,*

then (p, η) with $\eta = \{\eta_e \mid e \in E\}$ is called a piecewise straight embedding of $\mathcal{G}$ in S. A piecewise straight embedding is called minimal, *if for each triangle $T \in S$ and each edge $e \in E$ there is at most one segment $(e_i, f_i) \in \eta_e$ with $(e_i, f_i) \subset T$.*

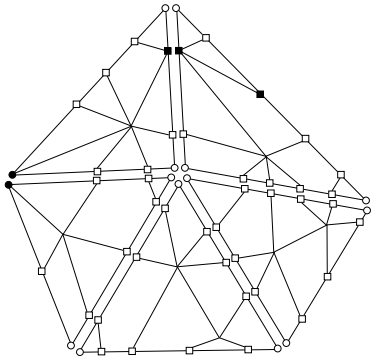

Figure 3: Implementation of a piecewise linear mapping Ξ as a graph on S_1. Graph node types are *corner* (●), *touching* (■), *intersection* (□), *ghost* (○), and *interior* (no symbol).

A graph embedded into a triangulated surface S subdivides S into regions. In general these regions do not coincide with the triangles of S (Figure 3).

Piecewise linear mappings induce graphs on the domain surface S_1. For a formal statement of this, we first define the preimage.

Definition 2.5. *Let $\Xi : S_1 \to S_2$ be piecewise linear and $A \subseteq S_2$. We define the preimage of A under Ξ as*

$$\Xi^{-1}(A) = \{x \in S_1 \mid \Xi(x) \in A\}.$$

The vertices $\mathcal{V}_2$ and the edges $\mathcal{E}_2$ of the surface S_2 form the *edge graph* $\mathcal{G}_2$ of S_2. Its preimage $\mathcal{G}_\Xi = \Xi^{-1}(\mathcal{G}_2)$ under the piecewise linear map $\Xi : D \subset S_1 \to S_2$ is a minimal piecewise straight embedding of $\mathcal{G}_2$ in S_1 (Figures 1 and 2).

Theorem 2.1. *Let S_1, S_2 be two simplicial surfaces and let $\Xi : D \subset S_1 \to S_2$ be a surjective, continuous, piecewise linear map. Denote by $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$ the edge graph of S_2. Then $\Xi^{-1}(\mathcal{G}_2)$ is a minimal piecewise straight embedding of $\mathcal{G}_2$ in S_1, with*

$$p(v) = \Xi^{-1}(v)$$

for all vertices $v \in \mathcal{V}_2$, and

$$\eta_e = \{T \cap \Xi^{-1}(e) \mid T \in \mathcal{T}_1, T \cap \Xi^{-1}(e) \neq \emptyset\}$$

for all edges $e \in \mathcal{E}_2$.

This graph embedding contains all relevant information about Ξ. Therefore, the mapping $\Xi : D \subset S_1 \to S_2$ can be stored using a data structure for graphs on S_1. This is the approach taken by PSurface, and we will now describe how this data structure looks like.

To reduce complexity, the data structure for $\mathcal{G}_\Xi$ on S_1 consists of data structures for plane graphs on each element of S_1. Some additional bookkeeping then allows to reconstruct the complete graph $\mathcal{G}_\Xi$ from its restrictions to the individual elements. For each element $T \in \mathcal{T}_1$ we store a set of nodes, each storing its local position on T, its position on S_2, and a list of its neighbors in

cyclic order. We introduce five different types of graph nodes (Figure 3). The first three correspond to vertices of S_2.

Interior nodes: Nodes in the interior of an element T.

Corner nodes: Nodes on the corners of T

Touching nodes: Nodes on an edge $k \subset \partial T$, but not on a corner of T.

Nodes on edges or corners have copies on all elements that share the common edge or corner. The same holds for the following two node types which are needed to make the restriction of $\mathcal{G}_\Xi$ on each element a graph in its own right, and to allow the evaluation of Ξ on any point of an element without having to use information from other elements.

Intersection nodes: For any edge $e = (v_0, v_1) \in \mathcal{E}_2$, the corresponding preimage $\Xi^{-1}(e)$ is generally not contained in a single element of S_1. Instead, from Theorem 2.1 we know that it consists of a sequence of segments (e_i, f_i), $0 \leq i \leq n_e$, with each segment (e_i, f_i) being contained in a single element. With the exception of e_0 and f_{n_e}, the points e_i, f_i do not correspond to vertices of the edge graph $\mathcal{G}_2$. Still, the data structure needs to keep them as vertices anyways, in order to have a consistent graph data structure on each element of S_1. Since, by Theorem 2.1, the embedding of $\mathcal{G}_2$ into S_1 is minimal, *intersection* nodes cannot occur in the interior of triangles.

Ghost nodes: If a corner c of an element $T \in \mathcal{T}_1$ does not get mapped onto a vertex of S_2, a *ghost node* without any neighbors is added at c.

Each node p stores its image $\Xi(p) \in S_2$ by storing the index of a triangle $T \in \mathcal{T}_2$ with $\Xi(p) \in T$ and local coordinates of $\Xi(p)$ with respect to T. Finally, each triangle $T \in \mathcal{T}_1$ keeps three arrays containing the nodes on the three triangle edges in cyclic order, and each node on an edge knows its index in the corresponding array. That way, corresponding nodes on adjacent triangles are identified, and it is possible, in a given graph data structure, to efficiently track preimages of edges of S_2 across multiple triangles of S_1.

Remark 2.1. While globally $\Xi^{-1}(\mathcal{G}_2)$ is a triangulation of its domain of definition, its restrictions to individual triangles of $\mathcal{T}_1$ need not be (Figure 3). However, for efficient point-location it is necessary that the graph on each triangle be a triangulation (see [5] and Section 2.3). Therefore, we add additional graph edges to the data structure such that $\Xi^{-1}(\mathcal{G}_2)$ is a triangulation on each $T \in \mathcal{T}_1$. We call this the *triangular closure*.

The data structure simplifies considerably if $d = 1$. Then the elements of S_1 and S_2 are simply line segments connecting pairs of vertices in $\mathbb{R}^2$. On each segment of S_1, the graph $\mathcal{G}_\Xi$ is merely a linear sequence of nodes ordered by local position $s \in [0, 1]$ and with adjacent nodes (possibly) connected by a graph edge. The simpler structure also limits the set of node types that can occur in $\mathcal{G}_2$. As there are no edges in S_1, node types that live on edges do not occur. Only *interior*, *corner*, and *ghost nodes* are needed. For the same reason, the arrays storing the nodes on surface edges in cyclic order do not appear in the one-dimensional data structure.

