CONSTRAINED DEFORMATION FOR EVOLUTIONARY OPTIMIZATION

Daniel Sieger

CONSTRAINED DEFORMATION *for* *Evolutionary Optimization*

Constrained Deformation for Evolutionary Optimization.
A dissertation for the degree Doctor of Engineering.
Faculty of Technology, Bielefeld University.

Printed on non-aging paper in compliance with DIN-ISO 9706.

To my parents.

I hate meshes. I cannot believe how hard this is. Geometry is hard.

David Baraff
Senior Scientist, Pixar Animation Studios

There is no royal road to geometry.

Euclid

ABSTRACT

This thesis investigates shape deformation techniques for their use in design optimization tasks. In the first part, we introduce state-of-the-art deformation methods and evaluate them in a set of representative benchmarks. Based on these benchmarking results, we derive essential criteria and features a deformation technique should satisfy in order to be successfully applicable within design optimization. In the second part, we concentrate on the application and improvement of deformation techniques based on radial basis functions. We present and evaluate a unified framework for surface and volume mesh deformation and investigate questions of performance and scalability. In the final third part, we concentrate on the integration of additional constraints into the deformation, thereby improving the overall effectiveness of the design optimization process and fostering the creation of more feasible and producible design variations. We present a novel shape deformation technique that effectively maintains different types of geometric constraints such as planarity, circularity, or characteristic feature lines during deformation. At the same time, our method provides a unique level of modeling flexibility, quality, robustness, and scalability. Finally, we integrate techniques for automatic constraint detection directly into our deformation framework, thereby making our method more easily applicable within complex design optimization scenarios.

ACKNOWLEDGMENTS

First and foremost, I am deeply thankful to my supervisor Prof. Dr. Mario Botsch for sparking my interest and passion for the fascinating areas of computer graphics and geometry processing. His boundless enthusiasm, scientific intuition, as well as his analytic and critical way of thinking provided me with invaluable guidance throughout these years.

I am also highly thankful to my second supervisor Dr.-Ing. Stefan Menzel for introducing me to the exciting field of evolutionary design optimization, for giving me good guidance, and for keeping me conscious of the application-oriented and user-centered perspective.

I also thank my former collaborators: Pierre Alliez from INRIA for the joint work on mesh optimization, Matthew Staten from Sandia for sharing his research data, Sebastian Martin from VirtaMed AG for his guidance on polygonal finite elements and moving least squares, and Sofien Bouaziz from EPFL for sharing his insights into constrained geometry processing. I also thank my former master students Jascha Achenbach and Sergius Gaulik for their contributions to automatic constraint detection. Special thanks go to Felix Rabe for providing me with inspiring discussions and probably even more important distractions, as well as to Hendrik Buschmeier for all the discussions about typography and style, and for ensuring a steady supply with high quality coffee beans. Furthermore, I thank my former colleagues of the Bielefeld Computer Graphics and Geometry Processing group for being such great companions and for making work a fun place to be.

I am also highly grateful to the Honda Research Institute Europe for giving me the opportunity to work on this fascinating research topic and for supporting me financially.

Finally, I am deeply thankful to my friends and family, in particular my parents Mara and Peter. Their support and encouragement was of tremendous value for me throughout all these years.

CONTENTS

CHAPTER 1

INTRODUCTION

We begin this thesis by providing a brief motivation for the use of computer tools to design, develop, and manufacture complex industrial products such as cars, aircrafts, or electronic devices. We continue with a concise introduction to the discipline of design optimization and its manifold benefits and goals. More specifically, we describe how shape deformation methods increase the performance and effectiveness of the optimization process. We motivate the integration of additional constraints into the deformation as effective means to increase the usefulness of the optimization results. Finally, we summarize our core contributions and formulate a set of research questions that we investigate throughout this thesis.

MOTIVATION

Ever since their inception during the midth of the 20th century, *computer-aided design and manufacturing* (CAD/CAM) technologies continuously changed the product development and production processes in virtually all branches of industry. What started with the use of spline-based modeling techniques for the design of car bodies as well as aircraft fuselages and wings, is now a cornerstone in a multitude of disciplines such as mechanical, electrical, and civil engineering, as well as architectural design. The key benefit of CAD methods is that they allow for the construction of physical artifacts—vehicles, buildings, electronic devices—of unparalleled complexity and performance. Even today one can observe a continuous growth within areas such as virtual fabrication and prototype development, which increasingly replace the construction of costly real-world prototypes and their evaluation through time-consuming experiments. Recent emerging technologies such as increasingly powerful and sophisticated 3D printing techniques further demonstrate that the developments in this field will continue to profoundly change our society's product development and production processes.

One of the key benefits of the use of computer-based modeling and simulation techniques is that they allow for the exploration and evaluation of different design

prototypes at an early stage of the development process, thereby reducing the need to actually build physical artifacts of the prototype and offering the possibility to identify potential problems with the design in advance. The discipline of *design optimization* aims at the discovery and evaluation of alternative designs with improved physical or aesthetic properties. The development process typically starts with the creation of an initial prototype using a CAD modeling tool. Depending on the particular optimization scenario, subsequent steps create a polygon surface mesh from the CAD model as well as a volumetric simulation mesh for physical performance evaluation, e.g., using computational fluid dynamics (CFD) simulations for aerodynamic performance calculation, or finite element methods (FEM) for structural mechanics simulations. Design variations are then created based on physical performance during simulation.

A challenging task within the optimization process is to provide effective means to create alternate designs. Changing the CAD model directly is typically prohibitive, since the repeated surface and volume meshing steps required for performing the physics simulations are highly time-consuming. In case of complex geometries the meshing steps might even require manual interaction by an expert. An alternative is to generate the meshes only once and to use *shape deformation techniques* to adapt the meshes of the initial design prototype directly. This way, the design optimization process can be performed in a fully automatic and parallel manner, which is of particular importance when using stochastic global optimization techniques—such as evolutionary algorithms—which typically require the creation and evaluation of a large number of design variations in order to find a feasible solution.

We illustrate a simplified and generic design optimization loop employing shape deformation methods in figure 1.1: An initial prototype's performance is evaluated using simulation. Based on the results an optimization algorithm determines possible search directions for new design variations, which are then created using shape deformation techniques. This process is iterated until convergence or until a maximum number of iterations is reached and finally the resulting optimized design is obtained.

While the use of shape deformation techniques allows to effectively create new design variations during optimization, a major challenge remains the creation of actually *useful* and *feasible* designs. In order to be of practical relevance the design variations have to meet essential *constraints*. Production limitations may impose constraints on the shape of components, e.g., certain parts might need to stay

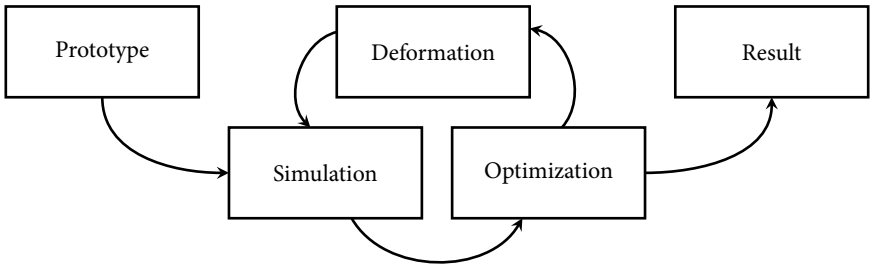


Figure 1.1: A simulation-based design optimization loop using shape deformation.

planar, circular, or cylindrical. Usability requirements typically impose additional constraints such as a minimum height of a car door to be accessible for humans. Especially when optimizing components with a strong influence on the visual appeal—such as the overall outer shape of a car body—the aesthetic demands of the designer or customer constitute important restrictions on the design. Finally, legal or regulatory requirements eventually apply, e.g., a modern Formula One racing car has to fulfill a multitude of technical norms and restrictions, including exact specifications of minimum and/or maximum dimensions and distances of certain components (FIA 2016). Similar restrictions apply for ordinary passenger cars as well, such as exhibited by the U.S. motor vehicle safety standards and regulations (NHTSA 1998).

Within a classical design optimization process such constraints are typically incorporated by penalizing design variations that violate given constraints, i.e., during performance evaluation a penalty term is added in case of a constraint violation. This approach, however, requires the costly *creation* and *evaluation* of unfeasible designs. Furthermore, the formulation of appropriate penalty terms is a non-trivial task of its own and potentially involves a certain amount of experimentation until a suitable set of penalty terms and parameters is found. Alternatively, one can explicitly check newly created design variations for constraint violations and discard them if they do so. While this approach circumvents the need to perform a full performance evaluation, it makes it difficult to balance between constraint satisfaction and potential performance improvements: In some cases a slight violation of a certain constraint might result in significant performance gains and therefore might be still acceptable or even desirable.

In contrast to both approaches, we propose to use *constrained deformation techniques* in order to incorporate the given constraints directly into the deformation. This approach effectively prevents infeasible designs from being created and evaluated, thereby increasing the overall performance and effectiveness of the design optimization process. Furthermore, the use of constrained deformation methods also eases the setup procedure for the engineer since constraints can be specified intuitively on the design rather than through the custom formulation of penalty terms.

CONTRIBUTIONS

The major goal of this thesis is to investigate the use of shape deformation methods within design optimization, with a particular focus on evolutionary optimization and—in the end—the incorporation of geometric constraints into the deformation. In order to meet this goal, we formulate a set of three research questions that we investigate and provide answers to throughout this thesis:

1. What is a *good* shape deformation technique for design optimization?
   We investigate state-of-the-art deformation methods, analyze and compare them in representative synthetic and application-oriented benchmarks, provide a detailed assessment of their individual strengths and weaknesses. Our thorough analysis of existing methods reveals significant differences between individual techniques and allows us to derive essential criteria a deformation method should satisfy in order to be successfully applicable within design optimization tasks.
2. How can we *apply* and *improve* existing techniques?
   We analyze shortcomings in traditional design optimization processes and present a unified framework for combined surface and volume mesh deformation in accordance to changes in the corresponding parametric CAD geometry. We investigate the prevention of self-intersections through successive splitting, and show how to boost deformation performance and scalability through the use of advanced linear solvers.

3. How can we incorporate additional *constraints* into the deformation?
   We devise a novel shape deformation method that directly integrates geometric constraints into the deformation process. While maintaining the quality and robustness of existing methods our new technique provides drastically improved modeling flexibility and scalability. Finally, in order to simplify the setup process of the deformation method we integrate automatic constraint detection based on geometric primitive fitting.

OUTLINE

The structure of this thesis follows the above research questions and is organized into three parts: Shape deformation methods, advanced deformation methods based on radial basis functions, and constrained deformation. We begin with an introduction of basic concepts as well as a review of related work in chapter 2. In the first part, we introduce state-of-the-art deformation methods (chapter 3) and analyze their individual strengths and weaknesses in a series of representative synthetic benchmarks (chapter 4). We conclude our analysis and evaluation by applying the most promising methods within a prototypical evolutionary design optimization scenario in chapter 5.

Based on the benchmarking results of the first part, we promote the use of scattered data approximation methods for shape deformation in the second part. We present advanced deformation techniques based on radial basis functions for design optimization in chapter 6. This includes a unified framework for combined surface and volume mesh deformation according to a modified CAD geometry, the prevention of self-intersections through splitting, as well as the use of advanced linear solvers for increased performance and scalability. We evaluate our framework by comparing it to other recent mesh deformation methods in chapter 7 and show that our approach achieves high quality results more reliably and robustly.

In the third part, we investigate the integration of additional constraints into the deformation process. To this end, we devise a novel shape deformation method based on moving least squares approximation (chapter 8). Our new method offers a high level of modeling flexibility, deformation quality, and scalability. At the same time, we directly incorporate constraints into the deformation, thereby fostering the creation of more feasible design variations during the optimization process. In order to make our method more easily applicable, we investigate methods for the

automatic detection of geometric constraints, thereby easing the setup procedure for the engineer. We compare our new method to our previous method in representative benchmarks and demonstrate its applicability in practical deformation examples. Finally, we conclude this thesis with a summary and discussion, and outline promising directions for future research.

CHAPTER 2

BACKGROUND

The primary goal of this chapter is to provide an introduction to shape deformation in general as well as to its application within design optimization in particular. We introduce key concepts employed throughout this thesis and provide a classification of different deformation methods. We describe the different interfaces used to control the deformation within both interactive modeling scenarios as well as design optimization tasks. At the same time, we review and discuss the current state of the art, and provide references to relevant surveys and introductory texts. Note, however, that we provide more detailed references for the individual topics and methods covered in this thesis within the corresponding chapters and sections.

2.1 SHAPE DEFORMATION

The manipulation of geometric shapes is a core task in geometric modeling, computer graphics, as well as several engineering disciplines. Consequently, shape deformation methods have been subject to extensive research, and a wide variety of techniques have been developed. However, since covering all of these methods in sufficient detail is beyond the scope of this thesis, we concentrate on a general overview and classification, as well as the introduction of key concepts frequently employed. Where appropriate, we refer to existing introductions and surveys, and we provide detailed references for the individual deformation methods covered in this thesis in the corresponding sections of chapter 3.

On a very abstract and general level, we can describe the task of shape deformation as follows: Given a concrete *geometric representation* of a model as well as a set of *input parameters*, compute an *updated representation* that conforms to some *desired properties*. In this general context, the geometry representation could be anything from a parametric CAD model, to an implicit surface, a polygon surface mesh, or a volumetric simulation mesh. The input parameters typically correspond to displacements specified by the user. Alternatively, they could also be the boundary conditions of a physical simulation used to compute the deforma-

tion. The desired properties vary widely depending on the particular application domain. Typical examples include the satisfaction of user-specified displacement constraints, physical plausibility, smoothness of the resulting shape, or visual appeal. In the following, we classify deformation methods based on the representations they operate on, their input parameters, as well as on how the deformation is actually computed.

On a fundamental level, we can classify deformation methods into two different classes: The first class are deformations based on actual *physical simulations*, such as presented in the seminal work of Terzopoulos et al. (1987). The second class employs purely *geometric methods*, such as the spline-based free-form deformation of Sederberg and Parry (1986). While physics-based methods inherently yield highly realistic results, they also require increased modeling effort and computation time, and they typically do not provide a very fine-grained level of control to the user. In contrast, geometric approaches typically only aim at physical plausibility and smoothness of the deformation, are rather simple to compute, and allow for more precise user control. In the context of design optimization, the precise control over the deformation is a particularly important aspect, e.g., in order meet production or usability requirements. Similarly, the ability to explore design variations that are outside of the space of shapes obtainable by physical simulation is important for discovering completely novel designs. Therefore, we focus on geometric—but physically inspired—methods throughout this thesis and refer the reader to the survey of Nealen et al. (2006) for an overview of physically-based deformation methods.

As a second step, we classify techniques by the type of the numerical problem they actually solve in order to compute the deformation, i.e., whether it is a *linear* or a *non-linear* one. While non-linear methods provide realistic results for deformations involving large rotational components, their computational costs and implementation complexity are drastically higher than that of linear methods. The selection of methods considered in this thesis is highly driven by our application domain—design optimization. Since the deformations occuring during an optimization procedure typically are rather small, we only have to deal with rather small-scale adjustments of the geometry without large rotations. Therefore, it is only reasonable to limit our investigation to linear deformation methods, i.e., approaches that solve a linear system in one form or another in order to compute the deformation.

As outlined in our general description of shape deformation a rather fundamental characteristic of a particular technique is the type of geometry representation the method operates on. Within a simulation-based design optimization scenario the initial prototype of a design is typically created using a parametric CAD modeling tool. However, creating new design variations directly based on the CAD model is typically prohibitive, since such an approach would require subsequent remeshing steps for each new design variation created. See the work of König and Wintermantel (2004) for an investigation of CAD-based evolutionary design optimization tools and the difficulties associated with such an approach. While recent advances in isogeometric analysis (Cottrell et al. 2009) and its approach to solve the simulation problem directly based on the CAD model give hope for future advancements in this direction, we concentrate on more traditional settings in this thesis. Therefore, we investigate methods operating on *discrete geometry representations* such as polygon surface meshes or volumetric simulation meshes in the following sections.

2.1.1 *Mesh-based Surface Deformation*

Common target shapes for design optimization are composed of sheet metal *surfaces*, such as car bodies, aircraft wings, or ship hulls. Probably the most widely used discrete geometry representation for such surfaces are *triangle meshes*. Therefore, we concentrate on mesh-based *surface deformation* techniques first. Mathematically, we can describe the deformation of a given triangle surface mesh $\mathcal{T}$ into a target mesh $\mathcal{T}'$ through a *deformation function* $\boldsymbol{d}\colon \mathcal{T} \to \mathbb{R}^3$ which defines a 3D displacement vector $\boldsymbol{\delta}$ for each vertex v with associated 3D coordinates $\boldsymbol{x}$ of the surface. After computing the deformation, we obtain the deformed mesh $\mathcal{T}'$ by evaluating the deformation function for each vertex of the mesh, i.e., $\mathcal{T}' \coloneqq \{\boldsymbol{x} + \boldsymbol{d}(\boldsymbol{x}) \mid \boldsymbol{x} \in \mathcal{T}\}$. Surface deformation techniques use the individual vertex positions as degrees of freedom, which allows for highly flexible control over the resulting deformation behavior. At the same time, this also makes them dependent on the mesh complexity as well as sensitive to mesh quality, i.e., defects in the connectivity or degenerate triangles. For a detailed overview of surface-based techniques we refer to the survey of Botsch and Sorkine (2008) and the textbook of Botsch et al. (2010). As a representative of surface deformations, we investigate the thin shell technique of Botsch and Kobbelt (2004b) in more detail in section 3.1.

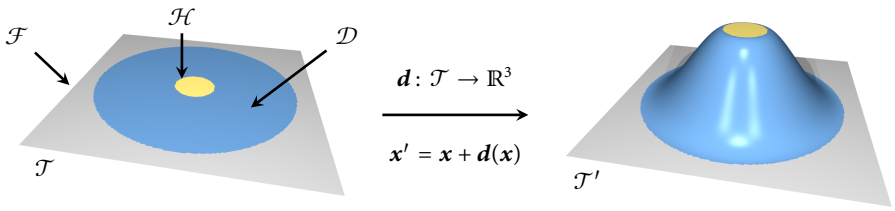


Figure 2.1: Surface deformation using a handle-based direct manipulation interface: The surface is divided into three distinct regions: The gray region $\mathcal{F}$ is kept fixed while the user manipulates the golden handle region $\mathcal{H}$, and the blue deformable region $\mathcal{D}$ is updated according to the deformation method, as shown on the right.

During surface deformation the user—being either a designer or an optimization algorithm—*directly manipulates* the vertices of the mesh. However, since the manipulation of individual vertex positions becomes rather tedious for complex models, we need a reasonably flexible, precise, and intuitive interface for specifying and controlling the desired deformation. To this end, we employ a *handle metaphor* (Kobbelt et al. 1998; Botsch and Kobbelt 2004b), where we distinguish three types of regions on the surface: The handle region $\mathcal{H}$ is directly displaced by the user. The fixed region $\mathcal{F}$ stays in place. The deformable region $\mathcal{D}$ is updated according to the deformation method while satisfying the prescribed displacement constraints given by $\mathcal{H}$ and $\mathcal{F}$. An example of this modeling metaphor is illustrated in figure 2.1, with the handle region $\mathcal{H}$ in gold, the fixed region $\mathcal{F}$ in gray, and the deformable region $\mathcal{D}$ in blue.

2.1.2 *Volume Mesh Deformation*

Within simulation-based design optimization the volumetric meshes required to actually perform physical simulations are a primary target geometry representation. In this context, the resulting mesh element quality after deformation becomes a critical factor: The deformation should preserve element quality as best as possible in order to maintain a mesh that is still feasible for obtaining reliable and accurate simulation results. The exact quality requirements heavily depend on the given application scenario as well as simulation and solver type. Consequently, such demands are not trivial to generalize. On a very basic level, however, the mesh should at least be free of *inverted elements*, i.e., the determinant of each element Jacobian should be positive (Knupp 2000).

The recent work of Staten et al. (2011) provides a comprehensive survey and comparison of volume mesh deformation approaches frequently used in the engineering community. The authors benchmark six volume deformation techniques based on a set of test cases with varying complexity and topology, including unstructured tetrahedral as well as both structured and unstructured hexahedral meshes. The methods covered in their benchmark can be roughly classified into three categories: Approaches based on generalized barycentric coordinates, mesh smoothing techniques, and mesh-based variational methods that minimize certain smoothness energies. Most of these techniques take updated boundary node positions as input and compute the new locations of interior volume mesh nodes from these boundary constraints.

Approaches based on *barycentric coordinates* determine the interior nodes as a linear (affine or convex) combination of the boundary nodes through a generalization of linear barycentric interpolation (Sukumar and Malsch 2006). Examples are Wachspress coordinates (Wachspress 1975), mean value coordinates (Floater et al. 2005), harmonic coordinates (Joshi et al. 2007), and maximum entropy coordinates (Sukumar 2004; Hormann and Sukumar 2008). See the recent work of Nieto and Susín (2013) for a comprehensive survey. The Simplex-linear method introduced along with the benchmarks of Staten et al. (2011), being a generalization of BMSWEEP (Staten et al. 1999), as well as its extension to natural neighbor interpolation (Sibson 1981), also belong to this category. While these approaches typically have rather simple geometric constructions and therefore are easy to implement and efficient to compute, the resulting deformations eventually are not smooth enough to reliably preserve mesh element quality.

Mesh smoothing methods adjust interior node locations in order to explicitly optimize element quality (Knupp 2000; Shontz and Vavasis 2003; Knupp 2008), where the Mesquite framework (Brewer et al. 2003) offers implementations based on mean ratio, untangling, and matrix condition number (Knupp 2000). In the context of mesh deformation, the updated boundary nodes act as fixed constraints while the optimization process determines the interior node locations. The mesh smoothing methods evaluated in Staten et al. (2011) worked well for small geometric changes, but were lacking robustness for larger deformations. In comparison, the LBWARP method (Shontz and Vavasis 2003), a weighted Laplacian smoothing method based on the log-barrier technique, gives considerably better results, but is computationally more complex.

Mesh-based variational methods compute smooth harmonic or biharmonic deformations by solving Laplacian or bi-Laplacian systems (Baker 2002; Helenbrook 2003), which is numerically more robust than most mesh smoothing techniques. The finite element-based FEMWARP technique (Baker 2002), which computes a harmonic deformation, was generalized from tetrahedra to hexahedra by Staten et al. (2011), and turned out to be the most successful approach in Staten's benchmarks. Note that harmonic coordinates of Joshi et al. (2007) are closely related to these approaches, since they are also derived by solving a Laplacian system.While the deformations produced by mesh-based variational methods tend to preserve element quality well, they have to be custom-tailored to each mesh element type, e.g., tetrahedral or hexahedral.

2.1.3 *Space Deformation*

In contrast to the mesh-based surface and volume deformations described above, so-called *space deformations* do not compute the deformation on the mesh directly, but in the embedding space Ω surrounding the object. The object is then deformed by deforming the embedding space around an object, thereby deforming the object implicitly. From a mathematical point of view a space deformation is a function $\boldsymbol{d} : \Omega \subset \mathbb{R}^3 \rightarrow \mathbb{R}^3$ that maps each point in the embedding space to a certain displacement $\boldsymbol{\delta}$. Assuming we are given such a space deformation function and an arbitrary discrete geometric model $\mathcal{M}$ with point coordinates $\boldsymbol{x} \in \mathbb{R}^3$. We can then transform $\mathcal{M}$ into the deformed model $\mathcal{M}'$ by computing updated point locations $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{d}(\boldsymbol{x})$ for each original point $\boldsymbol{x} \in \mathcal{M}$. In figure 2.2, we illustrate space deformations of different geometry representations of the Fandisk model, including an unstructured point set, triangle and quad surface meshes, a tetrahedral volume mesh, as well as a hexahedral voxel representation.

Surveys on space deformation techniques have been presented by Bechmann (1994) as well as Gain and Bechmann (2008). While the former concentrates on building a mathematical formalism for the different methods, the latter focuses on the interactive manipulation of a model by a designer. The work of Angelidis and Singh (2006) also provides a survey of space deformation techniques, but with a particular focus on different modeling operations and advanced user interfaces. The textbook of Botsch et al. (2010) as well as the tutorial notes of Sorkine and Botsch (2009) also contain an overview of space deformation methods.

