Eurographics 2008 Full-Day Tutorial

Geometric Modeling Based on Polygonal Meshes

Mario Botsch[1] Mark Pauly[1] Leif Kobbelt[2] Pierre Alliez[3] Bruno Lévy[4]
Stephan Bischoff[2] Christian Rössl[3]

[1]ETH Zurich
[2]RWTH Aachen
[3]INRIA Sophia Antipolis - Méditerranée
[4]INRIA Nancy - Grand Est

Course Organizers

Dr. Mario Botsch
Lecturer and Senior Researcher
Computer Graphics Laboratory, ETH Zurich
botsch@inf.ethz.ch
http://graphics.ethz.ch/~mbotsch

Dr. Mark Pauly
Assistant Professor
Applied Geometry Group, ETH Zurich
pauly@inf.ethz.ch
http://graphics.ethz.ch/~pauly

Course Presenters

Dr. Pierre Alliez
Senior Researcher
INRIA Sophia Antipolis - Méditérranée BP 93 GEOMETRICA
pierre.alliez@sophia.inria.fr
http://www-sop.inria.fr/geometrica/team/Pierre.Alliez/

Dr. Mario Botsch
Lecturer and Senior Researcher
Computer Graphics Laboratory, ETH Zurich
botsch@inf.ethz.ch
http://graphics.ethz.ch/~mbotsch

Dr. Leif Kobbelt
Professor
Computer Graphics Group, RWTH Aachen
kobbelt@cs.rwth-aachen.de
http://www.rwth-graphics.de

Dr. Bruno Lévy
Senior Researcher
INRIA Nancy Grand Est ALICE
Bruno.Levy@inria.fr
http://www.loria.fr/~levy/

Dr. Mark Pauly
Assistant Professor
Applied Geometry Group, ETH Zurich
pauly@inf.ethz.ch
http://graphics.ethz.ch/~pauly

Course Syllabus

09:00–09:10	**Introduction**	Botsch
09:10–09:50	**Surface Representations** – Explicit / implicit surface representations – Polygonal meshes	Kobbelt
09:50–10:30	**Mesh Repair** – Types of input data – Surface-based vs. volumetric repair	Kobbelt
11:00–11:50	**Mesh Smoothing** – Discrete differential geometry – Diffusion & curvature flow – Energy minimization, fairing	Pauly
11:50–12:30	**Mesh Decimation** – Vertex clustering – Incremental decimation	Pauly
14:00–14:45	**Remeshing** – Isotropic – Quadrangle – Error-driven	Alliez
14:45–15:30	**Mesh Parametrization** – Harmonic maps, conformal maps – Free boundary maps – Linear vs. non-linear methods	Lévy
16:00–17:00	**Mesh Editing** – Multiresolution editing – Differential coordinates – Numerics: Efficient linear system solvers – Linear vs. non-linear methods	Botsch
17:00–17:30	**Wrap-Up** – Course summary – Demo: Process one model through the whole pipeline – Demonstration of code examples – Q & A	All speakers

Contents

1 Introduction

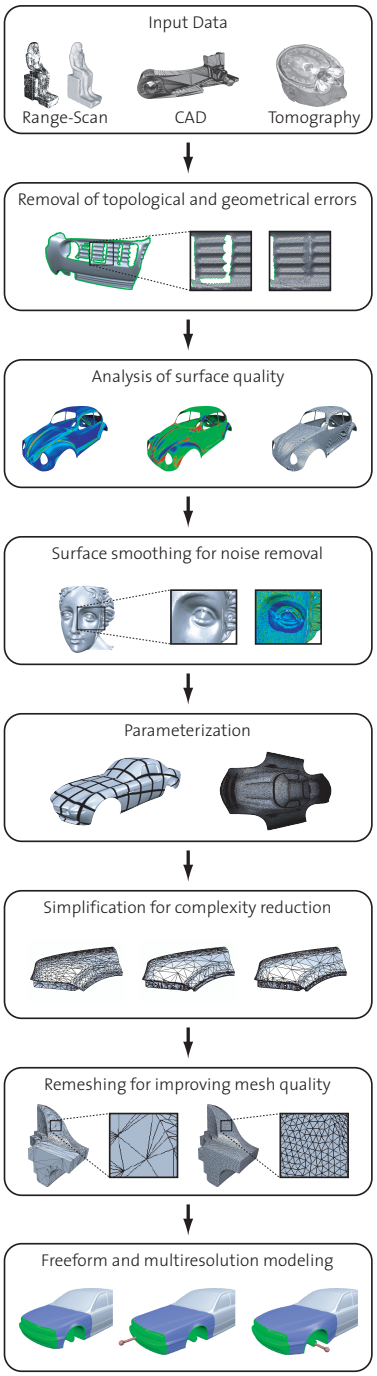
In the last years triangle meshes have become increasingly popular and are nowadays intensively used in many different areas of computer graphics and geometry processing. In classical CAGD irregular triangle meshes developed into a valuable alternative to traditional spline surfaces, since their conceptual simplicity allows for more flexible and highly efficient processing.

Moreover, the consequent use of triangle meshes as surface representation avoids error-prone conversions, e.g., from CAD surfaces to mesh-based input data of numerical simulations. Besides classical geometric modeling, other major areas frequently employing triangle meshes are computer games and movie production. In this context geometric models are often acquired by 3D scanning techniques and have to undergo post-processing and shape optimization techniques before being actually used in production.

This course discusses the whole geometry processing pipeline based on triangle meshes. We will first introduce general concepts of surface representations and point out the advantageous properties of triangle meshes in Chapter **??**, and present efficient data structures for their implementation in Chapter **??**.

The different sources of input data and types of geometric and topological degeneracies and inconsistencies are described in Chapter **??**, as well as techniques for their removal, resulting in clean two-manifold meshes suitable for further processing. Mesh quality criteria measuring geometric smoothness and element shape together with the corresponding analysis techniques are presented in Chapter **??**.

Mesh smoothing reduces noise in scanned surfaces by generalizing signal processing techniques to irregular triangle meshes (Chapter **??**). Similarly, the underlying concepts from differential geometry are useful for surface parametrization as well (Chapter **??**). Due to the enormous complexity of meshes acquired by 3D scanning, mesh decimation techniques are required for error-controlled simplification (Chapter **??**). The shape of triangles, which is important for the robustness of numerical simulations, can be optimized by general remeshing methods (Chapter **??**).



After optimizing meshes with respect to the different quality criteria, we finally present techniques for intuitive and interactive shape deformation (Chapter **??**). Since solving linear systems is a commonly required component for many of the presented mesh processing algorithms, we will discuss their efficient solution and compare several existing libraries in Chapter **??**.

These notes supplement the Eurographics 2008 full day tutorial "Geometric Modeling Based on Polygonal Meshes"*, which was previously also held at SIGGRAPH 2007. The material is partly based on the course notes of our previous tutorial* "Geometric Modeling Based on Triangle Meshes"*, co-authored by Mario Botsch, Mark Pauly, Christian Rössl, Stephan Bischoff, and Leif Kobbelt, which was held at SIGGRAPH 2006 and Eurographics 2006. The current document therefore contains contributions of the 2008, 2007, and 2006 editions of the tutorial.*

Mario Botsch & Mark Pauly
with Leif Kobbelt, Pierre Alliez, and Bruno Lévy
January 2008.

2 Surface Representations

The efficient processing of geometric objects requires — just like in any other field of computer science — the design of suitable data structures. For each specific problem in geometry processing we can identify a characteristic set of operations by which the computation is dominated and hence we have to choose an appropriate data structure which supports the efficient implementation of these operators. From a high level point of view, there are two major classes of surface representations: *parametric* representations and *implicit* representations.

Parametric surfaces are defined by a vector-valued parametrization function $\mathbf{f} : \Omega \to \mathcal{S}$, that maps a two-dimensional parameter domain $\Omega \subset \mathbb{R}^2$ to the surface $\mathcal{S} = \mathbf{f}(\Omega) \subset \mathbb{R}^3$. In contrast, an implicit (or volumetric) surface is defined to be the zero-set of a scalar-valued function $F : \mathbb{R}^3 \to \mathbb{R}$, i.e., $\mathcal{S} = \{\mathbf{x} \in \mathbb{R}^3 \mid F(\mathbf{x}) = 0\}$. Analogously we can define curves in a parametric fashion by functions $\mathbf{f} : \Omega \to \mathcal{S}$ with $\Omega = [a, b] \subset \mathbb{R}$. A corresponding implicit definition is only available for *planar* curves, i.e., $\mathcal{C} = \{\mathbf{x} \in \mathbb{R}^2 \,|\, F(\mathbf{x}) = 0\}$ with $F : \mathbb{R}^2 \to \mathbb{R}$. A simple two-dimensional example is the unit circle, which can be defined by the range of a parametric function:

$$\mathbf{f} : [0, 2\pi] \to \mathbb{R}^2\,, \quad t \mapsto \begin{pmatrix} \cos(t) \\ \sin(t) \end{pmatrix}$$

as well as by the kernel of the implicit function

$$F : \mathbb{R}^2 \to \mathbb{R}\,, \quad (x, y) \mapsto \sqrt{x^2 + y^2} - 1 \ .$$

For more complex shapes it is often not feasible to find an explicit formulation with a single function which approximates the given shape sufficiently accurately. Hence the function domain is usually split into smaller sub-regions and an individual function (*surface patch*) is defined for each segment. In this *piecewise* definition, each function needs to approximate the given shape only locally while the global approximation tolerance is controlled by the size and number of the segments. The mathematical challenge is to guarantee a smooth transition from each patch to its neighboring ones. The most common piecewise surface definition in the parametric case is the segmentation of Ω into triangles or quadrilaterals. For implicit surface definitions, the embedding space is usually split into cubical (*voxels*) or tetrahedral cells.

Both, parametric and implicit representations have their particular strengths and weaknesses, such that for each geometric problem the better suited one should be chosen. In order to analyze geometric operations and their requirements on the surface representation, one can classify them into the following three categories [**?**]:

Evaluation: The sampling of the surface geometry or of other surface attributes, e.g., the surface normal field. A typical application example is surface rendering.

Query: Spatial queries are used to determine whether or not a given point $\mathbf{p} \in \mathbb{R}^3$ is inside or outside of the solid bounded by a surface $\mathcal{S}$, which is a key component for solid modeling operations. Another typical query is the computation of a point's distance to a surface.

Modification: A surface can be modified either in terms of *geometry* (surface deformation), or in terms of *topology*, e.g., when different parts of the surface are to be merged.

We will see in the following that parametric and implicit surface representations have complementary advantages with respect to these three types of geometric operations, i.e., the strengths of the one are often the drawbacks of the other. Hence, for each specific geometric problem the more efficient representation should be chosen, which, in turn, requires efficient conversion routines between the two representations (Section **??**).

2.1 Surface Definition and Properties

The common definition of a surface in the context of computer graphics applications is that of an *orientable continuous two-dimensional manifold embedded in* $\mathbb{R}^3$. Intuitively, this can be understood as the boundary surface of a non-degenerate three-dimensional solid where non-degenerate means that the solid does not have any infinitely thin parts or features such that the surface properly separates the "interior" and "exterior" of the solid. A (non-closed) surface with boundaries is one that can be extended into a proper boundary surface by filling the holes.

Since in most applications the raw information about the input surface is obtained by discrete sampling (i.e., *evaluation* if there already exists a digital representation, or *probing* if the input comes from a real object), the first step in generating a mathematical surface representation is to establish *continuity*. This requires to build a consistent neighborhood relation between the samples.

While this so-called *geodesic* neighborhood relation (in contrast to *spatial* neighborhood) is difficult to access in implicit representations, it is easy to extract from parametric representations where two points on the surface are in geodesic proximity if the corresponding pre-images in Ω are close to each other. From this observation we can derive an alternative characterization of *local manifoldness*: A continuous parametric surface is locally manifold at a surface point $\mathbf{p}$ if for each other surface point $\mathbf{q}$ within a sufficiently small sphere of radius δ around $\mathbf{p}$ the corresponding pre-image is contained in a circle of some radius $\varepsilon = O(\delta)$ around the pre-image of $\mathbf{p}$. A more intuitive way to express this condition is that the surface patch which lies within a sufficiently small δ-sphere around $\mathbf{p}$ is topologically equivalent (homeomorphic) to a disk. Since this second definition does not require a parametrization, it applies to implicit representations as well.

When generating a continuous surface from a set of discrete samples, we can either require this surface to *interpolate* the samples or to *approximate* them subject to a certain prescribed tolerance. The latter case is considered more relevant in practical applications since samples are usually affected by position noise and the surface inbetween the samples is an approximation anyway. In the next section we will consider the issue of approximation in more detail.

Except for a well-defined set of sharp feature curves and corners, a surface should be *smooth* in general. Mathematically this is measured by the number k of continuous derivatives that the functions $\mathbf{f}$ or F have. Notice that this analytical definition of C^k smoothness coincides with the intuitive geometrical understanding of smoothness only if the partial derivatives of $\mathbf{f}$ or the gradient of F, respectively, do not vanish locally.

An even stricter requirement for surfaces is *fairness* where not only the continuity of the derivatives but also their variation is considered. There is no general formal definition of fairness, but a surface is usually considered fair if, e.g., the curvature or its variation is globally minimized (see. Figure **??**).

In Section **??** we will explain how the notion of curvature can be generalized to polygon meshes such that properties like smoothness and fairness can be applied to meshes as well.

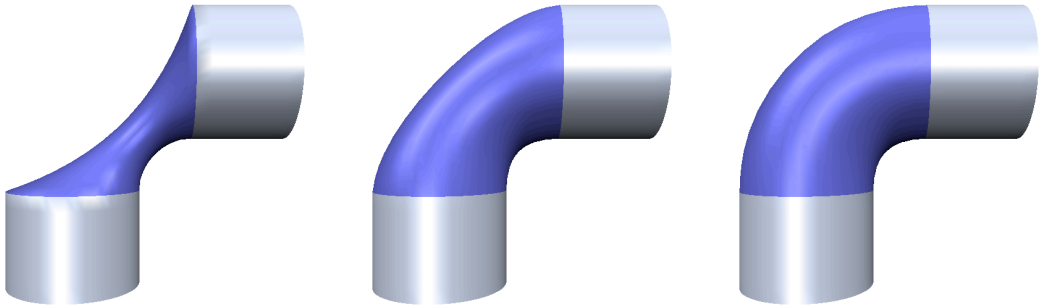


Figure 2.1: This figure shows three examples of fair surfaces, which define a blend between two cylinders. On the left there is a membrane surface which minimizes the surface area. In the center, a thin-plate surface which minimizes curvature. On the right there is a surface which minimizes the variation of mean curvature.

2.2 Approximation Power

The exact mathematical modeling of a real object or its boundary is usually intractable. Hence a digital surface representation can only be an approximation in general. In order to simplify the approximation tasks, the domain of the representation is often split into small segments and for each segment a function (a patch) is defined which locally approximates that part of the input that belongs to this segment.

Since our surface representations are supposed to support efficient processing, a natural choice is to restrict functions to the class of *polynomials* because those can be evaluated by elementary arithmetic operations. Another justification for the restriction to polynomials is the well-known Weierstrass theorem which guarantees that each smooth function can be approximated by a polynomial up to any desired precision.

From calculus we know that a C^∞ function g with bounded derivatives can be approximated over an interval of length h by a polynomial of degree p such that the approximation error behaves like $O(h^{p+1})$ (e.g., Taylor theorem, generalized mean value theorem). As a consequence there are, in principle, two possibilities to improve the accuracy of an approximation with piecewise polynomials. We can either raise the degree of the polynomial (*p-methods*) or we can reduce the size of the individual segments and use more segments for the approximation (*h-methods*).

In geometry processing applications, h-methods are usually preferred over p-methods since for a discretely sampled input surface we cannot make reasonable assumptions about the boundedness of higher order derivatives. Moreover, for piecewise polynomials with higher degree, the C^k smoothness conditions between segments are sometimes quite difficult to satisfy. Finally, with today's computer architectures, processing a large number of very simple objects is often much more efficient than processing a smaller number of more complex ones. This is why the somewhat extremal choice of C^0 piecewise linear surface representations has become the widely established standard in geometry processing.

While for parametric surfaces, the $O(h^{p+1})$ approximation error estimate follows from the mean value theorem in a straightforward manner, a more careful consideration is necessary for implicit representations. The generalized mean value theorem states that if a sufficiently smooth

function g over an interval $[a, a+h]$ is interpolated at the abscissae $t_0, \dots t_p$ by a polynomial f of degree p then the approximation error is bounded by

$$\|f(t) - g(t)\| \;\leq\; \frac{1}{(p+1)!} \max f^{(p+1)} \prod_{i=0}^{p} (t_i - t) \;=\; O(h^{p+1}) \,.$$

For an implicit representation $G : \mathbb{R}^3 \to \mathbb{R}$ and the corresponding polynomial approximant F this theorem is still valid but here the actual surface geometry is not defined by the function values $G(\mathbf{x})$, for which this theorem gives an error estimate, but by the zero level-set of G, i.e., by $\{\mathbf{x} \in \mathbb{R}^3 \,|\, G(\mathbf{x}) = 0\}$.

Consider a point $\mathbf{x}$ on the implicit surface defined by the approximating polynomial F, i.e., $F(\mathbf{x}) = 0$. We can find a corresponding point $\mathbf{x} + \mathbf{d}$ on the implicit surface defined by G, i.e., $G(\mathbf{x}+\mathbf{d}) = 0$ by shooting a ray in normal direction to F, i.e., $\mathbf{d} = d\,\nabla F / \|\nabla F\|$. For a sufficiently small voxel size h, we obtain

$$|F(\mathbf{x}+\mathbf{d})| \;\approx\; |d|\,\|\nabla F(\mathbf{x})\| \quad\Rightarrow\quad |d| \;\approx\; \frac{|F(\mathbf{x}+\mathbf{d})|}{\|\nabla F(\mathbf{x})\|} \,,$$

and from the mean value theorem

$$|F(\mathbf{x}+\mathbf{d}) - G(\mathbf{x}+\mathbf{d})| \;=\; |F(\mathbf{x}+\mathbf{d})| \;=\; O(h^{p+1}) \,.$$

which yields $|d| = O(h^{p+1})$ if the gradient $\|\nabla F\|$ is bounded from below by some $\varepsilon > 0$. In practice one tries to find an approximating polynomial F with low gradient *variation* in order to have a uniform distribution of the approximation error.

2.3 Parametric Surface Representations

Parametric surface representations have the advantage that the function $\mathbf{f} : \Omega \to \mathcal{S}$ enables the reduction of several three-dimensional problems on the surface $\mathcal{S}$ to two-dimensional problems in the parameter domain Ω. For instance, points on the surface can easily be generated by simple function evaluations of $\mathbf{f}$, which obviously allows for efficient evaluation operations. In a similar manner, geodesic neighborhoods, i.e., neighborhoods on the surface $\mathcal{S}$, can easily be found by considering neighboring points in the parameter domain Ω. A simple composition of $\mathbf{f}$ with a deformation function $\mathbf{d} : \mathbb{R}^3 \to \mathbb{R}^3$ results in an efficient modification of the surface geometry.

On the other hand, generating a parametric surface parameterization $\mathbf{f}$ can be very complex, since the parameter domain Ω has to match the topological and metric structure of the surface $\mathcal{S}$ (Chapter **??**). When changing the shape of $\mathcal{S}$, it might be necessary to update the parameterization accordingly in order to reflect the respective changes of the underlying geometry: A low-distortion parameterization requires the metrics in $\mathcal{S}$ and Ω to be similar, and hence we have to avoid or adapt to excessive stretching.

Since the surface $\mathcal{S}$ is defined as the range of the parameterization $\mathbf{f}$, its topology is equivalent to that of Ω if $\mathbf{f}$ is continuous and injective. This implies that changing the topology of a parametric surface $\mathcal{S}$ can be extremely complicated, since not only the parameterization but also the domain Ω has to be adjusted accordingly. The typical inside/outside or signed distance queries are in general also very expensive on parametric surfaces. The same applies to the detection of self-collisions (= non-injectivities). Hence, topological modification and spatial queries are definitely the weak points of parametric surfaces.

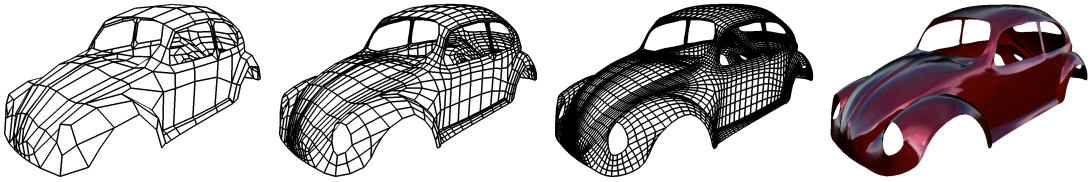


Figure 2.2: Subdivision surfaces are generated by an iterative refinement of a coarse control mesh.

2.3.1 Spline Surfaces

Tensor-product spline surfaces are the standard surface representation of today's CAD systems. They are used for constructing high-quality surfaces from scratch as well as for later surface deformation tasks. Spline surfaces can conveniently be described by the B-spline basis functions $N_i^n(\cdot)$, for more detail see [**?**, **?**, **?**].

A tensor product spline surface $\mathbf{f}$ of degree n is a piecewise polynomial surface that is built by connecting several polynomial patches in a smooth C^{n-1} manner:

$$\begin{aligned} \mathbf{f} : [0,1]^2 &\rightarrow \mathbb{R}^3 \\ (u,v) &\mapsto \sum_{i=0}^{m}\sum_{j=0}^{m} \mathbf{c}_{ij} N_i^n(u) N_j^n(v) \ . \end{aligned}$$

The *control points* $\mathbf{c}_{ij} \in \mathbb{R}^3$ define the so-called *control grid* of the spline surface. Because $N_i^n(u) \geq 0$ and $\sum_i N_i^n \equiv 1$, each surface point $\mathbf{f}(u,v)$ is a convex combination of the control points $\mathbf{c}_{ij}$, i.e., the surface lies within the convex hull of the control grid. Due to the small support of the basis functions, each control point has local influence only. These two properties cause spline surfaces to closely follow the control grid, thereby providing a geometrically intuitive metaphor for modeling surfaces by adjusting its control points.

A tensor-product surface — as the image of a rectangular domain under the parameterization $\mathbf{f}$ — always represents a rectangular surface patch embedded in $\mathbb{R}^3$. If shapes of more complicated topological structure are to be represented by spline surfaces, the model has to be decomposed into a large number of (possibly trimmed) tensor-product patches.

As a consequence of these *topological constraints*, typical CAD models consist of a huge collection of surface patches. In order to represent a high quality, globally smooth surface, these patches have to be connected in a smooth manner, leading to additional *geometric constraints*, that have to be taken care of throughout all surface processing phases. The large number of surface patches and the resulting topological and geometric constraints significantly complicate surface construction and in particular the later surface modeling tasks.

2.3.2 Subdivision Surfaces

Subdivision surfaces [**?**] can be considered as a generalization of spline surfaces, since they are also controlled by a coarse *control mesh*, but in contrast to spline surfaces they can represent surfaces of arbitrary topology. Subdivision surfaces are generated by repeated refinement of control meshes: After each topological refinement step, the positions of the (old and new) vertices are

adjusted based on a set of local averaging rules. A careful analysis of these rules reveals that in the limit this process results in a surface of provable smoothness (cf. Fig. **??**).

As a consequence, subdivision surfaces are restricted neither by topological nor by geometric constraints as spline surfaces are, and their inherent hierarchical structure allows for highly efficient algorithms. However, subdivision techniques are restricted to surfaces with so-called semi-regular *subdivision connectivity*, i.e., surface meshes whose triangulation is the result of repeated refinement of a coarse control mesh. As this constraint is not met by arbitrary surfaces, those would have to be *remeshed* to subdivision connectivity in a preprocessing step [**?**, **?**, **?**, **?**]. But as this remeshing corresponds to a resampling of the surface, it usually leads to sampling artifacts and loss of information. In order to avoid the restrictions caused by these *connectivity constraints*, our goal is to work on arbitrary triangle meshes, as they provide higher flexibility and also allow for efficient surface processing.

2.3.3 Triangle Meshes

In many geometry processing algorithms triangle meshes are considered as a collection of triangles without any particular mathematical structure. In principle, however, each triangle defines, via its barycentric parametrization, a linear segment of a piecewise linear surface representation.

Every point $\mathbf{p}$ in the interior of a triangle $[\mathbf{a}, \mathbf{b}, \mathbf{c}]$ can be written as a barycentric combination of the corner points:

$$\mathbf{p} \;= \alpha\,\mathbf{a} + \beta\,\mathbf{b} + \gamma\,\mathbf{c}$$

with

$$\alpha + \beta + \gamma \;=\; 1$$

By choosing an arbitrary triangle $[\mathbf{u}, \mathbf{v}, \mathbf{w}]$ in the parameter domain, we can define a linear mapping $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^3$ with

$$\alpha\,\mathbf{u} + \beta\,\mathbf{v} + \gamma\,\mathbf{w} \quad \mapsto \quad \alpha\,\mathbf{a} + \beta\,\mathbf{b} + \gamma\,\mathbf{c} \tag{2.1}$$

In Chapter **??** we will discuss methods to choose the triangulation in the parameter domain such that the distortion caused by the mapping from $\mathbb{R}^2$ to $\mathbb{R}^3$ is minimized.

A triangle mesh $\mathcal{M}$ consists of a geometric and a topological component, where the latter can be represented by a graph structure (simplicial complex) with a set of vertices

$$\mathcal{V} = \{v_1, \ldots, v_V\}$$

and a set of triangular faces connecting them

$$\mathcal{F} = \{f_1, \ldots, f_F\}\,, \quad f_i \in \mathcal{V} \times \mathcal{V} \times \mathcal{V}\,.$$

However, as we will see in Chapter **??**, it is sometimes more efficient to represent the connectivity of a triangle mesh in terms of the edges of the respective graph

$$\mathcal{E} = \{e_1, \ldots, e_E\}\,, \quad e_i \in \mathcal{V} \times \mathcal{V}\;.$$

The geometric embedding of a triangle mesh into $\mathbb{R}^3$ is specified by associating a 3D position $\mathbf{p}_i$ to each vertex $v_i \in \mathcal{V}$:

$$\mathcal{P} = \{\mathbf{p}_1, \ldots, \mathbf{p}_V\}\,, \quad \mathbf{p}_i := \mathbf{p}\,(v_i) = \begin{pmatrix} x\,(v_i) \\ y\,(v_i) \\ z\,(v_i) \end{pmatrix} \in \mathbb{R}^3\;,$$

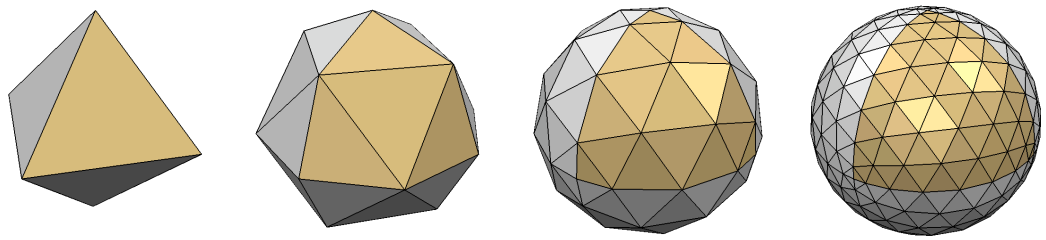


Figure 2.3: Each subdivision step halves the edge lengths, increases the number of faces by a factor of 4, and reduces the error by a factor of $\frac{1}{4}$.

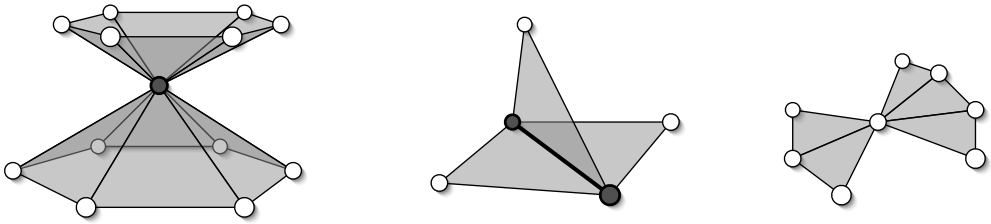


Figure 2.4: Two surface sheets meet at a non-manifold vertex (*left*). A non-manifold edge has more than two incident faces (*center*). The right configuration, although being non-manifold in the strict sense, can be handled by most data structures.

such that each face $f \in \mathcal{F}$ actually represents a triangle in 3-space specified by its three vertex positions. Notice that even if the geometric embedding is defined by assigning 3D positions to the (discrete) vertices, the resulting polygonal surface is still a *continuous* surface consisting of triangular pieces with linear parametrization functions (**??**).

If a sufficiently smooth surface is approximated by such a piecewise linear function, the approximation error is of the order $O(h^2)$, with h denoting the maximum edge length. Due to this quadratic approximation power, the error is reduced by a factor of $1/4$ when halving the edge lengths. As this refinement splits each triangle into four sub-triangles, it increases the number of triangles from F to $4F$ (cf. Fig. **??**). Hence, the approximation error of a triangle mesh is inversely proportional to the number of its faces. The actual magnitude of the approximation error depends on the second order terms of the Taylor expansion, i.e., on the curvature of the underlying smooth surface. From this we can conclude that a sufficient approximation is possible with just a moderate mesh complexity: The vertex density has to be locally adapted to the surface curvature, such that flat areas are sparsely sampled, while in curved regions the sampling density is higher.

As stated before, an important topological quality of a surface is whether or not it is *two-manifold*, which is the case if for each point the surface is locally homeomorphic to a disk (or a half-disk at boundaries). A triangle mesh is two-manifold, if it does neither contain non-manifold edges or non-manifold vertices, nor self-intersections. A *non-manifold edge* has more than two incident triangles and a *non-manifold vertex* is generated by pinching two surface sheets together at that vertex, such that the vertex is incident to two fans of triangles (cf. Fig. **??**). Non-manifold meshes are problematic for most algorithms, since around non-manifold configurations there exists no well-defined local geodesic neighborhood.

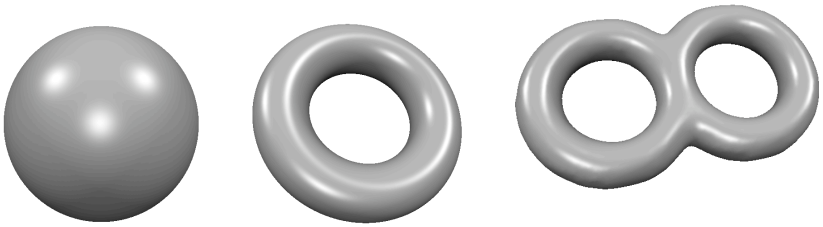


Figure 2.5: From left to right: sphere of genus 0, torus of genus 1, double-torus of genus 2.

The famous Euler formula [**?**] states an interesting relation between the numbers of vertices V, edges E and faces/triangles F in a closed and connected (but otherwise unstructured) mesh:

$$V - E + F = 2(1 - g) \ , \tag{2.2}$$

where g is the genus of the surface and intuitively represents the number of handles of an object (cf. Fig. **??**). Since for typical meshes the genus is small compared to the numbers of elements, the right-hand side of Eq. (**??**) can be assumed to be almost zero. Given this and the fact that each triangle is bounded by three edges and that each (interior) edge is incident to two triangles, one can derive the following mesh statistics:

- The number of triangles is twice the number of vertices: $F \approx 2V$.
- The number of edges is three times the number of vertices: $E \approx 3V$.
- The average vertex valence (number of incident edges) is 6.

These relations will become important when considering data structures or file formats for triangle meshes in Chapter **??**.

For piecewise (polynomial) surface definitions, the most difficult part is the construction of smooth transitions between neighboring patches. Since for triangle meshes, we only require C^0 continuity, we only have to make sure that neighboring faces share a common edge (two common vertices). This makes polygon meshes the most simple and flexible continuous surface representation. Since the development of efficient algorithms for triangle meshes depends on the availability of suitable data structures, we will discuss this topic in detail in Chapter **??**.

2.4 Implicit Surface Representations

The basic concept of *implicit* or *volumetric* representations of geometric models is to characterize the whole embedding space of an object by classifying each 3D point to lie either inside, outside, or exactly on the surface $\mathcal{S}$ bounding a solid object.

There are different representations for implicit functions, like continuous algebraic surfaces, radial basis functions, or discrete voxelizations. In any case, the surface $\mathcal{S}$ is defined to be the zero-level iso-surface of a scalar-valued function $F : \mathbb{R}^3 \rightarrow \mathbb{R}$. By definition, negative function values of F designate points inside the object and positive values points outside the object, such that the zero-level iso-surface $\mathcal{S}$ separates the inside from the outside.

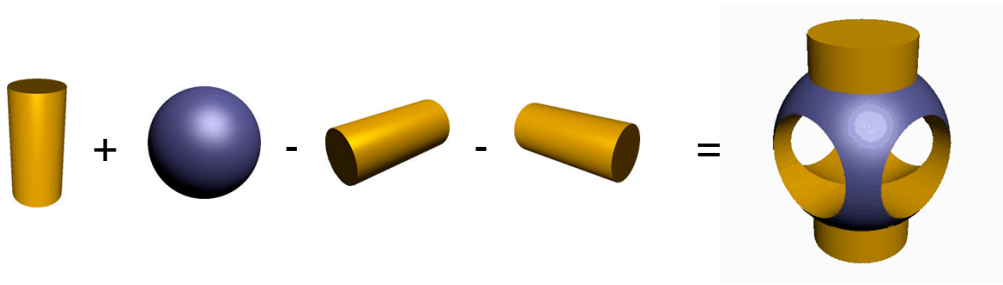


Figure 2.6: A complex object constructed by boolean operations.

As a consequence, geometric inside/outside queries simplify to function evaluations of F and checking the sign of the resulting value. This makes implicit representations well suited for constructive solid geometry (CSG), where complex objects are constructed by boolean operations of simpler ones (cf. Fig. **??**). The different boolean operations can easily be computed by simple min and max combinations of the objects' implicit functions. Hence, implicit surfaces can easily change their topology. Moreover, since an implicit surface is a level-set of a potential function, geometric self-intersections cannot occur, which will later be exploited for mesh repair (Chapter **??**).

The implicit function F for a given surface $\mathcal{S}$ is not uniquely determined, but the most common and most natural representation is the so-called *signed distance function*, which maps each 3D point to its signed distance from the surface $\mathcal{S}$. In addition to inside/outside queries, this representation also simplifies distance computations to simple function evaluations, which can be used to compute and control the global error for mesh processing algorithms [**?**, **?**].

On the other hand, enumerating points on an implicit surface, finding geodesic neighborhoods, and even just rendering the surface is quite difficult. Moreover, implicit surfaces do not provide any means of parameterization, which is why it is almost impossible to consistently paste textures onto evolving implicit surfaces. Furthermore, boundaries cannot be represented.

2.4.1 Regular Grids

In order to efficiently process implicit representations, the continuous scalar field F is typically discretized in some bounding box around the object using a sufficiently dense grid with nodes $\mathbf{g}_{ijk} \in \mathbb{R}^3$. The most basic representation therefore is a uniform scalar grid of sampled values $F_{ijk} := F(\mathbf{g}_{ijk})$, and function values within voxels are derived by tri-linear interpolation, thus providing quadratic approximation order. However, the memory consumption of this naive data structure grows cubically if the precision is increased by reducing the edge length of grid voxels.

2.4.2 Adaptive Data Structures

For better memory efficiecy the sampling density is often adapted to the local geometric significance in the scalar field F: Since the signed distance values are most important in the vicinity of the surface, a higher sampling rate can be used in these regions only. Instead of a uniform 3D grid, a hierarchical octree is then used to store the sampled values [**?**]. The further refinement of an octree cell lying completely inside or outside the object does not improve the approximation of the surface $\mathcal{S}$. Adaptively refining only those cells that are intersected by the surface yields a

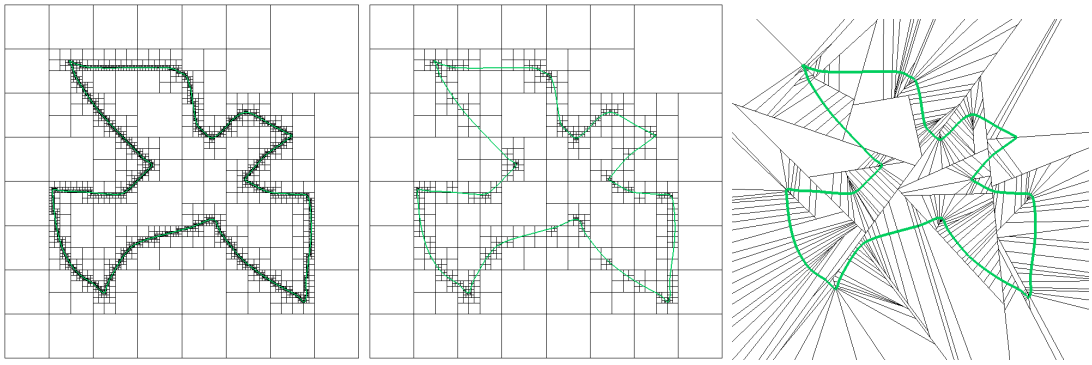


Figure 2.7: Different adaptive approximations of a signed distance field with the same accuracy: 3-color quadtree (left, 12040 cells), ADF [**?**] (center, 895 cells), and BSP tree [**?**] (right, 254 cells).

uniformly refined crust of leaf cells around the surface and reduces the storage complexity from cubic to quadratic (cf. Fig. **??**, left).

If the local refinement is additionally restricted to those cells where the tri-linear interpolant deviates more than a prescribed tolerance from the actual distance field, the resulting approximation adapts to the locality of the surface as well as to its shape complexity [**?**] (cf. Fig. **??**, center). Since extreme refinement is only necessary in regions of high surface curvature, this approach reduces the storage complexity even further and results in a memory consumption comparable to explicit representations. Similarly, an adaptive space-decomposition with linear (instead of tri-linear) interpolants at the leaves can be used [**?**]. Although the asymptotic complexity as well as the approximation power are the same, the latter method provides slightly better memory efficiency (cf. Fig. **??**, right).

2.5 Conversion Methods

In order to exploit the specific advantages of explicit and implicit surface representations efficient conversion methods between the different representations are necessary. However, notice that both kinds of representations are usually finite samplings (triangle meshes in the explicit case, uniform/adaptive grids in the implicit case) and that each conversion corresponds to a re-sampling step. Hence, special care has to be taken in order to minimize loss of information during these conversion routines.

2.5.1 Explicit to Implicit

The conversion of an explicit surface representation to an implicit one amounts to the computation or approximation of its signed distance field. This can be done very efficiently by voxelization or 3D scan-conversion techniques [**?**], but the resulting approximation is piecewise constant only. As a surface's distance field is in general not smooth everywhere, a piecewise linear or piecewise tri-linear approximation seems to be the best compromise between approximation accuracy and computational efficiency. Since we focus on triangle meshes as explicit representation, the conversion to an implicit representation basically requires the computation of signed distances to the triangle mesh at the nodes of a (uniform or adaptive) 3D grid.

Computing the exact distance of a grid node to a given mesh amounts to computing the distance to the closest triangle, which can be found efficiently by spatial data structures. Notice that in order to compute a *signed* distance field, one additionally has to determine whether a grid node lies inside or outside the object. If $\mathbf{g}$ denotes the grid node and $\mathbf{c}$ its closest point on the surface, then the orientation can be derived from the angle between $(\mathbf{g} - \mathbf{c})$ and the normal $\mathbf{n}(\mathbf{c})$: $\mathbf{g}$ is defined to be inside if $(\mathbf{g} - \mathbf{c})^T \mathbf{n}(\mathbf{c}) < 0$. The robustness and reliability of this test strongly depends on the way the normal $\mathbf{n}(\mathbf{c})$ is computed. Using barycentric normal interpolation within triangles' interiors and computing per-vertex normals using angle-weighted averaging of face normals was shown to yield correct results [**?**].

Computing the distances on a whole grid can be accelerated by *fast marching* methods [**?**]. In a first step, the exact signed distance values are computed for all grid nodes in the immediate vicinity of the triangle mesh. After this initialization, the fast marching method propagates distances to the unknown grid nodes in a breadth-first manner.

2.5.2 Implicit to Explicit

The conversion from an implicit or volumetric representation to an explicit triangle mesh, the so-called isosurface extraction, occurs for instance in CSG modeling (cf. Fig. **??**) and in medical applications, e.g., to extract the skull surface from a CT head scan. The de-facto standard algorithm for isosurface extraction is *Marching Cubes* [**?**]. This grid-based method samples the implicit function on a regular grid and processes each cell of the discrete distance field separately, thereby allowing for trivial parallelization. For each cell that is intersected by the iso-surface $\mathcal{S}$ a surface patch is generated based on local criteria. The collection of all these small pieces eventually yields a triangle mesh approximation of the complete iso-surface $\mathcal{S}$.

For each edge intersecting the surface $\mathcal{S}$ the Marching Cubes algorithm computes a sample point which approximates this intersection. In terms of the scalar field F this means that the sign of F differs at the edge's endpoints $\mathbf{p}_1$ and $\mathbf{p}_2$. Since the tri-linear approximation F is actually linear along the grid edges, the intersection point $\mathbf{s}$ can be found by linear interpolation of the distance values $d_1 := F(\mathbf{p}_1)$ and $d_2 := F(\mathbf{p}_2)$ at the edge's endpoints:

$$\mathbf{s} \;=\; \frac{|d_2|}{|d_1| + |d_2|}\,\mathbf{p}_1 \;+\; \frac{|d_1|}{|d_1| + |d_2|}\,\mathbf{p}_2 \;\;.$$

The resulting sample points of each cell are then connected to a triangulated surface patch based on a triangulation look-up table holding all possible configurations of edge intersections (cf. Fig. **??**). Since the possible combinatorial configurations are determined by the signs at a cell's corners, their number is $2^8 = 256$.

Notice that a few cell configuration are ambiguous, which might lead to cracks in the extracted surface. A properly modified look-up table yields a simple and efficient solution, however, at the price of sacrificing the symmetry w.r.t. sign inversion of F [**?**]. The resulting isosurfaces then are watertight 2-manifolds, which is exploited by many mesh repair techniques (Chapter **??**).

Notice that Marching Cubes computes intersection points on the edges of a regular grid only, which causes sharp edges or corners to be "chopped of". A faithful reconstruction of sharp features would instead require additional sample points within the cells containing them. The extended Marching Cubes [**?**] therefore examines the distance function's gradient ∇F to detect those cells and to find additional sample points by intersecting the tangent planes at the edge intersection points. This principle is depicted in Fig. **??**, and a 3D example of the well known

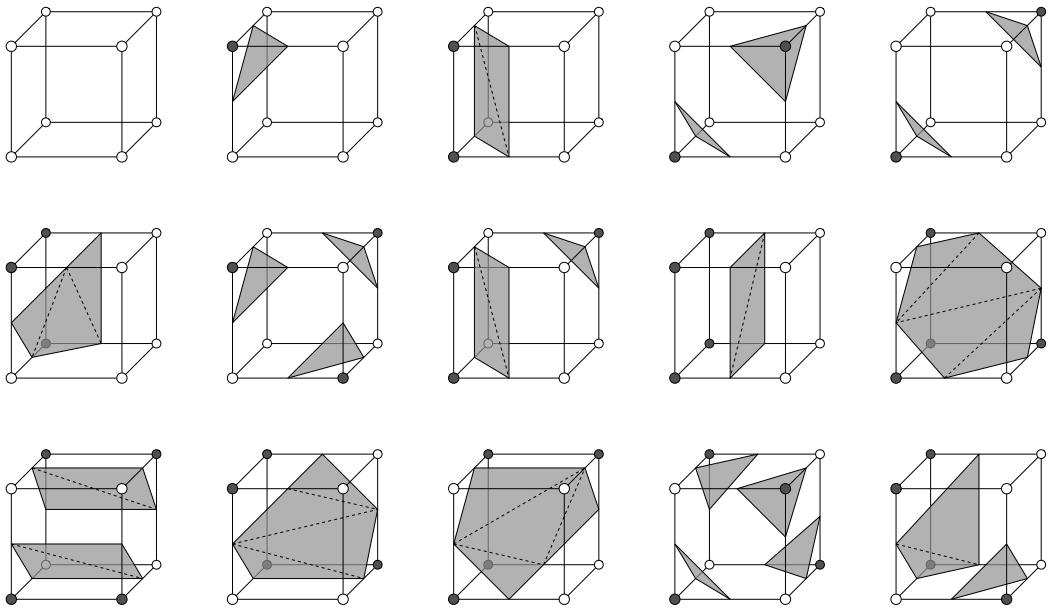


Figure 2.8: The 15 base configurations of the Marching Cubes triangulation table. The other cases can be found by rotation or symmetry.

fandisk dataset is shown in Fig. **??**. An example implementation of the extended Marching Cubes based on the OpenMesh data structure [**?**] can be downloaded from [**?**].

The high complexity of the extracted isosurfaces remains a major problem for Marching Cubes like approaches. Instead of decimating the resulting meshes (Chapter **??**), Ju et al. [**?**] proposed the *dual contouring* approach, which allows to directly extract adaptive meshes from an octree. Notice however that their approach yields non-manifold meshes for cell configurations containing multiple surface sheets. A further promising approach is the cubical marching squares algorithm [**?**], which also provides adaptive and feature-sensitive isosurface extractions.

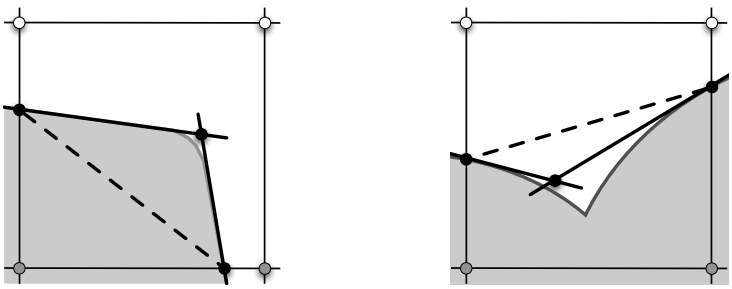


Figure 2.9: By using point and normal information on both sides of the sharp feature one can find a good estimate for the feature point at the intersection of the tangent elements. The dashed line is the result the standard Marching Cubes algorithm would produce.

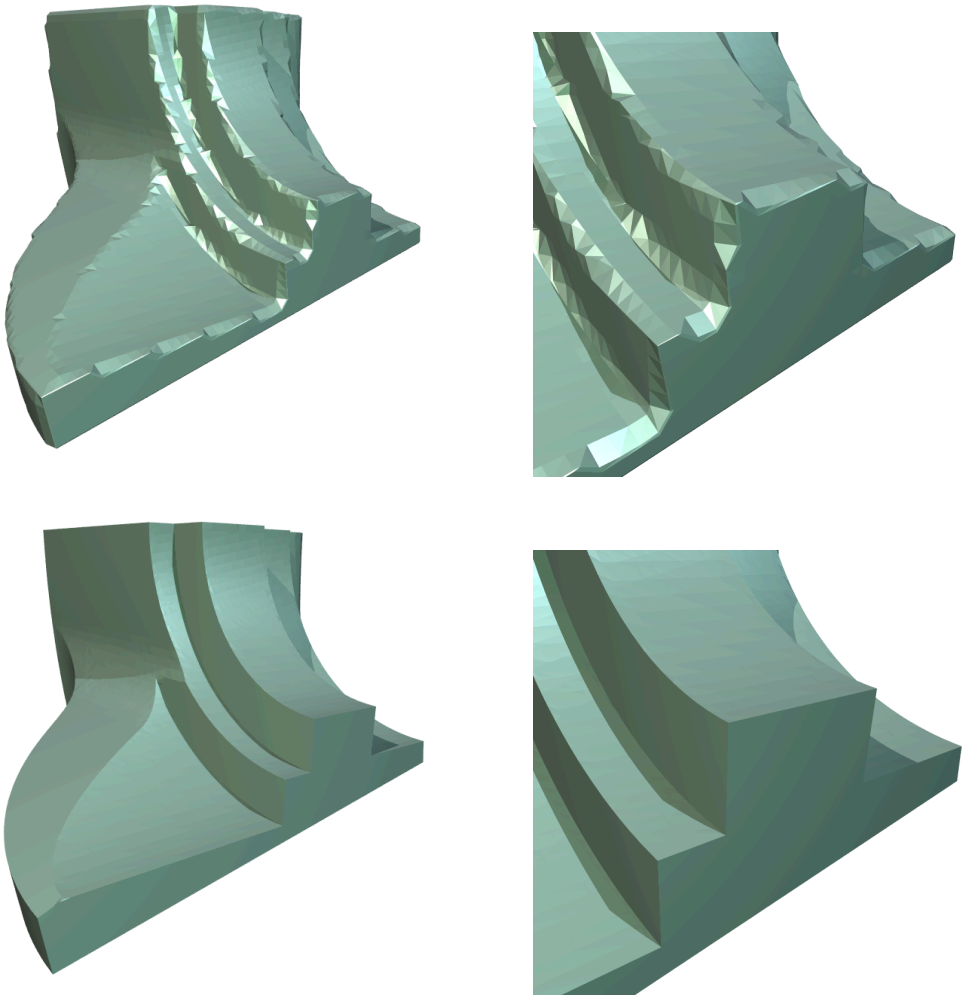


Figure 2.10: Two reconstructions of the "fandisk" dataset from a $65 \times 65 \times 65$ sampling of its signed distance field. The standard Marching Cubes algorithm leads to severe alias artifacts near sharp features (*top*), whereas the feature-sensitive iso-surface extraction faithfully reconstructs them (*bottom*).

3 Mesh Data Structures

The efficiency of the geometric modeling algorithms presented in this tutorial crucially depends on the underlying mesh data structures. A variety of data structures has been described in the literature, and a number of different implementations are available. We refer to [**?**] for an excellent overview and comparison of different mesh data structures and to [**?**, **?**] for references on data structures for representing non-manifold meshes.

In general, when choosing a data structure one has to take into account topological as well as algorithmic considerations:

Topological requirements. Which kinds of meshes need to be represented by the data structure? Do we need boundaries or can we assume closed meshes? Do we need to represent complex edges and singular vertices (see Chapter **??**) or can we rely on a manifold mesh? Can we restrict ourselves to pure triangle meshes or do we need to represent arbitrary polygonal meshes? Are the meshes regular, semi-regular or irregular? Do we want to build up a hierarchy of differently refined meshes or do we need only a flat data structure?

Algorithmic requirements. Which kinds of algorithms will be operating on the data structure? Do we simply want to render the mesh? Do we need to modify only the geometry of the mesh, or do we also have to modify the connectivity/topology? Do we need to associate additional data with the vertices, edges or faces of the mesh? Do we need to have constant-time access to the local neighborhoods of vertices, edges and faces? Can we assume the mesh to be globally orientable?

The simplest representation for triangle meshes would just store a set of *individual* triangles. Some data exchange formats use this representation as a common denominator (e.g., STL format). However, it is immediately clear that this is not sufficient for most requirements: connectivity information cannot be accessed explicitly, and vertices and associated data are replicated. The latter can be fixed by a shared vertex data structure, which stores a table of vertices and encodes triangles as triples of indices into this table. In fact this representation is used in many file formats because it is simple and efficient in storage (assuming no mesh compression is applied). Similarly, it is efficient for certain algorithms that assume static data, e.g., rendering. However, without additional connectivity information this is still not efficient for most algorithms.

Before we go on, we want to identify some minimal set of operations that are frequently used by most algorithms.

- Access of individual vertices, edges, faces. This includes enumeration of *all* elements (in no particular order).
- Oriented traversal of edges of a face, which refers to finding the *next* edge in a face. (This defines also *degree* of the face and the inverse operation for the *previous* halfedge. With additional access to vertices, e.g., rendering of faces is enabled.)

- Access of the faces attached to an edge. Depending on orientation this is either the left or right face in the manifold case. This enables access to neighboring faces and hence traversal of faces (and boundaries as special case).

- Given an edge access its starting and/or end vertex.

- Given a vertex at least one attached face or edge must be accessible. Then (for manifold meshes) all other elements in the one-ring neighborhood of a vertex can be enumerated, i.e., incident faces, edges, or neighboring vertices.

These operations enable local and global traversal of the mesh. They relate vertices, edges and faces by connectivity information (and orientation). We remark that all these operations are possible even for a shared vertex representation, however, this requires expensive searches.

Several data structures have been developed which enable fast traversal of meshes. Well-known are *winged-edge* [**?**], *quad-edge* [**?**], and *half-edge* [**?**] data structures in different flavors (see, e.g., [**?**]).

From our own experience, we have found two of these mesh data structures to be especially suitable for geometry processing: halfedge data structure (Section **??**) and directed edges structure [**?**] (Section **??**) as a special case for triangle meshes. Both data structures allow for efficient enumeration of neighborhoods of vertices and faces. This operation is frequently used in many algorithms, e.g., in mesh smoothing and mesh decimation. The halfedge data structure is able to represent arbitrary polygonal meshes that are subsets of a 2-manifold. The directed edges data structure is more memory efficient, but it can only represent 2-manifold triangle meshes.

3.1 Halfedge Data Structure

One of the most convenient and flexible data structures in geometry processing is the halfedge data structure [**?**, **?**]. This structure is able to represent arbitrary polygonal meshes that are subsets of orientable 2-manifolds. In this data structure each edge is split into two opposing halfedges such that all halfedges are oriented consistently in counter-clockwise order around each face and along the boundary, see Fig. **??**. For each halfedge we store a reference to

- the vertex it points to

- its adjacent face (a zero pointer, if it is a boundary halfedge)

- the next halfedge of the face or boundary (in counter-clockwise direction)

- its inverse (or opposite) halfedge

- the previous half-edge in the face (*optional* for better performance)

Additionally we store references for each face to one of its adjacent halfedges and for each vertex to one of its outgoing halfedges. Thus, a basic halfedge structure can be realized using the following classes:

```
struct Halfedge {
  HalfedgeRef     next_halfedge;
  HalfedgeRef     opposite_halfedge;
  FaceRef         face;
  VertexRef       to_vertex;
};
```

```
struct Face {
  HalfedgeRef     halfedge;
};

struct Vertex {
  HalfedgeRef     outgoing_halfedge;
};
```

This simple structure already enables us to enumerate for each element (i.e. vertex, edge, halfedge or face) its adjacent elements. As an example, the following procedure enumerates all vertices that are adjacent to a given center vertex (the so-called 1-ring)

```
enumerate_1_ring(Vertex * center)
{
   HalfedgeRef h = outgoing_halfedge(center);
   HalfedgeRef hstop = h;
   do {
      VertexRef v = to_vertex(h);
      // do something with v
      h = next_halfedge(opposite_halfedge(h));
   } while ( h != hstop );
}
```

The implementation of the references (e.g., HalfedgeRef) can be realized in different ways, for instance using pointers or indices. In practice, index representations (see, e.g., Section **??**) are more flexible even though memory access is indirect: using indices into data arrays enables efficient memory relocation (and simpler and more compact memory management) and *all* attributes of a vertex (edge, face) are identified by the same index. As a side effect, use of indices is platform compatible. More important in this context is the following observation: halfedges always come in pairs. Thus when we actually implement a halfedge data structure we group inverse halfedges pairwise in an array. This trick has two advantages: first, the opposite halfedge is given implicitly by an addition modulo two so there is no need to explicitly store it. Second, we obtain an explicit representation for "full" edges, which is important when we want to associate data with edges rather than halfedges. (Note that this is generally also possible with a pointer implementation.)

3.2 Directed Edges

The directed edges data structure [**?**] is a memory efficient variant of the halfedge data structure designed for triangles meshes. It has the following restrictions:

- Only triangle meshes can be represented.
- There is no explicit representation of edges.

The main benefit of directed edges is memory efficiency while they can represent all triangle meshes which can be represented by the general halfedge data structure. In addition some

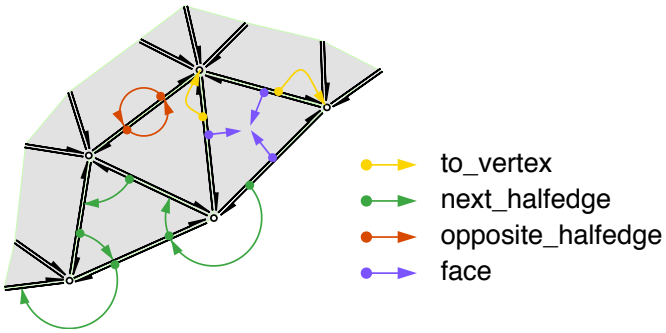


Figure 3.1: This figure shows the references stored with each halfedge. Note that the next_halfedge references enable traversing the boundary loop.

atomic operations are more efficient than for general halfedges. However, traversing boundary loops is more expensive as there is no atomic operation to enumerate the next boundary edge.

The directed edges data structure is based on indices as references to each element (vertex, face, halfedge). The indexing is not arbitrary but follows certain rules that *implicitly* encode some of the connectivity information of the triangle mesh. Instead of pairing opposite halfedges (see above), this data structure groups the three halfedges belonging to a common triangle. To be more precise, let f be the index of a face, then the indices of its three halfedges are given as

$$\mathsf{halfedge}(f, i) = 3f + i, \quad i = 0, 1, 2$$

Now let h be the index of a halfedge. Then the index of its adjacent face and its index within that face are simply given by

$$\mathrm{face}(h) = h/3$$

Not surprisingly, we can also compute the index of h's next halfedge as $(h + 1) \mod 3$. The remaining parts of the connectivity have to be stored explicitly in arrays. Thus for each vertex we store the index of an outgoing halfedge. For each halfedge, we store the index of its opposite halfedge and the index of the vertex, the halfedge points to.

Notes

- The directed edge data structure handles boundaries by special (e.g., negative) indices indicating that the inverse edge is invalid. This leads to a non-uniform treatment of the connectivity encoding and some special cases.
- We have described the directed edges data structure for pure triangle meshes. An adaption to pure quad meshes is straightforward. However, it is not possible to mix triangles and quads, which severely limits this extension to regular settings.

3.3 Mesh Libraries: CGAL and OpenMesh

Although the description of a halfedge data structure is straightforward, its implementation is not. Programming a basic mesh data structure might thus be a good exercise for an undergraduate course in geometric modeling, but designing and implementing a full-fledged mesh library

that is memory- and time-efficient, robust and easy to use and that is possibly equipped with a number of standard operations and algorithms is an advanced and time consuming task. Among others the following issues have to be taken into account:

- *Access:* How can we conveniently access vertices, edges and faces? How can we conveniently enumerate neighborhoods or navigate along mesh boundaries?
- *Modification:* How can a mesh be modified by the user? How can vertices and faces be added or deleted? How can we guarantee that after a modification the data structure is still consistent?
- *Composed operations:* How can high level operations like halfedge-collapses, face-splits etc. be implemented efficiently?
- *Parameterization:* How can arbitrary additional data efficiently be with the vertices, edges and faces of the mesh? What kind of memory management is efficient?
- *Input and output:* How to read and write data from different file formats? How to build up a halfedge-structure from an indexed face set?

Taking all these issues into account and coping with the often subtle problems when modifying the data structure, we strongly recommend to use one of full featured, publicly available mesh libraries. We refer the interested programmer to the following C++ libraries.

CGAL, the Computational Geometry Algorithms Library, is a generic C++ library for geometric computing. It provides basic geometric primitives and operations, as well as a collection of standard data structures and geometric algorithms, including 3D polyhedral surfaces with a halfedge data structure and a rich set of 2D and 3D triangulations. CGAL is specifically designed to provide reliable solutions to efficiency and robustness issues which are of crucial importance in geometric algorithms. Robustness and scalability of the algorithms are achieved by isolating a minimal number of predicates and constructors, and by the use of templated kernels. The CGAL library is available at http://www.cgal.org.