2.3 Evaluating a piecewise linear map

Given a consistent data structure as described above and a point $p \in S_1$ specified by a triangle $T \in \mathcal{T}_1$ and local coordinates ξ on T, the map Ξ can be evaluated at p in two steps. If p is contained in the domain of Ξ, then it must be contained in a region r of the graph $\mathcal{G}_\Xi = \Xi^{-1}(\mathcal{G}_2)$. First, this region r (with corners $c(r)$) is determined using a point-location algorithm, and the barycentric coordinates $\zeta_{c(r)}$ of p with respect to r are computed. If $d = 1$ the element T is a line segment, and the graph data structure on T has been prepared for point-location by sorting the graph nodes in increasing order of their local coordinates. Then the region $r = [v_i, v_{i+1}]$ containing p is found in logarithmic time by a bisection search.

If $d = 2$ we need a point location in a two-dimensional plane graph. Remember that we added the additional graph edges to make all regions of $\mathcal{G}_\Xi|_T$ triangles (Remark 2.1). For point-location in triangulations we use the randomized version of the algorithm presented by Brown and Faigle [5]. It is simple to implement and its expected run-time is in $O(\sqrt{n_e})$, with n_e the number of edges in the triangulation.

After the point-location step we have a triangular region r of $\mathcal{G}_\Xi$ on T with $p \in r$, barycentric coordinates ζ of p with respect to r and the set of corners $c(r)$ of r. By construction, the images of all corners of r are on the same element T_2 of S_2, and their local positions of their images $\Xi(c(r))$ with respect to T_2 are stored in the node data structure. Hence the local position of $\Xi(p)$ with respect to T_2 can be computed by linear interpolation

$$\Xi(p) = \sum_{c(r)} \zeta_{c(r)} \Xi(c(r)).$$

By this algorithm we have proved the following result.

Theorem 2.2. *Ξ is completely determined by the presented data structure.*

3 Constructing a piecewise linear mapping

There are various ways to set up a piecewise linear mapping, and these are related to different applications of PSurface. You can either directly construct a `PSurface` object by means of certain primitives, construct it by projection of S_1 onto S_2; or by starting with two identical surfaces $S_1 = S_2$ and the identity map, and then coarsening S_1.

3.1 Direct construction

PSurface allows to set up a map Ξ from scratch by explicitly specifying the vertices and triangles of S_1 and S_2, and their relations under Ξ. This can be used, for example, to read mappings provided in some file format.[1] Also, it provides a clear interface to PSurface creation for those wishing to implement their own construction algorithms.

[1] The PSurface library itself provides reading and writing facilities for `PSurface` objects in the AmiraMesh format [21]. The precise format is described in a text file distributed with the source code.

Direct creation of a `PSurface` object is implemented by means of a factory class. The `PSurfaceFactory` is used internally by all other construction methods. It accepts the specification of surfaces S_1 and S_2, and a graph on S_1 describing a piecewise linear map. Upon a call to the method `createPSurface`, the factory hands back a pointer to the newly created object.

Creating the surfaces S_1 and S_2 is done by providing positions for the vertices, and $d+1$-tuples of vertex indices for the elements. Since the data structure for S_2 does not contain information about Ξ it is also possible to build it outside of the factory. A pointer to S_2 is then handed to the `PSurfaceFactory`. At this point, the graph on S_1 is the empty graph, i.e., there is no mapping defined anywhere on S_1.

The actual mapping is specified in a separate step. In the simplest case, S_2 is a logical refinement of S_1 under Ξ, i.e., for each element $T_2 \in S_2$, the preimage $\Xi^{-1}(T_2)$ is entirely contained in a single element T_1 of S_1. In this case, vertices of S_1 in the domain $D \subset S_1$ of Ξ are mapped to vertices of S_2, and preimages of edges of S_2 do not cross edges of S_1. Graph nodes of $\mathcal{G}_\Xi$ can only be of *corner*, *touching*, or *interior* type, and the factory class provides a method to enter such nodes into the graph data structure on S_1. *Interior nodes* get entered once for the element they reside on. *Corner* and *touching nodes* get entered automatically on each element that borders the vertex and edge, respectively, that the node resides on. The method returns a handle to the newly created set of nodes. A second method allows to insert edges into the graph. Since, by assumption, an edge of $\mathcal{G}_\Xi$ is contained in a single element it can be inserted simply by giving the element number and handles to its two nodes. A convenience method allows to insert elements of S_2, which amounts to inserting their edges, if not already present.

If S_2 is not a logical refinement of S_1, more information is needed. First of all, *ghost nodes* appear at those vertices of S_1 in D that do not get mapped to a vertex of S_2. *Ghost nodes* can be added to the factory by calling the method `insertGhostNode`. The arguments are the vertex number on S_1, and an element number and local coordinates on S_2. If the vertex of S_1 is to be mapped onto an edge of S_2 any one of the adjacent triangles will do. Secondly, an edge $e = (p, q)$ of S_2 may now connect vertices whose preimages are not on the same element of S_1. In this case, all crossings of $\Xi^{-1}(e)$ with edges of S_1 need to be specified. This leads to the insertion of *intersection nodes* at the triangle boundaries. The preimage of an edge is inserted using the method `insertEdge`. Its arguments are the nodes on S_1 corresponding to p and q, an array of the crossed edges of S_1, and local coordinates of the *intersection nodes* on these edges. If $\Xi^{-1}(e)$ crosses a vertex of S_1 (on one-dimensional surfaces, this is even a certainty for edges visiting more than a single element of S_1), a *ghost node* is automatically inserted at the vertex instead of an intersection node, unless that *ghost node* has already been inserted previously by hand.

After all nodes and edges have been inserted, a finalization method creates the actual `PSurface` object. Among other things it computes the triangular closure (Remark 2.1), sorts the edges of each node in cyclic order,[2] and builds up the arrays that store the nodes on the element edges.

[2]For increased robustness this cyclic ordering is not computed by comparing angles. Instead, a graph algorithm is used to determine a longest path in the subgraph of all neighbors of the node.

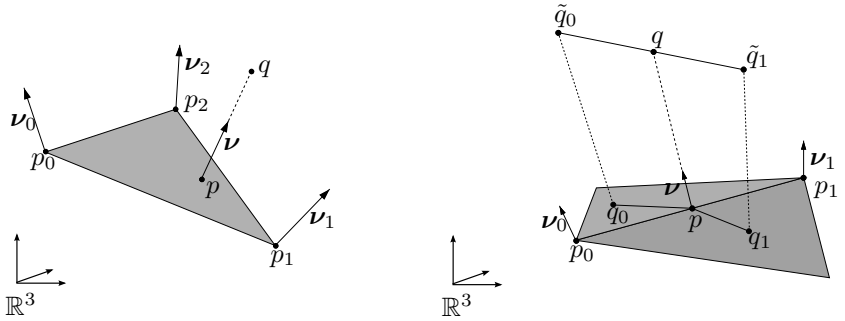

Figure 4: Backward normal projection from a point q onto a triangle (left), and from an edge $(\tilde{q}_0, \tilde{q}_1)$ onto an edge (p_0, p_1) (right).

