QEx: Robust Quad Mesh Extraction

Hans-Christian Ebke, David Bommes, Marcel Campen, Leif Kobbelt

QEx: Robust Quad Mesh Extraction

Hans-Christian Ebke* RWTH Aachen University
David Bommes† INRIA Sophia Antipolis
Marcel Campen* RWTH Aachen University
Leif Kobbelt* RWTH Aachen University

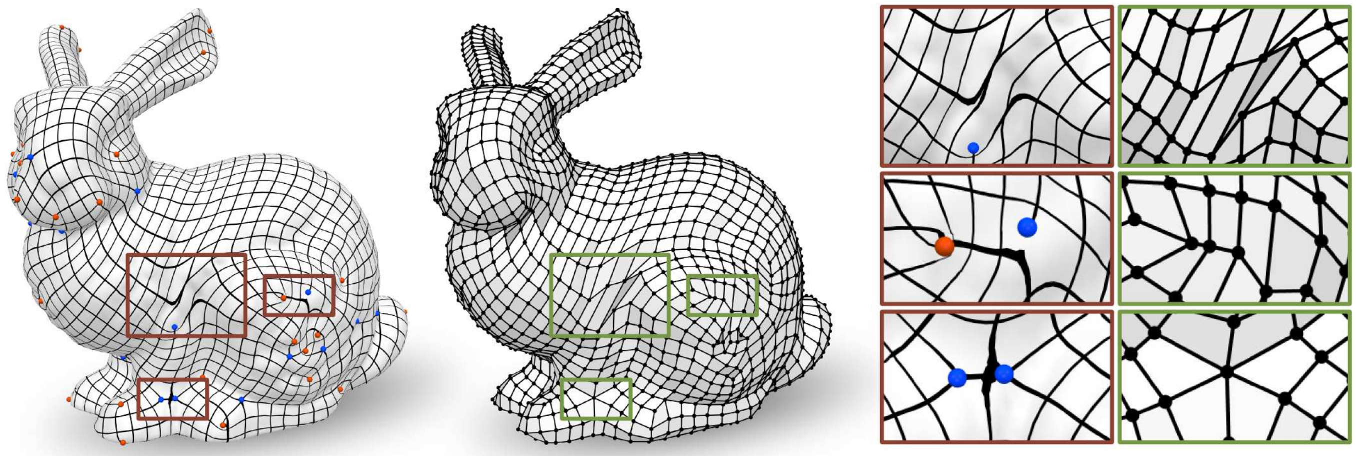


Figure 1: *Parametrization based quad meshing methods cannot guarantee integer-grid maps without fold-overs and degeneracies. The bunny on the left is textured using an integer-grid map generated by the Mixed Integer Quadgrangulation method without stiffening. It contains a number of fold-overs some of which are magnified on the right. The quad mesh generated using QEx, our fold-over tolerant quad extractor is displayed in the center.—Previously, degenerate integer-grid maps such as this one were considered useless and state of the art quad meshing methods spend a considerable amount of run time to fix them.*

Abstract

The most popular and actively researched class of quad remeshing techniques is the family of *parametrization based quad meshing methods*. They all strive to generate an *integer-grid map*, i.e. a parametrization of the input surface into $\mathbb{R}^2$ such that the canonical grid of integer iso-lines forms a quad mesh when mapped back onto the surface in $\mathbb{R}^3$. An essential, albeit broadly neglected aspect of these methods is the *quad extraction* step, i.e. the materialization of an actual quad mesh from the mere "quad texture". Quad (mesh) extraction is often believed to be a trivial matter but quite the opposite is true: numerous special cases, ambiguities induced by numerical inaccuracies and limited solver precision, as well as imperfections in the maps produced by most methods (unless costly countermeasures are taken) pose significant challenges to the quad extractor. We present a method to sanitize a provided parametrization such that it becomes numerically consistent even in a limited precision floating point representation. Based on this we are able to provide a comprehensive and sound description of how to perform quad extraction robustly and without the need for any complex tolerance thresholds or disambiguation rules. On top of that we develop a novel strategy to cope with common local fold-overs in the parametrization. This allows our method, dubbed *QEx*, to generate all-quadrilateral meshes where otherwise holes, non-quad polygons or no output at all would have been produced. We thus enable the practical use of an entire class of maps that was previously considered defective. Since state of the art quad meshing methods spend a significant share of their run time solely to prevent local fold-overs, using our method it is now possible to obtain quad meshes significantly quicker than before. We also provide `libQEx`, an open source C++ reference implementation of our method and thus significantly lower the bar to enter the field of quad meshing.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

Keywords: quad extraction, quad meshing, integer-grid maps

Links: DL PDF WEB CODE

1 Introduction

All algorithms in the very popular and actively researched class of parametrization based quad meshing approaches share one common trait: they strive to generate an *integer-grid map* [Bommes et al. 2013a] which refers to parametrizations which map the input mesh into $\mathbb{R}^2$ in such a way that the canonical grid of integer iso-lines forms a quad mesh when mapped back onto the surface in $\mathbb{R}^3$. While these parametrizations implicitly define a quad mesh, it is necessary to materialize an explicit polygonal quad mesh for virtually all applications. We call this process *quad extraction*.

Judging from the lack of attention the descriptions of all state of the art parametrization based quad meshing methods pay to the quad extraction post-process, one is led to conclude that quad extraction is a trivial matter. However, typical integer-grid parametrizations exhibit a plethora of pitfalls that cause naïve quad extractors to fail. Transition functions pedantically have to be kept track of and nu-

*e-mail:{ebke,campen,kobbelt}@cs.rwth-aachen.de
†e-mail:david.bommes@inria.fr

merous special cases such as integer iso-lines coincident with input mesh edges or integer vertices coinciding with input mesh vertices have to be dealt with.

An even more severe source of complexity are numerical inaccuracies. These have two causes: (1) the parametrizations are typically generated by a numerical optimization process which only satisfies the constraints required by an integer-grid map up to a certain error threshold and (2) the floating point representation of the parametrization introduces errors on top of that. As a consequence, integer iso-lines and especially intersections of them may fall into the "numerical crack" between two adjacent triangles or into both of them at once. Quad extractors have to detect such ambiguous cases and resolve them in a consistent manner. This increases code and run time complexity and is error prone.

In the end, even correctly implemented quad extractors fail in the presence of small degeneracies in the given parametrization: depending on the input mesh and other parameters such as the desired target edge length, some triangles may get mapped to triangles with a negative area in the parametrization domain (when the Jacobian has a negative determinant). Such triangles are commonly referred to as *flipped* or *inverted*. Since a flipped triangle, if adjacent to a regular one, folds over an unflipped area in the parametrization domain, we call such a configuration a *fold-over*. State of the art quad meshing methods spend a significant amount of run time on preventing fold-overs since quad extractors produce holes or non-quad faces when encountering them.

1.1 Related Work

Our quad extraction method is targeted at all parametrization based quad meshing methods. These methods can roughly be divided into those that employ some form of harmonic parametrization and those that generate parametrizations which minimize some alignment energy. Instances of the first class are [Dong et al. 2006], [Tong et al. 2006], [Huang et al. 2008], and [Zhang et al. 2010]. All of these papers show quad meshes while none of them goes into any detail regarding quad extraction, let alone mentions any problems caused by fold-overs or numerical inaccuracies.

Methods that generate parametrizations based on more complex energies, e.g. to achieve curvature and/or feature alignment, include [Ray et al. 2006], [Kälberer et al. 2007], [Bommes et al. 2009], [Myles et al. 2010], and [Myles and Zorin 2013]. None of these methods can guarantee fold-over freeness. In the Mixed-Integer Quadrangulation (MIQ) approach [Bommes et al. 2009] this problem was acknowledged and the stiffening method was introduced which iteratively updates the energy functional in order to heavier penalize local distortions. This may eventually lead to fold-over free parametrizations although there is no guarantee, especially for large target edge lengths. In difficult configurations the parameters have to be tuned in a trial-and-error manner before a non-degenerate integer-grid map is generated. Moreover, stiffening is expensive as was also observed in [Li et al. 2011] and [Li et al. 2012] where, as a remedy, a restricted class of fold-overs is tackled heuristically.

The recently introduced "Reliable MIQ" approach [Bommes et al. 2013a] seems to be the first method to effectively prevent fold-overs in aligned parametrizations by introducing a set of conservatively chosen linear anti-flip constraints at the cost of a slower run time as compared to the original MIQ approach.

Again, all of these works either omit the quad extraction process entirely or only mention it as a side note. No concern for numerical inaccuracies in the parametrization is raised.

Descriptions of quad edge tracing occur in [Alliez et al. 2003], [Marinov and Kobbelt 2004] and especially [Dong et al. 2005]. However, as none of these methods deal with parametrization based quad meshing, none of them have to deal with transition functions (or fold-overs) and thus with none of the difficulties they incur.

There exists a wealth of general techniques for robust geometric computations [Yap 1988; Edelsbrunner and Mücke 1990; Priest 1991; Fortune 1995; Hoffmann 2001]. However, since in our case the input is inconsistent to begin with and numerical problems do not solely occur in subsequent computations, these techniques do not resolve our issues. Nevertheless, we build upon the exact predicates in [Shewchuk 1996] after we numerically sanitized our input.

1.2 Contribution

Our contribution is threefold.