OpenMesh provides efficient halfedge data structures for polygonal meshes, their input/output and several standard geometry processing algorithms. OpenMesh is available at http://www.openmesh.org.

Comparing objectives and functionalities of these two libraries, CGAL is much more ambitious. Its rich foundation of algorithms is strongly biased by computational geometry with focus on robust and exact algorithms. CGAL has a wide user base and a number of research institutions actively contribute to its development. A major difference in data structures is the support for tetrahedral meshes. In contrast, OpenMesh is highly specialized on efficient processing of surface meshes based solely on halfedge data structures. It takes over some concepts of CGAL which provided one of the first publicly available halfedge data structures. It is much more focused on requirements of modeling with polygonal meshes and provides a set of standard geometry processing algorithms, like mesh smoothing, decimation, etc. We note that both libraries have different licensing policies.

As some authors of this tutorial were actively involved in the design and implementation of OpenMesh, we will describe this library in more detail here. Note that the same functionality is available in CGAL, however, the code reads differently.

- *Access:* Vertices, edges, halfedges and faces are all explicitly represented in OpenMesh and can easily be accessed through iterators or through handles (which replace indices as references). OpenMesh also provides so-called circulators that allow to enumerate the neighborhoods of each element. The following example shows how to compute the barycenter of the 1-ring of each vertex in a mesh:

```
TriangleMesh mymesh;
(...) // Read a mesh
// A VertexIter is an STL-compliant iterator to enumerate all vertices of a mesh
for ( VertexIter vi = mymesh.vertices_begin(); vi != mymesh.vertices_end(); ++vi )
{
   int cnt = 0;
   Point cog(0,0,0);
   // A VertexVertexIter is a circulator that enumerates the 1-ring of a vertex
   for ( VertexVertexIter vvi = mymesh.vv_iter(vi); vvi; ++vvi )
   {
      cnt += 1;
      cog += mymesh.point(vvi);
   }
   cog /= cnt;
   // Now cog equals the center of gravity of vi's neighbors
}
```

- *Modification:* OpenMesh provides functions to add and remove vertices and faces to and from a mesh. These operations are guaranteed to preserve a consistent state of the mesh. The following example shows how to add a triangle to a mesh:

```
TriangleMesh mymesh;
// Add three vertices to the mesh
VertexHandle v0 = mymesh.add_vertex( Point( 0, 0, 0 ) );
VertexHandle v1 = mymesh.add_vertex( Point( 0, 1, 0 ) );
VertexHandle v2 = mymesh.add_vertex( Point( 3, 0, 2 ) );
// Connect the vertices by a triangle
FaceHandle f = mymesh.add_face( v0, v1, v2 );
// Remove the face
mymesh.delete_face( f );
```

- *Composed operations:* OpenMesh provides a number of high-level operations, among them halfedge-collapse, vertex-split, face-split, edge-split and edge-flip. It also provides functions that test whether a certain operation is legal or not. The following snippet of code tries to collapse all edges that are shorter than a given threshold:

```
TriangleMesh mymesh;
(...)
for ( HalfedgeIter hi = mymesh.halfedges_begin(); hi != mymesh.halfedges_end(); ++hi )
   if ( ! mymesh.status( hi ).is_deleted() )
   {
      Point a = mymesh.point( mymesh.from_vertex_handle( hi ) );
      Point b = mymesh.point( mymesh.to_vertex_handle( hi ) );
```

```
        if ( (b-a).norm() < epsilon && mymesh.is_collapse_ok( hi ) )
            mymesh.collapse( hi );
    }
mymesh.garbage_collection();
```

- *Parameterization:* Arbitrary additional data can be associated with the vertices, edges, halfedges or faces of a mesh via OpenMesh's property mechanism. This mechanism allows to assign and remove data from the mesh at runtime. Thus it is for example possible to temporarily assign to each edge a weight:

```
TriangleMesh mymesh;
(...)
// Add a property (in this case a float) to each edge of mymesh
EdgePropertyHandle< float > weight;
mymesh.add_property(weight);
// Assign values to the properties
for ( EdgeIter ei = mymesh.edges_begin(); ei != mymesh.edges_end(); ++ei )
    mymesh.propery( weight, ei ) = some_value;
(...)
// Do something with the properties
for ( EdgeIter ei = mymesh.edges_begin(); ei != mymesh.edges_end(); ++ei )
    do_something_with( mymesh.propery( weight, ei ) );
(...)
// If the weights are not needed anymore, remove them to free some memory
mymesh.remove_property(weight);
```

- *Input and output:* OpenMesh reads and writes `stl` (ASCII and Binary), `off` and `obj` files. Handlers for other file types can easily be added by the user.

```
TriangleMesh mymesh;
read_mesh( mymesh, "a_filename.off" );
(...)
write_mesh( mymesh, "another_filename.stl" );
```

- *Standard algorithms:* OpenMesh provides a set of standard algorithms that can easily be customized to different needs. Among these algorithms are: smoothing (Chapter **??**), decimation (Chapter **??**) and subdivision (see also Chapter **??**).

3.4 Summary

Efficient data structures are crucial for geometry processing based on polygonal meshes. We recommend halfedge data structures (or directed edges as a special case for triangle meshes), for which full-featured and publicly available implementations already exist, e.g., CGAL or OpenMesh.

4 Model Repair

In short, model repair is the task of removing artifacts from a geometric model to produce an output model that is suitable for further processing by downstream applications that have certain quality requirements on their input. Although this definition is most often too general, it nonetheless captures the essence of model repair: the definition of what we mean by a "model", of what exactly constitutes an "artifact" and what is meant by "suitable for further processing" is highly dependent on the problem at hand and there generally is no single algorithm which is be applicable in all situations.

Model repair is a necessity in a wide range of applications. As an example, consider the design cycle in automotive CAD/CAE/CAM: Car models are typically manually designed in CAD systems that use trimmed NURBS surfaces as the underlying data structure for representing geometry. However, downstream applications like numerical fluid simulations cannot handle NURBS patches but need a watertight, manifold triangle mesh as input. Thus there is a need for an intermediate stage that converts the NURBS model into a triangle mesh. Unfortunately, this conversion process often produces artifacts that cannot be handled by downstream applications. Thus, the converted model has to be repaired — often in a manual and tedious post-process.

The goal of this tutorial is to give a practical view on the typical types of artifacts that occur in geometric models and to introduce the most common algorithms that address these artifacts. After giving a short overview on the common types of artifacts in Section **??**, we start out in Section **??** by classifying repair algorithms on whether they *explicitly* identify and resolve artifacts or on whether they rely on an intermediate *volumetric* representation that automatically enforces certain consistency constraints. This classification already gives a hint on the strengths and weaknesses of a particular algorithm and on the quality that can be expected from its output. In Section **??** we then give an overview on the different types of input models that are encountered in practice. We describe the specific artifacts and problems of each model and explain their origin. We also give references to algorithms that are designed to resolve these artifacts. Finally, we present some of the common model repair algorithms in more detail in Section **??**. We give a short description on how each algorithm works and to which models it is applicable. We hope that this provides a deeper understanding of the often subtle problems that occur in model repair and of ways to address these problems. Some of these algorithms are relatively straightforward, while others are more involved such that we can only show their basic mechanisms.

4.1 Artifact Chart

The chart in Fig. **??** shows the most common types of artifacts that occur in typical input models. Note that this chart is by no means complete and in particular in CAD models one encounters further artifacts like self-intersecting curves, points that do not lie on their defining planes and so on. While some of these artifacts, e.g., complex edges, have a precise meaning, others, like the distinction between small scale and large scale overlaps, are described intuitively rather than by strict definitions.

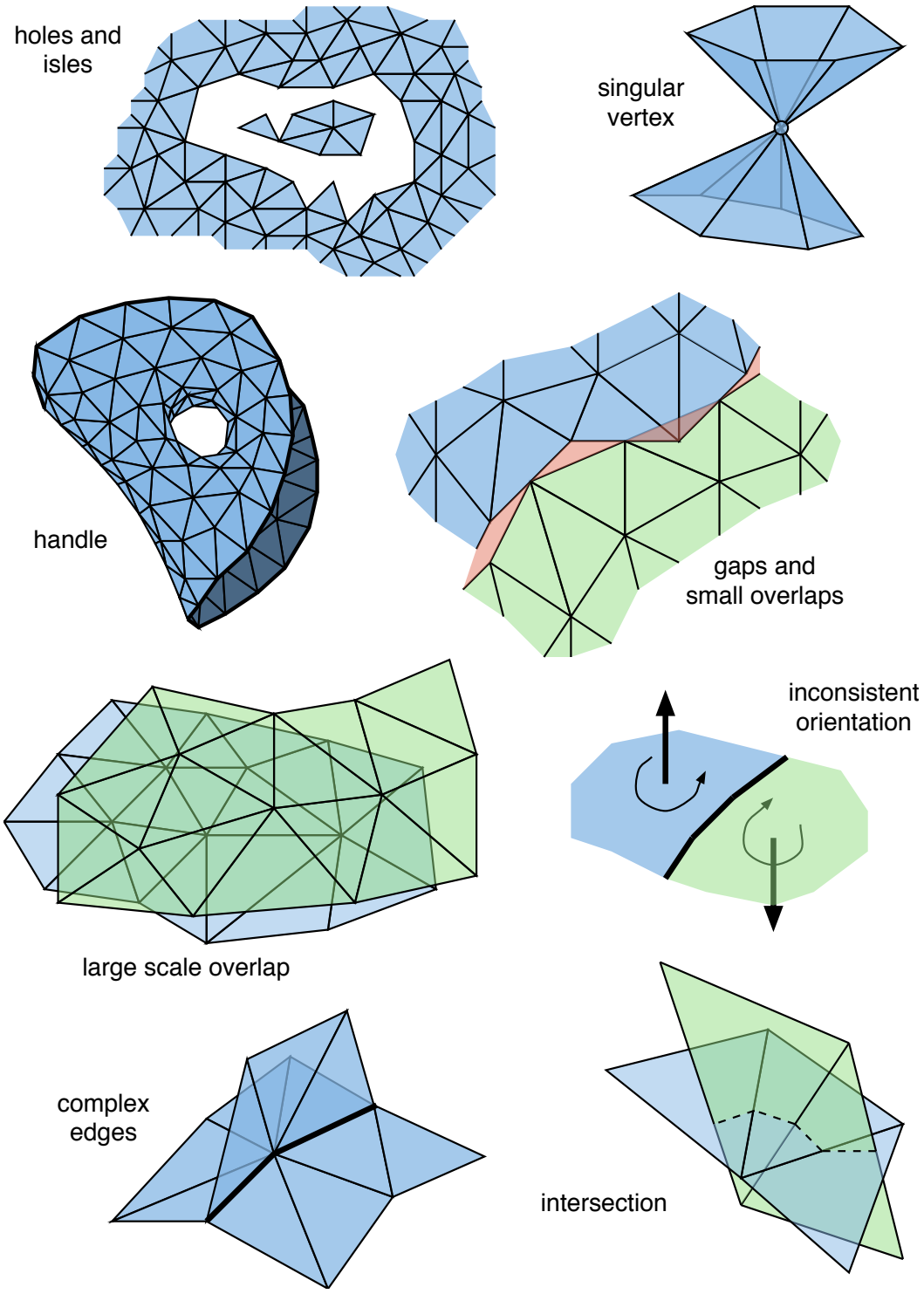


Figure 4.1: Artifact chart

4.2 Types of Repair Algorithms

Most model repair algorithms can roughly be classified as being either *surface oriented* or *volumetric*. Understanding these concepts already helps to evaluate the strengths and weaknesses of a given algorithm and the quality that can be expected of its output.

Surface oriented algorithms operate directly on the input data and try to explicitly identify and resolve artifacts on the surface. For example, gaps could be removed by snapping boundary elements (vertices and edges) onto each other or by stitching triangle strips in between the gap. Holes can be closed by a triangulation that minimizes a certain error term. Intersections could be located and resolved by explicitly splitting edges and triangles.

Surface oriented repair algorithms only minimally perturb the input model and are able to preserve the model structure in areas that are away from artifacts. In particular, structure that is encoded in the connectivity of the input (e.g. curvature lines) or material properties that are associated with triangles or vertices are usually well preserved. Furthermore, these algorithms introduce only a limited number of additional triangles.

To guarantee a valid output, surface oriented repair algorithms usually require that the input model already satisfies certain quality requirements (error tolerances). These requirements cannot be guaranteed or even be checked automatically, so these algorithms are rarely fully automatic but need user interaction and manual post-processing. Furthermore, due to numerical inaccuracies, certain types of artifacts (like intersections or large overlaps) cannot be resolved robustly. Other artifacts, like gaps between two closed connected components of the input model that are geometrically close to each other, cannot even be identified.

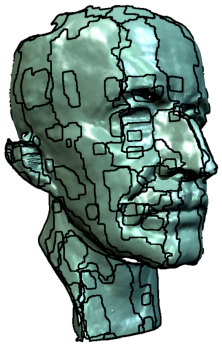
Volumetric algorithms convert the input model into an intermediate volumetric representation from which the output model is then extracted. Here, a volumetric representation is any kind of partitioning of space into cells such that each cell can be classified as either being *inside* or *outside*. Examples of volumetric representations that have been used in model repair include regular Cartesian grids, adaptive octrees, *kd*-trees, BSP-trees and Delaunay triangulations, see also Chapter **??**. The interface between inside and outside cells then defines the topology and the geometry of the reconstructed model. Due to their very nature, volumetric representations do not allow for artifacts like intersections, holes, gaps or overlaps or inconsistent normal orientations. Depending on the type of the extraction algorithm, one can often also guarantee the absence of complex edges and singular vertices. Handles, however, might still be present in the reconstruction.

Volumetric algorithms are typically fully automatic and produce watertight models (Section **??**). Depending on the type of volume, they can often be implemented very robustly. In particular, the discrete neighborhood relation of cells allows to reliably extract a consistent topology of the restored model. Furthermore, well-known morphological operators can be used to robustly remove handles from the volume.

On the downside, the conversion to and from a volume leads to a resampling of the model. It often introduces aliasing artifacts, loss of model features and destroys any structure that might have been present in the connectivity of the input model. The number of triangles in the output of a volumetric algorithm is usually much higher than that of the input model and thus has to be decimated in a post-processing step. Also the quality of the output triangles often degrades and has to be improved afterwards (see also Fig. **??**). Finally, volumetric representations are quite memory intensive so it is hard to run them at high resolutions.

4.3 Types of Input

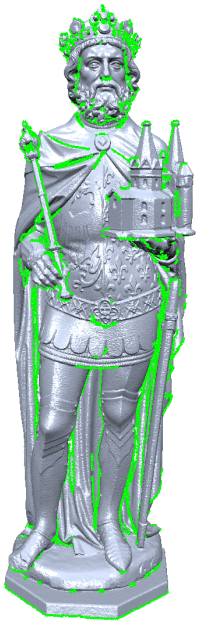
In this section we list the most common types of input models that occur in practice. For each type we describe its typical artifacts (see also Section **??**) and give references to algorithms that can be used to remove them.



Registered Range Scans are a set of patches (usually triangle meshes) that represent overlapping parts of the surface $\mathcal{S}$ of a scanned object. While large overlaps are a distinct advantage in registering the scans, they pose severe problems when these patches are to be fused into a single consistent triangle mesh. The main geometric problem in this setup are the potentially very large overlaps of the scans such that a point $\mathbf{x}$ on $\mathcal{S}$ is often described by multiple patches that do not necessarily agree on $\mathbf{x}$'s position. Furthermore, each patch has its own connectivity that is usually not compatible to the connectivity of the other patches. This is in particular a problem for surface oriented repair algorithms.

There are only a few surface oriented algorithms for fusing range images, e.g., Turk et al.'s mesh zippering algorithm [**?**]. The most well-known volumetric method is due to Curless and Levoy [**?**].

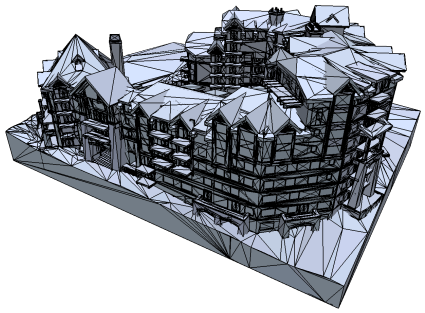
Fused Range Scans Fused range images are manifold meshes with boundaries, i.e., holes and isles. These artifacts are either due to obstructions in the line of sight of the scanner or result from bad surface properties of the scanned model such as transparency or glossiness. The goal is to identify and fill these holes. In the simplest case, the filling is a patch that minimizes some bending energy and joins smoothly to the boundary of the hole. Advanced algorithms synthesize new geometric detail that resembles the detail that is present in a local neighborhood of the hole or transplant geometry from other parts of the model in order to increase the realism of the reconstruction. The main obstacles in hole filling are the incorporation of isles into the reconstruction and the avoidance of self-intersections.



Kliencsek proposes an algorithm based on dynamic programming for finding minimum weight triangulations of planar polygons [**?**]. This algorithm is a key ingredient in a number of other model repair algorithms. Liepa proposes a surface oriented method to smoothly fill holes such that the vertex densities around the hole are interpolated [**?**]. Podolak et al. cast hole filling as a graph-cut problem and report an algorithm that is guaranteed to produce non-intersecting patches [**?**]. Davis et al. propose a volumetric method that diffuses a signed distance function into empty regions of the volume [**?**]. Pauly et al. use a database of geometric priors from which they select shapes to fill in regions of missing data [**?**].

Triangle Soups are mere sets of triangles with no or only little connectivity information. They most often arise in CAD models that are manually created in a boundary representation where users typically assemble predefined elements (taken from a library) without bothering about consistency constraints. Due to the manual layout, these models typically are made of only a

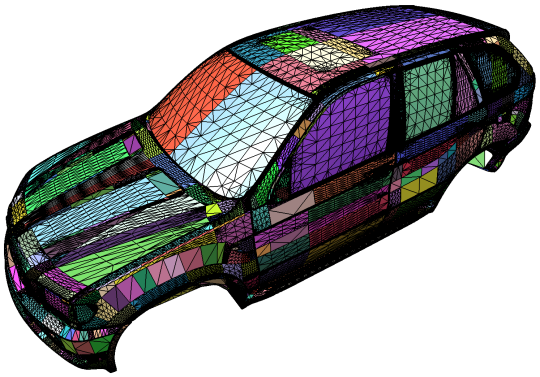
few thousands of triangles, but they may contain all kinds of artifacts. Thus triangle soups are well suited for visualization, but cannot be used in most downstream application.



Intersecting triangles are one of the most common type of artifact in triangle soups, as the detection and in particular the resolution of intersecting geometry would be much too time-consuming and numerically unstable. Complex edges and singular vertices are often intentionally created in order to avoid the duplication of vertices and the subsequent need to keep these duplicate vertices consistent. Other artifacts include inconsistent normal orientations, small gaps and excess interior geometry.

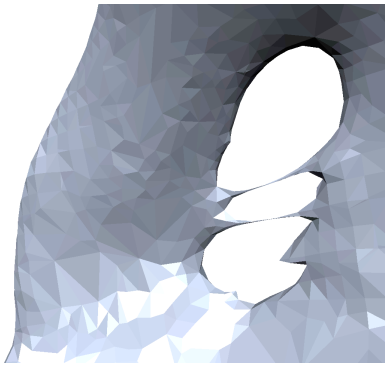
Surface oriented methods that are able to automatically repair triangle soups are not known. However, there are a number of volumetric methods that can be applied to triangle soups: Murali et al. produce a BSP tree from the triangle soup and automatically compute for each leaf a solidity [**?**]. Nooruddin et al. use ray-casting and filtering to convert the triangle soup into a volumetric representation from which they then extract a consistent, watertight model [**?**]. Shen et al. create an implicit representation by generalizing the moving least squares approach from point sets to triangle soups [**?**]. Bischoff and Kobbelt scan convert the soup into a binary grid, use morphological operators to determine inside/outside information and then invoke a feature-sensitive extraction algorithm [**?**]. Gress and Klein use a *kd*-tree to improve the geometric fidelity of the volumetric reconstruction [**?**].

Tringulated NURBS Patches typically are a set of triangle patches that contain gaps and small overlaps. These artifacts arise when triangulating two or more trimmed NURBS patches that join at a common boundary curve. Usually, each patch is triangulated separately, thus the common boundary is sampled differently from each side. Other artifacts present in such models include intersecting patches and inconsistent normal orientations. Triangulated NURBS patches are usually repaired using surface oriented methods. These methods first try to establish a consistent orientation of the input patches. Then they identify corresponding parts of the boundary and snap these parts onto each other. Thus any structure that might be present in the triangulation (like iso-lines, curvature lines, etc.) is preserved.



Barequet and Sharir use a geometric hashing technique to identify and bridge boundary parts that have a similar shape [**?**]. Barequet and Kumar describe an algorithm that identifies geometrically close edges and snaps them onto each other [**?**]. Borodin and Klein generalize the vertex-contraction operator to a vertex-edge contraction operator and thus are able to progressively close gaps [**?**]. Bischoff and Kobbelt use a volumetric repair method locally around the

artifacts and stitch the resulting patches into the remaining mesh [**?**]. Borodin et al. propose an algorithm to consistently orient the normals which takes visibility information into account [**?**].

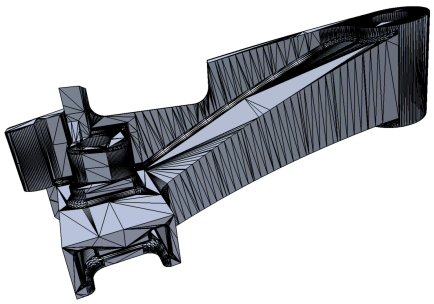


Contoured Meshes are meshes that have been extracted from a volumetric dataset by Marching Cubes, Dual Contouring or other extraction algorithms. Provided that the correct triangulation look-up tables are used, contoured meshes are always guaranteed to be watertight and manifold (Section **??**). However, these meshes often contain topological artifacts, such as small handles.

Volumetric data arises most often in medical imaging (CT, MRI,...), as an intermediate representation when fusing registered range scans or in constructive solid geometry (CSG). In a volumetric dataset, each voxel is classified as being either inside or outside the object. Unfortunately, due to the finite resolution of the underlying grid, voxels are often classified wrongly (so-called partial volume effect). This leads to topological artifacts in the reconstruction, like handles, holes, or disconnected components, that are not consistent with the model that should be represented by the volume. A famous example are MRI datasets of the brain: It is well known that the surface of the brain is homeomorphic to a sphere, but all too often a model of higher genus is extracted.

While disconnected components and small holes can easily be detected and removed from the main part of the model, handles are more problematic. Due to the simple connectivity of the underlying Cartesian grid, it is usually easiest to remove them from the volume dataset before applying the contouring algorithm or to identify and resolve them during reconstruction [**?**]. Guskov and Wood presented one of the few surface oriented algorithms to remove handles from an input mesh [**?**].

Badly Meshed Manifolds contain degenerate elements like triangles with zero area, caps, needles and triangle flips. These meshes result from the tessellation of CAD models or are the output of marching cubes like algorithms, in particular if they are enhanced by feature-preserving techniques. Although badly meshed manifolds are in fact manifold and even often watertight, the degenerate shape of the elements prevents further processing, e.g., in finite element meshers, and leads to instabilities in numerical simulations. The repair of such meshes is called *remeshing,* and we discuss this issue in depth in Chapter **??**.



4.4 Surface Oriented Algorithms

In this section we describe some of the most common surface oriented repair algorithms. These algorithms work directly on the input surface and try to remove artifacts by explicitly modifying the geometry and the connectivity of the input.

4.4.1 Consistent Normal Orientation

Consistently orientating the normals of an input model is part of most surface oriented repair algorithms and can even improve the performance of volumetric algorithms. Usually the orientation of the normals is propagated along a minimum spanning tree between neighboring patches either in a preprocessing step or implicitly during traversal of the input. Borodin et al. describe a more sophisticated algorithm that additionally takes visibility information into account [**?**].

The input is a set of arbitrarily oriented polygons. In a preprocessing phase the polygons are assembled into larger, manifold patches (possibly with boundary) as described in Section **??**. The algorithm then builds up a connectivity graph of neighboring patches where the label of each edge encodes the *normal coherence* of the two patches. Furthermore, for each side of each patch a *visibility coefficient* is computed that describes how much of the patch is visible when viewed from the outside. Finally, a globally consistent orientation is computed by a greedy optimization algorithm: If the coherence of two patches is high, normal consistency is favoured over front-face visibility and vice versa.

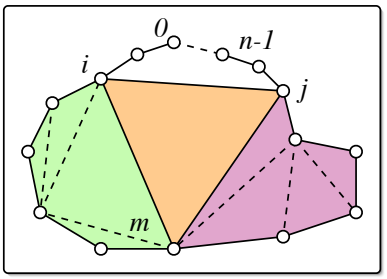
4.4.2 Surface Based Hole Filling

In this section we describe an algorithm for computing a fair triangulation of a hole. The algorithm was proposed by Liepa [**?**] and builds on work of Klincsek [**?**] and Barequet and Sharir [**?**]. It is a basic building block of many other repair algorithms.

The goal is to produce a triangulation of a polygon $\mathbf{p}_0, \ldots, \mathbf{p}_{n-1}$ that minimizes some given weight function. In the context of mesh repair, this weight function typically measures the fairness of the triangulation, e.g., its area or the variation of the triangle normals (see also Chapter **??**).

Let $\phi(i, j, k)$ be a weight function that is defined on the set of all triangles $(\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k)$ that could possibly appear during construction of the triangulation and let $w_{i,j}$ be the minimum total weight that can be achieved in triangulating the polygon $\mathbf{p}_i, \ldots, \mathbf{p}_j$, $0 \leq i < j < n$. Then $w_{i,j}$ can be computed recursively as

$$w_{i,j} = \min_{i<m<j} w_{i,m} + w_{m,j} + \phi(i, m, j) \ .$$



The triangulation that minimizes $w_{0,n-1}$ is computed by a dynamic programming algorithm that caches the intermediate values $w_{i,j}$.

Liepa suggests a weight function ϕ that is designed to take into account the dihedral angles between neighboring triangles as well as triangle area. It produces tuples

$$\phi(i, j, k) = (\alpha, A) \ ,$$

where α is the maximum of the dihedral angles to the neighbors of $(\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k)$ and A is its area. Note that this weight function in particular penalizes fold-overs. When comparing different values of ω, a low normal variation is favored over a low area:

$$(\alpha_1, A_1) < (\alpha_2, A_2) \quad :\Leftrightarrow \quad (\alpha_1 < \alpha_2) \vee (\alpha_1 = \alpha_2 \wedge A_1 < A_2)$$

Note that when evaluating ω one has to take into account that the neighboring triangles can either belong to the mesh that surrounds the hole or to the patch that is currently being created. A triangulation of a hole that is produced using this weight function is shown in Fig. **??**.

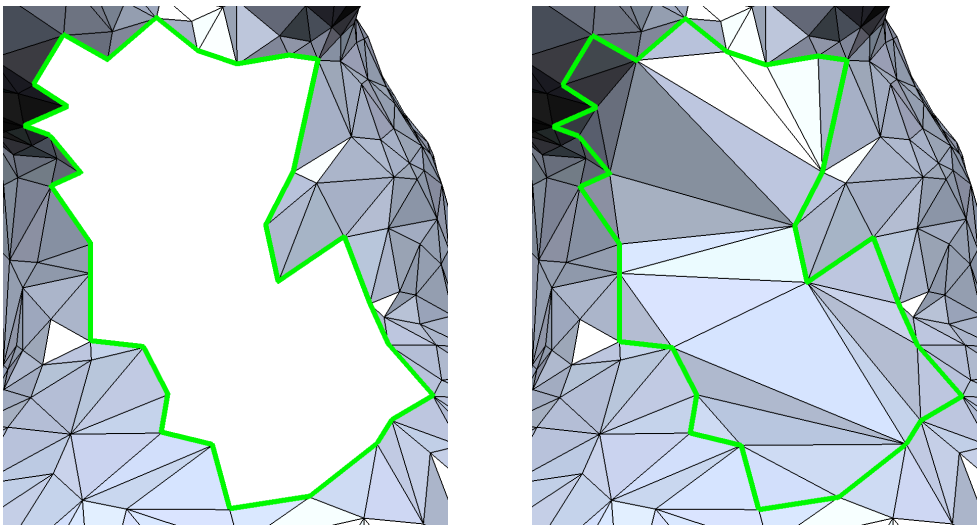


Figure 4.2: A hole triangulation that minimizes normal variation and total area.

To produce a fair hole filling, Liepa suggests to produce a tangent continuous fill-in of minimal thin plate energy: First the holes are identified and filled by a coarse triangulation as described above. These patches are then refined such that their vertex densities and edge lengths match that of the area surrounding the holes, see Chapter **??**. Finally, the patch is smoothed such as to blend with the geometry of the surrounding mesh, see Chapter **??**.

Discussion The algorithm reliably closes holes in models with smooth patches. The density of the vertices matches that of the surrounding surface, see Fig. **??**. The complexity of building the initial triangulation is $O(n^3)$, which is sufficient for most holes that occur in practice. However, the algorithm does not check or avoid self intersections and does not detect or incorporate isles into the filling.

4.4.3 Conversion to Manifolds

Gueziec et al. propose a method to remove complex edges and singular vertices from non-manifold input models [**?**]. The output is guaranteed to be a manifold triangle mesh, possibly with boundaries. As the algorithm operates solely on the connectivity of the input model, it does not suffer from numerical robustness issues. In a preprocessing phase all complex edges and singular vertices are identified. The input is then cut along these complex edges into manifold patches (usually with boundaries). Finally, pairs of matching edges (i.e., edges that have the same endpoints) are identified and – if possible – merged.

In the preprocessing phase the input is split into separate faces and all complex edges are identified by counting the number of adjacent faces: edges with one, two, or more than two adjacent faces are boundary, regular interior or complex respectively. Then the input model is

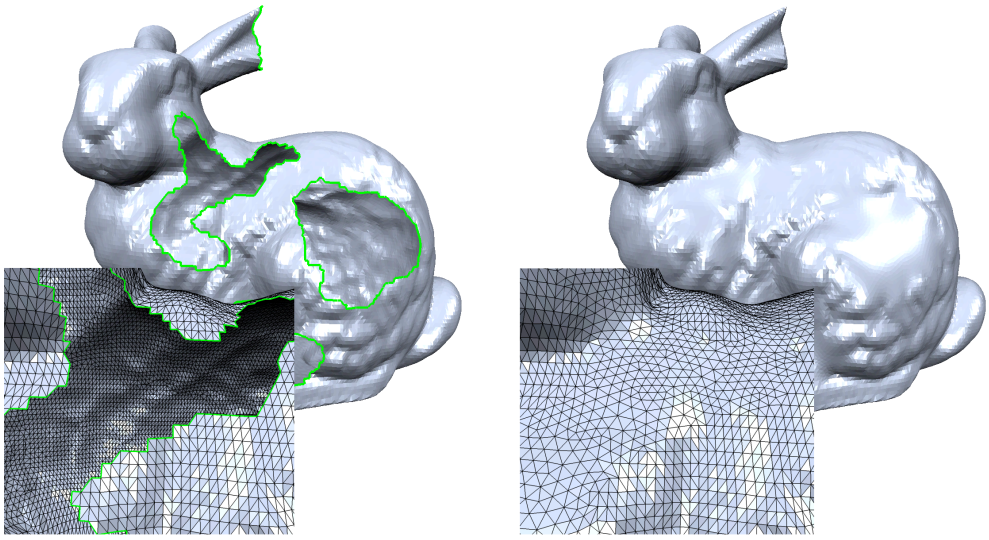


Figure 4.3: Liepa's hole filling algorithm. Note that the point density of the fill-in matches that of the surrounding area.

separated into manifold patches along the complex edges by stitching the two adjacent faces of each *interior regular* edge. This method implicitly handles stand-alone and singular vertices.

Gueziec et al. propose two different strategies for stitching further edges: *pinching* and *snapping*. The pinching strategy only stitches along edges that belong to the same connected component. Thus small erroneous connected components are separated from the main part of the model and can be easily detected and removed in a post-processing step. The algorithm iterates once over all boundary vertices. Let v be a boundary vertex, v_p its predecessor and v_n its successor along the boundary. If $v_p = v_n$ the two edges (v_p, v) and (v, v_n) are merged.

In contrast to pinching, the snapping strategy reduces the number of connected components of the model. The basic idea is to locate candidate pairs of boundary edges and to stitch them if a certain stitchability criterion is met. This criterion asserts that after stitching, the model does not contain new complex edges or singular vertices. The snapping strategy can be extended to also allow the stitching of edges that are geometrically close to each other.

Discussion The scope of this algorithm is limited to the removal of complex edges and singular vertices. This, however, is done efficiently and robustly.

4.4.4 Gap Closing

A number of surface oriented algorithms have been proposed to close the gaps and small overlaps that are typical for triangulated NURBS models.

Barequet and Sharir proposed one of the first algorithms to fill gaps and remove small overlaps [**?**]. The algorithm identifies matching parts of the boundaries by a geometric hashing technique and fills the gaps by patching them with triangle strips or by the technique presented in Section **??**.

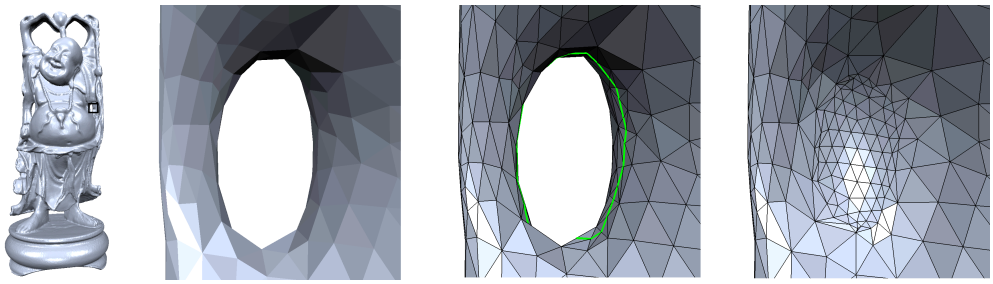


Figure 4.4: Left and middle left: The Happy Buddha model contains more than 100 handles. Middle right: A non-separating closed cycle along a handle. Right: The handle was removed by cutting along the non-separating cycle and closing the holes with triangle patches.

Barequet and Kumar propse an algorithm to repair CAD models that identifies and merges pairs of boundary edges [**?**]. For each pair of boundary edges the area between the two edges normalized by the edge lengths is computed. This score measures the geometric error that would be introduced by merging the two edges. Pairs of boundary edges are then iteratively merged in order of increasing score.

Borodin et al. [**?**] propose an algorithm that snaps boundary vertices to nearby boundary edges. The algorithm is based on a standard mesh-decimation technique, but replaces the vertex-vertex contraction operator by a vertex-edge contraction operator, that operates on boundary vertices v and boundary edges e: Let c be the closest point to v on e. If c is an interior point of e, c is inserted into e by splitting the adjacent triangle in two. Finally, v and c are merged. The cost of a vertex-edge collapse is defined as the distance of v to c. The algorithm maintains a priority queue of vertex/edge pairs and snaps them in order of increasing distance.

Discussion The semantics of these surface oriented algorithms is well defined and they are typically easy to implement. If the input data is well-behaved and the user parameters are chosen in accordance with the error that was accepted during triangulation, they also produce satisfying results. However, there are no guarantees on the quality of the output. Due to the simple heuristics, many artifacts remain unresolved. Therefore, these algorithms are usually run in an interactive loop that allows designers to override the decisions made by the algorithms or to steer the algorithms in a certain direction.

4.4.5 Topology Simplification

Guskov and Wood proposed an algorithm that detects and resolves all handles up to a given size ε in a manifold triangle mesh [**?**]. Handles are removed by cutting the input along a non-separating closed path and sealing the two resulting holes by triangle patches, see Fig. **??**.

Given a seed triangle s, the algorithm conquers a geodesic region $\mathcal{R}_\varepsilon(s)$ around s in the order that is given by Dijkstra's algorithm on the dual graph of the input mesh $\mathcal{M}$. Note that Dijkstra's algorithm not only computes the length of a shortest path from each triangle t to the seed s, but it also produces a parent $p(t)$ such that $t, p(t), p^2(t), \ldots, s$ actually *is* a shortest path from t to s.

The boundary of $\mathcal{R}_\varepsilon(s)$ consists of one or more boundary loops. Whenever a boundary loop touches itself along an edge, it is split into two new loops and the algorithm proceeds. However, when two different loops touch along a common edge, a handle is detected. Let t_1 and t_2 be the

two triangles that are adjacent to the common edge and $p^{n_1}(t_1) = p^{n_2}(t_2)$ a common ancestor of t_1 and t_2. The closed path

$$p^{n_1}(t_1), \ldots, p(t_1), t_1, t_2, p(t_2), \ldots, p^{n_2}(t_2)$$

is then a cycle of adjacent triangles that stretches around the handle. The input model is cut along this triangle strip and the two boundary loops that are created by this cut are then sealed, e.g., by the method presented in Section **??**.

To detect all handles of $\mathcal{M}$, one has to perform the region growing for all triangles $s \in \mathcal{M}$. Guskov and Wood describe a method to considerably reduce the necessary number of seed triangles and thus are able to significantly speed up the algorithm.

Discussion The proposed method reliably detects small handles up to a user-prescribed size and removes them. However, the algorithm is slow, it does not detect long, thin handles and it cannot guarantee that no self-intersections are created when a handle is removed.

4.5 Volumetric Repair Algorithms

This section presents recent repair algorithms that use an intermediate volumetric representation to implicitly remove the artifacts of a model. This volumetric representation might be as simple as a regular Cartesian grid or as complex as a binary space partition.

4.5.1 Volumetric Repair on Regular Grids

Nooruddin and Turk proposed one of the first volumetric techniques to repair arbitrary models that contain gaps, overlaps and intersections [**?**]. Additionally they employed morphological operators to resolve topological artifacts like holes and handles.

First, the model is converted into a Cartesian voxel grid: A set of projection directions $\{\mathbf{d}_i\}$ is produced, e.g., by subdividing an octahedron or icosahedron. Then the model is projected along these directions onto an orthogonal planar grid. For each grid point $\mathbf{x}$, the algorithm records the first and last intersection point of the ray $\mathbf{x} + \lambda\mathbf{d}_i$ and the input model. A voxel is classified by such a ray to be *inside*, if it lies between these two extreme depth samples, otherwise it is classified as *outside*. The final classification of each voxel is derived from the majority vote of all the rays passing through that voxel. A Marching Cubes algorithm is then used to extract the surface between inside and outside voxels.

In an optional second step, thin handles and holes are removed from the volume by applying *morphological operators* that are also known from image processing [**?**]. The *dilation* operator d_ε computes the distance from each outside voxel to the inside component. All voxels that are within a distance of ε to the inside are also set to *inside*. Thus the dilation operator closes small handles and bridges small gaps. The *erosion* operator e_ε works exactly the other way round and removes thin bridges and handles. Usually, dilation and erosion are used in conjunction, $e_\varepsilon \circ d_\varepsilon$ to avoid expansion or shrinkage of the model.

Discussion The classification of inside and outside voxels is rather heuristic and often not reliable. Furthermore, the algorithm is not feature-sensitive.

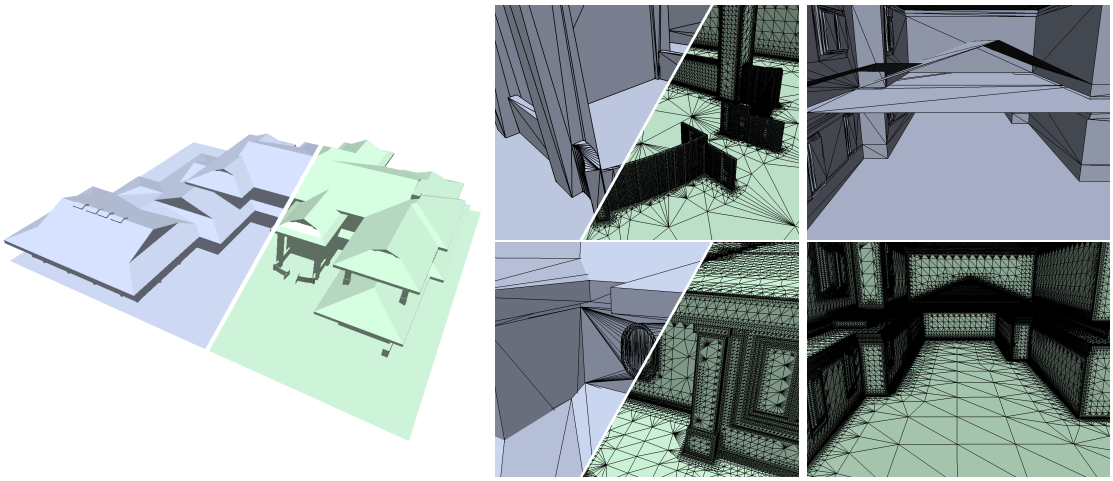


Figure 4.5: Reconstruction (green) of a triangle soup (blue). Left: Visually there is no difference between the triangle soup and the reconstruction. Middle: The reconstruction is a watertight mesh that is refined near the model features. Right: The volumetric approach allows to reliably detect and remove excess interior geometry from the input.

4.5.2 Volumetric Repair on Adaptive Grids

Bischoff et al. [**?**] propose an improved volumetric technique to repair arbitrary triangle soups. The user provides an error tolerance ε and a maximum diameter ρ up to which gaps should be closed. The algorithm first creates an adaptive octree representation of the input model where each cell stores the triangles intersecting with it. From these triangles a feature-sensitive sample point can be computed for each cell. Then a sequence of morphological operations is applied to the octree to determine the topology of the model. Finally, the connectivity and geometry of the reconstruction are derived from the octree structure and samples, respectively.

Let us assume that the triangle soup is scaled to fit into the root cell of the octree. We set the maximum depth of the octree cells such that the diameter of the finest level cells is smaller than ε. Each cell stores references to the triangles that intersect it and initially all triangles are associated with the root cell. Then cells that are not yet on maximum depth are recursively split if they either contain a boundary edge or if the triangles within the cell deviate too much from a common supporting plane. Whenever a cell is split, its triangles are distributed to its children. The result is a memory-efficient octree with large cells in planar or empty regions and fine cells along the features and boundaries of the input model (see Fig. **??**).

In the second phase, each leaf cell of the octree is classified as being either *inside* or *outside*. First, all cells that contain a boundary of the model are dilated by $n := \rho/\varepsilon$ layers of voxels such that all gaps of diameter $\leq \rho$ are closed. A flood fill algorithm then propagates the outside label from the boundary of the octree into its interior. Finally, the outside component is dilated again by n layers to avoid an expansion of the model.

A Dual Contouring algorithm then reconstructs the interface between the outside and the inside cells by connecting sample points. These sample points are the minimizers of the squared distances to their supporting triangle planes, thus features like edges and corners are well preserved (see also Chapter **??** on quadric error metrics). If no such planes are available (e.g., because

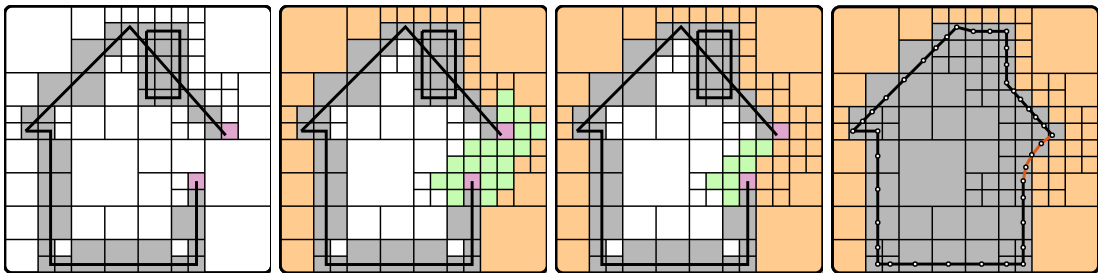


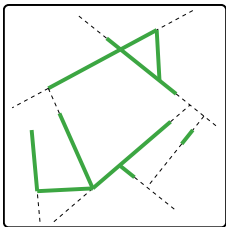
Figure 4.6: Left: Adaptive octree, boundary cells are marked red. Center left: Dilated boundary (green) and outside component (orange). Center right: Outside component dilated back into the boundary cells. Right: Final reconstruction

the cell was one of the dilated boundary cells), the corresponding sample point is smoothed in a post-processing step (Chapter **??**).

Discussion As this algorithm is based on a volumetric representation, it produces guaranteed manifold output (Fig. **??**). Features are also well preserved. However, despite the adaptive octree, the resolution of the reconstruction is limited.

4.5.3 Volumetric Repair with BSP Trees

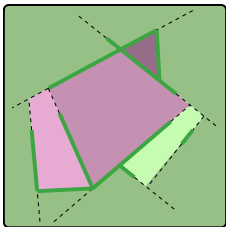
A unique method for converting triangle soups to manifold surfaces was presented by Murali and Funkhouser [**?**]. The polygon soup is first converted into a BSP tree, the supporting planes of the input polygons serve as splitting planes for the space partition. The leaves of the tree thus correspond to closed convex spatial regions C_i. For each C_i a *solidity coefficient* $s_i \in [-1, 1]$ is computed. Negative solidity coefficients designate empty regions, while positive coefficients designate solid regions.



BSP

All unbounded cells naturally lie outside the object and thus are assigned a solidity value of -1. Let C_i be a bounded cell and let $\mathcal{N}(i)$ be the indices of all its face neighbors. Thus for each $j \in \mathcal{N}(i)$ the intersection $P_{ij} = C_i \cap C_j$ is a planar polygon that might be partially covered by the input geometry. For each $j \in \mathcal{N}(i)$ let t_{ij} be the transparent area, o_{ij} the opaque area and a_{ij} the total area of P_{ij}. The solidity s_i is then related to the solidities s_j of its face neighbors by

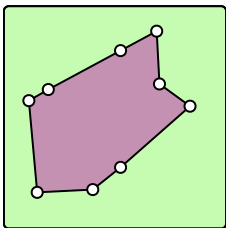
$$s_i = \frac{1}{A_i} \sum_{j \in \mathcal{N}(i)} (t_{ij} - o_{ij}) s_j \ , \tag{4.1}$$



solidity coefficients

where $A_i = \sum a_{ij}$ is the total area of the boundary of C_i. Note the two extreme cases: If P_{ij} is fully transparent, $t_{ij} - o_{ij} = a_{ij} > 0$ the correlation of s_i and s_j is positive, indicating that both cells should be solid or both cells should be empty. If, on the other hand, P_{ij} is fully opaque, $t_{ij} - o_{ij} = -a_{ij} < 0$, the negative correlation indicates that one cell should be solid and the other empty. Collecting all equations Eq. (**??**) leads to a sparse linear system

$$\mathbf{M}[s_1, \ldots, s_n]^T = \mathbf{b} \ ,$$



reconstruction

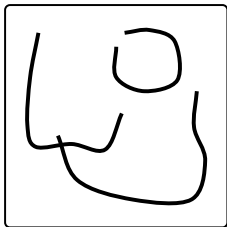
which can be solved efficiently using an iterative solver (Chapter **??**). It can be shown that $\mathbf{M}$ is always invertible and that the solidity coefficients of the solution in fact lie in the range $[-1, 1]$.

Finally, the surface of the solid cells is extracted by enumerating all neighboring pairs of leaf cells (C_i, C_j). If one of them is empty and the other is solid, the corresponding (triangulated) boundary polygon P_{ij} is added to the reconstruction.

Discussion This method does not need (but also cannot incorporate) any user parameters to automatically produce watertight models. The output might contain complex edges and singular vertices, but these can be removed using the algorithm presented in Section **??**. Unfortunately, a robust and efficient computation of the combinatoric structure of the BSP is hard to accomplish.

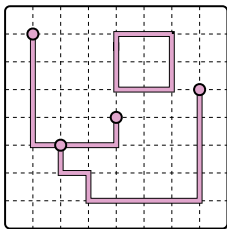
4.5.4 Volumetric Repair on the Dual Grid

Ju proposes an interesting volumetric algorithm to repair arbitrary triangle soups [**?**]. While the boundary loops are explicitly traced and filled, the overall scheme is volumetric.



Input

The algorithm first approximates the input model by a subset $\mathcal{F}$ of the faces of a Cartesian grid. For memory efficiency, these faces are stored in an adaptive octree. Additionally, a sample point (and possibly a normal) from the input model are associated with each face, to allow for a more accurate reconstruction. The boundary $\partial\mathcal{F}$ of $\mathcal{F}$ is defined to be the subset of the grid edges that are incident to an odd number of faces in $\mathcal{F}$. Note that if $\mathcal{G}$ is another face set, such that $\partial\mathcal{G} = \partial\mathcal{F}$, then $\partial(\mathcal{F} \ominus \mathcal{G}) = \emptyset$. Here, the *symmetric difference* (xor) of two sets $\mathcal{A}$ and $\mathcal{B}$ is defined as $\mathcal{A} \ominus \mathcal{B} = (\mathcal{A} \cup \mathcal{B}) \setminus (\mathcal{A} \cap \mathcal{B})$. Also, if $\partial\mathcal{F} = \emptyset$ then the grid voxels can be two-colored by *inside* and *outside* labels such that two adjacent voxels have the same label, while two voxels that are separated by a face of $\mathcal{F}$ have different labels.

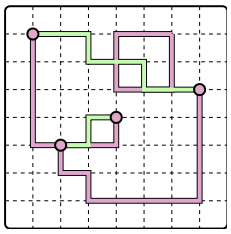


Face set

For each boundary loop B_i of $\mathcal{F}$, the algorithm constructs a minimal face set $\mathcal{G}_i$ such that $\partial\mathcal{G}_i = B_i$. Then $\mathcal{F}$ is replaced by

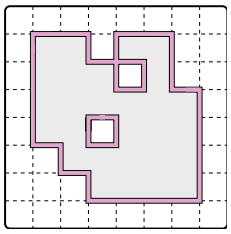
$$\mathcal{F}' = \mathcal{F} \ominus \mathcal{G}_1 \ominus \cdots \ominus \mathcal{G}_n,$$

thus $\partial\mathcal{F}' = \emptyset$. As voxels at the corners of the bounding box are known to be *outside*, they are used as seeds for propagating the *inside/outside* information over the grid. The interface between *inside* and *outside* voxels is then extracted using either a Marching Cubes or a Dual Contouring algorithm.



Patches

Discussion Ju's algorithm uses a volumetric representation and thus produces guaranteed manifold output. The algorithm is memory-less, i.e., insensitive to the size of the input and thus can process arbitrarily large meshes out-of-core. On the other hand, the algorithm has problems handling thin structures. In particular, if the discrete approximation that is used in the hole filling step overlaps with the input geometry, this part of the mesh may disappear or be shattered into many pieces. Due to the volumetric representation the whole input model is resampled and the output might become arbitrarily large for fine resolutions.



Reconstruction

4.5.5 Extending MLS to Triangle Soups

Shen et al. propose a volumetric repair algorithm that operates on arbitrary triangle soups [**?**]. It is a generalization of the moving least squares approach that can for instance be used for reconstructing geometry from point clouds. Instead of approximating positional information only, they also incorporate normal constraints into the reconstruction and thus avoid an oscillating solution. The details of this algorithm are involved and we restrict ourselves to the basic ideas.

Let $t_1, \ldots, t_N$ be a set of triangles and let $\mathbf{n}_1, \ldots, \mathbf{n}_N$ be their normals. The goal is to generate a function $s : \mathbb{R}^3 \to \mathbb{R}$ whose zero level-set matches $t_1, \ldots, t_N$ as close as possible. An arbitrary contouring algorithm can then be used to extract a reconstruction of $t_1, \ldots, t_N$ from s. For a single triangle t_k, the corresponding function s_k is of course linear

$$s_k(\mathbf{x}) = \mathbf{n}_k^T(\mathbf{x} - \mathbf{q}_k)$$

where $\mathbf{q}_k$ is an arbitrary point on t_k.

The function s is expressed as a linear combination of a set $\mathbf{b}(\mathbf{x}) = [b_1(\mathbf{x}), \ldots, b_M(\mathbf{x})]^T$ of basis functions, thus

$$s(\mathbf{x}) = \mathbf{b}(\mathbf{x})^T\mathbf{c} \tag{4.2}$$

for some vector $\mathbf{c}$. So-called *radial basis functions* are a common choice for $\mathbf{b}(\mathbf{x})$, but they lead to a large linear system that is hard to solve efficiently. Instead, Chen et al. follow an approach that is known as *Moving Least Squares (MLS),* see [**?**] and references therein.

The idea of MLS is to only use a very limited set of basis functions, typically $\mathbf{b}(x, y, z) = [1, x, y, z]^T$. To compensate the limited degrees of freedom in choosing a small number of basis functions, Eq. (**??**) is made dependent on the point $\mathbf{x}_0$ at which one plans to evaluate and triangles that are close to $\mathbf{x}_0$ are given a greater weight than those that are far away. Thus, for a fixed point $\mathbf{x}_0$ one seeks to minimize

$$\sum_k \int_{t_k} w_{\mathbf{x}_0}(\mathbf{x})^2 \left(\mathbf{b}(\mathbf{x})^T\mathbf{c}_{\mathbf{x}_0} - s_k(\mathbf{x})\right)^2 d\mathbf{x} \tag{4.3}$$

with respect to $\mathbf{c}_{\mathbf{x}_0}$ where the weight function $w_{\mathbf{x}_0}(\mathbf{x})$ is chosen as

$$w_{\mathbf{x}_0}(\mathbf{x}) = \frac{1}{||\mathbf{x} - \mathbf{x}_0||^2 + \varepsilon^2}$$

Setting the derivative of Eq. (**??**) w.r.t. $\mathbf{c}_{\mathbf{x}_0}$ to zero leads to a 4×4 linear system

$$\sum_k \mathbf{A}_k\mathbf{c}_{\mathbf{x}_0} = \sum_k \mathbf{a}_k$$

where

$$\mathbf{A}_k = \int_{t_k} w_{\mathbf{x}_0}(\mathbf{x})^2\mathbf{b}(\mathbf{x})\mathbf{b}(\mathbf{x})^T d\mathbf{x} \quad \text{and} \quad \mathbf{a}_k = \int_{t_k} w_{\mathbf{x}_0}(\mathbf{x})^2\mathbf{b}(\mathbf{x})^T s_k(\mathbf{x}) d\mathbf{x}$$

Thus, the function s is given as $s(\mathbf{x}_0) = \mathbf{b}(\mathbf{x}_0)^T\mathbf{c}_{\mathbf{x}_0}$.

The integrands that appear in $\mathbf{A}_k$ and $\mathbf{a}_k$ are rational polynomials and Chen et al. devise a suitable numerical integration scheme to evaluate them. They also propose a method to speed up the evaluation.

Discussion This algorithm produces watertight models and automatically bridges gaps in an intuitive way. The method can be modified to produce hulls of a different geometric complexities that enclose the input model. These hulls can then be used, e.g., for fast collision detection tests. Unfortunately, the algorithm does not cope well with models that contain interior excess geometry.

5 Discrete Curvatures

This section introduces differential properties of 2-manifold surfaces and discusses the corresponding approximations on arbitrary triangle meshes. These discrete differential operators play a central role in many mesh processing applications such as surface smoothing (Chapter **??**), parameterization (Chapter **??**), or mesh deformation (Chapter **??**).

5.1 Differential Geometry

We provide a brief review of important concepts from differential geometry that form the basis of the definition of the discrete operators on triangle meshes. For an in-depth discussion we refer to standard textbooks such as [**?**].

Let a continuous surface $\mathcal{S} \subset \mathbb{R}^3$ be given in parametric form as

$$\mathbf{x}(u,v) = \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix}, \ (u,v) \in \mathbb{R}^2,$$

where x, y, z are (sufficiently often) differentiable functions in u and v. The partial derivatives $\mathbf{x}_u$ and $\mathbf{x}_v$ span the tangent plane to $\mathcal{S}$ at $\mathbf{x}$. Assuming a regular parameterization, i.e., $\mathbf{x}_u \times \mathbf{x}_v \neq \mathbf{0}$, the normal vector is given as $\mathbf{n} = (\mathbf{x}_u \times \mathbf{x}_v)/\|\mathbf{x}_u \times \mathbf{x}_v\|$.

The *first fundamental form* of $\mathbf{x}$ is given by the matrix

$$\mathbf{I} = \begin{bmatrix} E & F \\ F & G \end{bmatrix} := \begin{bmatrix} \mathbf{x}_u^T\mathbf{x}_u & \mathbf{x}_u^T\mathbf{x}_v \\ \mathbf{x}_u^T\mathbf{x}_v & \mathbf{x}_v^T\mathbf{x}_v \end{bmatrix}, \tag{5.1}$$

which defines an inner product on the tangent space of $\mathcal{S}$. The corresponding arc element ds is given as

$$ds^2 = Edu^2 + 2Fdudv + Gdv^2.$$

The area element can be derived as

$$dA = \sqrt{EG - F^2}dudv.$$

The *second fundamental form* is defined as

$$\mathbf{II} = \begin{bmatrix} e & f \\ f & g \end{bmatrix} := \begin{bmatrix} \mathbf{x}_{uu}^T\mathbf{n} & \mathbf{x}_{uv}^T\mathbf{n} \\ \mathbf{x}_{uv}^T\mathbf{n} & \mathbf{x}_{vv}^T\mathbf{n} \end{bmatrix}. \tag{5.2}$$

Alternatively, $\mathbf{II}$ can be expressed using the identities $\mathbf{x}_{uu}^T\mathbf{n} = -\mathbf{x}_u^T\mathbf{n}_u$, $\mathbf{x}_{uv}^T\mathbf{n} = \mathbf{x}_{vu}^T\mathbf{n} = -\frac{1}{2}(\mathbf{x}_u^T\mathbf{n}_v + \mathbf{x}_v^T\mathbf{n}_u)$, and $\mathbf{x}_{vv}^T\mathbf{n} = -\mathbf{x}_v^T\mathbf{n}_v$.

The symmetric bilinear first and second fundamental forms allow to measure length, angles, area, and curvatures on the surface.

Let $\mathbf{t} = a\mathbf{x}_u + b\mathbf{x}_v$ be a unit vector in the tangent plane at $\mathbf{p}$, represented as $\bar{\mathbf{t}} = (a, b)^T$ in the local coordinate system. The *normal curvature* $\kappa_n(\bar{\mathbf{t}})$ is the curvature of the planar curve that results from intersecting $\mathcal{S}$ with the plane through $\mathbf{p}$ spanned by $\mathbf{n}$ and $\mathbf{t}$. The normal curvature in direction $\bar{\mathbf{t}}$ can be expressed in terms of the fundamental forms as

$$\kappa_n(\bar{\mathbf{t}}) \;=\; \frac{\bar{\mathbf{t}}^T \mathbf{II}\, \bar{\mathbf{t}}}{\bar{\mathbf{t}}^T \mathbf{I}\, \bar{\mathbf{t}}} \;=\; \frac{ea^2 + 2fab + gb^2}{Ea^2 + 2Fab + Gb^2}$$

The minimal normal curvature κ_1 and the maximal normal curvature κ_2 are called *principal curvatures*. The associated tangent vectors $\mathbf{t}_1$ and $\mathbf{t}_2$ are called *principal directions* and are always perpendicular to each other.