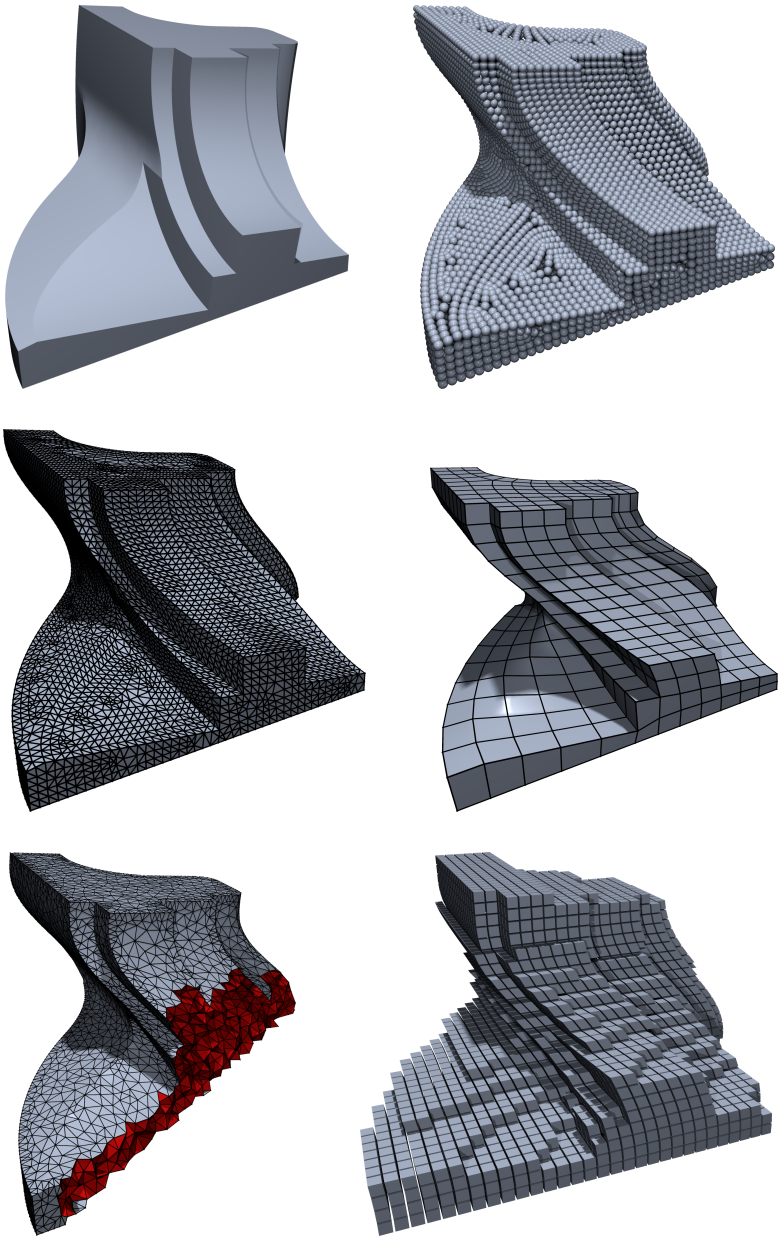


Figure 2.2: Space deformation of different representations of the Fandisk model. Top left to bottom right: Original model, deformed point set, triangle mesh, quad mesh, tetrahedral mesh, and voxel model.

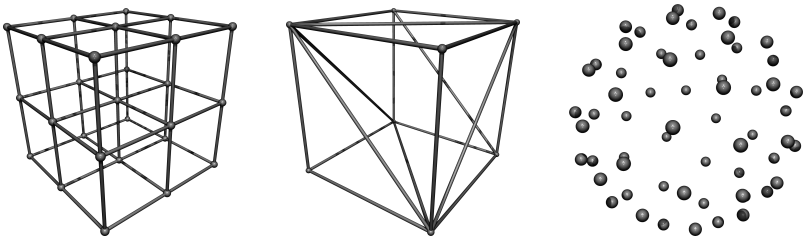


Figure 2.3: Control structures: Control grid, surface mesh, point sampling.

We can classify space deformation techniques by how they construct the deformation function $\boldsymbol{d}$. Typically, a control structure such as a volumetric lattice or a set of points is blended with some form of smooth basis functions, see figure 2.3 for example control structures. Classical spline-based free-from deformation techniques employ a *volumetric* structure such as a regular or adaptive grid (Sederberg and Parry 1986; MacCracken and Joy 1996), or a polyhedral space decomposition such as a tetrahedral mesh (Moccozet and Thalmann 1997). Cage-based deformations based on barycentric interpolation use a triangular *surface* mesh enclosing the model (Ju et al. 2005). Skinning methods based on *skeletal* structures (Jacobson et al. 2014) are widely used in character animation. Finally, point- or kernel-based techniques (Botsch and Kobbelt 2005; McDonnell and Qin 2007) allow to freely position control points in space. We cover different types of space deformations and their variations in more detail in chapter 3. Note, however, that we deliberately omit skeleton-based methods, since these are more suitable for specialized applications such as character animation and not easily applicable within a general design optimization scenario.

In contrast to mesh-based methods, space deformation techniques typically use the positions of the individual control points $\boldsymbol{c}$ constituting the control structure as DoFs to steer the deformation. This also implies that space deformations usually only provide *indirect manipulation* of the target design through manipulation of the respective control structure. This type of interface has two significant drawbacks: First, since the influence on the object is only indirect, it is difficult to exactly prescribe required displacement constraints on the object. Second, with increasing resolution of the control structure its setup and manipulation becomes increasingly difficult and tedious for the designer. Therefore, we investigate the combination of space deformations and direct manipulation interfaces in chapter 3.

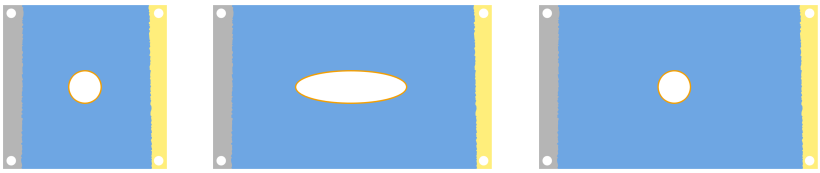


Figure 2.4: Constrained deformation example (from left to right): Setup, deformation without and with circularity constraint (marked in orange).

2.1.4 *Constrained Deformation*

A rather recent trend in the development of shape deformation methods is the integration of additional constraints. Such techniques are particularly interesting for design optimization, since the maintenance of characteristic geometric features and the adherence to critical production limitations are a major challenge for the successful application of a design optimization process.

Early examples of constrained deformation methods are the feature-preserving surface deformation technique of Masuda and colleagues (Masuda et al. 2007), as well as the iWires system (Gal et al. 2009) for the deformation of man-made objects. More recently, the latter approach was generalized to component-wise controllers (Zheng et al. 2011), and the work of Habbecke and Kobbelt (2012) presents an efficient technique for the linear analysis of non-linear constraints in geometric modeling systems. Yang et al. (2011) present a method for shape space exploration of constrained meshes. Deng et al. (2013) extended this approach towards exploration of local modifications of constrained meshes. Similar in spirit of the technique of Masuda et al. (2007) is the constrained surface deformation technique described by Fu et al. (2012). More recently, Tang et al. (2014) introduced a technique for form-finding in polyhedral meshes subject to user-prescribed constraints such as, e.g., planarity of mesh faces. For a comprehensive overview of geometry processing techniques incorporating constraints we refer to the recent survey of Mitra et al. (2013).

However, all of the above methods share a common limitation: They are inherently surface-based and therefore their applicability to design optimization tasks is rather limited. A notable exception in this regard is the projection-based constraint processing technique of Bouaziz et al. (2012) and its later extension (Deuss et al. 2015), since it allows to prescribe general constraints on arbitrary geometric data sets. We adapt and extend their approach for constraint preservation into our

space deformation framework in chapter 8, thereby fostering the creation of more feasible design variations during design optimization.

2.2 DESIGN OPTIMIZATION

As already outlined in the introduction, the general aim of design optimization is the exploration of design variations that exhibit some form of improved properties such as physical performance or aesthetic appeal. The success and effectiveness of a design optimization process depends on the interplay of three core components: An *optimization algorithm*, effective means to create *design variations*, and the formulation of a suitable *cost* or *fitness function* capturing the property to be optimized. In the following, we provide a brief overview of each of these components. However, for a more comprehensive introduction to design optimization, we refer to the textbooks of Delfour and Zolésio (2011), Held (2009), and Mohammadi and Pironneau (2010).

As for the optimization algorithm a wide variety of methods is used in practice. Gradient-based techniques (Nocedal and Wright 2006; Mohammadi and Pironneau 2010) aim to directly minimize the given cost function. While such approaches are highly effective in terms of computational cost and convergence rate, they easily get stuck in local minima. In contrast, stochastic global optimization methods such as evolutionary algorithms (Bäck 1996) aim at finding a global optimum of the fitness function, albeit at the cost of drastically increased computation time and slower convergence rates. In chapter 5, we describe the application of evolutionary algorithms for the solution of design optimization problems in more detail.

The formulation of the cost function heavily depends on the target property being optimized and includes characteristics such as aerodynamic performance or mechanical stability. In particular, the simultaneous optimization of multiple and even contradicting cost functions has gained increasing attention and is known as *multidisciplinary design optimization*. Besides pure performance characteristics the cost function typically takes into account additional *constraints* on the resulting shape, such as minimum or maximum widths or heights of certain components, which are necessary to meet production or usability limitations. As an example, only optimizing the aerodynamic drag of a car body would result in a raindrop-like shape that is not only difficult to produce but also rather impracticable and unattractive for the customer.

As for the creation of design variations, we concentrate on methods that operate on a discrete geometry representation such as a triangle mesh. Early approaches simply used the individual 3D point coordinates as degrees of freedom for the optimization algorithm. However, with increasing model complexity the number of points in the design—and therefore the number of degrees of freedom for the optimization—becomes too large to be feasible. Therefore, much attention has been given to so-called *representation* or *parametrization* methods representing the target design through a drastically smaller set of parameters which are then modified by the optimization algorithm.

A survey of shape parametrization techniques in the context of design optimization was presented by Samareh (2001). In recent years, the application of deformation methods from computer graphics has gained increased attention (Samareh 2004; Menzel et al. 2005, 2006; Menzel and Sendhoff 2008). In particular, free-form deformation now is a well-established tool and is available in commercial design optimization software packages such as DEP Morpher (Detroit Engineered Products 2015) or Sculptor (Optimal Solutions 2015). Yamazaki et al. (2010) investigated direct manipulation methods for geometry parametrization in the context of airfoil optimization. Similarly, G. R. Anderson et al. (2012) investigate lattice-based deformations with additional constraints for airfoil optimization. Finally, shape deformations based on radial basis functions (RBFs) have gained increasing attention, especially in the context of airfoil optimization (Boer et al. 2007; Jakobsson and Amoignon 2007; Michler 2011). More recently, RBF deformations also became available as an extension for ANSYS Fluent (Biancolini 2015).

PART I

SHAPE DEFORMATION FOR DESIGN OPTIMIZATION

This part investigates the first research question laid out for this thesis: *What is a good shape deformation technique for design optimization*? In order to provide a well-founded answer, we begin with an introduction of several state-of-the-art deformation methods (chapter 3). We will then analyze the individual strengths and weaknesses of these methods in a series of synthetic benchmarks (chapter 4), as well as an application-oriented benchmark based on the evolutionary design optimization of a passenger car (chapter 5).

CHAPTER 3

SHAPE DEFORMATION METHODS

This chapter introduces state-of-the-art deformation techniques for their use in design optimization. In order to obtain a diverse overview, we investigate a wide variety of methods, including a mesh-based surface deformation technique as well as multiple space deformation techniques employing different types of control structures and blending functions: We first introduce thin-shell deformation as a representative of a physically-inspired surface deformation method. We then cover different variants of classical free-form deformation, since these techniques are the de-facto standard in commercial modeling packages as well as industry-strength design optimization tools such as DEP Morpher (Detroit Engineered Products 2015) or Sculptor (Optimal Solutions 2015). As a generalization of free-form deformation employing a more flexible control structure and different blending functions, we include cage-based deformations based on mean value coordinates. Finally, we introduce deformations based on radial basis functions as a representative of a point- or kernel-based technique.

3.1 THIN SHELL DEFORMATION

Common target prototypes in design optimization represent sheet metal surfaces such as car bodies, aircraft wings, or ship hulls. Therefore, we begin our investigation with a deformation technique particularly suitable for deforming structures of this type. The *thin shell deformation* method introduced by Botsch and Kobbelt (2004b) allows for the flexible modeling of surface deformations based on physically-inspired energies, e.g., to model the resistance of the design towards stretching and bending forces. As an illustration of the method's modeling flexibility we show different deformation examples and the effect of varying sensitivity towards stretching and bending in figure 3.1.

In the following, we introduce the basic ideas and theory behind the thin shell deformation method. However, for a more comprehensive treatment we refer to the original publication (Botsch and Kobbelt 2004b), the survey of Botsch and

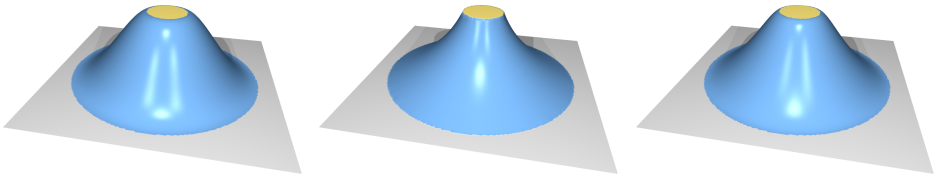


Figure 3.1: Shell-based deformation of a plane. Left: Pure bending minimization ($\gamma_b = 1, \gamma_s = 0$). Center: Pure stretching ($\gamma_b = 0, \gamma_s = 1$). Right: A mixture of bending and stretching ($\gamma_b = 10, \gamma_s = 1$).

Sorkine (2008), as well as the textbook of Botsch et al. (2010). The basic idea behind this method is that the surface deforms like a thin shell, i.e., similar to a thin plate of metal. To achieve this kind of behavior we formulate a suitable energy functional measuring resistance towards stretching and bending and construct a deformation function $\boldsymbol{d}\colon \mathcal{T} \to \mathbb{R}^3$ that minimizes this energy while satisfying the user-defined modeling constraints of our handle-based modeling interface, see figure 3.1 for an illustration.

As described in Botsch et al. (2010), we can measure the intrinsic geometric properties such as lengths, areas, and curvatures of a smooth parametric surface $\mathcal{S}$ in terms of the first and second fundamental forms $\mathbf{I}(u,v)$ and $\mathbf{II}(u,v)$. Consequently, we can formulate an elastic thin shell deformation energy (Terzopoulos et al. 1987) in terms of *differences* of the first and second fundamental forms of the original and deformed surfaces $\mathcal{S}$ and $\mathcal{S}'$:

$$\begin{aligned} E[\mathcal{S}'] \;=\; & \iint_\Omega \gamma_s \left\| \mathbf{I}'(u,v) - \mathbf{I}(u,v) \right\|_F^2 \\ & + \gamma_b \left\| \mathbf{II}'(u,v) - \mathbf{II}(u,v) \right\|_F^2 \mathrm{d}u\, \mathrm{d}v. \end{aligned} \tag{3.1}$$

Within this formulation the parameters γ_s and γ_b determine the sensitivity towards stretching and bending. Since equation (3.1) is a nonlinear energy its minimization is computationally expensive and typically prohibitive for an interactive modeling scenario. However, we can simplify equation (3.1) by substituting the fundamental forms with partial derivatives of the deformation function $\boldsymbol{d}$ (Celniker and Gossard 1991; Welch and Witkin 1992). This leads to the simplified shell energy

$$\begin{aligned} E[\boldsymbol{d}] \;=\; & \iint_\Omega \gamma_s \left(\|\boldsymbol{d}_u(u,v)\|^2 + \|\boldsymbol{d}_v(u,v)\|^2 \right) \\ & + \gamma_b \left(\|\boldsymbol{d}_{uu}(u,v)\|^2 + 2\,\|\boldsymbol{d}_{uv}(u,v)\|^2 + \|\boldsymbol{d}_{vv}(u,v)\|^2 \right) \mathrm{d}u\, \mathrm{d}v, \end{aligned} \tag{3.2}$$

where we employ the convention that $\boldsymbol{d}_u = \frac{\partial}{\partial u}\boldsymbol{d}$ denotes the first partial derivative with respect to u and $\boldsymbol{d}_{uv} = \frac{\partial^2}{\partial u \partial v}\boldsymbol{d}$ denotes second order mixed partial derivatives with respect to u and v . Applying variational calculus yields the corresponding *Euler-Lagrange* equation that minimizes equation (3.2):

$$-\gamma_s \Delta \boldsymbol{d} + \gamma_b \Delta^2 \boldsymbol{d} \;=\; \mathbf{0}. \tag{3.3}$$

When deforming triangle meshes $\mathcal{T}$ we can discretize equation (3.3) using the discrete cotangent Laplacian (Meyer et al. 2003) and can transform equation (3.3) into a linear system of equations with the displacements $\boldsymbol{\delta}_i$ of the deformable vertices being the unknowns. Since we know the prescribed displacements of the handle and fixed vertices $\boldsymbol{h}_i \in \mathcal{H} \cup \mathcal{F}$ in advance, we can move them to the right-hand side $\boldsymbol{b}$ of the system. Let $\boldsymbol{L}$ and $\boldsymbol{L}^2$ be the (bi-)Laplacian matrices containing the required cotangent weights. The resulting linear system then is

$$\left[-\gamma_s \boldsymbol{L} + \gamma_b \boldsymbol{L}^2\right] \underbrace{\begin{bmatrix} \boldsymbol{\delta}_1^{\mathrm{T}} \\ \vdots \\ \boldsymbol{\delta}_n^{\mathrm{T}} \end{bmatrix}}_{\boldsymbol{D}} \;=\; \underbrace{\begin{bmatrix} \boldsymbol{b}_1^{\mathrm{T}} \\ \vdots \\ \boldsymbol{b}_n^{\mathrm{T}} \end{bmatrix}}_{\boldsymbol{B}}, \tag{3.4}$$

where $\boldsymbol{D}$ and $\boldsymbol{B}$ are $n \times 3$ matrices with n being the number of deformable vertices. Consequently, we have to solve the system three times for the different right-hand sides.

In our implementation of thin shell deformation, we use the sparse Cholesky factorization of CHOLMOD (Chen et al. 2008) to solve the linear system of equation (3.4). In order to increase interactivity, we reduce the fixed and handle vertices to those being in a 2-ring neighborhood of a deformable vertex since other fixed or handle vertices do not influence the solution (Botsch and Kobbelt 2004b). In an interactive deformation system the boundary constraints—the right-hand side of equation (3.4)—change whenever the user manipulates the constraints. Therefore, we can pre-factorize the matrix only once, but we still need to solve for the different right-hand sides in each frame. However, we can circumvent this by restricting the handle deformations to affine transformations only and precomputing a set of special basis functions, see Botsch and Kobbelt (2004b) for details. This way we only have to evaluate the basis functions for each deformation step, thereby drastically increasing the interactivity of the system.

While thin shell deformation provides flexible control over the deformation behavior, it is also sensitive to the quality and number of triangles in the mesh.

Furthermore, due to its surface-based nature, the method is not capable of deforming a volumetric simulation grid along with the surface—a capability particularly important in design optimization. In the following, we therefore concentrate on space deformation methods being independent of the underlying geometry representation.

3.2 FREE-FORM DEFORMATION

A well-established deformation technique that has been widely used in both academia and industry is *free-form deformation* (FFD). The basic idea of the technique is to embed the target object into a volumetric control lattice and to deform it based on trivariate tensor-product Bézier or B-spline basis functions. Since it is widely used for both shape optimization in general (Manzoni et al. 2012; Samareh 2004) as well as simulation-based design optimization in particular (Menzel et al. 2005, 2006; Menzel and Sendhoff 2008; Olhofer et al. 2009; Sieger et al. 2012), this technique constitutes the starting point of our investigation of space deformation methods. Before describing the method in detail, we first review variations and extensions of the technique.

Free-form deformation using Bézier basis functions was originally introduced by Sederberg and Parry (1986). Since the global influence of Bézier basis functions—and hence of the deformation—limits the applicability of the technique, Griessmair and Purgathofer (1989) extended the approach to perform local deformations based on B-spline basis functions. An extension to more flexible control lattices, in particular cylindrical ones, has been proposed by Coquillart (1990). This approach was later extended to control lattices of arbitrary topology (MacCracken and Joy 1996). Free-form deformations using more flexible non-uniform rational B-splines are described by Lamousin and Waggenspack (1994). A highly flexible but computationally involved variant of FFD based on a 3D-Delaunay triangulation, its Voronoi dual, and Sibson coordinates (Sibson 1980) has been presented by Moccozet and Thalmann (1997). More recently, a variant of FFD using T-splines (Sederberg et al. 2003) as basis functions—thereby allowing for local refinement of the control lattice—has been presented by Song and Yang (2005).

The basic idea of classical FFD is based on embedding the object to be deformed in a parallelepiped lattice and deforming it using trivariate tensor-product Bézier or B-spline functions. The procedure to perform free-form deformation can be divided into several steps. First, a control lattice has to be generated and adapted

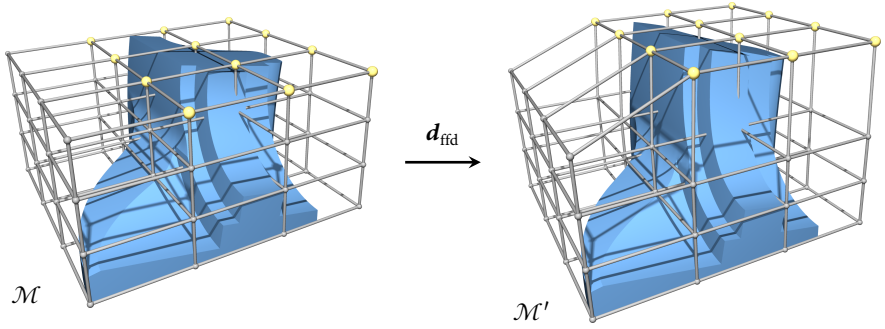


Figure 3.2: Free-form deformation applied to the Fandisk model. Left: The original model $\mathcal{M}$ is embedded in a regular lattice of control points (grey). Right: After moving the selected control points (golden), we compute the updated object points $\boldsymbol{x}'$ by evaluating the FFD space deformation function $\boldsymbol{d}_{\text{ffd}}$ for the local coordinates $\boldsymbol{u}$ of the point $\boldsymbol{x}$.

to the deformation scenario at hand. Second, the local coordinates with respect to the control lattice have to be computed. Third, the user manipulates the control grid. Fourth, the system computes the updated object points.

The computation of local coordinates for each point $\boldsymbol{x} \in \mathcal{M}$ to be deformed is a central step. Afterwards, each of these points can be expressed as a linear combination of lattice control points $\boldsymbol{c}_{ijk}$ and basis functions φ_i:

$$\boldsymbol{x} \;=\; \sum_{i=0}^{l} \sum_{j=0}^{m} \sum_{k=0}^{n} \boldsymbol{c}_{ijk} \varphi_i(u_1(\boldsymbol{x})) \varphi_j(u_2(\boldsymbol{x})) \varphi_k(u_3(\boldsymbol{x})),$$

where $(u_1(\boldsymbol{x}), u_2(\boldsymbol{x}), u_3(\boldsymbol{x}))$ are the local coordinates of $\boldsymbol{x}$ with respect to the control lattice, and l, m, n are the numbers of control points in each respective direction. For the sake of simplicity, we define

$$\boldsymbol{u}(\boldsymbol{x}) \coloneqq (u_1(\boldsymbol{x}), u_2(\boldsymbol{x}), u_3(\boldsymbol{x})), \quad \varphi_p(\boldsymbol{u}(\boldsymbol{x})) \coloneqq \varphi_i(u_1(\boldsymbol{x})) \varphi_j(u_2(\boldsymbol{x})) \varphi_k(u_3(\boldsymbol{x})),$$

as well as

$$\boldsymbol{\delta}_p \coloneqq \boldsymbol{\delta}_{ijk} = \boldsymbol{c}'_{ijk} - \boldsymbol{c}_{ijk},$$

where $\boldsymbol{c}'_{ijk}$ denotes an updated control point location. We then define the FFD space deformation function as

$$\boldsymbol{d}_{\text{ffd}}(\boldsymbol{x}) \;=\; \sum_{p} \boldsymbol{\delta}_p \varphi_p(\boldsymbol{u}(\boldsymbol{x})).$$

Finally, the deformation is performed by moving the control points and computing the updated object point locations, as we illustrate in figure 3.2.

In our implementation of FFD we use cubic B-splines with a uniform knot vector. While this type of basis functions requires an iterative root-finding technique such as a Newton method (Press et al. 2007) for computing the local coordinates, the important advantage is the capability to perform deformations with local influence only, e.g., in order to restrict the deformation to a given region of interest in a design optimization task. Since the local coordinate computation is independent for each object point it is trivial to parallelize.

3.3 DIRECT MANIPULATION FFD

In an interactive modeling system the manipulation of control points to perform a deformation becomes a tedious task—especially when using a complex control lattice with a large number of control points. A more flexible and intuitive interface for controlling a deformation is offered by *direct manipulation* approaches, as introduced for FFD by Hsu et al. (1992). Within direct manipulation FFD (referred to as DM-FFD for short) the user *directly* moves the object points instead of moving control points. The modeling system then computes control point displacements so that the desired object point positions are matched as precisely as possible. We show an example deformation of the Fandisk model using DM-FFD in figure 3.3, where we use the handle-based interface of figure 2.1 to control the deformation.

Direct manipulation interfaces are not only beneficial within an interactive modeling scenario, they can also be used effectively within simulation-based design optimization, as has been shown for direct manipulation FFD by Menzel et al. (2006). Due to the more direct influence of the parameters determined during optimization on the design, using such an interface can result in drastically faster convergence of the optimization process. Furthermore, in contrast to classical FFD, the ability to choose an arbitrary object point or handle region for optimization offers increased flexibility.