1. We introduce a method to numerically sanitize inaccurate integer-grid maps. Coordinate precision is locally truncated and transition functions are purified so that all subsequent geometric queries can be made consistently using exact predicates. Thus, all decisions can be made based on local information without the need to traverse entire neighborhoods of the mesh, leading to faster and less error prone quad mesh extraction.
2. We comprehensively describe a quad extraction method, pointing out all the pitfalls quad extraction involves and explaining in detail how to deal with transition functions and special cases that arise during iso-line tracing. Our method guarantees the extraction of valid quad meshes for fold-over free integer-grid maps. In addition we introduce a method to deal with common local fold-overs in the parametrization that violate the integer-grid constraints and which not rarely occur in the results of all state of the art parametrization methods unless expensive countermeasures are taken.
3. We provide `libQEx`, an open source C++ reference implementation of our method and thus significantly lower the bar for implementing quad meshing algorithms.

2 Method Overview

Our quad extraction method, dubbed *QEx*, roughly consists of three phases: (1) input preprocessing, (2) geometry extraction, and (3) connectivity extraction. Sections 3, 4 and 5 are devoted to these phases, respectively. Before we go into detail though, we establish the input requirements and introduce some terminology.

2.1 Input Requirements

QEx takes as input a triangle mesh $\mathcal{M} = (V, E, T)$ and a *relaxed* integer-grid map $\mathbf{f}$ mapping each triangle in $\mathcal{M}$ to $\mathbb{R}^2$. Following [Bommes et al. 2013a] an integer-grid map $\mathbf{f}$ is the union of linear maps $\mathbf{f}_i : \mathbb{R}^3 \to \mathbb{R}^2$ that map each triangle $(\mathbf{p}_i, \mathbf{q}_i, \mathbf{r}_i) \in \mathbb{R}^{3\times 3}$ of $\mathcal{M}$ to a triangle $(\mathbf{u}_i, \mathbf{v}_i, \mathbf{w}_i) \in \mathbb{R}^{2\times 3}$ in the plane. Moreover, it satisfies three constraints:

- The transition functions $\mathbf{g}_{ij}$ mapping the chart of triangle i to the chart of the adjacent triangle j have to be grid automorphisms [Kälberer et al. 2007], i.e. be of the form
$$\mathbf{g}_{ij}(\mathbf{a}) = R_{90}^{r_{ij}}\mathbf{a} + \mathbf{t}_{ij} \tag{1}$$
where R_{90} is a rotation by $\pi/2$, $r_{ij} \in \mathbb{Z}$, and $\mathbf{t}_{ij} \in \mathbb{Z}^2$ is an integer translation.
- Singular points have to be mapped to integer coordinates, i.e.
$$\mathbf{f}(\mathbf{s}_i) \in \mathbb{Z}^2 \ \forall \mathbf{s}_i \in S \tag{2}$$
where S is the set of singular points in $\mathcal{M}$.
- The image of each triangle has to have a positive area:
$$\det(\mathbf{v}_i - \mathbf{u}_i \quad \mathbf{w}_i - \mathbf{u}_i) > 0. \tag{3}$$

We define *relaxed* integer-grid maps to be parametrizations that only satisfy Constraints 1 and 2 approximately (due to numerical inaccuracies induced by the solver which generated the map as well as due to the floating point representation) and disregard Constraint 3 entirely. This means that in contrast to pure integer-grid maps, our relaxed integer-grid maps allow flipped triangles as well as triangles degenerated to a line or a point.

2.2 Terminology

We already established that $\mathcal{M} = (V, E, T)$ represents the vertices, edges and triangles of our input mesh and $\mathbf{f}$ somewhat informally represents a collection of maps $\mathbf{f}_i$. The bold letters $\mathbf{p}$, $\mathbf{q}$ and $\mathbf{r}$ refer to vertices of the input or output mesh (or, depending on context, their embedding in $\mathbb{R}^3$) whereas $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ refer to vertices in the parametrized mesh $\mathbf{f}(\mathcal{M})$ (or their embedding in $\mathbb{R}^2$).

Since every triangle t_i is mapped to its own chart using its own map $\mathbf{f}_i$, vertices $\mathbf{p} \in V$ can have multiple images. To express this we introduce the set of *triangle corners* $C \subseteq V \times T$ with

$$C = \{(\mathbf{p}, t) \mid \mathbf{p} \text{ incident to } t\}.$$

To enumerate all corners of a vertex $\mathbf{p} \in V$ we introduce

$$\angle : V \to \mathcal{P}(C), \quad \mathbf{p} \mapsto (\{\mathbf{p}\} \times T) \cap C$$

that maps $\mathbf{p}$ to all of its corners, as well as

$$\angle_t : V \to C, \quad \mathbf{p} \mapsto (\mathbf{p}, t)$$

that maps $\mathbf{p}$ to its corner in t.

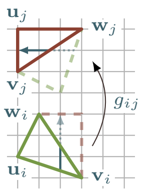
By $\mathbf{f}(c)$ with $c = (\mathbf{p}, t_i)$ we refer to the image $\mathbf{u} = \mathbf{f}_i(\mathbf{p})$ of $\mathbf{p}$ in the chart of triangle t_i. By $\mathbf{f}(t_i)$ we refer to the image of t_i. If all images of $\mathbf{p} \in \mathcal{M}$ coincide we refer to them by $\mathbf{f}(\mathbf{p})$.

We will also have to distinguish between the vertices of the input mesh and those of the output (quad-)mesh. To that end we introduce the names *q-vertices*, *q-edges* and *q-faces* that refer to the entities of the output mesh. In addition to q-edges we also make use of the concept of *q-ports*. A q-port represents an unconnected outgoing q-edge of a q-vertex. Two connected q-ports represent a q-edge.

3 Input Preprocessing

The input preprocessing phase consists of two steps: We (1) extract the transition functions between the charts of every pair of adjacent input triangles and (2) sanitize the parametrization. Before we extract the transition functions we have to make sure that no edges get mapped to a single point in the parameter domain. We do this by initially collapsing all edges that are degenerated in the parameter domain. As vertices get merged in this process, we store the original vertex positions in $\mathbb{R}^3$ in the triangle corners so as to retain the original surface geometry information.

3.1 Extracting the Transition Functions



In general, the parametrized input mesh consists of multiple charts. When tracing an integer isoline across multiple charts we require a transition function that translates coordinates between them. In some settings the rotational part of these transition functions is available (e.g. as period jumps in field guided parametrizations) in which case we can take it as input. However, in order to ensure wide applicability of our method we show how to extract both, the rotational and the translational component solely from the input parametrization.

Assume $(\mathbf{u}_i, \mathbf{v}_i, \mathbf{w}_i), (\mathbf{u}_j, \mathbf{v}_j, \mathbf{w}_j) \in \mathbb{R}^{2\times3}$ are the images of two adjacent triangles in $\mathcal{M}$ with $\mathbf{g}_{ij}(\mathbf{v}_i) = \mathbf{w}_j$ and $\mathbf{g}_{ij}(\mathbf{w}_i) = \mathbf{v}_j$ (as illustrated in the inset above). To compute the unknown transition

```
1:  Input: triangle mesh $(V, E, T)$, map $\mathbf{f}$
2:  for each vertex $\mathbf{p} \in V$ do
3:      let $\{c_0, \ldots, c_{n-1}\} = \angle(\mathbf{p})$ with $c_i = (\mathbf{p}, t_i)$
4:      let $(u_i, v_i)^t = \mathbf{u}_i = \mathbf{f}(c_i)$
5:      let $\mathbf{g}_i(\mathbf{u}) = R_{90}^{r_i}\mathbf{u} + \mathbf{t}_i$ be the transition from $t_i$ to $t_{i+1}$
6:      maxCoord $\leftarrow \max\{\|\mathbf{u}_0\|_\infty, \ldots, \|\mathbf{u}_{n-1}\|_\infty\}$
7:      $\delta \leftarrow 2^{\lceil \log_2(\text{maxCoord}) \rceil}$
8:      $\mathbf{s} \leftarrow (\text{sgn}(u_0), \text{sgn}(v_0))^t$
9:      if $\mathbf{p}$ is regular then
10:         $\mathbf{u}_0 \leftarrow (\mathbf{u}_0 + \delta\mathbf{s}) - \delta\mathbf{s}$                ▷ truncate precision of $\mathbf{u}_0$
11:     else
12:         FIXSINGULARPOINT($\mathbf{u}_0, \mathbf{g}_{n-1} \circ \ldots \circ \mathbf{g}_0$)
13:     for $i = 1, \ldots, n-1$ do                              ▷ propagate $\mathbf{u}_0$ along the fan
14:         $\mathbf{u}_i \leftarrow \mathbf{g}_{i-1}(\mathbf{u}_{i-1})$
```

Algorithm 1: *Parametrization Sanitization*

function $\mathbf{g}_{ij}(\mathbf{x}) = R_{90}^{r_{ij}}\mathbf{x} + \mathbf{t}_{ij}$ we take advantage of the complex number representation of the involved vertices. Let $c : \mathbb{R}^2 \to \mathbb{C}$, $(x_1, x_2)^t \mapsto x_1 + ix_2$. Then due to the fact that by construction $\|\mathbf{w}_i - \mathbf{v}_i\| = \|\mathbf{v}_j - \mathbf{w}_j\| \neq 0$,

$$r_{ij} = \Im\left(\ln\left(\frac{c(\mathbf{v}_j) - c(\mathbf{w}_j)}{c(\mathbf{w}_i) - c(\mathbf{v}_i)}\right)\right) / \frac{\pi}{2}$$

is the rotational part in multiples of $\pi/2$ with $r_{ij} \in [0, 4)$ and

$$\mathbf{t}_{ij} = \mathbf{w}_j - i^{r_{ij}}\mathbf{v}_i$$

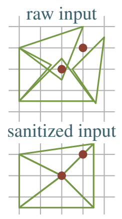
is the translational part of the transition function.