The principal curvatures are also obtained as eigenvalues of the *Weingarten curvature matrix* (or second fundamental tensor)

$$\mathbf{W} \;:=\; \frac{1}{EG - F^2} \begin{bmatrix} eG - fF & fG - gF \\ fE - eF & gE - fF \end{bmatrix} . \tag{5.3}$$

$\mathbf{W}$ represents the Weingarten map or shape operator, which measures the directional derivative of the normal, i.e. $\mathbf{W}\bar{\mathbf{t}} = \dfrac{\partial}{\partial \bar{\mathbf{t}}}\mathbf{n}$. This allows the normal curvature to be expressed as

$$\kappa_n(\bar{\mathbf{t}}) = \bar{\mathbf{t}}^T \mathbf{W} \bar{\mathbf{t}} \; .$$

With a local coordinate system defined by the principal directions $\mathbf{t}_1$ and $\mathbf{t}_2$, $\mathbf{W}$ is a diagonal matrix, or in general

$$\mathbf{W} \;=\; \begin{bmatrix} \bar{\mathbf{t}}_1 & \bar{\mathbf{t}}_2 \end{bmatrix} \begin{bmatrix} \kappa_1 & 0 \\ 0 & \kappa_2 \end{bmatrix} \begin{bmatrix} \bar{\mathbf{t}}_1 & \bar{\mathbf{t}}_2 \end{bmatrix}^{-1} . \tag{5.4}$$

Then the normal curvature can also be written as

$$\kappa_n(\bar{\mathbf{t}}) = \kappa_n(\phi) = \kappa_1 \cos^2\phi + \kappa_2 \sin^2\phi, \tag{5.5}$$

where ϕ is the angle between $\bar{\mathbf{t}}$ and $\bar{\mathbf{t}}_1$ (Euler's theorem).

The *curvature tensor* $\mathbf{T}$ is expressed as a symmetric 3×3 matrix with the eigenvalues κ_1, κ_2, 0 and the corresponding eigenvectors $\mathbf{t}_1$, $\mathbf{t}_2$, $\mathbf{n}$. The tensor $\mathbf{T}$ measures the change of the unit normal with respect to a tangent vector $\mathbf{t}$ independently of the parameterization. It can be constructed as

$$\mathbf{T} \;=\; \mathbf{P}\mathbf{D}\mathbf{P}^{-1} \; ,$$

with $\mathbf{P} = [\mathbf{t}_1, \mathbf{t}_2, \mathbf{n}]$ and $\mathbf{D} = \mathrm{diag}(\kappa_1, \kappa_2, 0)$.

The *Gaussian curvature* K is defined as the product of the principal curvatures, i.e.,

$$K = \kappa_1 \kappa_2 = \det(\mathbf{W}), \tag{5.6}$$

the *mean curvature* H as the average of the principal curvatures, i.e.,

$$H = \frac{\kappa_1 + \kappa_2}{2} = \frac{1}{2}\mathrm{trace}(\mathbf{W}). \tag{5.7}$$

The mean curvature can alternatively be expressed as the (continuous) average of the normal curvatures

$$H \;=\; \frac{1}{2\pi} \int_0^{2\pi} \kappa_n(\phi) d\phi \; . \tag{5.8}$$

In differential geometry, properties that only depend on the first fundamental form are called *intrinsic*. Intuitively, the intrinsic geometry of a surface can be perceived by 2D creatures that live on the surface without knowledge of the third dimension. Examples include length and angles of curves on the surface. Gauss' famous Theorema Egregium states that the Gaussian curvature is invariant under local isometries and as such also intrinsic to the surface [**?**]. Note that the term "intrinsic" is often also used to denote independence of a particular parametrization.

Laplace Operator. The following sections will make extensive use of the *Laplace operator* Δ, resp., the *Laplace-Beltrami operator* $\Delta_{\mathcal{S}}$. In general, the Laplace operator is defined as the divergence of the gradient, i.e. $\Delta = \nabla^2 = \nabla \cdot \nabla$. In Euclidean space this second order differential operator can be written as the sum of second partial derivatives

$$\Delta f = \mathrm{div} \nabla f = \sum_i \frac{\partial^2 f}{\partial x_i^2} \tag{5.9}$$

with Cartesian coordinates x_i. The *Laplace-Beltrami operator* extends this concept to functions defined on surfaces. For a given function f defined on a manifold surface $\mathcal{S}$ the Laplace-Beltrami is defined as

$$\Delta_{\mathcal{S}} f = \mathrm{div}_{\mathcal{S}} \, \nabla_{\mathcal{S}} f,$$

which requires a suitable definition of the divergence and gradient operators on manifolds (see [**?**] for details). Applied to the coordinate function $\mathbf{x}$ of the surface the Laplace-Beltrami operator evaluates to the mean curvature normal

$$\Delta_{\mathcal{S}} \, \mathbf{x} = -2H\mathbf{n}.$$

Note that the Laplace-Beltrami operator is an intrinsic property that only depends on the metric tensor of the surface and is thus independent of a specific parameterization.

5.2 Discrete Differential Operators

The differential properties defined in the previous section require a surface to be sufficiently often differentiable, e.g., the definition of the curvature tensor requires the existence of second derivatives. Since polygonal meshes are piecewise linear surfaces, the concepts introduced above cannot be applied directly. The following definitions of discrete differential operators are thus based on the assumption that meshes can be interpreted as piecewise linear approximations of smooth surfaces. The goal is then to compute approximations of the differential properties of this underlying surface directly from the mesh data. Different approaches have been proposed in recent years and we will provide a brief overview and comparison of the different techniques. For details we refer to the references given throughout the text and to the survey [**?**].

The general idea of the techniques described below is to compute discrete differential properties as spatial averages over a local neighborhood $\mathcal{N}(\mathbf{x})$ of a point $\mathbf{x}$ on the mesh. Often $\mathbf{x}$ coincides with a mesh vertex v_i, and n-ring neighborhoods $\mathcal{N}_n(v)$ or local geodesic balls are used as the averaging domain. The size of the local neighborhood critically affects the stability and accuracy of the discrete operators. The bigger the neighborhoods the more smoothing is introduced by the averaging operation, which makes the computations more stable in the presence of noise. For clean data sets, small neighborhoods, e.g., one-rings, are typically preferable, as they more accurately capture fine-scale variations of differential properties.

In order to estimate the curvature tensor at a vertex, a certain neighborhood of this vertex is considered, typically its one-ring. A common approach is to first discretize the normal curvature along edges. Given an edge (v_i, v_j), vertex positions $\mathbf{p}_i$, $\mathbf{p}_j$, and the normal $\mathbf{n}_i$,

$$\kappa_{ij} \;=\; 2\frac{(\mathbf{p}_j - \mathbf{p}_i)\mathbf{n}_i}{\|\mathbf{p}_j - \mathbf{p}_i\|^2} \tag{5.10}$$

provides an approximation of the normal curvature at $\mathbf{p}_i$ in the tangent direction that results from projecting $\mathbf{p}_i$ and $\mathbf{p}_j$ into the tangent plane defined by $\mathbf{n}_i$. This expression can be interpreted geometrically as fitting the osculating circle interpolating $\mathbf{p}_i$ and $\mathbf{p}_j$ with normal $\mathbf{n}_i$ at $\mathbf{p}_i$ (cf. [**?**]). Alternatively, the equation can be derived from discretizing the curvature of a smooth planar curve (see [**?**]). With estimates κ_{ij} of the normal curvature for all edges incident to vertex v_i, Euler's theorem (**??**) can be applied to relate the κ_{ij} to the unknown principal curvatures (and principal directions). Then approximations to the principal curvatures can be obtained either directly as functions of the eigenvalues of a symmetric matrix ([**?**, **?**]) or from solving a least-squares problem ([**?**, **?**]). Alternatively, [**?**] apply the trapezoid rule to get a discrete approximation of (**??**), which provides the mean curvature H, the Gaussian curvature K is obtained from a similar integral over κ_n^2, and the principal curvatures are then obtained from equations (**??**), (**??**). Exact quadrature formulas for curvature estimation are provided in [**?**].

A straightforward approach to estimating local surface properties uses a local higher-order reconstruction of the surface, followed by analytical evaluation of the desired properties on the reconstructed surface patch. Local surface patches, typically bivariate polynomials of low degree, are fitted to sample points [**?**, **?**, **?**] and possibly normals [**?**] within a local neighborhood. Special care is required to ensure good conditioning of the arising local least-squares problems which depend on local parameterization. A (rather expensive) global fitting of an implcit surface is applied in [**?**].

Taubin [**?**] proposed the uniform discretization of the Laplace-Beltrami operator

$$\Delta_{uni} f(v) \;:=\; \frac{1}{|\mathcal{N}_1(v)|} \sum_{v_i \in \mathcal{N}_1(v)} \left(f(v_i) - f(v)\right) \;\;, \tag{5.11}$$

where the sum is taken over all one-ring neighbors $v_i \in \mathcal{N}_1(v)$ (cf. Fig. **??**). This discretization does not take any local geometry of the domain mesh (edge lengths or angles) into account and hence cannot give a sufficient approximation for irregular tessellations. For example, when smoothing a planar (and hence perfectly smooth) triangulation, this operator may still shift vertices within the surface by moving each vertex to the barycenter of its neighbors. Although this leads to an improvement of the triangle shapes, it is a bad approximation to the Laplace-Beltrami of the surface (which should be parallel to the surface normal: $\Delta_{\mathcal{S}}\mathbf{p} = -2\,H\mathbf{n}$). A better (and the current standard) discretization was proposed in [**?**, **?**, **?**]:

$$\Delta_{\mathcal{S}} f(v) \;:=\; \frac{2}{A(v)} \sum_{v_i \in \mathcal{N}_1(v)} \left(\cot\alpha_i + \cot\beta_i\right)\left(f(v_i) - f(v)\right), \tag{5.12}$$

where $\alpha_i = \angle\left(\mathbf{p}(v), \mathbf{p}(v_{i-1}), \mathbf{p}(v_i)\right)$, $\beta_i = \angle\left(\mathbf{p}(v), \mathbf{p}(v_{i+1}), \mathbf{p}(v_i)\right)$, and $A(v)$ denotes the Voronoi area around the vertex v as shown in Fig. **??** (for an exact definition of the Voronoi region area see [**?**]). The same approach yields a discrete estimate for Gaussian curvature as

$$K(v) = \frac{1}{A(v)}\left(2\pi - \sum_{v_i \in \mathcal{N}_1(v)} \theta_i\right), \tag{5.13}$$

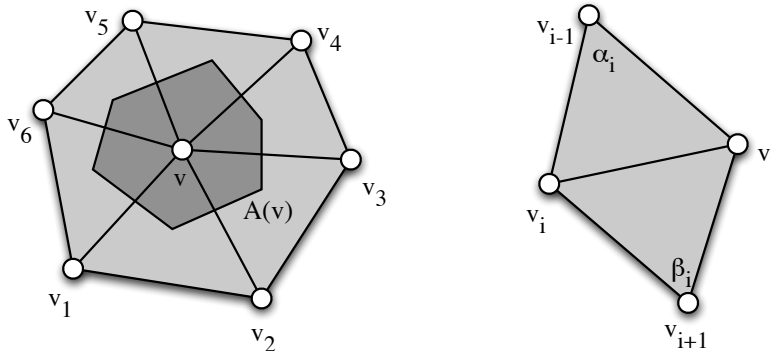


Figure 5.1: The Laplace-Beltrami $\Delta_{\mathcal{S}} f(v)$ of a vertex $v \in \mathcal{V}$ is computed by a linear combination of its function value $f(v)$ and those of its one-ring neighbors $f(v_i)$. The corresponding weights are given by the cotangent values of α_i and β_i and the Voronoi area $A(v)$.

where the angles of the incident triangles at vertex v are denoted by θ_i. This formula is a direct consequence of the Gauss-Bonnet theorem. Given the mean curvature normal as defined in (**??**) and the approximation of the Gaussian curvature of (**??**), the principal curvatures can be computed from (**??**) and (**??**) as

$$\kappa_{1,2}(v) = H(v) \pm \sqrt{H(v)^2 - K(v)}$$

where $H(v) = \frac{1}{2}\|\Delta_{\mathcal{S}}\mathbf{p}(v)\|$.

Eq. (**??**) is probably the most widely used discretization of the Laplace-Beltrami for triangle meshes and is typically applied for various geometry processing operations, such as surface smoothing (Chapter **??**), parameterization (Chapter **??**), and shape modeling (Chapter **??**). However, there are some disadvantages of the cotangent formula of (**??**):

- The cotangent weights $\omega_i = \cot\alpha_i + \cot\beta_i$ become negative if $\alpha_i + \beta_i > \pi$. This is well-known and can lead to flipped triangles in certain applications, e.g., when computing a parameterization (see Chapter **??**).
- The definition of the Laplace-Beltrami is not purely intrinsic, i.e., its evaluation can lead to different results even for two isometric surfaces, if their triangulation is different (see [**?**]).

The first point can possibly be fixed by using different weights. In [**?**] the positive mean value coordinates [**?**] are interpreted as an alternative, less accurate discretization of the Laplace-Beltrami operator where integration over the Voronoi area is replaced by integration over circle areas.

Bobenko and Springborn [**?**] propose an alternative definition that addresses these shortcomings for the case of piecewise flat surfaces, i.e., 2-dimensional manifolds that are equipped with a metric that is flat except at isolated points. The resulting formula is the same as (**??**), but with respect to an intrinsic Delaunay triangulation of the simplicial surface. For a piecewise flat surface, this triangulation is unique, which makes the evaluation of the discrete Laplace-Beltrami operator independent of the specific tessellation of the mesh. In addition, the Delaunay property guarantees positive weights by construction. Computing the intrinsic Laplace-Beltrami requires to first compute the restricted Delaunay triangulation using an edge flipping algorithm, which

is guaranteed to converge. Thus this approach is computationally more involved, in particular for applications that iteratively modify the vertex positions, e.g., curvature flow (Chapter **??**), where the re-tessellation is required after each time step.

Rusinkiewicz proposed a scheme that approximates the curvature tensor using finite differences of vertex normals [**?**]. As discussed above, the curvature tensor measures the change of the normal along the tangent directions. For a given triangle three such directions are given by the triangle edges. The change of normals along each of these edges can be approximated from the difference of the normals of the corresponding vertices. The resulting set of linear constraints on the elements of the curvature tensor can be used in a least-squares optimization to obtain a per-face estimate. The approximation of the curvature tensor for a vertex is then computed using weighted averaging of all per-face estimates of the one-ring based on an appropriate coordinate transformation as discussed in [**?**]. The paper also shows how this approach can be extended to higher order derivatives. Since the per-face estimates depend on vertex normals that are computed by standard weighted averaging of one-ring face normals, the averaging domain of this method is the two-ring neighborhood. As such, the results produced by this method are somewhat more stable for noisy data. The computation is efficient, however, since it can be performed using two passes over the one-rings of the mesh.

In [**?**] the piecewise linear surface is considered together with a piecewise linear normal field. Their discrete derivatives define the Weingarten map (**??**) and hence the tensor of curvature. The precomputed normal field replaces the second order derivatives, which are not available for piecewise linear functions. This idea is motivated by Phong-shading, and similarly the inherent inconsistencies lead to artifacts — the Weingarten matrix is not symmetric anymore — and hence approximation errors. However, [**?**] show convergence to curvatures of smooth surfaces and the errors are small enough to be competitive with other methods. The method yields a piecewise function for the curvature tensor which varies across faces as normals are interpolated. Gaussian and mean curvatures can be written as simple expressions of certain determinants. Evaluation is purely local and efficient, as curvature estimates at vertices are obtained by averaging.

Cohen-Steiner and Morvan [**?**] (see also [**?**] and [**?**]) propose a method for estimating the curvature tensor by averaging a line density of tensors defined on each edge of the mesh. This method is derived from the concept of normal cycles, which has been introduced to provide a unified way to define curvature for both smooth and polygonal surfaces. It includes a proof of convergence under certain sampling conditions based on measure theory. Intuitively, a curvature tensor can be defined for an edge by assigning a minimum curvature along the edge and a maximum curvature across the edge. Averaging over the local neighborhood region $\mathcal{N}(v)$ yields a simple summation formula over the edges intersecting $\mathcal{N}(v)$:

$$\mathbf{C}(v) = \frac{1}{|\mathcal{N}(v)|} \sum_{\mathbf{e} \in \mathcal{N}(v)} \beta(\mathbf{e}) \, \|\mathbf{e} \cap \mathcal{N}(v)\| \, \bar{\mathbf{e}} \, \bar{\mathbf{e}}^T,$$

where $|\mathcal{N}(v)|$ denotes the surface area of the local neighborhood around v, $\beta(\mathbf{e})$ is the signed dihedral angle between the normals of the two incident faces, $\|\mathbf{e} \cap \mathcal{N}(v)\|$ is the length of the part of the edge $\mathbf{e}$ that is contained in $\mathcal{N}(v)$, and $\bar{\mathbf{e}} = \mathbf{e}/\|\mathbf{e}\|$. The local neighborhood $\mathcal{N}(v)$ is typically chosen to be the one- or two-ring of the vertex v, but can also be computed as a local geodesic disk, i.e., all points on the mesh that are within a certain (geodesic) distance d from v. This can be more appropriate for non-uniformly tessellated surface, where the size of n-ring neighborhoods $\mathcal{N}_n(v)$ can vary significantly over the mesh. As noted in [**?**], tensor averaging can yield inaccurate results for low-valence vertices and small, e.g., one-ring, neighborhoods.

Wardetzky and collegues [**?**] classify the most common discrete Laplace operators according to a set of desirable properties derived from the smooth setting. They show that the discrete

operators cannot simultaneously satisfy all of the identified properties of symmetry, locality, linear precision and positivity. For example, the cotan formula of Eq. (**??**) satisfies the first three properties, but not the fourth, since edge weights can assume negative values. The choice of discretization thus depends on the specific application.

6 Mesh Quality

This section provides a brief overview of methods used to interactively evaluate the quality of triangle meshes. The techniques discussed here are adapted from smooth free-form surfaces (e.g. NURBS), and are mainly used to visualize surface quality in order to detect surface defects. Different applications may require different quality criteria. We distinguish between *smoothness* and *fairness*. While the former denotes the continuous differentiability (C^k) of a surface, e.g., C^2 for cubic splines, the latter is a more abstract concept required for high-quality surface design. Note that smoothness and fairness are not always used consistently. For example, surface smoothing typically denotes the process of improving the fairness of a surface (see Chapter **??**).

A surface may be smooth in a mathematical sense but still unsatisfactory from an aesthetical point of view. Fairness is an aesthetic measure of "well-shapedness" and therefore more difficult to define in technical terms than smoothness (distribution vs. variation of curvature) [**?**]. An important rule is the so called *principle of simplest shape* that is derived from fine arts. A surface is said to be well-shaped, if it is simple in design and free of unessential features. So a *fair* surface meets the mathematically defined goals (e.g. interpolation, continuity), while obeying this design principle. The most common measures for fairness are motivated by physical models like the strain energy of a thin plate

$$\int_{\mathcal{S}} \kappa_1^2 + \kappa_2^2 \, dA,$$

or are defined in terms of differential geometry, like the variation of curvature

$$\int_{\mathcal{S}} \left(\frac{\partial \kappa_1}{\partial \mathbf{t}_1} \right)^2 + \left(\frac{\partial \kappa_2}{\partial \mathbf{t}_2} \right)^2 dA,$$

with principal curvatures κ_i and principal directions $\mathbf{t}_i$ (see Chapter **??**). In general, some surface energy is defined that quantifies surface fairness, and curvature is used to express these terms as it is independent of the special parameterization of a surface. A fair surface is then designed by minimizing these energies (cf. Chapter **??**). Our current goal is not to improve, but to check surface quality, so we need to visualize these energies. Note that there are also different characterizations of fairness, such as aesthetical shape of isophotes/reflection lines [**?**].

Another important aspect of mesh quality is triangle shape. Some applications require "well shaped" triangles, e.g., simulations using *Finite Element Methods* (FEM). This requires constraints on shape parameters such as angles and area, which will also be discussed in Chapter **??**.

6.1 Visualizing smoothness

In order to interactively visualize surface quality, graphics hardware support should be exploited whenever possible. A given surface is tessellated into a set of triangles for rendering (in contrast to more involved rendering techniques like ray-tracing). Since a mesh can be interpreted as an accurate tessellation of, e.g., a set of NURBS patches, the same techniques for quality control can be used that are applied for smooth surfaces [**?**].

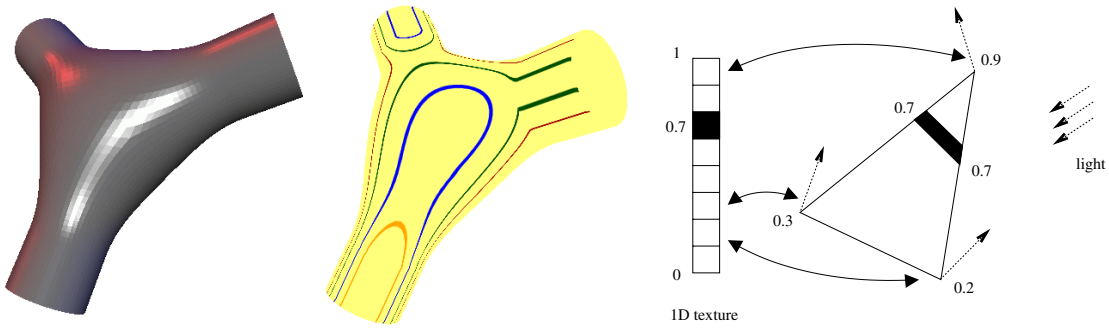


Figure 6.1: Isophotes. The center part of the surface was blended between the three tubes using C^1 boundary conditions. The discontinuities of the curvature at the joints are hard to detect from the flat shaded image (left), but clearly visualized by isophotes (middle) since C^1 blends cause C^0 isophotes. The right image sketches the rendering of isophotes with a 1D-texture: The illumination values are calculated for the vertices of a triangle from vertex normals and the light direction. These values are used as texture coordinates. The texel denoting the iso-value is colored black. Iso-lines are interpolated within the triangle.

Specular shading The simplest visualization technique is to use standard lighting and shading (Phong illumination model, flat- or Gouraud shading) as provided by the graphics subsystem. The local illumination of a vertex depends on the position of the light sources, on the surface normal, and on the view point/direction. This approach to surface interrogation is the most straightforward one, but it is difficult to find minor perturbations of a surface (cf. Fig. **??**, left).

Isophotes Isophotes are lines of constant illumination on a surface. For a Lambertian surface with purely diffuse reflection, isophotes are independent of the view point. When using a single, infinitely distant point light source, the illumination I_p of a surface point $\mathbf{p}$ is given by

$$I_p = \max\left\{\left\langle \mathbf{n} \middle| \mathbf{L} \right\rangle, 0\right\},$$

where $\mathbf{n}$ is the surface normal at $\mathbf{p}$ and $\mathbf{L}$ is the direction of light. Both vectors are normalized, so the value of I_p is in the interval $[0,1]$. Now some values $I_{c,j} \in [0,1] = \text{const}$ (e.g., $I_{c,j} = \frac{j}{n}$, $j = 0,\ldots,n$) are chosen and the isophotes/iso-curves $I = I_{c,j}$ are rendered.

The resulting image makes it easier to detect irregularities on the surface compared to standard shading. The user can visually trace the lines, rate their smoothness and transfer these observations to the surface: If the surface is C^k continuous then the isophotes are C^{k-1} continuous, since they depend on normals, i.e., on first derivatives (cf. Fig. **??**).

There are two main approaches to render iso-curves, such as isophotes: The first approach is to explicitly extract the curves or curve segments and then display them as lines. Here, in principle the same algorithms as for extracting iso-surfaces can be applied (Section **??**), reduced to the setting of extracting a curve on a surface.

The second approach takes advantage of the graphics hardware and allows direct rendering of isophotes from illumination values at the vertices of a triangle mesh: A one-dimensional texture is initialized with a default color C. Illumination values I_p are now treated as texture coordinates, and for the isophote values $I_{c,j}$ the corresponding texels are set to a color $C_j \neq C$. With

this setup the graphics subsystem will linearly interpolate the 1D texture within the triangles resulting in a rendered image of the isophotes (colors C_j) that are drawn onto the surface (color C) (cf. Fig. **??**). The 1D texture approach benefits more from the graphics hardware in contrast to explicitly calculating line segments. A drawback is that the width of the curves varies due to texture interpolation.

Reflection lines In contrast to isophotes, rendering of reflection lines assumes a specular surface. As a consequence reflection lines change when the point of view is modified and when the object is rotated or translated. The light source consists of a set of "light-lines" that are placed in 3-space space. Normally, the light-lines are parallel lines (cf. Fig. **??**).

Traditionally, reflection lines have been used in the process of designing cars. An arrangement of parallel fluorescent tubes is placed above the car model to survey the surface and its reflection properties.

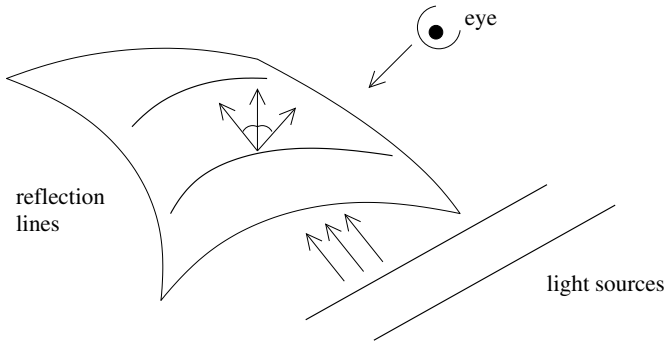


Figure 6.2: Reflection lines. The light source consists of parallel lines that are reflected by the surface. The reflection property requires that angles of incidence (light,normal) are equal to angles of emission (viewing direction,normal).

Under the assumption that the light source is infinitely far away from the object, *environment mapping* can be used to display reflection lines in real-time. A texture for environment mapping is generated once by ray-tracing the light sources over a sphere. The graphics subsystem will then automatically generate appropriate texture coordinates for every vertex depending on its relative position and normal.

Reflection lines are an effective and intuitive tool for surface interrogation. If the surface is C^k continuous then the reflection lines are C^{k-1} continuous. Just like isophotes, they can be efficiently rendered by taking advantage of graphics hardware and they are also sensitive to small surface perturbations. In addition, the concept that a real-world process is simulated makes their application very intuitive even for unexperienced users. Fig. **??** shows reflection lines for C^0, C^1 and C^2 surfaces.

6.2 Visualizing curvature and fairness

If fairness is expressed in terms of curvature, the techniques described in Chapter **??** can be used for visualization. Gaussian curvature $K = \kappa_1\kappa_2$ indicates the local shape of the surface (elliptic for $K > 0$, hyperbolic for $K < 0$ and parabolic for $K = 0 \wedge H \neq 0$ resp. flat for $K = 0 \wedge H = 0$).

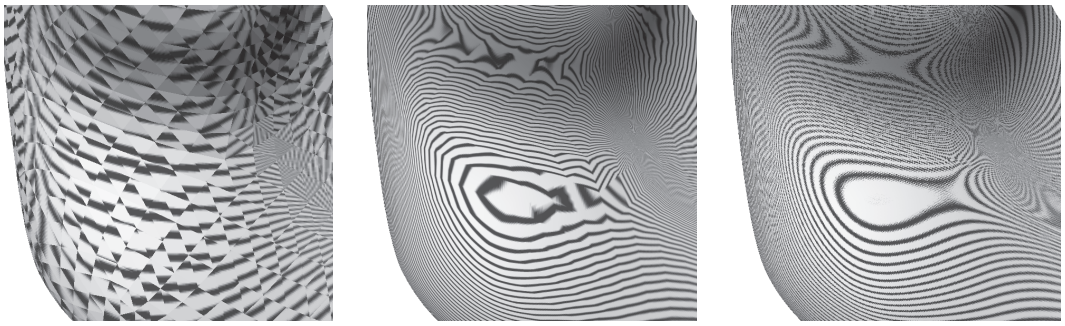


Figure 6.3: Reflection lines on C^0, C^1 and C^2 surfaces. One clearly sees that the differentiability of the reflection lines is one order lower, i.e., C^{-1}, C^0 and C^1 respectively.

A local change of the sign of K may denote a (even very small) perturbation of the surface. Additionally, mean curvature, principal curvatures, and total curvature $\kappa_1^2 + \kappa_2^2$ can be used. These scalar values are typically visualized using color-coding as shown in Fig. **??**

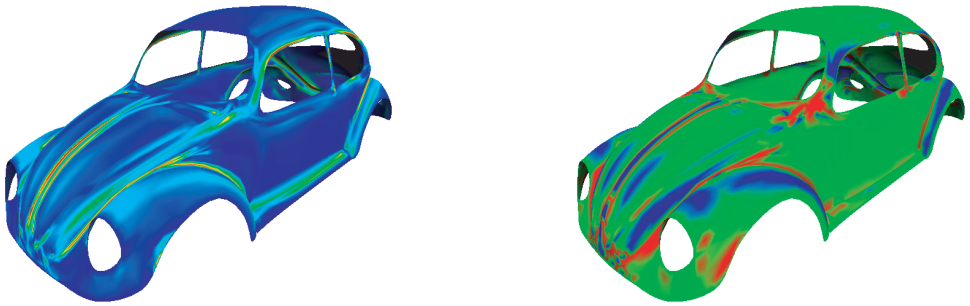


Figure 6.4: Color coding curvature values, mean curvature (left) and Gaussian curvature (right).

Iso-curvature lines Iso-curvature lines are lines of constant curvature on a surface. They can be displayed similarly to isophotes, where instead of illumination values, curvature values are used. If the surface is C^k continuous, then the iso-curvature lines are C^{k-2} continuous, so iso-curvature lines are even more sensitive to discontinuities than isophotes or reflection lines.

A problem when rendering iso-curvature lines with 1D-textures may be a wide range of curvature values that may not map appropriately to the $[0, 1]$ interval of texture coordinates or the actual texels. One solution is to clamp the curvature values to a suitable interval, the other solution is to explicitly extract the curves and draw them as lines.

Lines of curvature Besides the scalar principal curvatures, the principal directions also carry information on the local surface properties. They define discrete direction fields in the tangent space of the surface. By linearly interpolating principal directions computed at the mesh vertices over triangles using barycentric coordinates, a continuous field can be defined. Lines of curvature can then be traced on this direction field using Euler integration (see Section **??** for more details).

Figure 6.5: Lines of curvature. Lines of curvature are superimposed on a flat shaded image of a VW Beetle model.

Fig. **??** shows lines of curvature that provide very good and intuitive impression of the surface. Alternatively texture based techniques like *line integral convolution* (LIC)[**?**] can also be used on triangle meshes. However, tracing and constructing a large number of lines of curvature is rather expensive compared to the other techniques.

6.3 The shape of triangles

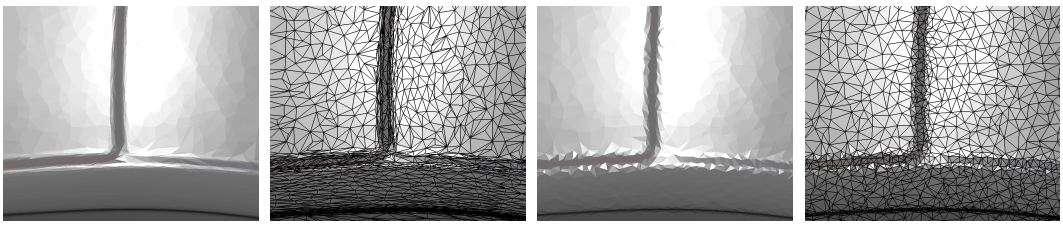


Figure 6.6: Triangle mesh optimized for smooth appearance, leading to skinny triangles (left), and for triangle shape, leading to rendering artifacts (right).

Some applications need "well-shaped", round triangles in order to prevent them from running into numerical problems, e.g., numerical simulations based on FEM. For this purpose, "round" triangles are needed, e.g., the ratio of the radius of the circumcircle to the shortest edge should be as small as possible [**?**] (cf. Fig. **??**).

The most common way to inspect the quality of triangles is to view a wireframe or hidden-line rendered image. This may not be an option for very complex meshes, however. A straightforward solution is a color coding criterion based on triangle shapes. This helps to identify even single "badly shaped" triangles (see also Chapter **??**).

7 Mesh Smoothing

Mesh smoothing is a central tool in geometry processing with many applications such as denoising of acquired data, surface blending and hole-filling, or design of high-quality surfaces. In addition, smoothing techniques constitute foundations for geometric filtering or signal processing used in multi-resolution shape editing and mesh deformation methods as will be discussed in Chapter **??**.

Many different techniques for mesh smoothing have been developed within the last decade. In this section, we will concentrate mainly on linear methods, namely Laplacian smoothing and (isotropic) mean curvature flow. Their main application is denoising and generation of fair surfaces as required in multi-resolution modeling.

7.1 General Goals

We distinguish two different goals of smoothing methods: The first is *denoising* of measured data. For instance meshes acquired by range scanners typically show high frequency noise, i.e., small perturbations in the vertex positions, which do not correspond to shape features. Fig. **??** shows a typical example. Here, the goal is to smooth out these artifacts in such a way that the global shape, or the low frequency components, is preserved. In signal processing this is called low-pass filtering, well-known, e.g., in image processing. Denoising algorithms must be able to handle fairly huge data sets efficiently, as they may be applied directly after acquisition and before simplification (Chapter **??**). This fact renders linear methods, i.e., those which only require numerical solution of a linear system, especially attractive. An additional requirement is often the preservation of certain surface features like sharp edges and corners, which should not be "blurred". However, this leads to non-linear methods.

A second goal is the design of high-quality, fair surfaces. This process is called *fairing*, and the resulting surfaces must satisfy certain aesthetic requirements. In order to find appropriate mathematical models these requirements are put essentially as *principle of the simplest shape* [**?**], meaning that an aesthetic surface is free of unnecessary detail such as noise or oscillations. Fig. **??** shows an example of fair surface design from boundary conditions. Mathematical formulations of this principle lead to the minimization of certain energy functionals, see Chapter **??**, which are often inspired by physical processes such as spanning a membrane or bending a thin plate. The energy functionals are typically formulated in terms of intrinsic shape properties, i.e., quantities that do not depend on the particular parameterization (or triangulation in the discrete setting), such as curvatures (see Chapter **??**). Hence the associated optimization problems are non-linear, and their numerical solution is more involved. Applications of fairing are for instance shape optimization or hole filling (see Chapter **??**). For the latter, the hole is first filled with a template mesh, which is then subject to fairing while the transition at the hole boundary is required to be smooth.

Finally, smoothing is often applied in order to make triangulations more regular. This is a well-known technique to ensure numerical robustness of finite element methods (usually for planar domains in bivariate settings). For surfaces this means that the distribution of vertices over the

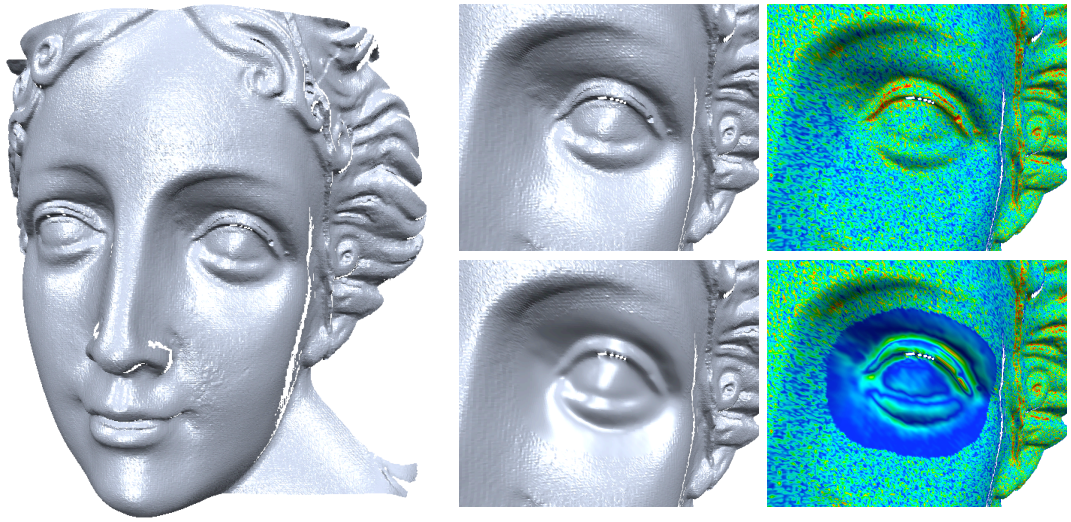


Figure 7.1: This scan of a statue's face contains typical measurement noise, which can be removed by low-pass filtering the surface geometry. The bottom row shows selective smoothing, for better visualization only the eye region is considered. Mean curvature is superimposed as color-code in the right column.

mesh is optimized. This process is part of (isotropic) remeshing described in Chapter **??**. In the following we review general approaches to mesh smoothing, their intuition and motivation.

7.2 Spectral Analysis and Filter Design

It is well-known from signal processing theory that Fourier transformation is a valuable tool for both, filter design and efficient implementation. For instance, every univariate signal function $f(t)$ is assumed to be a linear combination of periodic functions $e^{i\varphi t}$ (i.e., scaled and shifted sine waves) of different frequencies φ. Instead of observing the signal in the spatial domain, one considers its spectrum in the frequency domain. Assuming that noise is associated with high frequencies, an ideal denoising filter would cut off such high frequencies prior to the inverse Fourier transform to the spatial domain. This is called a low-pass filter.

We will see that a similar notion of *geometric frequencies* can be established for surfaces and used for filter design. (We refer also to multi-scale techniques for surface deformation in Chapter **??**.) However, contrary to image processing, analysis in the frequency domain will only serve as a theoretical tool and does not yield efficient implementations in general.

Let us for a moment consider the univariate case. The Fourier transform $F(\varphi)$ of a signal $f(t)$ is defined as

$$F(\varphi) = \frac{1}{2\pi} \int_{-\infty}^{\infty} f(t) e^{-i\varphi t} dt \ .$$

A low-pass filter would damp (or ideally cut off) high frequencies φ of F prior to the inverse transform, e.g., by multiplying F with a Gaussian. For (discrete) surfaces the situation is more

difficult, we require some generalization of the basis functions of type $e^{i\varphi t}$. Considering the identity

$$\frac{\partial^2}{\partial t^2} e^{i\varphi t} = \Delta e^{i\varphi t} = -\varphi^2 e^{i\varphi t} ,$$

it follows immediately that $e^{i\varphi t}$ are eigenfunctions of the Laplace operator Δ with eigenvalues $-\varphi^2$. Therefore, it seems natural to use eigenfunctions of the Laplace operator as basis also in the bivariate setting and for surfaces of arbitrary topology. As we know how to discretize the Laplacian on triangles meshes, this will provide the generalization of Fourier transformation for filter design.

7.2.1 The Discrete Setting: Spectral Graph Theory

The discrete Laplacian operator (see also Chapter **??**) on a piecewise linear surface, i.e., a triangle mesh, is expressed as

$$\Delta \mathbf{p}_i = \sum_{v_j \in \mathcal{N}_1(v_i)} \omega_{ij} (\mathbf{p}_j - \mathbf{p}_i) , \tag{7.1}$$

where for all vertices v_i weights are normalized such that

$$\sum_{v_j \in \mathcal{N}_1(v_i)} \omega_{ij} = 1 . \tag{7.2}$$

(Note that normalization and symmetry are not generally necessary for smoothing. In contrast, possibly required area terms destroy these properties, see also Chapter **??** and Chapter **??**.) We can now write the discrete Laplacian operator as a matrix $\mathbf{L}$ with non-zero entries

$$\mathbf{L}_{ij} = \begin{cases} -1 , & i = j \\ w_{ij} , & v_j \in \mathcal{N}_1(v_i) \end{cases}$$

$\mathbf{L}$ is generally sparse, the number of non-zeros in each row is one plus the valence of the associated vertex. For the uniform discretization Δ_{uni} we choose weights $\omega_{ij} = \frac{1}{\#\mathcal{N}_1(v_i)}$, i.e., the Laplacian depends only on the mesh connectivity. Then $\mathbf{L}$ is symmetric and has real eigenvalues and eigenvectors.

The eigenvectors of $\mathbf{L}$ form an orthogonal basis of $\mathbb{R}^n$, where n denotes the number of vertices, and the associated eigenvalues are commonly interpreted as *frequencies*. The projections of the coordinates $\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z \in \mathbb{R}^n$ into this basis is called *spectrum* of the geometry. Given eigenvectors $\mathbf{e}_i$, the x-components $\mathbf{p}_x$ of the mesh geometry can now be expressed as

$$\mathbf{p}_x = \sum_{i=1}^{n} \alpha_i^x \mathbf{e}_i ,$$

where the coefficients $\alpha_i^x = \mathbf{e}_i^T \mathbf{p}_x$, and similar for $\mathbf{p}_y, \mathbf{p}_z$. It shows that the eigenvectors associated with the first eigenvalues $0 \leq \lambda_1 \leq \cdots \leq \lambda_n$ correspond to low-frequency components: in other words, cancelling coefficients α_i^x associated with high-frequency components yields a smoothed version of the shape. Fig. **??** visualizes some eigenvectors on a model together with a synthesis using only very few low frequency components.

This is well-known from *spectral graph theory* [**?**]: the projection into the linear space spanned by the eigenvectors provides a generalization of the Discrete Fourier Transform. This can also

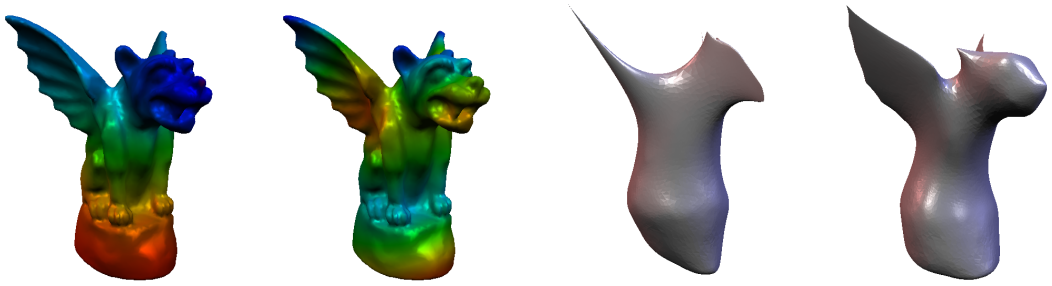


Figure 7.2: Spectral analysis of a gargoyle model. The first 20 of 10k eigenvectors were computed. Left: The 2nd and 10th eigenvector of the associated discrete Laplace operator are visualized by the color codes. (Values are uniformly scaled). Right: Reconstruction of the model using only the first 10 and 20 eigenvectors, respectively. (Reconstructions are rescaled.)

be seen immediately for the discrete univariate setting: here, the decomposition is equivalent to the discrete cosine transform (see, e.g., [**?**]).

For general surface meshes, their spectral decomposition defines a natural frequency domain. Taubin [**?**, **?**] uses this fact to motivate *geometric signal processing* and to define low-pass filters for smoothing meshes (see also [**?**]). In [**?**, **?**] spectral analysis is applied for mesh compression, taking advantage of low-pass and high-pass filter properties, respectively.

Although the matrix $\mathbf{L}$ is generally sparse it is in practice *not feasible* to explicitly compute eigenvalues and eigenvectors even for moderately sized meshes: computational costs are too high and one has to pay close attention to numerical robustness. (In practice, the computation of *some* eigenvalues in a specified range is possible, as shown in Fig. **??**.) Therefore, in [**?**] meshes are partitioned without enforcing smoothness across patch boundaries, whereas in [**?**, **?**], spectral analysis is applied as a theoretical tool. However, recent works [**?**, **?**] propose respectively to use multiresolution methods and algebraic transforms to make spectral mesh processing usable in practice.

Ideal low-pass filters are often too costly even in image processing. Instead of strictly truncating the frequency band, high frequencies are often damped, e.g., by weighting with an appropriate Gaussian kernel (often called Gaussian blurring). In a continuous setting, the Fourier transformation of a Gaussian kernel yields again a Gaussian. Therefore, in the spatial domain this corresponds to convolution with a Gaussian or more general to some weighted averaging. The situation is similar for mesh filtering.

7.2.2 The Continuous Setting: Manifold Harmonics

As shown in the previous subsection, the eigenvectors of the discrete combinatorial Laplacian $\mathbf{L}$ have interesting properties (orthogonality and spectral locality), that make them similar to the function basis used by the Fourier transform. However, this analogy is no-longer valid when the mesh presents poorly shaped triangles (see Chapter **??**). The obtained result is similar to Figure **??**b, whereas the result predicted by theory should look like Figure **??**d.

In fact, the analogy can be explained as follows: the discrete combinatorial Laplacian $\mathbf{L}$ is an approximation of the Laplace-Beltrami operator, and its eigenvectors are an approximation of its eigenfunctions. Note that for a 2D square, the eigenfunctions of the Laplace-Beltrami operator

Figure 7.3: Some of the elements of the MHB (Manifold Harmonic Basis) of the Gargoyle dataset. In a certain sense, the MHB generalizes Spherical Harmonics to arbitrary geometry and topology.

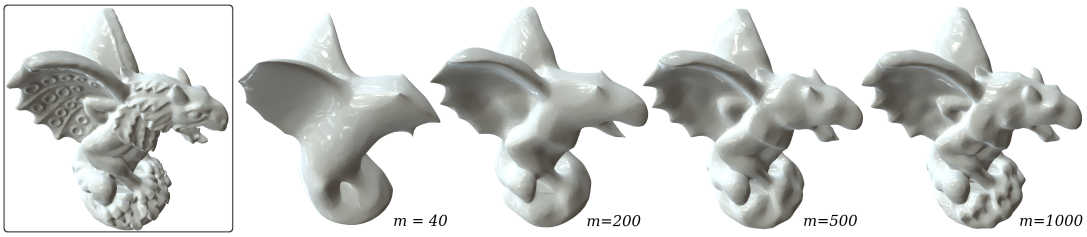


Figure 7.4: Reconstructions obtained with an increasing number of Manifold Harmonics. Since Manifold Harmonics take the geometry into account, no shrinking effect is observed.

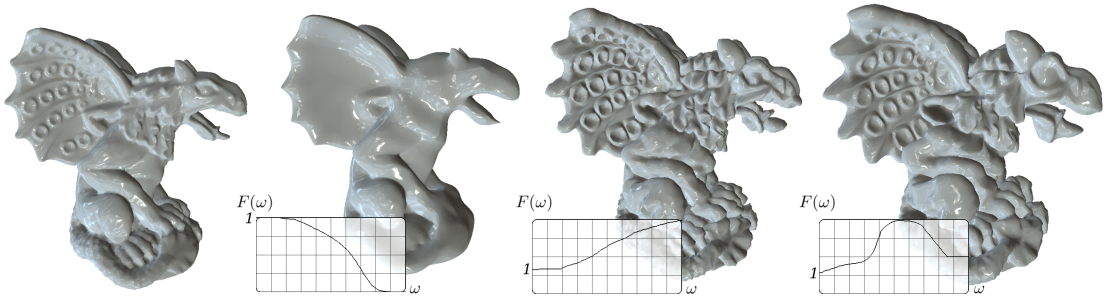


Figure 7.5: Once the MHB (Manifold Harmonics Basis) and MHT (Manifold Harmonic Transform) is computed, general convolution filtering can be performed in real time (left: low-pass, center: high-pass, right: enhancement).

correspond to the DCT function basis (used by the JPEG format), and for a sphere, they correspond to Spherical Harmonics. Thus, we understand that the eigenfunctions generalize these notions to arbitrary manifolds. Therefore, they are called *Manifold Harmonics* (or shape harmonics). Motivated by the very interesting results obtained by using the spectrum (eigenvalues) for shape classification [**?**], and obtained by using a single eigenvector for quad-remeshing [**?**], the idea of Manifold Harmonics was first experimented in [**?**], with a simple "symmetrization" of the matrix to preserve the orthogonality of the function basis. A more careful analysis of the discretization was conducted in [**?**], based on the Finite Element formalism, to compute both mesh-independent and orthogonal manifold harmonics. In addition, they proposed an efficient numerical solution mechanism to make spectral geometry processing usable in practice. Figure **??** shows some elements of the MHB (Manifold Harmonic Basis). Similar to the SHT (Spherical Harmonic Transform), one can define the MHT (Manifold Harmonic Transform), that converts the geometry into frequency space and computes MH coefficients. Figure **??** shows the geometry reconstructed with an increasing number of MH coefficients. Finally, using the MHB and MHT, as with the Fourier transform, it is easy to perform general convolution filtering in real-time, as shown in Figure **??**.

We illustrated the theoretical framework for ideal low-pass filtering and convolution filtering on meshes. Unfortunately this approach is generally too expensive to be practical in all applications. Therefore, we will now focus on two major techniques to mesh smoothing: diffusion flow and energy minimization. Note that although different in motivation for particular instances, these two approaches are closely related, and they can be justified by the above observations.

7.3 Diffusion Flow

Diffusion processes constitute a powerful and well-understood tool for smoothing signals. They often arise as physical processes in the real world, which makes them intuitive to understand. A common example is heat distribution in an object, where the local differences in temperature are equilibrated under conservation of energy. Let $x(u,t)$ denote the temperature at position u inside an object at time t, then the heat flow is given as $f = -\mu \nabla x$. Here, the diffusion constant $\mu > 0$ specifies the material conductivity. (Instead of a scalar in the isotropic case, we may set a positive definite symmetric matrix as diffusion tensor in general, see Section **??**.) Furthermore, due to conservation of energy the continuity equation $\frac{\partial x}{\partial t} = -\operatorname{div} f$ applies (assuming no heat injection). Then the *heat equation* is expressed as the linear diffusion equation

$$\frac{\partial}{\partial t} x \;=\; \operatorname{div} \mu \nabla x \;. \tag{7.3}$$

In the following we will consider this type of diffusion equation for mesh smoothing: the vertex positions are subject to diffusion such that small differences, i.e., noise, are equilibrated. For the steady state we have zero flow $\frac{\partial x}{\partial t} = 0$ and hence $\Delta x \;=\; 0$. We remark that for appropriate settings the solution $x(u,t)$ to the diffusion equation is a convolution of the initial value $x(u,0)$ with a Gaussian kernel depending on the time step t.

In the following, we review discrete solutions of linear diffusion equations for smoothing triangle meshes. Particular approaches differ in the differential operator and its particular discretization, and different numerical integration schemes can be applied.

7.3.1 Laplacian Smoothing

Laplacian smoothing is a simple and very effective technique based on linear diffusion of vertex positions $\frac{\partial \mathbf{p}}{\partial t} = \mu \Delta \mathbf{p}$. Obviously, for triangle meshes this method depends on the discretization of the Laplace operator (see Chapter **??**). The straightforward choice is a uniform discretization based on finite differences assuming a uniform triangulation.
Note that the uniform discretization smoothness geometry (shape) *and* triangulation, i.e., vertices move in normal direction as well as in their respective tangent planes.

7.3.2 Curvature Flow

Curvature is an intrinsic property of the surface that does not depend on parameterization (see Chapter **??**). Such independence of the particular triangulation of a shape is favorable for smoothing: only the geometry of the shape is supposed to be smoothed while at the same time the shape of each individual triangle should be preserved as much as possible. This means that vertices should be displaced only in normal direction rather than in the associated tangent plane. Tangential drift occurs indeed for the uniform discretization of the Laplacian (see above), and in most applications it is regarded as an undesirable artifact.

Mean curvature flow [**?**] considers the flow equation

$$\frac{\partial \mathbf{p}}{\partial t} \;=\; -\,\mu\, H \mathbf{n}\ . \tag{7.4}$$

For smoothing, vertex positions $\mathbf{p}$ move along the surface normal $\mathbf{n}$ with speed proportional to the mean curvature $H = \frac{1}{2}(\kappa_1 + \kappa_2)$. As $H = \operatorname{div} \mathbf{n}$, speed is reduced if the normal field spreads out less in a local region, and in the extreme case vertices stay in place for zero curvature. Using the identity $\Delta_{\mathcal{S}} \mathbf{p} = -2H\mathbf{n}$, we replace the right hand side of (**??**) and apply the well-known discretization of the Laplace-Beltrami operator $\Delta_{\mathcal{S}}$ (see Chapter **??**). This way, we can also interpret the mean curvature flow as diffusion using a more appropriate discretization of the Laplace operator on the surface (w.r.t. the initial mesh as parameter domain). The resulting linear diffusion equation reads as $\frac{\partial \mathbf{p}}{\partial t} = \mu \Delta_{\mathcal{S}} \mathbf{p}$. We remark that curvature flow has also been used in combination with parameterization regularization [**?**].

7.3.3 Higher Order Flows

Higher order flows based on Δ^k (or $\Delta_{\mathcal{S}}^k$) are used due to better low-pass properties (see, e.g, [**?**]). In practice, bi-Laplacian smoothing ($k = 2$) is a good trade-off between efficiency and quality: In the frequency domain higher orders of the Laplace operator yield better truncation (damping) of high frequencies. However, the associated discrete linear operator is less sparse (see also Chapter **??**). Note that higher order flows require (and are able to satisfy) higher order boundary conditions. This is similar to energy minimization methods discussed below.

7.3.4 Integration

A straightforward method for the numerical solution of the linear diffusion equations is *explicit* (or forward) Euler integration. This leads to an iterative algorithm using, e.g., the update rule

$$\mathbf{p}_i' \;=\; \mathbf{p}_i + \mu\, dt\, \Delta \mathbf{p}_i \tag{7.5}$$

on all vertex positions $\mathbf{p}_i$ for Laplacian smoothing. Updates can be applied simultaneously or sequentially [**?**] in iterative algorithms of Jacobi or Gauss-Seidel type, respectively. In practice, direct solvers (see Chapter **??**) in combination with implicit integration (see below) show superior efficiency and stability for most settings.

The above formula depends on the parameter $\mu\, dt$, which can be interpreted as time step and here should satisfy $0 < \mu\, dt < 1$ for stability reasons.

The explicit integration (**??**) of the (discrete) diffusion equation can be written in matrix form as

$$\mathbf{p}' \;=\; (\mathbf{I} + \mu\, dt\, \mathbf{L})\, \mathbf{p} \;,$$

where $\mu\, dt < 1$ is required. Desbrun et al. [**?**] propose the use of a *backward Euler method* for *implicit* smoothing, which is unconditionally stable without limitations on the time step. Such implicit integration reads as

$$(\mathbf{I} - \mu\, dt\, \mathbf{L})\, \mathbf{p}' \;=\; \mathbf{p}$$

and requires the solution of a (sparse) linear system for the unknowns $\mathbf{p}'$ (see Chapter **??**). The value of $\mu\, dt$ can be chosen arbitrarily, and it roughly corresponds to the number of explicit integration steps.

7.4 Energy Minimization

Methods based on energy minimization frequently appear in mesh fairing and fair surface design (see, e.g., [**?**, **?**, **?**, **?**, **?**]). The idea is to penalize unaesthetic behavior of the shape. For this purpose different fairness functionals have been proposed. Ideally such functionals depend only on intrinsic surface properties, such as curvature, and not on a particular parameterization. For the discrete setting one can then expect the same geometric shape of the solution regardless of the initial triangulation.

Best known in this context is the *total curvature* of a surface $\mathcal{S}$

$$\int_{\mathcal{S}} \kappa_1^2 + \kappa_2^2 \, dA \;, \tag{7.6}$$

expressed as the area integral of the sum of squared principal curvatures (see, e.g., [**?**] and Chapter **??**).

Parameter independence has a price, however: minimization problems are non-linear and the numerical computation of solutions (see, e.g., [**?**]) is generally too expensive to be practical for large meshes. For isometric parameterizations $\mathbf{x} : \Omega \to \mathbb{R}^3$, minimizing (**??**) is equivalent to minimizing

$$\iint_{\Omega} \|\mathbf{x}_{uu}\|^2 + 2\,\|\mathbf{x}_{uv}\|^2 + \|\mathbf{x}_{vv}\|^2 du\, dv \;. \tag{7.7}$$

This energy has a physical interpretation: it expresses the bending energy of a *thin plate* spanned across a domain Ω.

Generally, such approaches linearize curvature terms by higher order derivatives for the sake of giving up parameter independence. Still, ad hoc minimization of (**??**) is rather involved. Fortunately, for some fairness functionals the minimizers are characterized by solutions of *linear* systems. In this case applying variational calculus [**?**] yields the minimizer as solution of the associated Euler-Lagrange equation

$$\Delta^2 \mathbf{x} \;=\; 0 \;,$$

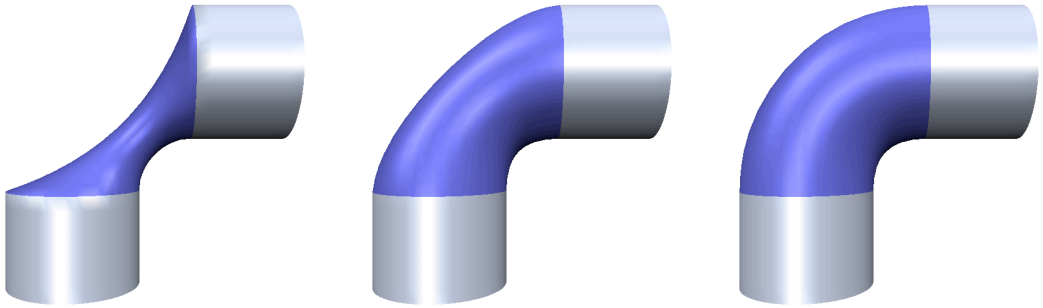


Figure 7.6: The order k of the energy functional and of the corresponding Euler-Lagrange PDE $\Delta_S^k \mathbf{x} = 0$ defines the stiffness of the surface in the support region and the maximum smoothness C^{k-1} of the boundary conditions. From left to right: membrane surface ($k = 1$), thin-plate surface ($k = 2$), minimum variation surface ($k = 3$).

subject to appropriate boundary conditions [**?**]. Note that this equation also characterizes the equilibrium of the linear diffusion $\frac{\partial \mathbf{x}}{\partial t} = -\mu\Delta^2\mathbf{x}$ [**?**], and its discretization leads to a linear system. In Chapter **??** we discuss efficient solvers for such systems.

Similarly, minimizing the membrane energy (**??**)

$$\iint_\Omega \|\mathbf{x}_u\|^2 + \|\mathbf{x}_v\|^2 du\, dv \;, \tag{7.8}$$

which captures the energy of a membrane spanned across a domain Ω, leads to solving $\Delta\mathbf{x} = 0$.

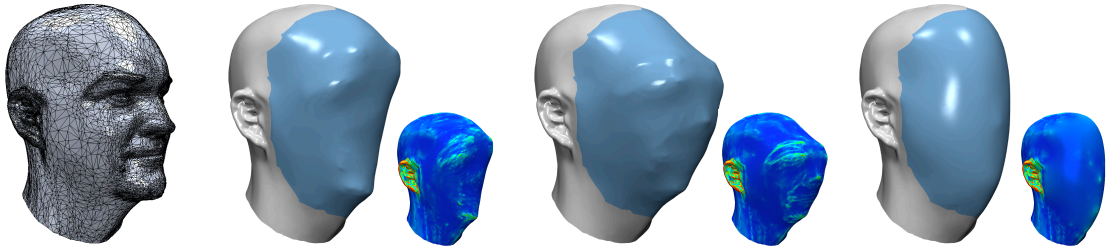
For achieving higher order fairness the following well-known functional is minimized

$$\int_{\mathcal{S}} \left(\frac{\partial \kappa_1}{\partial \mathbf{e}_1}\right)^2 + \left(\frac{\partial \kappa_2}{\partial \mathbf{e}_2}\right)^2 dA \tag{7.9}$$

to penalize variation of curvature, yielding *minimum variation surfaces* [**?**]. Giving up parameter independence corresponds to solving the sixth-order PDE $\Delta^3\mathbf{x} = 0$.