3.2 Projection

For two given simplicial surfaces S_1 and S_2 a mapping can also be constructed algorithmically by projecting S_1 onto S_2 along a piecewise linear vector field on S_1 given by directions at the vertices of S_1. This is an important ingredient, e.g., of algorithms for computing contact problems in continuum mechanics (see Section 4.2). Frequently, the projection direction is chosen as a piecewise linear averaged surface normal of S_1, but other sufficiently well-behaved vector fields on S_1 can also be used.

More formally, let $\boldsymbol{\nu} : \mathcal{V}_1 \to \mathbb{R}^{d+1}$ be a set of vectors associated to the vertices of S_1. We extend $\boldsymbol{\nu}$ by linear interpolation to a continuous vector field on S_1, which we denote again by $\boldsymbol{\nu}$. Our aim is to construct a set $D \subseteq S_1$ and a mapping $\Phi : D \to S_2$ such that D is as large as possible and

$$\Phi(p) - p = \mu \boldsymbol{\nu}(p) \qquad \text{for all } p \in D, \tag{1}$$

with p and $\Phi(p)$ interpreted as points in $\mathbb{R}^{d+1}$ and $\mu \in \mathbb{R}_0^+$ depending on p.

Assume that surfaces S_1 and S_2 and the vector field $\boldsymbol{\nu}$ are given. The construction of the set D and the mapping Φ consists of three steps.

1. *Computing $\Phi^{-1}(q)$ for all vertices $q \in \mathcal{V}_2$*
   For each $q \in \mathcal{V}_2$ we have to find a $p \in S_1$ such that

   $$q - p = \mu \boldsymbol{\nu}(p). \tag{2}$$

   We can then define $\Phi^{-1}(q) := p$. For each $q \in \mathcal{V}_2$ and each element T of S_1, (2) is a nonlinear system of equations for the barycentric coordinates $\lambda = (\lambda_0, \ldots, \lambda_{d+1})$ of p on T and the distance μ. For $d = 2$ it reads

   $$0 = p_2 - q + \lambda_0(p_0 - p_2) + \lambda_1(p_1 - p_2) + \mu\lambda_0(\boldsymbol{\nu}_0 - \boldsymbol{\nu}_2) + \mu\lambda_1(\boldsymbol{\nu}_1 - \boldsymbol{\nu}_2) + \mu\boldsymbol{\nu}_2, \tag{3}$$

   where $p_0, \ldots, p_d$ are the corners of T and $\boldsymbol{\nu}_0, \ldots, \boldsymbol{\nu}_d$ are the directions at these corners. A point p with (1) exists on T if (3) has a solution with $\lambda_0, \lambda_1, \mu \geq 0$ and $\lambda_0 + \lambda_1 \leq 1$. This system can be solved conveniently using a Newton solver.

   To find preimages for all vertices of S_2 we loop over the elements of S_1. For each element $T \in \mathcal{T}_1$ we use an octree to efficiently find all vertices of

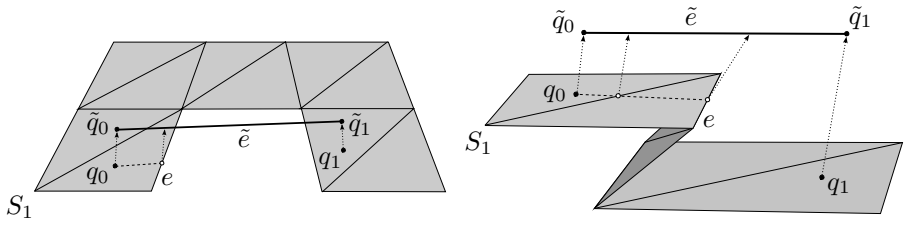

Figure 5: In certain cases the edge projection algorithm fails: the edge preimage may leave S_1 (left), or the projection may not be continuous (right).

S_2 reachable in the direction of $\boldsymbol{\nu}$ from T. A vertex v has a preimage p on T if (1) has a solution for p in T and there is no other solution $(\tilde{p}, \tilde{\mu})$ on an element $\tilde{T}$ with $\tilde{\mu} < \mu$. To increase robustness we additionally compare the normal $\tilde{\boldsymbol{\nu}}$ of S_2 at q and the segment $w = q - p$, and accept a preimage only if $\langle w, \tilde{\boldsymbol{\nu}} \rangle < 0$.

2. *Computing $\Phi(v)$ for all vertices $v \in \mathcal{V}_1$*
   At this stage all vertices of S_2 in the range of Φ appear as nodes in the graph on S_1. We have to add additional *ghost nodes* at those vertices of S_1 that are not mapped onto vertices of S_2. This is comparatively easy as it does not involve solving nonlinear equations. Given a vertex $v \in \mathcal{V}_1$, its image on S_2 can be found by considering the ray r in direction $\boldsymbol{\nu}(v)$ beginning in v, and looking for intersections of r with elements of S_2. Potential candidate elements for intersections are produced efficiently using an octree. If r intersects more than one triangle of S_1 the intersection closest to v is chosen. If no intersection is found then $\Phi(v)$ will be left undefined by modifying D such that $v \notin D$. If v is mapped onto a vertex of S_2 then it has already been treated in Step 1 and the data structure already contains a node for it. Otherwise a *ghost node* is inserted.

3. *Adding the graph edges*
   In order to enter an edge $\tilde{e} = (\tilde{q}_0, \tilde{q}_1)$ of S_2 into the graph on S_1 we try to 'walk' on S_1 along $\Phi^{-1}(\tilde{e})$ from $q_0 = \Phi^{-1}(\tilde{q}_0)$ to $q_1 = \Phi^{-1}(\tilde{q}_1)$. Since q_0 and q_1 will generally not be on the same triangle of S_1, we have to find the points where the path from q_0 to q_1 crosses edges of S_1. Let $T \in \mathcal{T}_1$ be the current triangle in this walking process. For an edge $e = (p_0, p_1)$ of T we have to check whether there are points $p \in e$ and $q \in \tilde{e}$ with $q - p$ collinear to $\boldsymbol{\nu}(p)$ (Figure 4, right). This can be formulated as a nonlinear system of equations
   $$p(\lambda) + \mu \boldsymbol{\nu}(\lambda) = q(\eta) \tag{4}$$
   for three variables $\lambda, \mu, \eta \in \mathbb{R}$ and
   $$p(\lambda) = p_0 + \lambda(p_1 - p_0), \quad \boldsymbol{\nu}(\lambda) = \boldsymbol{\nu}_0 + \lambda(\boldsymbol{\nu}_1 - \boldsymbol{\nu}_0), \quad q(\eta) = \tilde{q}_0 + \eta(\tilde{q}_1 - \tilde{q}_0),$$
   which can be solved with a damped Newton algorithm. We have found an intersection if (4) has a solution with $0 \le \lambda, \eta \le 1$ and $0 \le \mu$. This intersection is then inserted as an *intersection node* and the procedure is continued on the triangle which borders T on e.