Note that the input parametrization is typically the result of an optimization process performed by means of a numerical solver that only satisfies the constraints on the transition functions up to a small error. In addition, the parametrization is represented with limited precision floating point numbers. As a consequence, r_{ij} and $\mathbf{t}_{ij}$ generally are not elements of $\mathbb{Z}$ and $\mathbb{Z}^2$, respectively. In order to obtain transition functions that exactly adhere to Constraint 1 we round first r_{ij} and then $\mathbf{t}_{ij}$ to the nearest integers. Any subsequent references to $\mathbf{g}_{ij}$ refer to this rounded version. Note that from $\mathbf{g}_{ij}$ we can easily compute its inverse $\mathbf{g}_{ji}$:

$$\mathbf{g}_{ji}(\mathbf{a}) = \mathbf{g}_{ij}^{-1}(\mathbf{a}) = R_{90}^{4-r_{ij}}(\mathbf{a} - \mathbf{t}_{ij}).$$

In practice, non-identity transition functions are usually concentrated on a small subset of the edges of $\mathcal{M}$ ("cut graph") which we refer to as *cut edges*. For the sake of clarity we use this scenario in all illustrations with only one cut edge per singularity.

3.2 Sanitizing the Parametrization



Another problem incurred by the numerical inaccuracies mentioned above is that the parametrization is discontinuous on a very small scale: there is no guarantee that $\mathbf{g}_{ij}(\mathbf{u}_i) = \mathbf{u}_j$ holds for any $\{\mathbf{u}_i, \mathbf{u}_j\} \subseteq \mathbf{f}(\angle(\mathbf{p}))$ and in spite of $\mathbf{g}_{ji} \circ \mathbf{g}_{ij}$ being the identity it is not even guaranteed numerically that $\mathbf{g}_{ji}(\mathbf{g}_{ij}(\mathbf{v}_i)) = \mathbf{v}_i$. This turns into an actual problem when integer grid vertices fall through the "numeric crack" between two adjacent triangles or into both of them as illustrated on the right. The chances of this actually happening are high since, for instance, at aligned sharp features, integer grid vertices necessarily fall onto input triangle mesh edges.

To overcome this problem, we equalize the parametrization in precision such that the floating point arithmetics used in the transition functions become exact. It does not suffice to ensure that $\mathbf{g}_{ji}(\mathbf{g}_{ij}(\mathbf{v}_i)) = \mathbf{v}_i$ holds for every edge: if we accumulate the transition functions $\mathbf{g}_{n-1} \circ \ldots \circ \mathbf{g}_0 = \bar{\mathbf{g}}$ around a vertex $\mathbf{p}$ with corners $c_i = (\mathbf{p}, t_i)$, $\mathbf{u}_i = \mathbf{f}(c_i)$ we have to make sure that $\bar{\mathbf{g}}(\mathbf{u}_0) = \mathbf{u}_0$. Since the t_i can be in different charts, the floating point representations of their coordinates may exhibit different exponents. This

```
function FixSingularPoint($\mathbf{u}, \bar{\mathbf{g}}$)
    let $\bar{\mathbf{g}}(\mathbf{a}) = R_{90}^r \mathbf{a} + \mathbf{t}, \mathbf{t} = (t_1, t_2)^{\mathrm{t}}$
    $\mathbf{u} \leftarrow \begin{cases} \mathrm{round}(\mathbf{u}) & \text{if } r = 0 \mod 4 \\ \frac{1}{2}\begin{pmatrix} t_1 - t_2 \\ t_2 + t_1 \end{pmatrix} & \text{if } r = 1 \mod 4 \\ \frac{1}{2}\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} & \text{if } r = 2 \mod 4 \\ \frac{1}{2}\begin{pmatrix} t_1 + t_2 \\ t_2 - t_1 \end{pmatrix} & \text{if } r = 3 \mod 4 \end{cases}$
```

Algorithm 2: *Snapping Singularities to the Fixed Point of* $\bar{\mathbf{g}}$

implies that transforming $\mathbf{u}_i$ into $\mathbf{u}_j = \mathbf{g}_{ij}(\mathbf{u}_i)$ we may lose some of the least significant bits in the floating point representation of $\mathbf{u}_i$ and thus are not able to get back to $\mathbf{u}_0$ if we close the cycle.

We solve this problem in Algorithm 1 by determining the largest exponent the coordinates of any image $\mathbf{u}_i \in \mathbf{f}(\mathbf{p})$ have and truncate those least significant bits in the mantissa of $\mathbf{u}_0$ that cannot be represented in the charts of *every* triangle incident to $\mathbf{p}$. This operation is performed in line 10. It presents an algebraic no-op but implemented using limited precision floating point arithmetics has the desired effect of truncating the number's precision. Afterwards, we propagate the coordinates of $\mathbf{u}_0$ along the 1-ring triangle fan.

If $\mathbf{p}$ is regular (not a singularity, line 9) the accumulated transition function $\mathbf{g}_{n-1} \circ \ldots \circ \mathbf{g}_0$ around the vertex is the identity and hence its rotational component is zero (i.e. it has no angle defect). However, the inverse is not necessarily true as singularities with a valence of a multiple of four have $r = 0$ as well. To detect these cases we can either take valence information as input to our method or we compute the valence of the singularity as described in Section 5.6. Note that dealing with unsanitized coordinates at this point, we may get a relatively imprecise valence but as we only have to distinguish between valences 4 and ≥ 8 this is not a problem.

Once we know which vertices represent singularities, in order to guarantee that the sanitized parametrization satisfies Constraint 2 we make sure that their parametrization has integer coordinates in line 12, prior to propagation along the triangle fan. Note that around a singularity, $\mathbf{g}_{n-1} \circ \ldots \circ \mathbf{g}_0$ is not the identity. To ensure that $\mathbf{g}_{n-1}(\mathbf{u}_{n-1}) = \mathbf{u}_0$ still holds, it does not generally suffice to simply round the coordinates to the nearest integer. Instead we have to move $\mathbf{u}_0$ into the fixed point of the accumulated transition function. This is implemented in Algorithm 2. In most applications the input parametrization usually satisfies Constraint 2 up to an error of far less than $1/2$ which means that in these cases simple rounding of the coordinates would indeed suffice. Using the method in Algorithm 2 we are on the safe side for any input. Note that truncation (line 10) is not necessary in this case as the fixed point, being integral, can be represented precisely (unless its parametrization coordinates exceed the range of the mantissa of the floating point representation which is $\pm 2^{52}$ for standard IEEE 754 double precision floats—which is far from any practical relevance).

After this procedure, the parametrization and the transition functions are numerically consistent and compatible. If feature alignment is desired we can take the list of feature edges as additional input and snap these edges to their closest integer iso-line.

As a final step we again collapse all edges with zero length after the precision truncation. Note that this only serves to avoid special case handling later on and does not affect the parametrization.

We can now use exact predicates for any geometric query. In fact we only need a single predicate, $\text{Orient2D}(\mathbf{a}, \mathbf{b}, \mathbf{c})$ as introduced in [Shewchuk 1996] which returns a scalar with the same sign as $\det(\mathbf{b} - \mathbf{a} \quad \mathbf{c} - \mathbf{a})$. We also use variants of Orient2D, namely $\mathrm{sgn}(\mathrm{area}(t)) = \text{Orient2D}(\mathbf{p}_t, \mathbf{q}_t, \mathbf{r}_t)/|\text{Orient2D}(\mathbf{p}_t, \mathbf{q}_t, \mathbf{r}_t)|$ as well as $\text{isCW}(\mathbf{p}, \mathbf{q}, \mathbf{r})$, $\text{isCollinear}(\mathbf{p}, \mathbf{q}, \mathbf{r})$, and

```
1:  Input: triangle mesh $(V, E, T)$, sanitized map $\mathbf{f}$
2:  for each vertex $\mathbf{p} \in V$ do
3:      pick arbitrary parameter image $\mathbf{u} \in \mathbf{f}(\angle(\mathbf{p}))$
4:      if $\mathbf{u} \in \mathbb{Z}^2$ then
5:          generate vertex-q-vertex with embedding $\mathbf{p}$
6:  for each edge $(\mathbf{p}, \mathbf{q}) \in E$ do
7:      pick arbitrary parameter image $(\mathbf{u}, \mathbf{v})$ of $(\mathbf{p}, \mathbf{q})$
8:      for each $0 < \alpha < 1,\ \alpha\mathbf{u} + (1 - \alpha)\mathbf{v} \in \mathbb{Z}^2$ do
9:          generate edge-q-vertex with embedding $\alpha\mathbf{p} + (1 - \alpha)\mathbf{q}$
10: for each triangle $(\mathbf{p}, \mathbf{q}, \mathbf{r}) \in T$ do
11:     let $(\mathbf{u}, \mathbf{v}, \mathbf{w}) = \mathbf{f}(\mathbf{p}, \mathbf{q}, \mathbf{r})$
12:     for each $0 < \alpha, \beta < 1,\ \alpha\mathbf{u} + \beta\mathbf{v} + (1 - \alpha - \beta)\mathbf{w} \in \mathbb{Z}^2$ do
13:         generate face-q-vertex w/ emb. $\alpha\mathbf{p} + \beta\mathbf{q} + (1 - \alpha - \beta)\mathbf{r}$
```

Algorithm 3: *Q-Vertex Generation*

$\text{isCCW}(\mathbf{p}, \mathbf{q}, \mathbf{r})$ which are true if $\text{Orient2D}(\mathbf{p}, \mathbf{q}, \mathbf{r})$ is positive, zero or negative, respectively.