The Euler-Lagrange equations associated with minimizers of various fairing functionals show their relation to steady state solutions of diffusion flow (and hence signal processing and low-pass filters). It follows that fairing indeed refers to designing fair surfaces that ideally depend only on the given boundary conditions: for surfaces derived from $\Delta^k\mathbf{x} = 0$, boundary constraints of order $\mathcal{C}^{k-1}$ are interpolated. Fig. **??** illustrates the application of different fairing functionals with appropriate boundary conditions for a simple cylindrical shape. This is in contrast to denoising which is usually far from the steady state. Note that for the solution of the arising linear systems appropriate boundary conditions have to be applied to guarantee the existence of solutions. (The Laplacian matrix does not have full rank.)

Fig. **??** shows the effect of different discretizations of the Laplace-Beltrami operator (see Chapter **??**) when minimizing the thin-plate energy of an irregular mesh by solving the Euler-Lagrange equation $\Delta_S^2\mathbf{p} = \mathbf{0}$. Both the uniform Laplacian and the cotangent Laplacian without the area term yield artifacts in regions of high vertex density. The cotangent discretization including the per-vertex normalization clearly gives the best results.



(a) (b) (c) (d)

Figure 7.7: Comparison of different Laplace-Beltrami discretizations when solving $\Delta_S^2 \mathbf{p} = \mathbf{0}$. (a) irregular triangulation of the input mesh, (b) uniform Laplacian, (c) cotangent Laplacian without the area term, (d)cotangent discretization including the per-vertex normalization. The small images shows the respective mean curvatures.

7.5 Extensions and Alternative Methods

We classified smoothing schemes into two categories depending on whether they are based on diffusion flow or energy minimization. Both categories lead to PDE discretization, and both are tightly connected as we focus on linear methods and the required simplifications. In the following we briefly review some (non-linear) extensions and alternative methods.

7.5.1 Anisotropic Diffusion

Denoising is supposed to smooth out small perturbations in a surface or outliers from measurements. The techniques discussed so far assume smooth surfaces and are not aware of surface features, i.e., sharp edges or creases and corners. However, most shapes are only piecewise smooth and denoising will also blur features as these are also represented by high-frequency components similar to what is assumed for noise.

This problem has been well-studied in image processing and a common approach to feature-preserving filtering is anisotropic diffusion [**?**] (see also [**?**]). The basic idea is to consider the diffusion equation (**??**) and to replace the scalar diffusion constant μ by a data dependent *diffusion tensor* $\mathbf{D}$. This modification renders the equation non-linear and guides the directional (i.e., anisotropic) diffusion. A natural choice for $\mathbf{D}$ is the curvature tensor (in combination with an appropriate transfer function), which enables feature preservation or even enhancement: the speed of the flow is reduced in directions of high normal curvature, e.g., across sharp edges. There are various related approaches to feature preserving smoothing as for instance in [**?**, **?**, **?**, **?**].

7.5.2 Normal Filtering

The basic idea of normal filtering methods is as follows: instead of filtering the spatial coordinates, the normal field of the surface is smoothed. The resulting normals are then integrated in order to reconstruct a smooth surface. Hence, in contrast to smoothing surfaces, or vertex positions, directly, their derivatives are subject to smoothing. This is usually achieved by a diffusion process [**?**, **?**, **?**, **?**, **?**]. We remark that normal smoothing is commonly applied as a preprocess for stabilization (mollification) in order to get reliable estimates for other methods (see, e.g., [**?**]).

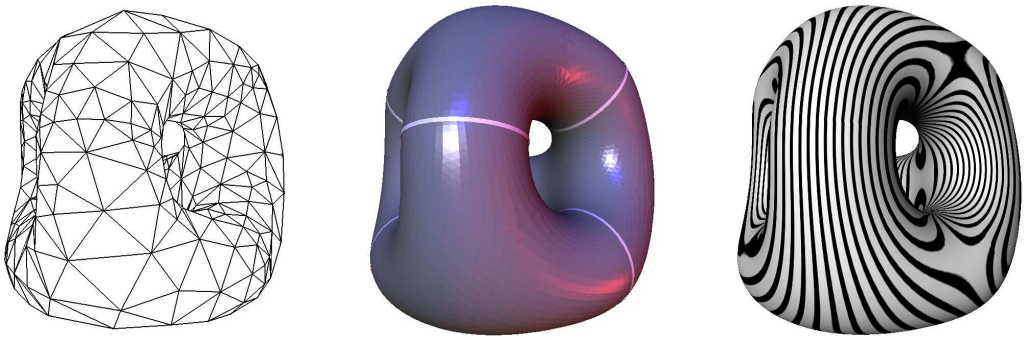


Figure 7.8: Six circles with C^1 boundary-conditions are used to design a "tetra thing". Due to the symmetry the final solution is actually G^2 continuous in this case, which is indicated by the smooth reflection lines (see Chapter **??**). Surfaces are constructed using the intrinsic fairing method [**?**] based on solving $\Delta_S H = 0$, hence the solution is independent of the triangulation (or parameterization, respectively).

7.5.3 Statistical Methods

Smoothing can also seen from a statistical point of view: signal and noise are assumed to be stochastic processes with known spectral characteristics or known autocorrelation and cross-correlation. The Wiener filter is a well-known example from image processing. Local adaptive Wiener filtering has been adapted to denoising discrete surfaces [**?**, **?**, **?**]. Also the following bilateral filtering relies on robust statistical estimations.

7.5.4 Bilateral Filtering

Bilateral filtering of images [**?**] (see also [**?**] for relation to nonlinear diffusion) is a powerful feature-preserving filtering technique. The central idea is to consider both, the image domain (as for classical filtering) and its range: each pixel becomes a weighted average of *similar* pixels in the neighborhood, where "similar" is defined in terms of spatial distance *and* intensity.

In [**?**, **?**] bilateral filtering is adapted to denoising surface meshes, where spatial distance and local variation of normals is taken into account. In [**?**] the normal displacement of vertex positions for smoothing is computed based on weighted averages of these measured. The non-iterative approach in [**?**] does not require explicit connectivity information and applies (mollified) normals to predict vertex positions, which are used for weighting. The rationale behind this is that prediction fails near shape features, i.e., distances to such predicted points are larger.

7.5.5 Approaches based on non-linear PDEs

Such methods should depend exclusively on intrinsic properties, i.e., be independent of the parameterization. In [**?**] a PDE-based method was developed for design of fair surfaces. The method enables G^1 boundary constraints (prescribed as vertices and unit normals), such that

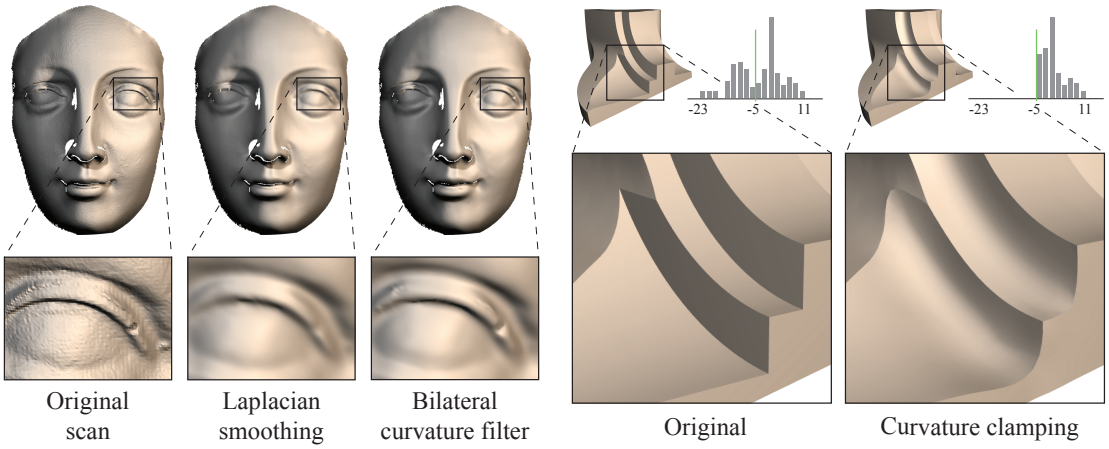


Figure 7.9: Curvature-domain shape processing. The scan on the left has been smoothed using a bilateral filter on the principal curvatures. The fandisk model on the right has been processed by clamping the negative curvatures to obtain smooth fillets for concave corners. The histograms show the distribution of the minimum signed curvature on a logarithmic scale before and after the optimization.

the resulting shape is independent of the particular triangulation. This particular approach is based on solving the fourth-order non-linear PDE

$$\Delta_{\mathcal{S}} H \;=\; 0, \tag{7.10}$$

i.e., it depends purely on intrinsic properties. This can be interpreted as one possible nonlinear analogon to thin plate splines minimizing (**??**), and the equation characterizes the equilibrium of the Laplacian of curvature flow [**?**]. Due to the mean value property of the Laplacian the extremal mean curvatures are obtained at the boundaries. As a consequence there are no local extrema in the interior [**?**], and thus the principle of simplest shape requirement is satisfied. Notice that the numeric solution of the PDE requires high-quality discretization of the mean curvature (following [**?**], see Chapter **??**). For efficiency reasons the fourth order PDE is factored into two second order problems. Bobenko and Schröder [**?**] used discrete Willmore flow for denoising and fair surface design. The minimizer of the associated energy functional also minimizes (**??**) for certain settings.

7.5.6 Curvature-Domain Shape Processing

Eigensatz and co-workers [**?**] propose a framework for 3D geometry processing that provides direct access to surface curvature to facilitate advanced shape editing, filtering, and synthesis algorithms. The central idea is to map a given surface to the curvature domain by evaluating its principle curvatures, apply filtering and editing operations to the curvature distribution, and reconstruct the resulting surface using an optimization approach. Their method allows the user to prescribe arbitrary principle curvature values anywhere on the surface. The optimization solves a nonlinear least-squares problem to find the surface that best matches the desired target curvatures while preserving important properties of the original shape. Figure **??** shows applications of this approach for anisotropic smoothing.

7.6 Summary

We gave a brief overview of mesh smoothing techniques with focus on linear methods based on diffusion flow and energy minimization, revealing relations between the two approaches and relations to spectral analysis. These techniques are linear and hence very efficient and well-understood, see also Chapter **??** for efficient numerical solvers and overview of computational costs. They constitute basic tools for further geometry processing steps, e.g., for shape deformation Chapter **??**. We listed several alternative techniques and summarized their main ideas. In conclusion we remark that there are several other aspects in smoothing that were not discussed here, such as volume preservation or existence of solutions (which is still unknown for minimization of many standard non-linear functionals).

8 Mesh Parameterization

This chapter aims at giving an intuition of the notion of parameterization and its implementation in the geometry processing setting. A more detailed version of this section (with the proofs of theorems and formula) is also available in SIGGRAPH 2007 *Mesh Parameterization, Theory and Practice* course notes (http://www2.in.tu-clausthal.de/~hormann/parameterization/index.html). See also the following surveys:

- M. S. Floater and K. Hormann. Surface Parameterization: a Tutorial and Survey. In Advances in Multiresolution for Geometric Modelling, Springer, 2005.
- A. Sheffer, E. Praun, K. Rose, Mesh Parameterization Methods and their Applications, Foundations and Trends in Computer Graphics and Vision, to appear.
- Some source code is also available from http://alice.loria.fr/software.

As we have seen in Chapter **??**, many different representations are used to encode the geometry of 3D objects. The choice of a representation depends on the acquisition process upstream, and on the application downstream. Unfortunately, the representations that are the easiest to reconstruct are in most cases not optimum for the applications.

In this chapter, we will review several methods that construct a parameterization of a triangulated mesh. Intuitively, this means attaching a "geometric coordinate system" to the object. This facilitates converting from one representation to another. For instance, it is possible to convert a mesh model into a piecewise bi-cubic surface, much easier to manipulate in Computer Aided Design packages. In a certain sense, this retrieves an "equation" of the geometry. One can also say that this constructs an *abstraction* of the geometry: Once the geometry is abstracted, re-instancing it into alternative representations is made easier. Before entering the heart of the matter, we list some important applications of mesh parameterization.

The abstract representation constructed by parameterization algorithms has many possible applications. Historically, the main application domain of mesh parameterization was texture mapping[1]. Figure **??** shows an example of the LSCM method [**?**] implemented in the Blender open-source modeler. The parameterization is used to put the surface into one-to-one correspondence with an image, stored in the 2D domain. It is possible to either map an existing image and deform it onto the model, or use the parameter space to paint onto the model, with a 3D paint system.

With the advent of programmable texture mapping hardware, texture mapping can be used to map more complex attributes onto surfaces. The example shown in Figure **??** demonstrates a technique referred to as *normal mapping* (see, e.g., [**?**]). The initial object (shown on the left) is replaced with a decimated version (center). Its appearance is preserved by keeping the normals

[1] with the exception of Floater's seminal paper [**?**], about surface approximation.

in a texture (right), and a fragment program computes the lighting model. The model shown here was first made homeomorphic to a disc [**?**], then parameterized using ABF++ [**?**], and decimated (see Chapter **??**). As can be seen, a much lighter version of the object can be used, while preserving its overall visual appearance. Since "pixels cost less than triangles", replacing triangles with pixels is an important benefit, especially for real-time rendering.

Another important class of applications concerns re-meshing algorithms. This aspect is detailed in Chapter **??**. Finally, the abstraction realized by the parameterization facilitates converting from a mesh representation into an alternative one. This is of paramount importance for modeling and simulation tasks, that use representations that are completely different from the dense triangulated meshes constructed by 3D scanners and the companion reconstruction software. More specifically, these applications require parametric representations (see Chapter **??** for an introduction about surface representations). For instance, Figure **??** shows how a mesh can be transformed into a parametric representation, using a global parameterization method [**?**]. This fills the gap between acquisition and CAD / Finite Element simulations.

To summarize, formally, a parameterization of a 3D surface is a function putting this surface in one-to-one correspondence with a 2D domain. This notion plays an important role in geometry processing, since it makes it possible to transform complex 3D problems into a 2D space where they are simpler to solve. The next section gives a simple example of a parameterization, to let the reader grasp the basic concepts.

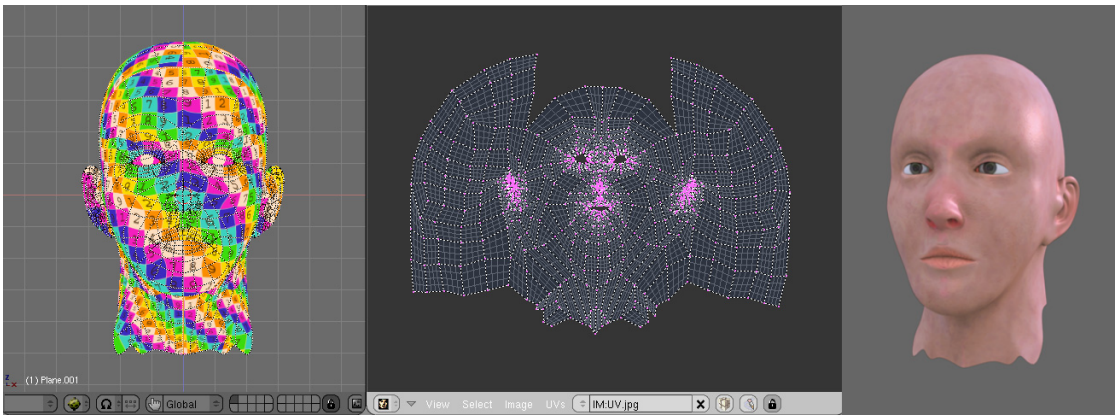


Figure 8.1: Application of parameterization: texture mapping (Least Squares Conformal Maps implemented in the Open-Source Blender modeler).

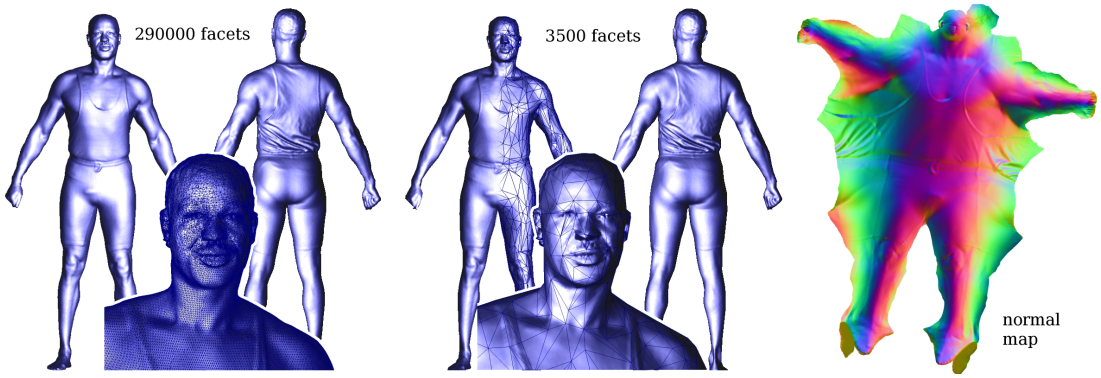


Figure 8.2: Application of parameterization: appearance-preserving simplification. All the details are encoded in a normal map, applied onto a dramatically simplified version of the model (1.5% of the original size).

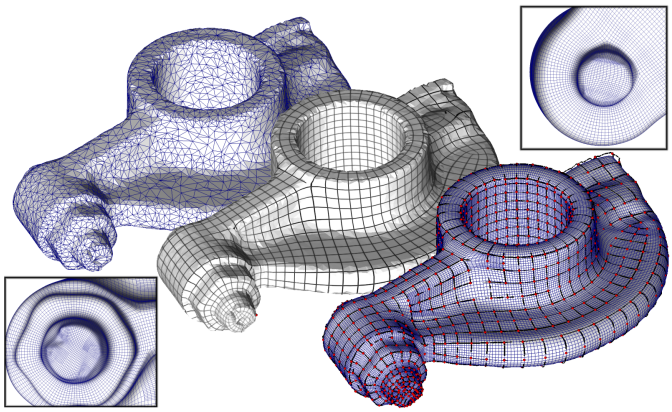


Figure 8.3: A global parameterization realizes an abstraction of the initial geometry. This abstraction can then be re-instantiated into alternative shape representations.

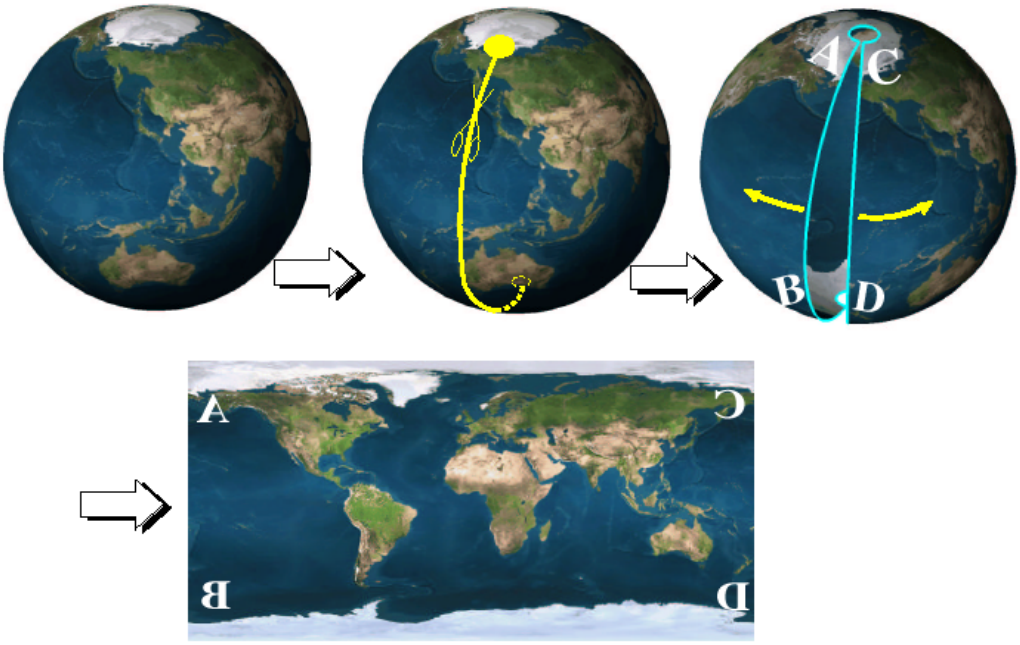


Figure 8.4: Cut me a meridian, and I will unfold the world !

8.1 World Map and Spherical Coordinates

Let us consider the problem of drawing a map of the world. As shown in Figure **??**, the problem is to find a way to 'unfold' the surface of the world, in order to obtain a flat 2D surface. Since the surface of the world is closed, to unfold it, it is necessary to cut it. For instance, it can be cut along a meridian, i.e., a curve joining the two poles. In the unfolding process, note that the two poles are stretched and become two curves. The North pole is transformed into the $[\mathbf{A}-\mathbf{C}]$ segment, and the south pole into the $[\mathbf{B}-\mathbf{D}]$ segment[2]. It can also be noticed that the meridian along which the sphere has been cut corresponds to two different curves: the $[\mathbf{A}-\mathbf{B}]$ and the $[\mathbf{C}-\mathbf{D}]$ segments. In other world, if a city is located exactly on this meridian, it appears on the map twice.

As shown in Figure **??**, it is possible to provide each point of the map with two coordinates (θ, ϕ). In the mapping shown in Figure **??**, the (x, y, z) coordinates in 3D space and the (θ, ϕ) coordinates in the map are linked by the following equation, referred to as a *parametric* equation of a sphere:

$$\begin{array}{rcl} \theta & \in & [0\ldots 2.\pi], \\ \phi & \in & [-\pi\ldots\pi] \end{array} \mapsto \left\{ \begin{array}{rcl} x(\theta,\phi) & = & R\cos(\theta).\cos(\phi) \\ y(\theta,\phi) & = & R\sin(\theta).\cos(\phi) \\ z(\theta,\phi) & = & R\sin(\phi) \end{array} \right. , \tag{8.1}$$

where R denotes the radius of the sphere. Note that this equation is different from the *implicit* equation of the sphere $x^2+y^2+z^2=R^2$. The implicit equation provides a mean of testing whether a given point is on the sphere, whereas the parametric equation describes a way of transforming the $[0\ldots 2.\pi]\times[-\pi\ldots\pi]$ rectangle into a sphere (see also Section **??**).

[2]Note that this can be different in a real world map. In our example, we have used a mapping having a simple equation, i.e., corresponding to a simpler parameterization than in a real world map.

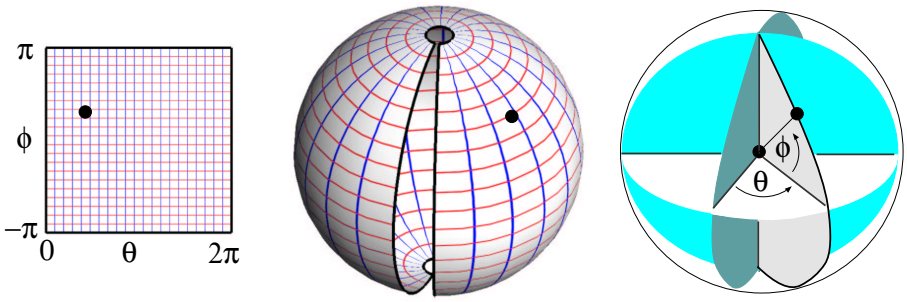


Figure 8.5: Spherical coordinates

Concerning the parametric equation, the following definitions can be given:

- The coordinates (θ, ϕ) at a point $\mathbf{p} = (x, y, z)$ are referred to as the *spherical coordinates* at $\mathbf{p}$.
- Each vertical line in the map, defined by $\theta =$ Constant, corresponds to a curve on the 3D surface, referred to as an *iso-θ*. In our case, the iso-θ curves are circles traversing the two poles of the sphere (the *meridians* of the globe).
- Each horizontal line in the map, defined by $\phi =$ Constant corresponds to an *iso-ϕ* curve. In our case, the iso-ϕ curves are the *parallels* of the globe, and the iso-ϕ corresponding to $\phi = 0$ is the *equator*.

As can be seen in Figure **??**, drawing the iso-θ and the iso-ϕ curves helps understanding how the map is *distorted* when applied onto the surface. In the map, the iso-θ and iso-ϕ curves are respectively vertical and horizontal lines, forming a regular grid. Visualizing what this grid becomes when the map is applied onto the surface makes it possible to see the distortions occurring near the poles. Near the poles, the squares of the grid are highly distorted. We will see further how to measure the corresponding distortions.

To sum-up, the parametric equation of the sphere explains how to construct a sphere, whereas the usual (implicit) equation makes it possible to test whether a given point belongs to the sphere. Each point of the sphere has unique θ and ϕ coordinates, therefore, the parameterization defines a (curvilinear) coordinate system on the sphere. Note that even if a point of the sphere has three (x, y, z) coordinates, two coordinates (θ, ϕ) are sufficient to refer to it. A surface of the 3D space is in fact a 2D object, which can be revealed by expressing a parameterization. The parameter space may be seen as a 'map' of the surface.

8.2 Distortion Analysis, Anisotropy

The previous section has shown a simple example of parameterization. This section now considers that a parameterization of a given surface is known, and describes means of quantifying how much the parameter space is distorted when transformed into the surface.

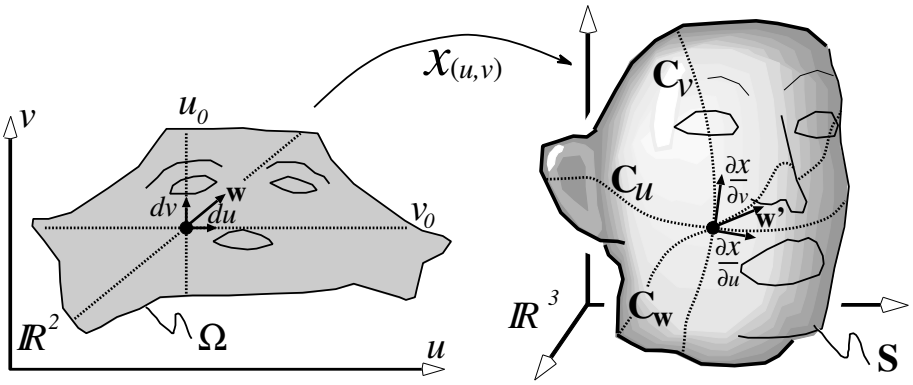


Figure 8.6: Elementary displacements from a point (u, v) of Ω along the u and the v axes are transformed into the tangent vectors to the iso-u and iso-v curves passing through the point $\mathbf{x}(u, v)$

The Jacobian matrix and the 1^{st} fundamental form

The first derivatives of the parameterization are involved in distortion analysis, it is then necessary to have an intuition of their geometric meaning. In physics, material point mechanics studies the movement of an object, approximated by a point $\mathbf{p}$, when forces are applied to it. The *trajectory* is the curve described by the point $\mathbf{p}$ when t varies from t_0 to t_1, where t denotes time. The function putting a given time t in correspondence with the position $\mathbf{p}(t) = \{x(t), y(t), z(t)\}$ of the point $\mathbf{p}$ is a parameterization of the trajectory, i.e., a parameterization of a *curve*.

It is well known that the vector of the derivatives $\mathbf{v}(t) = \partial\mathbf{p}/\partial t = \{\partial x/\partial t, \partial y/\partial t, \partial z/\partial t\}$ corresponds to the *speed* of $\mathbf{p}$ at time t.

As shown in Figure **??**, we consider now a function $\mathbf{x} : (u, v) \mapsto (x, y, z)$, putting a subspace Ω of $\mathbb{R}^2$ into one-to-one correspondence with a surface $\mathcal{S} \subset \mathbb{R}^3$. The scalars (u, v) are the coordinates in parameter space[3]. In the case of a curve parameterization, the curve is described by a single parameter t. In contrast, in our case, we consider a surface parameterization $\mathbf{x}(u, v) = \{x(u, v), y(u, v), z(u, v)\}$, and there are **two** parameters, u and v. Therefore, at a given point (u_0, v_0) of the parameter space Ω, there are two "speed" vectors to consider: $\mathbf{x}_u(u_0, v_0) = (\partial\mathbf{x}/\partial u)(u_0, v_0)$ and $\mathbf{x}_v(u_0, v_0) = (\partial\mathbf{x}/\partial v)(u_0, v_0)$. It is easy to check that $\mathbf{x}_u(u_0, v_0)$ is the "speed" vector of the curve $\mathcal{C}_u : t \mapsto \mathbf{x}(u_0 + t, v_0)$ at $\mathbf{x}(u_0, v_0)$ and that $\mathbf{x}_v(u_0, v_0)$ is the "speed" vector of the curve $\mathcal{C}_v : t \mapsto \mathbf{x}(u_0, v_0 + t)$. The curve $\mathcal{C}_u$ (resp. $\mathcal{C}_v$) is the iso-u (resp. the iso-v) curve passing through $\mathbf{x}(u_0, v_0)$, i.e., the image through $\mathbf{x}$ of the line of equation $u = u_0$ (resp. $v = v_0$).

At that point, one may think that the information provided by the two vectors $\mathbf{x}_u(u_0, v_0)$ and $\mathbf{x}_v(u_0, v_0)$ is not sufficient to characterize the distortions between Ω and $\mathcal{S}$ in the neighborhood of (u_0, v_0) and $\mathbf{x}(u_0, v_0)$. In fact, they can be used to compute how an arbitrary vector $\mathbf{w} = (a, b)$ in parameter space is transformed into a vector $\mathbf{w}'$ in the neighborhood of (u_0, v_0). In other words, we want to compute the "speed" vector $\mathbf{w}' = \partial\mathbf{x}(u_0 + t \cdot a, v_0 + t \cdot b)/\partial t$ of the curve

[3] Since the names θ and ϕ used in the previous section for the parameters evoke angles, which is not always appropriate in the general case, the more neutral u and v names are used from now.

corresponding to the image of the straight line $(u, v) = (u_0, v_0) + t \cdot w$. The vector $\mathbf{w}'$, i.e., the tangent to the curve $\mathcal{C}_{\mathbf{w}}$, can be simply computed by applying the chain rule, and one can check that it can be computed from the derivatives of $\mathbf{x}$ as follows: $\mathbf{w}' = a\mathbf{x}_u(u_0, v_0) + b\mathbf{x}_v(u_0, v_0)$. The vector $\mathbf{w}'$ is referred to as the *directional derivative* of $\mathbf{x}$ at (u_0, v_0) relative to the direction $\mathbf{w}$.

In matrix form, $\mathbf{w}'$ is obtained by $\mathbf{w}' = \mathbf{J}(u_0, v_0)\mathbf{w}$, where $\mathbf{J}(u_0, v_0)$ is the matrix of all the partial derivatives of $\mathbf{x}$:

$$\mathbf{J}(u_0, v_0) = \begin{bmatrix} \frac{\partial x}{\partial u}(u_0, v_0) & \frac{\partial x}{\partial v}(u_0, v_0) \\ \frac{\partial y}{\partial u}(u_0, v_0) & \frac{\partial y}{\partial v}(u_0, v_0) \\ \frac{\partial z}{\partial u}(u_0, v_0) & \frac{\partial z}{\partial v}(u_0, v_0) \end{bmatrix} = [\mathbf{x}_u(u_0, v_0)\ ,\ \mathbf{x}_v(u_0, v_0)]\ , \tag{8.2}$$

The matrix $\mathbf{J}(u_0, v_0)$ is referred to as the *Jacobian matrix* of $\mathbf{x}$ at (u_0, v_0).

The notion of directional derivative makes it possible to know what an elementary displacement $\mathbf{w}$ from a point (u_0, v_0) in parameter space becomes when it is transformed by the function $\mathbf{x}$. The Jacobian matrix helps also computing dot products and vector norms onto the surface $\mathcal{S}$. This can be done using the matrix $\mathbf{J}^T\mathbf{J}$, referred to as the 1^{st} *fundamental form* of $\mathbf{x}$, also described in the differential geometry section (Chapter **??**). This matrix is denoted by $\mathbf{I}$, and defined by:

$$\mathbf{I}(u_0, v_0) = \mathbf{J}^T\mathbf{J} = \begin{bmatrix} \mathbf{x}_u^T\mathbf{x}_u & \mathbf{x}_u^T\mathbf{x}_v \\ \mathbf{x}_v^T\mathbf{x}_u & \mathbf{x}_v^T\mathbf{x}_v \end{bmatrix}\ , \tag{8.3}$$

The 1^{st} fundamental form $\mathbf{I}(u_0, v_0)$ is also referred to as the *metric tensor* of $\mathbf{x}$, since it makes it possible to measure how distances and angles are transformed in the neighborhood of (u_0, v_0). The squared norm of the image $\mathbf{w}'$ of a vector $\mathbf{w}$ is given by $||\mathbf{w}'||^2 = \mathbf{w}^T\mathbf{I}\mathbf{w}$, and the dot product $\mathbf{w}_1'^T\mathbf{w}_2' = \mathbf{w}_1^T\mathbf{I}\mathbf{w}_2$ determines how the angle between $\mathbf{w}_1$ and $\mathbf{w}_2$ is transformed. The next section gives a geometric interpretation of the 1^{st} fundamental form and its eigenvalues.

The anisotropy ellipse

The previous section has studied how an elementary *displacement* from a parameter-space location (u_0, v_0) is transformed through the parameterization $\mathbf{x}$. As shown in Figure **??**, further characterization is obtained by seeing that an elementary *circle* becomes an elementary ellipse. Considering the eigenvectors $\mathbf{e}_1$ and $\mathbf{e}_2$ of the metric tensor, and the associated eigenvalues λ_1, λ_2, one can show that:

- The axes of the anisotropy ellipse are $\mathbf{J}\mathbf{e}_1$ and $\mathbf{J}\mathbf{e}_2$;
- The lengths of the axes are $\sqrt{\lambda_1}$ and $\sqrt{\lambda_2}$.

Note that the lengths of the axes $\sqrt{\lambda_1}$ and $\sqrt{\lambda_2}$ also correspond to the singular values of the Jacobian matrix $\mathbf{J}$.

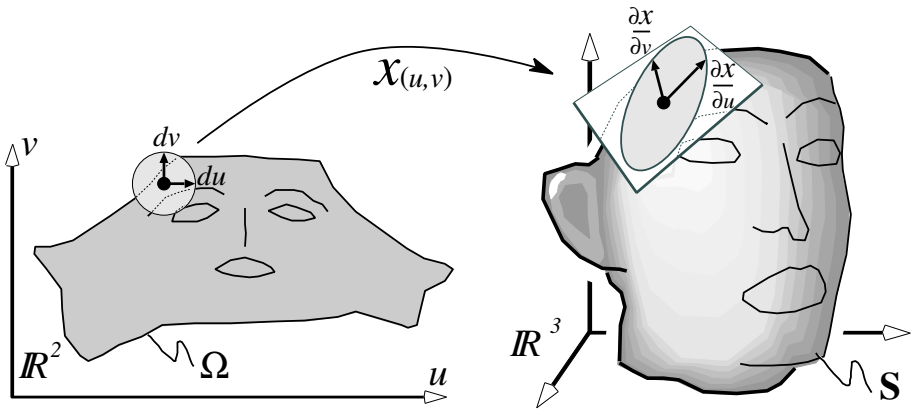


Figure 8.7: Anisotropy: an elementary circle is transformed into an elementary ellipse.

8.3 Triangulated Surfaces

A triangulated surface is a set $\mathcal{V}$ of vertices v_i with positions $\mathbf{p}_i$, $i = 1 \dots n$, connected by a set $\mathcal{F}$ of triangles. Each triangle is defined by the triplet (i, j, k) that denotes the indices of its vertices. It is also sometimes useful to introduce the set $\mathcal{E}$ of all the edges (i, j) of the mesh. Thus, a triangulated surface is defined by the triplet $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ of vertices $\mathcal{V}$, edges $\mathcal{E}$ and triangles (or facets) $\mathcal{F}$. More details on data structures for meshes are given in Chapter **??**.

A natural idea to define a parameterization of a triangulated surface consists in using piecewise linear functions (the pieces correspond to the triangles of the surface). Thus, it is possible to represent the parameterization by the set of all (u_i, v_i) coordinates associated with each vertex (x_i, y_i, z_i). Figure **??** shows an example of a parameterized triangulated surface in 3D space and in parametric (u, v) space.

Note that the previous section considered an existing parameterization, whereas this section considers the problem of constructing a parameterization for an existing surface. For this reason, in contrast with the conventions of differential geometry that we used in the previous section, it is more natural to place the 3D space (known) to the left of the figures, and the 2D parametric space (unknown) to the right of the figures. We will see more fundamental implications of this "swapping", when we will explain the formulation and behavior of classical methods.

At a given point (u, v) of the parametric space Ω, the parameterization $\mathbf{x}$ is given by:

$$\mathbf{x}(u, v) = \lambda_1 \mathbf{p}_i + \lambda_2 \mathbf{p}_j + \lambda_3 \mathbf{p}_k \ ,$$

where (i, j, k) denotes the index triplet such that the triangle (u_i, v_i), (u_j, v_j), (u_k, v_k) in parameter space contains the point (u, v). The triplet $(\lambda_1, \lambda_2, \lambda_3)$ denotes the barycentric coordinates at point (u, v) in that triangle.

To summarize, constructing a parameterization of a triangulated surface means finding a set of couples (u_i, v_i), associated with each vertex i. Moreover, these coordinates need to be such that the image of the surface in parameter space does not self-intersect. We will see in what follows several methods to compute these coordinates.

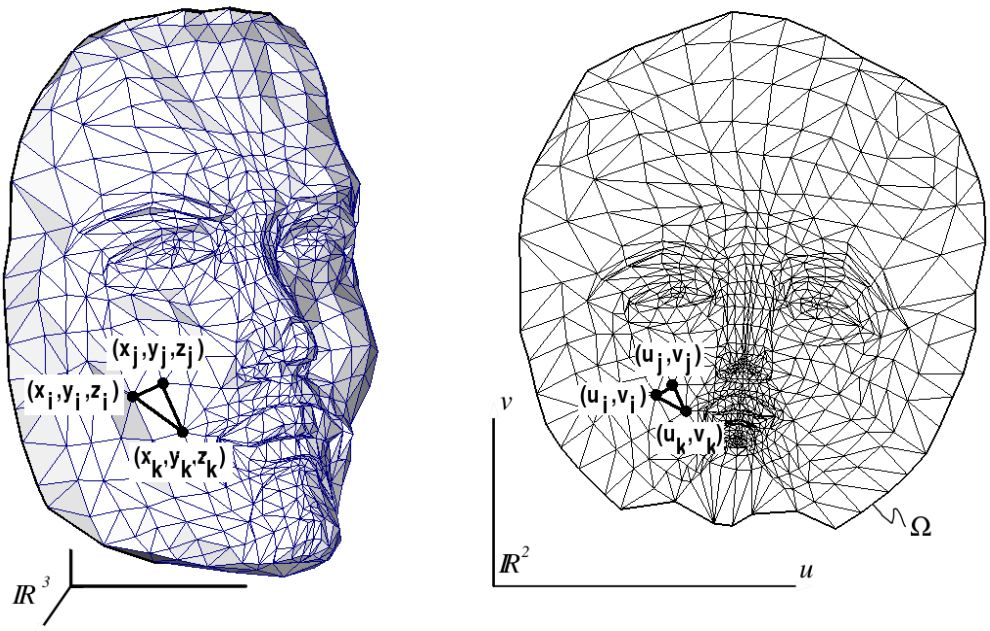


Figure 8.8: A parameterization of a triangulated surface can be defined as a piecewise linear function, determined by the coordinates (u_i, v_i) at each vertex (x_i, y_i, z_i).

8.3.1 Gradient in a Triangle

Distortion analysis, introduced in the previous section, involves the computation of the gradients of the parameterization as a function of the parameters u and v. In the case of a triangulated surface, the parameterization is a piecewise linear function. Therefore, the gradients are constant in each triangle.

Before studying the computation of these gradients, we need to mention that our setting is slightly different from the previous section. In our case, as previously mentioned, the 3D surface is given, and our goal is to construct the parameterization. In this setting, it seems more natural to characterize the inverse of the parameterization, i.e., the function that goes from the 3D surface (known) to the parametric space (unknown). This function is also piecewise linear. To port distortion analysis to this setting, it is possible to provide each triangle with an orthonormal basis X, Y, as shown in Figure **??** (and we can use one of the vertices $\mathbf{p}_i$ of the triangle as the origin). In this basis, we can study the inverse of the parameterization, that is to say the function that maps a point (X, Y) of the triangle to a point (u, v) in parameter space. The gradients of this function are given by:

$$\nabla u = \begin{pmatrix} \partial u/\partial X \\ \partial u/\partial Y \end{pmatrix} = \mathbf{M}_T \begin{pmatrix} u_i \\ u_j \\ u_k \end{pmatrix} = \frac{1}{2|T|} \begin{pmatrix} Y_j - Y_k & Y_k - Y_i & Y_i - Y_j \\ X_k - X_j & X_i - X_k & X_j - X_i \end{pmatrix} \begin{pmatrix} u_i \\ u_j \\ u_k \end{pmatrix} , \qquad (8.4)$$

where the matrix $\mathbf{M}_T$ is constant over triangle T, and $|T|$ denotes the area of T. Note that these gradients are different (but strongly related with) the gradients of the inverse function, manipulated in the previous section. The gradient of u (resp. v) intersects the iso-us (resp. the

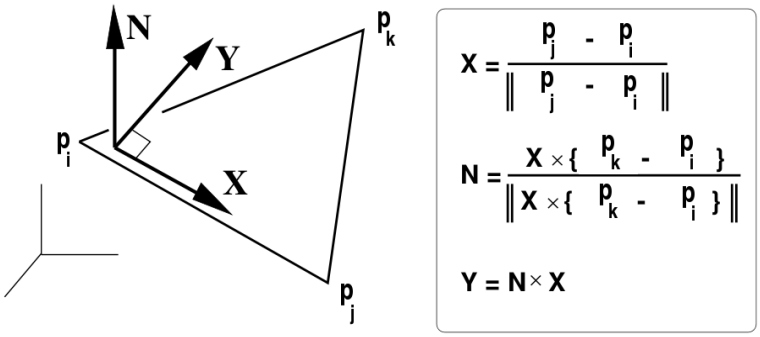


Figure 8.9: Local X, Y basis in a triangle.

iso-*v*s) with a right angle (instead of being tangent to them), and its norm is the inverse of the one computed in the previous section.

8.3.2 Piecewise Linear Distortion Analysis

Using this expression of the gradient, we can now conduct distortion analysis in a triangle. To do so, we first compute the first fundamental form $\mathbf{I}_T$, that is constant in triangle T:

$$\mathbf{I}_T = \mathbf{J}^T\mathbf{J} = \begin{pmatrix} E & F \\ F & G \end{pmatrix} . \tag{8.5}$$

As we have seen in the previous section, the lengths σ_1 and σ_2 of the axes of the anisotropy ellipse correspond to the singular values of $\mathbf{J}$, or to the square root of the eigenvalues of $\mathbf{I}_T$. Their expression can be found by computing the square roots of the zeros of the characteristic polynomial $\det(\mathbf{I}_T - \sigma\,\mathbf{Id})$, where $\mathbf{Id}$ denotes the identity matrix:

$$\begin{aligned} \sigma_1 &= \sqrt{1/2(E+G) + \sqrt{(E-G)^2 + 4F^2}} \\ \sigma_2 &= \sqrt{1/2(E+G) - \sqrt{(E-G)^2 + 4F^2}} \end{aligned} . \tag{8.6}$$

Before studying how to use these eigenvalues to minimize distortions, we will see a classical parameterization method.

8.4 Barycentric Maps

Barycentric maps are one of the most widely used methods to construct a parameterization of a triangulated surface. This methods is based on Tutte's barycentric mapping theorem [**?**], from graph theory, that states:

Given a triangulated surface homeomorphic to a disc, if the (u, v) coordinates at the boundary vertices are on a convex polygon, and if the coordinates of the internal vertices are a barycentric

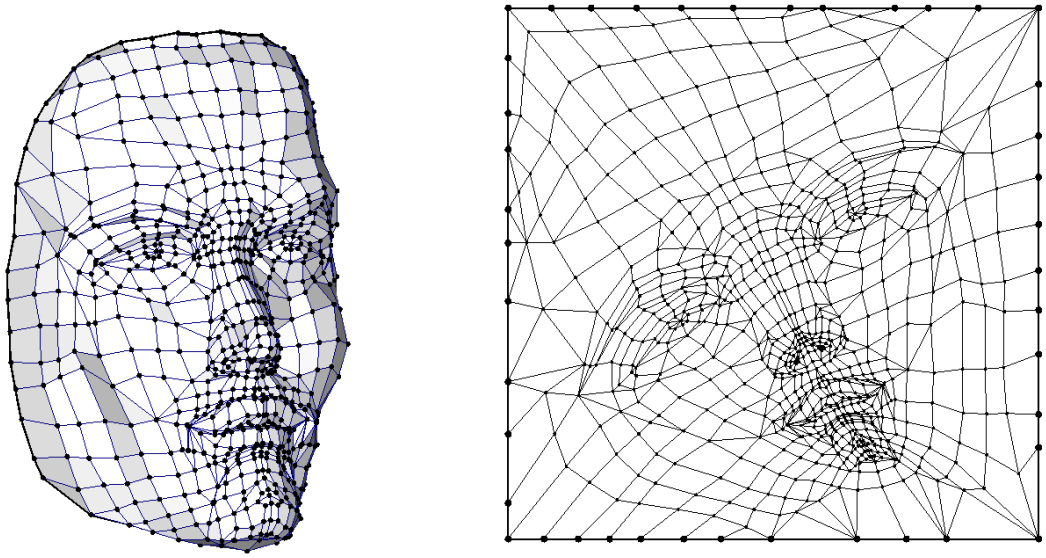


Figure 8.10: Parameterization with Floater's method. The parametric coordinates on the boundary of the surface are fixed on a convex polygon, and the interior ones are obtained by solving a linear system.

combination of their neighbors, then the (u, v) coordinates form a valid parameterization (without self-intersections). In formula, the second condition can be written as :

$$\forall i : \quad -a_{i,i} \begin{pmatrix} u_i \\ v_i \end{pmatrix} = \sum_{j \in N_i} a_{i,j} \begin{pmatrix} u_j \\ v_j \end{pmatrix} ,$$

where N_i denotes the set of vertices connected to vertex i by an edge, and where the coefficients $a_{i,j}$ satisfy:

$$\forall i \text{ internal vertex } : \quad \begin{cases} a_{i,j} > 0 \text{ if } i \neq j \\ a_{i,i} = -\sum_{j \neq i} a_{i,j} \end{cases} . \tag{8.7}$$

Michael Floater [**?**] had the idea to use this theorem – that *characterizes* a family of valid parameterizations – as a method to *construct* a parameterization. The idea consists in first fixing the vertices of the boundary on a convex polygon. Then, the coordinates at the internal vertices are found by solving Equation **??**. This means solving two linear systems $\mathbf{A}\mathbf{u} = \mathbf{u}_0$ and $\mathbf{A}\mathbf{v} = \mathbf{v}_0$, where the vectors $\mathbf{u}$ and $\mathbf{v}$ gather all the u (resp. v) coordinates at the internal vertices, and where the right hand side $\mathbf{u}_0$ (resp. $\mathbf{v}_0$) contains the coordinates at the vertices on the boundary. Figure **??** shows an example of a parameterization computed by this method (using weights $a_{i,j}$ given further).

The initial proof by Tutte uses sophisticated graph theory tools [**?**]. More recently, a simpler proof was established by Colin de Verdire [**?**]. Finally, a proof based on the notion of discrete one form was discovered [**?**]. Since it simply uses simple counting arguments, this latter proof is accessible without requiring the important graph theory background involved in the two other ones.

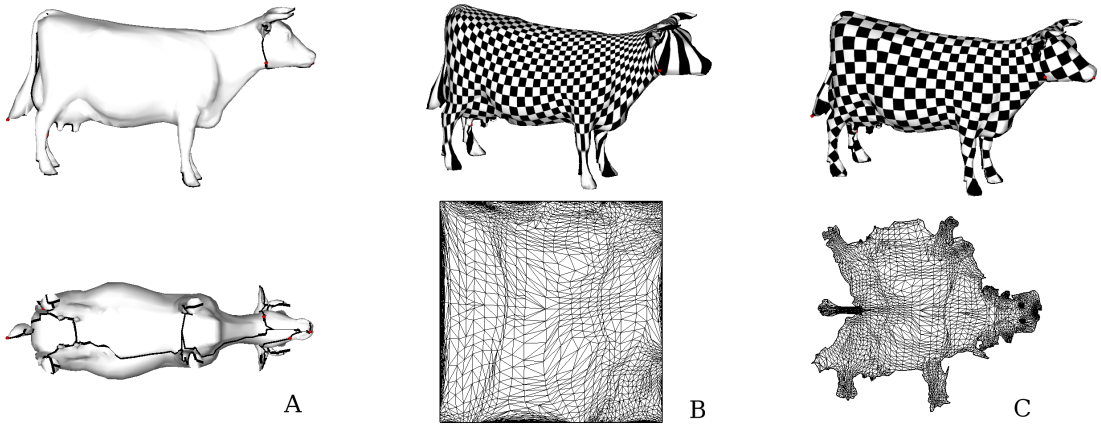


Figure 8.11: A: a mesh cut in a way that makes it homeomorphic to a disk, using the *seamster* algorithm [**?**]; B: Tutte-Floater parameterization obtained by fixing the boundary on a square; C: parameterization obtained with a free-boundary parameterization [**?**].

A possible valid choice for the coefficients $a_{i,j}$ is given by $a_{i,j} = 1$ if $i \neq j$ and $a_{i,i} = -|N_i|$, where $|N_i|$ denotes the number of neighbors of vertex i. However, this choice introduces distortions, that most applications need to avoid. For this reason, the next subsection introduces a means of choosing these weights that minimizes the distortions.

8.4.1 Discrete Laplacian

The Laplacian, or Laplace operator, is a generalization of the second order derivative for multivariate functions. In flat 2D space, this operator is defined by:

$$\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \ .$$

Intuitively, the Laplacian measures the regularity (or the irregularity) of a function. For instance, for a linear function, the Laplacian is equal to zero. The Laplacian can be generalized to curved surfaces, and the generalized form is called the Laplace-Beltrami operator. Chapter **??** reviews several discrete versions of this operator. A so-defined discrete Laplacian is a matrix $(a_{i,j})$ whose non-zero pattern corresponds to the connectivity of the mesh, and that satisfies $a_{i,i} = -\sum_{i \neq j} a_{i,j}$. It is then possible to use the discrete Laplacian to define the coefficients $a_{i,j}$ used in Floater's method. We will elaborate further on the link between the discrete Laplacian and parameterization in Section **??**.

However, it should be noticed that for some meshes with obtuse angles, the coefficients of the discrete Laplacian may become negative. This violates the requirements of Tutte's theorem, such that the validity of the mapping can no longer be guaranteed. More recently, Floater discovered another definition of weights (*Mean Value Coordinates*) [**?**] that does not suffer from this problem (the coefficients are always positive).

Therefore, Tutte's theorem combined with mean value weights provides a provably correct way of constructing a valid parameterization for a disk-like surface. However, for some surfaces, the necessity to fix the boundary on a convex polygon may be problematic (cf. Figure **??**), for the following reasons: (1) in general, it is difficult to find a "natural" way of fixing the

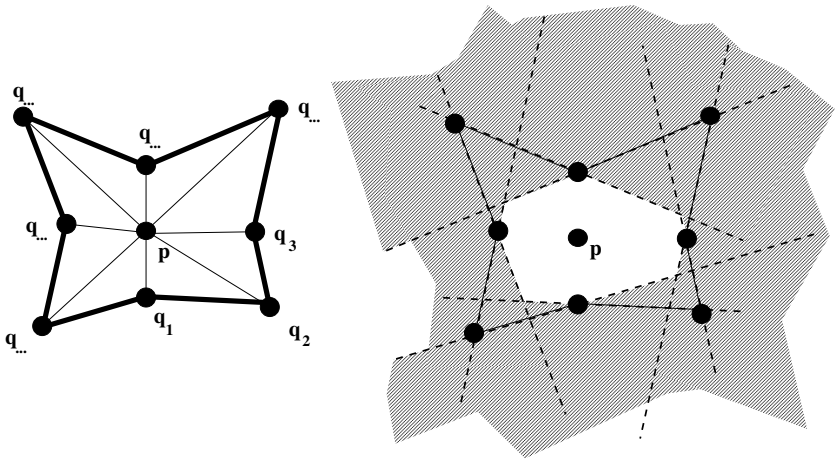


Figure 8.12: Left: to avoid triangle flips, each vertex $\mathbf{p}$ is constrained to remain in the kernel of the polygon defined by its neighbors $\mathbf{q}_i$; Right: the kernel of a polygon (white) is defined by the intersection of the half-planes defined by the support lines of its edges (dashed).

boundary on a convex polygon, and (2) for some surfaces, the shape of the boundary is far from convex. Therefore the obtained parameterization shows high distortions. Even if one can imagine different ways of improving the result shown in the Figure, the so-obtained parameterization will be probably not as good as the one shown in Figure **??**-C, that better matches what a tanner would expect for such a mesh. For these reasons, the next section studies the methods that can construct parameterizations with free boundaries.

8.5 Setting the Boundary Free

In the second half of the 90's, Floater's method [**?**], based on Tutte's theorem [**?**], was well known by the community and applied to a wide class of problems. The advantages of this method are its strong theoretical guarantees and its ease of implementation. However, the necessity to constrain the boundary on a convex polygon limits the efficiency of some applications. For this reason, the community started to investigate methods that do not suffer from this limitation, and that minimize the distortions in a similar way. This section reviews these methods, using the formalism introduced in Section **??**. However, before going further, we need to warn the reader about a possible source of confusion:

- Half of the methods study the function that goes from the surface to the parametric space (as in the previous section). This is justified by the fact that the (u, v) coordinates are unknown. Therefore, it is more natural to go from the known world (the surface) to the unknown world (the parameter space).
- The other half of the methods use the inverse convention, and study the function that goes from parameter space to the surface (as in Section **??**). This is justified by the fact that it makes the formalism compatible with classical differential geometry books [**?**] that use this convention.

Armed with the definition of distortion analysis, we can now proceed to review several methods, and express them in a common formalism. Note that in the literature, one needs to take care

of identifying whether the surface → parametric-space function or parametric-space → surface function is used. Before evoking these methods, we give two more precisions:

- To avoid triangle flips, some of the methods constrain each vertex $\mathbf{p}$ to remain in the kernel of the polygon defined by its neighbors $\mathbf{q}_i$. This notion is illustrated in Figure **??**. To compute the kernel of a polygon, it is for instance possible to apply Sutherland and Hogdman's re-entrant polygon clipping algorithm to the polygon (clipped by itself). The algorithm is described in most general computer graphics books [**?**];
- Since they are based on the eigenvalues of the first fundamental form, the objective functions involved in distortion analysis are often non-linear, and therefore difficult to minimize in an efficient way. To accelerate the computations, a commonly used technique consists in representing the surface in a multi-resolution manner, based on Hoppe's *Progressive Mesh* data structure [**?**]. The algorithm starts by optimizing a simplified version of the object, then introduces the additional vertices and optimizes them by iterative refinements.

Now that we have seen the general notions related with distortion analysis and the particular aspects that concern the optimization of objective functions involved in distortion analysis, we can review several classical methods that belong to this category.

8.5.1 Green-Lagrange Deformation Tensor

Historically, to minimize the distortions of a parameterization, one of the first methods was developed by Maillot, Yahia, and Verroust [**?**]. The main idea behind their approach consists in minimizing a matrix norm of the Green-Lagrange deformation tensor. This notion comes from mechanics, and measures the deformation of a material. Intuitively, we know that if the metric tensor $\mathbf{I}$ is equal to the identity matrix $\mathbf{Id}$, then we have an isometric parameterization. The Green-Lagrange deformation tensor is given by $\mathbf{L} = \mathbf{I} - \mathbf{Id}$ and measures the "non-isometry" of the parameterization.

8.5.2 MIPS

Hormann and Greiner's MIPS (Most Isometric Parameterization of Surfaces) method [**?**] was to our knowledge the first mesh parameterization method that computes a natural boundary. This method is based on the minimization of the ratio between the two lengths of the axes of the anisotropy ellipse. This corresponds to the 2-norm of the Jacobian matrix:

$$K_2(\mathbf{J}_T) = \|\mathbf{J}_T\|_2 \|\mathbf{J}_T^{-1}\|_2 = \sigma_1/\sigma_2 \ .$$

Since minimizing this energy is a difficult numerical problem, Hormann and Greiner have replaced the 2-norm $\|.\|_2$ by the Frobenius norm $\|.\|_F$, i.e., the square root of the sum of the squared singular values:

$$K_F(\mathbf{J}_T) \quad = \quad \|\mathbf{J}_T\|_F \|\mathbf{J}_T^{-1}\|_F \quad = \quad \frac{\text{trace}(\mathbf{I}_T)}{\det(\mathbf{J}_T)} \ .$$

As can be seen, fortunate cancellations of terms yield a simple expression in the end. The final expression corresponds to the ratio between the trace of the metric tensor and the determinant of the Jacobian matrix. As indicated in the original article, this value can also be interpreted as

Figure 8.13: Some results computed by stretch L_2 minimization (parameterized models courtesy of Pedro Sander and Alla Sheffer).

the *Dirichlet energy per parameter-space area*: the term trace($\mathbf{I}_T$) corresponds to the Dirichlet energy, and the Jacobian det($\mathbf{J}_T$) to the ratio between triangle's area in 3D and in parameter space.

8.5.3 Stretch Minimization

Motivated by texture mapping applications, Sander et al. [**?**] studied the way a signal stored in parameter space is distorted when it is texture-mapped onto the surface (by applying the parameterization). For this reason, their formalism uses the inverse function, that maps the parametric space onto the surface. However, it is easy to check that this simply means replacing σ_1 with $1/\sigma_2$ (resp. σ_2 with $1/\sigma_1$) in the computations.

A possible way of characterizing the distortions of a texture is to consider a point and a direction in parameter space and analyze how the texture is deformed along that direction. Sander et al. called this value the "stretch". This exactly corresponds to the notion of directional derivative, which we introduced in Section **??**. For a triangle T, they defined two energies that correspond to the average value of the stretch for all directions (stretch $L_2(T)$):

$$L_2(T) = \sqrt{\left((1/\sigma_1)^2 + (1/\sigma_2)^2\right)/2}\,.$$

The local energies of each triangle T are combined into a global energy $L_2(\mathcal{S})$ defined by:

$$L_2(\mathcal{S}) = \sqrt{\frac{\sum_T |T| L_2(T)}{\sum_T |T|}}\,.$$

Figure **??** shows some results computed with this approach. This formalism is particularly well suited for texture mapping applications, since it minimizes the distortions that are responsible of the visual artifacts that this type of application wants to avoid. Moreover, a simple modification of this method allows the contents of the texture to be taken into account, and therefore to define a signal-adapted parameterization [**?**].