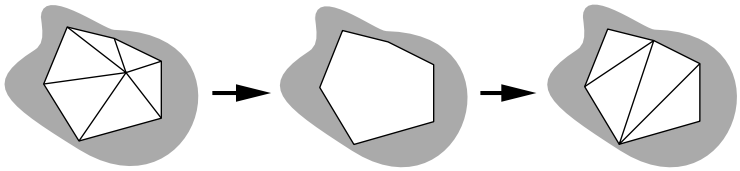

Figure 6: The point-removal primitive

There are various corner cases where this approach fails. For example, even though $\Phi^{-1}(\tilde{e})$ intersects e, there may not be another triangle that borders T across e (Figure 5, left). Alternatively, there may be such a triangle, but it may be hidden from S_2 by T (Figure 5, right). In both cases, the edge insertion algorithm currently simply aborts and does not insert the edge $\tilde{e}$ at all. For the future one may consider adding smarter algorithms that can handle the first case. In the second case, the projection in the direction of $\boldsymbol{\nu}$ is not continuous, and hence storing it is beyond the scope of the PSurface data structure.

Assuming that the Newton solvers for (3) and (4) terminate after a constant number of iterations and that each edge of S_2 crosses only a constant number of elements from S_1, the projection algorithm described above requires an expected $O(N_{\max} \log N_{\max})$ time, with $N_{\max} = \max(|\mathcal{V}_1|, |\mathcal{V}_2|)$. In the worst case the time needed is quadratic. This is optimal, because there are configurations where (almost) every edge of S_1 gets mapped onto (almost) every edge of S_2, leading to a quadratic number of *intersection nodes*.

3.3 Constructing parametrizations by surface simplification

In this section we give an algorithm that constructs piecewise linear mappings by surface simplification. It appeared originally in [18]. Given a not too coarse triangulated surface S_2, the result is a coarser surface S_1 and a parametrization Ξ of S_2 over S_1, i.e., a homeomorphism from S_1 to S_2. The parametrization should be of good quality, a criterion to be specified more formally in Section 3.3.2.

Parametrizations of a simplicial surface over a coarser copy of itself are useful for the treatment of domain boundaries in finite element problems. These boundaries are sometimes given as triangulated surfaces of very high resolution, and need to be coarsened before being able to serve as input for a mesh generator [18]. The geometric information lost by coarsening can be regained later during mesh refinement if a parametrization of the fine boundary over the coarse boundary is available.

The idea of constructing a parametrization function by surface simplification originates in computer graphics [16], where it is used for a variety of problems, such as surface remeshing and texture map generation [17]. Our algorithm consists of two steps:

1. construct the base domain surface S_1 and a parametrization function $\Xi : S_1 \to S_2$ by means of a simplification algorithm (Section 3.3.1),
2. smooth Ξ to obtain refined surfaces of optimal quality (Section 3.3.2).

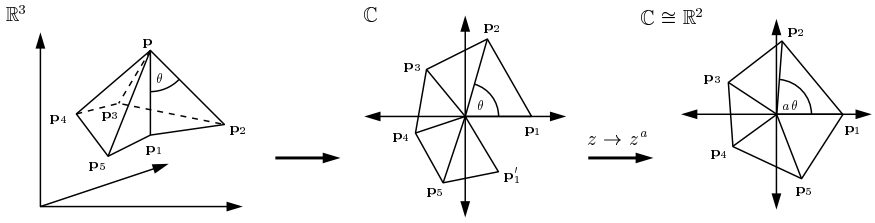

Figure 7: The polar map: The star of p is first cut along the edge (p, p_1) and flattened out into the complex plane. The conformal map $z \to z^a$, $a = 2\pi / \sum_i \theta_i$, then stretches it out so that it covers one full circle.

We describe the algorithm assuming that S_2 is a piecewise linear manifold. The algorithm also works well in the non-manifold case. See [18] for details.

3.3.1 Surface simplification

Let S_2 be a simplicial surface of sufficiently high resolution. We initialize the algorithm by setting $S_1^0 = S_2$ and $\Xi^0 : S_1^0 \to S_2$, $\Xi^0 = \mathrm{Id}$. The algorithm will construct a sequence of simplicial surfaces S_1^i and functions $\Xi^i : S_1^i \to S_2$ such that S_1^i contains less elements than S_1^{i-1} and $\Xi^i(S_1^i) = S_2$ for all $0 \leq i \leq N$.

A surface simplification algorithm consists of two parts. The first is a geometric operation that allows the controlled local reduction of surface complexity. We chose point removal since it does not create new vertices and is easily extendible to non-manifold surfaces. Point removal works by repeatedly removing a vertex and its neighboring triangles, and retriangulating the resulting hole (Figure 6).

The second ingredient is a scalar oracle that allows to rank the different possible simplification steps according to the error they would introduce. Many different strategies have been described [12, 18]. We combine the following aspects:

- monitor a modified Hausdorff distance between different simplification stages to control geometric error,
- favor steps that remove triangles with a high aspect ratio and introduce triangles with a low aspect ratio, to maintain a high-quality triangulation,
- reject simplification steps that lead to surface intersections,
- penalize long edges, to obtain surfaces of uniform element size.

Let $i \in \mathbb{N}$ be the current simplification step and assume that v_i is the current best vertex according to the error oracle. We define $\mathrm{St}\, v_i$, the star of v_i, as the set of all edges and elements that contain v_i. Speaking first strictly in terms of the domain surface S_1^i, a simplification step consists of removing v_i and its star from S_1^i. The resulting hole is then filled by a constrained Delaunay triangulation with a set of elements P_i. The resulting surface S_1^{i+1} has one vertex and two elements less than S_1^i.

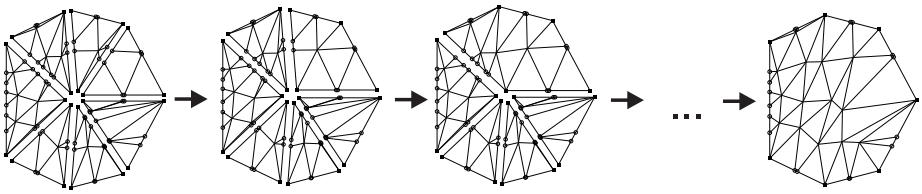

Figure 8: The flattened star of a vertex is merged into a single polygon.