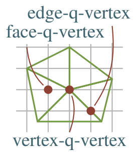
Without the guarantees the truncation provides we would have to account for small errors. For instance, to decide which part of a mesh some point lies on (see Section 4), we would have to intersect an ϵ-ball around that point with an entire neighborhood of the mesh and make consistent decisions in the presence of ambiguities. This would complicate algorithms and impact their run time complexity.

4 Geometry Extraction

We now start by extracting the positions of our output q-vertices. We also examine the local neighborhood of the q-vertices in order to decide how many outgoing q-edges they will have in the output quad mesh.

4.1 Generating Q-Vertices

Due to the sanitization we performed before, using exact predicates we can locally decide which vertex, edge, or face of the parametrized mesh any given grid point $\mathbf{p} \in \mathbb{Z}^2$ intersects, in order to extract all q-vertices. We distinguish between three different types of q-vertices: (1) vertex-q-vertices, (2) edge-q-vertices and (3) face-q-vertices. Of the first type are all q-vertices that coincide with a vertex of the parametrized mesh. Q-vertices of the second type are those that are not of the first type but still intersect with an edge. Face-q-vertices are all remaining q-vertices, i.e. those that merely intersect with a parametrized face. Note that there is no 1:1 correspondence between integer grid points and q-vertices since chart-based parametrizations may overlap so that multiple q-vertices correspond to the same integer location in $\mathbb{R}^2$.



We extract the three types of q-vertices using the three simple loops over the entities of our input mesh $\mathcal{M} = (V, E, T)$ in Algorithm 3. Note that at this point, in the presence of fold-overs, some q-vertices may get extracted that vanish in the vertex merging step (Section 5.4) later on.

4.2 Fold-Overs

A fold-over is a configuration where some triangles $t \in T$ get mapped to triangles $\mathbf{f}(t)$ with a negative area in the parametrization domain. Integer-grid parametrizations are fold-over free due to Constraint 3. However, since typical quad meshing approaches generate the minimizer of an energy functional which, for performance reasons, usually is not constrained by this (non-linear) inequality, we are often confronted with fold-overs in practice. Since some of the following steps of our algorithm have to consider fold-

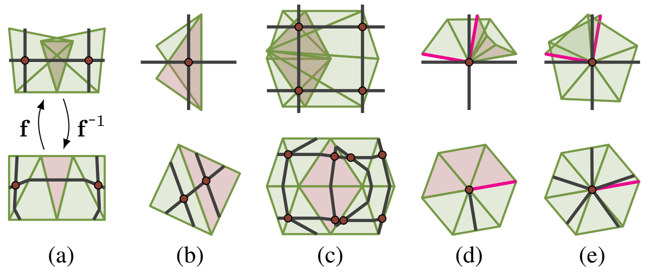


Figure 2: *Meshes in parametrization domain $\mathbb{R}^2$ (top row) and surface domain (bottom row). (a)–(d): examples of parametrizations with fold-overs. (e): the same mesh as in (d) parametrized using identical transition functions but without fold-overs.*

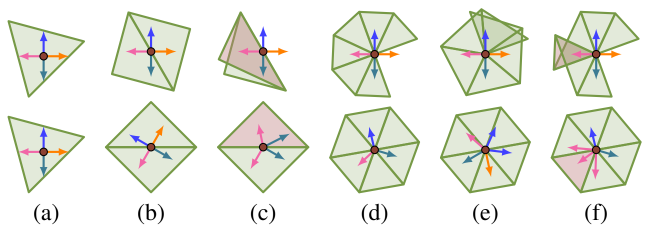


Figure 3: *Top: different parametrizations around an integer grid point. Outgoing integer iso-lines are illustrated as colored arrows. Flipped triangles are highlighted. Bottom: back-projection of the iso-lines onto the surface domain. Here, each arrow corresponds to an extracted q-port.*

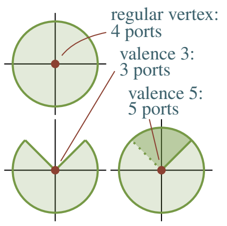
overs and handle them appropriately, we introduce the effects they can have in this section.

The simplest type of a fold-over is one which is locally restricted to the interior of an integer grid cell. Since we never traverse such triangles during integer iso-line tracing our method is oblivious to this class of imperfections. Figure 2 illustrates some more intricate fold-over configurations. Figure 2 (a) shows a fold-over which extends over part of an integer iso-line, not affecting a grid point. This type of fold-over is handled during q-port tracing (Section 5.1).

Figure 2 (b) illustrates a fold-over which contains a grid point. Such fold-overs lead to the same integer-grid point intersecting the parametrization multiple times, ultimately leading to non-quad faces getting extracted. Multiple duplicated grid points can even lead to ghost faces as Figure 2 (c) illustrates. The q-port enumeration (Section 4.3), face extraction (Section 5.2), and vertex merging (Section 5.4) all play a role in properly handling this type of fold-over.

Figure 2 (d) illustrates the most intricate type of fold-over which, in a way, is a degeneration of the nonproblematic parametrization illustrated in Figure 2 (e). Here a singular vertex of valence 5 loses an entire 360° of its signed inner angle sum, resulting in a valence 1 vertex. This configuration leads to an 8-gon (with one edge occurring twice in a row, see Figure 5). Such fold-overs are handled during q-edge recovery (Section 5.5).

4.3 Enumerating Q-Ports



Once all q-vertices are generated, for each q-vertex we generate all of its q-ports which correspond to the intersections of the integer iso-lines with an infinitesimal neighborhood of the parametrized mesh around the q-vertex. For a regular non-boundary q-vertex there are always four q-ports (cf. Figure 3 (a), (b)). As irregular vertices are parametrized with an angle defect, there are more or less than four intersections of parameter lines with the mesh (cf. Figure 3 (d), (e)).

Care has to be taken to store the q-ports consistently in clockwise order (w.r.t. the surface in $\mathbb{R}^3$) so that when extracting the q-faces we are able to turn left at a q-vertex simply by moving to the next q-port in the q-vertex' list. Note that clockwise order on the surface does not imply any particular order in the parametrization. This is because non-identity transition functions between triangles may introduce jumps in the q-ports' directions and triangles that are flipped in the parametrization invert the order of their q-ports when mapped back to the surface (cf. Figure 3 (c), (f)).

```
1:  Input: triangle mesh, sanitized map $\mathbf{f}$, q-vertices
2:  for each vertex-q-vertex $\mathbf{q}$ do
3:      let $\mathbf{p}$ be the input mesh vertex $\mathbf{q}$ coincides with
4:      ports ← [ ]
5:      for each corner $c = (\mathbf{p}, t) \in \angle(\mathbf{p})$ in CCW order do
6:          let $(\mathbf{u}, \mathbf{v}, \mathbf{w}) \in \mathbb{R}^{2\times 3}$ be the parameter image of $t$
7:          let $\mathbf{u}^* = \mathbf{u}$ be the parameter image of $\mathbf{q}$ in $t$
8:          let $\mathbf{d}(r) := R_{90}^{r}\binom{1}{0}$
9:          orientation ← ORIENT2D($\mathbf{u}, \mathbf{v}, \mathbf{w}$)
10:         if orientation $= 0$ then
11:             continue
12:         else if orientation $> 0$ then
13:             $r \leftarrow 0$
14:             while POINTSINTO($\mathbf{d}(r), \mathbf{u}, \mathbf{v}, \mathbf{w}$) do
15:                 $r \leftarrow r + 1$    ▷ Find iso-line outside of $t$ to ensure correct order.
16:             for $i = 1, \ldots, 3$ do    ▷ Add ports in CW order.
17:                 if POINTSINTO($\mathbf{d}(r - i), \mathbf{u}, \mathbf{v}, \mathbf{w}$) or
18:                         ISCOLLINEAR($\mathbf{v} - \mathbf{u}, \mathbf{d}(r - i)$) then
19:                     ports ← ports ∪ [($\mathbf{q}, \mathbf{u}^*, t, \mathbf{d}(r - i)$)]
20:         else    ▷ Inverse order for flipped faces.
21:             $r \leftarrow 0$
22:             while POINTSINTO($\mathbf{d}(r), \mathbf{u}, \mathbf{w}, \mathbf{v}$) do
23:                 $r \leftarrow r + 1$
24:             for $i = 1, \ldots, 3$ do
25:                 if POINTSINTO($\mathbf{d}(r + i), \mathbf{u}, \mathbf{w}, \mathbf{v}$) or
26:                         ISCOLLINEAR($\mathbf{v} - \mathbf{u}, \mathbf{d}(r + i)$) then
27:                     ports ← ports ∪ [($\mathbf{q}, \mathbf{u}^*, t, \mathbf{d}(r + i)$)]
```

Algorithm 4: *Q-Port Enumeration. Per q-port we store its q-vertex, the direction it points into in the parameter domain, the triangle it points into and its q-vertx' parameter image in that triangle's chart.*

The q-port extraction for vertex-q-vertices is outlined in Algorithm 4. On meshes with boundaries, the if-conditions in lines 17 and 25 have to be adjusted so that iso-lines collinear with $\mathbf{w} - \mathbf{u}$ are accepted if t has no counterclockwise neighbor. The predicate POINTSINTO($\mathbf{d}, \mathbf{u}, \mathbf{v}, \mathbf{w}$) checks whether $\mathbf{d}$ points from $\mathbf{u}$ into triangle $(\mathbf{u}, \mathbf{v}, \mathbf{w})$ and is equivalent to ISCCW($\mathbf{u}, \mathbf{v}, \mathbf{u} + \mathbf{d}$) ∧ ISCCW($\mathbf{u}, \mathbf{u} + \mathbf{d}, \mathbf{w}$).