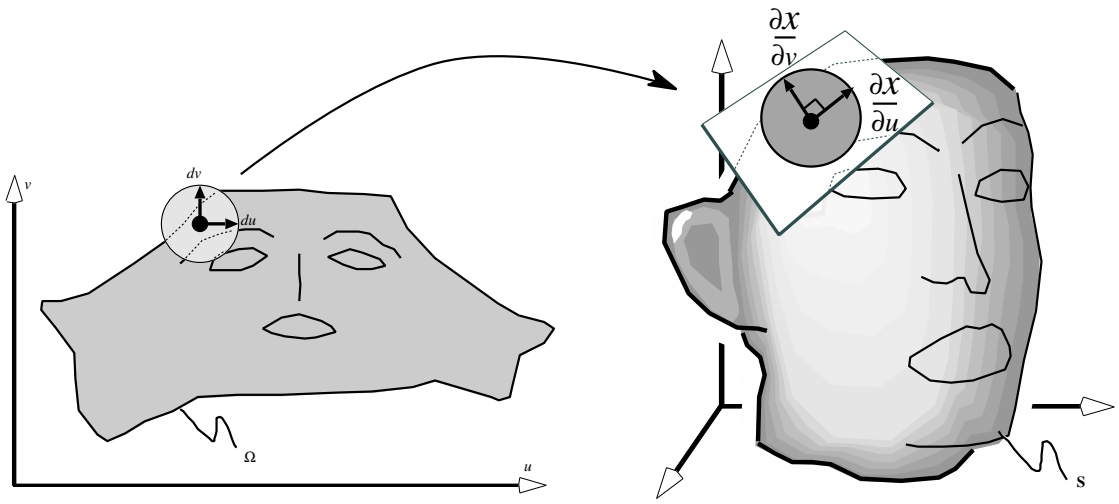


Figure 8.14: A conformal parameterization transforms an elementary circle into an elementary circle.

8.5.4 Conformal Methods

Conformal methods are related with the formalism of complex analysis. The involved conformality condition defines a criterion with sufficient "rigidity" to offer good extrapolation capabilities, that can compute natural boundaries. The reader interested with this formalism may read the excellent book by Stanley Needham [**?**].

As seen in this section, distortion analysis, introduced in Section **??**, plays a central role in the definition of (non-distorted) parameterization methods. We now focus on a particular family of methods, for which the anisotropy ellipse is a circle for all point of the surface. As shown in Figure **??**, this also means that the two gradient vectors $\mathbf{x}_u$ and $\mathbf{x}_v$ are orthogonal and have the same norm. The condition can also be written as $\mathbf{x}_v = \mathbf{n} \times \mathbf{x}_u$, where $\mathbf{n}$ denotes the normal vector. Interestingly, if a parameterization is conformal, this is also the case for the inverse function (since the Jacobian matrix of the inverse is equal to the inverse of the Jacobian matrix). Intuitively, if the iso-u,v curves are orthogonal, it is also the case of their normal vectors in the tangent plane. Finally, conformality also means that the Jacobian matrix is composed of a rotation and a scaling (in other words, a *similarity* transform). Therefore, conformal mappings locally correspond to similarities. We now review different methods that compute a conformal parameterization.

LSCM

In contrast with the exposition of the initial paper [**?**], we will present the method in terms of simple geometric relations between the gradients. We will then elaborate with the complex analysis formalism, and establish the relation with other methods.

The LSCM method (Least Squares Conformal Maps) simply expresses the conformality condition of the functions that maps the surface to parameter space. We now consider one of the

triangles of the surface, provided with an orthonormal basis (X, Y) of its support plane. In this context, conformality can be written as :

$$\nabla v = \mathrm{rot}_{90}(\nabla u) = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \nabla u \ , \tag{8.8}$$

where rot_{90} denotes the counter-clockwise rotation of 90 degrees.

Using the expression of the gradient in a triangle (derived at the end of Section **??**), Equation **??**, which characterizes piecewise linear conformal maps, becomes :

$$\mathbf{M}_T \begin{pmatrix} v_i \\ v_j \\ v_k \end{pmatrix} - \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \mathbf{M}_T \begin{pmatrix} u_i \\ u_j \\ u_k \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \ ,$$

where $\mathbf{M}_T$ is given by Equation **??**.

In the continuous setting, Riemann proved that any surface admits a conformal parameterization. However, in our specific case of piecewise linear functions, only developable surfaces admit a conformal parameterization. For a general (non-developable) surface, LSCM minimizes an energy E_{LSCM} that corresponds to the "non-conformality" of the application, and called the *discrete conformal energy*:

$$E_{LSCM} = \sum_{T=(i,j,k)} |T| \left\| \mathbf{M}_T \begin{pmatrix} v_i \\ v_j \\ v_k \end{pmatrix} - \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \mathbf{M}_T \begin{pmatrix} u_i \\ u_j \\ u_k \end{pmatrix} \right\|^2 \ . \tag{8.9}$$

We have considered conformal maps from the point of view of the gradients. In the next section, we exhibit relations between conformal maps and harmonic functions. This also shows some connections with Floater's barycentric mapping method and with its more recent generalizations.

Conformal maps and harmonic maps

Conformal maps play a particular role in complex analysis and Riemannian geometry. The following system of equations characterizes conformal maps:

$$\begin{aligned} \frac{\partial v}{\partial x} &= -\frac{\partial u}{\partial y} \ , \\ \frac{\partial v}{\partial y} &= \frac{\partial u}{\partial x} \ . \end{aligned}$$

This system of equations is known as Cauchy-Riemann equations They play a central role in complex analysis, since they characterize differentiable complex functions (also called analytic functions).

Another interesting property of complex differentiable functions is that their order-1 differentiability makes them differentiable at any order. We can then use the Cauchy-Riemann equations to compute the order 2 derivatives of u and v, and establish interesting relations with the Laplacian (see Section **??**):

$$\begin{aligned} \Delta u &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \ , \\ \Delta v &= \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} = 0 \ . \end{aligned}$$

In other words, the real part and the imaginary part of a conformal map are two harmonic functions (i.e., two functions with zero Laplacian). This justifies the idea of using the discrete Laplacian to define Floater's weights, mentioned in the previous section. This is the point of view adopted by Desbrun et al. to develop their conformal parameterization method [**?**], nearly equivalent to LSCM. Thus, Desbrun et al. compute two harmonic functions while letting the boundary evolve. On the boundary, a set of constraints enforce the conformality of the parameterization, and introduce a coupling term between the u's and the v's.

Another way of considering both approaches, mentioned by Pinkall and Polthier [**?**], and probably at the origin of Desbrun et al.'s intuition, is given by Plateau's problem [**?**, **?**]. Given a closed curve, this problem concerns the existence of a surface with minimum area, such that its boundary matches the closed curve. To minimize the area of a surface, Douglas [**?**] and Rado [**?**], and later Courant [**?**] considered Dirichlet's energy (i.e., the integral of the squared norm of the gradients), easier to manipulate. A discretization of this energy was proposed by Pinkall and Polthier [**?**], with the aim of giving a practical solution to Plateau's problem in the discrete case. Dirichlet's energy differs from the area of the surface. The difference is a term that depends on the parameterization, called the *conformal energy*. The conformal energy is equal to zero if the parameterization is conformal. The relation between these three quantities is explained below:

$$\underbrace{\int_S \det(\mathbf{J})ds}_{\text{area of the surface}} = \underbrace{\frac{1}{2}\int_S \|\mathbf{x}_u\|^2 + \|\mathbf{x}_v\|^2 ds}_{\text{Dirichlet's energy}} - \underbrace{\frac{1}{2}\int_S \|\mathbf{x}_v - \text{rot}_{90}(\mathbf{x}_u)\|^2}_{\text{conformal energy}} .$$

This relation is easy to prove, by expanding the integrated terms. Therefore, LSCM minimizes the conformal energy, and Desbrun *et.al*'s method minimize Dirichlet's energy. Since the difference between these two quantities corresponds to the (constant) area of the surface, both methods are equivalent.

All the methods mentioned above are based on relations between the gradients, the Jacobian or the first fundamental form of the parameterization. We also refer the reader to [**?**], that provides another way of "setting the boundary free", by separating computations into several steps involving simpler (linear) computations. The notion of derivative and its connection with geometry (or differential geometry) play a central role in the methods mentioned above. For this reason, they can be qualified as *analytical* methods. In the next section, we focus on *geometric* methods, that consider the shape of the triangles.

8.5.5 Geometric Methods

The ease of implementation of analytical methods (especially the quadratic ones) favored their diffusion, in both the scientific community and the industrial word. However, the non-linear methods need a progressive mesh, that makes them quite delicate to tune, and the quadratic methods, with their two pinned vertices, may generate results that are unbalanced in terms of distortions (Figure **??**), if the input surface has high Gaussian curvature. For this reason, we focus on geometric methods, that do not suffer from these problems. We will review ABF (Angle Based Flattening). Note that one may also classify in this category circle packings [**?**] and circle patterns [**?**], not covered here. We also cite a very recent result [**?**], that gives a deep understanding of how angular defect and deformations relate.

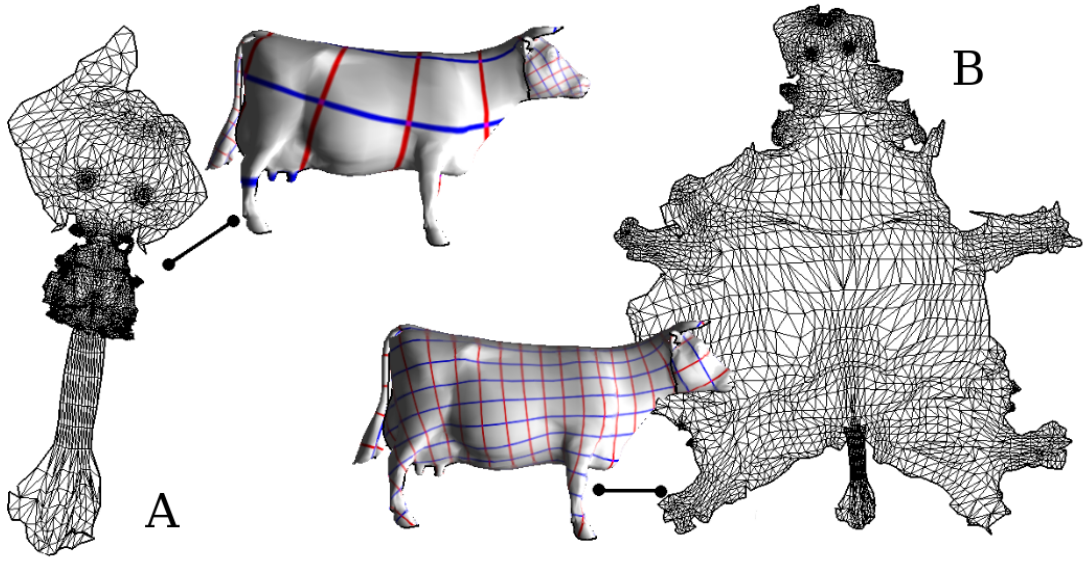


Figure 8.15: For surfaces that have a high Gaussian curvature, conformal methods may generate highly distorted results, different from what the user might expect (A). The ABF method and its derivatives better balances the distortions, and gives better results (B).

The ABF method (Angle Based Flattening), developed by Sheffer et al. [**?**], is based on the following observation: the parameter space is a 2D triangulation, uniquely defined by all the angles at the corners of the triangles (modulo a similarity in parameter space). This simple remark made the authors reformulate the parameterization problem – finding (u_i, v_i) coordinates – in terms of angles, that is finding the angles α_i^t, where α_i^t denotes the angle at the corner of triangle t incident to vertex i.

The energy minimized by ABF is given by :

$$E(\alpha) = \sum_{t \in T} \sum_{k=1}^{3} \frac{1}{w_k^t} (\alpha_k^t - \beta_k^t)^2 \ , \tag{8.10}$$

where the α_k^t's are the unknown 2D angles, and where the β_k^t's denote the "optimal" angles, measured on the 3D mesh. The weights w_k^t are set to $(\beta_k^t)^{-2}$ to measure a *relative* angular distortions rather than an *absolute* one.

To ensure that the 2D angles define a valid triangulation, a set of constraints needs to be satisfied. These constraints are introduced in the formulation using the Lagrange method:

- Validity of the triangles (for each triangle t):

$$\forall t \in T, \quad C_{Tri}(t) = \alpha_1^t + \alpha_2^t + \alpha_3^t - \pi = 0 \ . \tag{8.11}$$

- Planarity (for each internal vertex v):

$$\forall v \in V_{int}, \quad C_{Plan}(v) = \sum_{(t,k) \in v^*} \alpha_k^t - 2\pi = 0 \ , \tag{8.12}$$

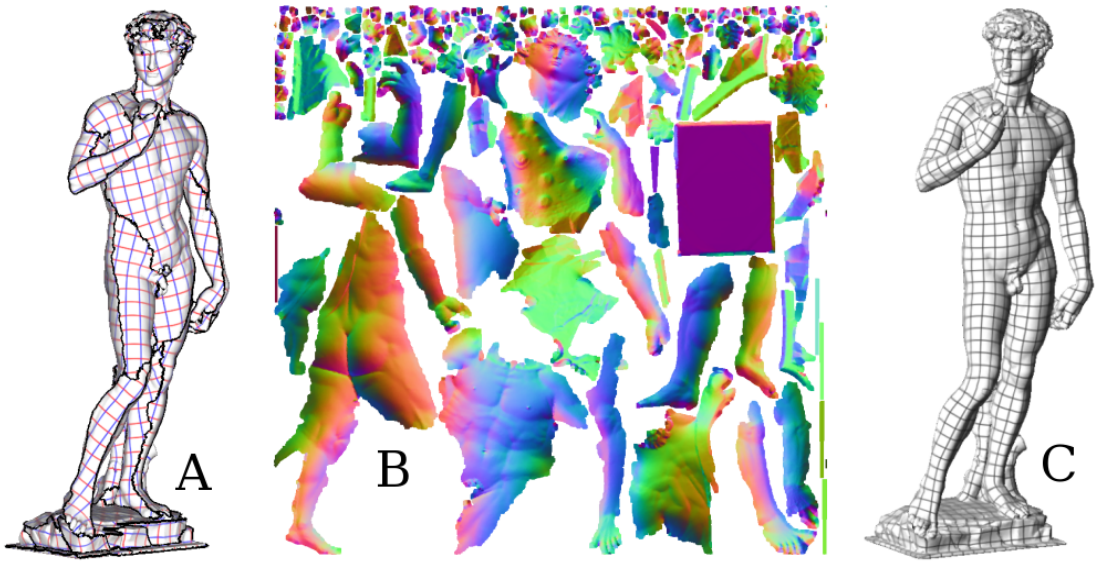


Figure 8.16: A,B: Parameterization methods for disk-topology combined with segmentation algorithms can create a texture atlas from a shape of arbitrary topology. However, the large number of discontinuities can be problematic for the applications. C: Global parameterization algorithms do not suffer from this problem. (Data courtesy of the Digital Michelangelo Project, Stanford).

where V_{int} denotes the set of internal vertices, and where v^* denotes the set of angles incident to vertex v.

- Reconstruction (for each internal vertex) — this constraints ensures that the edge shared by a pair of triangles has the same length:

$$\forall v \in V_{int}, \quad C_{Len}(v) = \prod_{(t,k)\in v^*} \sin \alpha^t_{k\oplus 1} - \prod_{(t,k)\in v^*} \sin \alpha^t_{k\ominus 1} = 0 \,. \tag{8.13}$$

The indices $k \oplus 1$ and $k \ominus 1$ denote the next and previous angle in the triangle. Intuitively, note that the product $\sin \alpha^t_{k\oplus 1} \sin \alpha^t_{k\ominus 1}$ corresponds to the product of the ratio between the lengths of two consecutive edges around vertex k. If they do not match, it is the possible to "turn around" vertex k without "landing" on the starting point.

Sheffer and de Sturler [**?**] compute a stationary point of the Lagrangian of the constrained quadratic optimization problem by using Newton's method. An improvement of the numerical solution mechanism was proposed [**?**]. More rencently, Zayer *et.al* proposed a linearized approximation [**?**], that solves for the approximation error. The dual formulation leads to a least-norm problem, that simply means solving a linear system.

8.6 Global Parameterization

As seen in the previous section, parameterization methods can put a 3D shape with disk topology in one-to-one correspondence with a 2D domain. For a shape with arbitrary topology, it is possible to decompose the shape into a set of charts, using a segmentation algorithm (e.g. VSA [**?**]).

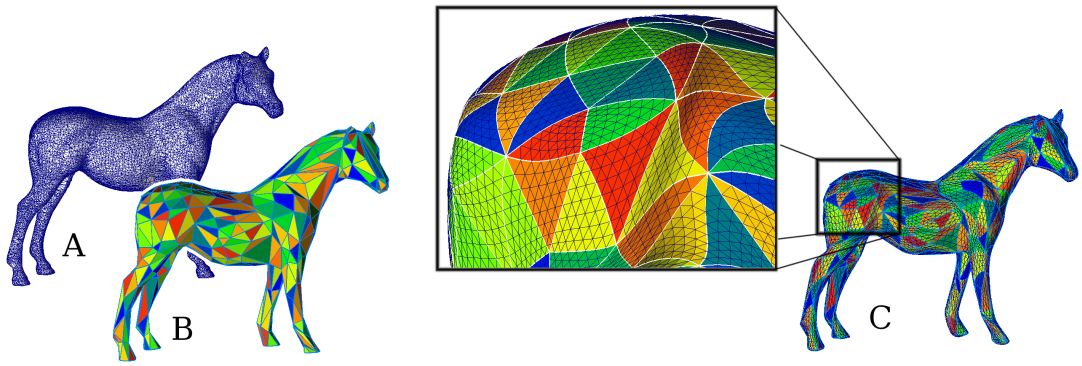


Figure 8.17: The MAPS method and its derivatives compute a global parameterization by decomposing the initial surface (A) into a set of triangular charts (B) and regularly re-samples the geometry in the parameter space of these charts (C).

Each chart is then parameterized (see Figure **??**-A,B). Even if this solution works, it is not completely satisfactory: why one should "damage" the surface just to define a coordinate system on it? From the application point of view, chart boundaries are difficult to handle in re-meshing algorithms, and introduce artifacts in texture mapping applications. For this reason, we focus in this section on *global* parameterization algorithms, that do not require segmenting the surface. (Figure **??**-C). Since they play a central role in remeshing algorithms, global parameterization method are also described in Chapter **??**.

8.6.1 Base Complex

To compute such a global parameterization, the geometry processing community first developed methods that operate by segmenting / parameterizing / and resampling the object. To our knowledge, this idea was first developed in the MAPS method [**?**] (Multiresolution Adaptive Parameterization of Surfaces). As shown in Figure **??**, this method starts by partitioning the initial object (Figure **??**-A) into a set of triangular charts, called the *base complex* (Figure **??**-C). Then, a parameterization of each chart is computed, and the object is regularly re-sampled in parametric space (Figure **??**-C). Further refinements of the method improved the inter-chart continuity [**?**], formalized by the notion of *transition function*, explained further in this section. This representation facilitates defining hierarchical representations and implementing multiresolution processing tools on top of it [**?**].

This family of methods use a set of triangular charts to define the base complex. For some applications, such as texture mapping, or surface approximation with tensor-product splines, it is preferred to use a base complex composed of quadrilaterals. The difference seems subtle at first sight, but automatically constructing a good quadrilateral base complex is still an open problem. A variant of the MIPS [**?**] method, applied to a quadrilateral base complex, was proposed [**?**]. The method lets the user interactively define the base complex. More recently, advances to automate the process were made, as shown further in this section.

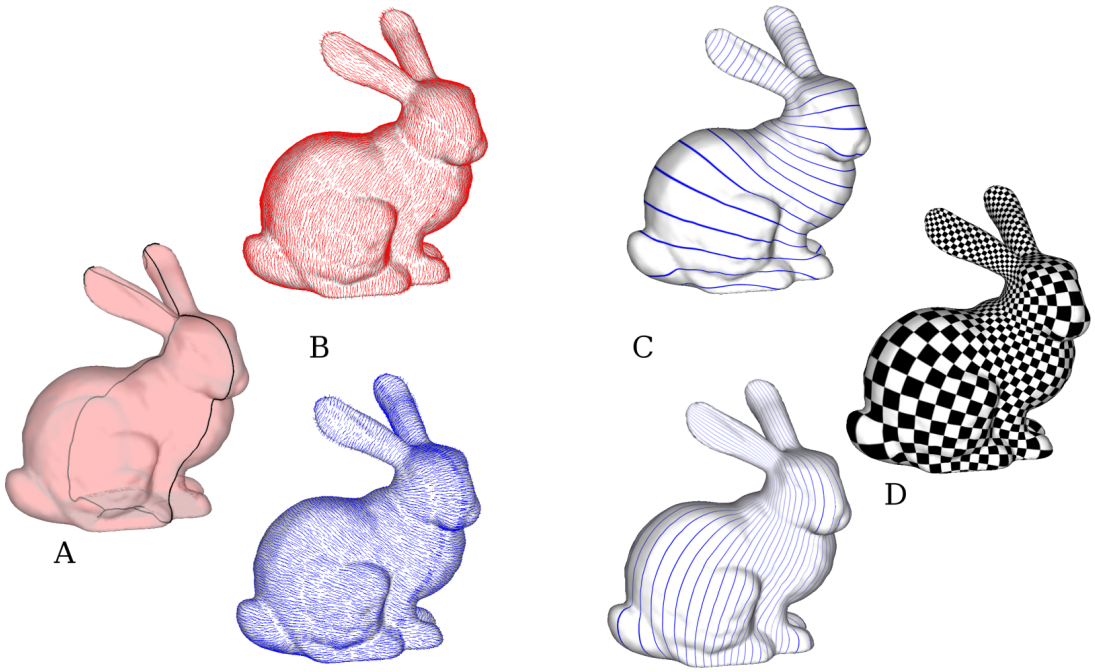


Figure 8.18: The method developed by Gu and Yau to construct a differential manifold first computes a homology basis (A), then deduces a co-homology basis and finds the (unique) harmonic one form in each co-homology class (B). Finally, the u and v potentials are obtained by integrating these harmonic one-forms (C), that together define a holomorphic function (D) (data courtesy of Stanford, parameterization courtesy of X. Gu)

8.6.2 Methods Based on Co-homology

As shown in Figure **??**, Gu and Yau used notions from exterior calculus to compute a global conformal parameterization on a surface of arbitrary genus [**?**, **?**]. To do so, they compute a holomorphic function (i.e., the generalization of conformal functions mentioned in Section **??**), based on an important theorem that states that each co-homology class contains a unique harmonic one-form. We have already seen that the coordinates of a conformal map are two harmonic functions. Similarly, a holomorphic function is composed of two conjugate (i.e., orthogonal) harmonic one-forms. Then, their method operates as follows: they first compute a homology basis of the surface (using for instance Erickson's method [**?**]) (A), then they deduce a co-homology basis, and find a pair of conjugate harmonic one-forms (B). Finally, they integrate the one-forms to find the parameterization (C).

Some authors have proposed methods that directly compute the parameterization, based on modified Floater conditions. For instance, Steiner and Fischer have proposed to make the object equivalent to a disk using a cut graph, and insert translation vectors in Tutte's conditions related to the vertices located on the cut graph [**?**]. Tong *et.al* developed independently the same idea [**?**], using the formalism of exterior calculus, and introducing singularities of fractional index. Such singularities of fractional index can be elegantly taken into account by a structure called *quad-cover* [**?**]. See also Chapter **??** for examples of these latter two methods.

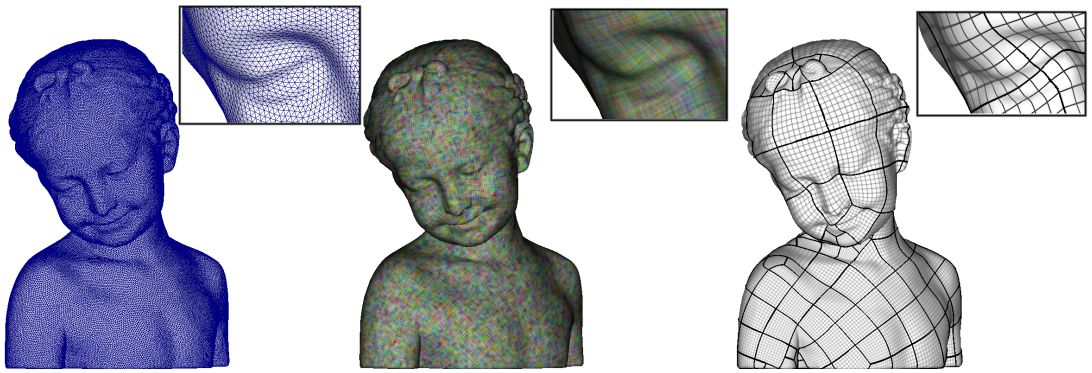


Figure 8.19: From a triangulated mesh (left), the Periodic Global Parameterization method starts by smoothing a vector field (center) and then computes a parameterization such that the gradient vectors are aligned with the vector field (right).

8.6.3 Periodic Global Parameterization

The Periodic Global Parameterization method [**?**], shown in Figure **??**, aims at letting the singularities naturally emerge from the optimization of the parameterization. As a consequence, it is not possible to determine the homology basis in advance. As in Alliez et al.'s anisotropic remeshing method [**?**], the method first computes a guidance vector field by smoothing the principal directions of curvature. Then the difficulty is to allow the coordinates to wind around the features of the object. To do so, the method uses the natural periodicity of the sine and cosine function. The complete optimization problem is restated in terms of new variables, that correspond to the sine and cosine of the actual coordinates. The main difficulty is that the method generates invalid vertices, edges and triangles around the singularities. Therefore it requires a post-processing step. At this point, we can either use methods based on co-homology (but they require manual intervention), or PGP (but it requires an inelegant post-processing to fix the singularities). However, another category of methods, based on eigenvector computations, seem a promising research avenue to define both automatic and simple methods.

8.6.4 Spectral Methods

Spectral methods study the eigenfunctions of operators (or eigenvectors of matrices in the discrete setting). Several reviews on this topic are available [**?**, **?**], and we also give some web references on the following webpage http://alice.loria.fr/publications. Spectral methods are also mentioned in Chapter **??** of these course notes, that deals with mesh smoothing.

We focus on the Laplace operator, that plays a fundamental role in conformal mesh parameterization, as shown in Section **??**.

Before elaborating on the eigenfunctions, we give more details about the Laplacian and its generalizations. The Laplacian plays a fundamental role in physics and mathematics. In $\mathbb{R}^n$, it is defined as the divergence of the gradient:

$$\Delta = \text{div grad} = \nabla \cdot \nabla = \sum_i \frac{\partial^2}{\partial x_i^2} .$$

Intuitively, the Laplacian generalizes the second order derivative to higher dimensions, and is a characteristic of the irregularity of a function as $\Delta f(\mathbf{x})$ measures the difference between $f(\mathbf{x})$

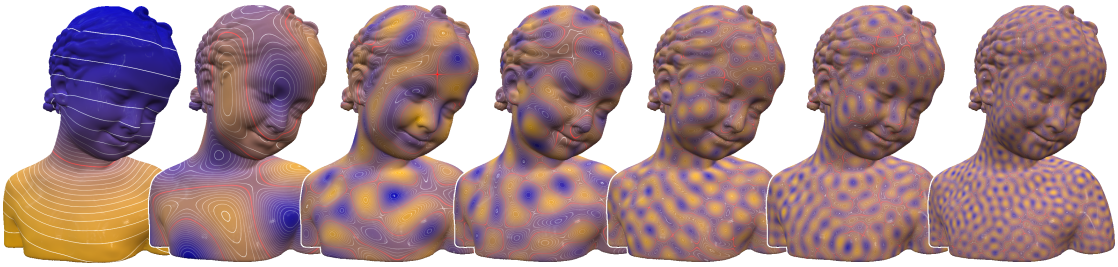


Figure 8.20: Some of the eigenfunctions of the Laplace operators.

and its average in a small neighborhood of $\mathbf{x}$. Generalizing the Laplacian to curved surfaces require complex calculations, that can be greatly simplified by a mathematical tool called exterior calculus (EC). See the course notes [**?**] for more details about exterior calculus.

The eigenfunctions and eigenvalues of the Laplacian on a (manifold) surface $\mathcal{S}$, are all the pairs (H^k, λ_k) that satisfy $-\Delta H^k = \lambda_k H^k$. The "$-$" sign is here required for the eigenvalues to be positive. On a closed curve, the eigenfunctions of the Laplace operator define the function basis (sines and cosines) of Fourier analysis[4]. On a square, they correspond to the function basis of the DCT (Discrete Cosine Transform), used for instance by the JPEG image format. Finally, the eigenfunctions of the Laplace-Beltrami operator on a sphere define the Spherical Harmonics basis. Figure **??** shows how they look like for a more general object. Since it generalizes spherical harmonics to arbitrary manifolds, this function basis is naturally called the Manifold Harmonics Basis (MHB) [**?**]. This tech-report gives their formal definition using the finite element formalism, and explains how to efficiently compute the MHB. In the discrete setting, one can approximate the eigenfunctions by computing the eigenvectors of a discrete Laplacian (see Section **??**). In some cases, it is even possible to replace it with a combinatorial Laplacian, that only takes graph connectivity into account [**?**].

In the context of data analysis, the MDS method (*multidimensional scaling*) [**?**] was introduced, to compute an embedding that best approximates given distances between the vertices of a graph. Multidimensional scaling simply minimizes an objective function that measures the deviation between the geodesic distances in the initial space and the Euclidean distances in the embedding space (GDD for Geodesic Distance Deviation), by computing the eigenvectors of the matrix $D = (d_{i,j})$ where $d_{i,j}$ denotes the geodesic distance between vertex i and vertex j. Isomaps and Multidimensional scaling were used to define parameterization algorithms in [**?**], and more recently in the *ISO-charts* method [**?**], used in Microsoft's DirectX combined with the packing algorithm presented in [**?**]. Interestingly, the ISO-charts method uses MDS for both segmenting the model and parameterizing the charts. This provides a nice and coherent theoretical framework, that can be relatively easily translated into efficient implementations.

However, the spectral parameterization methods listed above still need to partition the mesh into charts. More recently, Dong et al. used the Laplacian to decompose a mesh into quadrilaterals [**?**, **?**], in a way that facilitates constructing a globally smooth parameterization. As shown in Figure **??**, their method first computes one eigenfunction of the Laplacian (the 38^{th} in this example), then extract the Morse complex of this function, filters and smooths the Morse

[4] This is easy to check by noticing that in 1D, the Laplace operator corresponds to the standard second order derivative. The eigenfunctions are simply $\sin(\omega t)$ (resp. cos) associated with the eigenvalues $-\omega^2$.

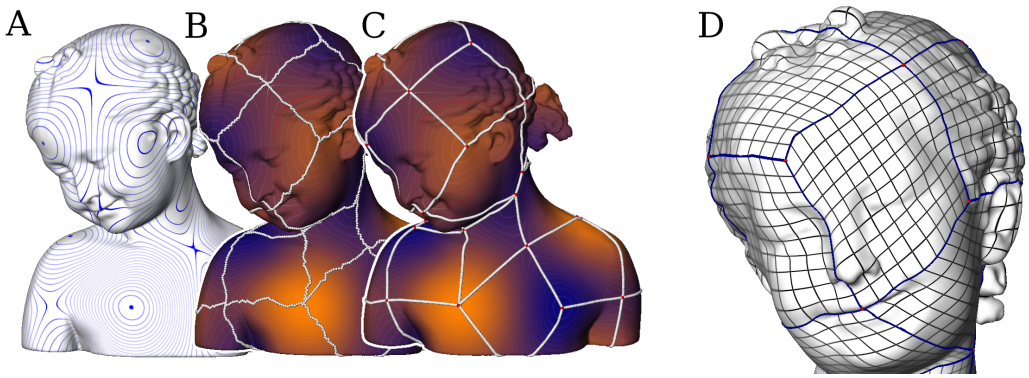


Figure 8.21: Spectral Surface Quadrangulation first computes a Laplace eigenfunction (A), then extracts its Morse complex (B), smooths it (C) and uses it to partition the mesh into quads, that can be parameterized (D).

complex, and uses it to partition the mesh into quads. These quads are parameterized, and inter-chart smoothness can be further optimized using global relaxation [**?**, **?**].

8.7 Open Issues

To conclude this section on mesh parameterization, we list some open issues. After Floater's initial article in 1995, time and effort was devoted to the very specific issue of mesh parameterization. However, the "mesh parameterization" story is still unfinished, since important problems still remain open:

- **Singularities** Automatically placing the singularities in global parameterization methods is still an open issue;
- **Negative cotangents** The possibly negative cotangents obtained with the discretization of the Laplacian are still a problem for some parameterization methods. A deeper understanding of what we loose in the discretization may help solving this issue;
- **Real-world meshes (1)** Most meshes manipulated in the industry are quite different from the scanning repositories used by academic research (AIM@SHAPE, Stanford). Most of them have creases. For this reason, methods based on differential geometry cannot be directly applied, since the differential quantities they estimate are undefined on the creases;
- **Real-world meshes (2)** Moreover, those meshes often have invalid topology (holes, duplicated surfaces ...) and/or ill-shaped triangles. Designing numerically robust methods is also an issue of paramount importance for the industrial applications;
- **3D hexahedral meshing** This is the natural generalization of quad-remeshing to 3D volumes. Exterior calculus (discrete or continuous) may be the right formalism to tackle this difficult issue. In particular, not only singular points may appear, but one may also encounter singular curves and singular plates.

9 Mesh Decimation

Mesh decimation describes a class of algorithms that transform a given polygonal mesh into another mesh with fewer faces, edges and vertices [**?**]. The decimation procedure is usually controlled by user defined quality criteria which prefer meshes that preserve specific properties of the original data as well as possible. Typical criteria include geometric distance (e.g. Hausdorff-distance) or visual appearance (e.g. color difference, feature preservation, ...) [**?**].

There are many applications for decimation algorithms. First, they obviously can be used to *adjust the complexity* of a geometric data set. This makes geometry processing a scalable task where differently complex models can be used on computers with varying computing performance. Second, since many decimation schemes work iteratively, i.e. they decimate a mesh by removing one vertex at a time, they usually can be inverted. Running a decimation scheme backwards means to reconstruct the original data from a decimated version by inserting more and more detail information. This inverse decimation can be used for *progressive transmission* of geometry data [**?**]. Obviously, in order to make progressive transmission effective we have to use decimation operators whose inverse can be encoded compactly (cf. Fig. **??**).

There are several different conceptual approaches to mesh decimation. In principle we can think of the complexity reduction as a one step operation or as an iterative procedure. The vertex positions of the decimated mesh can be obtained as a subset of the original set of vertex positions, as a set of weighted averages of original vertex positions, or by resampling the original piecewise linear surface. In the literature the different approaches are classified into

- Vertex clustering algorithms
- Incremental decimation algorithms
- Resampling algorithms

The first class of algorithms is usually very efficient and robust. The computational complexity is typically linear in the number of vertices. However, the quality of the resulting meshes is not always satisfactory. *Incremental algorithms* in most cases lead to higher quality meshes. The iterative decimation procedure can take arbitrary user-defined criteria into account, according to which the next removal operation is chosen. However, their total computation complexity in the average case is $O(n \log n)$ and can go up to $O(n^2)$ in the worst case, especially when a global error threshold is to be respected. Finally, *resampling techniques* are the most general approach to mesh decimation. Here, new samples are more or less freely distributed over the original piecewise linear surface geometry. By connecting these samples a completely new mesh is constructed. The major motivation for resampling techniques is that they can enforce the decimated mesh to have a special connectivity structure, i.e. subdivision connectivity (or semi-regular connectivity). By this they can be used in a straight forward manner to build multiresolution representations based on subdivision basis functions and their corresponding (pseudo-) wavelets [**?**]. The most serious disadvantage of resampling, however, is that *alias errors* can occur if the sampling pattern is not perfectly aligned to features in the original geometry. To avoid alias effects, many resampling

schemes to some degree require manual pre-segmentation of the data for reliable feature detection. Resampling techniques will be discussed in detail in Chapter **??**.

In the following sections we will explain the different approaches to mesh decimation in more detail. Usually there are many choices for the different ingredients and sub-procedures in each algorithm and we will point out the advantages and disadvantages for each class (see also [**?**] for a comparison of different decimation techniques for point-sampled surfaces).

9.1 Vertex Clustering

The basic idea of vertex clustering is quite simple: for a given approximation tolerance ε we partition the bounding space around the given object into cells with diameter smaller than that tolerance. For each cell we compute a representative vertex position, which we assign to all the vertices that fall into that cell. By this clustering step, original faces degenerate if two or three of their corners lie in the same cell and consequently are mapped to the same position. The decimated mesh is eventually obtained by removing all those degenerate faces [**?**].

The remaining faces correspond to those original triangles whose corners all lie in different cells. Stated otherwise: if $\mathbf{p}$ is the representative vertex for the vertices $\mathbf{p}_0, ..., \mathbf{p}_n$ in the cluster P and $\mathbf{q}$ is the representative for the vertices $\mathbf{q}_0 ..., \mathbf{q}_m$ in the cluster Q then $\mathbf{p}$ and $\mathbf{q}$ are connected in the decimated mesh if and only if at least one pair of vertices $(\mathbf{p}_i, \mathbf{q}_j)$ was connected in the original mesh.

One immediately obvious draw-back of vertex clustering is that the resulting mesh might no longer be 2-manifold even if the original mesh was. Topological changes occur when the part of a surface that collapses into a single point is not homeomorphic to a disc, i.e., when two different sheets of the surface pass through a single ε-cell. However, this disadvantage can also be considered as an advantage. Since the scheme is able to change the topology of the given model we can reduce the object complexity very effectively. Consider, e.g., applying mesh decimation to a 3D-model of a sponge. Here, any decimation scheme that preserves the surface topology cannot reduce the mesh complexity significantly since all the small holes have to be preserved.

The computational efficiency of vertex clustering is determined by the effort it takes to map the mesh vertices to clusters. For simple uniform spatial grids this can be achieved in linear time with small constants. Then for each cell a representative has to be found which might require fairly complicated computations but the number of clusters is usually much smaller than the number of vertices.

Another apparently nice aspect of vertex clustering is that it automatically guarantees a global approximation tolerance by defining the clusters accordingly. However, in practice it turns out that the actual approximation error of the decimated mesh is usually much smaller than the radius of the clusters. This indicates that for a given error threshold, vertex clustering algorithms do not achieve optimal complexity reduction. Consider, as an extreme example, a very fine planar mesh. Here decimation down to a single triangle without any approximation error would be possible. The result of vertex clustering instead will always keep one vertex for every ε-cell.

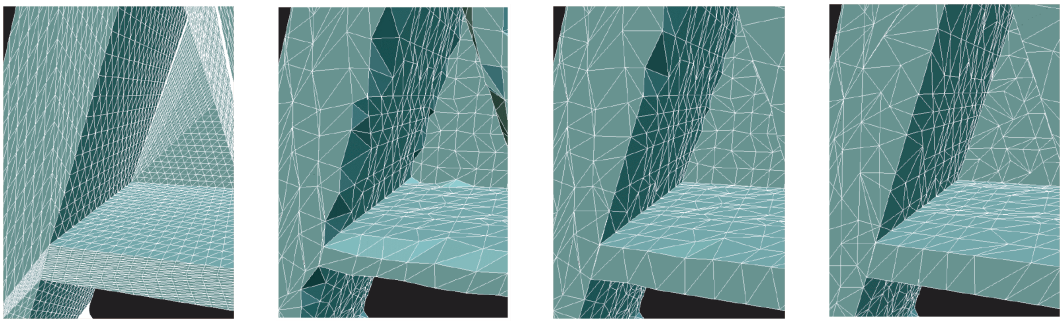


Figure 9.1: Different choices for the representative vertex when decimating a mesh using clustering. From left to right: Original, average, median, quadric-based.

9.1.1 Computing Cluster Representatives

The way in which vertex clustering algorithms differ is mainly in how they compute the representative. Simply taking the center of each cell, the straight average, or the median of its members are obvious choices which, however, rarely lead to satisfying results (cf. Fig. **??**).

A more reasonable choice is based on finding the optimal vertex position in the least squares sense. For this we exploit the fact that for sufficiently small ε the polygonal surface patch that lies within one ε-cell is expected to be piecewise flat, i.e., either the associated normal cone has a small opening angle (totally flat) or the patch can be split into a small number of sectors for which the normal cone has a small opening angle.

The optimal representative vertex position should have a minimum deviation from all the (regression) tangent planes that correspond to these sectors. If these approximate tangent planes do not intersect in a single point, we have to compute a solution in the least squares sense.

Consider one triangle t_i belonging to a specific cell, i.e., whose corner vertices lie in the same cell. The quadratic distance of an arbitrary point $\mathbf{x}$ from the supporting plane of that triangle can be computed by

$$(\mathbf{n}_i^T \mathbf{x} - d_i)^2 \ ,$$

where $\mathbf{n}_i$ is the normal vector of t_i and d_i is the scalar product of $\mathbf{n}_i$ times one of t_i's corner vertices. The sum of the quadratic distances to all the triangle planes within one cell is given by

$$E(\mathbf{x}) \ = \ \sum_i (\mathbf{n}_i^T \mathbf{x} - d_i)^2 \ . \tag{9.1}$$

The iso-contours of this error functional are ellipsoids and consequently the resulting error measure is called *quadric error metric (QEM)* [**?**, **?**]. The point position where the quadric error is minimized is given by the solution of

$$\left(\sum_i \mathbf{n}_i \, \mathbf{n}_i^T \right) \mathbf{x} \ = \ \left(\sum_i \mathbf{n}_i \, d_i \right) \ . \tag{9.2}$$

If the matrix has full rank, i.e. if the normal vectors of the patch do not lie in a plane, then the above equation could be solved directly. However, to avoid special case handling and to make the solution more robust, a pseudo-inverse based on a *singular value decomposition* should be used.

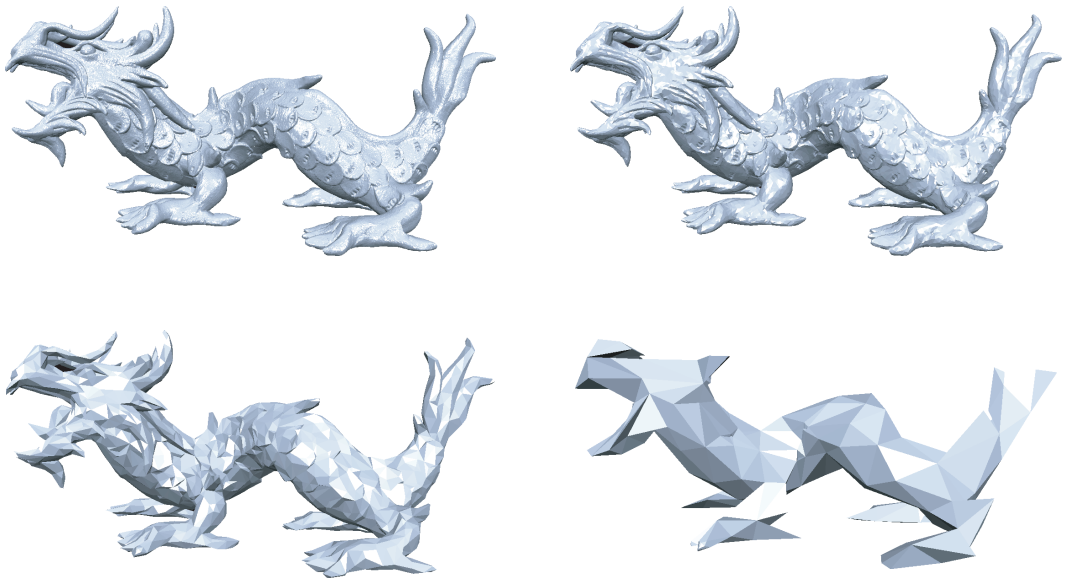


Figure 9.2: Decimation of the dragon mesh consisting of 577.512 triangles (top left) to simplified version with 10%, 1%, and 0.1% of the original triangle count.

9.2 Incremental Mesh Decimation

Incremental algorithms remove one mesh vertex at a time (see Fig. **??**). In each step, the best candidate for removal is determined based on user-specified criteria. Those criteria can be *binary* (= removal is allowed or not) or *continuous* (= rate the quality of the mesh after the removal between 0 and 1). Binary criteria usually refer to the global approximation tolerance or to other minimum requirements, e.g., minimum aspect ratio of triangles. Continuous criteria measure the *fairness* of the mesh in some sense, e.g., "round" triangles are better than thin ones, small normal jumps between neighboring triangles are better than large normal jumps.

Every time a removal has been executed, the surface geometry in the vicinity changes. Therefore, the quality criteria have to be re-evaluated. During the iterative procedure, this re-evaluation is the computationally most expensive part. To preserve the order of the candidates, they are usually kept in a *heap data structure* with the best removal operation on top. Whenever removal candidates have to be re-evaluated, they are deleted from the heap and re-inserted with their new value. By this, the complexity of the update-step increases only like $O(\log n)$ for large meshes if the criteria evaluation itself has constant complexity.

9.2.1 Topological operations

There are several different choices for the basic removal operation. The major design goal is to keep the operation as simple as possible. In particular this means that we do not want to remove large parts of the original mesh at once but rather remove a single vertex at a time. Strong decimation is then achieved by applying many simple decimation step instead of a few

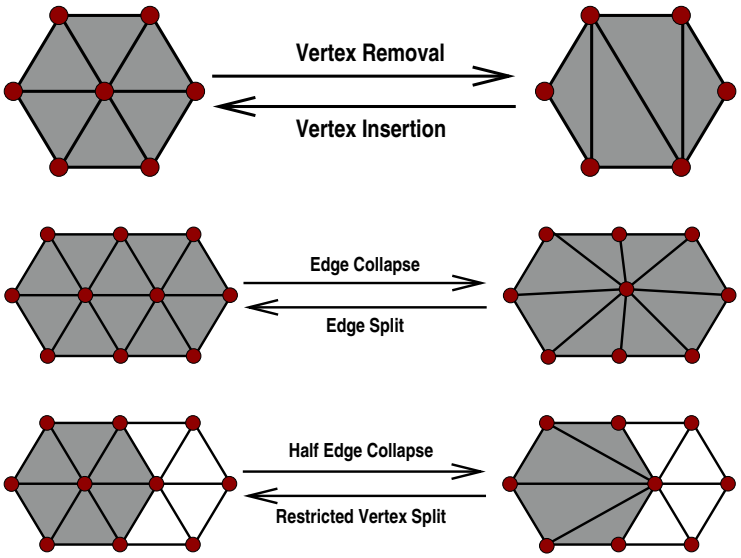


Figure 9.3: Euler-operations for incremental mesh decimation and their inverses: vertex removal, full edge collapse, and half-edge collapse.

complicated ones. If mesh consistency, i.e., topological correctness matters, the decimation operator has to be an *Euler-operator* (derived from the Euler formula for graphs) [**?**].

The first operator one might think of *deletes one vertex* plus its adjacent triangles. For a vertex with valence k this leaves a k-sided hole. This hole can be fixed by any polygon triangulation algorithm [**?**]. Although there are several combinatorial degrees of freedom, the number of triangles will always be $k - 2$. Hence the removal operation decreases the number of vertices by one and the number of triangles by two (cf. Fig. **??**, top).

Another decimation operator takes two adjacent vertices $\mathbf{p}$, $\mathbf{q}$ and collapses the edge between them, i.e., both vertices are moved to the same new position $\mathbf{r}$ [**?**] (cf. Fig. **??**, middle). By this two adjacent triangles degenerate and can be removed from the mesh. In total this operator also removes one vertex and two triangles. The degrees of freedom in this *edge collapse* operator emerge from the freedom to choose the new position $\mathbf{r}$.

Both operators that we discussed so far are not unique. In either case there is some optimization involved to find the best local triangulation or the best vertex position. Conceptually this is not well-designed since it mixes the global optimization (which candidate is best according to the sorting criteria for the heap) with local optimization.

A possible way out is the so-called *half-edge collapse* operation: for an ordered pair ($\mathbf{p}$, $\mathbf{q}$) of adjacent vertices, $\mathbf{p}$ is moved to $\mathbf{q}$'s position [**?**] (cf. Fig. **??**, bottom). This can be considered as a special case of edge collapsing where the new vertex position $\mathbf{r}$ coincides with $\mathbf{q}$. On the other hand, it can also be considered as a special case of vertex deletion where the triangulation of the k-sided hole is generated by connecting all neighboring vertices with vertex $\mathbf{q}$.

The half-edge collapse has no degrees of freedom. Notice that ($\mathbf{p} \rightarrow \mathbf{q}$) and ($\mathbf{q} \rightarrow \mathbf{p}$) are treated as independent removal operations which both have to be evaluated and stored in the candidate heap. Since half-edge collapsing is a special case of the other two removal operations, one might expect an inferior quality of the decimated mesh. In fact, half-edge collapsing merely sub-samples the set of original vertices while the full edge collapse can act as a low-pass filter

where new vertex positions are computed, e.g., by averaging original vertex positions. However, in practice this effect becomes noticeable only for extremely strong decimation where the exact location of individual vertices really matters.

The big advantage of half-edge collapsing is that for moderate decimation, the global optimization (i.e., candidate selection based on user specified criteria) is completely separated from the decimation operator which makes the design of mesh decimation schemes more orthogonal.

All the above removal operations preserve the mesh consistency and consequently the topology of the underlying surface. No holes in the original mesh can be closed, no handles can be eliminated completely. If a decimation scheme should be able to also simplify the topology of the input model, we have to use non-Euler removal operators. The most common operator in this class is the *vertex contraction* where two vertices $\mathbf{p}$ and $\mathbf{q}$ can be contracted into one new vertex $\mathbf{r}$ even if they are not connected by an edge [**?**, **?**]. This operation reduces the number of vertices by one but it does keep the number of triangles constant. The implementation of mesh decimation based on vertex contraction requires flexible data structures that are able to represent non-manifold meshes since the surface patch around vertex $\mathbf{r}$ after the contraction might no longer be homeomorphic to a (half-)disc.

9.2.2 Distance measures

Guaranteeing an approximation tolerance during decimation is the most important requirement for most applications. Usually an upper bound ε is prescribed and the decimation scheme looks for the mesh with the least number of triangles that stays within ε to the original mesh. However, exactly computing the geometric distance between two polygonal mesh models is computationally expensive [**?**, **?**] and hence conservative approximations are used that can be evaluated quickly.

The generic situation during mesh decimation is that each triangle t_i in the decimated mesh is associated with a sub-patch $\mathcal{S}_i$ of the original mesh. Distance measures have to be computed between each triangle t_i and either the vertices or faces of $\mathcal{S}_i$. Depending on the application, we have to take the maximum distance or we can average the distance over the patch.

The simplest technique is error accumulation [**?**]. For example each edge collapse operation modifies the adjacent triangles t_i by shifting one of their corner vertices from $\mathbf{p}$ or $\mathbf{q}$ to $\mathbf{r}$. Hence the distance of $\mathbf{r}$ to t_i is an upper bound for the approximation error introduced in this step. Error accumulation means that we store an error value for each triangle and simply add the new error contribution for every decimation step. The error accumulation can be done based on scalar distance values or on distance vectors. Vector addition takes the effect into account that approximation error estimates in opposite directions can cancel each other.

Another distance measure assigns distance values to the vertices $\mathbf{p}_j$ of the decimated mesh. It is based on estimating the squared average of the distances of $\mathbf{p}_j$ from all the supporting planes of triangles in the patches $\mathcal{S}_i$ which are associated with the triangles t_i surrounding $\mathbf{p}_j$. This is, in fact, what the quadric error metric does [**?**].

Initially we compute the error quadric E_j for each original vertex $\mathbf{p}_j$ according to (**??**) by summing over all triangles which are directly adjacent to $\mathbf{p}_j$. Since we are interested in the *average* squared distance, E_j has to be normalized by dividing through the valence of $\mathbf{p}_j$ Then, whenever the edge between two vertices $\mathbf{p}$ and $\mathbf{q}$ is collapsed, the error quadric for the new vertex $\mathbf{r}$ is found by $E_r = (E_p + E_q)/2$.

The quadric error metric is evaluated by computing $E_j(\mathbf{p}_j)$. Hence when collapsing $\mathbf{p}$ and $\mathbf{q}$ into $\mathbf{r}$, the optimal position for $\mathbf{r}$ is given by the solution of (**??**). Notice that due to the

averaging step the quadric error metric does neither give a strict upper nor a strict lower bound on the true geometric error.

Finally, the most expensive but also the sharpest distance error estimate is the *Hausdorff-distance* [**?**]. This distance measure is defined to be the maximum minimum distance, i.e., if we have two sets $\mathcal{A}$ and $\mathcal{B}$ then $H(\mathcal{A}, \mathcal{B})$ is found by computing the minimum distance $d(\mathbf{p}, \mathcal{B})$ for each point $\mathbf{p} \in \mathcal{A}$ and then taking the maximum of those values. Notice that in general $H(\mathcal{A}, \mathcal{B}) \neq H(\mathcal{B}, \mathcal{A})$ and hence the *symmetric Hausdorff-distance* is the maximum of both values.

If we assume that the vertices of the original mesh represent sample points measured on some original geometry then the faces have been generated by some triangulation pre-process and should be considered as piecewise linear approximations to the original shape. From this point of view, the correct error estimate for the decimated mesh would be the one-sided Hausdorff-distance $H(\mathcal{A}, \mathcal{B})$ from the original sample points $\mathcal{A}$ to the decimated mesh $\mathcal{B}$.

To efficiently compute the Hausdorff-distance we have to keep track of the assignment of original vertices to the triangles of the decimated mesh. Whenever an edge collapse operation is performed, the removed vertices $\mathbf{p}$ and $\mathbf{q}$ (or $\mathbf{p}$ alone in the case of a half-edge collapse) are assigned to the nearest triangle in a local vicinity. In addition, since the edge collapse changes the shape of the adjacent triangles, the data points that previously have been assigned to these triangles, must be re-distributed. By this, every triangle t_i of the decimated mesh at any time maintains a list of original vertices belonging to the currently associated patch $\mathcal{S}_i$. The Hausdorff-distance is then evaluated by finding the most distant point in this list.

A special technique for exact distance computation is suggested in [**?**], where two offset surfaces to the original mesh are computed to bound the space where the decimated mesh has to stay in.

9.2.3 Fairness criteria

The distance measures can be used to decide which removal operation among the candidates is legal and which is not (because it violates the global error threshold ε). In an incremental mesh decimation scheme we have to provide an additional criterion which ranks all the legal removal operations. This criterion determines the ordering of the candidates in the heap.

One straightforward solution is to use the distance measure for the ordering as well. This implies that the decimation algorithm will always remove that vertex in the next step that increases the approximation error least. While this is a reasonable heuristic in general, we can use other criteria to optimize the resulting mesh for special application dependent requirements.

For example, we might prefer triangle meshes with faces that are as close as possible to equilateral. In this case we can measure the quality of a vertex removal operation, e.g., by the *longest edge to inner circle radius ratio* of the triangles after the removal.

If we prefer visually smooth meshes, we can use the maximum or average normal jump between adjacent triangles after the removal as a sorting criterion. Other criteria might include color deviation or texture distortion if the input data does not consist of pure geometry but also has color and texture attributes attached [**?**, **?**, **?**].

All these different criteria for sorting vertex removal operations are called *fairness criteria* since they rate the quality of the mesh beyond the mere approximation tolerance. If we keep the fairness criterion separate from the other modules in an implementation of incremental mesh decimation, we can adapt the algorithm to arbitrary user requirement by simply exchanging that

one procedure. This gives rise to a flexible tool-box for building custom tailored mesh decimation schemes [**?**].

9.3 Out-of-core Methods

Mesh decimation is frequently applied to very large data sets that are too complex to fit into main memory. To avoid severe performance degradation due to virtual memory swapping, *out-of-core* algorithms have been proposed that allow an efficient decimation of polygonal meshes without requiring the entire data set to be present in main memory. The challenge here is to design suitable data structures that avoid random access to parts of the mesh during the simplification.

Lindstrom [**?**] presented an approach based on vertex clustering combined with quadric error metrics for computing the cluster representatives (see Section **??**). This algorithm only requires limited connectivity information and processes meshes stored as a triangle soup, where each triangle is represented as a triplet of vertex coordinates. Using a single pass over the mesh data an in-core representation of the simplified mesh is build incrementally. A dynamic hash table is used for fast localization and quadrics associated with a cluster are aggregated until all triangles have been processed. The final simplified mesh is then produced by computing a representative from the per-cluster quadrics and the corresponding connectivity information as described above.

Lindstrom and Silva [**?**] improve on this approach by removing the requirement for the output model to fit into main memory by using a multi-pass approach. Their method only requires a constant amount of memory that is independent of the size of the input and output data. This improvement is achieved by a careful use of (slower, but cheaper) disk space, which typically leads to performance overheads between a factor of two and five as compared to [**?**]. To avoid storing the list of occupied clusters and associated quadrics in main memory, the required information from each triangle to compute the quadrics is stored to disk. This file is then sorted according to the grid locations using an external sort algorithm. Finally, quadrics and final vertex positions are computed in a single linear sweep over the sorted file. The authors also apply a scheme similar to the one proposed in [**?**] to better preserve boundary edges.

Wu and Kobbelt [**?**] proposed an streaming approach to out-of-core mesh decimation based edge collapse operations in connection with quadric error metric. Their method uses a fixed-size active working set and is independent of the input and output model complexity. In contrast to the previous two approaches for out-of-core decimation, their method allows to prescribe the size of the output mesh exactly and supports explicit control over the topology during the simplification. The basic idea is to sequentially stream the mesh data and incrementally apply decimation operations on an active working set that is kept in main memory. Assuming that the geometry stream is approximately pre-sorted, e.g., by one coordinate, the spatial coherency then guarantees that the working set can be small as compared to the total model size (see Fig. **??**) For decimation they apply randomized multiple choice optimization, which has been shown to produce results of similar quality than the standard greedy optimization. The idea is to select a small random set of candidate edges for contraction and only collapse the edge with smallest quadric error. This significantly reduces computation costs, since no global heap data structure has to be maintained during the simplification process. In order to avoid inconsistencies during the simplification, edges can only be collapsed, if they are not part of the boundary between the active working set and the parts of the mesh that are held out-of-core. Since no global connectivity information is available, this boundary cannot be distinguished from the actual mesh boundary of the input model. Thus the latter can only be simplified after the entire mesh has been processed, which can be problematic for meshes with large boundaries.

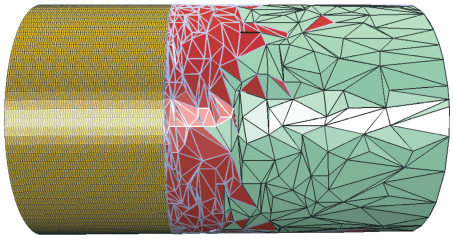


Figure 9.4: This snapshot of a stream decimation shows the yet unprocessed part of the input data (left), the current in-core portion (middle) and the already decimated output (right). The data in the original file happened to be pre-sorted from right to left (from [**?**]).

Isenburg et al. introduces *mesh processing sequences*, which represent a mesh as a fixed interleaved sequence of indexed vertices and triangles [**?**]. Processing sequences can be used to improve the out-of-core decimation algorithms described above. Both memory efficiency and mesh quality are improved for the vertex clustering method of [**?**], while increased coherency and explicit boundary information help to reduce the size of the active working set in [**?**].