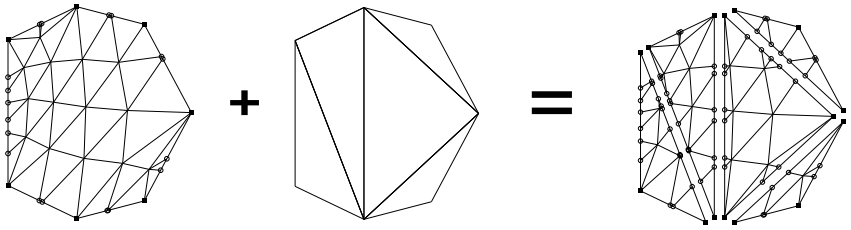

Figure 9: The polygon is cut along the new triangulation.

At the same time we have to make sure that the mapping defined on $\operatorname{St} v_i$ is correctly transferred to P_i. This means that the graphs defined on the elements of $\operatorname{St} v_i$ must be transferred to the elements of P_i. For this, after having removed $\operatorname{St} v_i$ from S_1^i, we flatten it out into $\mathbb{R}^d$. This is easy if $d = 1$. If $d = 2$ we use the *polar map* introduced in [8]. It maps $\operatorname{St} v_i$ bijectively and conformally onto a star-shaped polygon in $\mathbb{R}^2$ (Figure 7). We then merge the graphs on the flattened elements of $\operatorname{St} v_i$ into a single plane graph on the polygon consisting of the union of the elements of $\operatorname{St} v_i$ (Figure 8).

This polygon implements that part of the parametrization function Ξ that had formerly been implemented by the triangles in $\operatorname{St} v_i$. We then repartition the graph according to the new triangulation P_i obtained by Delaunay triangulation (Figure 9). In the process, edges are cut and *intersection nodes* are added, and *interior nodes* may turn into *touching nodes*. The result is a set of graphs for the triangles that are inserted into the domain surface.

Denoting by n the number of vertices of S_2, the surface simplification algorithm needs only $O(n \log n)$ steps to remove a constant fraction of the vertices. However, due to the overhead for maintaining a consistent parametrization at each step the constant is fairly large. The space requirements remain linear in the number of surface vertices.

3.3.2 Parametrization smoothing

The mappings constructed by the simplification algorithm are frequently of poor quality. By this we mean that the maximal relative condition number

$$\kappa_\Xi = \max_{A \in \mathcal{A}_\Xi} \|A\|_2 \|A^{-1}\|_2$$

can get very large. Here, A is the matrix corresponding to one linear piece of a piecewise linear map Ξ, and $\mathcal{A}_\Xi$ is the set of all these matrices. As a consequence, surfaces obtained by refining S_1 and using Ξ to place the newly

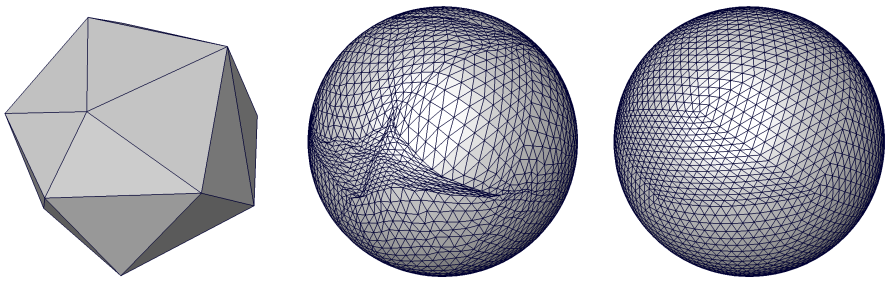

Figure 10: The edge relaxation algorithm significantly increases the quality of the refined mesh. The figure shows a surface constructed by simplification of a sphere (left) and a refined surface without (center) and with (right) smoothing.

inserted vertices can have large triangle aspect ratios (see Figure 10). In the context of finite element methods this is very undesirable. We will now shortly describe an algorithm that considerably improves the quality of the refined grids.

Assume that we have a method that computes high-quality parametrizations of surface patches over convex polygons. Various algorithms for this have been proposed in the literature. It follows from Gauß' Theorema Egregium that such a parametrization can in general not be isometric, i.e., distortion-free. We use the shape-preserving parametrization introduced by Floater [9]. Inspired by planar graph-drawing, it places the vertices in the polygon such that each vertex is a convex combination of its neighbors. The weights are chosen such that if there is an affine mapping of the surface onto the polygon, then this mapping is reproduced by the parametrization algorithm. Floater [9] showed that such mappings always exist, and that they can be computed by solving a sparse non-symmetric system of equations. This type of mapping is widely used in surface parametrization algorithms [10, 11].

We can use this method to recompute the parametrization on each element of S_1, while keeping the values of Ξ on the edges of S_1 fixed. This will already greatly improve the parametrization quality. A further improvement can be achieved if the domains are enlarged. In PSurface we have implemented smoothing on quadrilaterals. Let T_1, T_2 be two triangles of S_1 that share a common edge. Consider them to be separated from the remaining surface. They can then be 'folded' isometrically to lie in a single plane. The common edge is removed and the graphs on T_1 and T_2 are merged into a single graph on the quadrilateral formed by T_1 and T_2. The Floater parametrization method can now be applied to this quadrilateral. Afterwards the common edge is reintroduced and the graph is cut along this edge. Merging and cutting of graphs was already a part of the surface simplification algorithm of Section 3.3.1, and the code can be reused.

To achieve a global improvement of the parametrization quality we now loop over all edges of S_1 with two adjacent triangles. For each such edge we smooth the quadrilateral formed by the two adjacent triangles. The result after one iteration is a marked improvement of the parametrization quality. Since the smoothing across edges introduces a global coupling, it pays to loop several times over all edges. In practice, a satisfying parametrization quality is reached in less than ten iterations. See [18] for more information.

4 Integration in Dune

Several features of PSurface are useful in the context of the numerical analysis of partial differential equations (PDEs). For these cases, PSurface can be used by the Distributed and Unified Numerics Environment (Dune). Dune is a set of C++ libraries for grid-based methods for solving PDEs [4]. It has been designed with flexibility as the main goal, and consists of interface definitions to core components such as grids, linear algebra, or shape functions. These components can then be implemented in various ways, allowing the user to always choose the most appropriate implementation for the task at hand. PSurface is currently used by the Dune modules `dune-grid` and `dune-grid-glue`. The `dune-grid` module provides the interface to finite element grids, together with various implementations of this interface. `dune-grid-glue` provides infrastructure for the coupling of several grids.