The q-port extraction methods for edge-q-vertices and face-q-vertices are simplifications of this method: For edge-q-vertices the loop in line 5 has to iterate only over the two incident faces and for face-q-vertices simply one q-port for each of the Cartesian directions is generated and the list is reversed if the surrounding face is flipped. In both cases, $\mathbf{u}^\star$ in line 7 is computed as a convex combination of $\mathbf{u}, \mathbf{v}, \mathbf{w}$.

5 Connectivity Extraction

5.1 Tracing Q-Ports

Now that we enumerated all q-ports we trace the integer iso-lines from each q-port looking for the opposite q-port. We connect each q-port with its counterpart. Every opposing pair of connections corresponds to a q-edge.

```
1:  Input: triangle mesh, sanitized map $\mathbf{f}$, q-vertices, q-ports
2:  for each port $p = (\mathbf{q}, \mathbf{u}, t, \mathbf{d})$ do
3:                  ▷ $\mathbf{q}$: start q-vertex, $t$: start triangle, $\mathbf{d}\in\mathbb{R}^2$: tracing direction
4:      if isConnected($p$) then
5:          continue
6:      $\mathbf{a} \leftarrow \mathbf{u}$                          ▷ source UVs
7:      $\mathbf{b} \leftarrow \mathbf{a} + \mathbf{d}$                      ▷ target UVs
8:      $\mathbf{g} \leftarrow$ Id                   ▷ accumulated transition function
9:      $t' \leftarrow t$
10:     $e' \leftarrow$ NIL
11:     $s \leftarrow \mathrm{sgn}(\mathrm{area}(\mathbf{f}(t)))$
12:     while not isInside($\mathbf{b}, \mathbf{f}(t')$) do
13:         $e \leftarrow$ PICKNEXTEDGE($e', t', \mathbf{a}, \mathbf{b}$)
14:         let $\mathbf{g}_e$ be the transition function from $t'$ over $e$
15:         $t' \leftarrow$ opposite triangle of $e$
16:         if $t'$ = NIL then                    ▷ Trace into boundary?
17:             abort trace, leave $p$ dangling
18:         $e' \leftarrow e$
19:         $s' \leftarrow \mathrm{sgn}(\mathrm{area}(\mathbf{f}(t')))$
20:         $\mathbf{g} \leftarrow \mathbf{g}_e \circ \mathbf{g}$
21:         if $s' \neq 0$ and $s' \neq s$ then          ▷ Orientation change?
22:             swap($\mathbf{a}, \mathbf{b}$)
23:             $\mathbf{d} \leftarrow -\mathbf{d}$
24:             $s \leftarrow s'$
25:     let $p' = (\mathbf{b}, t', -\mathbf{d})$
26:     store connection $(p, p', \mathbf{g})$                    ▷ create q-edge
27:     store connection $(p', p, \mathbf{g}^{-1})$
```

Algorithm 5: *Q-Port Tracing*

Algorithm 5 implements the tracing. Using exact predicates on the sanitized parametrization, we do not require any neighborhood searches in order to accommodate for a numeric error.

PICKNEXTEDGE (line 13) returns the edge of t' that is unequal to e' and intersects the line segment $(\mathbf{a}, \mathbf{b}]$. If both remaining edges of t' intersect that line segment, it returns the one which is incident to fewer vertices which fall onto the line segment. (In case of a tie it is irrelevant which one is returned.) This way we effectively deal with integer iso-lines intersecting input mesh edges and triangles degenerated to a line. The required intersection tests are performed using the exact predicate ORIENT2D.

Care has to be taken when tracing into boundaries and across orientation flips. If we trace the iso-line across an edge with only one incident face we reached a boundary and abort the trace, leaving a dangling q-port (line 16). Dangling ports are ignored in the face tracing phase. If the orientation of the parametrized mesh flips across the trace path, i.e. if we cross from t_1 to t_2 and $\mathrm{sgn}(\mathrm{area}(\mathbf{f}(t_1))) \neq \mathrm{sgn}(\mathrm{area}(\mathbf{f}(t_2)))$, we have to invert our tracing direction (line 21).

To enable compact notation, for a connection $c = (p, p', \mathbf{g})$ we define FROMPORT$(c) := p$, TOPORT$(c) := p'$ and for a q-port $p = (\mathbf{q}, \mathbf{u}, t, \mathbf{d})$ we define VERTEX$(p) := \mathbf{q}$. We also define the shorthands FROMVERTEX$(c) :=$ VERTEX(FROMPORT(c)) as well as TOVERTEX$(c) :=$ VERTEX(TOPORT(c)).

5.2 Extracting Q-Faces

Now that we have connectivity information between q-vertices we can extract the actual q-faces. We do this by cycling around each q-face along its edges in counterclockwise order. Consequently, we traverse each q-edge twice, once in each direction, to capture both incident q-faces. This is why in the previous step we created two connections (or half-edges) between each adjacent pair of q-ports.

The basic idea – which is implemented in Algorithm 6 – is simple: Starting at any unvisited connection, we follow the connection to the next q-vertex, turn left and repeat until we get back to the starting q-vertex. Since for every q-vertex we maintain a clockwise

```
1:  Input: q-vertices, q-ports, q-edges
2:  for each connection $c_{\text{start}}$ do
3:      if $c_{\text{start}}$ is visited then
4:          continue
5:      $\mathbf{q}_{\text{start}} \leftarrow$ FROMVERTEX($c_{\text{start}}$)
6:      $V \leftarrow [\,], U \leftarrow [\,]$             ▷ List of vertices and local UVs
7:      $\bar{\mathbf{g}} \leftarrow$ Id                 ▷ Accumulated transition function
8:      $c \leftarrow c_{\text{start}}$
9:      repeat
10:         let $c = (p, q, \mathbf{g})$
11:         let FROMPORT$(c) = (\mathbf{q}, \mathbf{u}, t, \mathbf{d})$
12:         mark $c$ as visited
13:         $V \leftarrow V + [\mathbf{q}]$
14:         $U \leftarrow U + [\bar{\mathbf{g}}^{-1}(\mathbf{u})]$
15:         $\mathbf{q}_{\text{next}} \leftarrow$ TOVERTEX$(c)$
16:         $p_{\text{next}} \leftarrow$ NEXTPORT($\mathbf{q}_{\text{next}}$, TOPORT$(c)$)
17:         $\bar{\mathbf{g}} \leftarrow$ FANTRANSITIONCW(TOPORT$(c), p_{\text{next}}$) $\circ\, \mathbf{g} \circ \bar{\mathbf{g}}$
18:         if $p_{\text{next}}$ is dangling port then
19:             abort, continue with next iteration of outer loop
20:         else
21:             $c \leftarrow$ OUTGOINGCONNECTIONFROM($p_{\text{next}}$)
22:     until FROMVERTEX$(c) = \mathbf{q}_{\text{start}}$
23:     store face with vertices $V$ and local UVs $U$
```

Algorithm 6: *Face Extraction*

ordered list of q-ports we turn left simply by picking the next q-port in the list after the q-port we used to get to the q-vertex. This is what NEXTPORT (line 16) does.

Algorithm 6 additionally computes parameter coordinates for every q-vertex, all expressed in the chart of the initial q-port's triangle. We call these coordinates *local UVs* as they represent a parametrization of each q-face in its own local coordinate system. To compute the local UVs, the transition functions along the traced face boundary are accumulated (line 17). FANTRANSITIONCW(p_1, p_2) with p_1, p_2 belonging to the same q-vertex, accumulates the transition functions on the triangle fan when moving from p_1's triangle to p_2's triangle in clockwise order.

The generated q-faces form the mesh as induced by the input parametrization. In the presence of fold-overs, it may potentially contain non-quad faces. In the following sections we will see that these can be turned into quad faces without affecting the surrounding quads by examining the local UVs.

5.3 Local Face-UVs

To understand what types of q-faces Algorithm 6 constructs, we examine the computed local UVs. We can always translate the reference coordinate system of the q-face so that the first q-vertex has local UVs $\mathbf{u}_0 = (0,0)^t$ and rotate it so that the first connection's outgoing q-port points into direction $\mathbf{d}_0 = (1,0)^t$. In line 15 the face extraction follows the current connection to the next vertex.