Shaffer and Garland [**?**] proposed a scheme that combines an out-of-core vertex clustering step with an in-core iterative decimation step. The central observation, which is also the rationale behind the randomized multiple choice optimization, is that the exact ordering of edge collapses is only relevant for very coarse approximations. Thus the decimation process can be simplified by combining many edge collapse operations into single vertex clustering operations to obtain an intermediate mesh, which then serves as input for the standard greedy decimation (Section **??**). Shaffer and Garland use quadric error metrics for both types of decimation and couple the two simplification steps by passing the quadrics computed during clustering to the subsequent iterative edge collapse pass. This coupling achieves significantly improvements when compared to simply applying the two operations in succession.

10 Remeshing

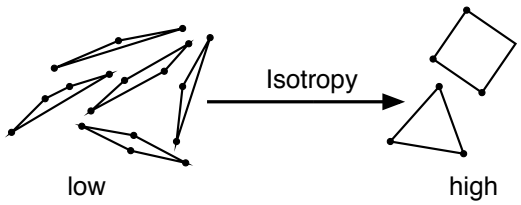
Remeshing is a key technique for mesh quality improvement in many geometric modeling algorithms, e.g., shape editing, animation, morphing, and numerical simulation. As such, it has received considerable attention in recent years and a wealth of remeshing algorithms have been developed. The first goal of remeshing is to reduce the complexity of an input mesh subject to certain quality criteria. This process is commonly referred to as *mesh decimation* or *mesh simplification,* a topic that is covered in Chapter **??** in more detail. The second goal of remeshing is to *improve the quality* of a mesh, such that it can be used as input for various downstream applications. Different applications, of course, imply different quality criteria and requirements. For a more complete coverage of the topic we refer the reader to a survey [**?**]. The latter proposes the following basic definition for remeshing: *Given a 3D mesh, compute another mesh whose elements satisfy some quality requirements, while approximating well the input.* Here the term approximation can be understood with respect to locations as well as to normals or higher order differential properties.

In contrast to general mesh repair (see Chapter **??**), the input of remeshing algorithms is usually assumed to already be a manifold triangle mesh or part of it. The term mesh quality thus refers to non-topological properties, such as sampling density, regularity, size, alignment, and shape of the mesh elements. This chapter in particular deals with these latter aspects of remeshing and presents various methods that achieve this goal. We begin our discussion by structuring the different types of remeshing algorithms and by clarifying some concepts that are commonly used in the remeshing literature. In the following sections we discuss several remeshing methods in more detail, focusing on the key paradigms behind each of them.

Local Structure The local structure of a mesh is described by the type, shape, orientation, and distribution of the mesh elements.

- *Element type:* The most common target element types in remeshing are *triangles* and *quadrangles.* Triangle meshes are usually easier to produce, while in quadrangular remeshing one often has to content oneself with results that are only *quad-dominant.* Note that in principle any quadrangle mesh can be converted trivially into a triangle mesh by inserting a diagonal into each quadrangle. Converting a triangle mesh into a quadrangle mesh can be performed either trivially by barycentric subdivision (splitting each triangle into three quadrangles by inserting its barycenter and linking it to edge midpoints), or by splitting each triangle at its barycenter into three new triangles (1-to-3 split) and discarding the original mesh edges.
- *Element shape:* Elements can be classified as being either *isotropic* or *anisotropic.* The shape of isotropic elements is locally uniform in all directions. Ideally, it is close to circular, thus a triangle/quadrangle is isotropic if it is close to equilateral/square. For triangles this "roundness" can be measured by dividing the length of the shortest edge by the circumcircle radius, see [**?**]. Isotropic elements are favored in numerical applications (FEM or geometry

processing), as the local uniform shape of their elements often leads to a better conditioning of the resulting systems, see [**?**] for a more detailed discussion. The shape of anisotropic elements locally varies according to the orientation on the surface. Anisotropic meshes are preferred for shape approximation as they usually need fewer elements than their isotropic pendants to achieve the same approximation quality. Anisotropic elements are commonly aligned with the principal curvature directions of the surface (see Chapter **??**). Furthermore anisotropic elements are shown to better express the structure of geometric primitives (plane, cylinders, spheres, ...) inherent in many technical models.



- *Element density:* In a *uniform* distribution, the mesh elements are evenly spread across the entire model. In a *non-uniform* or *adaptive* distribution, the number of elements varies, e.g., smaller elements are assigned to areas with small local feature size. When carefully designed, adaptive meshes need significantly fewer elements to achieve an approximation quality that is comparable to that of uniform meshes.
- *Element alignment:* Converting a piecewise smooth input surface into a (re-)mesh corresponds to a (re-)sampling process. Hence sharp features may be affected by alias-artifacts. In order to prevent this, elements should be aligned to sharp features such that they properly represent tangent discontinuities.

Global Structure A vertex in a triangle mesh is called *regular,* if its valence (i.e., number of neighboring vertices) is 6 for interior vertices or 4 for boundary vertices. In quadrangle meshes, the regular valences are 4 and 3, respectively. Vertices that are not regular are called *irregular*, *singular*, or *extraordinary.*

The global structure of a remesh can be classified as being either *completely regular, semi-regular, highly regular*, or *irregular*, see Fig. **??**.

- In a *completely regular* mesh all vertices are regular. A regular mesh can compactly be stored in a two-dimensional array which can be used to speed up the visualization (a so-called *geometry image*), see [**?**, **?**, **?**].
- *Semi-regular* meshes are produced by regular subdivision of a coarse initial mesh. Thus the number of extraordinary vertices in a semi-regular mesh is small and constant [**?**, **?**, **?**, **?**] under uniform refinement.
- In *highly regular* meshes most vertices are regular. In contrast to semi-regular meshes, highly regular meshes need not be the result of a subdivision process [**?**, **?**, **?**, **?**].
- *Irregular* meshes do not exhibit any kind of regularities in their connectivity.

Besides this topological characterization, the suitability of a remeshing algorithm usually depends on its ability to capture the global structure of the input geometry by aligning groups of

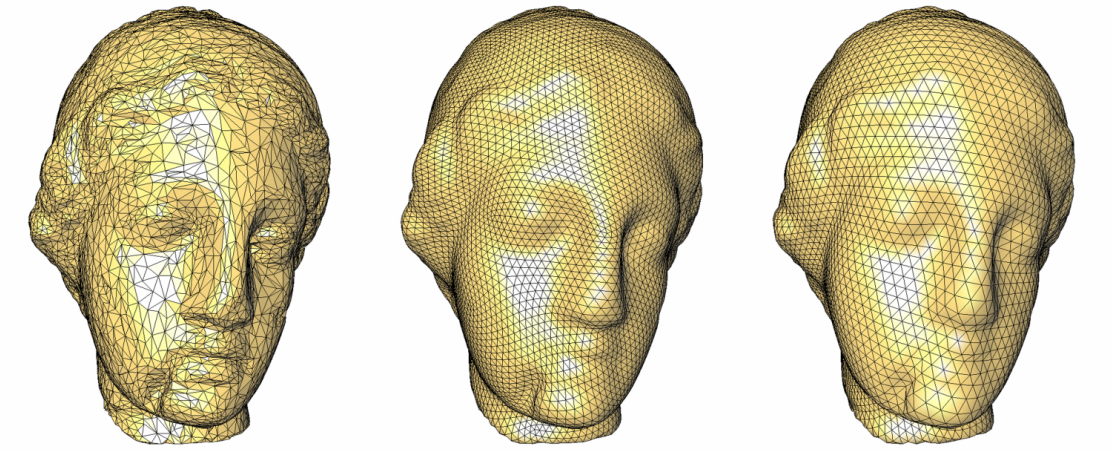


Figure 10.1: Meshes: Irregular, semi-regular and regular.

elements to the dominant geometric features. Since this corresponds to the alignment of entire submeshes, e.g., to global curvature lines of geometric primitives, it is strongly related to mesh segmentation techniques [**?**].

Fully regular meshes can be generated only for a very limited number of input models, namely those that topologically are (part of) a torus. All other models have to be cut into one or more topological disks before processing (and then the global regularity is broken at the seams). Furthermore, special care has to be taken to correctly identify and handle the seams that result from the cutting. Semi-regular meshes are in particular suitable for multi-resolution analysis and modeling [**?**, **?**]. They define a natural parameterization of a model over a coarse base mesh. Thus, some algorithms for semi-regular remeshing are described in Chapter **??**. Highly regular meshes require different techniques for multi-resolution analysis, but still they are well-suited for numerical simulations. In particular, mesh compression algorithms can take advantage of the mostly uniform valence distribution and produce a very efficient connectivity encoding [**?**, **?**, **?**].

Correspondences All remeshing algorithms compute point locations on or near the original surface. Most algorithms furthermore iteratively relocate sample points in order to improve the quality of the mesh. Thus, a key issue in all remeshing algorithms is to compute or to maintain correspondences between sample points $\mathbf{p}$ on the remesh and their counterparts $\phi(\mathbf{p})$ on the input mesh. There are a number of approaches to address this problem:

- *Global parameterization:* The input model is globally parameterized onto a 2D domain. Sample points can then be easily distributed and relocated in the 2D domain and later be "lifted" to 3D.
- *Local parameterization:* The algorithm maintains a parameterization of a local geodesic neighborhood around $\phi(\mathbf{p})$. When the sample leaves this neighborhood, a new neighborhood has to be computed.
- *Projection:* The sample point is directly projected to the nearest element (point or triangle) on the input model.

Global parameterization is in general expensive and may suffer from parametric distortion. Naive direct projection may produce local and global fold overs if the points are too far away

from the surface. However, in practice the projection operator can be stabilized by constraining the movement of the sample points to their tangent planes. Although no theoretical guarantees can be provided, this makes sure that the samples do not move away too far from the surface, such that the projection can safely be evaluated. The local parameterization approach is stable and produces currently the best results, however, it needs expensive book keeping to track, cache, and re-parameterize the local neighborhoods.

10.1 Isotropic

In an isotropic mesh all triangles are well-shaped, i.e., ideally equilateral. One may further require a globally uniform vertex density or allow a smooth change in the triangle sizes, i.e., a smooth *gradation*. There are a number of algorithms for isotropic remeshing of triangle meshes, see [**?**]. In this section we describe three different paradigms commonly employed for isotropic remeshing, then describe three representative algorithms for these paradigms.

Existing algorithms could be roughly classified as being *greedy*, *variational*, or *pliant*. Greedy algorithms commonly perform one local change at a time, such as vertex insertion, until the initial stated goal is satisfied. Variational techniques cast the initial problem into the one of minimizing an energy functional such that low levels of this energy correspond to good solutions for this problem (reaching a global optimum is in general elusive). A minimizer for this energy commonly performs global relaxation, i.e., vertex relocations and re-triangulation until convergence. Finally, an algorithm is pliant when it combines both refinement and decimation, possibly interleaved with a relaxation procedure (see [**?**]).

10.1.1 Greedy

The greedy surface meshing algorithm described in [**?**] is flexible enough to be used for isotropic surface remeshing. The core principle behind this algorithm relies on refining and filtering a 3D Delaunay triangulation. At each refinement step one point taken on the input surface is inserted to the 3D triangulation. The point location is chosen among the intersections of the input surface $\mathcal{S}$ with the Voronoi edges of the triangulation. The filtering process consists of selecting a subset of the 3D Delaunay facets, whose dual edges intersect $\mathcal{S}$. After refinement, the resulting subcomplex, called Delaunay triangulation restricted to $\mathcal{S}$, enjoys several guarantees.

Guarantees Although remarkably simple in principle, the algorithm summarized above is shown to terminate after a finite number of refinement steps. Moreover, the number of vertices added is asymptotically within a constant factor of the optimal. Upon termination, the output of the algorithm (i.e., the piecewise linear interpolation derived from the restricted Delaunay triangulation), is shown to enjoy both *approximation* guarantees in terms of topology and geometry, and *quality* guarantees in terms of shape of the mesh elements. More precisely, the restricted Delaunay triangulation is homeomorphic to the input surface $\mathcal{S}$ and approximates it in terms of Hausdorff distance, normals, curvature, and area. All angles of the triangles are bounded, which provides us with a mesh quality amenable to reliable mesh processing operations and faithful simulations. Further theoretical developments for this approach have been presented in [**?**].

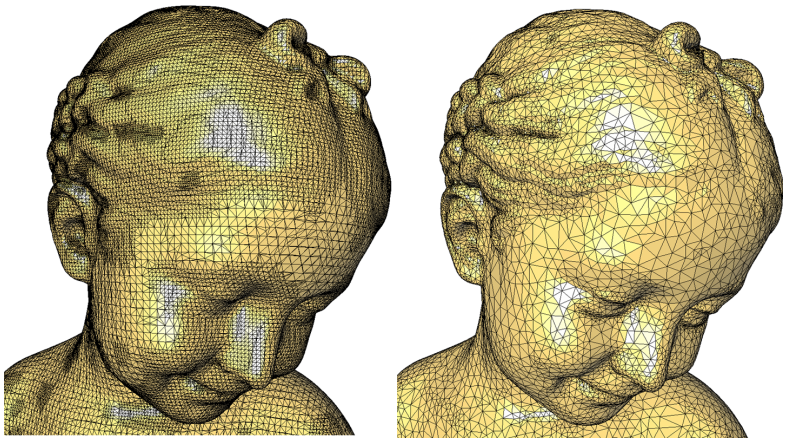


Figure 10.2: Remeshing by Delaunay refinement and filtering. The input mesh (left) is the output of a marching cubes algorithm over an octree.

Flexibility The elementary operation of the meshing process reduces to the insertion of a new vertex into the 3D Delaunay triangulation which interpolates the input surface. The only assumption made is that the input surface representation is amenable to simple geometric computations, namely its intersection with a line. In other words, the shape to be discretized is only known through an *Oracle* which provides answers to intersection predicates. The input shape can thus be represented as a surface mesh, hence its use for remeshing, see examples Fig. **??** and Fig. **??**. Furthermore, the input shape can be a surface reconstructed from another representation such as a point set or a set of slices.

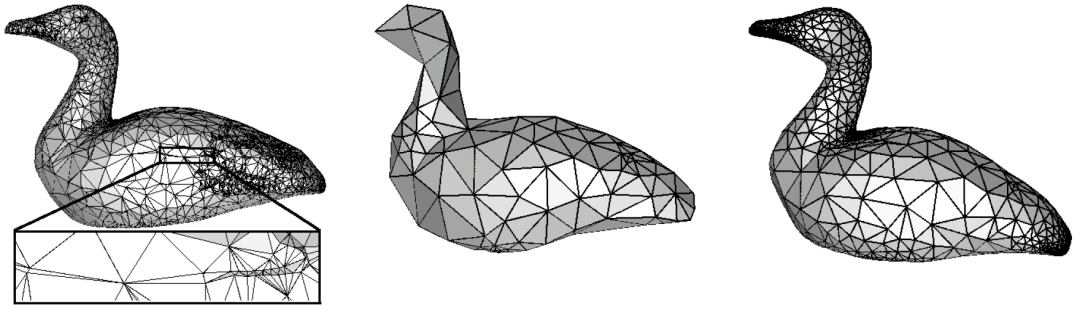


Figure 10.3: Isotropic remeshing by Delaunay refinement and filtering. The input mesh (left) is the output of an interpolatory surface reconstruction algorithm.

Discussion The main advantages of such greedy algorithm are its guaranteed properties. It is also quite robust as it does not resort to any local or global parameterization technique but constructs a 3D tetrahedral mesh instead. The following questions may arise: Can we construct a mesh of higher quality? With fewer vertices while satisfying the same set of constraints? These questions are partially addressed by some variational techniques.

10.1.2 Variational

When high quality meshes are sought after, it may be desirable to resort to an optimization procedure. Two questions now arise: Which criterion should we optimize? By exploiting which degrees of freedom? The optimized criterion can be directly related to the shape and size of the triangles, but we will describe next how other criteria achieve satisfactory results as well. As the number of degrees of freedom are both continuous and discrete (vertex positions and mesh connectivity), there is a need for narrowing the space of possible triangulations.

Mesh optimization, also commonly referred to as *mesh smoothing* in the meshing community, has addressed parts of these questions, although some work remains to be done in order to specialize these technique to remeshing of surfaces. We refer the reader to a comprehensive survey of mesh optimization techniques [**?**].

Designing a variational algorithm requires defining an energy to minimize, and a minimizer for this energy. Ideally, the minimizer is fast, robust, and converges to a global optimum. In practice however, the space of possible solutions is so vast that reaching a global optimum is a mirage, even more so when the notion of "best possible mesh" is not uniquely defined. The zoo of criteria used for the optimization (see e.g., [**?**]) reveals the difficulty of choosing one criterion to optimize: Should we optimize over the triangle angles? The edge lengths? The compactness of the triangles? Although one optimization technique has been specifically designed for optimizing the shape of the triangles [**?**], a class of mesh smoothing techniques rely on the observation that isotropic 2D point samplings lead to well-shaped triangles [**?**]. Note that in 3D this observation does not hold anymore as sliver tetrahedra can occur. Isotropic remeshing can therefore be casted into the problem of isotropic point sampling, which amounts to distribute a set of points on the input mesh in as even a manner as possible.

One approach to evenly distribute a set of points in 2D is to construct a centroidal Voronoi tessellation [**?**]. Given a density function defined over a bounded domain Ω, a centroidal Voronoi tessellation (denoted CVT) of Ω is a class of Voronoi tessellations, where each site coincides with the centroid (i.e., center of mass) of its Voronoi region. The centroid $\mathbf{c}_i$ of a Voronoi region V_i is calculated as:

$$\mathbf{c}_i = \frac{\int_{V_i} \mathbf{x} \cdot \rho(\mathbf{x}) \, \mathrm{d}\mathbf{x}}{\int_{V_i} \rho(\mathbf{x}) \, \mathrm{d}\mathbf{x}} \, , \tag{10.1}$$

where $\rho(\mathbf{x})$ is the density function of $\mathcal{V}_i$. This structure turns out to have a surprisingly broad range of applications for numerical analysis, location optimization, optimal repartition of resources, cell growth, vector quantization, etc. This follows from the mathematical importance of its relationship with the energy function

$$E(z, V) = \sum_{i=1}^{n} \int_{V_i} \rho(\mathbf{x}) \left\| \mathbf{x} - \mathbf{z}_i \right\|^2 \mathrm{d}\mathbf{x} \, , \tag{10.2}$$

where $V \in \Omega$ and $z \in V$. We can show that (i) the energy function is minimized at the mass centroid of a given region, and (ii) for a given set of centers $Z = \{\mathbf{z}_i\}$, the energy function $E(Z, V)$ is minimized when V is a Voronoi tessellation.

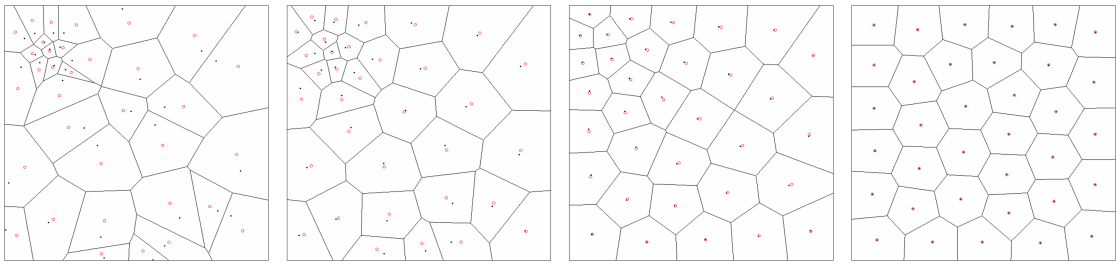


Figure 10.4: Left: ordinary Voronoi tessellation. Middle left: Voronoi tessellation after one Lloyd iteration. Middle right: Voronoi tessellation after three Lloyd iterations. Right: centroidal Voronoi tessellation obtained after convergence of the Lloyd iteration. Each generator coincides with the centroid (center of mass) of its Voronoi cell.

One way to build a centroidal Voronoi tessellation is to use Lloyd's relaxation method. The Lloyd algorithm is a deterministic, fixed point iteration [**?**]. Given a density function and an initial set of n sites, it consists of the following three steps (see Fig. **??**):

1. Construct the Voronoi tessellation corresponding to the n sites;

2. Compute the centroids of the n Voronoi regions with respect to the density function, and move the n sites to their respective centroids;

3. Repeat steps 1 and 2 until satisfactory convergence is achieved.

Alliez et al. [**?**] propose a surface remeshing technique based on Lloyd relaxation. It uses a global conformal planar parameterization and then applies relaxation in the parameter space using a density function designed to compensate for the area distortion due to flattening (see Fig. **??**).

To alleviate the numerical issues for high isoperimetric distortion, as well as the artificial cuts required for closed or models with non-trivial topology, Surazhsky et al. apply the Lloyd relaxation procedure on a set of local overlapping parameterizations, see Fig. **??**. The Lloyd-based isotropic remeshing approach has been later extended in two directions: One uses the geodesic distance on triangle meshes to generate a centroidal geodesic-based Voronoi diagram [**?**], while the other is an efficient discrete analog of the Lloyd relaxation applied onto the input mesh triangles [**?**].

10.1.3 Pliant

In this section we present an efficient remeshing algorithm that produces isotropic triangle meshes. The algorithm was presented in [**?**] and is a simplified version of [**?**] and an extension to [**?**]. It produces results that are comparable to the ones by the original algorithm, but has the advantage of being simpler to implement and robust. In particular, it does not need a (global or local) parameterization or the involved computation of (geodesic) Voronoi cells as, e.g., [**?**]. The algorithm takes as input a target edge length and then repeatedly splits long edges, collapses short edges, and relocates vertices until all edges are approximately of the desired target edge length. Thus the algorithm runs the following loop:

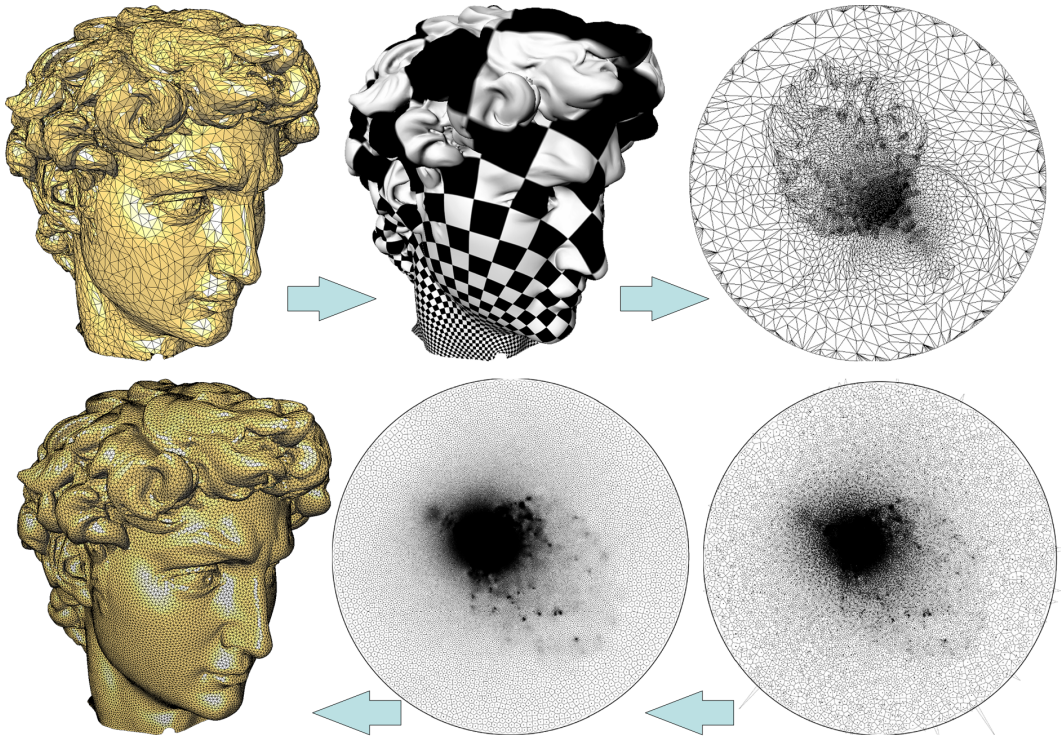


Figure 10.5: Isotropic remeshing of the Michelangelo David head.

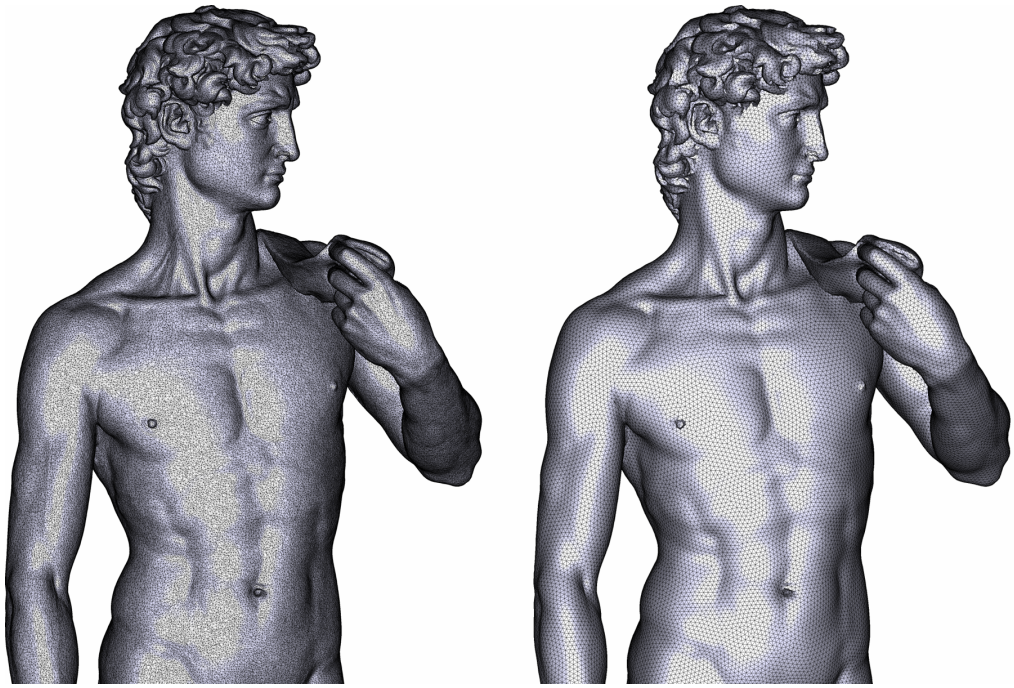


Figure 10.6: Isotropic remeshing using overlapping parameterizations.

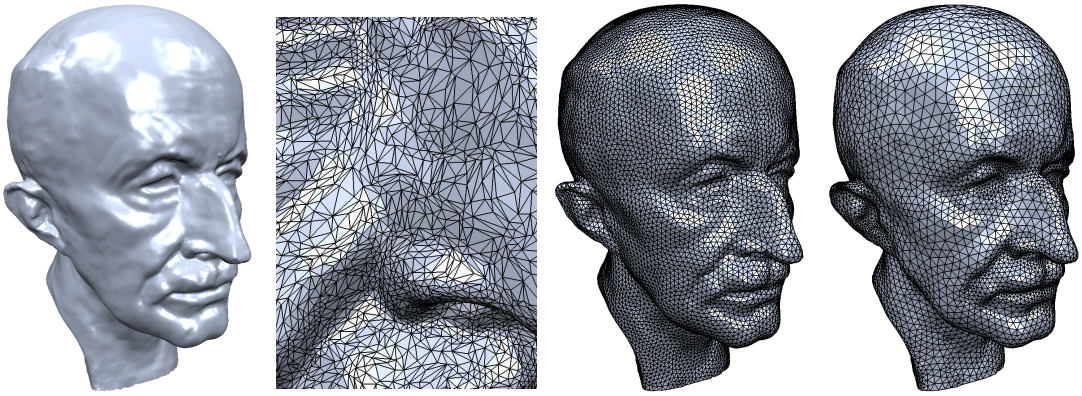


Figure 10.7: Isotropic remeshing. Left and center left: Max Planck model at full resolution. Center right and right: Uniform and adaptive remeshes.

```
remesh(target_edge_length)
  low = 4/5 * target_edge_length
  high = 4/3 * target_edge_length
  for i = 0 to 10 do
    split_long_edges(high)
    collapse_short_edges(low,high)
    equalize_valences()
    tangential_relaxation()
    project_to_surface()
```

Notice that the proper thresholds $\frac{4}{5}$ and $\frac{4}{3}$ are essential to converge to a uniform edge length [**?**]. The values are derived from considerations to make sure that the edge lengths are closer to the target lengths after a split or collapse operation than before. A hysteresis behavior is induced by the interleaved tangential smoothing operator.

The split_long_edges(high) function visits all edges of the current mesh. If an edge is longer than the given threshold high, the edge is split at its midpoint and the two adjacent triangles are bisected (2-4 split).

```
split_long_edges(high)
  while exists edge e with length(e)>high do
    split e at midpoint(e)
```

The collapse_short_edges(low, high) function collapses and thus removes all edges that are shorter than a threshold low. Here one has to take care of a subtle problem: by collapsing along chains of short edges the algorithm may create new edges that are arbitrarily long and thus undo the work that was done in split_long_edges(high). This issue is resolved by testing before each collapse whether the collapse would produce an edge that is longer than high. If so, the collapse is not executed.

```
collapse_short_edges(low, high)
  finished = false
  while exists edge e with length(e)<low and not finished do
```

```
    finished = true
    let e=(a,b) and let a[1],...,a[n] be the 1-ring of a
    collapse_ok = true
    for i = 1 to n do
      if length(b,a[i])>high then
        collapse_ok = false
    if collapse_ok then
      collapse a into b along e
      finished = false
```

The equalize_valences() function equalizes the vertex valences by flipping edges. The target valence target_val(v) is 6 and 4 for interior and boundary vertices, respectively. The algorithm tentatively flips each edge e and checks whether the deviation to the target valences decreases. If not, the edge is flipped back.

```
equalize_valences()
  for each edge e do
    let a,b,c,d be the vertices of the two triangles adjacent to e
    deviation_pre = abs(valence(a)-target_val(a)) + abs(valence(b)-target_val(b))
                  + abs(valence(c)-target_val(c)) + abs(valence(d)-target_val(d))
    flip(e)
    deviation_post = abs(valence(a)-target_val(a)) + abs(valence(b)-target_val(b))
                   + abs(valence(c)-target_val(c)) + abs(valence(d)-target_val(d))
    if deviation_pre ≤ deviation_post do
      flip(e)
```

The tangential_relaxation() function applies an iterative smoothing filter to the mesh. Here the vertex movement has to be constrained to the vertex' tangent plane in order to stabilize the following projection operator. Let $\mathbf{p}$ be an arbitrary vertex in the current mesh, let $\mathbf{n}$ be its normal, and let $\mathbf{q}$ be the position of the vertex as calculated by a smoothing algorithms with uniform Laplacian weights (see Chapter **??**). The new position $\mathbf{p}'$ of $\mathbf{p}$ is then computed by projecting $\mathbf{q}$ onto $\mathbf{p}$'s tangent plane

$$\mathbf{p}' = \mathbf{q} + \mathbf{n}\mathbf{n}^T(\mathbf{p} - \mathbf{q}) \ .$$

Again, this can be easily implemented:

```
tangential_relaxation()
  for each vertex v do
    q[v] = the barycenter of v's neighbor vertices
  for each vertex v do
    let p[v] and n[v] be the position and normal of v, respectively
    p[v] = q[v] + dot(n[v],(p[v]-q[v]))*n[v]
```

Finally, the project_to_surface() function maps the vertices back to the surface.

Feature preservation A few simple rules suffice to make sure that the remeshing algorithm preserves the features of the input model, see Fig. **??**. Here we assume, that the feature edges and vertices have already been marked in the input model, e.g., by automatic feature detection algorithms or by manual specification [**?**, **?**].

- Corner vertices with more than two or exactly one incident feature edge have to be preserved and are excluded from all topological and geometric operations.
- Feature vertices may only be collapsed along their incident feature edges.
- Splitting a feature edge creates two new feature edges and a feature vertex.
- Feature edges are never flipped.
- Tangential smoothing of feature vertices is restricted to univariate smoothing along the corresponding feature lines.

As can be seen in Fig. **??** and Fig. **??**, the algorithm above produces quite good results. It is also possible to incorporate additional regularization terms by adjusting the weights that are used in the smoothing phase. This allows to achieve a uniform triangle area distribution or to implement an adaptive remeshing algorithm that produces finer elements in regions of high curvature.

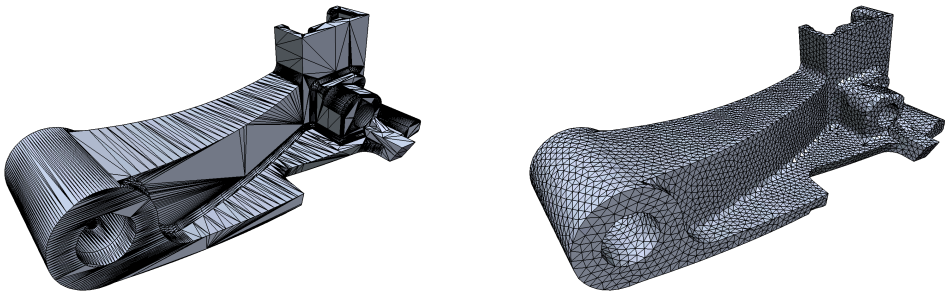


Figure 10.8: Isotropic, feature sensitive remeshing of a CAD model.

10.2 Quadrangle

Partitioning a surface into quadrilateral regions is a common requirement in computer graphics, computer aided geometric design and, reverse engineering. Such quad tilings are amenable to a variety of subsequent applications due to their tensor-product nature, such as B-spline fitting, simulation, texture atlasing, and complex rendering with highly detailed modulation maps. Quad meshes are also useful in modeling as they aptly capture the symmetries of natural or man-made geometry.

In an anisotropic mesh the elements align to the principal curvature directions, i.e., they are elongated along the minimum curvature direction and shortened along the maximum curvature direction (see Chapter **??**). Anisotropic triangle remeshes of a given target complexity can easily be produced by incrementally decimating the input model down to a desired target complexity (see also Section **??**). No matter whether one uses quadric error metrics, (one-sided) Hausdorff-distance, or the normal deviation to rank the priorities of removal operations, the result will always be an anisotropic triangle mesh that naturally aligns to the principal curvature directions. The remeshes that are produced by this method satisfy the definition of being anisotropic, but unfortunately they do not convey the orthogonal structure of the curvature lines. To produce such a structure, it is usually better to first compute a quadrangle remesh.

Automatically converting a triangulated surface (issued, e.g., from a 3D scanner) into a quad mesh is a notoriously difficult task. Stringent topological conditions make quadrangulating a domain or a surface a rather constrained and global problem compared to triangulating a domain. Application-dependent meshing requirements such as edge orthogonality, alignment of the elements with the geometry, sizing, and mesh regularity add further hurdles.

Several paradigms have been proposed for generating quadrangle meshes:

- Quadrangulation: A number of techniques have been proposed to quadrangulate point sets. A subset of these techniques allow generating all-convex quadrangles by adding Steiner points [**?**], and well-shaped quadrangles using some circle packing technique [**?**]. Quadrangle meshing thus amounts to carefully placing a set of points, which are then automatically quadrangulated. In the context of surface remeshing, the main issue with this paradigm is the lack of control over the alignment of the edges and over the mesh regularity.

- Conversion: One way to generate quadrangle meshes is to first generate a triangle or polygon mesh, then convert it to a quadrangle mesh. Examples of such approaches typically proceed by pairwise triangle merging and 4-8 subdivision, or by bisection of hex-dominant meshes followed by barycentric subdivision [**?**]. As for quadrangulation of point sets, this approach provides the user with little control over alignment of the mesh edges.

- Curve-based sampling: One way to control the edge alignment of the mesh edges is to place a set of curves which are everywhere tangent to direction fields. The vertices of the final remesh are obtained by intersecting the networks of curves. When using lines of curvatures the output meshes are quad-dominant, although not pure quadrangle meshes as T-junctions can appear due to the greedy process used for tracing the lines of curvatures. Another curve-based approach consists of placing a set of minimum-bending curves.

- Contouring: When pure quadrangle meshes are sought after (without T-junctions), a robust approach consists of computing two scalar functions, and extracting a quadrangle surface tiling by contouring these functions along well-chosen isovalues. These methods include the parameterization-based techniques (see Chapter **??**).

In the following sections, we restrict ourselves to the approaches based upon curve-based sampling and contouring.

10.2.1 Curve-Based Sampling

Lines of Curvatures. The remeshing technique introduced by Alliez et al. [**?**] generates a quad-dominant mesh that reflects the symmetries of the input shape by sampling the input shape with curves instead of the usual points (see Fig. **??**). The algorithm has three main stages. The first stage recovers a continuous model from the input triangle mesh by estimating one 3D curvature tensor per vertex (see Chapter **??**). The normal component of each tensor is then discarded and a 2D piecewise linear curvature tensor field is built after computing a discrete conformal parameterization. This field is then altered to obtain smoother principal curvature directions. The singularities of the tensor field (the umbilics) are also extracted. The second stage consists of resampling the original mesh in parameter space by building a network of lines of curvatures (a set of "streamlines" approximated by polylines) following the principal curvature directions. A user-prescribed approximation precision in conjunction with the estimated curvatures is used to define the local density of lines of curvatures at each point in parameter space during the

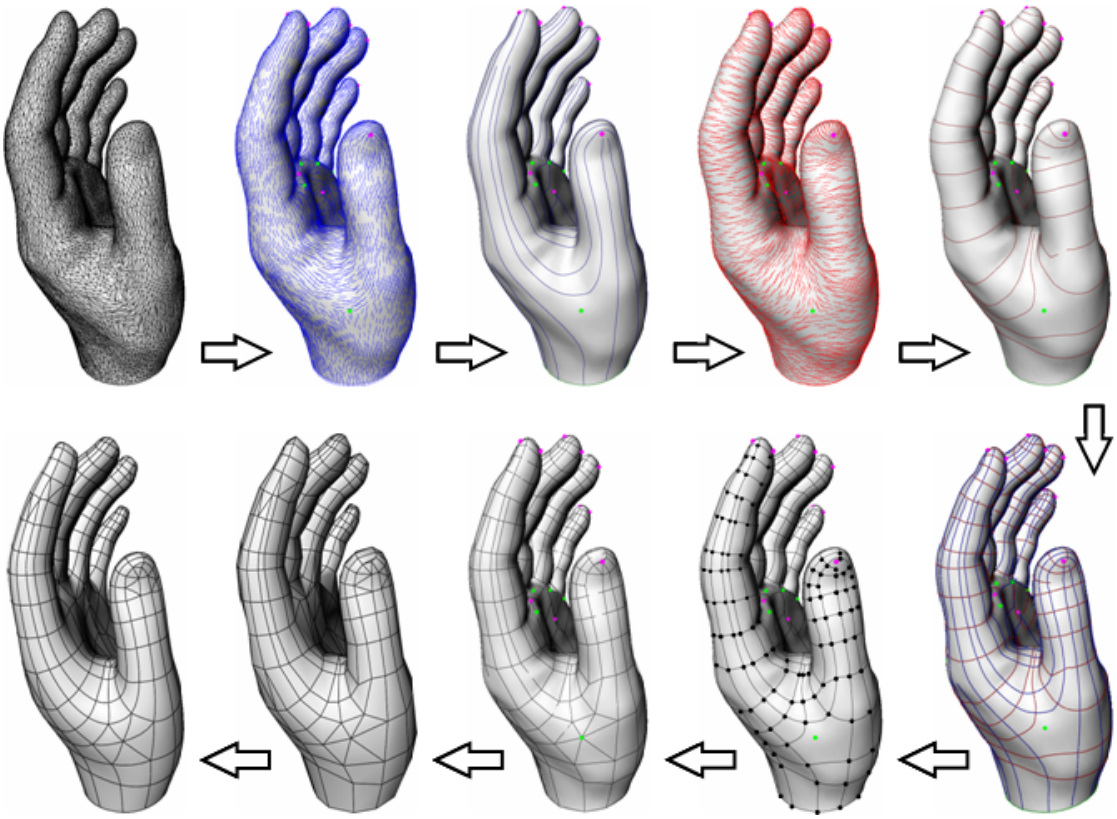


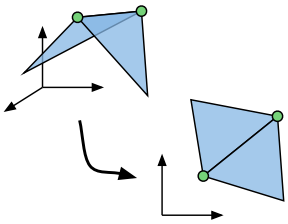
Figure 10.9: Anisotropic remeshing: From an input triangulated geometry, the curvature tensor field is estimated, then smoothed, and its umbilics are deduced (colored dots). Lines of curvatures (following the principal directions) are then traced on the surface, with a local density guided by the principal curvatures, while usual point-sampling is used near umbilic points (spherical regions). The final mesh is extracted by subsampling, and conforming-edge insertion. The result is an anisotropic mesh, with elongated quads aligned to the original principal directions, and triangles in isotropic regions.

integration of streamlines. The third stage deduces the vertices of the new mesh by intersecting the lines of curvatures in anisotropic areas and by selecting a subset of the umbilics in isotropic areas (estimated to be spherical). The edges are obtained by straightening the lines of curvatures in-between the newly extracted vertices in anisotropic areas, and deduced from the Delaunay triangulation in isotropic areas. The final output is a polygon mesh with mostly elongated quadrilateral elements in anisotropic areas, and triangles on isotropic areas. Quads are placed mostly in regions with two estimated axis of symmetry, while triangles are used to either tile isotropic areas or to generate conforming convex polygonal elements. In flat areas the infinite spacing of streamlines will not produce any polygons, except for the sake of convex decomposition.

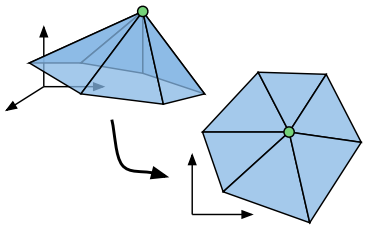
Marinov and Kobbelt [?] propose a variant of Alliez et al.'s algorithm, that differs from the original work in two aspects (see Fig. ??):

- Curvature line tracking and meshing are all done in 3D space: There is no need to compute a global parameterization such that objects of arbitrary genus can be processed.
- The algorithm is able to compute a quad-dominant, anisotropic remesh even in flat regions of the model, where there are no reliable curvature estimates by extrapolating directional information from neighboring anisotropic regions.

In addition to mere curvature directions, a confidence value for each face and vertex of the input mesh is estimated as well. The estimate is based on the coherence of the principal directions at the face's vertices. This confidence estimate is then used to propagate the curvature tensors from regions of high confidence (highly curved regions) into regions of low confidence (flat regions and noisy regions). Curvature lines are traced directly on the 3D mesh, i.e., at any time a line sample position is identified by a tuple $(f, (u, v, w))$ where f is the index of a triangle and (u, v, w) are the barycentric coordinates of the sample within that triangle. To advance the current sample point, the face f and its neighborhood are locally flattened, either by a hinge map (if the curvature line crosses an edge of f) or by a polar map (if the curvature line crosses one of f's vertices).



Hinge map



Polar map

When the traced line enters a region of low confidence, the algorithm switches the tracing mode: Instead of integrating along the principal curvatures, the line is simply extrapolated from its last sample points along a geodesic curve until it enters a region of high confidence again. At this point the line is then "snapped" to the most similar principal curvature direction.

Due to the strong visual and structural importance of curvatures, remeshing algorithms that track these lines produce results that are similar to those that would have been created by a human designer. However, reliably estimating and tracking the principal curvatures on a discrete triangle mesh is not that easy, in particular for coarse or noisy meshes. Alliez et al.'s algorithm out-sources most of the computationally hard work to a constrained Delaunay triangulation (e.g., the one provided by CGAL) by globally paramaterizing the whole input model. Apart from being hard to compute for large models, a global parameterization restricts the inputs to genus-0 manifolds with a single boundary loop. Higher genus objects have to be cut open along each handle. The approach of Marinov et al. is parameterization-free and has no restrictions on the topology of the input model. However, the extraction of the final mesh might lead to non-manifold configurations that have to be handled and fixed in a post-processing step.

Minimum bending energy curves. In [?] a quad-dominant remeshing algorithm is proposed that exploits the mesh segmentation $\mathcal{R}$ produced by the algorithm described in [?]. First sample points

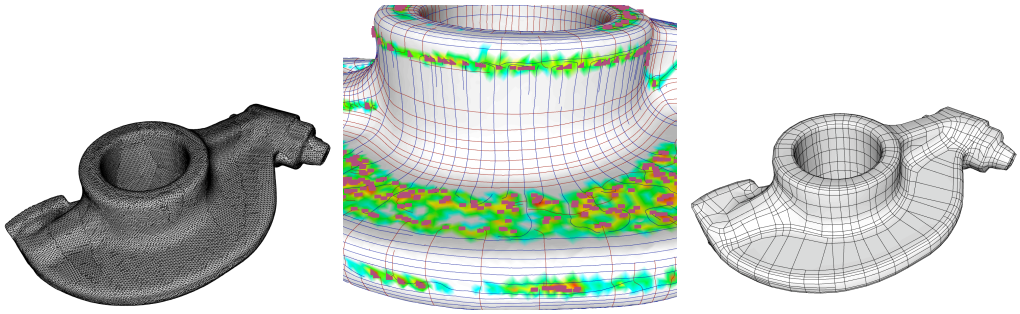


Figure 10.10: Quad-dominant remeshing. Left: The input is a manifold triangle mesh. Middle: In regions of low confidence, the curvature lines are not well-defined. The algorithm bridges these regions by extrapolation and produces the result on the right.

are uniformly distributed on the boundaries of the patches $\mathcal{R}_i$ and each patch is parameterized over a 2D domain. There, each pair of sample points is connected by a cubic curve that minimizes its bending energy. A discrete optimization algorithm selects a subset of the cubics that produces the most well-shaped elements. The resulting quad-dominant mesh is then projected back to 3D. This algorithm is able to bridge flat, isotropic or noisy regions of the input mesh in a robust manner, see Fig. **??**.

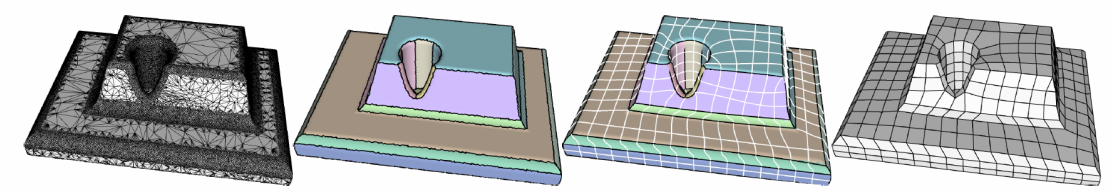


Figure 10.11: Quadrilateral remeshing by segmentation and fitting of minimum-bending curves.

10.2.2 Contouring

In our taxonomy a quad-dominant remeshing technique based upon contouring consists of computing two scalar functions, and extracting a quadrangle tiling by contouring these functions along well-chosen isovalues. Although any mesh parameterization technique can be used for computing quadrangle surface tilings by contouring isoparameter lines, we wish to focus on the techniques which allows the user to control the alignment of the mesh edges, and refer the reader to Chapter **??** otherwise.

Dong et al. [**?**] propose a hybrid technique which combines contouring of a function, and placement of streamlines. A harmonic function is computed on the input mesh, then drawing isocontours of this function, and placing a set of orthogonal streamlines results in a good quad remesh (see Fig. **??**). The few T-junctions which remain due to the termination of streamlines are removed by adding conforming edges. Examples are shown where harmonic line singularities are specified by the user in order to locally control the alignment of the mesh edges.

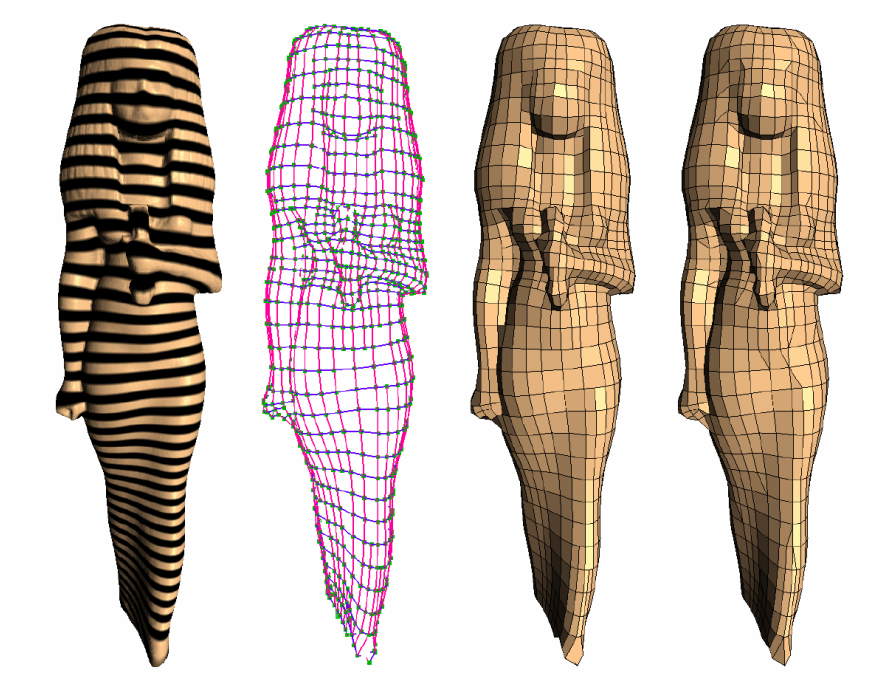


Figure 10.12: Quadrilateral remeshing of arbitrary manifolds: A harmonic function is computed over the manifold. A set of crossings along each flow line is constructed. A non-conforming mesh is extracted from this net of flow crossings. A post-process produces a conforming mesh composed solely of triangles and quadrilaterals. Figure taken from [**?**].

Ray et al. [**?**] introduce another contouring technique performing a non-linear optimization of periodic parameters to best align directions along two given orthogonal vector fields, offering more freedom on the type of singularities than any previous approach. In particular, indices of type 1/2 and 1/4 can be introduced, allowing a balance between area distortion and alignment control. A curl-correction step modulating the norm of the vector field is used to minimize the number of point singularities, hence the number of irregular vertices in the final remesh. When the input vector fields are derived from estimated principal curvature directions, this technique generates high quality meshes both automatically and efficiently (see Fig. **??**).

Dong et al. [**?**] propose to generate quadrangle tilings by computing Laplacian eigenfunctions of the input triangle mesh. A Morse-Smale complex is extracted from one Laplacian eigenvector, and refined using a globally smooth parameterization technique. The results show that the eigenfunctions distribute their extrema evenly across the mesh, which generates reasonably uniform quadrangulations (see Fig. **??**). The mesh density can be adjusted by selecting the appropriate Laplacian eigenvalue.

Given a triangle surface mesh and a user-defined singularity graph, Tong et al. [**?**] compute two harmonic scalar functions, whose isolines tile the input surface into quadrangles. The key idea is to extend the discrete Laplace operator which encompasses several types of line singularities. The resulting two discrete differential 1-forms are either regular, opposite, or switched along the singularity graph edges. This modification is shown to guarantee the continuity of the isolines across the lines, while the locations of the isolines themselves depend on the global solution to the modified Laplace equation over the whole surface (see Fig. **??**). The idea originates from

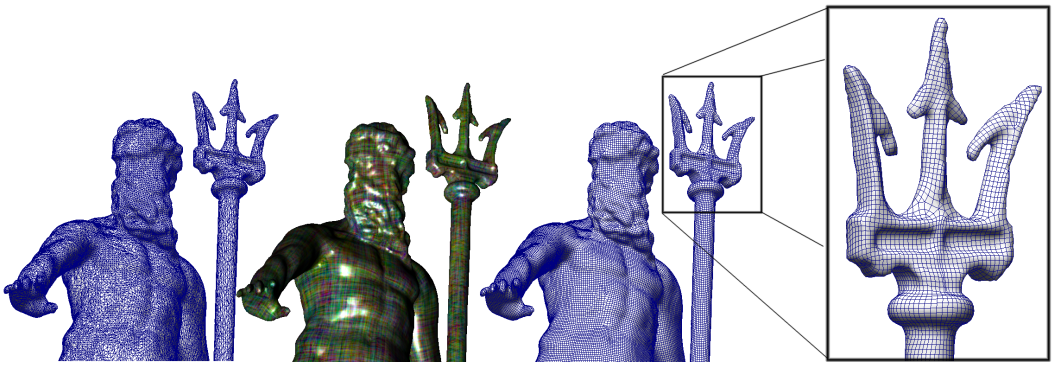


Figure 10.13: Left: input triangle mesh. Middle: estimated curvature directions. Right: quad-dominant tiling.

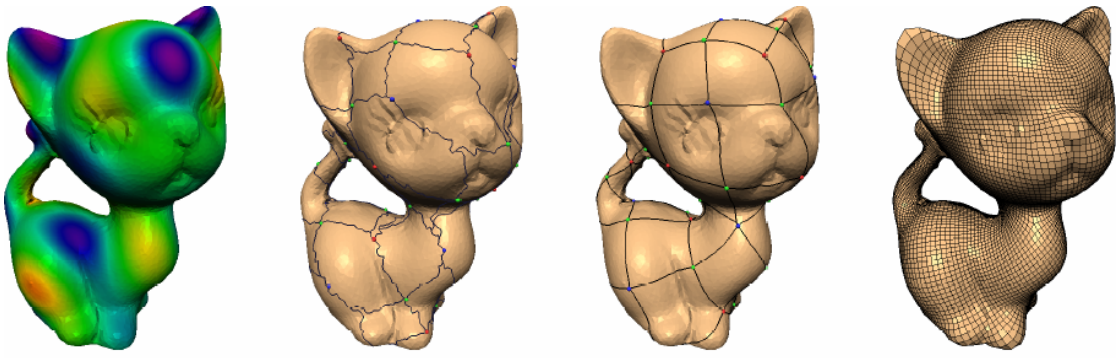


Figure 10.14: Spectral surface quadrangulation. Figure taken from [**?**].

the observation that it is sufficient to search for two scalar functions whose *union of isocontours* match across the line singularities. Design flexibility is provided through specification of the type of each line singularity of the graph, as well as the number of isolines along independent meta-edges to control quad sizes.

Given a triangle surface mesh, a frame field and a user-defined cut graph, Kälberer et al. [**?**] generate a quadrangulation whose edges best align to the input frame field. The key idea herein is to construct a 4-branched covering of the input triangle mesh from the cut graph, such that the input frame field can be converted into an integrable vector field. The latter conversion is performed by Hodge decomposition on the covering space, which leads to a locally integrable (harmonic) vector field. The final quadrangulation is obtained by contouring the two associated piecewise linear harmonic functions (see Fig. **??**).

Discussion. All methods based upon contouring of scalar functions are in general robust and well suited to generate quadrangulations with no T-junctions. The periodic global parameterization approach proposed by Ray et al. [**?**] currently provides the best automatic solution to generated quadrangulations with controlled alignment of the mesh edges. Although the notion of harmonic line singularity proposed by Tong et al. [**?**] advances the knowledge for further control over edge alignment, it requires a user-defined singularity graph. The next challenge would be to turn this approach into a fully automatic algorithm, ultimately where the design of the singularity graph itself would be part of an optimization process.

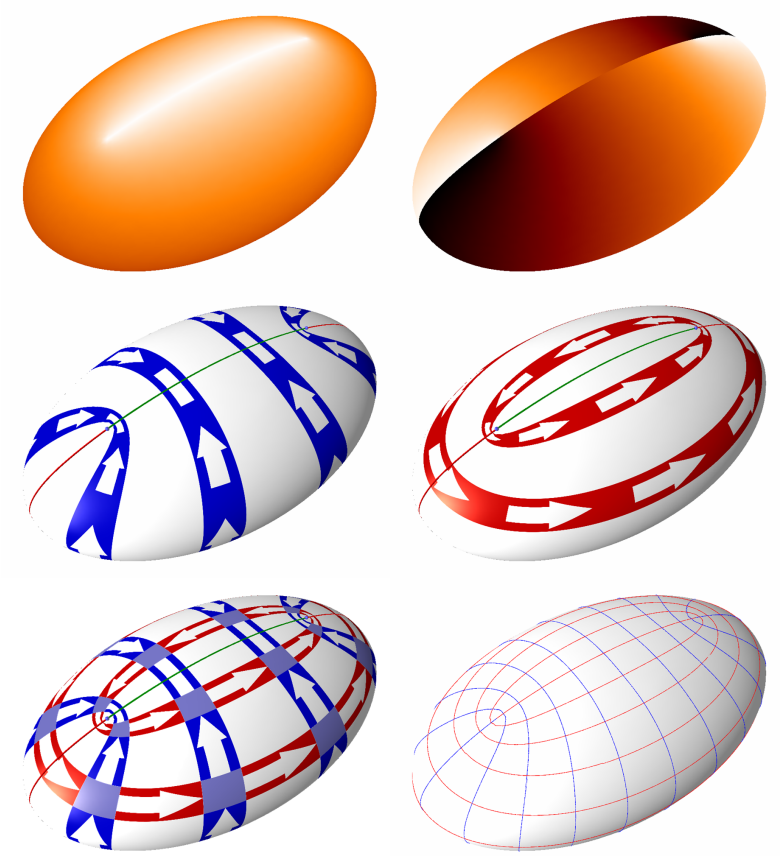


Figure 10.15: Quadrangle surface tiling through contouring. Top: Two harmonic potentials with user-defined line singularities. Middle: Pair of 1-forms associated to the potentials. Bottom right: Isocontouring these potentials results in a quadrangle tiling.

10.3 Error-Driven

Error-driven remeshing amounts to generating meshes which maximize the trade-off between complexity and accuracy. The complexity is expressed in terms of the number of mesh elements, while the geometric accuracy is measured relative to the input mesh and according to a predefined distortion error measure. The efficiency of a mesh is qualified by the error per element ratio (the smaller, the better). One usually wants to minimize the approximation error for a given budget of elements, or conversely, minimize the number of elements for a given error tolerance.

Efficient representation of complex shapes is of fundamental importance, in particular for applications dealing with digital models generated by laser scanning or isosurfacing of volume data. This is mainly due to the fact that the complexity of numerous algorithms is proportional to the number of mesh primitives. Examples of related applications are modeling, processing, simulation, storage, or transmission. Even for most rendering algorithms, polygon count is still the main bottleneck. Being able to automatically adapt the newly generated mesh to the local shape complexity is of crucial importance in this context. Mesh simplification or refinement methods are obvious ways for generating efficient meshes. We refer the reader to Chapter **??** for a treatment of this topic. In this section we focus on techniques that are specifically designed

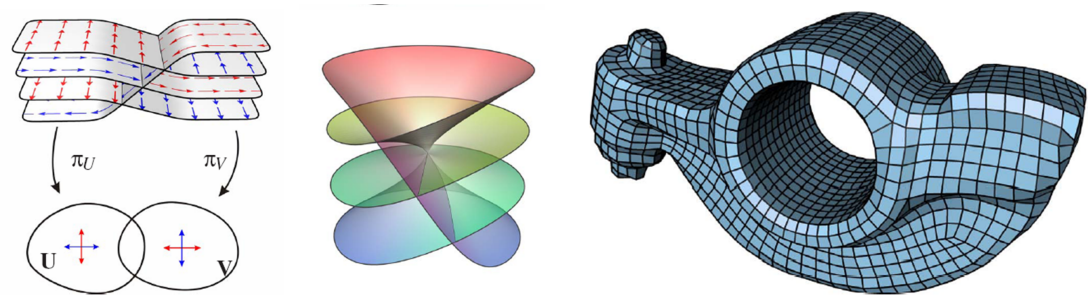


Figure 10.16: QuadCover. Left: A frame field lifted to a vector field on the covering. Middle: Branch point of the covering. Right: Final quadrangulation.

to exploit a shape's local planarity, symmetry, and features in order to optimize its geometric representation.