4.1 Parametrized boundaries with `dune-grid`

In the numerical solution of PDEs, the shape of the domain may be given as a high-resolution triangulated surface. Such a surface may need to be coarsened before grid generation is possible. The geometric information lost should be regained when adaptively refining the grid. For this, the construction and handling of a mapping from the coarse grid boundary to the original boundary is necessary. This is provided by the PSurface library.

Several of the Dune grid implementations can handle the refinement of grids with a parametrized boundary, provided the geometric information is given. `dune-grid` contains code that, given that the PSurface and AmiraMesh libraries are installed on the system, loads a `PSurface` from an AmiraMesh file into a `PSurfaceBoundary` object. This object then hands out information about the number of boundary segments, their types and vertices. Most importantly, for each boundary segment it hands out an object of a class derived from `Dune::BoundarySegment`. These objects implement mappings $f_i : \mathbb{R}^{d_b} \to \mathbb{R}^{d_w}$, with d_b the dimension of the grid boundary and d_w the dimension of the world space, and can be handed directly to the grid factory class at grid creation time.[3] Grid managers that are able to will then honor the geometric information for grid refinement (see Figure 12).

As a variation of this idea, it is also possible to parametrize the elements of certain grids themselves. Consider a two-dimensional grid embedded in a three-dimensional space. As a finite element grid it will be piecewise linear (or piecewise polynomial at most), but in many cases it will be intended to approximate a smooth surface. Hence, upon grid refinement one would expect the grid to approximate the smooth surface better. This is only possible if the shape of the actual surface is known as a function on the coarsest grid. If the target surface is not given analytically but by a second triangulated surface, then this function can be implemented using PSurface. Unlike the previous case, the parametrizing function is now defined on the entire grid instead of only on the grid boundary.

[3] At the time of writing, the actual handling of parametrized boundaries in `dune-grid` has not been standardized yet. Therefore the details are still subject to change. Please consult the `dune-grid` documentation at `www.dune-project.org` for up-to-date information.

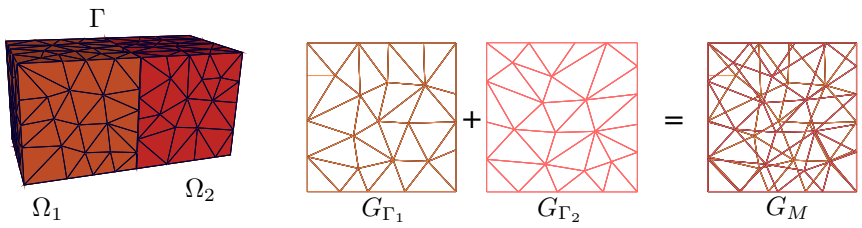

Figure 11: Two domains Ω_1 and Ω_2 that meet at a common interface Γ (left). The restrictions of the two grids on Γ (center). Together they form the set of remote intersections G_M (right).

4.2 Domain decomposition methods with `dune-grid-glue`

A different use of PSurface occurs in the treatment of domain decomposition problems. In the simplest case, a partial differential equation is to be solved on two subdomains $\Omega_1, \Omega_2 \subset \mathbb{R}^d$, connected through suitable coupling conditions on a common interface Γ. This interface is assumed to be a $d-1$-dimensional manifold. Two grids G_1, G_2 on Ω_1 and Ω_2, respectively, induce trace grids $G_{\Gamma,1}$, $G_{\Gamma,2}$ on Γ, which are not necessarily related to each other in any way. Various domain decomposition methods, in particular the popular mortar method, require the evaluation of integrals of the form

$$m(v,w) = \int_\Gamma vw\,dx,$$

where v and w are finite element functions with respect to $G_{\Gamma,1}$ and $G_{\Gamma,2}$, respectively [22]. To compute these integrals exactly, the domain of integration Γ needs to be split up in intersections $I_{ij} \subset \Gamma$ such that each I_{ij} is the set intersection of two elements $T_i \in G_{\Gamma,1}$, $T_j \in G_{\Gamma,2}$, (Fig. 11). If the grid elements are convex polyhedra, then each intersection is a convex polyhedron itself and can be decomposed in simplices. On each such simplex, the product vw is a polynomial and a quadrature rule is available, hence vw can be integrated exactly. The total integral is given as the sum of the integrals over the individual intersections I_{ij}.

In more general cases, the two domains may be separated by a positive gap. In this case, the coupling boundaries $\Gamma_1 \subset \partial\Omega_1$ and $\Gamma_2 \subset \partial\Omega_2$ are identified through a homeomorphism $\Phi : \Gamma_1 \to \Gamma_2$ (Fig. 12, left). The integrals used in the mortar method take the form

$$m_\Phi(v,w) = \int_{\Gamma_1} v(x)w(\Phi(x))\,dx. \tag{5}$$

Note that this degenerates to the previous simpler case if $\Gamma_1 = \Gamma_2$ and $\Phi = \mathrm{id}$. The intersections I_{ij} are now defined as

$$I_{ij} = T_i \cap \Phi^{-1}(T_j).$$

In Dune, the task of computing the intersections I_{ij} from the coupling boundary grids $G_{\Gamma,1}$ and $G_{\Gamma,2}$ and the function Φ is handled by the module

`dune-grid-glue` [2]. The set of intersections can be accessed by `stl`-style iterators. Each intersection provides information such as its embedding in T_i and T_j, its embedding in the world space $\mathbb{R}^d$, normal vectors, etc.

The general problem of computing intersections between two grids is too diverse to be handled efficiently by a single implementation. A very general implementation will not be efficient in simple cases, while an efficient implementation cannot be sufficiently general. The software design of `dune-grid-glue` therefore follows the general DUNE philosophy of declaring interfaces and allowing the user to choose between various implementations of this interface.

One such implementation uses the PSURFACE library. During setup, the coupling grids $G_{\Gamma,1}$ and $G_{\Gamma,2}$ are extracted and a new `PSurface` object is created with $S_1 = G_{\Gamma,1}$ and $S_2 = G_{\Gamma,2}$. If $G_{\Gamma,1}$ or $G_{\Gamma,2}$ contain elements that are not simplicial, these are triangulated to obtain valid input surfaces for PSURFACE. A mapping $\Phi : S_1 \to S_2$ is then constructed using the projection algorithm of Section 3.2. The default projection direction is the field of surface vertex normals of S_1, but user-specified directions are also possible. Note that this yields the identity mapping when $\Gamma_1 = \Gamma_2$. The intersections I_{ij} can then be read off as the regions of the graph $\mathcal{G}_\Xi$ on S_1 implementing Φ. Since PSURFACE triangulates all regions of $\mathcal{G}_\Xi$ to facilitate point location queries, all intersections are simplices. `dune-grid-glue` then implements the various iterators over the intersections.