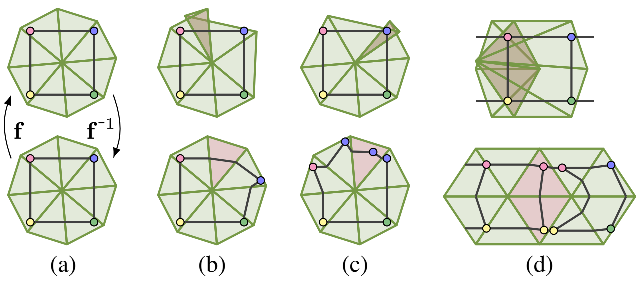


Figure 4: *Examples of different face UVs for different types of fold-overs. The upper row shows parametrizations of the meshes in the lower row. The integer iso-grid of the upper row is projected back into the meshes in the lower row. Fold-overs are highlighted in red.*

```
1: Input: extracted mesh $(V, E, F)$
2: initialize merge-graph $G = (V_G, E_G),\ V_G \leftarrow V,\ E_G \leftarrow \varnothing$
3: for each q-face $f$ do
4:     for each pair of q-vertices $\mathbf{p}_1, \mathbf{p}_2 \in f$ do
5:         if $\mathbf{p}_1$ and $\mathbf{p}_2$ have identical local UVs in $f$ then
6:             $E_G \leftarrow E_G \cup \{(\mathbf{p}_1, \mathbf{p}_2)\}$
7: for each connected component $C = (V_C, E_C) \subseteq G$ do
8:     create Vertex $\mathbf{p}^*$, use c.o.g. of $C$ as embedding
9:     $V \leftarrow V \cup \{\mathbf{p}^*\}$
10:    for each $(\mathbf{p}, \mathbf{p}_C) \in E, \mathbf{p} \in V \setminus V_C, \mathbf{p}_C \in V_C$ do
11:        if $(\mathbf{p}, \mathbf{p}^*) \notin E$ then
12:            $E \leftarrow E \cup \{(\mathbf{p}, \mathbf{p}^*)\}$
13:    $E \leftarrow E \setminus (V_C \times V)$        ▷ delete merged edges
14:    $V \leftarrow V \setminus V_C$                   ▷ delete merged vertices
15: SIMPLIFYFACES(V, E, F)
```

Algorithm 7: *Vertex Merging*

Each connection was generated by tracing the integer iso-line at $\mathbf{u}_0$ in direction $\mathbf{d}_0$ across the input mesh, following all transition functions along the way until it reaches the next integer coordinates. Every time there is an orientation change along the way (i.e. every time we cross from a regular triangle into an inverted one and vice-versa) the tracing direction is inverted. As a consequence, when we arrive at the next q-vertex we either do so on a regular triangle after an even number of orientation changes walking into direction $\mathbf{d}_0$ or on an inverted triangle after an odd number of orientation changes walking in direction $-\mathbf{d}_0$. In the former case the local UVs of the next q-vertex are $\mathbf{u}_1 = \mathbf{u}_0 + \mathbf{d}_0$, in the latter case we ran back to the same local UVs $\mathbf{u}_1 = \mathbf{u}_0$. Figures 4 (a), (b) illustrate tracing with an even number of orientation changes along the connections. In Figure 4 (c) we can see two connections with an odd number of orientation changes that lead to a face where the local UVs of the blue vertex occur three times.

Key to understanding how the tracing continues is the function NEXTPORT (line 16) that selects the next q-port and hence the next connection that the extraction will continue on. Let p_{in} be the port pointing in direction $-\mathbf{d}_{\text{in}}$ through which we entered the current q-vertex and $p_{\text{out}} = \text{NEXTPORT}(p_{\text{in}})$ the port pointing in direction $\mathbf{d}_{\text{out}}$ through which we will leave the current q-vertex. NEXTPORT simply picks the next q-port in the q-vertex's q-port list. Due to the way this list was constructed in Algorithm 4 we know that $\mathbf{d}_{\text{out}}$, expressed in the coordinate system of p_{in}, is equal to either (1) $R_{90}^{-1}(-\mathbf{d}_{\text{in}})$, (2) $R_{90}(-\mathbf{d}_{\text{in}})$ or (3) $-\mathbf{d}_{\text{in}}$. All three cases occur in Figure 3. Case (1) is the regular case that occurs if both, the triangle of p_{in} and the triangle of p_{out} are not inverted. Case (2) occurs if both triangles are inverted. Case (3) occurs if one triangle is inverted and the other one is not. On face-q-vertices, only Cases (1) and (2) can ever occur. These findings hold for q-vertices on a boundary as well as we keep dangling q-ports that point across the boundary (and discard the face intersecting the boundary). There is one intricate special case in which the prerequisites for these findings are not given. We go into detail on that in Section 5.5.

Consequently, no matter for how many iterations the inner loop in Algorithm 6 runs, we always trace the sequence of local UVs $(0,0)^t$, $(1,0)^t$, $(1,1)^t$, $(0,1)^t$: as long as we trace in counterclockwise order in this unit quad we automatically take left turns at the q-vertices. The tracing direction gets inverted whenever we enter fold-overs either along a connection or on a q-vertex. As long as we trace in inverted regions, i.e. clockwise, we automatically take right turns at the q-vertices.

5.4 Vertex Merging

The rationale behind the vertex merging is the observation that during tracing due to orientation changes we visit the same local UVs multiple times, creating clones of what would be only one q-vertex if we would "iron out" the fold-overs that cause them. Algorithm 7 identifies q-vertices that share the same local UVs and merges them into a single vertex. Since all vertices within the same q-face share at most four unique local UVs, after the merging algorithm the only possible q-face types are quads, 2-gons and 1-gons. The latter two are collapsed trivially which is what SIMPLIFYFACES (line 15) does. 3-gons cannot occur: For a triangle we would need a diagonal connection in the UV unit-quad. However, during vertex merging no new connections are created and during face extractions only horizontal, vertical and cyclic connections are generated.

5.5 Recovering Lost Q-Edges

In Section 5.2 we explained why tracing any face in our extracted mesh we can only visit a limited set of parameter coordinates in the local coordinate system of the face. Our argument was based on the assumption that two subsequent q-ports at a q-vertex only point into the same direction if one of them lies in a flipped triangle. However, this assumption fails in one special case: around a vertex-q-vertex we can arrange the fold-overs in such a way that the entire triangle fan spans an absolute range of less than 180°. An instance of this is illustrated in Figure 2 (d).

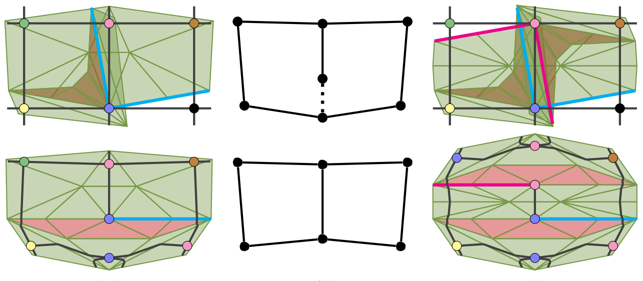


Figure 5: *A q-vertex with only one extracted q-port. (a) The unprocessed configuration, (b) during preprocessing a q-edge is inserted resulting in two quads after merging, (c) a synthetic failure case for our strategy. Cut edges are highlighted in cyan and magenta.*

In this case the only extracted q-port of the q-vertex lies next to itself so that two subsequent ports point into the same direction even though the orientation of their triangles does not differ. This results in a face which covers more than one unit quad in the parameter domain and does not permit the extraction of consistent local UVs since it includes a singularity. Figure 5 (a) illustrates such a face.

If we apply Algorithm 7 with inconsistent local UVs, a non-manifold, non-quad configuration is the result. The problem is that two instances of what would be the same q-vertex, if the fold-over was ironed out, exist but without an integer iso-line between them they cannot be merged so that two should-be separate faces become one.

An approach towards a general solution to this problem could involve a global search for instances of the same q-vertex. However, the only instances of this problem we ever encountered in practice were such constellations where all copies of the same q-vertex are in the same extracted q-face. Thus, our pragmatic approach to this problem is a preprocessing step performed before Algorithm 7: For each q-vertex *with missing ports* we traverse all of its incident q-faces, looking for q-vertices that share its local UVs. We then create a pair of new ports for each q-vertex we find and connect them. This way the faces are successfully separated and the precondition for Algorithm 7 is established as Figure 5 (b) illustrates.

Q-vertices with missing ports are determined by comparing their valence as induced by the parametrization to the actual number of

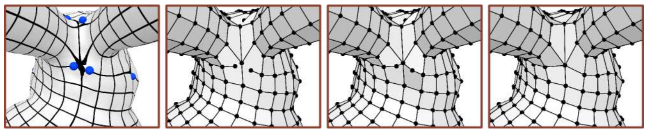


Figure 6: *The back of the* BUNNY*'s head exhibits two singularities with missing parameter lines next to each other. Both singularities* should *have valence 5 but are only intersected by one parameter line. At these locations two 8-gons (with one edge occurring twice in a row) are extracted. Then one of the missing q-edges is restored. After the vertex merging algorithm, the extracted mesh is all-quadrilateral and the two valence 5 singularities are merged to one vertex of valence 6.*

```
1:  Input: triangle mesh $(V, E, T)$, sanitized map $\mathbf{f}$
2:  for each vertex $\mathbf{p} \in T$ do
3:      let $\{c_0, \ldots, c_{n-1}\} = \angle(\mathbf{p}),\ c_i = \{\mathbf{p}, t_i\}$
4:      let $\phi_i$ be the signed inner angle of $\mathbf{f}(t_i)$ at $c$
5:      $\Phi \leftarrow 0$
6:      split sequence $\{c_i\}$ into subsequences with uniform $\mathrm{sgn}(\phi_i)$
7:      for each subsequence $\{c_j, \ldots, c_k\}$ do
8:          $\phi \leftarrow \sum_{i=j,\ldots,k} \phi_i$
9:          if $\mathrm{sgn}(\phi) < 0$ then
10:             $\Phi \leftarrow \Phi + 2\pi + \phi$
11:         else
12:             $\Phi \leftarrow \Phi + \phi$
13:     $\mathrm{val}_\mathbf{p} \leftarrow \mathrm{ROUND}(2\Phi/\pi)$
```

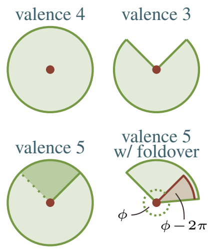
Algorithm 8: *Valence Counting*

extracted ports. Section 5.6 explains how we determine a q-vertex' valence from the parametrization. Figure 6 illustrates the three steps involved in extracting such a configuration.