Within a direct manipulation interface the user—be it an engineer or an optimization algorithm—prescribes a set of m displacement constraints $\{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m\}$ at handle points $\boldsymbol{h}_i \in \mathcal{H} \cup \mathcal{F}$, for which the deformation function has to satisfy certain displacement values $\boldsymbol{d}(\boldsymbol{h}_i) = \boldsymbol{h}_i' - \boldsymbol{h}_i = \boldsymbol{b}_i$. We then compute the displace-

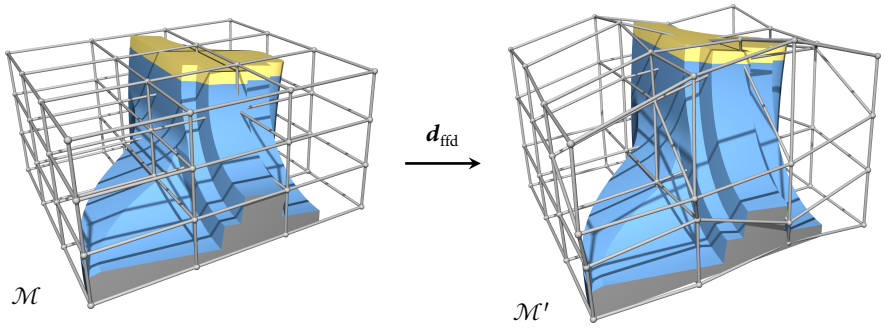


Figure 3.3: Direct manipulation FFD of the Fandisk model using a handle-based interface. We lift the top handle while keeping the bottom fixed. The system automatically computes the control point displacements necessary to satisfy the prescribed displacement constraints.

ments $\{\boldsymbol{\delta}_1, \ldots, \boldsymbol{\delta}_n\}$ of the n control points satisfying the prescribed displacements by solving the linear system

$$\underbrace{\begin{bmatrix} \varphi_1(\boldsymbol{u}(\boldsymbol{h}_1)) & \ldots & \varphi_n(\boldsymbol{u}(\boldsymbol{h}_1)) \\ \vdots & \ddots & \vdots \\ \varphi_1(\boldsymbol{u}(\boldsymbol{h}_m)) & \ldots & \varphi_n(\boldsymbol{u}(\boldsymbol{h}_m)) \end{bmatrix}}_{\boldsymbol{\Phi}} \underbrace{\begin{bmatrix} \boldsymbol{\delta}_1^{\mathrm{T}} \\ \vdots \\ \boldsymbol{\delta}_n^{\mathrm{T}} \end{bmatrix}}_{\boldsymbol{D}} = \underbrace{\begin{bmatrix} \boldsymbol{b}_1^{\mathrm{T}} \\ \vdots \\ \boldsymbol{b}_m^{\mathrm{T}} \end{bmatrix}}_{\boldsymbol{B}}. \tag{3.5}$$

Since the linear system in equation (3.5) can be over-determined as well as under-determined, it is typically solved by computing the *pseudo-inverse* $\boldsymbol{\Phi}^+$ of the basis function matrix $\boldsymbol{\Phi}$. This is typically done by performing a *singular value decomposition* (SVD) (Hsu et al. 1992; Golub and Van Loan 2013) so that $\boldsymbol{\Phi} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{T}}$, where $\boldsymbol{U}$ is a $m \times m$ orthogonal matrix, $\boldsymbol{\Sigma}$ is a $m \times n$ diagonal matrix containing the singular values of $\boldsymbol{\Phi}$, and $\boldsymbol{V}^{\mathrm{T}}$ is a $n \times n$ orthogonal matrix. The pseudo-inverse of $\boldsymbol{\Phi}$ then is $\boldsymbol{\Phi}^+ = \boldsymbol{V}\boldsymbol{\Sigma}^+\boldsymbol{U}^{\mathrm{T}}$, where we can compute the pseudo-inverse of the diagonal matrix $\boldsymbol{\Sigma}$ as

$$(\boldsymbol{\Sigma}^+)_{ij} = \begin{cases} \frac{1}{\sigma_i}, & \text{if } i = j \wedge \sigma_i \neq 0, \\ 0, & \text{otherwise,} \end{cases} \tag{3.6}$$

where σ_i is the i-th singular value of $\boldsymbol{\Phi}$. We note that for values close to zero σ_i has to be clamped in order to prevent numerical instabilities. Once $\boldsymbol{\Phi}^+$ has been computed the control point displacements can be computed by

$$\boldsymbol{D} = \boldsymbol{\Phi}^+\boldsymbol{B}, \tag{3.7}$$

where $\boldsymbol{D}$ is the matrix of control point displacements and $\boldsymbol{B}$ is the matrix of constraint displacements. However, solving for $\boldsymbol{D}$ using the pseudo-inverse has its drawbacks. If the system is under-determined, a *least-norm* solution is found, i.e., the amount of movement of the control points $\|\boldsymbol{\delta}_i\|$ is minimized. If the system is overdetermined, a *least-squares* solution is found, i.e., the error in satisfying the specified constraints is minimized. This means that depending on the resolution of the control lattice the system might not be able to satisfy the constraints specified by the user in an exact manner. In both cases, however, the solution does not necessarily result in a physically plausible deformation.

3.4 CAGE-BASED DEFORMATION

An extension of the concept of free-form deformation are *cage-based deformation* methods. Instead of embedding the model within a regular volumetric control grid, it is enclosed by an irregular triangular surface mesh called *control cage*. Typically, the cage is much coarser than the mesh of the original model. In comparison to the volumetric control grids used in FFD, control cages allow for a more tight matching between the original model and the controlling structure. The increased flexibility of cage deformations stems from the use of more flexible *generalized barycentric coordinates* instead of Bézier or B-spline basis functions as used in free-form deformation.

Cage-based deformation was originally introduced by Ju et al. (2005). The authors derive and employ a generalization of mean value coordinates (Floater et al. 2005) to triangular surface meshes. Following their initial publication, cage-based deformations have become widely used especially in character animation, and several variants such as the harmonic coordinates of Joshi et al. (2007), Green coordinates (Lipman et al. 2008), or the recent work of Zhang et al. (2014) introducing local barycentric coordinates have been proposed. These methods also found their way into professional modeling software packages, as exemplified by the integration of harmonic coordinates into Blender (2015). For a comprehensive treatment and analysis of these methods, we refer to the recent survey of Nieto and Susín (2013). For our investigation of cage-based deformations, we follow the original approach of Ju et al. (2005). We illustrate cage-based deformation of the Fandisk model in figure 3.4.

The actual process for performing cage-based deformation of a model is very similar to the FFD process. First, the model $\mathcal{M}$ is embedded in a control cage $\mathcal{R}$.

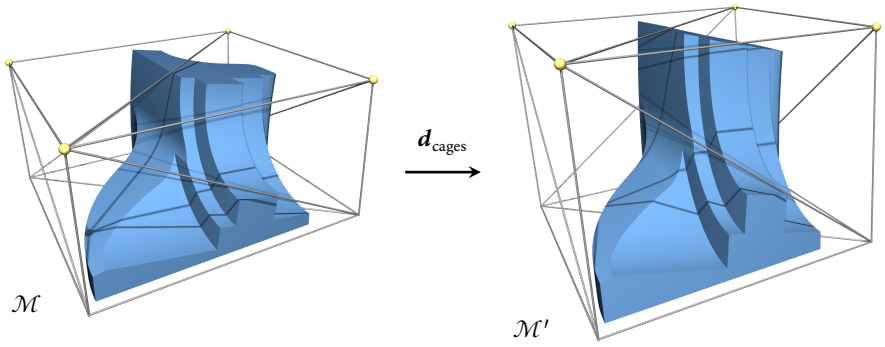


Figure 3.4: Cage-based deformation of the Fandisk model. Left: Initial setup. Right: The top vertices (golden) of the bounding cage are moved and the mesh vertices are updated accordingly using mean value coordinates. For illustrative purposes, we employ a very simple cage based on the object bounding box.

Then each point $\boldsymbol{x} \in \mathcal{M}$ can be represented by a weighted linear combination of the cage vertices $\boldsymbol{c}_j \in \mathcal{R}$:

$$\boldsymbol{x} = \sum_{j=1}^{n} \boldsymbol{c}_j \varphi_j(\boldsymbol{x}),$$

where the $\varphi_j(\boldsymbol{x})$ are *generalized barycentric coordinates*. When we move the cage vertices to updated positions $\boldsymbol{c}_j'$ we can define the according space deformation function as

$$\boldsymbol{d}_{\text{cages}}(\boldsymbol{x}) = \sum_{j=1}^{n} \boldsymbol{\delta}_j \varphi_j(\boldsymbol{x}) \ ,$$

where $\boldsymbol{\delta}_j = \boldsymbol{c}_j' - \boldsymbol{c}_j$ are the cage vertex displacements. The actual computation of mean value coordinates is purely based on simple geometric constructions, and Ju et al. (2005) describe the algorithm in detail, including complete pseudo-code. Therefore, we do not reproduce a description here.

Similar to FFD, the manipulation of cage vertices becomes tedious for high resolution cages and/or complex models. Therefore, we employ a handle-based direct manipulation interface identical to section 3.3, see figure 3.5 for an illustration. Since both the mathematical derivations to compute the cage vertex displacements as well as the actual implementation using SVD are essentially identical to the ones for direct manipulation FFD, we omit a detailed presentation at this point.

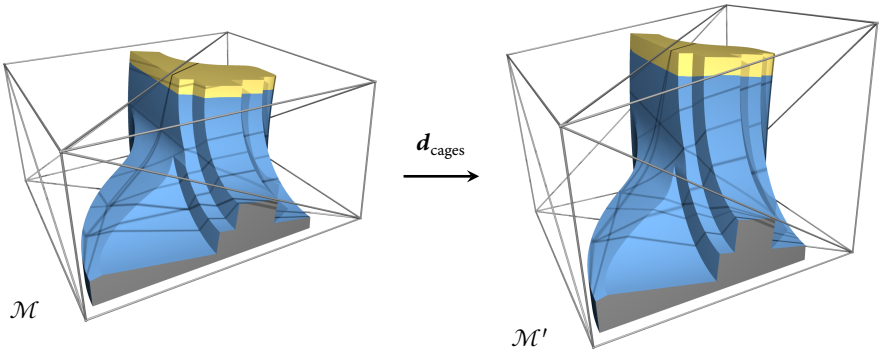


Figure 3.5: Cage-based deformation of the Fandisk model using a handle-based direct manipulation interface. Left: Initial setup. Right: Deformed model.

3.4.1 *Cage Generation*

In contrast to classical FFD where the control grid is typically a regular subdivision of the object bounding box, the construction of a coarse *bounding cage* is a non-trivial task of its own. In the original publications the cage is typically created manually, which becomes tedious and time-consuming for complex geometries. In this section, we briefly derive general criteria for an automatic *cage generation* procedure as well as requirements for the resulting bounding cage. We then describe different cage generation methods of increasing output quality but also implementation complexity.

Even though it is not yet fully understood what properties a *good* cage should exhibit, it is still possible to formulate a set of general criteria. First of all, the cage should fully enclose the object and match its topology as good as possible. The complexity of the cage, i.e., the number of cage vertices should be smaller than the number of object vertices, while being controllable by the user. Given a target number of cage vertices, the shape of the triangles in the cage should be as good as possible, i.e., close to equilateral triangles. This is desirable since the shape of the triangles also influences the shape of the deformation (see also section 4.4). Depending on the particular deformation scenario, symmetry of the cage can be an important characteristic as well, e.g., in case the target model consists of highly symmetric components.

A rather straightforward and simple way to generate a bounding cage is to compute the *bounding box* of the model, slightly increase the size of the box to

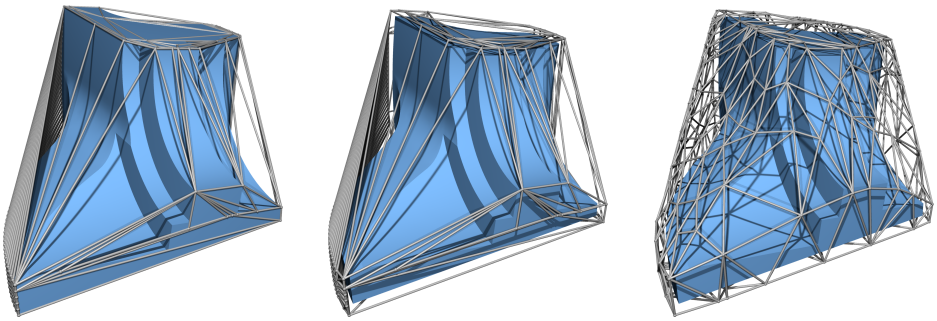


Figure 3.6: Cage generation based on convex hull computation (left), offsetting (center), and feature-preserving remeshing (right).

make sure the model is fully contained, and to create a minimal triangulation. See figures 3.4 and 3.5 where we use such a cage for the sake of simplicity. In case we require a higher resolution cage, we can simply employ a *remeshing* (Botsch and Kobbelt 2004a) or *subdivision* (Loop 1987) algorithm. However, a cage generated this way will always be only a crude approximation of the original shape. An alternative approach is to use the *convex hull* (de Berg et al. 2008) of the object as an initial cage, see figure 3.6. While this approach improves the matching between cage and object, it is still problematic in case of large concave regions in the model. Even though we could achieve a more tight matching between the cage and the model by employing *non-rigid registration* techniques (Bouaziz et al. 2014), a cage generated through convex hull computation is topologically equivalent to a sphere and therefore not capable of reproducing the topology of more complex target models.

In the following, we introduce a cage generation procedure based on *offset surface* computation and *surface reconstruction* that overcomes the limitations of the previous simplistic approaches and that is able to compute tightly matching bounding cages—both in terms of geometry and topology—in a semi-automatic manner. The individual steps of the procedure are:

1. *Sampling*: Since for low resolution models the sampling density might be insufficient for faithful surface reconstruction, we optionally perform Loop subdivision (Loop 1987) to increase sampling density.

2. *Offsetting*: In order to obtain a cage enclosing the original mesh we offset each vertex towards its normal direction. Depending on the exact shape of the model and the size of the offset this step might cause self-intersections.

3. *Reconstruction*: In order to get rid of the self-intersections, we reconstruct the surface from its points and normals by using Poisson surface reconstruction (Kazhdan et al. 2006).

4. *Remeshing*: In case a high quality cage is desired, the resulting cage is remeshed using either isotropic or adaptive remeshing based on edge splits, collapses, and flips (Botsch and Kobbelt 2004a).

5. *Decimation*: In case a coarse cage is desired, we decimate the cage using halfedge-collapses and error quadrics (Garland and Heckbert 1997).

The individual processing stages of our cage generation process are illustrated in figure 3.7. Note that the sampling, remeshing, and decimation steps are optional. Their usage depends on the particular input model and the user requirements to be satisfied by the cage. While this method is sufficiently flexible to generate bounding cages for a wide variety of input geometries, the design and implementation of an efficient, robust and *fully* automatic cage generation method is a challenging task for future work. The volumetric cage generation approaches described by Xian et al. (2009, 2012) or the robust offset surface computation method of Pavić and Kobbelt (2008) constitute promising starting points towards this goal. The recent work of Sacht et al. (2015) introduces a powerful cage generation method, although the technique has convergence issues in some cases. Still, the need generate and maintain a coarse bounding cage remains a serious obstacle in adopting cage-based deformations for design optimization. In the next section, we introduce a more flexible technique that does not suffer from this burden.

3.5 RBF DEFORMATION

In this section, we present a shape deformation method based on scattered data interpolation techniques that improves upon the previously presented approaches in two significant aspects: First of all, the method is point-based in nature, i.e., the technique allows to freely position the desired degrees of freedom at arbitrary locations in space, thereby freeing the user of the need to generate and maintain a

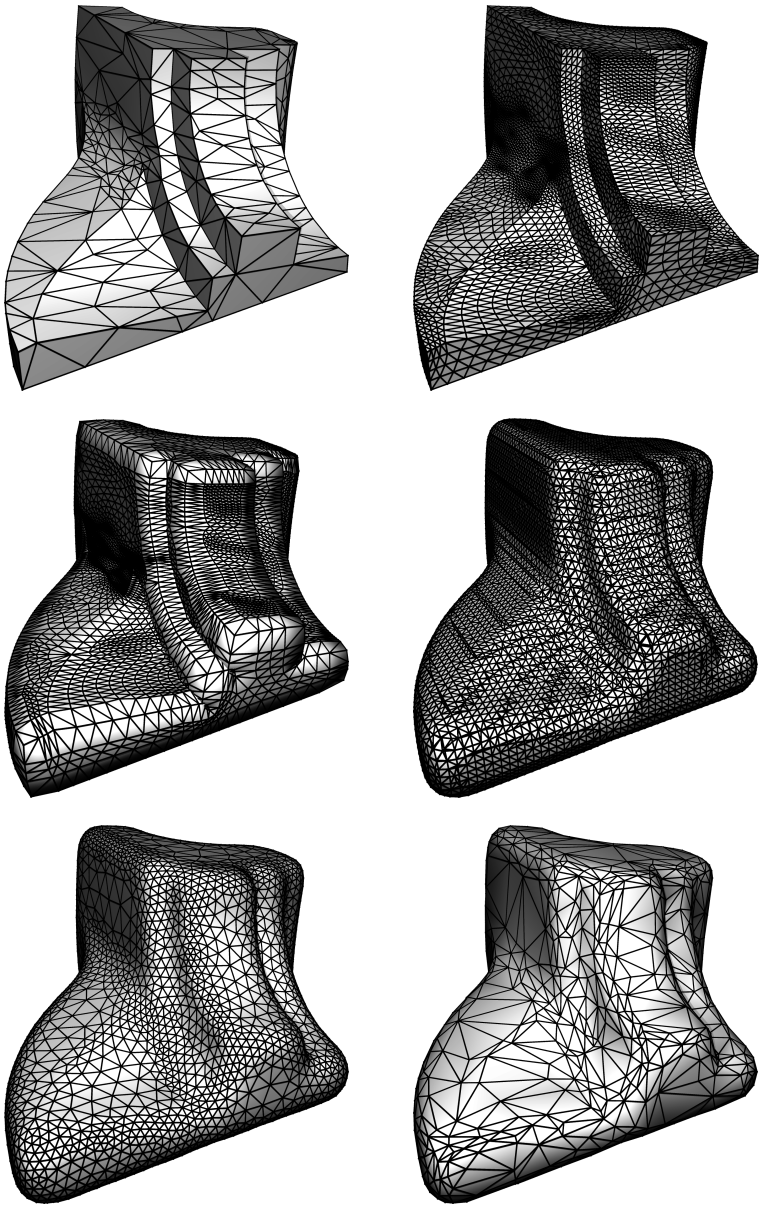


Figure 3.7: Processing stages during cage generation. Top row: Low resolution mesh and subdivision. Middle row: Offset mesh and surface reconstruction. Bottom row: Adaptive remeshing and decimation.

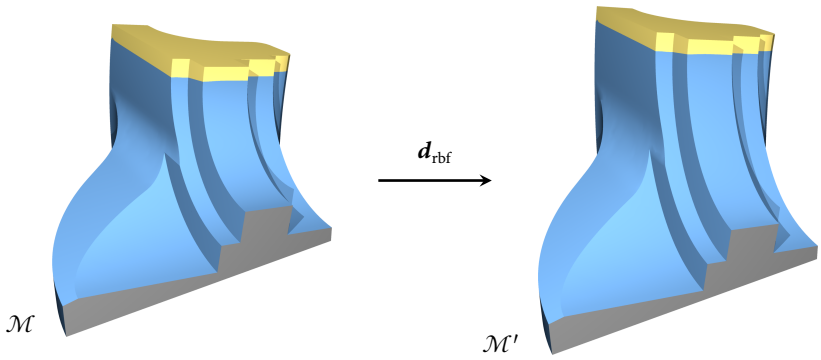


Figure 3.8: Deformation of the Fandisk model using a handle-based direct manipulation interface for RBFs.

potentially complex control structure such as a control grid or a bounding cage enclosing the target shape. Second, by exploiting the aforementioned flexibility and by choosing an appropriate interpolation method, we construct the space deformation function in such a way that it *smoothly* interpolates displacements through space, *exactly* satisfies the user-specified modeling constraints, and directly minimizes a *physically-inspired* deformation energy—thereby resulting in a smooth, highly precise as well as physically plausible deformation.

Shape deformations based on scattered data approximation and interpolation have been proposed by different authors and within a variety of application contexts such as computer graphics, physics simulation, as well as design optimization. In particular, methods based on radial basis functions (RBFs) have gained increasing attention (Ruprecht et al. 1995; Botsch and Kobbelt 2005; Boer et al. 2007; Jakobsson and Amoignon 2007; Michler 2011), and we concentrate our investigation on this particular method. We illustrate an example deformation based on radial basis functions employing a handle-based direct manipulation interface in figure 3.8.

On an abstract level, we can treat space deformation as a scattered data interpolation problem: We search for a function $\boldsymbol{d}\colon \mathbb{R}^3 \to \mathbb{R}^3$ that (i) exactly interpolates the prescribed displacements $\boldsymbol{d}(\boldsymbol{h}_i) = \boldsymbol{h}_i' - \boldsymbol{h}_i = \boldsymbol{b}_i$ and (ii) smoothly interpolates these displacements through space. Radial basis functions are well known to be suitable for solving this type of problem (Wendland 2010). Using RBFs, we define the deformation function as a linear combination of radially symmetric kernel

functions $\varphi_j(\boldsymbol{x}) = \varphi(\|\boldsymbol{x} - \boldsymbol{c}_j\|)$, located at centers $\boldsymbol{c}_j \in \mathbb{R}^3$ and weighted by $\boldsymbol{w}_j \in \mathbb{R}^3$, plus a linear polynomial to guarantee linear precision:

$$\boldsymbol{d}_{\text{rbf}}(\boldsymbol{x}) \;=\; \sum_{j=1}^{m} \boldsymbol{w}_j \varphi_j(\boldsymbol{x}) \;+\; \sum_{k=1}^{4} \boldsymbol{q}_k \pi_k(\boldsymbol{x})\,, \tag{3.8}$$

where $\{\pi_1, \pi_2, \pi_3, \pi_4\} = \{x, y, z, 1\}$ is a basis of the space of linear trivariate polynomials, weighted by coefficients $\boldsymbol{q}_k \in \mathbb{R}^3$. Note that the polynomial term is important, since it guarantees to find the optimal affine motion (translation, rotation, scaling) contained in the prescribed displacements $\boldsymbol{b}_i$.

The choice of the kernel function $\varphi\colon \mathbb{R} \to \mathbb{R}$ basically determines the shape of the interpolant. Commonly used kernels include Gaussians, (inverse) multiquadrics, and polyharmonic splines (see table 3.1 for an overview). In our application, we aim for high quality deformations minimizing the distortion of mesh elements. To meet this goal, we have to use a sufficiently smooth kernel function. While Gaussian and multiquadric basis functions provide infinite smoothness, i.e., they are C^∞, they require the choice of an additional *shape parameter* (the ϵ in table 3.1). Small values of ϵ increase the approximation accuracy, but lead to numerically instabilities, and *vice versa*. Therefore, finding the optimal shape parameter for a given radial basis function and the particular application is a non-trivial task on its own, see Fasshauer (2007) for an overview of different strategies.

In contrast, polyharmonic splines are free of shape parameters, but only provide finite smoothness. Therefore, depending on the application scenario, we have to choose a sufficiently high degree of smoothness. In $\mathbb{R}^3$ the polyharmonic spline

Table 3.1: Commonly used RBFs. For Gaussians and (inverse) multiquadrics ϵ denotes the shape parameter. For polyharmonic splines k denotes the order of smoothness.

Basis function	Definition
Gaussian	$\varphi(r) = \mathrm{e}^{-(\epsilon r)^2}$
Multiquadric	$\varphi(r) = \sqrt{1 + (\epsilon r)^2}$
Inverse multiquadric	$\varphi(r) = 1/\sqrt{1 + (\epsilon r)^2}$
Polyharmonic spline in $\mathbb{R}^d$	$\varphi_k(r) = \begin{cases} r^{2k-d}, & d \text{ odd}, \\ r^{2k-d}\log(r), & d \text{ even}. \end{cases}$

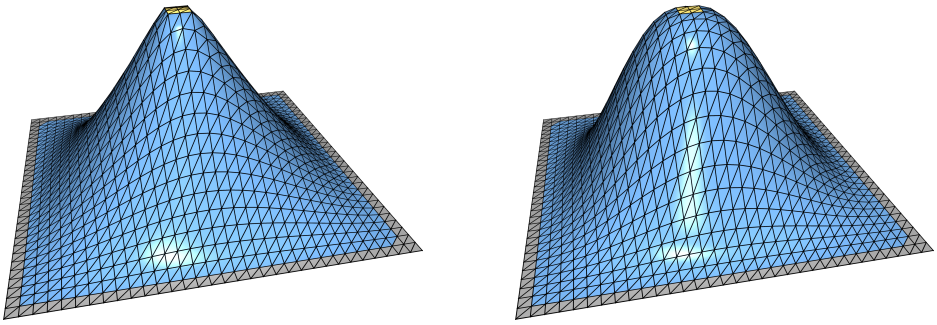


Figure 3.9: Comparison between a biharmonic (left) and a triharmonic (right) RBF deformation of a plane.

$\varphi_k(r) = r^{2k-3}$ is a fundamental solution of the k-th order Laplace equation, such that also the RBF deformation of equation (3.8) is k-harmonic, i.e., $\Delta^k \boldsymbol{d} = 0$. Being the strong form of a variational energy minimization, this is equivalent (Wendland 2010) to $\boldsymbol{d}$ minimizing the weak form

$$\iiint_{\mathbb{R}^3} \left\| \frac{\partial^k \boldsymbol{d}}{\partial x^k} \right\|^2 + \left\| \frac{\partial^k \boldsymbol{d}}{\partial x^{k-1} \partial y} \right\|^2 + \ldots + \left\| \frac{\partial^k \boldsymbol{d}}{\partial z^k} \right\|^2 \mathrm{d}x\, \mathrm{d}y\, \mathrm{d}z. \tag{3.9}$$

In order to preserve mesh quality during deformation, we should construct a deformation function that at least minimizes the change of first-order derivatives of the mesh elements (Staten et al. 2011), and therefore the first-order derivatives of the deformation function. With $k = 1$ in equation (3.9), this is achieved by the harmonic RBF $\varphi(r) = 1/r$, but these basis functions are singular at their centers. The biharmonic spline $\varphi(r) = r$ is well defined, but not sufficiently smooth at the center and therefore not suitable for our application, see figure 3.9 for an illustration. By choosing $\varphi(r) = r^3$, we obtain a deformation function that is triharmonic, therefore penalizes third-order derivatives in equation (3.9), and is globally C^2 smooth. With these properties, it is the lowest-order polyharmonic RBF suitable for our application. Since for the sake of numerical robustness a low order is preferable, we chose triharmonic RBFs for our deformation method.