Variational shape approximation (VSA) is a relatively new approach to remeshing (and to shape approximation in general) introduced by Cohen-Steiner et al. [**?**]. VSA is highly sensitive to features and symmetries and produces anisotropic remeshings of high approximation quality. In VSA the input shape is approximated by a set of proxies. The approximation error is iteratively decreased by clustering faces into best-fitting regions. In contrast to the remeshing methods presented in the previous sections, VSA does not require a parameterization of the input or local estimates of differential quantities. Apart from remeshing, VSA techniques can also be used in mesh segmentation.

Let $\mathcal{M}$ be a triangle mesh and let $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_k\}$ be a partition of $\mathcal{M}$ into k regions, i.e., $\mathcal{R}_i \subset \mathcal{M}$ and

$$\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_k = \mathcal{M} \ .$$

Furthermore let $\mathcal{P} = \{P_1, \ldots, P_k\}$ be a set of *proxies*. A proxy $P_i = (\mathbf{x}_i, \mathbf{n}_i)$ is simply a plane in space through the point $\mathbf{x}_i$ with normal direction $\mathbf{n}_i$. Cohen-Steiner et al. consider two metrics that measure a generalized distance of a region $\mathcal{R}_i$ to its proxy P_i. The standard $\mathcal{L}^2$ metric is defined as

$$\mathcal{L}^2(\mathcal{R}_i, P_i) = \int_{\mathbf{x} \in \mathcal{R}_i} ||\mathbf{x} - \pi_i(\mathbf{x})||^2 \, \mathrm{d}\mathbf{x} \ ,$$

where $\pi_i(\mathbf{x}) = \mathbf{x} - \mathbf{n}_i \mathbf{n}_i^T (\mathbf{x} - \mathbf{x}_i)$ is the orthogonal projection of $\mathbf{x}$ onto P_i. They also introduce a new shape metric $\mathcal{L}^{2,1}$ that is based on a measure of the normal field

$$\mathcal{L}^{2,1}(\mathcal{R}_i, P_i) = \int_{\mathbf{x} \in \mathcal{R}_i} ||\mathbf{n}(\mathbf{x}) - \mathbf{n}_i||^2 \, \mathrm{d}\mathbf{x} \ .$$

The goal of variational shape approximation is then the following: Given a number k and an error metric E (i.e., either $E = \mathcal{L}^2$ or $E = \mathcal{L}^{2,1}$) find a set $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_k\}$ of regions and a set $\mathcal{P} = \{P_1, \ldots, P_k\}$ of proxies such that the global distortion

$$E(\mathcal{R}, \mathcal{P}) = \sum_{i=1}^{k} E(\mathcal{R}_i, P_i) \tag{10.3}$$

is minimized. For remeshing purposes one can then extract a remesh of the original input from the proxies.

In the following we describe and compare two algorithms for computing an (approximate) minimum of Eq. (**??**). The first algorithm is due to Cohen-Steiner et al. and uses Lloyd-clustering to produce the regions $\mathcal{R}_i$. The second method is a greedy approximation to VSA with additional injectivity guarantees.

10.3.1 Variational

Cohen-Steiner et al. [**?**] use a method to minimize Eq. (**??**) that is inspired by Lloyd's clustering algorithm, which has been used for mesh segmentation in [**?**]. The algorithm iteratively alternates between a *geometry partitioning phase* and a *proxy fitting* phase. In the geometry partitioning phase the algorithm computes a set of regions that best fit a given set of proxies. In the proxy fitting phase, the partitioning is kept fixed and the proxies are adjusted.

Geometry partitioning In the geometry partitioning phase, the algorithm modifies the set $\mathcal{R}$ of regions to achieve a lower approximation error Eq. (**??**) while keeping the proxies $\mathcal{P}$ fixed. It does so by first selecting a number of seed triangles and then greedily growing new regions $\mathcal{R}_i$ around these seeds.

First the algorithm picks the triangle t_i from each region $\mathcal{R}_i$ that is most similar to its associated proxy P_i. This can easily be done by iterating once over all triangles t in $\mathcal{R}_i$ and finding the one that minimizes $E(t, P_i)$.

After initializing $\mathcal{R}_i = \{t_i\}$, the algorithm simultaneously grows the sets $\mathcal{R}_i$. A priority queue contains candidate pairs (t, P_i) of triangles and proxies. The priority of a triangle/proxy pair (t, P_i) is naturally given as $E(t, P_i)$. For each seed triangle t_i its neighboring triangles r are found and the pairs (r, P_i) are inserted into the queue. The algorithm then iteratively removes pairs (t, P_i) from the queue, checks whether t has already been conquered by the region growing process, and if not assigns t to $\mathcal{R}_i$. Again the unconquered neighbor triangles r of t are selected and the pairs (r, P_i) are inserted to the queue. This process is iterated until the queue is empty and all triangles are assigned to a region. Note that a given triangle can appear up to three times simultaneously in the queue. One could of course check for each triangle, whether it already is in the queue and if so take appropriate measures. Instead of this expensive check the algorithm rather keeps a status bit *conquered* for each triangle and checks this bit before assigning a triangle to a region. The following pseudo-code summarizes the geometry partitioning procedure:

partition($\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_k\}, \mathcal{P} = \{P_1, \ldots, P_k\}$)

```
// find the seed triangles and initialize the priority queue
queue = ∅
for i = 1 to k do
    select the triangle t ∈ R_i that minimizes E(t, P_i)
    R_i = {T}
    set t to conquered
    for all neighbors r of t do
        insert (r, P_i) into queue

// grow the regions
while the queue is not empty do
    get next (t, P_i) from the queue
    if t is not conquered then
        set t to conquered
```

$\mathcal{R}_i = \mathcal{R}_i \cup \{t\}$
for all neighbors r of t **do**
  if r is not conquered **then**
    insert (r, P_i) into queue

To initialize the algorithm one randomly picks k triangles $t_1, \ldots, t_k$ on the input model, sets $\mathcal{R}_i = \{t_i\}$ and initializes $P_i = (\mathbf{x}_i, \mathbf{n}_i)$ where $\mathbf{x}_i$ is an arbitrary point on t_i and $\mathbf{n}_i$ is t_i's normal. Then regions are grown as in the geometry partitioning phase.

Proxy fitting In the proxy fitting phase, the partition $\mathcal{R}$ is kept fixed while the proxies $P_i = (\mathbf{x}_i, \mathbf{n}_i)$ are adjusted in order to minimize Eq. (**??**). For the $\mathcal{L}^2$ metric the best proxy is the area weighted least-squares fitting plane. It can be found using standard principal component analysis. When using the $\mathcal{L}^{2,1}$ metric, the proxy normal $\mathbf{n}_i$ is just the area-weighted average of the triangle normals. The base point $\mathbf{x}_i$ is irrelevant for $\mathcal{L}^{2,1}$, but is set to the barycenter of $\mathcal{R}_i$ for remeshing purposes.

Extracting the remesh From an optimal partitioning $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_k\}$ and corresponding proxies $\mathcal{P} = \{P_1, \ldots, P_k\}$ one can now extract an anisotropic remesh as follows: First, all vertices in the original mesh that are adjacent to three or more different regions are identified. These vertices are projected onto each proxy and their average position is computed. These so-called anchor vertices are then connected by tracing the boundaries of the regions $\mathcal{R}$. The resulting faces are triangulated by performing a "discrete" Delaunay triangulation, see example Fig. **??**).

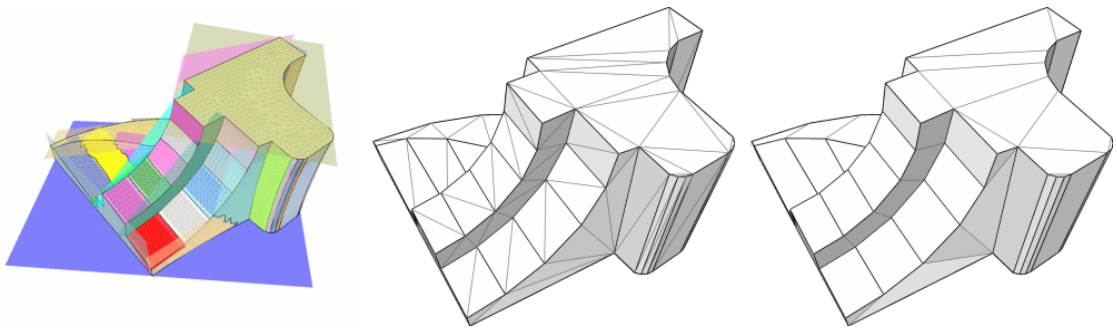


Figure 10.17: Variational Shape Approximation applied to the fandisk model.

Generalizations In [**?**] the variational shape approximation approach is taken a step further by allowing for proxies other than simple planes, e.g., spheres, cylinders, and rolling-ball blends. Apart from requiring fewer primitives to achieve a certain fitting approximation, this method can also recover the "semantic structure" of an input model to some extend, see Fig. **??**. In [**?**] a similar idea is used to decompose the input mesh into nearly developable segments.

Another recent extension of this algorithm to handle general quadric proxies has been elaborated by Yan et al. [**?**]. Faithful approximations are obtained using fewer proxies than when using planar proxies, see Fig. **??**.

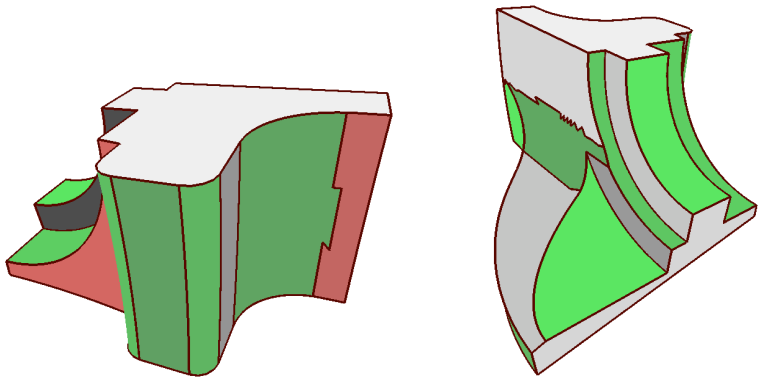


Figure 10.18: Hybrid Variational Surface Approximation: In addition to planes, Wu and Kobbelt also use more general proxies like spheres, cylinders, and rolling ball blends. These proxies allow to recover the semantic structure of the input model.

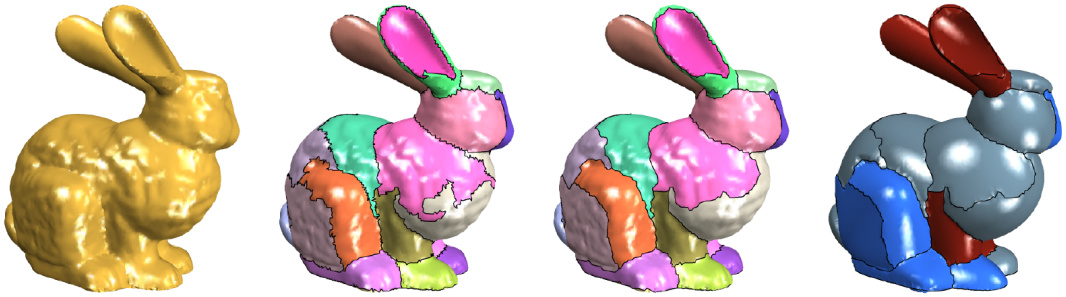


Figure 10.19: Stanford bunny approximated by 28 quadric proxies. Figure taken from [**?**].

10.3.2 Greedy

In [**?**] a greedy algorithm to compute an approximate minimum of Eq. (**??**) is proposed (see Fig. **??**). It's main advantages are:

- The algorithm naturally generates a multi-resolution hierarchy of shape approximations (Fig. **??**).
- The output is guaranteed to be free of fold-overs and degenerate faces.

On the downside, due to its greedy approach, it is more likely that the algorithm gets stuck in a local minimum (although this is rarely observed in practice). Furthermore, its implementation is involved and requires the robust computation of Delaunay triangulations.

Setup In addition to the partition $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_k\}$ and the proxies $\mathcal{P} = \{P_1, \ldots, P_k\}$, the algorithm maintains a set of polygonal faces $\mathcal{F} = \{f_1, \ldots, f_k\}$. Each face f_i can be an arbitrary connected polygon, i.e., it has an outer boundary and possibly a number of inner boundaries around interior holes. At the beginning of the algorithm we initialize the sets $\mathcal{R}, \mathcal{P}$, and $\mathcal{F}$ as follows:

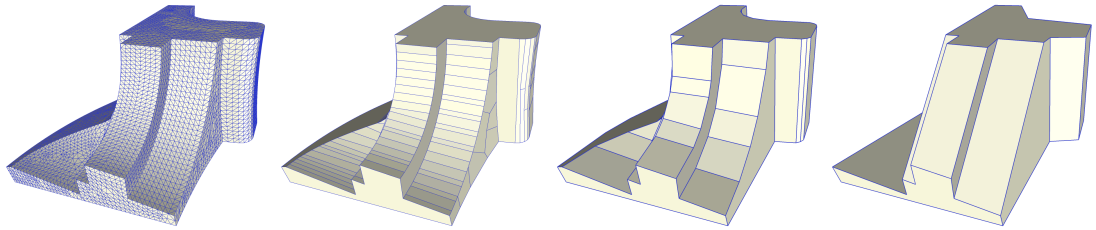


Figure 10.20: A multi-resolution hierarchy of differently detailed meshes that was created by variational shape approximation.

- $\mathcal{R}_i = \{t_i\}$, i.e., each triangle makes up a region on its own.
- The proxy of $\mathcal{R}_i$ is set to $P_i = (\mathbf{x}_i, \mathbf{n}_i)$ where $\mathbf{x}_i$ is an arbitrary point on t_i and $\mathbf{n}_i$ is t_i's normal.
- $f_i = t_i$, in particular the projection of f_i onto P_i is injective.

Algorithm Invariant The goal of the algorithm is to guarantee a valid shape approximation that is free of fold-overs and degenerate faces. This is achieved by maintaining the following invariant at all times during the run of the algorithm:

> *Injectivity constraint:* The projection of f_i onto P_i is injective.

Note that the initial settings for the sets $\mathcal{R}$, $\mathcal{P}$, and $\mathcal{F}$ satisfy this constraint.

Due to the injectivity constraint, one is able to extract a valid triangle mesh at all times during the run of the algorithm. To produce a triangulation $\mathcal{D}_i$ of a face f_i one simply projects f_i onto P_i (which is a plane), performs a (planar) constrained Delaunay triangulation there, and lifts the triangles of the Delaunay triangulation back to f_i.

Greedy Optimization The partitioning is now greedily optimized in a loop that stops when a predefined maximum error or a predefined number of regions is reached. In each iteration one selects (subject to the injectivity constraint) two regions $\mathcal{R}_i$ and $\mathcal{R}_j$ and merges them into a new region $\mathcal{R}' = \mathcal{R}_i \cup \mathcal{R}_j$. (The order in which the merging is performed is described in the next paragraph.) Then a new proxy $P' = (\mathbf{x}', \mathbf{n}')$ is computed as an area-weighted average of P_i and P_j

$$\mathbf{n}' = \frac{a_i\mathbf{n}_i + a_j\mathbf{n}_j}{||a_i\mathbf{n}_i + a_j\mathbf{n}_j||} \quad \text{and} \quad \mathbf{x}' = \frac{a_i\mathbf{x}_i + a_j\mathbf{x}_j}{a_i + a_j} \ ,$$

where $a_i = area(\mathcal{R}_i)$. Finally, a new face f' is computed by identifying and removing the common boundary edges of f_i and f_j. The algorithm then checks for valence two vertices: If it finds an interior valence two vertex, it is immediately removed. Boundary valence-two vertices are only removed, if their distance from the proxy is smaller than a user-defined threshold.

Note again, that all the operations described above (merging of faces, removal of valence two vertices) are only performed if the injectivity constraint is not violated by the operation!

Merge priorities For each adjacent pair $\mathcal{R}_i$ and $\mathcal{R}_j$ of regions we could compute the shape measure $E(\mathcal{R}', P')$ as described in Eq. (**??**) and order the region pairs by increasing shape error. In order to speed up the algorithm, the exact $\mathcal{L}^2$ measure is approximated by

$$L^2(f') = \mathcal{L}^2(\mathcal{D}_i, P') + \mathcal{L}^2(\mathcal{D}_j, P') \ .$$

Since $\mathcal{D}_i$ usually contains much less triangles than $\mathcal{R}_i$ this will significantly speed up the algorithm. The $\mathcal{L}^{2,1}$ error is replaced by

$$L^{2,1}(f') = a_i||\mathbf{n}_i - \mathbf{n}'||^2 + a_j||\mathbf{n}_j - \mathbf{n}'||^2 \ ,$$

where $a_i = area(\mathcal{R}_i)$ as before. The two error measures are combined into a single, scale-independent measure

$$E(f') = \big(1 + L^2(f')\big) \cdot \big(1 + L^{2,1}(f')\big) \ ,$$

which does not require any user selected weight parameters.

Cohen-Steiner's algorithm is fast, efficient, and generally produces high quality results with low approximation error. However, the mesh extraction step might produce degenerate triangles and fold-overs. The extensions presented by Marinov produce a hierarchy of reconstructions which are guaranteed to be free of fold-overs. However, due to the greedy approach, Marinov's algorithm is more likely to get stuck in a local optimum. To achieve acceptable running times, they furthermore have to resort to an approximation of the true $\mathcal{L}^2$ or $\mathcal{L}^{2,1}$ errors.

10.4 Summary

We have provided a brief overview of surface remeshing techniques with focus on isotropic, quadrangle, and error-driven techniques.

Isotropic remeshing is now a well-studied problem, and robust software components are available for large meshes. Although the variational or pliant approaches generate the best practical results, the greedy technique based upon Delaunay refinement and filtering provides guarantees over the shape of the elements as well as other useful properties such as the absence of self-intersection. The latter property is often of crucial importance for, e.g., generating volumetric meshes for simulation.

Quadrangle remeshing has been investigated with various ideas ranging from mesh conversion to contouring of harmonic functions through placement of streamlines. We remark that only one technique [**?**] provides the user with control over the alignment of the mesh edges while being robust, versatile, and fully automatic. The variety of techniques proposed to tackle this problem reveals its intrinsic difficulty: quadrangle remeshing is a global problem by nature. The motivation for controlling the shape and alignment of the elements takes some of its roots in the approximation theory: it is known that optimal bilinear elements must align with the principal directions, with aspect ratios depending on the principal curvatures. A less known fact is that this assertion is true for elliptic areas only, the mesh edges of the optimal elements being aligned to the asymptotic directions (zero curvature lines) on hyperbolic areas. Some research must be carried on in order to elaborate upon automatic quadrangle remeshing techniques which would be driven only by the approximation error (instead of, e.g., fitting estimated curvature directions).

Recent work on error-driven remeshing propose to fit simple primitives ranging from planes to general quadrics through CAD primitives such as cylinders, spheres, or rolling ball patches. One recent trend is to elaborate upon methods which aim at recovering the global structure of

the initial surface for modeling or high level geometry processing applications. For the quadric-fitting approach [**?**] it is interesting to notice how the initial remeshing goal joins the "resurfacing" problem for reverse engineering, and to some extend the shape recognition and reconstruction problem.

11 Shape Deformation

The field of interactive shape deformation has seen a lot of attention throughout the last years, with a large number of approaches having been proposed. It is a very challenging research field, since complex mathematical formulations (i) have to be hidden behind an intuitive user interface and (ii) have to be implemented in a sufficiently efficient and robust manner to allow for interactive applications. In this section we will give an overview of different kinds of shape deformation techniques, classify them into different categories, and show their interrelations.

We start by discussing surface-based deformation methods in Section **??**, roughly classified into (multiresolution) bending energy minimization (Section **??** and **??**) and differential coordinates (Section **??**). We point out the inherent limitations of linear(ized) deformation approaches (Section **??**), which are avoided by fully nonlinear techniques (Section **??**). Finally we also give an overview of linear and nonlinear space deformation techniques in Section **??**.

11.1 Surface-Based Deformation

In the case of *surface-based* deformations we are looking for a displacement function $\mathbf{d} : \mathcal{S} \rightarrow \mathbb{R}^3$ that maps the given surface $\mathcal{S}$ to its deformed version $\mathcal{S}'$:

$$\mathcal{S}' \ := \ \{\mathbf{p} + \mathbf{d}\,(\mathbf{p}) \ | \ \mathbf{p} \in \mathcal{S}\} \quad .$$

In particular in engineering applications, exact control of the deformation process is crucial, i.e., one has to be able to specify displacements for a set of constrained points $\mathcal{C}$:

$$\mathbf{d}\,(\mathbf{p}_i) = \mathbf{d}_i \ , \quad \forall\, \mathbf{p}_i \in \mathcal{C} \ .$$

Since we are targetting interactive shape deformations, another important aspect is the amount of user interaction required to specify the desired deformation function $\mathbf{d}$.

11.1.1 Tensor-Product Spline Surfaces

The traditional surface representation in CAGD are spline surfaces (see Section **??**). They are controlled by an intuitive control point metaphor and yield high quality smooth surfaces. A single tensor-product spline patch is defined as

$$\mathbf{f}\,(u,v) \ = \ \sum_{i=0}^{m} \sum_{j=0}^{m} \mathbf{c}_{ij} N_i^n(u) N_j^n(v) \ ,$$

i.e., each control point $\mathbf{c}_{ij}$ is associated with a smooth basis function $N_{ij}\,(u,v) := N_i^n(u) N_j^n(v)$. A translation of a control point $\mathbf{c}_{ij}$ therefore adds a smooth bump of rectangular support to the

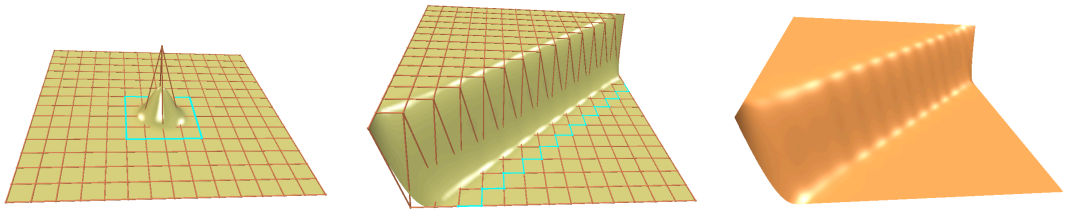


Figure 11.1: A modeling example using a bi-cubic tensor-product spline surface. Each control point is associated with a smooth basis function of fixed rectangular support (*left*). This fixed support and the fixed regular placement of the control points, resp. basis functions, prevents a precise support specification (*center*) and can lead to alias artifacts in the resulting surface, that are revealed by more sensitive surface shading (*right*).

surface (cf. Fig. **??**, left). Every more sophisticated modeling operation has to be composed from such smooth elementary modifications, such that the displacement function has the form

$$\mathbf{d}\,(u,v) \;=\; \sum_{i=0}^{m}\sum_{j=0}^{m} \delta\mathbf{c}_{ij} N_{ij}\,(u,v) \quad ,$$

where $\delta\mathbf{c}_{ij}$ denotes the change of the control point $\mathbf{c}_{ij}$. The support of the deformation is the union of the supports of individual basis functions. As the positions of the basis functions are fixed to the initial grid of control points, this prohibits a fine-grained control of the desired support region. Moreover, the composition of fixed basis functions located on a fixed grid might lead to alias artifacts in the resulting surface, as shown in Fig. **??**.

It was also shown in Section **??** that tensor-product spline surfaces are restricted to rectangular domains, and that complex surfaces therefore have to be composed by a large number of spline patches. Specifying complex deformations in terms of control point movements thus involves a lot of user interaction, since smoothness constraints across patch boundaries have to be considered during the deformation process. Also notice that prescribing constraints $\mathbf{d}\,(u_i,v_i) = \mathbf{d}_i$ requires to solve a linear system for the control point displacements $\delta\mathbf{c}_{ij}$. These systems can be over- as well as under-determined, and hence are typically solved by least squares and least norm techniques. However, in the first case, the system cannot be solved exactly, and in the latter case the minimization of control point displacements does not necessarily lead to fair deformations, which would require to minimize some fairness energy (Section **??**).

11.1.2 Transformation Propagation

The main drawback of spline-based deformations is that the underlying mathematical surface representation is identical to the basis functions that are used for the surface deformation. To overcome this limitation, the deformation basis functions consequently should be independent of the actual surface representation.

A popular approach falling into this category works as follows (cf. Fig. **??**): In a first step the user specifies the support of the deformation (the region which is allowed to change) and a handle region $\mathcal{H}$ within it. The handle region is directly deformed using any modeling interface, and its transformation is smoothly interpolated within the support region in order to blend between the transformed handle $\mathcal{H}$ and the fixed part $\mathcal{F}$ of the surface. This smooth blend is controlled by a scalar field $s : \mathcal{S} \to [0,1]$, which is 1 at the handle (full deformation), 0 outside the support (no

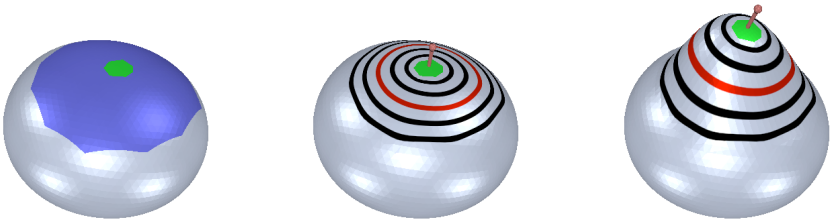


Figure 11.2: After specifying the blue support region and the green handle regions (*left*), a smooth scalar field is constructed that is 1 at the handle and 0 outside the support (*center*). This scalar field is used to propagate and damp the handle's transformation (*right*).

deformation), and smoothly blends between 1 and 0 within the support region. A typical way to construct such a scalar field is to compute geodesic (or Euclidean) distances $\text{dist}_{\mathcal{F}}(\mathbf{p})$ and $\text{dist}_{\mathcal{H}}(\mathbf{p})$ from $\mathbf{p}$ to the fixed part $\mathcal{F}$ and the handle region $\mathcal{H}$, respectively, and to define

$$s(\mathbf{p}) = \frac{\text{dist}_{\mathcal{F}}(\mathbf{p})}{\text{dist}_{\mathcal{F}}(\mathbf{p}) + \text{dist}_{\mathcal{H}}(\mathbf{p})} ,$$

similar to [**?**, **?**]. This scalar field can further be enhanced by a transfer function $t(s(\mathbf{p}))$, which provides more control of the blending process. The damping of the handle transformation is then performed separately on the rotation, scale/shear, and translation components, for instance like in [**?**]. In case the individual transformation components are not given, they can be computed by polar decomposition [**?**].

As shown in Fig. **??**, the major problem with this approach is that the distance-based propagation of transformations will typically not result in the geometrically most intuitive solution. This would require the smooth interpolation of the handle transformation by the displacement function $\mathbf{d}$, while otherwise minimizing some fairness energies.

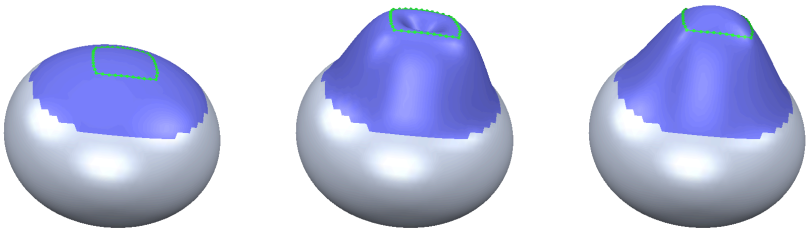


Figure 11.3: A sphere is deformed by lifting a closed handle polygon (*left*). Propagating this translation based on geodesic distance causes a dent in the interior of the handle polygon (*center*). The more intuitive solution of a smooth interpolation (*right*) cannot be achieved with this approach; it was produced by variational energy minimization (Section **??**).

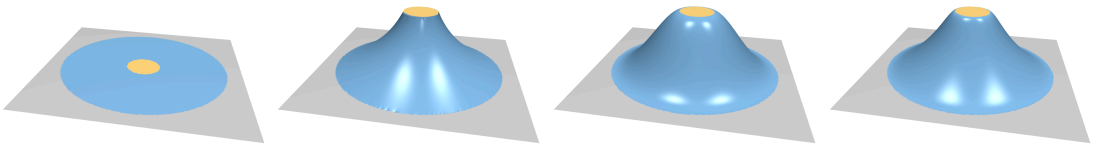


Figure 11.4: The surface $\mathcal{S}$ (*left*) is edited by minimizing its deformation energy subject to user-defined constraints that fix the gray part $\mathcal{F}$ of the surface and prescribe the transformation of the yellow handle region $\mathcal{H}$. The deformation energy (**??**) consists of stretching and bending terms, and the examples show pure stretching with $k_s = 1$, $k_b = 0$ (*center left*), pure bending with $k_s = 0$, $k_b = 1$ (*center right*), and a weighted combination with $k_s = 1$, $k_b = 10$ (*right*).

11.1.3 Variational Energy Minimization

More intuitive surface deformations $\mathbf{d}$ with prescribed geometric constraints $\mathbf{d}(\mathbf{p}_i) = \mathbf{d}_i$ can be modeled by minimizing physically-inspired elastic energies. The surface is assumed to behave like a physical skin that stretches and bends as forces are acting on it. Mathematically, this behavior can be captured by an energy functional that penalizes both stretching and bending.

As introduced in Section **??** and Section **??**, the first and second fundamental forms, $\mathbf{I}(u, v)$ and $\mathbf{II}(u, v)$, can be used to measure geometrically intrinsic (i.e., parameterization independent) properties of $\mathcal{S}$, such as lengths, areas, and curvatures. When the surface $\mathcal{S}$ is deformed to $\mathcal{S}'$, and its fundamental forms change to $\mathbf{I}'$ and $\mathbf{II}'$, the difference of the fundamental forms can be used as an elastic thin shell energy that measures stretching and bending [**?**]:

$$E_{\text{shell}}(\mathcal{S}') = \int_\Omega k_s \left\|\mathbf{I}' - \mathbf{I}\right\|^2 + k_b \left\|\mathbf{II}' - \mathbf{II}\right\|^2 \mathrm{d}u\mathrm{d}v\,. \tag{11.1}$$

The stiffness parameters k_s and k_b are used to control the resistance to stretching and bending, respectively. In a modeling application one would have to minimize the elastic energy (**??**) subject to user-defined deformation constraints. As shown in Fig. **??**, this typically means fixing certain surface parts $\mathcal{F} \subset \mathcal{S}$ and prescribing displacements for the so-called *handle* region(s) $\mathcal{H} \subset \mathcal{S}$.

However, this nonlinear minimization is computationally too expensive for interactive applications. It is therefore simplified and linearized by replacing the difference of fundamental forms by partial derivatives of the displacement function $\mathbf{d}$ [**?**, **?**]:

$$\tilde{E}_{\text{shell}}(\mathbf{d}) = \int_\Omega k_s \left(\|\mathbf{d}_u\|^2 + \|\mathbf{d}_v\|^2\right) + k_b \left(\|\mathbf{d}_{uu}\|^2 + 2\,\|\mathbf{d}_{uv}\|^2 + \|\mathbf{d}_{vv}\|^2\right) \mathrm{d}u\mathrm{d}v\,, \tag{11.2}$$

where we use the notation $\mathbf{d}_x = \frac{\partial}{\partial x}\mathbf{d}$ and $\mathbf{d}_{xy} = \frac{\partial^2}{\partial x \partial y}\mathbf{d}$. For the efficient minimization of (**??**) we apply variational calculus, which yields the corresponding *Euler-Lagrange* equations that characterize the minimizer of (**??**), again subject to user constraints:

$$-k_s\,\Delta_{\mathcal{S}}\mathbf{d} + k_b\,\Delta_{\mathcal{S}}^2\mathbf{d} = \mathbf{0}\,. \tag{11.3}$$

Notice that for the second derivatives in (**??**) to closely approximate surface curvatures (i.e., bending), the parameterization $\mathbf{p} : \Omega \to \mathcal{S}$ should be as close to isometric as possible. Therefore Ω is typically chosen to equal $\mathcal{S}$, such that $\mathbf{d} : \mathcal{S} \to \mathbb{R}^3$ is defined on the manifold $\mathcal{S}$ itself. As a consequence, the Laplace operator in (**??**) corresponds to the Laplace-Beltrami operator (see Section **??**). Notice that the variational minimization of stretching and bending energies is closely related to the minimization of surface area and surface curvature introduced in Section **??**

in the context of mesh fairing. The difference is that now the displacement function $\mathbf{d}$ (instead of the coordinate function $\mathbf{p}$) minimizes certain fairness energies.

The order k of partial derivatives in the energy (**??**) or in the corresponding Euler-Lagrange equations $(-1)^k \, \Delta_{\mathcal{S}}^k \mathbf{d} = 0$ defines the maximum continuity C^{k-1} for interpolating displacement constraints [**?**]. Hence, minimizing (**??**) by solving (**??**) provides C^1 continuous surface deformations, as can also be observed in Fig. **??**. On a discrete triangle mesh, the C^1 constraints are defined by the first two rings of fixed vertices $\mathcal{F}$ and handle vertices $\mathcal{H}$.

Using the cotangent discretization of the Laplace-Beltrami defined in Section **??**, the Euler-Lagrange PDE (**??**) turns into a sparse bi-Laplacian linear system:

$$\begin{aligned} -k_s \, \Delta_{\mathcal{S}} \mathbf{d} \; + \; k_b \, \Delta_{\mathcal{S}}{}^2 \mathbf{d} \;\; &= \;\; \mathbf{0} \,, \qquad && \mathbf{p}_i \notin \mathcal{H} \cup \mathcal{F} \,, \\ \mathbf{d}\,(\mathbf{p}_i) \;\; &= \;\; \mathbf{d}_i \,, && \mathbf{p}_i \in \mathcal{H} \,, \\ \mathbf{d}\,(\mathbf{p}_i) \;\; &= \;\; \mathbf{0} \,, && \mathbf{p}_i \in \mathcal{F} \,. \end{aligned} \tag{11.4}$$

Interactively manipulating the handle region $\mathcal{H}$ changes the boundary constraints of the optimization, i.e., the right-hand side of the linear system Eq. (**??**). As a consequence, this system has to be solved in each frame. In Chapter **??** we will discuss efficient linear system solvers that are particularly suited for this multiple right-hand side problem. Also notice that restricting to *affine* transformation of the handle region $\mathcal{H}$ (which is usually sufficient) allows to precompute basis functions of the deformation, such that instead of solving (**??**) in each frame, only the basis functions have to be evaluated [**?**].

The approaches of [**?**] and [**?**] can be considered as instances of the framework described in this section, since both methods solve bi-Laplacian system to derive fair shape deformations. Other methods are conceptually similar, but achieve smooth deformations, for instance by hierarchical smoothing [**?**] or subdivision surfaces [**?**].

11.1.4 Multiresolution Deformation

The variational optimization techniques introduced in the last section provide C^1 continuous, smooth, and fair surface deformations. Interactive performance is achieved by simplifying or linearizing the nonlinear shell energy (**??**), such that the techniques become linear in the sense that they only require solving a linear system for the deformed surface $\mathcal{S}'$. However, as a consequence of this linearization, such methods typically do not correctly handle fine-scale surface details, as depicted in Fig. **??**. The local rotation of geometric details is an inherently nonlinear behavior, and hence cannot be modeled by purely linear techniques. One way to preserve geometric details under global deformations, while still using a linear deformation approach, is to use *multiresolution* techniques, as described in this section.

Multiresolution (or multi-scale) techniques perform a frequency decomposition of the object in order to provide global deformations with intuitive local detail preservation. Chapter **??** describes how signal processing techniques, such as low-pass filtering, can be generalized to (signals on) surfaces. In this setting the fine surface details correspond to the high frequencies of the surface signal and the global shape is represented by its low frequency components. However, in contrast to surface smoothing, one now wants to explicitly modify the low frequencies and preserve the high frequency details, resulting in the desired multiresolution deformation. Fig. **??** shows a simple 2D example of this concept.

The complete multiresolution editing process is depicted in Fig. **??**. In a first step a low-frequency representation of the given surface $\mathcal{S}$ is computed by removing the high frequencies,

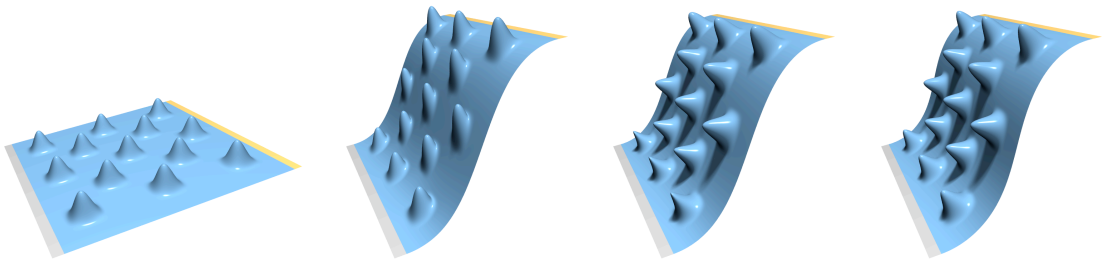


Figure 11.5: The right strip $\mathcal{H}$ of the bumpy plane (*left*) is lifted. The intuitive local rotations of geometric details cannot be achieved by a linearized deformation alone (*center left*). A multiresolution approach based on normal displacements (*center right*) correctly rotates local details, but also distortions them, which can be seen in the left-most row of bumps. The more accurate result of a nonlinear technique is shown on the *right*.

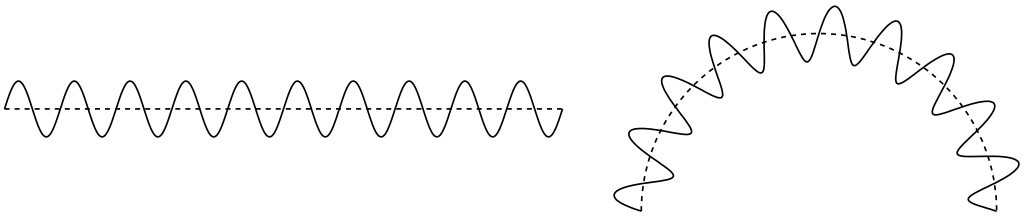


Figure 11.6: A multiresolution deformation of a sine wave. A frequency decomposition yields the dashed line as its low frequency component (*left*). Bending this line and adding the higher frequencies back onto it results in the desired global shape deformation (*right*).

yielding a smooth base surface $\mathcal{B}$. The geometric details $\mathcal{D} = \mathcal{S} \ominus \mathcal{B}$, i.e., the fine surface features that have been removed, represent the high frequencies of $\mathcal{S}$ and are stored as detail information. This allows reconstructing the original surface $\mathcal{S}$ by adding the geometric details back onto the base surface: $\mathcal{S} = \mathcal{B} \oplus \mathcal{D}$. The special operators $\ominus$ and $\oplus$ are called the *decomposition* and the *reconstruction* operator of the multiresolution framework, respectively. This multiresolution surface representation is now enhanced by an *editing* operator, that is used to deform the smooth base surface $\mathcal{B}$ into a modified version $\mathcal{B}'$. Adding the geometric details onto the deformed base surface then results in a multiresolution deformation $\mathcal{S}' = \mathcal{B}' \oplus \mathcal{D}$.

Notice that in general more than one decomposition step is used to generate a hierarchy of meshes $\mathcal{S} = \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_k = \mathcal{B}$ with decreasing geometric complexity. In this case the frequencies that are lost from one level $\mathcal{S}_i$ to the next smoother one $\mathcal{S}_{i+1}$ are stored as geometric details $\mathcal{D}_{i+1} = \mathcal{S}_i \ominus \mathcal{S}_{i+1}$, such that after deforming the base surface to $\mathcal{B}'$, the modified original surface can be reconstructed by $\mathcal{S}' = \mathcal{B}' \bigoplus_{i=0}^{k-1} D_{k-i}$. Since the generalization to several hierarchy levels is straightforward, we restrict our explanations to the simpler case of a two-band decomposition, as shown in Fig. **??**.

A complete multiresolution deformation framework has to provide the three basic operators shown in Fig. **??**: the decomposition operator (*detail analysis*), the editing operator (*shape deformation*), and the reconstruction operator (*detail synthesis*). The decomposition is typically performed by mesh smoothing or fairing (Chapter **??**), and surface deformation has been discussed in the previous sections. The missing component is a suitable representation for the geometric detail $\mathcal{D} = \mathcal{S} \ominus \mathcal{B}$, which we describe in the following.

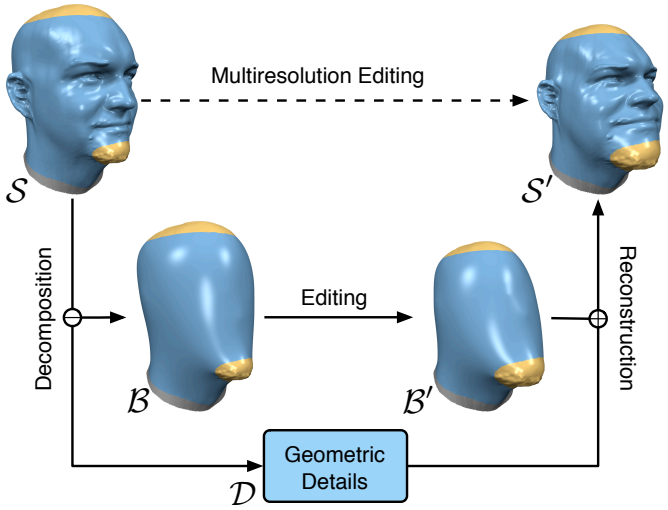


Figure 11.7: A general multiresolution editing framework consists of three main operators: the *decomposition* operator, that separates the low and high frequencies, the *editing* operator, that deforms the low frequency components, and the *reconstruction* operator, that adds the details back onto the modified base surface. Since the lower part of this scheme is hidden in the multiresolution kernel, only the multiresolution edit in the top row is visible to the designer.

Displacement Vectors The straightforward representation for multiresolution details is a displacement of the base surface $\mathcal{B}$, i.e., the detail information is a vector valued displacement function $\mathbf{h} : \mathcal{B} \rightarrow \mathbb{R}^3$ that associates a displacement vector $\mathbf{h}(\mathbf{b})$ with each point $\mathbf{b}$ on the base surface. In a typical setting $\mathcal{S}$ and $\mathcal{B}$ will have the same connectivity, leading to per-vertex *displacement vectors* $\mathbf{h}_i$ [**?**, **?**, **?**]:

$$\mathbf{p}_i \;=\; \mathbf{b}_i + \mathbf{h}_i \;, \quad \mathbf{h}_i \in \mathbb{R}^3,$$

where $\mathbf{b}_i \in \mathcal{B}$ is the vertex corresponding to $\mathbf{p}_i \in \mathcal{S}$. The vectors $\mathbf{h}_i$ have to be encoded in *local frames* w.r.t. $\mathcal{B}$ [**?**, **?**], determined by the normal vector $\mathbf{n}_i$ and two vectors spanning the tangent plane (cf. Fig. **??**). When the base surface $\mathcal{B}$ is deformed to $\mathcal{B}'$, the displacement vectors rotate

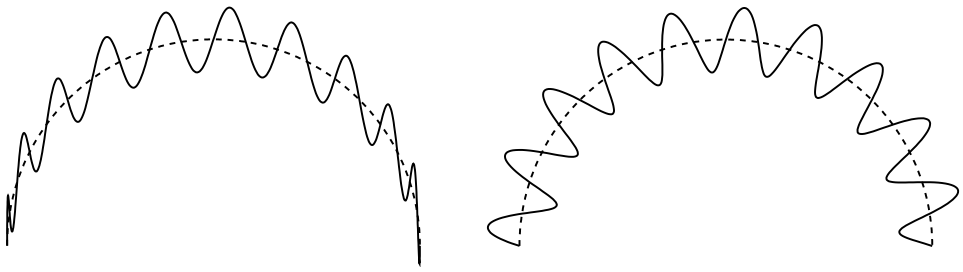


Figure 11.8: Representing the displacements w.r.t. the global coordinate system does not lead to the desired result (*left*). The geometrically intuitive solution is achieved by storing the detail w.r.t. local frames that rotate according to the local tangent plane's rotation of $\mathcal{B}$ (*right*).

according to the rotations of the base surface's local frames, which then leads to a plausible detail reconstruction for $\mathcal{S}'$.

Normal Displacements As we will see below, long displacement vectors might lead to instabilities, in particular for bending deformations. As a consequence, for numerical robustness the displacement vectors should be as short as possible, which is the case if they connect vertices $\mathbf{p}_i \in \mathcal{S}$ to their closest surface points on $\mathcal{B}$ instead of to their corresponding vertices of $\mathcal{B}$. This idea leads to *normal displacements* that are perpendicular to $\mathcal{B}$, i.e., parallel to its normal field $\mathbf{n}$:

$$\mathbf{p}_i \;=\; \mathbf{b}_i + h_i \cdot \mathbf{n}_i \;, \quad h_i \in \mathbb{R}. \tag{11.5}$$

Since the displacements are in general not parallel to the surface normal, generating normal displacements has to involve some kind of resampling. Shooting rays in normal direction from each base vertex $\mathbf{b}_i \in \mathcal{B}$ and deriving new vertex positions $\mathbf{p}_i \in \mathcal{S}$ at their intersections with the detailed surface leads to a resampling of the latter [**?**, **?**]. Because $\mathcal{S}$ might be a detailed surface with high frequency features, such a resampling is likely to introduce alias artifacts. Hence, Kobbelt et al. [**?**] go the other direction: for each vertex position $\mathbf{p}_i \in \mathcal{S}$ they find a base point $\mathbf{b}_i \in \mathcal{B}$ (now *not* necessarily a vertex of $\mathcal{B}$), such that the displacements are normal to $\mathcal{B}$, i.e., $\mathbf{p}_i = \mathbf{b}_i + h_i \cdot \mathbf{n}(\mathbf{b}_i)$. This avoids a resampling of $\mathcal{S}$ and therefore allows for the preservation of all of its sharp features (see also [**?**] for a comparison and discussion). Since the base points $\mathbf{b}_i$ are arbitrary surface points of $\mathcal{B}$, the connectivity of $\mathcal{S}$ and $\mathcal{B}$ is no longer restricted to be identical. This can be exploited in order to remesh the base surface $\mathcal{B}$ for the sake of higher numerical robustness [**?**].

Displacement Volumes While normal displacement are extremely efficient, their main problem is that neighboring displacement vectors are not coupled in any way. When bending the surface in a convex or concave manner, the angle between neighboring displacement vectors increases or decreases, leading to an undesired distortion of geometric details (cf. Figs. **??** and **??**). In the extreme case of neighboring displacement vectors crossing each other (which happens if the curvature of $\mathcal{B}'$ becomes larger than the displacement length h_i), the surface even self-intersects locally.

Both problems, the unnatural change of volume and local self-intersections, are addressed by displacement *volumes* instead of displacement *vectors* [**?**]. Each triangle $\left(\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k\right)$ of $\mathcal{S}$, together with the corresponding points $(\mathbf{b}_i, \mathbf{b}_j, \mathbf{b}_k)$ on $\mathcal{B}$, defines a triangular a prism. The volumes of those prisms are used as detail coefficients $\mathcal{D}$, and are kept constant during deformations. For a modified base surface $\mathcal{B}'$ the reconstruction operator therefore has to find $\mathcal{S}'$ such that the enclosed prisms have the same volumes as for the original shape. The local volume preservation leads to more intuitive results and avoids local self-intersections (cf. Figs. **??**, **??**). However, the improved detail preservation comes at the higher computational cost of a nonlinear detail reconstruction process.

Deformation Transfer Botsch et al. [**?**] use the deformation transfer approach of [**?**] to transfer the base surface deformation $\mathcal{B} \mapsto \mathcal{B}'$ onto the detailed surface $\mathcal{S}$, resulting in a multiresolution deformation $\mathcal{S}'$. This method yields results similar in quality to displacement volumes (cf. Figs. **??**, **??**), but only requires solving a sparse linear Poisson system. Both in terms of results and of computational efficiency this method can be considered as lying in between displacement vectors and displacement volumes.

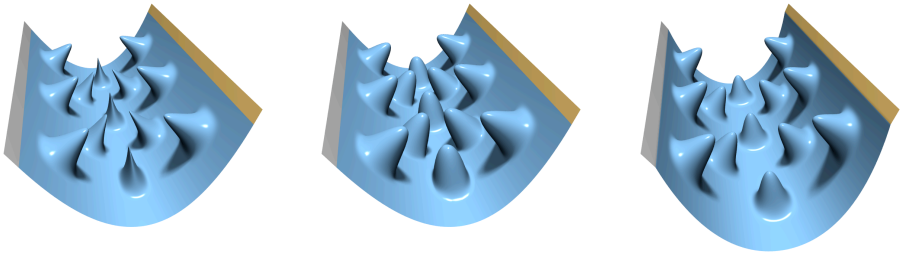


Figure 11.9: For a bending of the bumpy plane, normal displacements distort geometric details and almost lead to self-intersections (*left*), whereas displacement volumes (*center*) and deformation transfer (*right*) achieve more natural results.

11.1.5 Differential Coordinates

While multiresolution or multi-scale hierarchies are an effective tool for enhancing freeform deformations by fine-scale detail preservation, the generation of the hierarchy can become quite involved for geometrically or topologically complex models. To avoid the explicit multi-scale decomposition, another class of methods modifies differential surface properties instead of spatial coordinates, and then reconstructs a deformed surface having the desired differential coordinates.

We will first describe two typical differential representations, gradients and Laplacians, and how to derive the deformed surface from the manipulated differential coordinates. We then explain how to compute the local transformations of differential coordinates based on the user's deformation constraints. More details on these topics, such as methods based on local frames [**?**, **?**, **?**], sketching interfaces [**?**], or volumetric Laplacians [**?**], can be found in the recent survey [**?**].

Gradient-Based Deformation

The methods of [**?**, **?**] deform the surface by prescribing a target gradient field and finding a surface that matches this gradient field in the least squares sense. In the continuous setting, consider a function $f : \Omega \to \mathbb{R}$ that should match a user-prescribed gradient field g by minimizing

$$\int_\Omega \|\nabla f - g\|^2 \, \mathrm{d}u\mathrm{d}v \; .$$

Applying variational calculus yields the Euler-Lagrange equation

$$\Delta f = \mathrm{div} g \; , \tag{11.6}$$

which has to be solved for the optimal f. On a discrete triangle mesh, a piecewise linear function $f : \mathcal{S} \to \mathbb{R}$ is defined by its values $f_i := f(\mathbf{p}_i)$ at the mesh vertices. Its gradient $\nabla f : \mathcal{S} \to \mathbb{R}^3$ is a constant vector $\mathbf{g}_j \in \mathbb{R}^3$ within each triangle f_j. If instead of a scalar function f the piecewise linear coordinate function $\mathbf{p}(v_i) = \mathbf{p}_i \in \mathbb{R}^3$ is considered, then the gradient within a face f_j is a constant 3×3 matrix

$$\nabla \mathbf{p}|_{f_j} \; =: \; \mathbf{G}_j \in \mathbb{R}^{3\times 3} \; .$$

For a mesh with V vertices and F triangles, the discrete gradient operator can be expressed by a $3F \times V$ matrix $\mathbf{G}$:

$$\begin{pmatrix} \mathbf{G}_1 \\ \vdots \\ \mathbf{G}_F \end{pmatrix} \; = \; \mathbf{G} \cdot \begin{pmatrix} \mathbf{p}_1^T \\ \vdots \\ \mathbf{p}_V^T \end{pmatrix} \; .$$

The face gradients are then modified explicitly (as discussed later), yielding new gradients $\mathbf{G}'_j$ per triangle f_j. Reconstructing a mesh having these desired gradients is an overdetermined problem, and therefore is solved in a weighted least squares sense using the normal equations [**?**]:

$$\underbrace{\mathbf{G}^T\mathbf{D}\mathbf{G}}_{\Delta_\mathcal{S}} \cdot \begin{pmatrix} {\mathbf{p}'_1}^T \\ \vdots \\ {\mathbf{p}'_V}^T \end{pmatrix} = \underbrace{\mathbf{G}^T\mathbf{D}}_{\text{div}} \cdot \begin{pmatrix} \mathbf{G}'_1 \\ \vdots \\ \mathbf{G}'_F \end{pmatrix} , \tag{11.7}$$

where $\mathbf{D}$ is a diagonal matrix containing the face areas as weighting factors. Since the matrix $\mathbf{G}^T\mathbf{D}$ corresponds to the discrete divergence operator, and since $\text{div}\,\nabla = \Delta$, this system actually is a Poisson equation. It corresponds to the discretization of the Euler-Lagrange PDE (**??**). Hence, these methods prescribe a guidance gradient field $(\mathbf{G}'_1, \ldots, \mathbf{G}'_F)$, compute its divergence, and solve three sparse linear Poisson systems for the x, y, and z coordinates of the modified mesh vertices $\mathbf{p}'_i$.

Laplacian-Based Deformation

Other methods manipulate Laplacians of the vertices instead of gradient fields [**?**, **?**, **?**, **?**]. They compute initial Laplacian coordinates $\boldsymbol{\delta}_i = \Delta_\mathcal{S}(\mathbf{p}_i)$ and manipulate them to $\boldsymbol{\delta}'_i$ as discussed below. The goal is to find a new coordinate function $\mathbf{p}'$ that matches the target Laplace coordinates. In the continuous setting one has to minimize

$$\int_\Omega \left\| \Delta_\mathcal{S}\mathbf{p}' - \boldsymbol{\delta}' \right\|^2 \mathrm{d}u\mathrm{d}v \,,$$

which leads to the Euler-Lagrange equations

$$\Delta_\mathcal{S}^2\mathbf{p}' \;=\; \Delta_\mathcal{S}\boldsymbol{\delta}' \,.$$

On a discrete mesh, this yields a bi-Laplacian system to be solved for the deformed surface $\mathcal{S}'$:

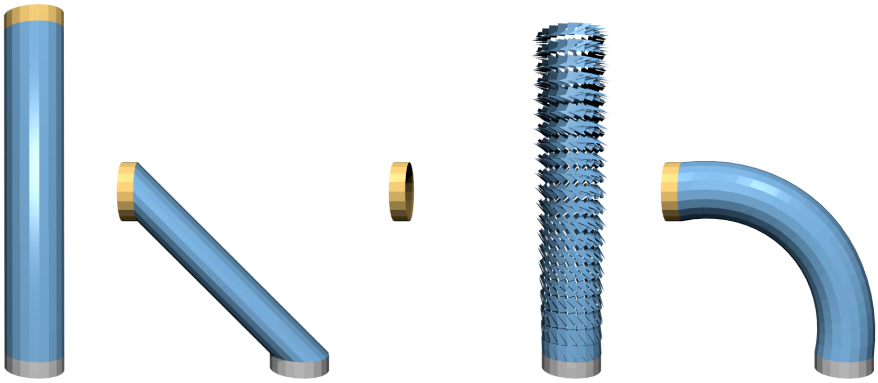
$$\Delta_\mathcal{S}^2 \cdot \begin{pmatrix} {\mathbf{p}'_1}^T \\ \vdots \\ {\mathbf{p}'_V}^T \end{pmatrix} \;=\; \Delta_\mathcal{S} \cdot \begin{pmatrix} {\boldsymbol{\delta}'_1}^T \\ \vdots \\ {\boldsymbol{\delta}'_V}^T \end{pmatrix} \,.$$

Although the original approaches use the uniform Laplacian discretization [**?**, **?**], the cotangent weights can be shown to yield better results for irregular triangle meshes (see Fig. **??** and [**?**]).

When we do not consider the local transformation $\boldsymbol{\delta}_i \mapsto \boldsymbol{\delta}'_i$, but instead reconstruct the surface from the original Laplacians $\boldsymbol{\delta}_i$, then the Euler-Lagrange equation $\Delta_\mathcal{S}^2\mathbf{p}' = \Delta_\mathcal{S}\boldsymbol{\delta}$ reveals the connection to the variational bending minimization (Section **??**), whose Euler-Lagrange PDE is $\Delta_\mathcal{S}^2\mathbf{p}' = 0$. Using the identities $\mathbf{p}' = \mathbf{p} + \mathbf{d}$ and $\boldsymbol{\delta} = \Delta_\mathcal{S}\mathbf{p}$ one immediately sees that the two approaches are equivalent. The methods differ in the way they model the local rotations of geometric details or differential coordinates, either by multiresolution methods (Section **??**) or by local transformations, as discussed in the following.

Local Transformations

The missing component is a technique for modifying the gradients $\mathbf{G}_j$ or Laplacians $\boldsymbol{\delta}_i$ based on the affine handle transformation provided by the user. The methods discussed below derive local transformations $\mathbf{T}_i$ in order to transform gradients $(\mathbf{G}'_j = \mathbf{G}_i \cdot \mathbf{T}_j)$ or Laplacians $(\boldsymbol{\delta}'_i = \mathbf{T}_i \cdot \boldsymbol{\delta}_i)$.



(a) (b) (c) (d)

Figure 11.10: Using gradient-based editing to bend the cylinder (a) by 90°. Reconstructing the mesh from new handle positions, but *original* gradients distorts the object (b). Applying damped local rotations derived from (**??**) to the individual triangles breaks up the mesh (c), but solving the Poisson system (**??**) re-connects it and yields the desired result (d).

The gradient-based approaches [**?**, **?**] use the gradient of this affine deformation, i.e., its rotation and scale/shear components, for transforming the surface gradients. They first construct a smooth scalar blending field $s : \mathcal{S} \rightarrow [0, 1]$ based on either geodesic distances (Section **??**) or harmonic fields. The gradient $\mathbf{T} = \mathbf{RS}$ of the affine handle transformation $\mathbf{x} \mapsto \mathbf{Tx} + \mathbf{t}$ is decomposed into rotation $\mathbf{R}$ and scale/shear $\mathbf{S}$ using polar decomposition [**?**]. Both components are then interpolated over the support region:

$$\mathbf{T}_i \;=\; \text{slerp}\left(\mathbf{R}, \mathbf{I}, 1 - s_i\right) \cdot \left((1 - s_i)\mathbf{S} + s_i\mathbf{I}\right) , \tag{11.8}$$

where $\text{slerp}\left(\cdot\right)$ denotes quaternion interpolation, $s_i = s(\mathbf{p}_i)$ is the vertex' blending value, and $\mathbf{I}$ denotes the identity matrix. This method works well for rotations, since those are handled explicitly, but it is insensitive to handle translations: Adding a translation $\mathbf{t}$ to a given deformation does not change its gradient, and thus has no influence on the resulting surface gradients. But as there is a (nonlinear) connection between translations and local rotations of gradients, these methods yield counter-intuitive results for modifications containing large translations (Section **??**).

To address this issue, Sorkine et al. [**?**] implicitly optimize for the local rotations $\mathbf{T}_i$ of vertex neighborhoods by minimizing the following energy functional

$$E\left(\mathbf{p}'_1, \ldots, \mathbf{p}'_V\right) \;=\; \sum_{i=1}^{V} \left\|\mathbf{T}_i\boldsymbol{\delta}_i - \Delta\left(\mathbf{p}'_i\right)\right\|^2 \;+\; \sum_{i \in \mathcal{C}} \left\|\mathbf{p}'_i - \mathbf{u}_i\right\|^2 \quad ,$$

where $\mathbf{u}_i$ are the target positions for the constrained vertices $\mathbf{p}_i$, $i \in \mathcal{C}$. For the sake of computational efficiency they had to linearize the local frame transformations $\mathbf{T}_i$, which on the one hand allows to formulate the optimization as a single linear system, but one the other hand leads to artifacts in case of large rotations.