4.3 Example application: two-body contact

The possibilities of PSURFACE in DUNE are demonstrated by the following short example. Consider a two-body contact problem occurring as part of a biomechanical application. We simulate the left distal femur and proximal tibia from the Visible Human data set [1]. The data was segmented and a high-resolution boundary surface was extracted. The femur surface consisted of 14 468 triangles, and the tibia surface of 14 902 triangles. They were simplified as described in Section 3.3 to yield coarse surfaces with 532 triangles for the femur and 444 triangles for the tibia. The AMIRA [20] grid generator produced two tetrahedral grids with 1 328 and 1 044 elements, respectively (Figure 12).

We modeled bone with an isotropic, homogeneous, linear elastic material. The bottom section of the proximal tibia was clamped and a downward displacement of 6 mm was prescribed on the upper section of the femur. The parts of the bones usually covered with articular cartilage were marked as the coupling boundaries Γ_1 and Γ_2, but the actual coupling boundary was smaller, because the normal projection Φ could only be constructed on a part of the prescribed coupling boundary.

Mutual nonpenetration of the two objects was modeled by a linearized inequality constraint. Let $\mathbf{u}_i : \Gamma_i \to \mathbb{R}^3$, $i = 1, 2$, be the traces of the displacement functions on the contact boundaries. We required

$$\langle \mathbf{u}_1 - \mathbf{u}_2 \circ \Phi, \mathbf{n} \rangle \leq g, \tag{6}$$

where $\mathbf{n}$ is the outer unit normal on Γ_1 and $g : \Gamma_1 \to \mathbb{R}$ is the normal distance in the undeformed state. The mortar discretization of (6) is

$$\int_{\Gamma_1} \langle \mathbf{u}_1 - \mathbf{u}_2 \circ \Phi, \mathbf{n} \rangle \mu \, ds \leq \int_{\Gamma_1} g\mu \, ds \tag{7}$$

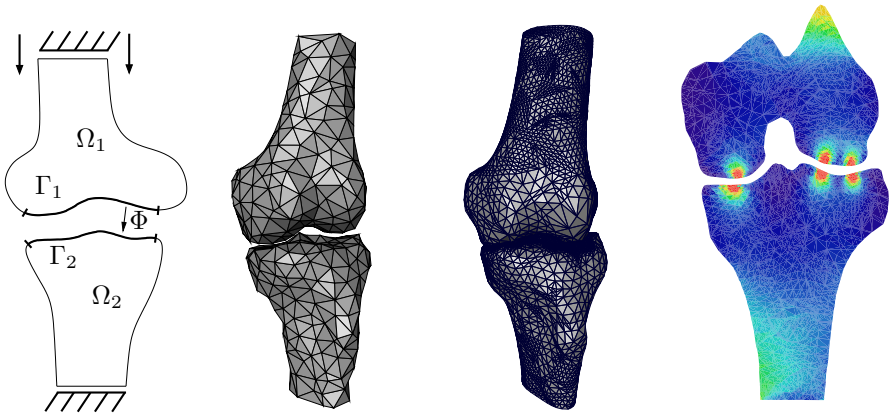

Figure 12: Two-body contact problem. Note the stress peaks on the condyles due to the contact, and how the refined grid boundary approximates the smooth domain boundary.

for a set of test functions μ from a suitable test function space M. Expressing this algebraically in terms of basis functions leads to terms of the form (5). See [19] for the details. Figure 12 show a result of the computation after three steps of adaptive refinement. It can be seen that the refined grid boundary approaches the original finer boundary. The effect of the weak nonpenetration condition (7) can be seen by the stress peaks at the condyles.

5 Integration in Amira

Amira is a software for the visualization of scientific data. Originally developed at the Zuse Institute Berlin (ZIB), Amira is now distributed commercially.[4] Amira is written in C++ and has a plug-in infrastructure. Plug-ins have access to most internal data structures, and visualization and data processing algorithms. Conversely, users can control the plug-ins with the Amira GUI and the tcl scripting language integrated into Amira.

The development of PSurface started when the author was employed at the visualization department of ZIB. Therefore, the original code consisted entirely of Amira plug-ins. Later, the core data structures and algorithms were moved into a generic C++ library which could be compiled and used independently from Amira. However, some useful functionality still exists in form of Amira code. It is collected in the plug-in `hxpsurface`.

The core of the PSurface Amira integration is a class `HxPSurface` which encapsulates a `PSurface` and makes it available as an Amira data object. Using this wrapper, an Amira viewer module allows to visualize the `PSurface`. Both surfaces S_1 and S_2 are shown as triangulated surfaces, and the mapping is visualized as the edge graph of S_2 on S_1 (Figure 2). Such a representation gives useful insight into the structure and quality of the parametrization, among other things for debugging purposes. Additionally, the display module can visualize the graph nodes with a color code according to the node type, and the images

[4] `www.amira.com`

of the edges of S_1 under Ξ. The regularity of these edges is another indicator of the parametrization quality.

Besides the direct visualization of `PSurface` objects, the AMIRA plug-in offers several other useful features. In particular, parts of the surface simplification algorithm of Section 3.3 are contained in the plug-in. While PSURFACE itself contains the code for removing single vertices, retriangulating the hole and transferring the preimage graphs, the error criteria and ranking algorithms reside in `hxpsurface`. That way, the simplification can be controlled conveniently using the AMIRA GUI. Similarly, the code to smooth a parametrization across an edge is contained in PSURFACE, while the global loop and a GUI are in the plug-in. A general PSURFACE editor, modeled after the AMIRA surface editor, allows to interactively modify the domain surface S_1. Among other things it is possible to delete single vertices, and flip or split edges, while always maintaining a valid parametrization. Also, various quality measures of S_1 such as the triangle aspect ratio can be computed.

Another useful feature is the remesher, which allows to visualize a uniformly refined S_1, where all new vertices have been moved to the positions prescribed by a parametrization Ξ. For a parametrization intended for use as a finite element grid boundary, this allows to visually inspect the quality of the resulting mesh boundary.

As a conclusion, using PSURFACE as an AMIRA plug-in offers a simple way to implement and experiment with computer graphics and visualization algorithms involving parametrized surfaces. For example, the construction of texture and bump maps and their visualization would be straightforward. As another example, we show the linear morphing of one triangulated surface into another. This example originally appeared in [18].

5.1 Morphing

Morphing is the continuous deformation of a surface into another one. Given two homeomorphic surfaces S and S', we are looking for a one-parameter family of surfaces $\mathcal{S} = \{S(\rho) \,|\, \rho \in [0,1]\}$ with $S(0) = S$, $S(1) = S'$ and such that the map $\rho \to S(\rho) \in \mathcal{S}$ is continuous. The user is usually requested to mark sets of feature points on the surfaces that will be required to correspond via the morphing. These points will generally have some geometric significance. For example, when morphing two heads, one might choose the tip of the nose, the chin, the corners of the mouth and the like. The algorithm should then construct a surface family under those constraints.