Now that we have chosen an appropriate basis function type, the next question is where to *place* the RBF kernels. Since in our application it is desirable to *exactly* satisfy the interpolation constraints $\boldsymbol{d}(\boldsymbol{h}_i) = \boldsymbol{b}_i$, we simply place the RBF kernels at the constraint positions (i.e., $\boldsymbol{c}_j = \boldsymbol{h}_j$). Computing the deformation function

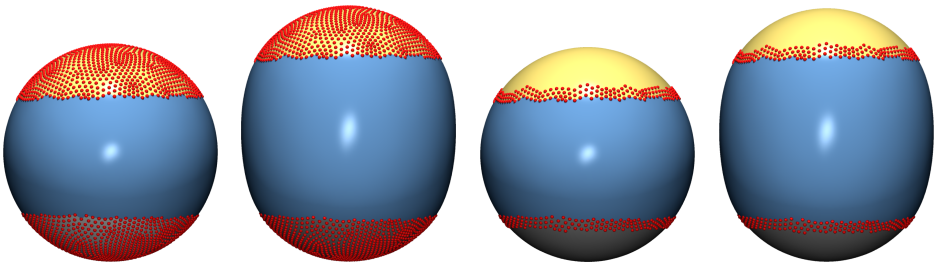


Figure 3.10: Reducing the number of RBF constraints: Left: Full constraint set for fixed and handle vertices using 3537 constraints. Right: Reduced constraint set using 779 constraints only.

then amounts to finding the coefficients $\boldsymbol{w}_j$ and $\boldsymbol{q}_k$, which we do by solving the $(m+4) \times (m+4)$ linear system

$$\boldsymbol{\Phi} \cdot \boldsymbol{W} = \boldsymbol{B}, \tag{3.10}$$

where

$$\boldsymbol{\Phi} = \begin{bmatrix} \varphi_1(\boldsymbol{h}_1) & \cdots & \varphi_m(\boldsymbol{h}_1) & \pi_1(\boldsymbol{h}_1) & \cdots & \pi_4(\boldsymbol{h}_1) \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \varphi_1(\boldsymbol{h}_m) & \cdots & \varphi_m(\boldsymbol{h}_m) & \pi_1(\boldsymbol{h}_m) & \cdots & \pi_4(\boldsymbol{h}_m) \\ \pi_1(\boldsymbol{h}_1) & \cdots & \pi_1(\boldsymbol{h}_m) & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \pi_4(\boldsymbol{h}_1) & \cdots & \pi_4(\boldsymbol{h}_m) & 0 & \cdots & 0 \end{bmatrix},$$

$$\boldsymbol{W} = \left[\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m, \boldsymbol{q}_1, \ldots, \boldsymbol{q}_4\right]^{\mathrm{T}},$$

and

$$\boldsymbol{B} = \left[\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m, \boldsymbol{0}, \ldots, \boldsymbol{0}\right]^{\mathrm{T}}.$$

After solving equation (3.10) we can compute the deformed model $\mathcal{M}'$ by simply evaluating the RBF deformation function at each point $\boldsymbol{x}$ of the model.

We solve the linear system in equation (3.10) using the $\mathrm{LDL}^{\mathrm{T}}$ decomposition of LAPACK (E. Anderson et al. 1999) and use the same handle-based direct manipulation interface as described in figure 2.1. Handle and fixed vertices enter the equation system as constraints with given displacements (either the handle displacements directly or zero displacements for fixed vertices). If the number of handle and fixed vertices is large, equation (3.10) becomes time-consuming to solve. In case of deforming a triangle mesh, however, it is rather straightforward to

reduce the number of constraints. Following the argument of Botsch and Kobbelt (2005) that a band of three points thickness is sufficient to provide C^2 boundary constraints, we only include fixed and handle vertices in the 3-ring of a deformable vertex, see figure 3.10.

3.6 SUMMARY

In this chapter, we introduced a variety of state-of-the-art shape deformation techniques for their use in design optimization. Due to their wide-spread use in both academic as well as industrial contexts, we started our investigation with classical free-form deformation as well as its direct manipulation variant. We introduced cage-based deformations as a generalization of FFD to more flexible surface-based control structures and extensively discussed the problem of generating a coarse bounding cage. Finally, we introduced RBF deformations as a technique allowing for an even simpler and more flexible control structure—a set of points in space—while offering a high degree of precision and deformation quality. While we already highlighted selected important aspects of the individual methods in this chapter, we continue with a more thorough investigation in the next chapter—benchmarks.

CHAPTER 4

BENCHMARKS

In this chapter, we investigate the individual strengths and weaknesses of the different deformation methods introduced in the previous chapter 3 based on a set of representative yet synthetic benchmarks. The overall goal of these benchmarks is to capture the basic properties and capabilities relevant for the method's use in design optimization scenarios. We perform our evaluation based on the following criteria: computational performance, robustness (both numerical as well as regarding defects in the input data), adaptivity and precision, as well as quality of the deformation. For each of our evaluation criteria we first describe our tests and methodology and then present the results for each of the deformation methods.

4.1 INTRODUCTION

Providing meaningful benchmarks for a set of different deformation methods comes with a number of challenges. In some cases a direct comparison between all methods and variants is only of limited significance due to the differences in modeling capabilities. In particular, comparisons with the classical *in*-direct FFD and Cage deformation methods are problematic, since in these cases it is impossible to perform exactly the same deformation on a representative basis, e.g., satisfying the same user-defined deformation constraints with a given number of degrees of freedom. In such cases, we concentrate on the corresponding direct manipulation variants of FFD and Cages and provide only qualitative results for the classical formulations.

We performed all tests on a Dell T7500 workstation with an Intel Xeon E5645 2.4 GHz CPU and 18GB RAM running Ubuntu Linux 12.04 x86_64. We compiled all code with `gcc` 4.6.3, optimization turned on (using `-O3`) and debugging checks disabled (`-DNDEBUG`). In order to rule out processor and memory caching as well as power saving issues, we averaged performance timings over five deformation steps. Unless noted otherwise, we used the same setup for all benchmark results presented in this thesis.

4.2 PERFORMANCE

While the impact of the performance of a deformation method is often negligible when used within a design optimization loop, it is still an important and fundamental characteristic. Furthermore, it is crucial for interactive modeling, e.g., during initial setup of the method as well as for pre-optimization experimentation. In the following, we first discuss the theoretical performance of each method before providing an actual comparison.

Within control point-based FFD, the only performance-critical component is the computation of the local coordinates of each object point with respect to the control lattice. When using B-spline basis functions, this computation requires the use of a numerical technique such as a golden section search or a Newton method (Press et al. 2007). However, since the local coordinate computation is independent for each object point, this part is trivial to parallelize.

Naturally, direct manipulation FFD also requires the local coordinate computation discussed above. In addition, however, the linear system of equation (3.5) has to be solved. The standard approach for this is based on computing the pseudo-inverse using singular value decomposition, which has a computational cost of $4m^2n + 22n^3$ floating point operations (Golub and Van Loan 2013), where m and n are the number of constraints and control points, respectively. Additional computational costs come from the matrix multiplications required to actually compute the pseudo-inverse $\boldsymbol{\Phi}^+ = \boldsymbol{V}\boldsymbol{\Sigma}^+\boldsymbol{U}^\mathrm{T}$ from the SVD.

For cage-based deformations the most expensive part is the computation of mean value coordinates. However, since these rely on simple geometric constructions only they do not require the use of an expensive numerical technique. Still, due to the global support of the basis functions we have to loop over all cage vertices for evaluating basis functions (unlike FFD where we can use locally supported B-spline basis functions). Within the direct manipulation formulation of cage-based deformations the same performance limitations as in case of DM-FFD apply.

Within the RBF deformation technique the most expensive part is the solution of the linear system of equation (3.10), which is dense due to the global support of the chosen radial basis functions $\varphi(r) = r^3$. The resulting asymptotic complexity is $\mathcal{O}(m^3)$ when using standard solvers for dense linear systems. Since the linear system in equation (3.10) is symmetric but not positive definite, efficient Cholesky-type solvers are not applicable. However, we can still solve the system efficiently by

using a $\mathrm{LDL}^{\mathrm{T}}$ factorization, which has computational costs of $\frac{1}{3}m^3$ floating point operations, and also only requires half the storage space due to symmetry. For a comprehensive investigation of advanced linear solvers for RBF deformation we refer to section 7.5.

In case of Shells the performance is dominated by the solution of the linear system of equation (3.4). Since the Laplacian matrices $\boldsymbol{L}^k$ in equation (3.4) are highly sparse we can use efficient direct sparse linear solvers (Botsch et al. 2005), and the resulting asymptotic complexity is approximately $\mathcal{O}(m)$.

However, benchmarking the performance by simply measuring the time it takes to deform a given mesh is not really meaningful since the methods pre-compute different amounts of information. Comparing the performance of control point-based FFD to directly manipulated DM-FFD or RBFs is not feasible, since there is no way to perform the same deformation with all three methods. In order to facilitate a representative and objective comparison between the methods, we present an alternative formulation of all deformation methods which allows us to fully pre-compute the deformation. The deformation methods we investigate are linear, i.e., they require solving a linear problem in one form or another. Therefore, the deformations can be pre-computed by solving a sequence of linear systems, see, e.g., Botsch and Kobbelt (2005). Even more, the methods can be handled in a uniform manner by expressing the deformation in terms of *handle basis functions*.

Let m be the number of displacement constraints $\boldsymbol{b}_i$ which are given as prescribed values of the deformation function $\boldsymbol{d}(\boldsymbol{h}_i) = \boldsymbol{b}_i$. In case of DM-FFD, the control point displacements $\boldsymbol{\delta}_j$ satisfying these constraints are found by solving the linear system of equation (3.5). In case of RBFs, we find the weights $\boldsymbol{w}_j$ for the deformation function $\boldsymbol{d}_{\mathrm{rbf}}$ by solving equation (3.10). What we are searching for are the displacements $\boldsymbol{x}_i' - \boldsymbol{x}_i = \boldsymbol{a}_i$ for each deformable vertex $\boldsymbol{x}_i$. Written in matrix form this becomes $\boldsymbol{A} = [\boldsymbol{a}_1, \dots, \boldsymbol{a}_k]^{\mathrm{T}}$, where k is the number of deformable vertices. In case of DM-FFD, $\boldsymbol{A}$ can be computed using

$$\boldsymbol{A} \;=\; \boldsymbol{\Theta} \cdot \boldsymbol{D}, \quad \boldsymbol{\Theta}_{ij} = \varphi_j(\boldsymbol{u}(\boldsymbol{x}_i)), \tag{4.1}$$

where $\varphi_j(\boldsymbol{u}(\boldsymbol{x}_i))$ is the trivariate tensor-product B-spline basis function of control point $\boldsymbol{c}_j$ evaluated at point $\boldsymbol{x}_i$, and $\boldsymbol{D}$ is the matrix of control point displacements $\boldsymbol{\delta}_j$. By substituting $\boldsymbol{D}$ using equation (3.7) we can rewrite equation (4.1) as

$$\boldsymbol{A} \;=\; \underbrace{\boldsymbol{\Theta} \cdot \boldsymbol{\Phi}^{+}}_{\boldsymbol{\Psi}} \cdot \boldsymbol{B},$$

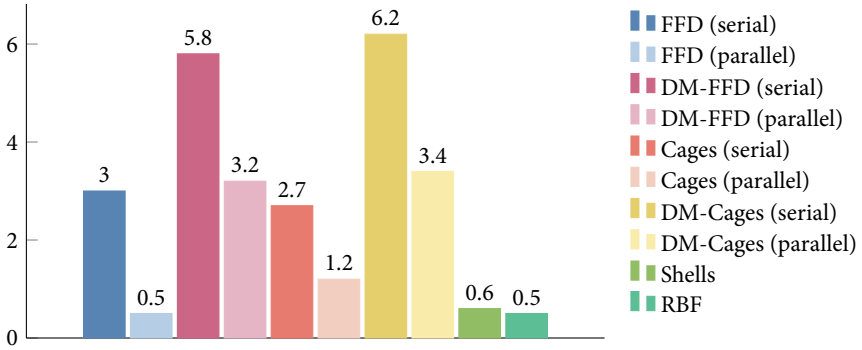


Figure 4.1: Performance comparison of the deformation methods on a simple sphere mesh with 7800 vertices and 526 handle and 404 fixed vertices. Times in seconds. For FFD methods a control grid of resolution 8^3 was used. For Cage deformations a cage with 512 vertices was used. For RBFs 427 centers were used.

where $\boldsymbol{B}$ is the $m \times 3$ matrix of prescribed handle displacements. Using the $k \times m$ matrix $\boldsymbol{\Psi}$ we can then directly evaluate the vertex displacements in terms of handle displacements. In case of direct manipulation cage deformation the formulation follows exactly the FFD procedure, but with using mean value coordinates instead of B-spline basis functions.

The corresponding formulation for RBF deformations is similar: The matrix $\boldsymbol{A}$ can be computed by

$$\boldsymbol{A} = \boldsymbol{\Theta} \cdot \boldsymbol{W}, \quad \boldsymbol{\Theta}_{ij} = \varphi_j(\boldsymbol{x}_i), \tag{4.2}$$

where $\boldsymbol{W}$ is the matrix of radial basis function weights. Based on equation (3.10) the weight matrix $\boldsymbol{W}$ can be computed by inverting $\boldsymbol{\Phi}$, i.e., as $\boldsymbol{W} = \boldsymbol{\Phi}^{-1}\boldsymbol{B}$. This yields

$$\boldsymbol{A} = \underbrace{\boldsymbol{\Theta} \cdot \boldsymbol{\Phi}^{-1}}_{\boldsymbol{\Psi}} \cdot \boldsymbol{B}.$$

Then $\boldsymbol{\Psi}$ is the desired $k \times m$ basis function matrix that can be used to compute the vertex displacements from the given handle displacements.

The precomputation in terms of handle basis functions for thin shell deformations is described in detail by Botsch and Kobbelt (2004b). We therefore do not repeat it here.

Based on this formulation, we compare the performance of the methods by pre-computing a deformation with 427 constraints. In figure 4.1 we present the

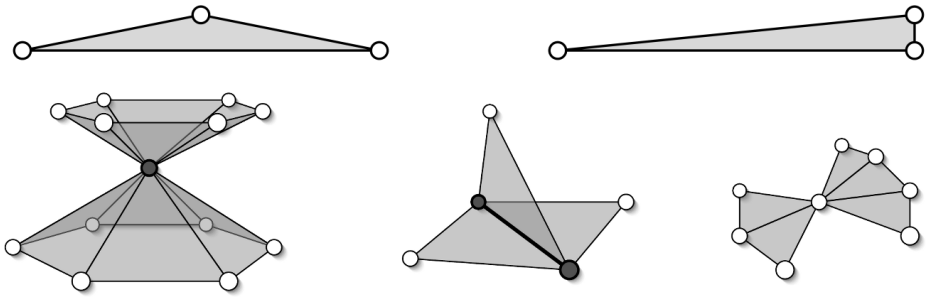


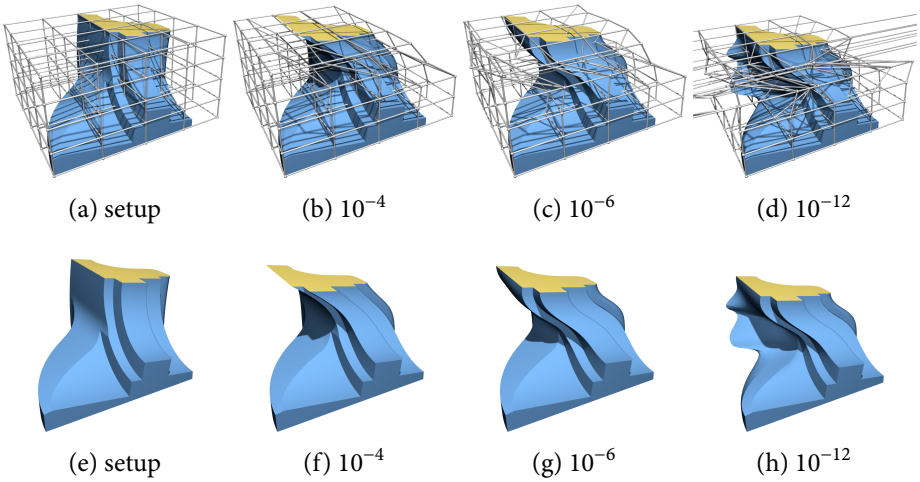
Figure 4.2: Examples of common surface mesh degeneracies. Top row: Low quality triangles. Bottom row: Non-manifold connectivity. Bottom row image reproduced with permission from Botsch (2008).

results comparing FFD, Cages, Shells, and RBFs. In case of FFD and Cages, we include both parallel and serial implementations of their respective control point and direct manipulation variants. We use OpenMP (Dagum and Menon 1998) for parallelization. As to be expected from theory, DM-FFD and DM-Cages offer the worst performance. While parallel coordinate computation clearly improves (DM-)FFD and (DM-)Cages performance, RBFs and Shells require roughly the same amount of time to solve the full problem. Furthermore, we note that for Cages the numbers do not include the time required for cage generation. Depending on the different processing options outlined in section 3.4.1 the cage generation process based on offsetting and surface reconstruction takes ~15s in this scenario and would therefore be the dominating factor for cage-based deformations.

4.3 ROBUSTNESS

The robustness of a deformation method describes its robustness towards defects in the input data as well as general numerical stability. Common defects in the input mesh include low-quality triangles with very large or very small angles, such as caps or needle elements, non-manifold configurations, or self-intersections. We illustrate selected examples in figure 4.2.

Due to their space-based nature, FFD, Cages, and RBFs are highly robust with respect to defects in the input data. However, in the direct manipulation variants of FFD and Cages the singular value decomposition used to compute the pseudo-inverse can be a source for numerical instabilities. In order to prevent division



(a) setup (b) 10^{-4} (c) 10^{-6} (d) 10^{-12}

(e) setup (f) 10^{-4} (g) 10^{-6} (h) 10^{-12}

Figure 4.3: Artifacts in the deformation of the Fandisk model depending on the clamping value used for small singular values. DM-FFD with a $5 \times 5 \times 5$ control lattice. Original setup in (a) and (e), different clamping values as fractions of the largest singular value $\sigma_{\max}$: 10^{-4} (b) and (f), 10^{-6} (c) and (g), and 10^{-12} (d) and (h).

by zero, artifacts in the deformation, as well as extreme distortions of the control lattice, it is necessary to clamp small singular values σ_i in equation (3.6). The clamping value is typically determined as a fraction of the largest singular value $\sigma_{\max}$, see Press et al. (2007) for details. In figure 4.3, we illustrate examples of unwanted artifacts in the deformation depending on different clamping values: Choosing a too high value damps the deformation in an undesirable manner (10^{-4} in figure 4.3), while a too low value (10^{-12} in figure 4.3) leads to severe artifacts in the deformed model as well as the control grid. Since a suitable clamping value for a given deformation setup is not known a-priori, it has to be determined heuristically—thereby constituting a source of increased effort and potential failure.

Non-manifold configurations are problematic in general, since in this case it is no longer possible to reliably traverse the local neighborhood of a vertex within a surface. While this does not pose a direct problem for the space deformation methods we investigate, it can prevent the use of a more efficient reduced set of handle constraints in a direct manipulation interface, such as described for RBFs in section 3.5. For purely surface-based methods such as the thin shell deformation of section 3.1 such severe defects render the technique practically inapplicable.

4.4 QUALITY

The quality of a deformation includes several aspects. On the most general level, the deformation should be free of any unexpected oscillations or artifacts. Following the principle of *simplest shape* (Sapidis 1994), the deformation should be smooth, fair, and physically plausible. Furthermore, we want the deformation to maintain mesh element quality as good as possible in order to allow for as large as possible deformations while maintaining meshes suitable for downstream simulation. We note, however, that the methods we consider do not incorporate explicit mesh optimization steps that are possibly required for particularly large deformations.

As a first benchmark we investigate the smoothness of the deformation techniques by analyzing the curvature of a surface mesh after deformation. More specifically, we consider *mean curvature* defined as

$$H = \frac{\kappa_1 + \kappa_2}{2},$$

where κ_1 and κ_2 are the principal (maximum and minimum) curvatures of the surface. Using the cotangent weight discretization of the Laplace-Beltrami operator (Meyer et al. 2003) we compute the discrete absolute mean curvature for a given vertex $v_i \in \mathcal{T}$ with coordinates $\boldsymbol{x}_i$ as

$$H(v_i) = \frac{1}{2} \left\| \Delta \boldsymbol{x}_i \right\|,$$

where Δ is the discrete Laplace-Beltrami operator. For more details on discrete curvature computation we refer to chapter 3 of Botsch et al. (2010).

We present a color-coded mean curvature visualization after performing a pre-defined deformation with DM-FFD, DM-Cages, RBFs and Shells in figure 4.4. The visualizations demonstrate that the DM-FFD and DM-Cages techniques suffer from aliasing artifacts due to their lattice-based nature. Note that the same artifacts occur in control point based FFD and Cage deformations. In contrast, both RBFs and Shells result in highly smooth deformations due to their built-in minimization of physically-inspired energies as described in sections 3.1 and 3.5. Note that for illustration purposes we deliberately chose a deformation setup that is not aligned with FFD grid axes. In other cases the aliasing artifacts might be less severe.

Both FFD and Cages result in deformations that are not necessarily physically plausible. Especially their direct manipulation variants do not optimize for a high

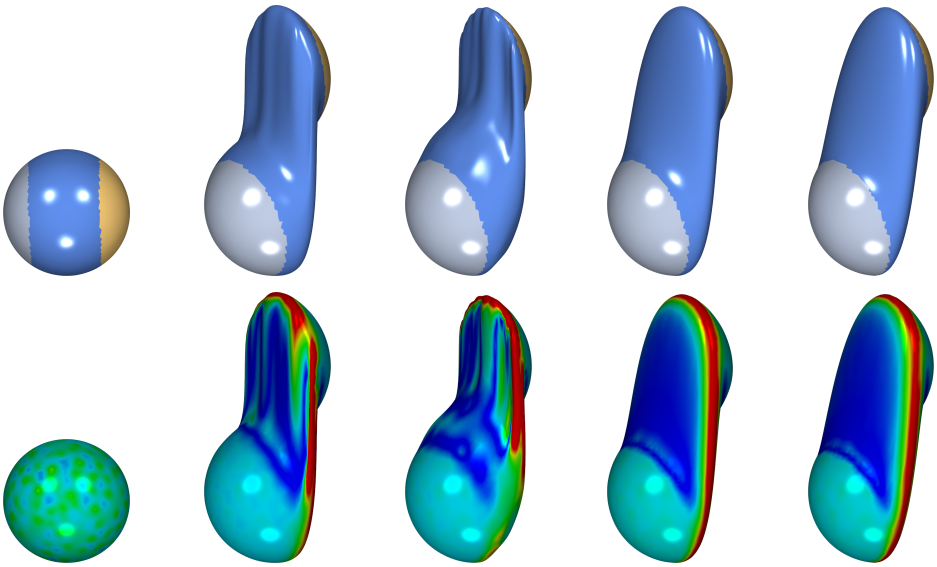


Figure 4.4: Mesh smoothness after deformation. Top row: Deformed meshes. Bottom row: Mean curvature plots. From left to right: Setup, DM-FFD (729 control points), DM-Cages (790 cage vertices), RBFs (792 kernels), and Shells.

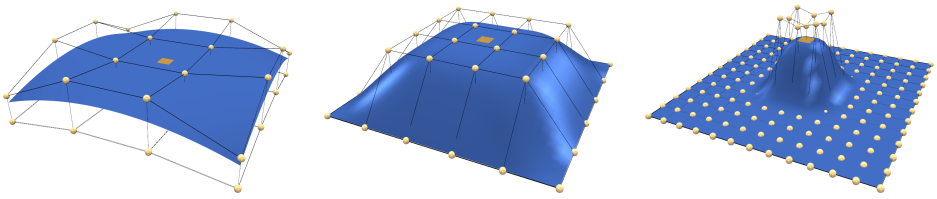


Figure 4.5: Dependency of the deformation on the control lattice resolution. For all examples the same handle region was moved by the same translation.

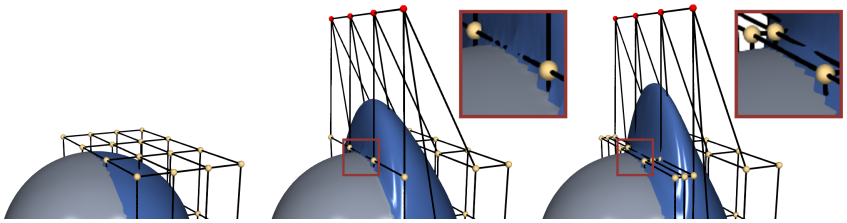


Figure 4.6: Continuity problems in FFD in case of partial control grids. From left to right: Original setup, non-smooth transition, smooth transition after refinement.

quality deformation, but for minimization of control point or cage vertex movement. In general, when using a lattice-based method the shape of the deformation strongly depends on the *resolution* and *form* of the control lattice, as shown for FFD in figure 4.5. Therefore, it becomes highly difficult to predict the shape resulting from a particular deformation setup in advance.

Another problem with lattice-based methods is the continuity in case of partial control grids. If the control grid covers only a subset of the object, non-smooth transitions between object points inside the control volume and those outside may occur (see figure 4.6, center). In such cases additional sheets of control points have to be inserted into the grid in order to ensure a smooth transition (figure 4.6, right). This not only complicates the setup process of FFD, it also introduces unnecessary degrees of freedom due to bad adaptivity (see also the next section 4.5). While control cages are more flexible to refine locally, the global nature of mean value coordinates makes partial control cages not feasible at all. In contrast, in case of RBFs a C^2 smooth transition between deformable and fixed/handle regions is easily achieved by using three rings of vertices in the local neighborhood of deformable vertices, see Botsch and Kobbelt (2005) and section 3.5 for details.