Figure 5 (c) illustrates a contrived example in which the described strategy fails: multiple ports are missing on singularities adjacent to one another due to an intricate fold-over configuration. However, even by prescribing extremely large target edge lengths (cf. Figure 8) we were unable to produce a failure case in our experiments.

5.6 Valence Counting

In order to determine whether fold-overs caused the q-port extraction step to miss certain parameter lines we need to know the valence a q-vertex *is supposed to* have and then compare it to the number of q-ports that were actually extracted. A naïve approach to obtaining this information is to accumulate the inner angles of all incident triangles in the parameter domain. Figure 2 (d), however, demonstrates why in the presence of fold-overs this approach fails: We would count an inner angle of $\pi/2$ instead of $5\pi/2$. This is because a local fold-over turns a run of triangles with an integrated inner angle of ϕ into flipped triangles with an integrated signed inner angle of $\phi - 2\pi$. Thus, for correct valence counting Algorithm 8 divides the triangle fan into runs of triangles with equal orientation and sums up the corrected inner angles.



Note that in the presence of almost degenerate triangles numerical inaccuracies can lead to large errors in the inner angles ϕ_i if regular floating point arithmetic is used to compute them. Thus, if valence information is available from a processing step performed prior to quad extraction, as is the case in all cross-field based parametrization methods such as QuadCover [Kälberer et al. 2007] or MIQ [Bommes et al. 2009], it is generally preferable to use that information instead of recomputing the valence from the parametrization.

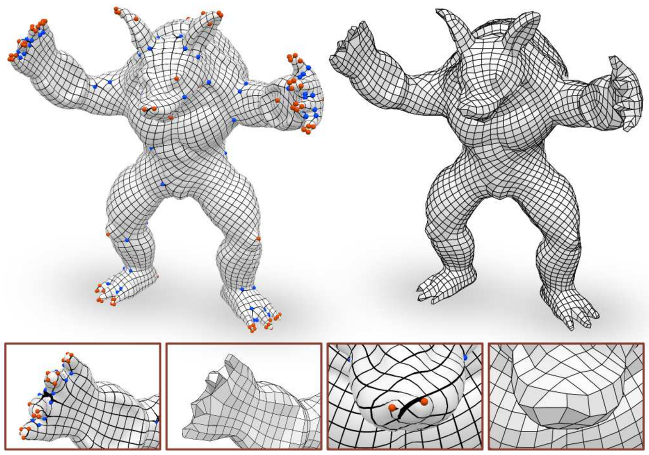


Figure 7: *The* ARMADILLO *parametrized using MIQ without stiffening in 7.4s. Contains 78 flipped faces and two with zero area.*

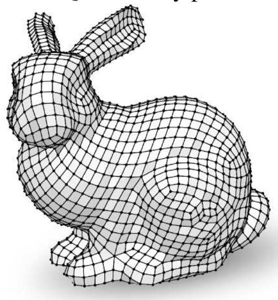
6 Results

To demonstrate the effectiveness of our approach we used parametrizations obtained using the QuadCover method [Kälberer et al. 2007] and the MIQ method [Bommes et al. 2009] (without stiffening). We have put no effort into tuning parameters or making manual adjustments so as to demonstrate QEx' robustness against imperfect parametrizations. Both methods generate parametrizations in a relatively fast manner but tend to produce more and more fold-overs with increasing target edge length. Figures 7–11 show how in a great variety of settings our approach extracts consistent quad meshes out of fold-over ridden parametrizations.

In Algorithm 7 we positioned the merged vertices in the center of gravity. A better choice is to use [Zhang et al. 2005] to embed these vertices which is what we did in Figure 1, 7 and with the BOTIJO in Figure 9.

We also produced MIQ parametrizations and applied stiffening iterations until all fold-overs were eliminated (if possible). Stiffening incurs a run time impact and as target edge lengths grow it cannot effectively guarantee fold-over-freeness. Table 1 shows some timings. A "fail" in the stiffening row represents a failure to generate a fold-over free parametrization after allotting 100 times the run time the pure MIQ approach took. Figure 10 shows the differences between a stiffened mesh and one extracted from an unstiffened parametrization. The run time of our algorithm was between 30ms and 160ms for all examples depicted here except Figure 11.

We also performed experiments with the Reliable MIQ method [Bommes et al. 2013a] (see Figure 10). RMIQ effectively prevents fold-overs but does so at a significant run time impact as compared to pure MIQ.



To mitigate the suboptimal element quality that usually arises in the vicinity of fold-overs, tangential smoothing [Zhang et al. 2005] can also be applied globally. See the example on the right where we post-processed the BUNNY quad mesh from Figure 1.

7 Limitations and Future Work

As explained in Section 5.5 QEx is not robust against *all* conceivable types of fold-overs. We managed to hand-craft synthetic fold-

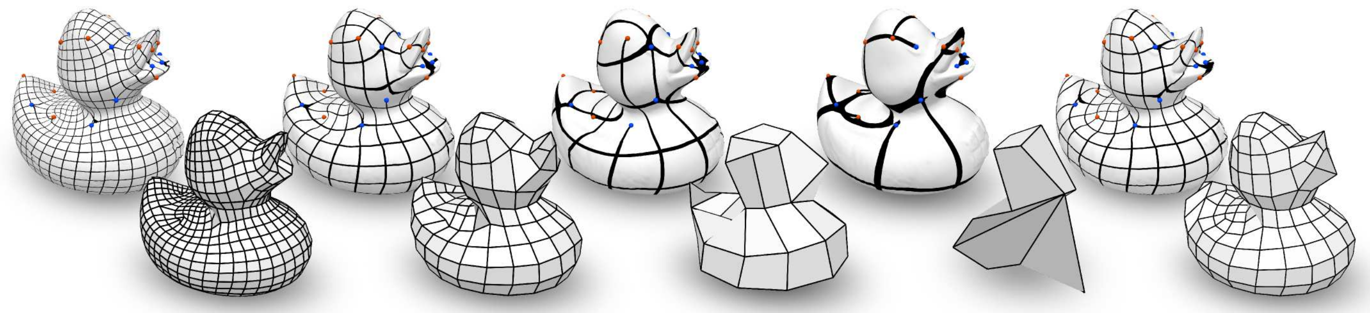


Figure 8: *The* DUCK. *Largest bounding box edge is 84. Back: parametrizations as generated by MIQ without stiffening (first four) and QuadCover (rightmost) using a given cross field and target edge lengths 4, 8, 16, 28 and 8. The MIQ parametrizations took* 1.2 *seconds each, the QuadCover parametrization took* $750ms$. *Front: extracted quad meshes. Note that even the coarsest quad mesh still consists of one single, manifold, watertight component. While meshes that coarse obviously do not properly approximate the surface geometry, they may well be of interest as layouts or base meshes in various applications [Campen et al. 2012; Bommes et al. 2013b].*

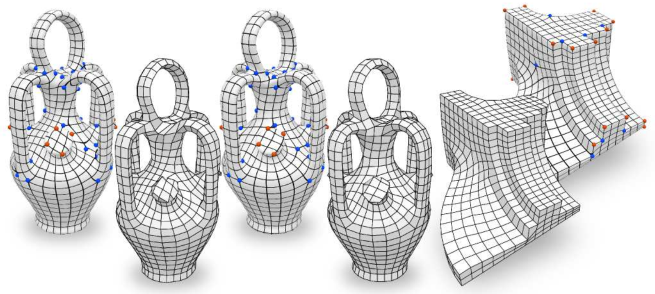


Figure 9: *Left: The* BOTIJO, *parametrized using MIQ without stiffening with target edge length 10 (in 3.9s, 47 flipped triangles). Center: QuadCover (in 1.6s, 116 flipped triangles). Right: The* FANDISK *parametrized using MIQ with feature alignment constraints (in 0.6s, 9 flipped triangles).*

overs that lead to non-quad meshes. It might be possible to provoke such fold-overs in the real world using excessively large target edge lengths, however we have yet to see such a case in practice. It might be worth investigating if our method could be made provably robust against any type of fold-over maybe by employing some more global strategy to resolve flipped configurations.

Note that our quad extraction method does not guarantee to preserve singularities as given by the parametrization. That being said, all singularities with *sane* neighborhoods in the parameter domain *are preserved*. However, if fold-overs span across multiple singularities, they may get merged. Looking at the rightmost meshes in Figure 8 or even at the back of the BUNNY's head in Figure 6 it becomes apparent that actually preserving the singularities would require global changes to the given quad layout and might not be the desired result.

Other methods which require tracing of lines in a parametrization might benefit from adaptations of our approach: to HexCover [Nieser et al. 2012] our method should be applicable with little modifications. Extending the q-ports representation and vertex merging method, it should even be adaptable to hexahedral extraction for, e.g. the CubeCover approach [Nieser et al. 2011].