Lipman et al. [**?**, **?**] minimize surface bending by preserving the relative orientations of per-vertex local frames. This is done by first solving a linear least squares system for the modified per-vertex local frame rotations $\mathbf{T}_i$, and reconstructing the modified vertex positions $\mathbf{p}'_i$ in a second step. However, since the first system does not consider the positional constraints, one has to ensure that the positional constraints and the orientation constraints are compatible. While

their method works very well even for large rotations, it exhibits the same translation-insensitivity as the gradient-based methods.

11.1.6 Limitations on Linear Methods

In this section we compare the linear surface deformation techniques discussed so far, and point out their limitations. The goal is therefore *not* to show the best-possible results each method can produce, but rather to show under which circumstances each individual method fails. Hence, in Fig. **??**we picked extreme deformations that identify the respective limitations of the different techniques. For comparison we show the results of the non-linear surface deformation PriMo [**?**], which does not suffer from linearization artifacts. For more detailed comparisons see [**?**].

The variational bending energy minimization [**?**], in combination with the multiresolution technique [**?**] works fine for pure translations, and yields fair and detail preserving deformations. However, due to the linearization of the shell energy this approach fails for large rotations. The gradient-based editing [**?**, **?**] updates the surface gradients using the gradient of the deformation (its rotation and scale/shear components), and therefore works very well for rotations. However, as mentioned in the last section, the explicit propagation of local rotations is translation-insensitive, such that the plane example is neither smooth nor detail preserving. The Laplacian surface editing [**?**] implicitly optimizes for local rotations, and hence works similarly well for translations and rotations. However, the required linearization of rotations yields artifacts for large rotations.

As the physical equations governing the surface deformation process are inherently nonlinear, all linearized techniques fail under certain circumstances. While the variational energy minimization typically works for translations, but have problems with large rotations, it is the other way around for differential approaches. Another comparison on a large-scale transformation is shown in Fig. **??**. To overcome the limitations for large-scale deformations, those either have to be split up into sequences of smaller deformations — thereby complicating the user interaction — or nonlinear approaches have to be considered, as discussed in the next section.

11.1.7 Nonlinear Surface Deformation

Thanks to the rapid increase in both computational power and available memory of today's workstations, nonlinear deformation methods become more and more tractable, which in the last years already lead to a first set of nonlinear, yet interactive, surface deformation approaches. Due to space limitations we will only briefly mention some nonlinear approaches and refer the reader to the original papers. While a nonlinear implementation of the previously discussed approaches seems to be straightforward (simply do not use any linearization), in the nonlinear case special attention has to be paid to computational efficiency and numerical robustness.

PriMo [**?**] is a nonlinear version of the variational minimization of bending and stretching energies. The surface is modeled as a thin layer of triangular prisms, which are coupled by a nonlinear elastic energy. During deformation the prisms are kept rigid, which allows for an extremely robust geometric optimization.

The pyramid coordinates Sheffer and Kraevoy [**?**, **?**] can be considered as a nonlinear version of Laplacian coordinates, leading to differential coordinates invariant under rigid motions, which can be used for deformation as well as for morphing.

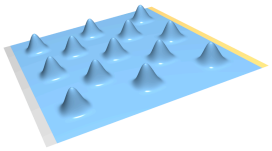
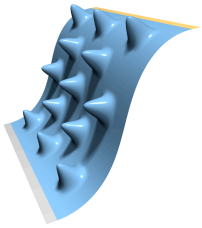
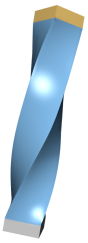
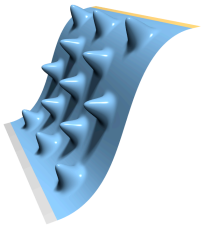
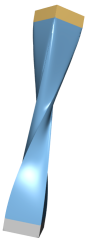
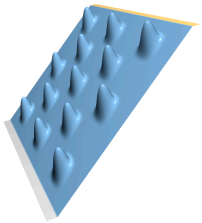
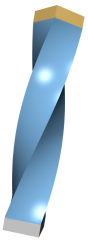
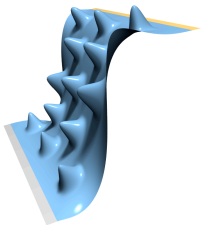
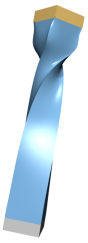
Approach	Pure Translation	120° bend	135° twist
Original model			
Nonlinear prism-based modeling [?]			
Variational minimization [?] + deformation transfer [?]			
Gradient-based editing [?]			
Laplacian-based editing with implicit optimization [?]			

Figure 11.11: The extreme examples shown in this comparison matrix were particularly chosen to reveal the limitations of the respective deformation approaches.

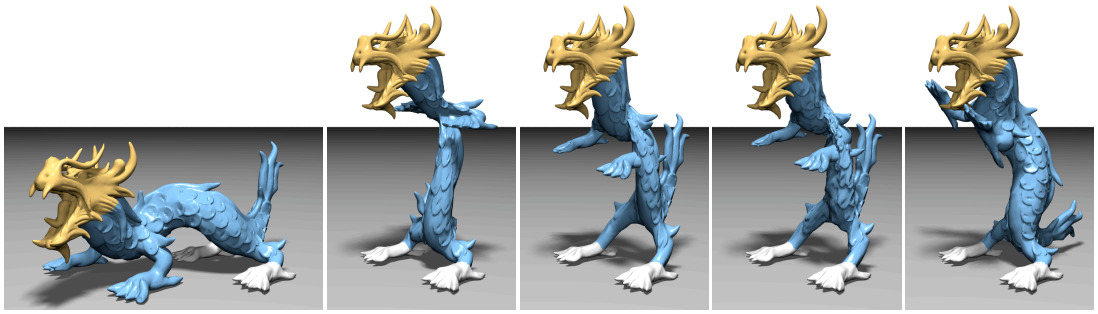


Figure 11.12: The crouching dragon was lifted by fixing its hind feet and moving its head to the target position in a single, large-scale deformation. Similar to Fig. **??**, the linear deformation methods yield counter-intuitive results. The nonlinear PriMo technique yields a more natural deformation.

Huang et al. [**?**] employ a nonlinear version of the volumetric graph Laplacian, which also features nonlinear volume preservation constraints. In order to increase performance and efficiency of their optimization they use a subspace approach: The original mesh is embedded in a coarse control mesh, and the optimization is performed on the control mesh while considering the constraints from the original mesh in a least squares manner. An extension of nonlinear gradient-based deformation to mesh sequences was presented by [**?**]. In order to solve the involved nonlinear systems more efficiently they alternatively solved least squares systems for vertex positions $\mathbf{p}'_i$ and local rotations $\mathbf{T}_i$.

An alternative approach to subspace methods is the handle-aware isoline technique of [**?**]. In a preprocessing step they construct a set of isolines of the geodesic distance from either the fixed regions or the handle regions, similar in spirit to [**?**]. For each of these isolines they find a local transformation $\mathbf{T}_i$ for a Laplacian-based deformation, based on a nonlinear optimization. The number of required isolines is relatively small, which guarantees an efficient numerical optimization and thereby allows for interactive editing. Shi et a. [**?**] combine Laplacian-based deformation with skeleton-based inverse kinematics. Their approach allows for easy and intuitive character posing, featuring control of lengths, rigidity, and joint limits, but it in turn requires a complex cascading optimization for the involved nonlinear energy minimization.

11.2 Space Deformation

All the surface-based approaches described in Section **??** compute a smooth deformation field *on* the surface $\mathcal{S}$. For linear methods this typically amounts to solving a (bi-)Laplacian system as the Euler-Lagrange PDE of some quadratic energy, whereas nonlinear approaches minimize higher order energies using Newton- or Gauss-Newton-like techniques. An apparent drawback of such methods is that their computational effort and numerical robustness are strongly related to the complexity and quality of the surface tessellation.

In the presence of degenerate triangles the discrete Laplacian operator is not well-defined and thus the involved linear systems become singular. Similarly, topological artifacts like gaps or non-manifold configurations lead to problems as well. In such cases quite some effort has to be spent to still be able to compute smooth deformations for the numerically problematic

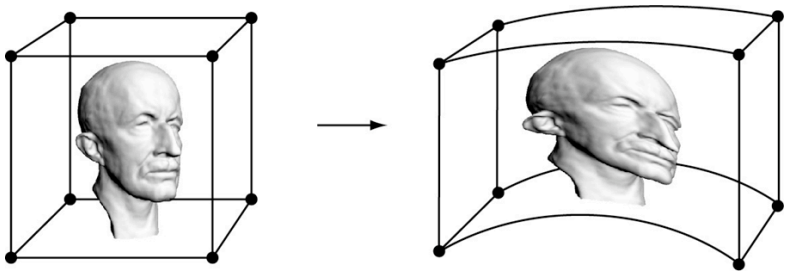


Figure 11.13: Freeform space deformations warp the space around an object, and by this deform the embedded object itself.

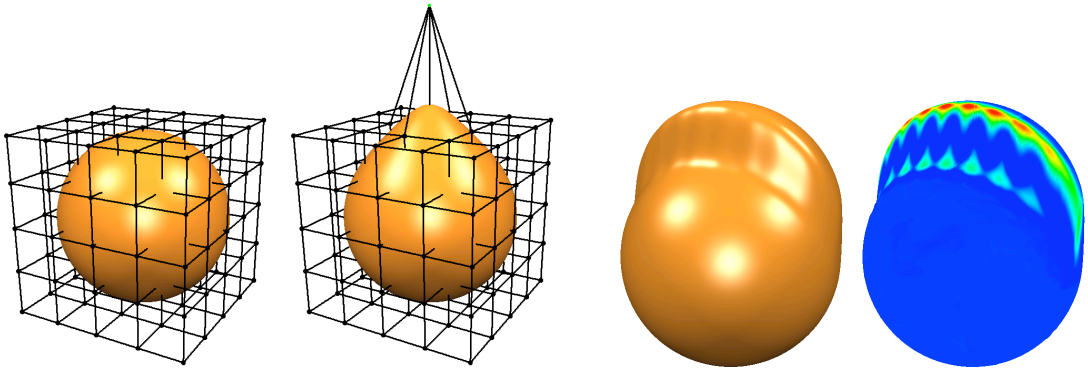


Figure 11.14: In the freeform deformation approach a regular 3D control lattice is used to specify a volumetric displacement function (*left*). Similar to tensor-product spline surfaces, the tri-variate tensor-product splines can also lead to alias artifacts in the deformed surface (*right*).

meshes, like eliminating degenerate triangles (Chapter **??**) or even remeshing the complete surface (Chapter **??**). Even when the mesh quality is sufficiently high, extremely complex meshes will result in linear or nonlinear systems which cannot be solved simply due to their size.

These problems are avoided by volumetric *space deformation* techniques, that deform the ambient 3D space and by this implicitly deform the embedded objects (cf. Fig. **??**). In contrast to surface-based methods, space deformation approaches employ a trivariate deformation function $\mathbf{d} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ to transform all points of the original surface $\mathcal{S}$ to the modified surface $\mathcal{S}' = \{\mathbf{p} + \mathbf{d}(\mathbf{p}) \mid \mathbf{p} \in \mathcal{S}\}$. Since the space deformation function $\mathbf{d}$ does not depend on a particular surface representation, it can be used to deform all kinds of explicit surface representations, e.g., by transforming all vertices of a triangle mesh or all points of a point-sampled model. Analogously to surface-based techniques, we will see that approaches based on a global energy minimization typically lead to highest quality results.

11.2.1 Freeform Deformation

The classical freeform deformation (FFD) method [**?**] represents the space deformation by a tensor-product Bezier or spline function

$$\mathbf{d}(u,v,w) \;=\; \sum_i \sum_j \sum_k \delta\mathbf{c}_{ijk}\, N_i^l(u)\, N_j^n(v)\, N_k^m(w) \quad .$$

Because of the same reasons as for spline surfaces (Section **??**), these approaches require complex user-interactions and can cause aliasing problems, as shown in Fig. **??**. In order to satisfy given displacement constraints, the inverse FFD method [**?**] solves a linear system for the required movements of control points $\mathbf{c}_{ijk}$, which again does not necessarily imply a fair deformation of low curvature energy.

11.2.2 Transformation Propagation

Handle transformations can be propagated analogously to the surface-based techniques described in Section **??** by constructing the scalar field $s(\cdot)$ based on Euclidean distances, instead of geodesic distances [**?**]. While this typically leads to inferior results compared to geodesic-based propagation, this method even works if a surface-based propagation fails due to topological problems like gaps or holes.

Besides from that, the limitations of the surface-based propagation also apply to this method. A smooth interpolation of arbitrary constraints might not be possible, and the resulting surface fairness is typically inferior to techniques based on energy minimization.

11.2.3 Radial Basis Functions

In the case of surface-based deformations, the highest quality results are achieved by interpolating user constraints by a displacement function $\mathbf{d} : \mathcal{S} \to \mathbb{R}^3$ that additionally minimizes fairness energies (Section **??**). Motivated by this, we therefore are looking for smoothly interpolating tri-variate *space deformation* functions $\mathbf{d} : \mathbb{R}^3 \to \mathbb{R}^3$ that minimizes analogous fairness energies.

Radial basis functions (RBFs) are known to be well suited for scattered data interpolation problems [**?**]. A trivariate RBF deformation is defined in terms of centers $\mathbf{c}_j \in \mathbb{R}^3$ and weights $\mathbf{w}_j \in \mathbb{R}^3$ as

$$\mathbf{d}(\mathbf{x}) \;=\; \sum_j \mathbf{w}_j \cdot \varphi(\|\mathbf{c}_j - \mathbf{x}\|) + \mathbf{p}(\mathbf{x}) \quad , \tag{11.9}$$

where $\varphi(\|\mathbf{c}_j - \cdot\|)$ is the basis function corresponding to the jth center $\mathbf{c}_j$ and $\mathbf{p}(\mathbf{x})$ is a polynomial of low degree used to guarantee polynomial precision. In order to construct an RBF interpolating the constraints $\mathbf{d}(\mathbf{p}_i) = \mathbf{d}_i$, the centers are typically placed on the constraints ($\mathbf{c}_i = \mathbf{p}_i$) and a linear system is solved for the RBF's weights $\mathbf{w}_i$ and the coefficients of the polynomial $\mathbf{p}(\mathbf{x})$ (see for instance [**?**]).

The choice of φ has a strong influence on the computational complexity and the resulting surface's fairness: While compactly supported radial basis functions lead to sparse linear systems and hence can be used to interpolate several hundred thousands of data points [**?**, **?**], they do not provide the same degree of fairness as basis functions of global support [**?**]. It was shown

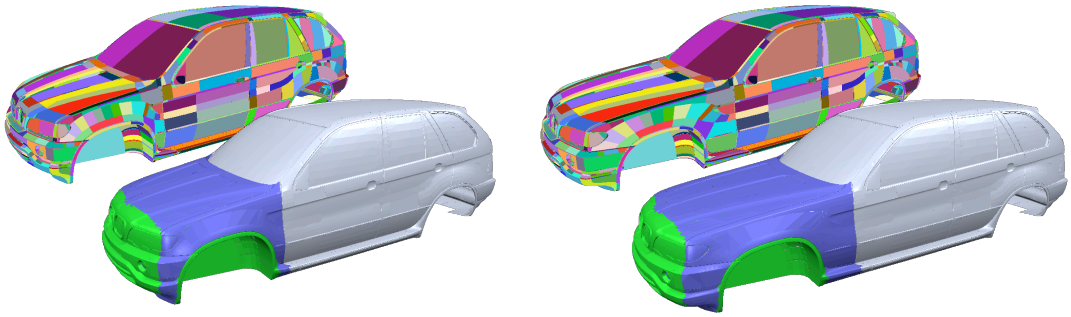


Figure 11.15: Using multiple independent handle components allows to stretch the hood while rigidly preserving the shape of the wheel houses. This 3M triangle model consists of 10k individual connected components, which are neither two-manifold nor consistently oriented.

by Duchon [**?**] that for the basis function $\varphi(r) = r^3$ and quadratic polynomials $\mathbf{p}(\cdot) \in \Pi_2$, the function (**??**) is triharmonic ($\Delta^3 \mathbf{d} = 0$) and minimizes the energy

$$\int_{\mathbb{R}^3} \|\mathbf{d}_{xxx}(\mathbf{x})\|^2 + \|\mathbf{d}_{xxy}(\mathbf{x})\|^2 + \ldots + \|\mathbf{d}_{zzz}(\mathbf{x})\|^2 \, \mathrm{d}\mathbf{x} \, .$$

Notice that these trivariate functions are *conceptually* equivalent to the minimum variation surfaces of [**?**] and the triharmonic surfaces used in [**?**], and hence provide the same degree of fairness. The difference is that for triharmonic RBFs the energy minimization is "built-in", whereas for surface-based approaches we explicitly optimized for it (Section **??**). The major drawback is that the fairness property comes at the price of having to solve a dense linear system, due to the global support of the triharmonic basis function $\varphi(r) = r^3$.

However, Botsch and Kobbelt [**?**] propose an incremental least squares method that efficiently solves the linear system up to a prescribed error bound. Using this solver to pre-compute deformation basis functions allows interactively deforming even complex models. Moreover, evaluating these basis functions on the graphics card further accelerates this approach and provides real-time space deformations at a rate of 30M vertices/sec. As shown in Fig. **??**, even complex surfaces consisting of disconnected patches can be handled by this technique, whereas all surface-based techniques would fail in this situation.

However, for the discussed space deformation approaches the deformed surface $\mathcal{S}'$ linearly depends on the displacement constraints $\mathbf{d}_i$. As a consequence, nonlinear effects such as local detail rotation cannot be achieved, similar to the linear surface-based methods. Although space deformations can be enhanced by multiresolution techniques as well (see, e.g., [**?**]), they suffer from the same limitations as discussed in Section **??**, which lead to the development of nonlinear space deformation approaches.

11.2.4 Nonlinear Space Deformation

In this section, similar to Section **??**, we only mention some very recent nonlinear space deformation methods without providing details, and refer the reader to the cited papers for more details.

Sumner et al. [**?**] compute detail-preserving space deformations by formulating an energy functional that explicitly penalizes deviation from local rigidity, by optimizing the local deformation

gradients to be rotations. In addition to static geometries, their method can also be applied to hand-crafted animations and precomputed simulations.

Botsch et al. [**?**] extend the PriMo framework [**?**] to deformations of solid objects. The input model is voxelized in an adaptive manner, and the resulting hexahedral cells are kept rigid under deformations to ensure numerical robustness. The deformation is governed by a nonlinear elastic energy coupling neighboring rigid cells.

Another class of approaches uses divergence-free vector fields to deform shapes [**?**, **?**]. The advantage of those techniques is that they by construction yield volume preserving and intersection-free deformations. As a drawback, it is harder to construct vector fields that exactly satisfy user-defined deformation constraints.

12 Numerics

In this section we describe different types of solvers for sparse linear systems. Within this class of systems, we will further concentrate on symmetric positive definite (so-called *spd*) matrices, since exploiting their special structure allows for the most efficient and most robust implementations. Examples of such matrices are Laplacian system (to be analyzed in Section **??**) and general least squares systems. However, the general case of a non-symmetric indefinite system is outlined afterwards in Section **??**.

Following [**?**], we propose the use of direct solvers for sparse spd systems, since their superior efficiency — although well known in the field of high performance computing — is often neglected in geometry processing applications. After reviewing the commonly known and used direct and iterative solvers, we introduce sparse direct solvers and point out their advantages.

For the following discussion we restrict ourselves to sparse spd problems $\mathbf{A}\mathbf{x} = \mathbf{b}$, with $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n\times n}$, $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$, and denote by $\mathbf{x}^*$ the exact solution $\mathbf{A}^{-1}\mathbf{b}$. The general case of non-symmetric indefinite systems is then outlined in Section **??**.

12.1 Laplacian Systems

Since Laplacian systems play a major role in several geometry processing applications, like smoothing (Chapter **??**), conformal parametrization (Chapter **??**), and shape deformation (Chapter **??**), we will shortly describe general Laplacian matrices first.

In each row the matrix $\Delta_{\mathcal{S}}$ contains the weights for the discretization of the Laplace-Beltrami of a function $f : \mathcal{S} \to \mathbb{R}$ at one vertex v_i (see Chapter **??**):

$$\Delta_{\mathcal{S}} f(v_i) \;=\; \frac{2}{A(v_i)} \sum_{v_j \in \mathcal{N}_1(v_i)} \left(\cot\alpha_{ij} + \cot\beta_{ij}\right)\left(f(v_j) - f(v_i)\right) \quad .$$

This can be written in matrix notation as

$$\begin{pmatrix} \vdots \\ \Delta_{\mathcal{S}} f(v_i) \\ \vdots \end{pmatrix} \;=\; \mathbf{D}\cdot\mathbf{M}\cdot \begin{pmatrix} \vdots \\ f(v_i) \\ \vdots \end{pmatrix} \quad ,$$

where $\mathbf{D}$ is a diagonal matrix of normalization factors $\mathbf{D}_{ii} = 2/A(v_i)$, and $\mathbf{M}$ is a symmetric matrix containing the cotangent weights. Since the Laplacian of a vertex v_i is defined *locally* in terms of its one-ring neighbors, the matrix $\mathbf{M}$ is highly sparse and has non-zeros in the ith row only on the diagonal and in those columns corresponding to v_i's one-ring neighbors $\mathcal{N}_1(v_i)$.

For a closed mesh, Laplacian systems $\Delta_{\mathcal{S}}^k \mathbf{P} = \mathbf{B}$ of any order k can be turned into symmetric ones by moving the first diagonal matrix $\mathbf{D}$ to the right-hand side:

$$\mathbf{M}\left(\mathbf{D}\mathbf{M}\right)^{k-1}\mathbf{P} \;=\; \mathbf{D}^{-1}\mathbf{B} \quad . \tag{12.1}$$

Boundary constraints are typically employed by restricting the values at certain vertices, which corresponds to eliminating their respective rows and columns and hence keeps the matrix symmetric. The case of meshes with boundaries is equivalent to a patch bounded by constrained vertices and therefore also results in a symmetric matrix. Pinkal and Polthier [**?**] additionally showed that this system is positive definite, such that the efficient solvers presented in the next section can be applied.

12.2 Dense Direct Solvers

Direct linear system solvers are based on a factorization of the matrix $\mathbf{A}$ into matrices of simpler structure, e.g., triangular, diagonal, or orthogonal matrices. This structure allows for an efficient solution of the factorized system. As a consequence, once the factorization is computed, it can be used to solve the linear system for several different right hand sides.

The most commonly used examples for *general* matrices $\mathbf{A}$ are, in the order of increasing numerical robustness and computational effort, the LU factorization, QR factorization, or the singular value decomposition. However, in the special case of a spd matrix the Cholesky factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, with $\mathbf{L}$ denoting a lower triangular matrix, should be employed, since it exploits the symmetry of the matrix and can additionally be shown to be numerically very robust due to the positive definiteness of the matrix $\mathbf{A}$ [**?**].

On the downside, the asymptotic time complexity of all dense direct methods is $O(n^3)$ for the factorization and $O(n^2)$ for solving the system based on the pre-computed factorization. Since for the problems we are targeting at, n can be of the order of 10^5, the total cubic complexity of dense direct methods is prohibitive. Even if the matrix $\mathbf{A}$ is highly sparse, the naïve direct methods enumerated here are not designed to exploit this structure, hence the factors are dense matrices in general (cf. Fig. **??**, top row).

12.3 Iterative Solvers

In contrast to dense direct solvers, iterative methods are able to exploit the sparsity of the matrix $\mathbf{A}$. Since they additionally allow for a simple implementation [**?**], iterative solvers are the de-facto standard method for solving sparse linear systems in the context of geometric problems. A detailed overview of iterative methods with valuable implementation hints can be found in [**?**].

Iterative methods compute a converging sequence $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(i)}$ of approximations to the solution $\mathbf{x}^*$ of the linear system, i.e., $\lim_{i\to\infty} \mathbf{x}^{(i)} = \mathbf{x}^*$. In practice, however, one has to find a suitable criterion to stop the iteration if the current solution $\mathbf{x}^{(i)}$ is accurate enough, i.e., if the norm of the error $\mathbf{e}^{(i)} := \mathbf{x}^* - \mathbf{x}^{(i)}$ is less than some ε. Since the solution $\mathbf{x}^*$ is not known beforehand, the error has to be estimated by considering the residual $\mathbf{r}^{(i)} := \mathbf{A}\mathbf{x}^{(i)} - \mathbf{b}$. These two are related by the *residual equations* $\mathbf{A}\mathbf{e}^{(i)} = \mathbf{r}^{(i)}$, leading to an upper bound $\left\|\mathbf{e}^{(i)}\right\| \leq \left\|\mathbf{A}^{-1}\right\| \cdot \left\|\mathbf{r}^{(i)}\right\|$, i.e., the norm of the inverse matrix has to be estimated or approximated in some way (see [**?**]).

In the case of spd matrices the method of conjugate gradients (CG) [**?**, **?**] is suited best, since it provides guaranteed convergence with monotonically decreasing error. For a spd matrix $\mathbf{A}$ the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$ is equivalent to the minimization of the quadratic form

$$\phi(\mathbf{x}) := \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} \ .$$

The CG method successively minimizes this functional along a set of linearly independent $\mathbf{A}$-*conjugate* search directions, such that the exact solution $\mathbf{x}^* \in \mathbb{R}^n$ is found after at most n steps (neglecting rounding errors). The complexity of each CG iteration is mainly determined by the matrix-vector product $\mathbf{Ax}$, which is of order $O(n)$ if the matrix is sparse. Given the maximum number of n iterations, the total complexity is $O(n^2)$ in the worst case, but it is usually better in practice.

As the convergence rate mainly depends on the spectral properties of the matrix $\mathbf{A}$, a proper pre-conditioning scheme should be used to increase the efficiency and robustness of the iterative scheme. This means that a slightly different system $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ is solved instead, with $\tilde{\mathbf{A}} = \mathbf{PAP}^T$, $\tilde{\mathbf{x}} = \mathbf{P}^{-T}\mathbf{x}$, $\tilde{\mathbf{b}} = \mathbf{Pb}$, using a regular pre-conditioning matrix $\mathbf{P}$, that is chosen such that $\tilde{\mathbf{A}}$ is well conditioned [**?**, **?**]. However, the matrix $\mathbf{P}$ is restricted to have a simple structure, since an additional linear system $\mathbf{Pz} = \mathbf{r}$ has to be solved each iteration.

The iterative conjugate gradients method manages to decrease the computational complexity from $O(n^3)$ to $O(n^2)$ for sparse matrices. However, this is still too slow to compute exact (or sufficiently accurate) solutions of large and possibly ill-conditioned systems.

12.4 Multigrid Iterative Solvers

One characteristic problem of most iterative solvers is that they are *smoothers*: they attenuate the high frequencies of the error $\mathbf{e}^{(i)}$ very fast, but their convergence stalls if the error is a smooth function. This fact is exploited by multigrid methods, that build a fine-to-coarse hierarchy $\{\mathcal{M} = \mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_k\}$ of the computation domain $\mathcal{M}$ and solve the linear system hierarchically from coarse to fine [**?**, **?**].

After a few (pre-)smoothing iterations on the finest level $\mathcal{M}_0$ the high frequencies of the error are removed and the solver becomes inefficient. However, the remaining low frequency error $\mathbf{e}_0 = \mathbf{x}^* - \mathbf{x}_0$ on $\mathcal{M}_0$ corresponds to higher frequencies when restricted to the coarser level $\mathcal{M}_1$ and therefore can be removed efficiently on $\mathcal{M}_1$. Hence the error is solved for using the residual equations $\mathbf{Ae}_1 = \mathbf{r}_1$ on $\mathcal{M}_1$, where $\mathbf{r}_1 = R_{0\to1}\mathbf{r}_0$ is the residual on $\mathcal{M}_0$ transferred to $\mathcal{M}_1$ by a restriction operator $R_{0\to1}$. The result is prolongated back to $\mathcal{M}_0$ by $\mathbf{e}_0 \leftarrow P_{1\to0}\mathbf{e}_1$ and used to correct the current approximation: $\mathbf{x}_0 \leftarrow \mathbf{x}_0 + \mathbf{e}_0$. Small high-frequency errors due to the prolongation are finally removed by a few post-smoothing steps on $\mathcal{M}_0$. The recursive application of this two-level approach to the whole hierarchy can be written as

$$\Phi_i \;=\; S_\mu \, P_{i+1\to i} \, \Phi_{i+1} \, R_{i\to i+1} \, S_\lambda \;\;,$$

with λ and μ pre- and post-smoothing iterations, respectively. One recursive run is known as a *V-cycle* iteration.

Another concept is the method of *nested iterations*, that exploits the fact that iterative solvers are very efficient if the starting value is sufficiently close to the actual solution. One starts by computing the exact solution on the coarsest level $\mathcal{M}_k$, which can be done efficiently since the system $\mathbf{A}_k\mathbf{x}_k = \mathbf{b}_k$ corresponding to the restriction to $\mathcal{M}_k$ is small. The prolongated solution $P_{k\to k-1}\mathbf{x}_k^*$ is then used as starting value for iterations on $\mathcal{M}_{k-1}$, and this process is repeated until the finest level $\mathcal{M}_0$ is reached and the solution $\mathbf{x}_0^* = \mathbf{x}^*$ is computed.

The remaining question is how to iteratively solve on each level. The standard method is to use one or two V-cycle iterations, leading to the so-called *full multigrid* method. However, one can also use an iterative smoothing solver (e.g., Jacobi or CG) on each level and completely avoid V-cycles. In the latter case the number of iterations m_i on level i must not be constant, but

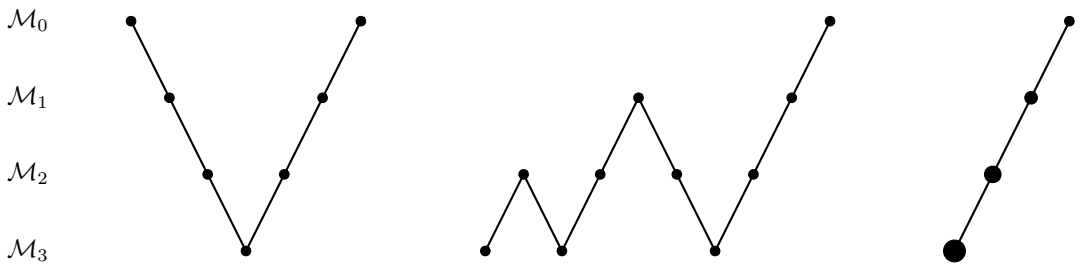


Figure 12.1: A schematic comparison in terms of visited multigrid levels for V-cycle (*left*), full multigrid with one V-cycle per level (*center*), and cascading multigrid (*right*).

instead has to be chosen as $m_i = m\,\gamma^i$ to decrease exponentially from coarse to fine [**?**]. Besides the easier implementation, the advantage of this *cascading multigrid* method is that once a level is computed, it is not involved in further computations and can be discarded. A comparison of the three methods in terms of visited multigrid levels is given in Fig. **??**.

Due to the logarithmic number of hierarchy levels $k = O(\log n)$ the full multigrid method and the cascading multigrid method can both be shown to have linear asymptotic complexity, as opposed to quadratic for non-hierarchical iterative methods. However, they cannot exploit synergy for multiple right hand sides, which is why factorization-based approaches are clearly preferable in such situations, as we will show in the next section.

Since in our case the discrete computational domain $\mathcal{M}$ is an irregular triangle mesh instead of a regular 2D or 3D grid, the coarsening operator for building the hierarchy is based on mesh decimation techniques [**?**]. The shape of the resulting triangles is important for numerical robustness, and the edge lengths on the different levels should mimic the case of regular grids. Therefore the decimation usually removes edges in the order of increasing lengths, such that the hierarchy levels have uniform edge lengths and triangles of bounded aspect ratio. The simplification from one hierarchy level $\mathcal{M}_i$ to the next coarser one $\mathcal{M}_{i+1}$ should additionally be restricted to remove a *maximally independent set* of vertices, i.e., no two removed vertices $v_j, v_l \in \mathcal{M}_i \setminus \mathcal{M}_{i+1}$ are connected by an edge $e_{jl} \in \mathcal{M}_i$. In [**?**] some more efficient alternatives to this kind of hierarchy are described.

The linear complexity of multi-grid methods allows for the highly efficient solution even of very complex systems. However, the main problem of these solvers is their quite involved implementation, since special care has to be taken for the hierarchy building, for special multigrid pre-conditioners, and for the inter-level conversion by restriction and prolongation operators. Additionally, appropriate numbers of iterations per hierarchy level have to be chosen. These numbers have to be chosen either by heuristic or experience, since they not only depend on the problem (structure of $\mathbf{A}$), but also on its specific instance (values of $\mathbf{A}$). A detailed overview of these techniques is given in [**?**]. A highly efficient multigrid solver with specially tuned restriction and prolongation operators was proposed for interactive shape deformation in [**?**].

12.5 Sparse Direct Solvers

The use of direct solvers for large sparse linear systems is often neglected, since naïve direct methods have complexity $O(n^3)$, as described above. The problem is that even when the matrix

$\mathbf{A}$ is sparse, the factorization will not preserve this sparsity, such that the resulting Cholesky factor is a dense lower triangular matrix.

However, an analysis of the factorization process reveals that a *band-limitation* of the matrix $\mathbf{A}$ will be preserved. If the matrix $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ has a certain bandwidth β then so has its factor $\mathbf{L}$. An even stricter bound is that the so-called *envelope* (the leading zeros of each row) is preserved [**?**]. This additional structure can be exploited in both the factorization and the solution process, such that their complexities reduce from $O(n^3)$ and $O(n^2)$ to linear complexity in the number of non-zeros $\text{nz}(\mathbf{A})$ of $\mathbf{A}$ [**?**]. Since usually $\text{nz}(\mathbf{A}) = O(n)$, this is the same linear complexity as for multigrid solvers. However, in particular for multiple right-hand side problems, sparse direct methods turned out to be more efficient compared to multigrid solvers.

If matrices are sparse, but not band-limited or profile-optimized, the first step is to minimize the matrix envelope, which can be achieved by symmetric row and column permutations $\mathbf{A} \leftarrow \mathbf{P}^T\mathbf{A}\mathbf{P}$ using a permutation matrix $\mathbf{P}$, i.e., a re-ordering of the mesh vertices. Although this problem is NP complete, several good heuristics exist, of which we will outline the most commonly used in the following. All of these methods work on the undirected *adjacency graph* $\text{Adj}(\mathbf{A})$ corresponding to the non-zeros of $\mathbf{A}$, i.e., two nodes $i, j \in \{1, \ldots, n\}$ are connected by an edge if and only if $\mathbf{A}_{ij} \neq 0$.

The standard method for envelope minimization is the *Cuthill-McKee* algorithm [**?**], that picks a start node and renumbers all its neighbors by traversing the adjacency graph in a greedy breadth-first manner. Reverting this permutation further improves the re-ordering, leading to the *reverse Cuthill-McKee* method (RCMK) [**?**]. The result $\mathbf{P}^T\mathbf{A}\mathbf{P}$ of this matrix re-ordering is depicted in the second row of Fig. **??**.

Since no special pivoting is required for the Cholesky factorization, the non-zero structure of its matrix factor $\mathbf{L}$ can symbolically be derived from the non-zero structure of the matrix $\mathbf{A}$ alone, or, equivalently, from its adjacency graph. The *minimum degree* algorithm (MD) and its variants [**?**, **?**] directly work on the graph interpretation of the Cholesky factorization and try to minimize fill-in elements $\mathbf{L}_{ij} \neq 0 = \mathbf{A}_{ij}$. While the resulting re-orderings do not yield a band-structure (which implicitly limits fill-in), they usually lead to better results compared to RCMK (cf. Fig. **??**, third row).

The last class of re-ordering approaches is based on graph partitioning. A matrix $\mathbf{A}$ whose adjacency graph has m separate connected components can be restructured to a block-diagonal matrix of m blocks, such that the factorization can be performed on each block individually. If the adjacency graph is connected, a small subset S of nodes, whose elimination would separate the graph into two components of roughly equal size, is found by one of several heuristics [**?**]. This graph-partitioning results in a matrix consisting of two large diagonal blocks (two connected components) and $|S|$ rows representing their connection (separator S). Recursively repeating this process leads to the method of *nested dissection* (ND), resulting in matrices of the typical block structure shown in the bottom row of Fig. **??**. Besides the obvious fill-in reduction, these systems also allow for easy parallelization of both the factorization and the solution.

Analogously to the dense direct solvers, the factorization can be exploited to solve for different right hand sides in a very efficient manner, since only the back-substitution has to be performed again. Moreover, for sparse direct methods no additional parameters have to be chosen in a problem-dependent manner (like iteration numbers for iterative solvers). The only degree of freedom is the matrix re-ordering, which only depends on the symbolic structure of the problem and therefore can be chosen quite easily. A highly efficient implementation is publicly available in the TAUCS library [**?**] or recently in COLMOD [**?**].

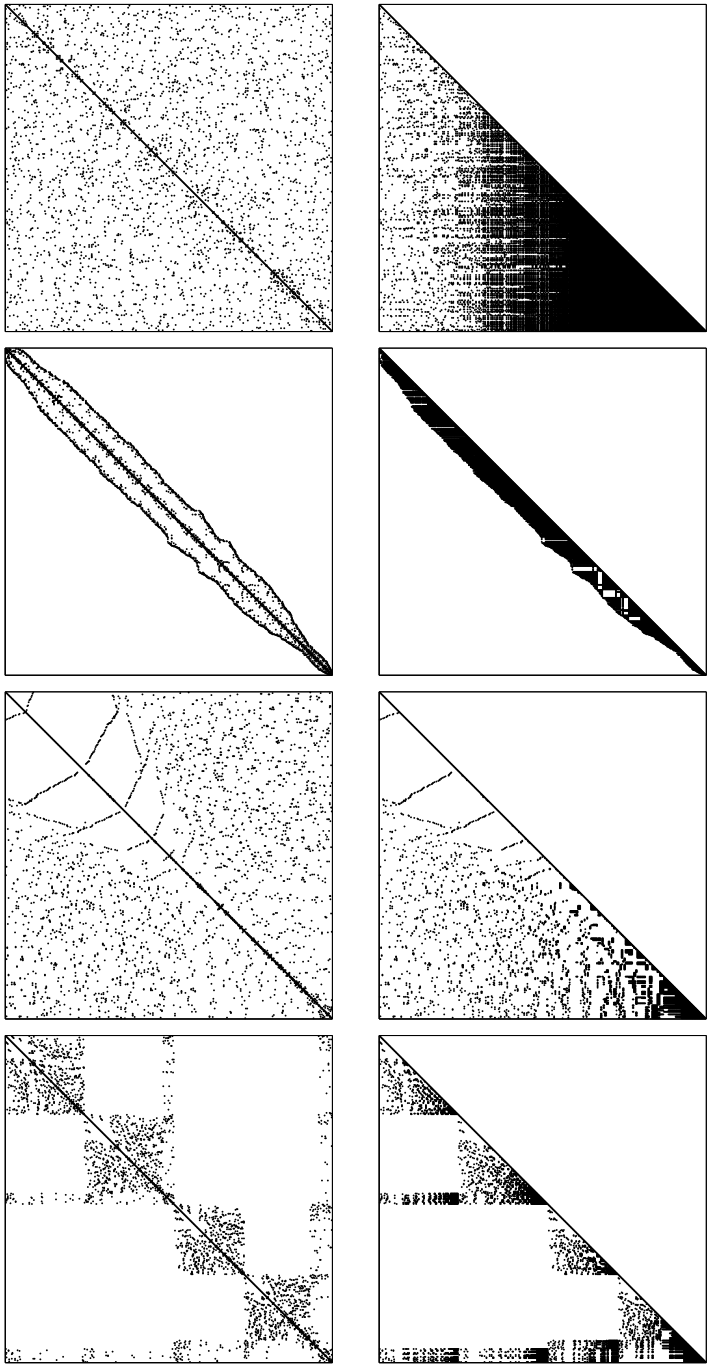


Figure 12.2: The top row shows the non-zero pattern of a typical 500 × 500 matrix **A** and its Cholesky factor **L**, corresponding to a Laplacian system on a triangle mesh. Although **A** is highly sparse (3502 non-zeros), the factor **L** is dense (36k non-zeros). The reverse Cuthill-McKee algorithm minimizes the envelope of the matrix, resulting in 14k non-zeros of **L** (*2nd row*). The minimum degree ordering avoids fill-in during the factorization, which decreases the number of non-zeros to 6203 (*3rd row*). The last row shows the result of a nested dissection method (7142 non-zeros), that allows for parallelization due to its block structure.

12.6 Non-Symmetric Indefinite Systems

When the assumptions about the symmetry and positive definiteness of the matrix $\mathbf{A}$ are not satisfied, optimal methods like the Cholesky factorization or conjugate gradients cannot be used. In this section we shortly outline which techniques are applicable instead.

From the class of iterative solvers the bi-conjugate gradients algorithm (BiCG) is typically used as a replacement of the conjugate gradients method [**?**]. Although working well in most cases, BiCG does not provide any theoretical convergence guarantees and has a very irregular non-monotonically decreasing residual error for ill-conditioned systems. On the other hand, the GMRES method converges monotonically with guarantees, but its computational cost and memory consumption increase in each iteration [**?**]. As a good trade-off, the stabilized Bi-CGSTAB [**?**] represents a mixture between the efficient BiCG and the smoothly converging GMRES; it provides a much smoother convergence and is reasonably efficient and easy to implement.

When considering dense direct solvers, the Cholesky factorization cannot be used for general matrices. Therefore the LU factorization is typically employed (instead of QR or SVD), since it is similarly efficient and also extends well to sparse direct methods. However, (partial) row and column pivoting is essential for the numerical robustness of the LU factorization, since this avoids zeros on the diagonal during the factorization process.

Similarly to the Cholesky factorization, it can be shown that the LU factorization also preserves the band-width and envelope of the matrix $\mathbf{A}$. Techniques like the minimum degree algorithm generalize to non-symmetric matrices as well. But as for dense matrices, the banded LU factorization relies on partial pivoting in order to guarantee numerical stability. In this case, two competing types of permutations are involved: symbolic permutations for matrix re-ordering and pivoting permutations ensuring numerical robustness. As these permutations cannot be handled separately, a trade-off between stability and fill-in minimization has to be found, resulting in a considerably more complex factorization. A highly efficient implementation of a sparse LU factorization is provided by the SuperLU library [**?**].

12.7 Comparison

In the following we compare the different kinds of linear system solvers for Laplacian as well as for bi-Laplacian systems. All timings reported in this and the next section were taken on a 3.0GHz Pentium4 running Linux. The iterative solver (CG) from the `gmm++` library [**?**] is based on the conjugate gradients method and uses an incomplete $\mathbf{LDL}^T$ factorization as preconditioner. The cascading multigrid solver of [**?**] (MG) performs preconditioned conjugate gradient iterations on each hierarchy level and additionally exploits SSE instructions in order to solve for up to four right-hand sides simultaneously. The direct solver (LL^T) of the TAUCS library [**?**] employs nested dissection re-ordering and a sparse complete Cholesky factorization. Although our linear systems are symmetric, we also compare to the popular SuperLU solver [**?**], which is based on a sparse LU factorization.

Iterative solvers have the advantage over direct ones that the computation can be stopped as soon as a sufficiently small error is reached, which — in typical computer graphics applications — does not have to be the highest possible precision. In contrast, direct methods always compute the exact solution up to numerical round-off errors, which in our application examples actually was more precise than required. The stopping criteria of the iterative methods have therefore been chosen to yield sufficient results, such that their quality is comparable to that achieved by

direct solvers. The resulting residual errors were allowed to be about one order of magnitude larger than those of the direct solvers.

Table **??** shows timings for the different solvers on Laplacian systems $\Delta_{\mathcal{S}}\mathbf{P} = \mathbf{B}$ of 10k to 50k and 100k to 500k unknowns. For each solver three columns of timings are given:

Setup: Computing the cotangent weights for the Laplace discretization and building the matrix structure (done per-level for the multigrid solver).

Precomputation: Preconditioning (*iterative*), building the hierarchy by mesh decimation (*multigrid*), matrix re-ordering and sparse factorization (*direct*).

Solution: Solving the linear system for three different right-hand sides corresponding to the x, y, and z components of the free vertices $\mathbf{P}$.

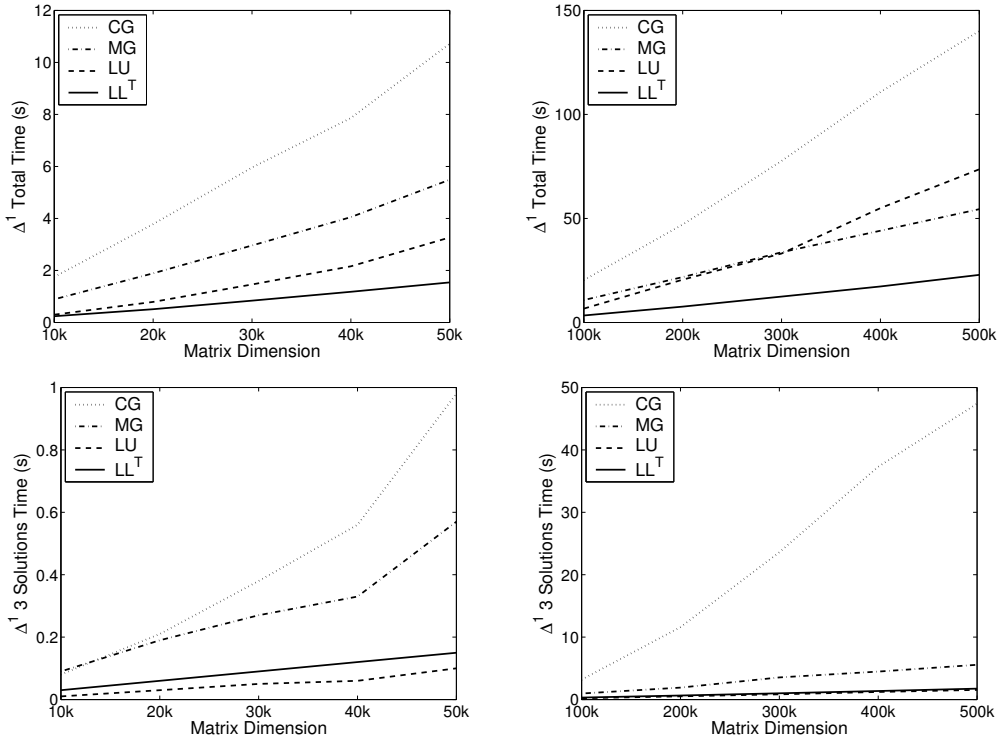
Due to its effective preconditioner, which computes a sparse incomplete factorization, the iterative solver scales almost linearly with the system complexity. However, for large and thus ill-conditioned systems it breaks down. Notice that without preconditioning the solver would not converge for the larger systems. The experiments clearly verify the linear complexity of multigrid and sparse direct solvers. Once their sparse factorizations are pre-computed, the computational costs for actually solving the system are about the same for the LU and Cholesky solver. However, they differ significantly in the factorization performance, because the numerically more robust Cholesky factorization allows for more optimizations, whereas pivoting is required for the LU factorization to guarantee robustness. This is the reason for the break-down of the LU solver, such that the multigrid solver is more efficient in terms of total computation time for the larger systems.

Interactive applications often require to solve the same linear system for several right-hand sides (e.g. once per frame), which typically reflects the change of boundary constraints due to user interaction. For such problems the solution times, i.e., the third columns of the timings, are more relevant, as they correspond to the per-frame computational costs. Here the precomputation of a sparse factorization pays off and the direct solvers are clearly superior to the multigrid method.

Table **??** shows the same experiments for bi-Laplacian systems $\Delta_{\mathcal{S}}^2\mathbf{P} = \mathbf{B}$ of the same complexity. In this case, the matrix setup is more complex, the matrix condition number is squared, and the sparsity decreases from 7 to 19 non-zeros per row. Due to the higher condition number the iterative solver takes much longer and even fails to converge on large systems. In contrast, the multigrid solver converges robustly without numerical problems; notice that constructing the multigrid hierarchy is almost the same as for the Laplacian system (up to one more ring of boundary constraints). The computational costs required for the sparse factorization are proportional to the increased number of non-zeros per row. The LU factorization additionally has to incorporate pivoting for numerical stability and failed for larger systems. In contrast, the Cholesky factorization worked robustly in all experiments.

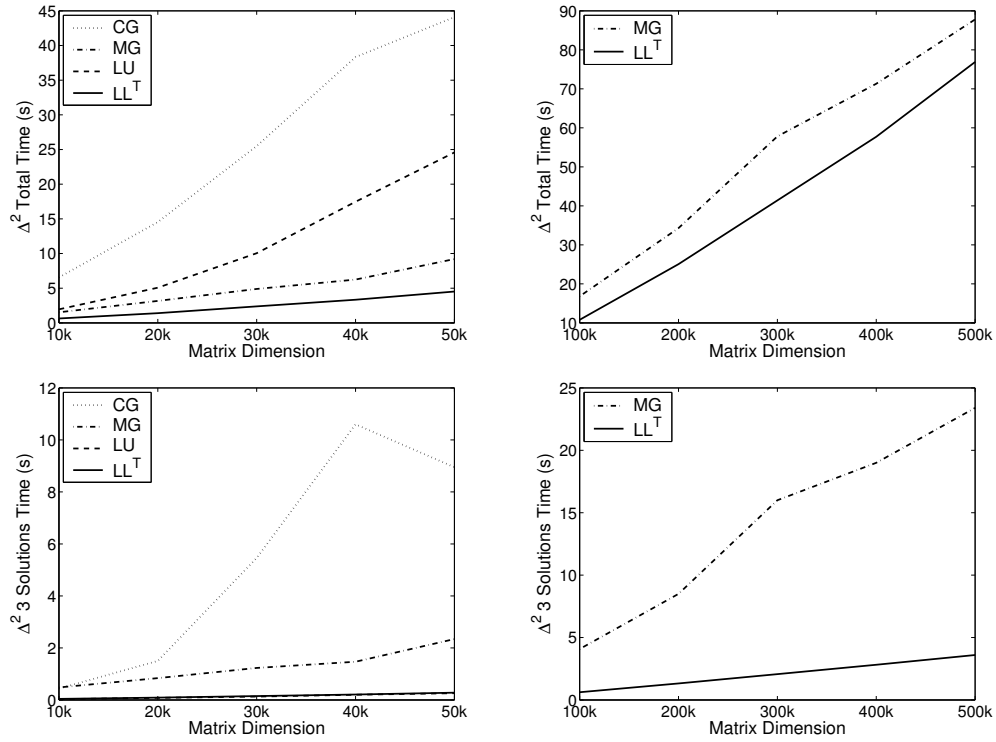
The memory consumption of the multigrid method is mainly determined by the meshes representing the different hierarchy levels. In contrast, the memory required for the Cholesky factorization depends significantly on the sparsity of the matrix, too. On the 500k example the multigrid method and the direct solver need about 1GB and 600MB for the Laplacian system, and about 1.1GB and 1.2GB for the bi-Laplacian system. Hence, the direct solver would not be capable of factorizing Laplacian systems of higher order on current PCs, while the multigrid method would succeed.

These comparisons show that direct solvers are a valuable and efficient alternative to multigrid methods even if the linear systems are highly complex. In all experiments the sparse Cholesky



Size	Iterative CG	Multigrid	LU	LL^T
10k	0.11/1.56/0.08	0.15/0.65/0.09	0.07/0.22/0.01	0.07/0.14/0.03
20k	0.21/3.36/0.21	0.32/1.38/0.19	0.14/0.62/0.03	0.14/0.31/0.06
30k	0.32/5.26/0.38	0.49/2.20/0.27	0.22/1.19/0.05	0.22/0.53/0.09
40k	0.44/6.86/0.56	0.65/3.07/0.33	0.30/1.80/0.06	0.31/0.75/0.12
50k	0.56/9.18/0.98	0.92/4.00/0.57	0.38/2.79/0.10	0.39/1.00/0.15
100k	1.15/16.0/3.19	1.73/8.10/0.96	0.79/5.66/0.21	0.80/2.26/0.31
200k	2.27/33.2/11.6	3.50/16.4/1.91	1.56/18.5/0.52	1.59/5.38/0.65
300k	3.36/50.7/23.6	5.60/24.6/3.54	2.29/30.0/0.83	2.35/9.10/1.00
400k	4.35/69.1/37.3	7.13/32.5/4.48	2.97/50.8/1.21	3.02/12.9/1.37
500k	5.42/87.3/47.4	8.70/40.2/5.57	3.69/68.4/1.54	3.74/17.4/1.74

Table 12.1: Comparison of different solvers for Laplacian systems $\Delta_{\mathcal{S}}\mathbf{P} = \mathbf{B}$ of 10k to 50k and 100k to 500k free vertices $\mathbf{P}$. The three timings for each solver represent matrix setup, pre-computation, and three solutions for the x, y, and z components of $\mathbf{P}$. The graphs in the upper row show the total computation times (sum of all three columns). The center row depicts the solution times only (3rd column), as those typically determine the per-frame cost in interactive applications.



Size	Iterative CG	Multigrid	LU	LL^T
10k	0.33/5.78/0.44	0.40/0.65/0.48	0.24/1.68/0.03	0.24/0.35/0.04
20k	0.64/12.4/1.50	0.96/1.37/0.84	0.49/4.50/0.08	0.49/0.82/0.09
30k	1.04/19.0/5.46	1.40/2.26/1.23	0.77/9.15/0.13	0.78/1.45/0.15
40k	1.43/26.3/10.6	1.69/3.08/1.47	1.07/16.2/0.20	1.08/2.05/0.21
50k	1.84/33.3/8.95	2.82/4.05/2.34	1.42/22.9/0.26	1.42/2.82/0.28
100k	—	4.60/8.13/4.08	2.86/92.8/0.73	2.88/7.29/0.62
200k	—	9.19/16.6/8.50	—	5.54/18.2/1.32
300k	—	17.0/24.8/16.0	—	8.13/31.2/2.07
400k	—	19.7/32.6/19.0	—	10.4/44.5/2.82
500k	—	24.1/40.3/23.4	—	12.9/60.4/3.60

Table 12.2: Comparison of different solvers for bi-Laplacian systems $\Delta_S^2 \mathbf{P} = \mathbf{B}$ of 10k to 50k and 100k to 500k free vertices $\mathbf{P}$. The three timings for each solver represent matrix setup, pre-computation, and three solutions for the components of $\mathbf{P}$. The graphs in the upper row again show the total computation times, while the center row depicts the solution times only (3rd column). For the larger systems, the iterative solver and the sparse LU factorization fail to compute a solution.

solver was faster than the multigrid method, and if the system has to be solved for multiple right-hand sides, the precomputation of a sparse factorization is even more beneficial.

Speaker Biographies

Pierre Alliez is a researcher at the GEOMETRICA project-team of INRIA Sophia Antipolis - Méditerranée, France. His research interests include various topics commonly referred to as geometry processing: Surface reconstruction, mesh generation, surface remeshing, mesh parameterization, mesh compression. He studied Image Processing, Computer Vision and Computational Geometry at the University of Nice Sophia-Antipolis, France, where he received his MS degree in 1997. He was awarded a Ph.D. in Image and Signal Processing in 2000 from the École Nationale Supérieure des Télécommunications, Paris. He then spent a year as a post-doctoral researcher at the University of Southern California with Mathieu Desbrun. Dr. Alliez has served on various program committees, including EUROGRAPHICS, SIGGRAPH and the Symposium on Geometry Processing. He was awarded in 2005 the EUROGRAPHICS young researcher award for his contributions to computer graphics and geometry processing. He is co-chair of the Symposium on Geometry Processing 2008.

Mario Botsch is a post-doctoral lecturer and senior researcher at the Computer Graphics Laboratory of ETH Zurich, Switzerland. He received his MS in Mathematics from the University of Erlangen-Nuremberg, Germany, in 1999. From 1999 to 2000 he worked as research associate at the Max-Planck Institute for Computer Science in Saarbrücken, Germany. From 2001 to 2005 he worked as research associate and PhD candidate with Prof. Dr. Leif Kobbelt at the RWTH Aachen, Germany, from where he received his PhD in 2005. He is an experienced speaker and presented papers and courses at SIGGRAPH and EUROGRAPHICS. Dr. Botsch has served on various program committees including EUROGRAPHICS and the Symposium on Geometry Processing, and has co-chaired the Symposium on Point-Based Graphics in 2006 and 2007. Recently, he received the EUROGRAPHICS 2007 young researcher award for his contributions to computer graphics and geometry processing. Dr. Botsch's research interests include geometry processing in general, and mesh generation, mesh optimization, and shape editing in particular.

Leif Kobbelt is a full Professor of Computer Science and the Head of the Computer Graphics group at the RWTH Aachen University of Technology, Germany. His research interests include all areas of Computer Graphics and Geometry Processing with a focus on multiresolution and freeform modeling, 3D model optimization, as well as the efficient handling of polygonal mesh data. He was a senior researcher at the Max-Planck Institute for Computer Science in Saarbrücken, Germany, from 1999 to 2000 after he received his Habilitation degree from the University of Erlangen, where he worked from 1996 to 1999. In 1995/96 he spent a post-doc year at the University of Wisconsin, Madison. He received his PhD and MS degrees from the University of Karlsruhe, Germany, in 1994 and 1992, respectively. Dr. Kobbelt's research work during the last years resulted in numerous publications in top scientific journals and international conferences. He is invited regularly to give keynote presentations and tutorial lectures. For his contributions he received several scientific awards. He has ongoing collaborations with colleagues in Europe, North America, and Asia, and frequently serves on international program committees. He organized and co-chaired several workshops and conferences.

Bruno Lévy is a researcher with INRIA. He is the head of the ALICE research group. He did a Ph.D. (1996-1999) with J.-L. Mallet, on 3D modeling for oil exploration, in the INPL (Nancy, France). His Ph.D. thesis was awarded the SPECIF price in 2000 (best French Ph.D. thesis in Computer Sciences). He then did a post-doc in Stanford university, in the SCCM group (headed by G. Golub) where he learned numerical optimization, and in the earth sciences group (headed by A. Journel and K. Aziz) where he learned finite element modeling. He has served on various program committees, including Eurographics, Visualization and the Symposium on Geometry Processing. He was program co-chair of the ACM Symposium on Solid and Physical Modeling in 2007 and 2008. His main contributions concern texture mapping and parameterization methods for triangulated surfaces, that are now used by several popular 3D modeling software (including Maya, Catia, Silo, Blender and Gocad).

Mark Pauly is an assistant professor at the computer science department of ETH Zurich, Switzerland. From August 2003 to March 2005 he was a postdoctoral scholar at Stanford University, where he also held a position as visiting assistant professor during the summer of 2005. He received his Ph.D. degree in 2003 from ETH Zurich and his M.S. degree in computer science in 1999 from the Technical University of Kaiserslautern, Germany. Dr. Pauly has served on various program committees including ACM SIGGRAPH, EUROGRAPHICS, and the Symposium on Geometry Processing, and has co-chaired the Symposium on Point-Based Graphics. He is an experienced speaker and has previously presented courses at SIGGRAPH and EUROGRAPHICS. Dr. Pauly was awarded the EUROGRAPHICS 2006 young researcher award for his contributions to computer graphics and geometry processing. His research interests include geometry processing, multi-scale shape modeling and analysis, physics-based animation, and computational geometry.