4.5 ADAPTIVITY

In general, the adaptivity of a deformation method describes how well the method is capable of approximating a certain shape with an as low as possible number of degrees of freedom (DoFs). In the context of shape optimization the ability to dynamically add additional DoFs in regions of high interest or sensitivity towards physical performance is particularly important.

In order to evaluate adaptivity, we use a benchmark that matches a source shape to a given target shape. In this test, we use all vertices of the mesh as prescribed constraints, and the deformation method has to match the shape as closely as possible. For each of the methods we start with a low number of DoFs and successively refine the method to include more and more DoFs. We stop refinement once the number of DoFs is equal to the number of constraints.

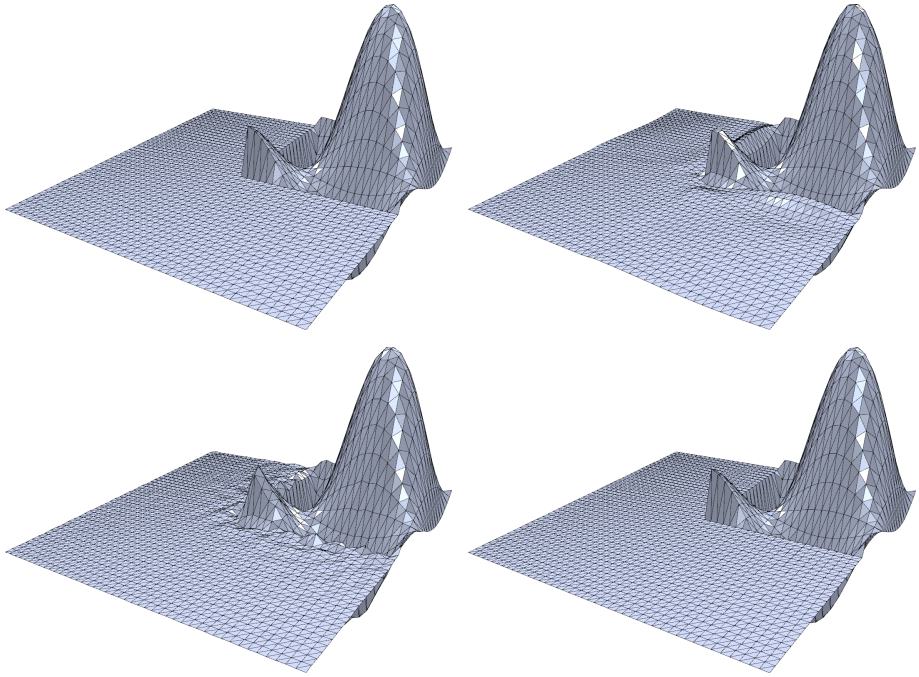
Adaptivity can be measured best when approximating a target shape that is identical to the source shape for a large part of vertices while having sharp local features in another region, i.e., some parts require almost no DoFs to satisfy the constraints while others do require a large number. Therefore, we use a target shape based on a simple plane with vertices displaced in the upper right area of the

mesh, see figure 4.7 (top left) for an illustration. Note that this shape is particularly demanding since the transition from the plane to the feature area is non-smooth and highly steep.

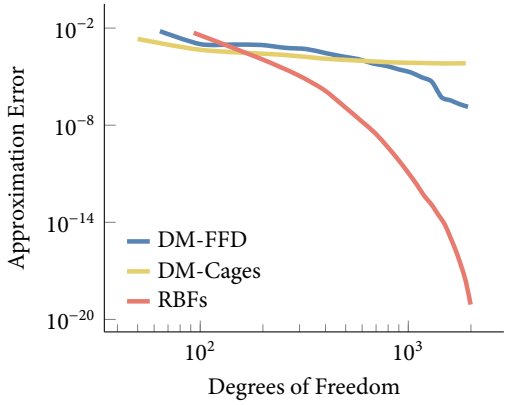
In case of DM-FFD, we perform adaptive refinement by inserting additional control point planes in $x-$ and $y-$directions in those cells containing the vertex with the largest error. We do not perform refinement in z-direction, since in our example this would only result in wasted degrees of freedom. As becomes clear from figure 4.7, the adaptivity of DM-FFD is generally poor, since it depends on the resolution of the lattice being used. While increasing the resolution of the lattice leads to sufficient degrees of freedom to approximate fine details as well, at the same time the insertion process also alters the deformation itself, which is particularly undesirable if using refinement during an optimization process. We note, however, that we could overcome this limitation by using *knot insertion* instead of control point refinement.

Since the control cage of DM-Cages is a triangle mesh, we can adaptively refine the cage by using remeshing algorithms. Adaptive remeshing alters the mesh in such a way so that important regions contain smaller triangles and therefore more DoFs, while less important regions contain larger triangles. A challenge for adaptive remeshing is the construction of a suitable *sizing function* that describes how fine the mesh resolution should be in a certain area of the mesh. Typically, the sizing information is derived from geometric properties of the mesh, such as its curvature. In our application, however, we need to employ a sizing field in accordance to the regions of highest approximation errors, which changes during the refinement process. Therefore, we use a simple mapping from approximation errors to target edge lengths as sizing function. Similar to DM-FFD, the results in figure 4.7 demonstrate that the adaptivity of DM-Cages is generally poor, which is due to the global nature of mean value coordinates as well as a lack in precision (see also section 4.6).

In case of RBFs we use straightforward adaptive greedy refinement (Schaback and Wendland 2000). Initially, we uniformly sample the plane with a given number of kernels. We then successively add additional kernels at the vertices of the mesh having the largest errors. The results in figure 4.7 clearly confirm that RBFs provide superior approximation accuracy compared to both DM-FFD and DM-Cages. Note that RBFs preserve the sharp features so well because we place kernels directly on the mesh vertices and thereby exactly interpolate the prescribed displacements.



(a)



(b)

Figure 4.7: Adaptive refinement benchmark results. (a) Example results. Top row: target shape (left) and DM-FFD (900 DoFs, right). Bottom row: DM-Cages (1094 DoFs, left) and RBFs (993 DoFs, right). (b) Approximation error vs. degrees of freedom.

Since in the surface-based shell deformation the DoFs are the vertex positions of the mesh, performing adaptive refinement for this method is not directly possible, at least not without changing the object representation through remeshing or through modifying the thin shell formulation similar to the least squares meshes approach of Sorkine and Cohn-Or (2004).

4.6 PRECISION

The precision of a deformation method describes its accuracy in satisfying the positional constraints as prescribed by the user or optimization method. In general, we can distinguish at least three different levels of accuracy: The constraints are typically satisfied either *exactly*, in a *least-squares* sense, or only in a *qualitative* manner.

Manipulating control points of a lattice as in case of FFD or Cages can only provide qualitative precision since there is no way to automatically satisfy prescribed constraints. Directly manipulated FFD and Cages improve on this by providing precision in a least-squares sense through the solution of equation (3.5). Finally, by solving equation (3.10), RBFs allow for exact satisfaction of constraints, thereby offering the highest level of precision. The quantitative results of section 4.5 underline these differences in precision. Finally, Shell-based deformation also allows for the exact satisfaction of user constraints due to its surface-based nature.

4.7 SUMMARY AND CONCLUSION

The results of the individual benchmarks show that there are significant differences between the deformation methods. A compact and simplified summary of the results is presented in table 4.1. For each of the benchmarks and methods we assign either a positive (+), neutral (•), or a negative (–) assessment. Both FFD and Cages obtain mixed results. While their direct manipulation variants provide improvements in terms of precision, both computational costs increase and robustness issues might occur. Both FFD- and Cage-based techniques expose significant weaknesses with regards to adaptive refinement and quality of the deformation. In contrast, RBFs score the largest number of positive and only one neutral assessment. The thin shell deformation method yields positive results in general. However, due to its surface-based nature its applicability to design

optimization is limited: There are no means for adaptive refinement and it is not possible to deform a volumetric simulation mesh along with a surface.

However, the choice of a deformation method heavily depends on the needs of a given design optimization scenario. If the scenario neither demands for precise constraint satisfaction nor for adaptive refinement but only aims for general exploration of the design space, FFD offers a simple and robust deformation technique. In many cases, however, exact control is highly important in order to obtain valid designs that meet production limitations such as keeping critical components fixed or deforming them only rigidly. In such cases, we clearly recommend RBFs over both FFD and Cages including their respective direct manipulation variants.

In some optimization scenarios the locality of the deformation might also be an important aspect. Both FFD methods allow for local deformations, depending on control grid resolution and setup as well as basis function degree. In contrast, our RBF deformations are global due to our choice of triharmonic basis functions. While there exist compactly supported RBFs (Wendland 2010), these basis functions lack the built-in energy minimization of equation (3.9), see also chapter 8. However, in many cases a proper setup of fixed and handle regions in the direct manipulation interface eventually provides a sufficient degree of locality. The recently proposed local barycentric coordinates (Zhang et al. 2014) provide an interesting alternative enabling localized cage-based deformations.

Naturally, all of three methods can be enhanced in several ways. In case of both FFD methods the use of more flexible basis functions such as T-splines (Sederberg et al. 2003) or truncated hierarchical B-splines (Giannelli et al. 2012) would drastically improve the adaptivity of the respective methods. As for RBFs, constraining

Table 4.1: Summary of evaluation results. For each benchmark test and deformation method we assign a negative (−), neutral (•), or a positive (+) assessment.

	Performance	Robustness	Quality	Adaptivity	Precision
FFD	•	+	•	−	−
DM-FFD	−	•	•	−	•
Cages	•	•	•	−	−
DM-Cages	−	•	•	−	•
RBF	•	+	+	+	+
Shells	+	•	+	−	+

the deformation function to be positive—similar to the bounded biharmonic weights introduced in (Jacobson et al. 2011)—offers an interesting possibility for future enhancement.

Taking into account the different results of our benchmarks as well as the individual capabilities of the deformation methods, we will from now on narrow down our selection of techniques. Cages—at least in their incarnation of mean value coordinates—do not offer a significant improvement over FFD, except in terms of flexibility and ease of implementation. Furthermore, the cage generation problem forms a serious obstacle to its seamless adoption within fully automatic design optimization. Similarly, it becomes clear that the thin shell deformation method—while providing a high and desirable level of modeling flexibility—is too limited for design optimization in general due to its surface-based nature. Therefore, we focus on FFD and RBF deformation methods from now on and further analyze these methods in our application-oriented design optimization benchmark.

CHAPTER 5

EVOLUTIONARY DESIGN OPTIMIZATION

In this chapter, we investigate the use of shape deformation techniques within design optimization scenarios in more detail. To this end, we first introduce basic concepts of *evolutionary algorithms* and outline their utilization in *evolutionary design optimization*. Finally, we present an application-oriented benchmark comparing classical free-form deformation, its direct manipulation variant, as well as deformations based on radial basis functions in a passenger car design optimization task. Our results indicate that shape deformation techniques directly manipulating the design prototype lead to faster convergence of the optimization process. In addition, we conclude that the flexibility of kernel-based deformation methods such as RBFs leads to reduced time and effort in the setup of the optimization scenario.

5.1 OVERVIEW

Evolutionary algorithms employ principles of biological evolution such as reproduction and mutation for solving global optimization problems in a stochastic manner (Bäck 1996). Widely used techniques include genetic algorithms, genetic programming, evolutionary programming, and evolution strategies, see the works of Bäck and Schwefel (1993), Fonseca and Fleming (1995), and Coello (1999) as well as the recent textbook of Simon (2013) for a comprehensive overview and introduction. The basic idea behind these techniques is to represent candidate solutions through a set of parameters and to measure the quality of a solution by means of a fitness or cost function. The optimization then successively improves the fitness of an initial starting solution by adapting its parameters. On a general level evolutionary algorithms have compelling advantages such as the potential ability to find global optima, the capability to generate novel and unexpected designs, robustness to noise and uncertainty, as well as the ability to deal with non-smooth, discontinuous, and multi-objective fitness functions.

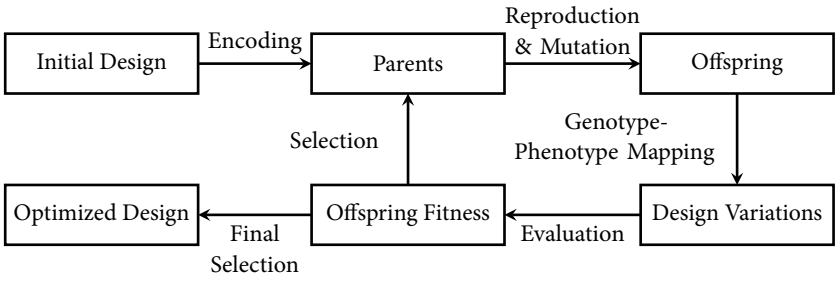


Figure 5.1: A generic evolutionary design optimization process

An overview about a generic evolutionary design optimization process is illustrated in figure 5.1. The initial design is encoded into a parent chromosome. Then a set of offspring chromosomes is created by means of reproduction and mutation, i.e., by combining selected parent chromosomes and applying changes to the new chromosomes. By mapping the offspring's *genotype*—its genetic code—to its *phenotype*—the detectable expression of its genotype—a set of design variations is created. The new designs are then evaluated with regards to a specific fitness function. The most successful offspring are selected to be the parents of the next generation and the evolution cycle starts anew. This process is repeated until a desired fitness value is reached, the optimization converges, or a maximum number of generations is reached. After a final selection step we obtain the optimized design.

Among the vast variety of evolutionary algorithms there are four prominent classes of techniques (Simon 2013): Genetic algorithms, genetic programming, evolutionary programming, and evolution strategies. On a basic level, these approaches differ in how they represent and adapt candidate solutions, thereby affecting what recombination and mutation mechanisms available, how they are actually implemented, and how suitable a given approach is for a given optimization problem. Genetic algorithms traditionally encode the solutions as string of binary numbers. Genetic programming approaches represent solutions as programs—often based on a tree structure—that are then modified during optimization. Similarly, evolutionary programming techniques employ programs as representation but only allow to change the parameters of the program during optimization. Finally, evolution strategies represent the solutions as vectors of real numbers, and we focus on this class of algorithms from this point on.

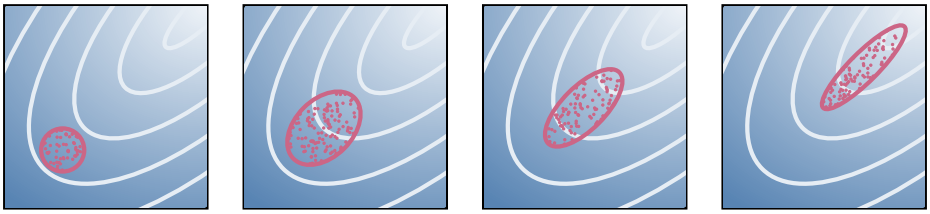


Figure 5.2: Covariance matrix adaptation in CMA-ES. Within each generation (left to right) the candidate solutions (red dots) and their distribution (red ellipse) are adapted towards successful solutions as indicated by the fitness landscape (shaded area).

5.2 Evolution Strategies

Evolution strategies (ES) are a prominent class of evolutionary algorithms for solving a wide variety of optimization problems (Rechenberg 1973, 1994). Potential solutions are typically represented using vectors of real numbers. After copying the parent chromosome for reproduction the offspring chromosomes are mutated by adding a normally distributed random vector with zero mean. In comparison to other evolutionary algorithms ES have two significant advantages: First, the strategy parameters can be adapted during optimization, thereby leading to faster convergence. Second, due to its simple vector-based encoding, incorporating constraints on the parameters is much simpler than in other methods. A comprehensive introduction to evolution strategies and its variants is given by Beyer and Schwefel (2002).

A state-of-the-art variant of ES that provides high convergence rates and is able to deal with small population sizes is the covariance matrix adaptation evolution strategy (CMA-ES), see Hansen and Ostermeier (2001) for a complete introduction. This property is particularly important when using a computationally expensive fitness functions based on large-scale CFD or FEM simulations. Within this variant of ES the covariance matrix determines the shape of the random distribution being used to create new candidate solutions. During the optimization process the covariance matrix is then adapted towards previously successful candidate solutions. We illustrate this adaptation process schematically in figure 5.2. A second characteristic that enables CMA-ES to provide fast convergence rates on small populations is the cumulative adaptation of the step size, i.e., an automatic control of the amount of change applied during each iteration.

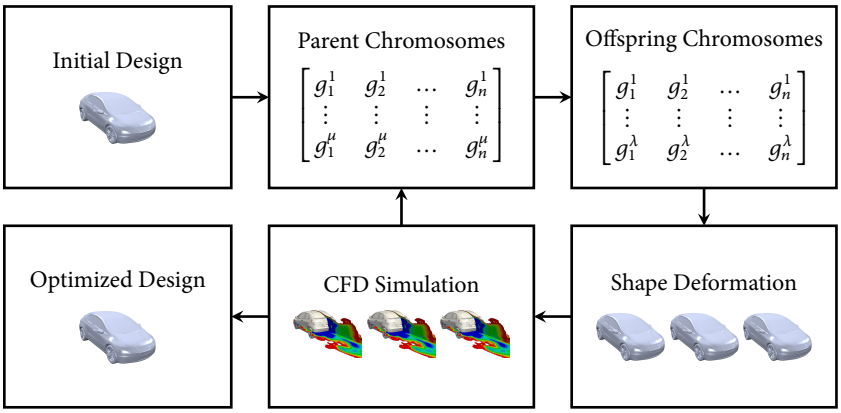


Figure 5.3: An evolutionary design optimization loop for the aerodynamic performance optimization of a passenger car.

5.3 PASSENGER CAR DESIGN OPTIMIZATION

In order to compare the individual strengths and weaknesses of the deformation methods under consideration, we choose a practical design optimization scenario: We apply the different methods in a passenger car design optimization scenario for improving the aerodynamic performance of a simplified Honda Civic. We illustrate the setup of our evolutionary design optimization loop in figure 5.3. We use the CMA-ES implementation provided by the Shark machine learning library (Igel et al. 2008) and employ a $(2, 15)$-strategy, i.e., we consider two parents and 15 offspring individuals. The selection is only performed on the offspring individuals. We encode the design using $n = 23$ parameters corresponding to spatial displacements. Furthermore, we constrain the feasible range of each parameter in order to prevent the design from becoming too flat or stretched out, as it would happen in an unconstrained optimization.

5.3.1 *Deformation Setups*

We set up the different deformation methods to deform the back part of the Civic model, a region being particular important for the aerodynamic performance of the car. In case of FFD, we pre-deform the generated control grid to roughly match the shape of the car. We define several control point groups to be deformed according to the same parameter, and we keep certain control points on the boundary of

Figure 5.4: Initial setups of the different deformation methods: FFD, DM-FFD, RBFs.

the grid fixed in order to prevent discontinuities in the boundary region. In case of DM-FFD, we use the same control grid, however, instead of using individual control point groups all non-constrained control points are allowed to move in order to satisfy the prescribed object point displacements. We choose the object points to be displaced to approximately reflect the FFD control point groups. Similar to DM-FFD, we select a set of 20 handle regions for the RBF deformation method. Note that at the time of conducting these experiments a completely equivalent handle-based direct manipulation interface was not available for DM-FFD. We illustrate the initial setups of the different deformation methods in figure 5.4. The number of vertices being deformed is around 54k for all deformation methods.

5.3.2 *Genotype-Phenotype Mapping*

The genotype of an offspring chromosome is mapped to its phenotype by establishing a correspondence between each component of the chromosome and a displacement into one spatial direction. Depending on the deformation method a given control point group (FFD), group of object points (DM-FFD), or a handle region (RBF) is displaced by the parameter. In case of RBFs, e.g., the first parameter corresponds to the vertical displacement of the top roof handle region.

5.3.3 *Fitness Function Evaluation*

We evaluate the individual designs based on a fitness function $f_{\text{fit}} : \mathbb{R}^n \rightarrow \mathbb{R}$, that maps a given parameters vector $\boldsymbol{g} \in \mathbb{R}^n$ to a fitness value. We combine two different performance values: $f_{\text{fit}}(\boldsymbol{g}) = w_d D(\boldsymbol{g}) + w_v V(\boldsymbol{g})$, where $D(\boldsymbol{g})$ is the aerodynamic drag as computed by the CFD simulation and $V(\boldsymbol{g})$ is an additional volume term to prevent the optimization from producing overly flat shapes. We compute the aerodynamic drag $D(\boldsymbol{g})$ by solving the incompressible laminar Navier-Stokes equations using the SIMPLE algorithm as provided by the OpenFOAM (2012) CFD

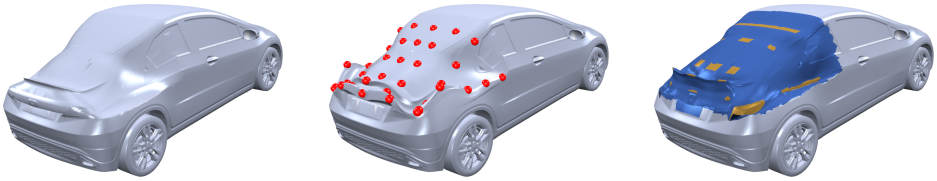


Figure 5.5: Best designs using FFD (left), DM-FFD (center), and RBFs (right).

toolkit. The initial simulation mesh contains 2M points and 3.2M cells. In this simple scenario, we only deform the surface mesh and regenerate the volume mesh for each individual of the population (see part II for a scenario directly deforming an initial volume mesh). Therefore, the actual mesh complexity varies during the optimization process. For each of the 15 offspring individuals the CFD simulation runs in parallel on 4 processor cores using OpenMPI (Gabriel et al. 2004) in a cluster environment with systems containing two 2.4GHz Quad Core Xeon processors with 24GB RAM. We compute the volume term as $V(\boldsymbol{g}) = 1/\left\|\boldsymbol{b}_{\max} - \boldsymbol{b}_{\min}\right\|$, where $\boldsymbol{b}_{\max}$ and $\boldsymbol{b}_{\min}$ are the maximum and minimum points of the bounding box of deformable object points. In order to preserve the volume in an effective manner we choose $w_d = 1$ and $w_v = 50$ as weights.

5.3.4 *Results*

We illustrate the progression of the best solution fitness and of the step size in figure 5.6. For each of the deformation methods we run the optimization for 100 generations. We show examples of the designs with the best fitness values for each deformation method in figure 5.5. We note that the resulting deformed cars have no practically relevant background and only serve for benchmarking purposes. The overall run time is approximately two weeks for each method. As can be seen from figure 5.6 (left), both DM-FFD and RBF yield a better solution fitness than FFD. However, as becomes clear from figure 5.6 (right), the step size of FFD is not as close to convergence as for the other methods, which might be an explanation for the higher fitness value. However, the step sizes of all optimizations did not fully converge within the number of iterations performed. This is mainly due to the initial step size being too small—a common problem when dealing with an unknown objective function.

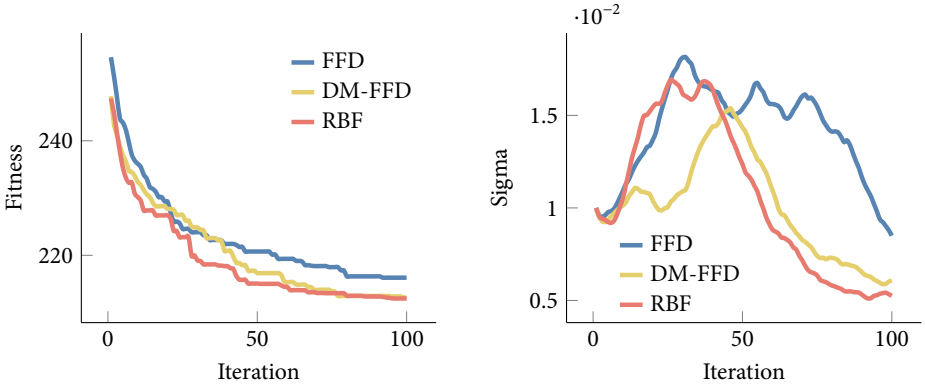


Figure 5.6: Progression of best solution fitness (left) and step size (right) during optimization. Initial fitness values are equal for all methods and therefore not included here.

5.4 CONCLUSION

From our application benchmark we can draw two major conclusions: First, we can affirm that in contrast to lattice-based free-form deformation methods radial basis functions are significantly easier and more flexible to set up while providing equivalent or better results. The manual adaptation of the control lattice to the shape of the model takes a significant amount of time, while selecting the handle regions for the RBF method is rather fast and straightforward. Our second conclusion is that the slightly better results of both direct manipulation methods seems to confirm the importance of a *strong coupling* between the optimization parameters—the genotype—and resulting phenotype within an evolutionary optimization process.

Even though we heavily constrained the range of parameters in the optimization and we took the change of volume into account for fitness evaluation the resulting optimized shapes do not directly provide design alternatives ready for production. To a certain degree, this is due to the simple but inaccurate volume term used in our benchmark. For a practical scenario, we highly recommend to use a more accurate volume computation. Nevertheless, the results essentially confirm our hypothesis that maintaining constraints through penalty terms in the fitness function is error-prone at best and requires time-consuming experiments to provide usable results. Therefore, we feel that additional constraints yielding more meaningful results should be directly integrated into the deformation, as we explore in the third part of this thesis.

However, we conclude this chapter—and thereby the first part of this thesis—by revisiting the research question set out for this part: What is a good shape deformation technique for design optimization? Considering the results of our application-oriented benchmark, the results of our synthetic benchmarks of chapter 4, as well as general criteria for applicability in design optimization we propose the following answers:

1. Applicability: Since the geometry representations encountered in design optimization widely vary, e.g., from simple triangle surface meshes to complex polyhedral volume meshes, the deformation method should be able to transparently deal with such differing representations. This is a strong argument for employing *space deformation* methods.

2. Robustness: Even though the volumetric meshes used for simulation-based design optimization generally have rather strict requirements on the mesh quality, this is not necessarily true for the design prototypes created during optimization. Especially when performing the optimization in a fully automatic manner it is neither always possible nor strictly required that the prototypes are free of any artifacts. Therefore, the deformation method should be able to robustly deal with defects in the input geometry. Again, this is a strong argument for the use of *space deformation* techniques.