8 Conclusion

We presented QEx, a method to robustly extract consistent quad meshes even out of imperfect integer-grid maps as they arise from state of the art parametrization based quad meshing methods. Our method not only handles numerical inaccuracies in the parametrization by locally truncating parameter precision but is also robust against local fold-overs. This robustness can be leveraged to generate valid all-quadrilateral meshes using the quickest known quad meshing methods, enabling, for instance, better interactivity when manually designing and fine-tuning quadrangulations.

Mesh	DUCK		ROCKERARM		BOTIJO	
Edge Length	4	8	.03	.1	3	10
QuadCover	.75s	.75s	3.0s	3.0s	1.6s	1.6s
MIQ	1.2s	1.2s	4.7s	4.2s	3.9s	3.9s
w/ stiffening	5.9s	fail	135s	fail	10.1s	fail

Table 1: *Run time comparison of several approaches. Without QEx, stiffening is necessary to remove fold-overs. Using our quad extraction method we can make use of parametrizations generated in a fraction of the time using MIQ or QuadCover.*

We provide `libQEx`, an open source reference implementation of QEx available for download at:
`http://www.rwth-graphics.de/software/libQEx`.

9 Acknowledgements

This project was funded by the DFG Cluster of Excellence on Ultra High-Speed Mobile Information and Communication (UMIC), German Research Foundation grant DFG EXC 89 and the European Research Council (ERC Starting Grant "Robust Geometry Processing", Grant agreement 257474). We would like to thank Jan Möbius for creating and maintaining the geometry processing framework *OpenFlipper* as well as the reviewers for their helpful comments.

References

ALLIEZ, P., COHEN-STEINER, D., DEVILLERS, O., LÉVY, B., AND DESBRUN, M. 2003. Anisotropic polygonal remeshing. In *Proc. SIGGRAPH 2003*.

BOMMES, D., ZIMMER, H., AND KOBBELT, L. 2009. Mixed-integer quadrangulation. In *Proc. SIGGRAPH 2009*.

BOMMES, D., CAMPEN, M., EBKE, H.-C., ALLIEZ, P., AND KOBBELT, L. 2013. Integer-grid maps for reliable quad meshing. In *Proc. SIGGRAPH 2013*.

BOMMES, D., LÉVY, B., PIETRONI, N., PUPPO, E., SILVA, C., TARINI, M., AND ZORIN, D. 2013. Quad-mesh generation and processing: A survey. *Computer Graphics Forum*.

CAMPEN, M., BOMMES, D., AND KOBBELT, L. 2012. Dual loops meshing: quality quad layouts on manifolds. *ACM Trans. Graph. 31*, 4 (July), 110:1–110:11.

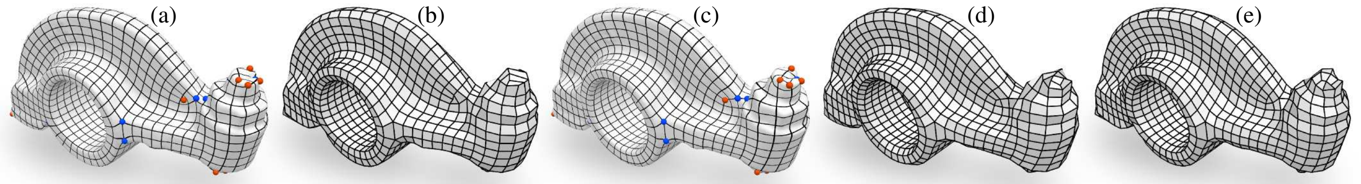


Figure 10: *The* ROCKERARM. *Largest bounding box edge is 1. The target edge length used here was .03. Parametrization (a) was generated without stiffening (in 4.0 seconds), contains 21 flipped triangles and needs a quad-extractor robust against fold-overs to create (b). Parametrization (c) was generated using MIQ with stiffening until all fold-overs were eliminated (in 135 seconds) and was extracted (d) using an existing quad-extractor. Quad mesh (e) was extracted from an RMIQ parametrization (which took 13.3 seconds to generate).*

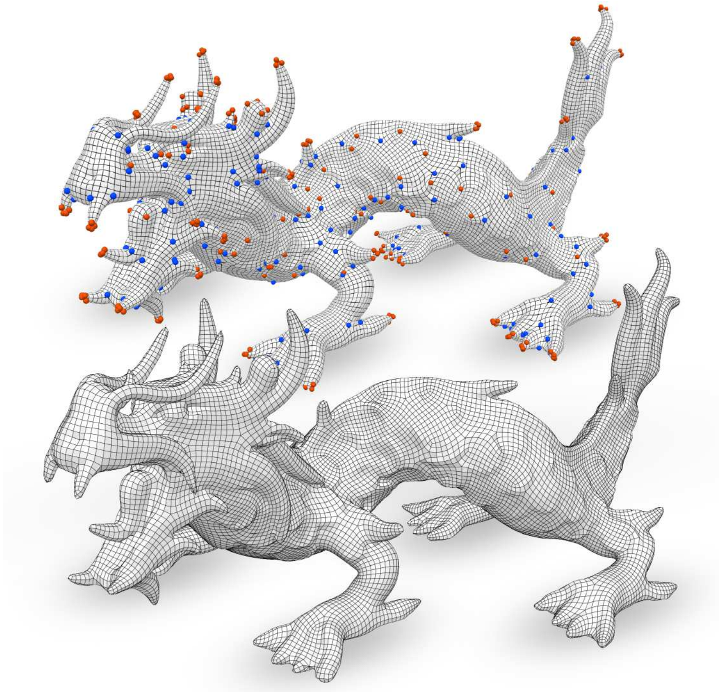


Figure 11: *The* DRAGON. *Top: the parametrization (generated with QuadCover in 25s) contains 632 singularities, 411 flipped triangles and 4 zero area triangles. Bottom: the quad mesh with 23157 faces was extracted in 550ms and tangentially smoothed.*

DONG, S., KIRCHER, S., AND GARLAND, M. 2005. Harmonic functions for quadrilateral remeshing of arbitrary manifolds. *Comput. Aided Geom. Des. 22*, 5 (July), 392–423.

DONG, S., BREMER, P.-T., GARLAND, M., PASCUCCI, V., AND HART, J. C. 2006. Spectral surface quadrangulation. In *ACM SIGGRAPH 2006 Papers*, ACM.

EDELSBRUNNER, H., AND MÜCKE, E. P. 1990. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph. 9*, 1 (Jan.), 66–104.

FORTUNE, S. 1995. Numerical stability of algorithms for 2d delaunay triangulations. *International Journal of Computational Geometry & Applications 5*, 01n02, 193–213.

HOFFMANN, C. M. 2001. Robustness in geometric computations. *Journal of Computing and Information Science in Engineering 1*, 2, 143–156.

HUANG, J., ZHANG, M., MA, J., LIU, X., KOBBELT, L., AND BAO, H. 2008. Spectral quadrangulation with orientation and alignment control. In *Proc. SIGGRAPH Asia 2008*.

KÄLBERER, F., NIESER, M., AND POLTHIER, K. 2007. Quadcover - surface parameterization using branched coverings. *Computer Graphics Forum 26*, 3, 375–384.

LI, E., LÉVY, B., ZHANG, X., CHE, W., DONG, W., AND PAUL, J.-C. 2011. Meshless quadrangulation by global parameterization. *Computers & Graphics 35*, 5.

LI, Y., LIU, Y., XU, W., WANG, W., AND GUO, B. 2012. All-hex meshing using singularity-restricted field. *ACM Trans. Graph. 31*, 6 (Nov.).

MARINOV, M., AND KOBBELT, L. 2004. Direct anisotropic quad-dominant remeshing. In *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, PG '04.

MYLES, A., AND ZORIN, D. 2013. Controlled-distortion constrained global parametrization. *ACM Trans. Graph. 32*, 4.

MYLES, A., PIETRONI, N., KOVACS, D., AND ZORIN, D. 2010. Feature-aligned t-meshes. *ACM Trans. Graph. 29*, 4, 1–11.

NIESER, M., REITEBUCH, U., AND POLTHIER, K. 2011. Cubecover– parameterization of 3d volumes. *Computer Graphics Forum 30*, 5, 1397–1406.

NIESER, M., PALACIOS, J., POLTHIER, K., AND ZHANG, E. 2012. Hexagonal global parameterization of arbitrary surfaces. *IEEE Trans. Vis. Comput. Graph. 18*, 6, 865–878.

PRIEST, D. 1991. Algorithms for arbitrary precision floating point arithmetic. In *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*, 132–143.

RAY, N., LI, W. C., LÉVY, B., SHEFFER, A., AND ALLIEZ, P. 2006. Periodic global parameterization. *ACM Trans. Graph. 25*, 4.

SHEWCHUK, J. R. 1996. Robust Adaptive Floating-Point Geometric Predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*.

TONG, Y., ALLIEZ, P., COHEN-STEINER, D., AND DESBRUN, M. 2006. Designing quadrangulations with discrete harmonic forms. In *Proc. SGP '06*.

YAP, C. K. 1988. A geometric consistency theorem for a symbolic perturbation scheme. In *Proceedings of the fourth annual symposium on Computational geometry*.

ZHANG, Y., BAJAJ, C., AND XU, G. 2005. Surface smoothing and quality improvement of quadrilateral/hexahedral meshes with geometric flow. In *Proc. of the 14th IMR*, B. Hanks, Ed.

ZHANG, M., HUANG, J., LIU, X., AND BAO, H. 2010. A wave-based anisotropic quadrangulation method. In *Proc. SIGGRAPH 2010*.