A basic algorithm looks like this: given two triangulated surfaces S and S' of identical topological type, and an equal number of feature points on each of them. On each surface, the points are then connected by paths that triangulate each surface such that the two triangulations are combinatorially equivalent. This can be done manually or automatically. From these triangulations coarse surfaces B and B' are constructed. The vertices of B and B' are the feature points of S and S', respectively, and two vertices in B (B') are connected by an edge if there is a corresponding path in S (S'). The triangulated surfaces are then parametrized over these base surfaces B, B': We map points on patch boundaries onto base grid edges via an arc-length parametrization and we use the Floater scheme for all other vertices. The quality of the parametrization can be improved further by applying the relaxation scheme described in

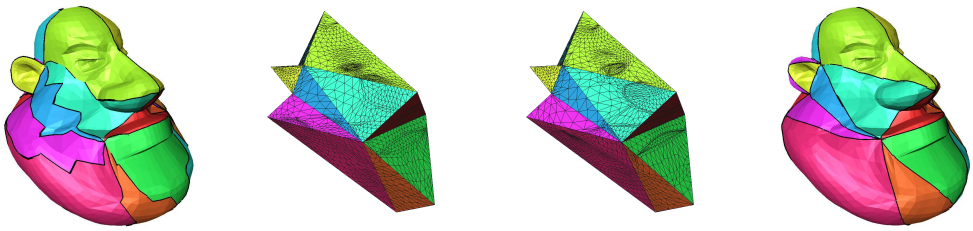

Figure 13: Constructing a parametrization of 'Al', the cartoon figure, that is suitable for morphing. i) the surface with marked points and a rough triangulation; ii) the corresponding base grid; iii) the base grid after the application of edge-relaxation; iv) the original surface showing the relaxed triangulation.

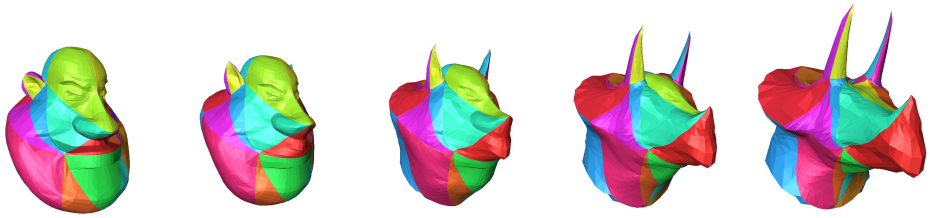

Figure 14: A morphing sequence from the cartoon figure into a dinosaur head showing five equidistant steps.

Chapter 3.3.2. This sequence of steps is shown for the cartoon figure head in Figure 13.

Since the two triangulations were required to be equivalent, we now have the two surfaces S and S' parametrized over two isomorphic base grids B and B' using the respective parametrization functions Ξ and Ξ'. Thus, there exists a bijection θ from B to B', which in turn creates a homeomorphism Φ from S to S' by

$$\Phi(p) = \Xi' \circ \theta \circ \Xi^{-1}(p) \qquad \forall p \in S.$$

Any intermediate model can now be generated by linearly interpolating between corresponding points p and $\Phi(p)$ (Figure 14.)

References

[1] The Visible Human Project. `http://www.nlm.nih.gov/research/visible/visible_human.html`.

[2] P. Bastian, G. Buse, and O. Sander. Infrastructure for the coupling of Dune grids. In *Proc. Enumath '09.* accepted.

[3] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuß, H. Rentz–Reichert, and C. Wieners. UG – a flexible software toolbox for solving partial differential equations. *Comp. Vis. Sci*, 1:27–40, 1997.

[4] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic interface for parallel and adaptive

scientific computing. Part II: Implementation and tests in DUNE. *Computing*, 82(2–3):121–138, 2008.

[5] P. J. Brown and C. T. Faigle. A robust efficient algorithm for point location in triangulations. Technical report, Cambridge University, Feb. 1997.

[6] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. SIGGRAPH, pages 115–122, 1998.

[7] R. Diestel. *Graphentheorie*. Springer Verlag, 3rd edition, 2006.

[8] T. Duchamp, A. Certain, A. DeRose, and W. Stuetzle. Hierarchical computation of PL harmonic embeddings. Technical report, University of Washington, July 1997.

[9] M. S. Floater. Parametrization and smooth approximations of surface triangulations. *Computer Aided Geometric Design*, 14:231–250, 1997.

[10] I. Guskov, K. Vidimče, W. Sweldens, and P. Schröder. Normal meshes. In *SIGGRAPH*, 2000.

[11] I. Guskov, A. Khodakovsky, P. Schröder, and W. Sweldens. Hybrid meshes: multiresolution using regular and irregular refinement. In *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry*, pages 264–272, 2002.

[12] L. Kobbelt, S. Campagna, and H.-P. Seidel. A general framework for mesh decimation. In *Graphics Interface Proceedings*, pages 43–50, 1998.

[13] R. Krause and O. Sander. Fast solving of contact problems on complicated geometries. In R. K. et al., editor, *Domain Decomposition Methods in Science and Engineering*, pages 495–502. Springer Verlag, 2005.

[14] R. Krause and O. Sander. Automatic construction of boundary parametrizations for geometric multigrid solvers. *Comp. Vis. Sci*, 9:11–22, 2006.

[15] H. Lamecker. *Variational and Statistical Shape Modeling for 3D Geometry Reconstruction*. PhD thesis, Freie Universität Berlin, 2008.

[16] A. W. F. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: Multiresolution adaptive parametrization of surfaces. In *SIGGRAPH*, pages 95–104, 1998.

[17] M. Maruya. Generating a texture map from object-surface texture data. *Computer Graphics Forum*, 14(3):397–406, 1995.

[18] O. Sander. Constructing boundary and interface parametrizations for finite element solvers. Diplomarbeit, Freie Universität Berlin, 2001.

[19] O. Sander. *Multidimensional Coupling in a Human Knee Model*. PhD thesis, Freie Universität Berlin, 2008.

[20] D. Stalling, M. Westerhoff, and H.-C. Hege. Amira: A highly interactive system for visual data analysis. In C. Hansen and C. Johnson, editors, *The Visualization Handbook*, chapter 38, pages 749–767. Elsevier, 2005.

[21] *Amira 5 Developer's Guide*. Visage Imaging. URL `www.amira.com/documentation/manuals-and-release-notes.html`.

[22] B. I. Wohlmuth. *Discretization Methods and Iterative Solvers Based on Domain Decomposition*, volume 17 of *LNCSE*. Springer Verlag, 2001.