3. Flexibility: Throughout our benchmarks, we observe that the need to generate and maintain complex control structure such as a FFD control grid or a coarse bounding cage poses a significant burden with regards to the seamless and flexible application of the method. Especially in case of adaptive refinement and precise satisfaction of user-defined constraints a *kernel-based* method such as RBFs offers significant advantages.

4. Quality: Smoothness and fairness of deformed surfaces are a primary concern in creating design variations. To this end, the use of *triharmonic* radial basis functions provides high quality results.

5. Performance: The performance of a deformation method is only a minor factor in the overall optimization run-time, especially in case of simulation-based optimization loops involving computationally expensive volume mesh generation steps and physics simulations. Therefore, run-time performance of the deformation method is not a primary selection criterion.

In summary, we conclude that:

> A good shape deformation technique for design optimization is a kernel-based space deformation method with high deformation quality such as provided by global triharmonic radial basis functions.

Consequently, we continue the second part of this thesis by investigating how to apply and improve RBF-based shape deformation techniques for their use in design optimization.

PART II

ADVANCED RBF DEFORMATION TECHNIQUES

In this part, we investigate the second research question of this thesis: *How to apply and improve existing techniques*? More specifically, we concentrate on the application and improvement of RBF deformation techniques for their use in design optimization. To this end, we first introduce a unified framework for combined surface and volume deformation according to an updated CAD geometry (chapter 6). We then compare our framework to several other state-of-the-art techniques (chapter 7), investigate advanced linear solvers to boost performance, and we examine techniques to prevent self-intersections in the resulting meshes.

CHAPTER 6

UNIFIED RBF DEFORMATION FRAMEWORK

We begin this chapter with an analysis of the limitations of the design optimization process presented in the previous chapter 5. Based on this analysis, we present a flexible and powerful solution to these shortcomings. By exploiting the versatility of our kernel-based RBF deformations, we develop a unified framework for a more generalized design optimization process that allows for the combined deformation of surface and volume meshes according to updated parameters in an initial CAD-based prototype.

6.1 INTRODUCTION

The passenger car design optimization loop considered in the previous chapter 5 represents a typical and widely used optimization procedure. However, this approach has at least two essential limitations: First of all, the complete process is essentially surface-based, i.e., it starts with a polygonal surface mesh as input and for the setup of the deformation technique, it then performs the optimization on the surface by deforming the initial design, and the outcome is a deformed surface mesh, too. In contrast, as already outlined in chapter 1, the more general starting point of the product development process is an initial prototype created using a parametric CAD modeling tool. In consequence, this means that there is no direct connection between the design resulting from the optimization process and its original prototype. The task of retrieving a CAD-based representation from the mesh-based design is a non-trivial challenge of its own and poses a serious obstacle to transferring results obtained during optimization back to the general product development process.

The second drawback is that the process involves and heavily relies on costly and complicated mesh generation steps. After the shape deformation step an additional mesh generation step creates a new volumetric simulation mesh for each individual of the current population, see figure 6.1. This essentially prevents the implementation of fully automatic and concurrent optimization processes,

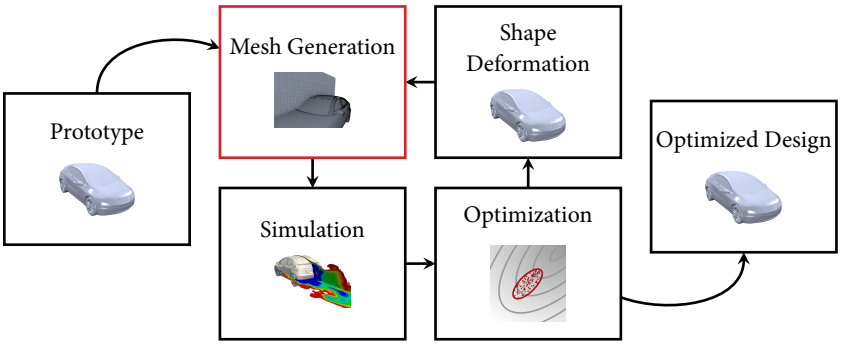


Figure 6.1: The surface-based design optimization process requires time-consuming mesh generation steps for the initial design and *for each* design variation created.

since—depending on the complexity of the input geometry—the meshing step eventually requires manual interaction of an expert.

As for addressing the first issue—the lack of a relation between optimized design and initial prototype—an alternative is not to use a discrete polygonal surface mesh as target representation during optimization, but to create design variations based on changes of the parameters in the initial CAD geometry. Such a CAD-based approach to evolutionary design optimization has been presented by König and Wintermantel (2004). We illustrate such a process in figure 6.2. However, as König and Wintermantel (2004) freely admit, this approach is still limited by the need for automatic mesh generation steps in each iteration of the optimization loop.

In this chapter, we propose a unified framework for design optimization based on radial basis functions that overcomes both of these issues. Our framework allows for the implementation of a design optimization process that allows to change an initial CAD-based design prototype without the need for costly re-meshing of the volumetric simulation meshes during the optimization process. We accomplish this goal by employing a two step procedure: We first compute updated surface nodes according to an updated CAD geometry and then use this updated surface as input to a volume deformation. For both of these steps, we exploit the flexibility of our RBF-based space deformations. This way, we only require a single initial mesh generation step and we can then perform the optimization loop in a fully automatic and parallel manner, as illustrated in figure 6.3. Furthermore, our approach offers the following compelling advantages: It is easy to understand and straightforward to implement; it is applicable to tetrahedral, hexahedral, or general polyhedral

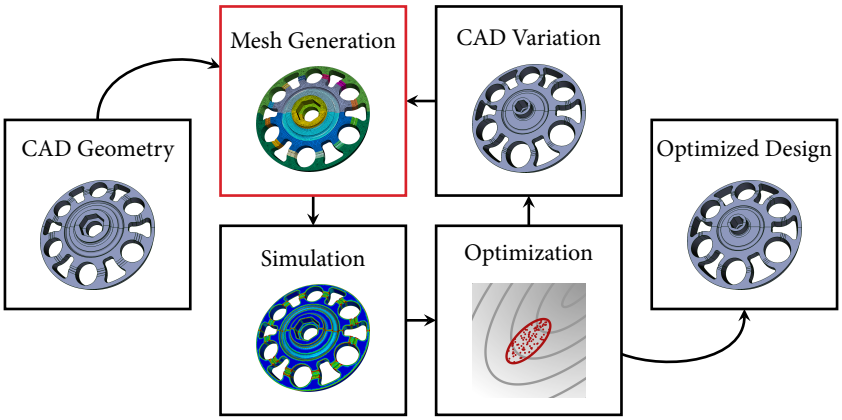


Figure 6.2: A CAD-based optimization loop relying on automated mesh generation.

meshes using a single code base; and finally, it more robustly achieves higher quality results. We note, however, that our framework is directly motivated by the work of Staten et al. (2011). We build upon their benchmarks, extend upon their ideas, and contribute to their comparisons through our evaluation in chapter 7.

The general setting for a CAD-based design optimization task is as follows: Given an initial CAD model $\mathcal{G}$ and a volumetric mesh $\mathcal{V}$ within that geometry, we generate a shape variation $\mathcal{G}'$ by changing the geometric embedding of $\mathcal{G}$ while keeping its topology fixed. The shape deformation technique then adapts the volume mesh $\mathcal{V}$ such that the updated version $\mathcal{V}'$ conforms to the updated boundary surface $\mathcal{G}'$. Analogously to the geometric changes in the CAD model, we only update the geometric embedding of $\mathcal{V}$ (i.e., its node positions) in this process, while keeping the mesh topology (i.e., its connectivity) fixed. See figure 6.3 for an illustration of a CAD-based design optimization loop based on shape deformation techniques. Within this scenario we can actually break down the deformation task into two separate steps: First, we have to compute the updated surface nodes matching the updated CAD model $\mathcal{G}'$. Second, we compute the locations of interior volume nodes in accordance to the updated surface nodes.

6.2 Surface Deformation

In this section, we describe how to compute the updated surface nodes according to a modified CAD geometry, see figure 6.4 for an illustration of the process.

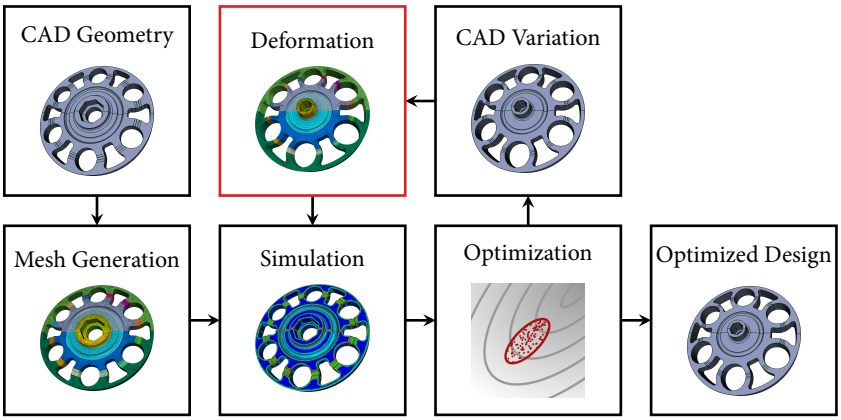


Figure 6.3: A CAD-based design optimization loop relying on shape deformation.

Since in our setting the topology of the CAD surface $\mathcal{G}$ stays constant, there is a one-to-one correspondence between the faces, curves, and corner vertices of $\mathcal{G}$ and $\mathcal{G}'$. Staten and colleagues (Staten et al. 2011) exploit this fact for deforming curves: Let $\boldsymbol{y}: \mathbb{R} \to \mathbb{R}^3$ be a *curve* in the initial geometry of the initial geometry $\mathcal{G}$, e.g., the boundary of a surface patch. Then for each *curve node* $\boldsymbol{n}_i \in \mathcal{V}$, i.e., a node of the initial mesh lying on a feature curve, they assume its parameter value u with $\boldsymbol{n}_i = \boldsymbol{y}(u)$ to be constant during the transformation and compute the deformed node as $\boldsymbol{n}_i' = \boldsymbol{y}'(u)$, where $\boldsymbol{y}' \subset \mathcal{G}'$ is the deformed curve corresponding to $\boldsymbol{y} \subset \mathcal{G}$. The deformed curve nodes then act as boundary constraints for deforming the surface nodes, which Staten et al. performed using either mesh smoothing or the weighted residual technique. Both, however, lead to a certain amount of distortion or even inverted surfaces triangles, which in turn negatively impact the volume deformation. We note, however, that Staten and colleagues did not focus on the surface deformation aspect but on a comparison of volume deformation techniques.

In our approach, we extend the curve deformation idea of Staten and colleagues to the surface case: For each surface node $\boldsymbol{s}_i$ we find its corresponding face $\boldsymbol{f}: \Gamma \to \mathbb{R}^3$ and its (u, v)-parameters, and define the deformed surface node as the corresponding point on the deformed face $\boldsymbol{f}'(u, v)$. We first describe how to find the (u, v)-parameters of a surface node $\boldsymbol{s}_i$, before explaining the actual mapping from $\boldsymbol{f}$ to $\boldsymbol{f}'$. Note that the explicit computation of (u, v)-parameters is only required in case this information is not provided by the meshing process.

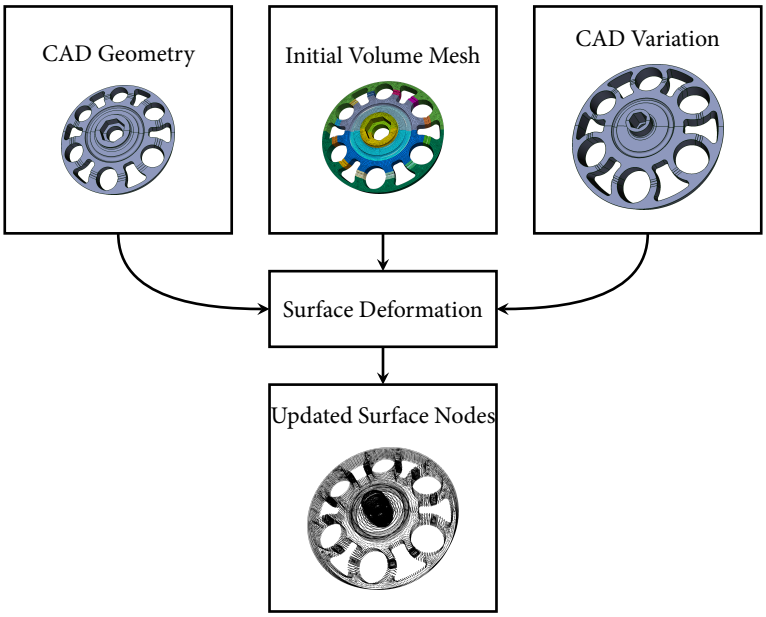


Figure 6.4: Surface deformation procedure. Based on the initial CAD geometry, the initial surface mesh, and the updated CAD geometry, the surface deformation determines the updated surface node locations matching the updated CAD geometry.

Given a surface node $\boldsymbol{s}_i$, finding its corresponding face $\boldsymbol{f}$ and (u, v) parameters in theory amount to projecting $\boldsymbol{s}_i$ onto each face $\boldsymbol{f}_k \in \mathcal{G}$ and selecting the closest one. Although most CAD kernels (Open CASCADE (2012) in our case) provide this functionality, in practice these projections are both computationally expensive and numerically instable for complex, trimmed faces. We address both problems by densely sampling the CAD surface $\mathcal{G}$, which requires only robust and efficient evaluations and results in samples $\boldsymbol{r}_j = \boldsymbol{f}_k(u_j, v_j)$. For each surface node $\boldsymbol{s}_i$ we then find its closest sample point $\boldsymbol{r}_j$ and project $\boldsymbol{s}_i$ onto $\boldsymbol{f}_k$ with (u_j, v_j) as initial guess. We perform this search efficiently through space partitioning: when storing the samples $(\boldsymbol{r}_j, u_j, v_j, k)$ in a kD-tree (Samet 1990), finding the closest sample for a given $\boldsymbol{s}_i$ takes less than 0.01ms for a highly dense sampling of about 15M points. In our experiments, this approach drastically improved the efficiency and robustness of the projections.

After finding the face $\boldsymbol{f}$ and the (u, v) parameters, we need to move the node $\boldsymbol{s}_i$ to the corresponding point on the deformed CAD face $\boldsymbol{f}'$. This part is more

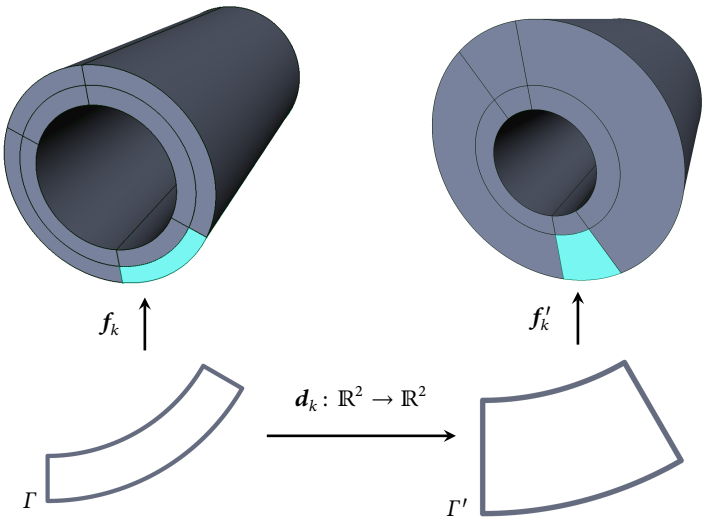


Figure 6.5: Overview of the parameter deformation process for one face.

challenging than for the curve case, since the geometric embedding of a face $\boldsymbol{f} : \Gamma \to \mathbb{R}^3$ can change in two ways: (i) an adjustment of its geometric parameters, e.g., spline control points or cylinder radii, and (ii) a change of its parameter domain Γ, e.g., due to adjusted trimming curves, see figure 6.5. While (i) simply amounts to evaluating $\boldsymbol{f}'$ instead of $\boldsymbol{f}$, (ii) requires to deform the parameter values $(u, v) \in \Gamma$ to $(u', v') \in \Gamma'$.

In order to deform the parameter values to the updated parametric domain, we exploit the versatility of our approach and construct a 2D RBF deformation function $\boldsymbol{d}_k : \mathbb{R}^2 \to \mathbb{R}^2$ for each face $\boldsymbol{f}_k$ of the CAD model (see figure 6.5). To this end we uniformly sample the (u, v)-boundary curves of the faces $\boldsymbol{f}_k$ and $\boldsymbol{f}'_k$, resulting in 2D point samples $\{\boldsymbol{r}_1, \dots, \boldsymbol{r}_n\} \in \Gamma$ and $\{\boldsymbol{r}'_1, \dots, \boldsymbol{r}'_n\} \in \Gamma'$. Constructing a suitable 2D RBF warp requires only minor changes to the 3D formulation of section 3.5. In contrast to the 3D case, the 2D biharmonic basis function $\varphi_2(r) = r^2 \log(r)$ is differentiable at the center and therefore smooth enough for our scenario. The polynomial part consists of the basis $\{\pi_1, \pi_2, \pi_3\} = \{x, y, 1\}$, and the coefficients $\boldsymbol{w}_j, \boldsymbol{q}_k$ are two-dimensional. With these changes, and replacing $\boldsymbol{h}_i$ by $\boldsymbol{r}_i$, we solve a linear system analogous to equation (3.10) for computing the two-dimensional

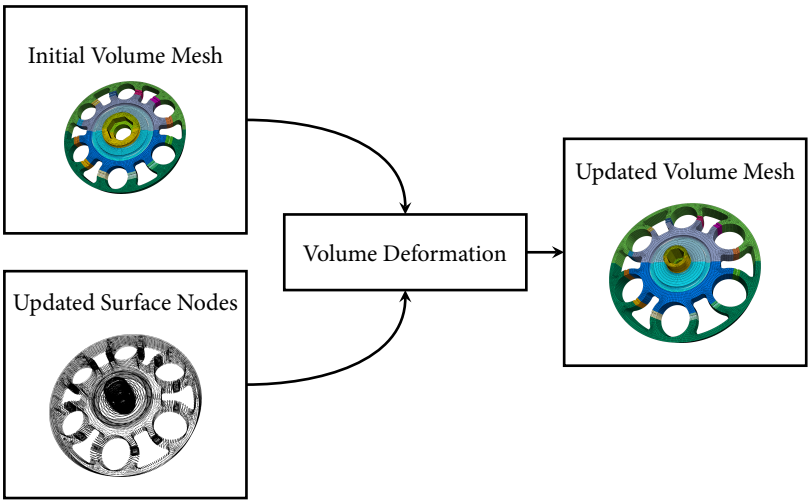


Figure 6.6: Volume deformation procedure. Based on the initial volume mesh and the updated surface node locations the volume deformation component determines an updated volume mesh matching the CAD geometry.

RBF warp $\boldsymbol{d}_k$. After performing the parameter warp $(u', v') = (u, v) + \boldsymbol{d}_k(u, v)$ we compute the deformed surface node as $\boldsymbol{s}_i' = \boldsymbol{f}_k'(u', v')$.

6.3 VOLUME DEFORMATION

In this section, we describe how to deform volume meshes using our RBF approach. The input for the volume deformation consists of three sets of mesh vertices: *Surface nodes* $\{\boldsymbol{s}_1, \dots, \boldsymbol{s}_m\}$ and interior *volume nodes* $\{\boldsymbol{v}_1, \dots, \boldsymbol{v}_n\}$ of the initial mesh $\mathcal{V}$, as well as the updated *surface nodes* $\{\boldsymbol{s}_1', \dots, \boldsymbol{s}_m'\}$ of $\mathcal{V}'$, where the $\boldsymbol{s}_i$ and $\boldsymbol{s}_i'$ conform to the CAD geometries $\mathcal{G}$ and $\mathcal{G}'$, respectively. The goal is to find updated *volume node* positions $\{\boldsymbol{v}_1', \dots, \boldsymbol{v}_n'\}$, such that the element quality of the deformed mesh $\mathcal{V}'$ is as good as possible.

As already motivated in section 3.5, we can treat volume deformation as a scattered data interpolation problem: We search for a function $\boldsymbol{d}\colon \mathbb{R}^3 \to \mathbb{R}^3$ that (i) exactly interpolates the prescribed boundary displacements $\boldsymbol{d}(\boldsymbol{s}_i) = (\boldsymbol{s}_i' - \boldsymbol{s}_i)$ and (ii) smoothly interpolates these displacements into the volume mesh interior. We use RBFs to solve this type of problem and define the deformation function $\boldsymbol{d}(\boldsymbol{x})$ as a linear combination of radially symmetric kernel functions $\varphi_j(\boldsymbol{x}) = \varphi(\|\boldsymbol{x} - \boldsymbol{c}_j\|)$,

located at centers $\boldsymbol{c}_j \in \mathbb{R}^3$ and weighted by $\boldsymbol{w}_j \in \mathbb{R}^3$, plus a linear polynomial to guarantee linear precision:

$$\boldsymbol{d}(\boldsymbol{x}) \;=\; \sum_{j=1}^{m} \boldsymbol{w}_j \varphi_j(\boldsymbol{x}) \;+\; \sum_{k=1}^{4} \boldsymbol{q}_k \pi_k(\boldsymbol{x})\,, \tag{6.1}$$

where $\{\pi_1, \pi_2, \pi_3, \pi_4\} = \{x, y, z, 1\}$ is a basis of the space of linear trivariate polynomials, weighted by coefficients $\boldsymbol{q}_k \in \mathbb{R}^3$. Since we already extensively discussed the choice of a suitable basis function in section 3.5, we do not repeat it here and simply state that we choose basis functions $\varphi(r) = r^3$ in order to obtain a deformation function that is triharmonic, globally C^2 smooth, and thereby the lowest-order polyharmonic RBF suitable for our application.

Satisfying the interpolation constraints $\boldsymbol{d}(\boldsymbol{s}_i) = (\boldsymbol{s}_i' - \boldsymbol{s}_i)$ amounts to placing RBF kernels at the constraint positions (i.e., $\boldsymbol{c}_j = \boldsymbol{s}_j$) and finding the coefficients $\boldsymbol{w}_j$ and $\boldsymbol{q}_k$ by solving the $(m + 4) \times (m + 4)$ linear system

$$\boldsymbol{\Phi} \cdot \boldsymbol{W} \;=\; \boldsymbol{S}, \tag{6.2}$$

where

$$\boldsymbol{\Phi} \;=\; \begin{bmatrix} \varphi_1(\boldsymbol{s}_1) & \cdots & \varphi_m(\boldsymbol{s}_1) & \pi_1(\boldsymbol{s}_1) & \cdots & \pi_4(\boldsymbol{s}_1) \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \varphi_1(\boldsymbol{s}_m) & \cdots & \varphi_m(\boldsymbol{s}_m) & \pi_1(\boldsymbol{s}_m) & \cdots & \pi_4(\boldsymbol{s}_m) \\ \pi_1(\boldsymbol{s}_1) & \cdots & \pi_1(\boldsymbol{s}_m) & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \pi_4(\boldsymbol{s}_1) & \cdots & \pi_4(\boldsymbol{s}_m) & 0 & \cdots & 0 \end{bmatrix},$$

$$\boldsymbol{W} \;=\; \left[\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m, \boldsymbol{q}_1, \ldots, \boldsymbol{q}_4\right]^{\mathrm{T}},$$

and

$$\boldsymbol{S} \;=\; \left[(\boldsymbol{s}_1' - \boldsymbol{s}_1), \ldots, (\boldsymbol{s}_m' - \boldsymbol{s}_m), \boldsymbol{0}, \ldots, \boldsymbol{0}\right]^{\mathrm{T}}.$$

After solving equation (6.2) we can compute the deformed mesh $\mathcal{V}'$ by simply evaluating the RBF deformation at each volume node: $\boldsymbol{v}_i' = \boldsymbol{v}_i + \boldsymbol{d}(\boldsymbol{v}_i)$. This part can easily be parallelized and therefore is highly efficient. The computationally most expensive part is the solution of the linear system of equation (6.2), which is dense due to the global support of $\varphi(r)$. We discuss the performance and the scalability of our method in section 7.5.

6.4 SUMMARY

In this chapter, we presented a unified framework for combined surface and volume deformation according to a modified CAD geometry based on polyharmonic radial basis functions. The important contribution of this framework is the insight that by exploiting the versatility of our kernel-based RBF deformations, we are able implement fully automatic simulation-based design optimization process based on changes in an initial CAD-based design prototype. Furthermore, our deformations are easy to implement and a single code base is applicable to arbitrary surface and volume mesh deformations as well as the deformation of parametric coordinates within the CAD model.

However, it is important to note that our framework also has its limitations. A basic assumption of our approach is that the topology of the CAD prototype stays fixed. While this is a rather conservative assumption for design optimization in general, it nevertheless limits the space of shapes that we can explore during the optimization process. Similarly, the search space is limited to that of the CAD model while in some scenarios the shape space of the discrete mesh might be preferable. Furthermore, a reliable and automatic assessment of the suitability of the deformed meshes for simulation can be difficult. However, in the next chapter, we evaluate our framework by comparing our results to those reported by Staten et al. (2011). We show that our approach also leads to higher quality elements in the deformed meshes, thereby allowing for a wider range of deformations during the optimization process.

CHAPTER 7

FRAMEWORK EVALUATION

The major goal of this chapter is to provide an evaluation of our proposed unified deformation framework. To this end, we compare our framework to other state-of-the-art mesh deformation techniques in terms of quality, performance, as well as scalability. We demonstrate that our approach more robustly achieves higher quality results. Furthermore, we extend our approach by analyzing how to explicitly prevent inverted mesh elements by successively splitting the deformation into smaller steps. Finally, we investigate the performance of different linear solvers as well as an incremental least squares solver for the sake of improved scalability.

A fundamental requirement for a shape deformation technique to be usable in design optimization is to preserve the element quality as much as possible, thereby allowing for as large as possible geometric changes before inevitably requiring some remeshing due to element inversion. Staten and coworkers recently proposed and evaluated several shape deformation techniques, which they compared with respect to computational performance and element quality on different tetrahedral and hexahedral meshes (Staten et al. 2011). In this chapter, we build on their results and contribute to their benchmarking by comparing our RBF-based approach to the most successful techniques reported in their comparison.

7.1 SURFACE DEFORMATION QUALITY

As already noted in section 6.2, the quality of the surface deformation is of particular importance since it essentially constitutes an upper bound on the mesh element quality obtainable during volume deformation. We compare our surface deformation to that of Staten et al. (2011) in figure 7.1. In contrast to the method of Staten et al. (2011), our approach produces high quality surface deformations of minimal parametric distortion. Thanks to its meshless nature, we can apply our method to all kinds of faces, such as simple non-trimmed rectangular faces, trimmed faces with curved boundaries, as well as faces with trimmed holes (see the Bore and Pipe examples in the next section).

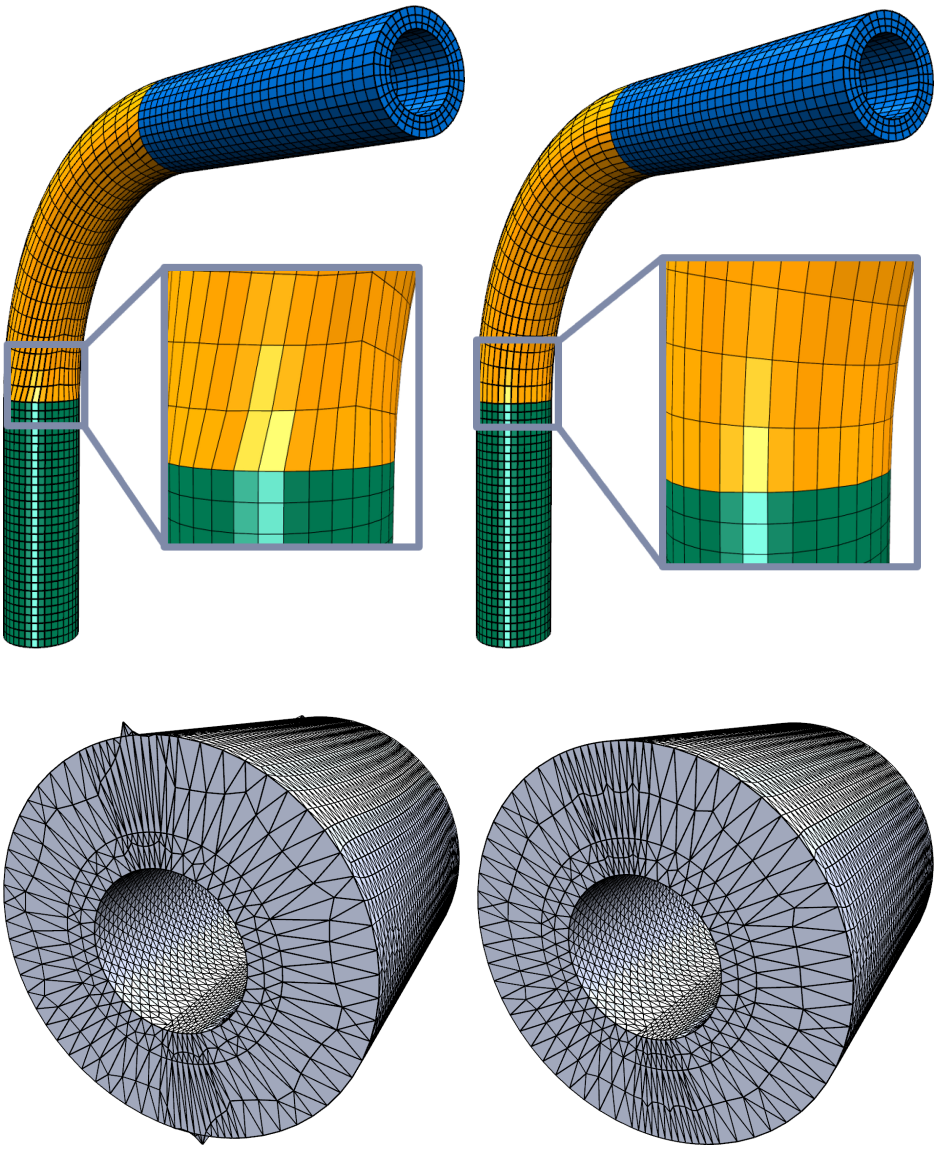


Figure 7.1: Comparison of element quality in the deformed surface mesh. While there are distortions in the meshes of Staten et al. (2011) (left), our surface (right) is perfectly aligned to the updated CAD geometry. Note that distortions in the front of the Bore model (bottom row) are due to the irregularity of the surface triangulation and not the deformation.

7.2 VOLUME DEFORMATION QUALITY

In this section, we compare the resulting volume mesh element quality of our mesh deformation technique to the results published in Staten et al. (2011). The examples include meshes of varying topology, including structured and unstructured hexahedral as well as tetrahedral meshes. The complexity of the models ranges from 10-15k vertices for the Bore and Pipe models up to 130k vertices for the Courier model. The Canister model used in Staten et al. (2011) was not available to us. Since Staten et al. provide a detailed description of the different geometric parameters in the CAD models and how they vary them, we do not reproduce them here.

For the sake of a clear representation, we restrict our comparison to those methods that either delivered the best results (FEMWARP and LBWARP), or that have been recommended by the authors for sake of simplicity and efficiency (Simplex). In order to ensure comparability of the results, we also measure element quality based on the minimum scaled Jacobian as described by Knupp (2000). For our RBF volume deformation, we include results for both the original surface node locations from Staten et al. (2011) (denoted RBF) as well as those obtained by our surface deformation (denoted RBF-S).

Following the benchmarks of Staten et al. (2011), we investigate two different types of deformation: *relative* and *absolute* deformation. In the former case, we incrementally update the mesh from the initial design to the full parameter change. In the latter case, we directly deform the initial mesh to the corresponding parameter change. As in the benchmarks of Staten et al. (2011), we use $N = 20$ steps for both types of deformation. In the following subsections, we present detailed results for the individual test cases. In order to give the reader an impression of where in the meshes the element quality becomes particularly low, we present selected cut-views of the deformed volume meshes in figure 7.2. After performing an absolute deformation to the full parameter change on both hexahedral and tetrahedral meshes, we highlight the worst 5% of the elements in red.

7.2.1 *Bore Model*

The change of parameters in the Bore model tests the ability of the different methods to deal with scaling and rotation. An example deformation from the initial mesh to the full parameter change is shown in figure 7.3. As can be seen from this figure, our method is on par with or better than the FEMWARP and

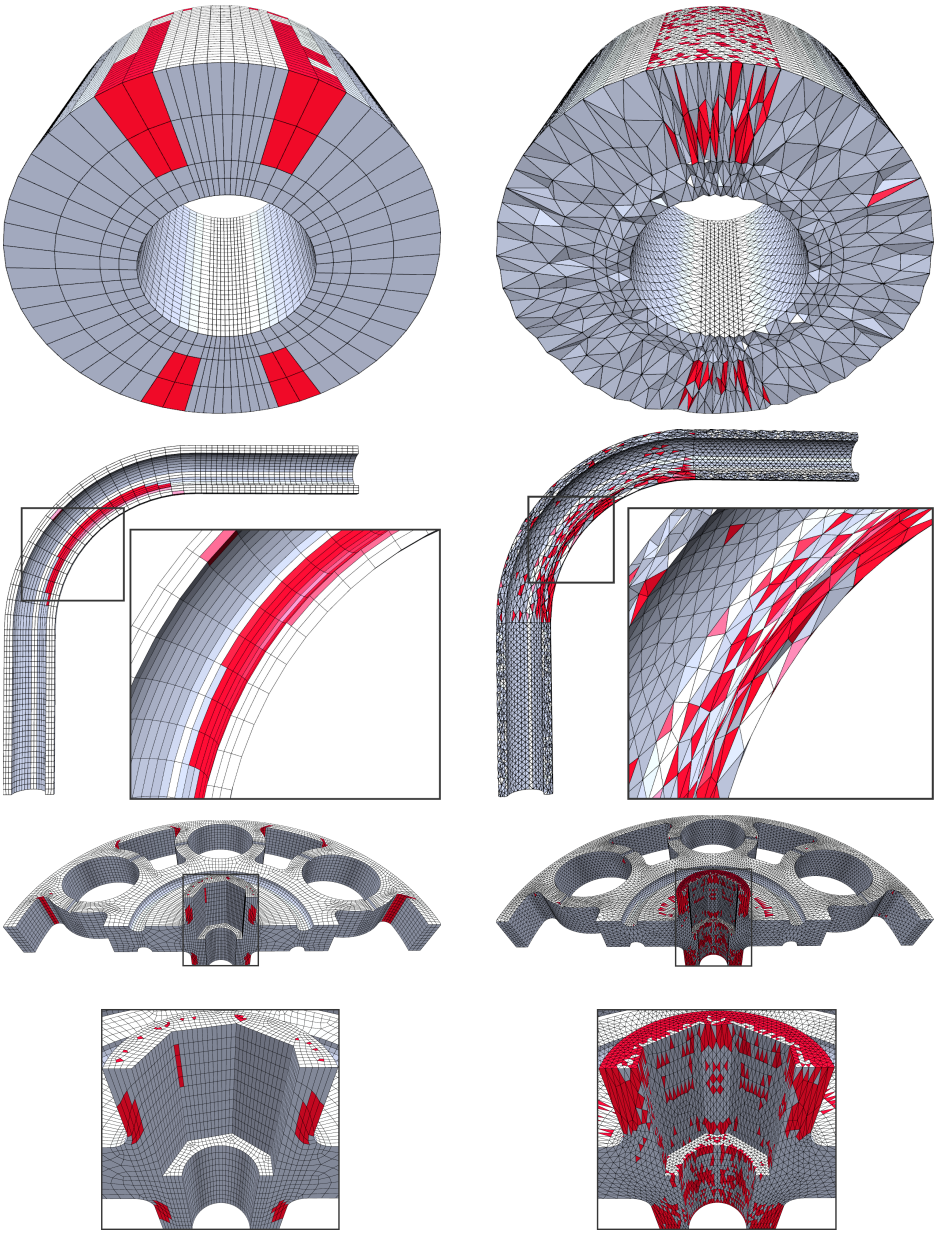


Figure 7.2: Cut views of the hexahedral (left) and tetrahedral (right) meshes for the Bore, Pipe, and Courier models after performing an absolute deformation to the full parameter change. We highlight the worst 5% of the elements in red.

LBWARP methods. It is important to note that the element inversion after 75% parameter change in case of the tetrahedral model is due to a defect in the deformed surface mesh of Staten et al. (2011). By using our more robust surface deformation we are able to perform both the relative and the absolute deformation up to a parameter change of 100% without any inverted elements.

7.2.2 *Pipe Model*

The change of parameters in the Pipe model tests the ability of the different methods to deal with nonlinear stretching. We present the initial and final shapes as well as detailed results in figure 7.4. While our method provides superior results on the hex model, those on the tetrahedral model are comparable. However, in contrast to other methods ours does not result in inverted elements at 95% parameter change for the absolute deformation of the tetrahedral model. Again, the results obtained using our combined volume and surface deformation are superior.

7.2.3 *Courier Model*

The Courier model is the most complex model in our comparison. In contrast to previous examples, the hexahedral mesh of this model is an unstructured one. Especially in case of the absolute tetrahedral mesh deformation, all methods presented in Staten et al. (2011) result in inverted elements as soon as reaching a change of parameter values of 65%. In contrast, our method results in inverted elements only after a parameter change of 75% (see the results in figure 7.5). Unfortunately, due to a mismatch between CAD geometry and initial tetrahedral mesh provided to us, we could not evaluate our surface deformation method for the tetrahedral Courier model.

7.2.4 *Comparison of RBF Deformations*

In table 7.1 we report the resulting element quality for different RBF deformations after performing an absolute deformation to the full parameter change. In all but one case the triharmonic deformation delivers better results than the biharmonic one. The higher-order quadharmonic RBF (using $\varphi_4(r) = r^5$) does not result in noteworthy improvements. Even worse, in case of the Courier hex model it even leads to inverted elements due to numerical instabilities. Investigating the condition number indeed reveals a drastic increase with the order of the basis

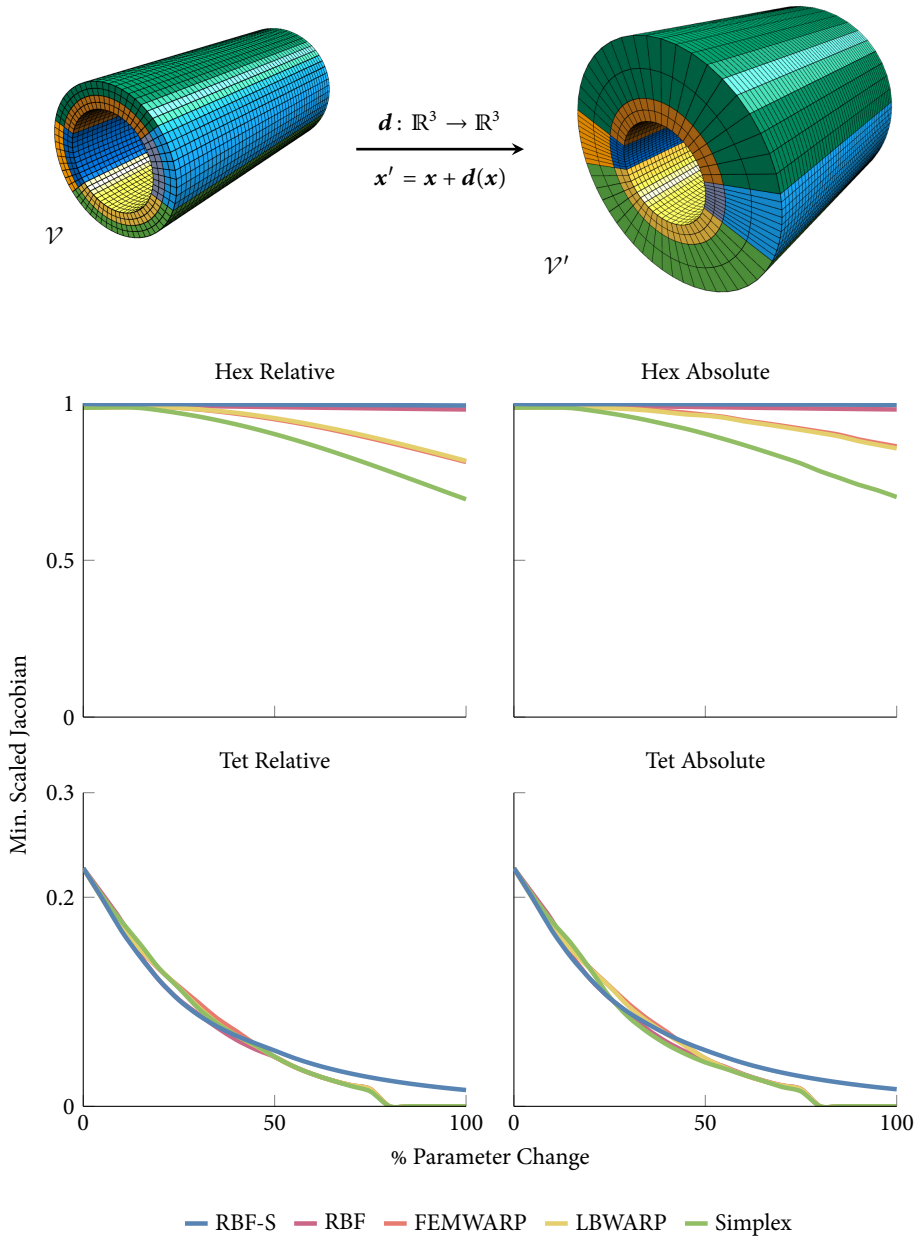


Figure 7.3: Deformation results of the Bore model.

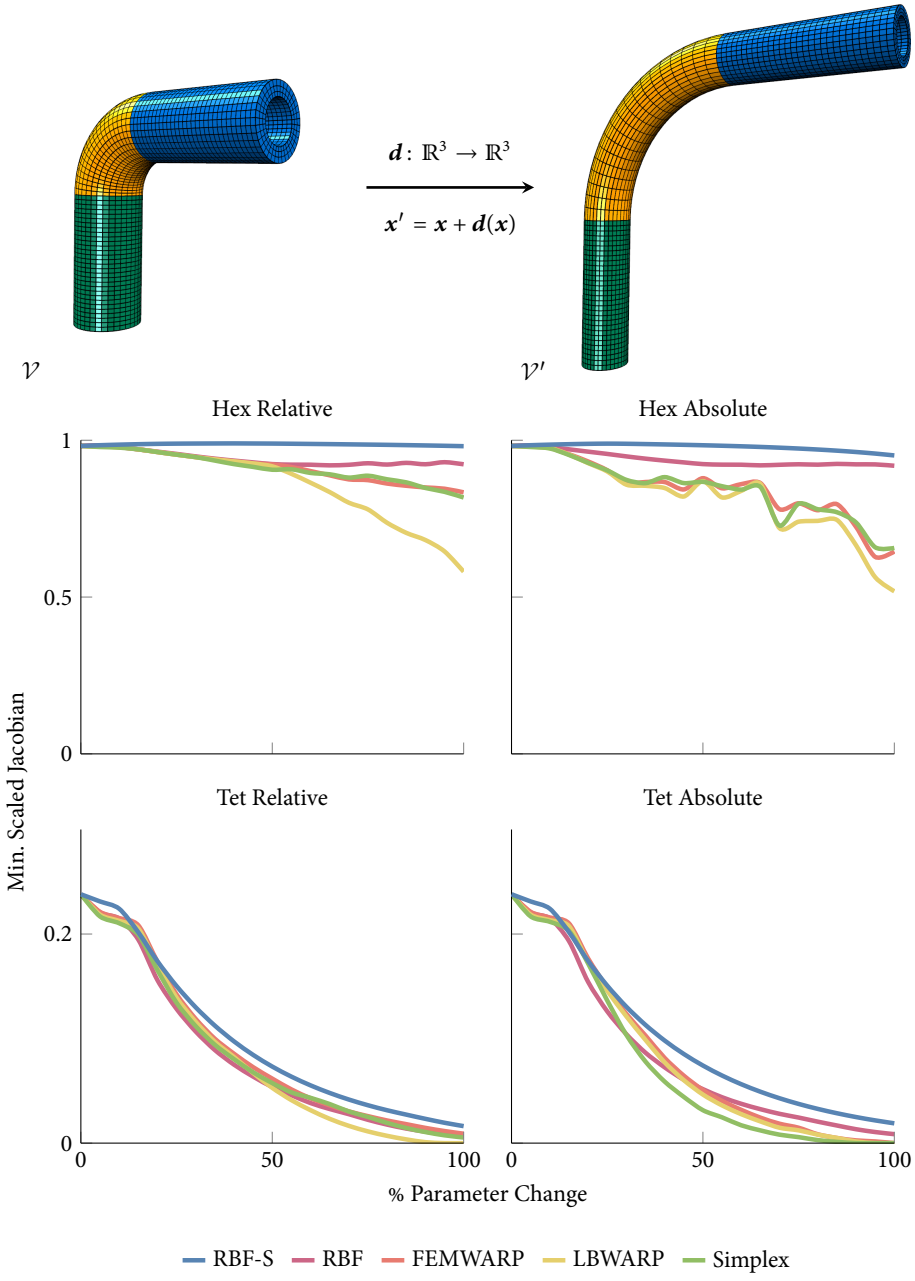


Figure 7.4: Deformation results of the Pipe model.

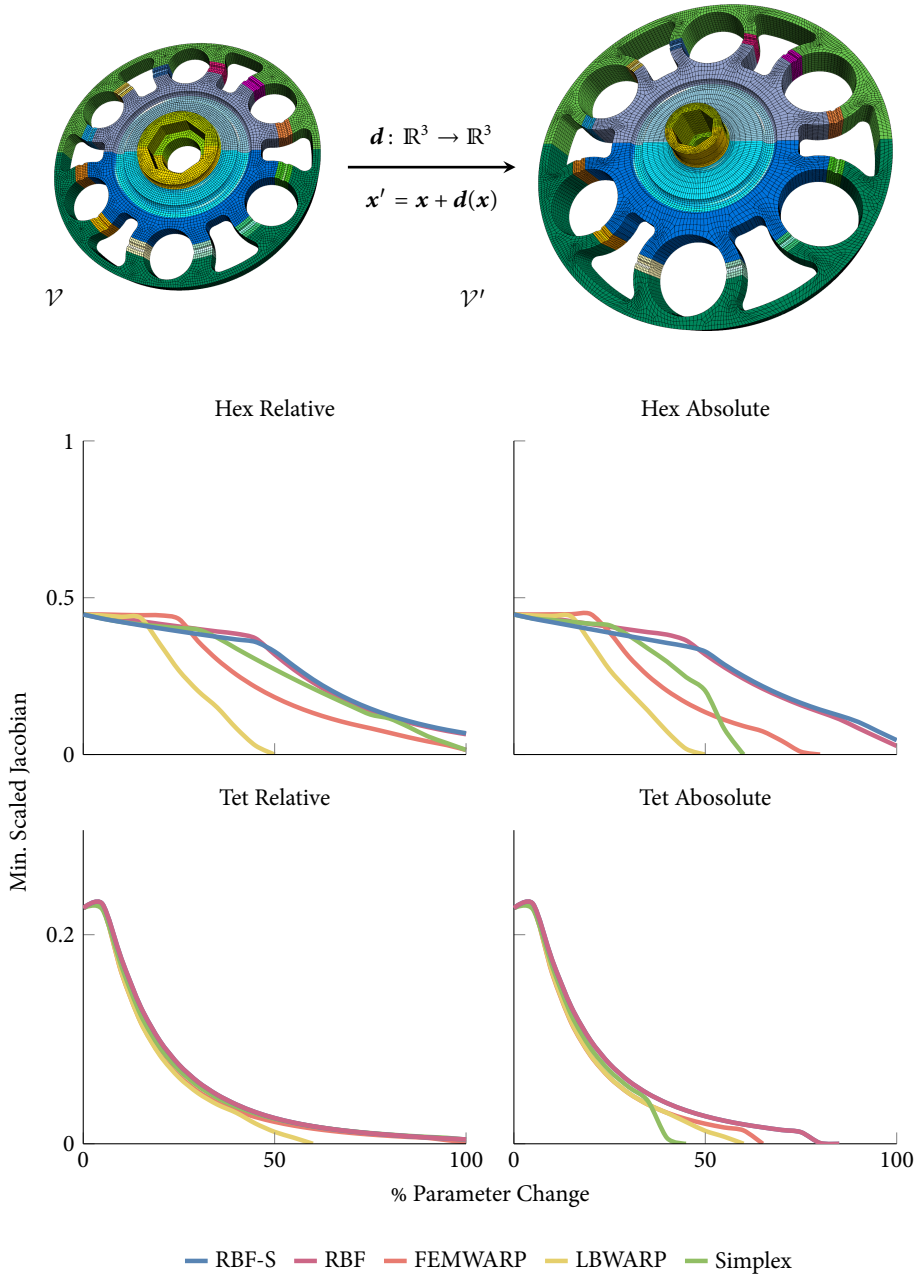


Figure 7.5: Deformation results for the Courier model.

function, being 2.3^6 for the biharmonic, 2.3^{11} for the triharmonic, and 6.3^{15} for the quadharmonic deformation.

7.3 COMPARISON WITH FFD

In this section, we provide an additional evaluation of deformation quality between RBFs and direct manipulation FFD. Even though both techniques allow for deformations of arbitrary volume meshes due to their space-based nature, there are significant differences in the resulting mesh element quality. We present two different test scenarios involving three different mesh types: First, we investigate deformation of unstructured tetrahedral and structured hexahedral meshes of the Pipe model. Second, we present a test-case of the DrivAer aerodynamics performance reference model (Heft et al. 2012) involving a mesh with arbitrary polyhedral elements for CFD computations.

In case of DM-FFD, we generate a uniform control lattice enclosing the complete volume mesh. Unfortunately, the resolution required to satisfy given deformation constraints as precisely as possible is not known in advance and heavily depends on the complexity of the deformation and the geometry to be deformed. To accommodate for this, we investigate different control grid resolutions, namely with 5^3, 10^3, 15^3, and 25^3 control points, to which we refer to as DM-FFD-5/10/15/25 below. Still, the problem of automatic control grid generation is a largely unsolved problem and a serious obstacle for the application of FFD within fully automated optimization procedures.

Table 7.1: A comparison of RBF deformations showing the minimum scaled Jacobian after an absolute deformation to the full parameter change.

	Bore		Pipe		Courier
	Hex	Tet	Hex	Tet	Hex
biharmonic	0.985	0.015	0.92	0.018	0.09
triharmonic	0.995	0.016	0.95	0.019	0.045
quadharmonic	0.995	0.016	0.96	0.017	-0.99

7.3.1 *Pipe Model*

In this section, we compare the resulting mesh quality of RBFs and DM-FFD based on absolute deformation of the unstructured tetrahedral and structured hexahedral Pipe meshes as a representative example. As before, we present the results by means of minimum scaled Jacobian in the volume mesh vs. percentage of parameter change in the CAD model in figure 7.7. The plots show that RBFs better preserve element quality. Note that while the low resolution DM-FFD-5 test case results in more or less reasonable mesh quality, the displacement constraints on the surface mesh are not fulfilled exactly, i.e., the boundary nodes of the volume mesh do not match the updated CAD surface (see figure 7.6, right, for an example). In contrast, higher resolutions are not capable of dealing with large changes due to the increasing locality of the deformation.

7.3.2 *DrivAer Model*

As an application-oriented benchmark, we investigate an exemplary CFD test case for the DrivAer model. We use OpenFOAM (2012) for the CFD setup and generate the volume mesh using the `snappyHexMesh` utility. The resulting polyhedral mesh contains 1.2M cells and 1.6M points. To investigate the resulting mesh quality we use OpenFOAM's `checkMesh` tool, which analyzes general mesh properties, such as connectivity, ordering, and orientation, but also essential mesh quality characteristics, such as cell orthogonality, aspect ratio, and face skewness. For the deformation setup we select three handle vertices on the car roof while keeping the outer boundary of the volume mesh fixed. For the deformation itself we simply lift the handle vertices upwards.

A cut view of the resulting volume mesh and car surface patch is shown in figure 7.8. A summary of results as obtained by OpenFOAM's `checkMesh` utility is given in table 7.2. For a detailed description of the individual mesh checks we refer to the OpenFOAM (2012) documentation. In case of RBFs the volume mesh is still usable and all mesh quality checks succeed. In case of DM-FFD-10 and DM-FFD-15 the meshes are still usable, but the cell orthogonality check warns about one non-orthogonal cell that might spoil the accuracy and/or convergence of the simulation. Furthermore, we note that for more complex deformations the 10^3 and 15^3 resolutions might not be sufficient to satisfy the displacement constraints with acceptable precision. In case of the higher resolution DM-FFD-25 setup

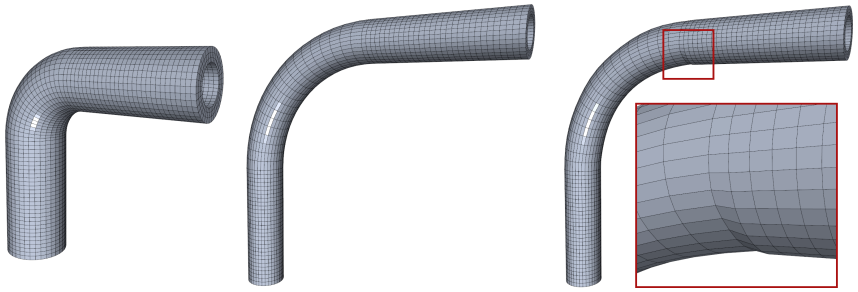


Figure 7.6: Pipe model deformation examples. Left to right: Initial mesh, RBF, DM-FFD-5.

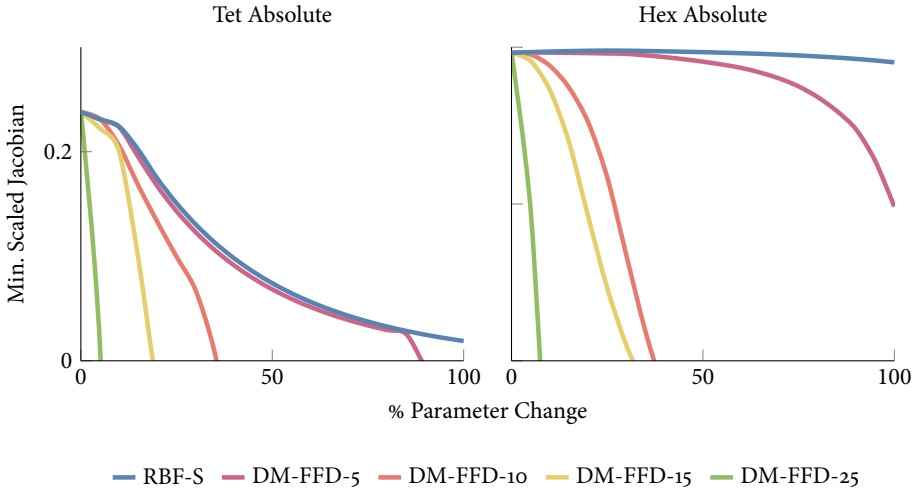


Figure 7.7: Pipe model deformation results. Mesh quality vs. parameter change.

Table 7.2: Results reported by OpenFOAM's `checkMesh` utility. Successful tests are indicated by a ✓, warnings by !, and errors by ✳. Numbers denote the worst quality element in the mesh.

	Aspect Ratio	Orthogonality	Face Skewness	Face Pyramids
Original	6.9 ✓	64.7 ✓	3.4 ✓	✓
RBF	6.6 ✓	68.6 ✓	3.7 ✓	✓
DM-FFD-10	7.0 ✓	71.3 !	3.6 ✓	✓
DM-FFD-15	7.0 ✓	70.7 !	3.4 ✓	✓
DM-FFD-25	2.5e+195 ✳	179.7 ✳	1031.8 ✳	✳

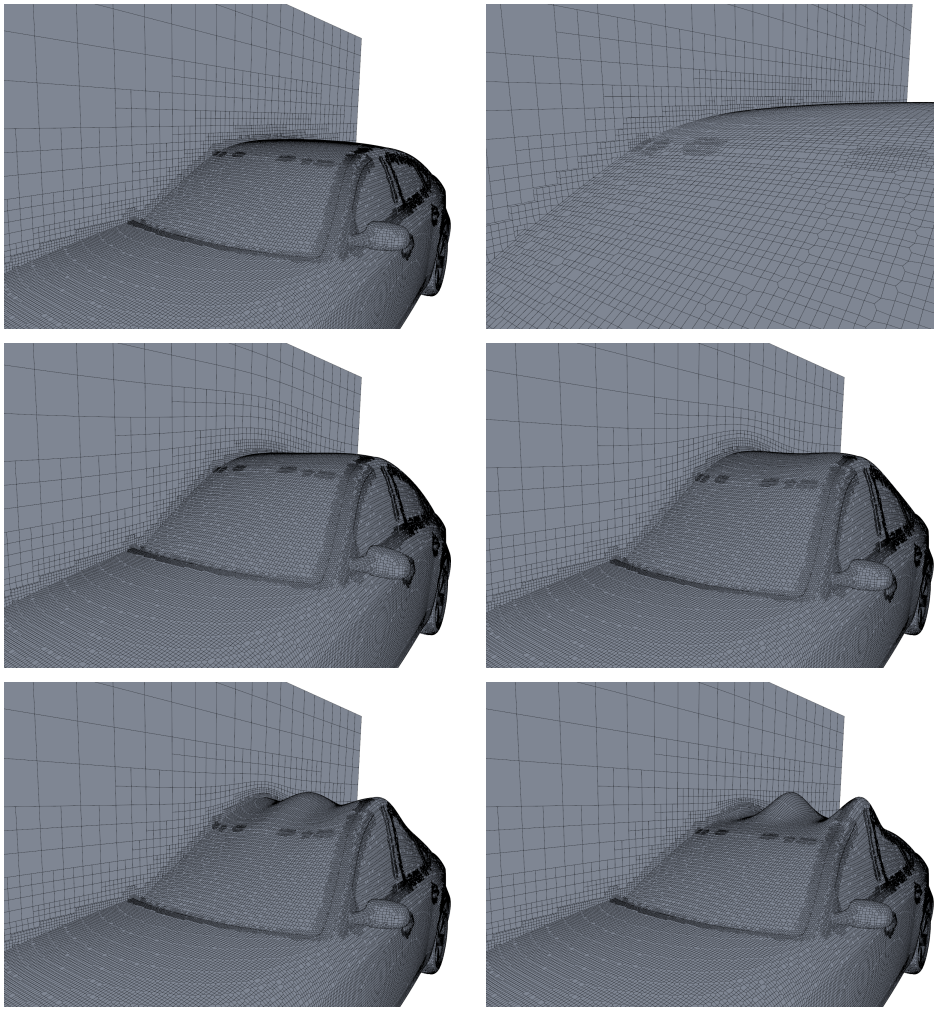


Figure 7.8: Cut-view and car surface patch of the resulting volume mesh after deformation. From top left to bottom right: Original model, zoom to the surface, RBF deformation, DM-FFD with control grid resolutions of 10^3, 15^3, and 25^3.

several mesh quality checks fail and the mesh is no longer usable for simulation at all: The mesh contains 151 high aspect ratio cells, 1353 non-orthogonal faces, 1414 incorrectly oriented face pyramids, and 62 highly skewed faces. For all DM-FFD setups, figure 7.8 again demonstrates the strong dependence of the resulting shape on the chosen control grid resolution.

7.4 Inversion-free Deformation

In this section, we investigate approaches for preventing inverted elements in the deformed mesh. The prevention of self-intersections and element inversions under deformation has also been subject to substantial research (Gain and Dodgson 2001; Angelidis and Singh 2006; Harmon et al. 2011; Shontz and Vavasis 2010; Schüller et al. 2013; Paillé et al. 2015; Kovalsky et al. 2015; Liu et al. 2016). A particularly powerful approach is the construction and integration of a smooth space-time vector field, which theoretically guarantees the absence of intersections and element inversions (Angelidis and Singh 2006; Funck et al. 2006; Esturo et al. 2011).

As already observed by Staten and colleagues, relative deformation tends to better preserve element quality compared to absolute deformation. Therefore, one could be inclined trying to avoid inverted elements by using smaller and smaller steps of relative deformation. However, this approach would rely on the relative geometric parameters in the CAD model to generate the intermediate boundary nodes for the volume deformation. Therefore, preventing element inversions in this manner is rather complicated and computationally expensive.

A simple and efficient approach for preventing inversions is to iteratively split the deformation. Gain and Dodgson (2001) show that a space deformation is guaranteed to be free of self-intersections if (i) it has continuous first partial derivatives and (ii) the determinant of its Jacobian is larger than zero. The first criterion is naturally fulfilled by our smooth RBF deformations. The second one is fulfilled if the displacements to be interpolated are sufficiently small. We therefore use the following procedure to prevent inversions: We initially perform the full deformation. If the deformation results in at least one inverted element, we uniformly split the deformation into n steps, where n is the current iteration. We repeat this process until no more inversions occur. We illustrate this approach schematically in figure 7.9. In the limit case of arbitrarily small displacements our splitting approach is roughly equivalent to vector field-based approaches. In contrast, however, our method avoids the increased computational costs for the construction of the

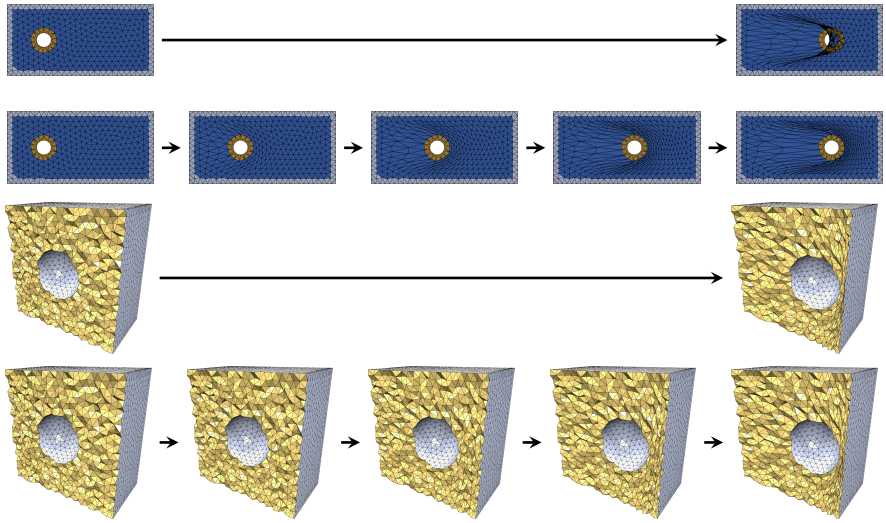


Figure 7.9: Preventing element inversion through splitting in 2D (top) and 3D (bottom). Performing a large deformation in a single step leads to inverted elements while splitting the deformation into smaller steps leads to a mesh without inverted elements.

space-time vector field (Esturo et al. 2011), making our method more practical in the current volume mesh deformation scenario.

We finally emphasize that even a smooth, inversion-free space deformation does not necessarily guarantee the absence of inverted elements. Applying the deformation to all mesh nodes eventually turns the smooth space deformation into a piecewise linear C^0 per-element deformation. Therefore, a fundamental requirement for an inversion-free deformation is a mesh resolution sufficiently high to faithfully represent the deformation field.

7.5 Performance and Scalability

In this section, we investigate the performance and the scalability of our method. As already noted in chapter 6, the computationally most expensive part within our technique is the solution of the linear system of equation (6.2) for the volume deformation. This linear system is dense due to the global support of the chosen radial basis functions $\varphi(r) = r^3$, resulting in an asymptotic complexity of $\mathcal{O}(m^3)$ when using standard solvers for dense linear systems.

While there exist sophisticated techniques for efficiently solving RBF-based systems like equation (6.2), such as multipole expansion, multi-level approximation, or greedy center selection (Carr et al. 2001; Wendland 2010; Michler 2011), this was not necessary in all our test cases. Since most CAD geometries $\mathcal{G}$ and their corresponding volume meshes $\mathcal{V}$ are constructed from multiple solid components, we can simply perform the volume deformation individually for each of these (reasonably small) components. In all our examples this could be done using a standard linear solver, e.g., the LU factorization of the LAPACK library (E. Anderson et al. 1999).

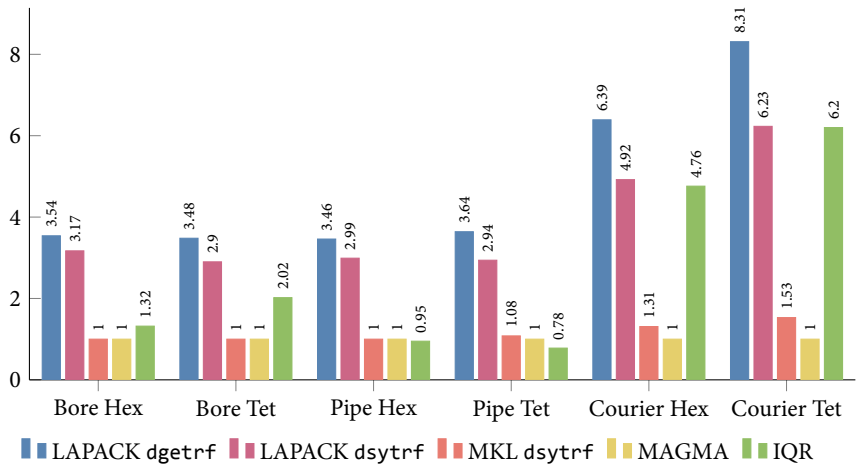
Nevertheless, we investigate in the following how to (i) improve the performance by using efficient implementations of standard solvers and (ii) improve the scalability to larger models using an incremental least squares solver.

Since the linear system in equation (6.2) is symmetric but not positive definite, efficient Cholesky-type solvers are not applicable, leaving us with solvers based on LU and $\mathrm{LDL}^{\mathrm{T}}$ factorizations as the default choices. We compared four solver implementations:

- The general LU decomposition (`dgetrf`) of LAPACK,
- the $\mathrm{LDL}^{\mathrm{T}}$ factorization for symmetric matrices (`dsytrf`) of LAPACK,
- the multi-core $\mathrm{LDL}^{\mathrm{T}}$ decomposition (`dsytrf`) of the Intel Math Kernel Library (MKL) (Intel 2013),
- the GPU-accelerated LU decomposition of MAGMA (Tomov et al. 2010).

The results in table 7.3 show that the performance differs significantly between solvers. The largest differences exist in case of the Courier model where the MAGMA-based solver is up to eight times faster than other implementations. The results for the MKL are almost identical to the MAGMA results for smaller models, but the difference increases with the model size. The comparison between LAPACK's general LU and symmetric $\mathrm{LDL}^{\mathrm{T}}$ factorizations also shows considerable differences. Unfortunately, specialized symmetric factorizations were not available in MAGMA.

Despite the impressive performance improvements the scalability of all these methods is still limited by their time complexity $\mathcal{O}(m^3)$ and memory consumption $\mathcal{O}(m^2)$, preventing their use for high resolution meshes. One can observe, however, that even for densely tessellated models the geometric deformations are still rather



	Bore		Pipe		Courier	
	Hex	Tet	Hex	Tet	Hex	Tet
LAPACK `dgetrf`	8.0	12.0	4.5	8.5	140	483
LAPACK `dsytrf`	7.1	10.0	3.9	6.9	108	362
MKL `dsytrf`	2.3	3.4	1.3	2.5	29	89
MAGMA	2.3	3.4	1.3	2.3	22	58
IQR	3.0	7.0	3.3	1.8	104	360

Table 7.3: Performance comparison of the RBF volume deformation using different solvers for the linear system of equation (6.2), averaged over five runs. The table reports deformation times in seconds. The chart depicts performance differences relative to the GPU-based MAGMA solver.

simple and smooth, so that a moderate number of RBF kernels is sufficient to represent the deformation, see also Botsch and Kobbelt (2005). We can therefore compute the deformation by using only a subset of the surface nodes $\boldsymbol{s}_j$ as centers $\boldsymbol{c}_j$. This turns the interpolation problem of equation (6.2) into a least squares approximation problem, where the required number of centers depends on the complexity of the deformation only—instead of on the complexity of the mesh.

As an implementation of this concept we use the incremental QR solver (IQR) initially presented in Botsch and Kobbelt (2005). This solver starts by fitting the polynomial term of equation (3.8) only and then incrementally adds more and more

RBF kernels φ_j until the least squares error falls below a user-prescribed threshold. Below we sketch the main ideas of incremental QR solver for completeness.

Using just k basis functions (polynomial and RBFs) instead of the full set of $(m+4)$ basis functions corresponds to replacing the quadratic $(m+4)\times(m+4)$ linear system $\boldsymbol{\Phi W} = \boldsymbol{S}$ of equation (6.2) by the reduced $(m + 4) \times k$ system $\boldsymbol{\Phi}_k \boldsymbol{W}_k = \boldsymbol{S}$, where $\boldsymbol{\Phi}_k$ is composed from the k columns of $\boldsymbol{\Phi}$ corresponding to the k selected basis functions, and $\boldsymbol{W}_k$ are their respective coefficients. This over-determined system can be solved robustly using the QR factorization:

$$\boldsymbol{\Phi}_k = \boldsymbol{Q}_k \boldsymbol{R}_k, \quad \boldsymbol{W}_k = \boldsymbol{R}_k^{-1} \boldsymbol{Q}_k^{\mathrm{T}} \boldsymbol{S}. \tag{7.1}$$

The main observation of Botsch and Kobbelt (2005) is that computing the QR factorization of the *full* matrix $\boldsymbol{\Phi}$ iteratively processes column by column for $k = 1, \ldots, m+4$, and that iteration k basically computes $\boldsymbol{Q}_k$ and $\boldsymbol{R}_k$. In addition, the least squares error $\|\boldsymbol{\Phi}_k \boldsymbol{W}_k - \boldsymbol{S}\|^2$ can be determined almost for free without actually having to compute $\boldsymbol{W}_k$. The IQR solver therefore works similar to a standard QR solver, but can stop as soon as at iteration k the first k columns of $\boldsymbol{\Phi}$ yield a sufficiently accurate least squares solution. Since this algorithm simply chooses the first k columns of $\boldsymbol{\Phi}$, a suitable re-ordering of the columns, i.e., of the corresponding basis functions, is performed in a pre-process. We use a farthest point center selection strategy, i.e., we select centers such that the minimum distance between centers is maximized. This type of selection can be computed at negligible cost and has the advantage that it guarantees a good matrix condition number (Botsch and Kobbelt 2005).

The complexity of solving the linear least squares system of equation (7.1) is $\mathcal{O}(mk^2)$. Hence, in the worst case that all $m + 4$ columns have to be used the complexity still is $\mathcal{O}(m^3)$ as for all other solvers. Since the computational overhead compared to a standard QR solver is negligible, the performance is on par with standard (CPU-based) solvers even if the deformation is complex and requires a large number of centers (see Courier example in table 7.3). However, for simple deformations, such as the Bore and Pipe examples in table 7.3, the incremental solver even outperforms the GPU-based MAGMA solver. We note that the user-prescribed error might negatively influence the resulting element quality if it is not small enough, thereby offering a trade-off between performance and quality.

Comparing the performance of our RBF-based technique with those investigated by Staten and colleagues shows that our method is computationally more expensive. However, in all but one case our method allows to perform an absolute deformation

to the full parameter change without resulting in inverted elements. Other methods might only reach this goal by falling back to several steps of relative deformation, thereby becoming computationally more expensive than our approach.

7.6 Summary and Conclusion

In this chapter, we presented a comprehensive evaluation of our simple yet versatile framework for high-quality mesh deformation of both surface and volume meshes. The smoothness of our triharmonic RBF deformations leads to similar or superior element quality compared to all techniques evaluated in (Staten et al. 2011). The implementation of our framework is straightforward and essentially requires solving linear systems using standard solvers—at least in its basic version. Therefore, it can be considered easier to implement than, e.g., the LBWARP method. While being similarly straightforward to implement as our method, the FEMWARP technique has to be derived explicitly for each element type. In contrast, our RBF deformations are highly flexible, since the same unified code can deform arbitrary geometries in arbitrary dimensions. Finally, we note that the full implementation of our framework including the CAD-based surface deformation relies on the availability of additional support structures and libraries, i.e., efficient spatial search structures and a modern CAD kernel.

We also investigated solutions to potential issues arising in mesh deformation for design optimization. We presented a simple and efficient technique for constructing inversion-free deformations. Furthermore, we illustrated how the performance of our approach can be drastically improved by employing efficient GPU-based linear solvers or incremental least squares solvers.

We conclude this chapter—and thereby the second part of this thesis—by revisiting the second research question set out for this thesis: How can we *apply* and *improve* existing techniques?

As for the question of *application*, we have shown that by consistently exploiting the flexibility of kernel-based RBF deformations we are able to develop a unified framework for shape deformation in design optimization. This framework allows for the implementation of more practical and powerful design optimization processes, namely the fully automatic and parallel optimization of CAD-based design prototypes without the need for costly and error-prone meshing steps during the optimization loop.

As for the question of *improvement*, we highlighted several ways to improve the basic RBF deformation approach with regards to essential criteria, such as quality, performance, and scalability. Our simple splitting technique allows for larger deformations while maintaining high element quality. The use of advanced linear solvers drastically boosts both performance and scalability to more complex geometries.

Now that we completed the *analysis* and *application* parts of this thesis, we continue the third part by investigating how to incorporate geometric constraints directly into the deformation, thereby fostering the creation of more usable and practical designs during the optimization process.

PART III

CONSTRAINED DEFORMATION TECHNIQUES

In this part, we investigate the third research question of this thesis: *How to incorporate additional constraints directly into the deformation*? To this end, we first investigate the use of explicit deformation energies for the flexible modeling of different material behavior. We then present a scalable deformation technique based on moving least squares approximation that incorporates geometric constraints such as planarity or circularity directly into the deformation, thereby fostering the creation of more feasible designs during optimization. Finally, we incorporate a technique for the automatic detection of geometric primitives, thereby reducing the effort to setup the deformation.

CHAPTER 8

CONSTRAINED DEFORMATION

In this final chapter, we present a novel shape deformation method for its use in design optimization tasks. Our space deformation technique based on moving least squares approximation improves upon existing approaches in crucial aspects: It offers the same level of modeling flexibility as surface-based deformations, but it is independent of the underlying geometry representation and therefore highly robust against defects in the input data. It overcomes the scalability limitations of existing space deformation techniques based on globally supported radial basis functions while providing the same high level of deformation quality. Finally, unlike existing space deformation approaches, our technique directly incorporates geometric constraints—such as preservation of critical feature lines, circular couplings, planar or cylindrical construction parts—into the deformation, thereby fostering the exploration of more favorable shape variations during the design optimization process.

8.1 INTRODUCTION

Throughout the preceding chapters, we illustrated how deformation techniques can be effectively used in order to create design variations during a simulation-based design optimization process. However, even though these techniques drastically simplify the creation of design variations, their successful application within practical design optimization tasks comes with a number of challenges:

1. In terms of performance the method might not scale to complex optimization scenarios. The RBFs proposed in previous chapters offer high deformation quality due to their built-in minimization of fairness energies, but the involved dense linear system restrict the method to moderately sized problems.

2. The method might not offer a sufficient level of modeling flexibility, e.g., to simulate inhomogeneous material behavior during deformation. Global

triharmonic RBFs, which implicitly minimize bending-type energies, fail to simulate stretching-dominant materials.

3. Critical features required for functionality and realization of design prototypes might not be properly preserved during deformation.

In this chapter, we present a shape deformation technique based on moving least squares (MLS) discretization (Fries and Matthies 2004) that improves upon previous approaches in all of the above aspects: Since we follow a space deformation approach our method is independent of the underlying geometry representation and highly robust towards defects in the input data. In terms of deformation quality, our method is competitive to global triharmonic RBFs. We drastically improve on the latter in terms of scalability, having to solve sparse linear systems only. By incorporating explicit stretching and bending energies, we offer the same level of modeling flexibility as surface-based methods. The key advantage of our novel technique, however, is that it directly incorporates geometric constraints into the deformation, thereby fostering the exploration of more meaningful and producible shape variations during the design optimization process. Finally, in order to make the setup procedure of our deformation method easier for the designer or engineer, we incorporate a technique for the automatic detection of geometric primitives into our deformation framework.

In the following sections, we describe our deformation technique in detail, going from the fundamentals to the specifics. We begin with a description of a general deformation model suitable for design optimization (section 8.2). We describe our approach to space deformation based on subspace techniques in section 8.3, where we also analyze and compare different choices of subspaces. In order to make our technique fully independent from the underlying geometry representation, we describe a spatial discretization of deformation energies in section 8.4. Finally, we describe how to integrate constraints into the deformation in section 8.5.

8.2 Mesh-based surface deformation

In this section, we describe a *mesh-based* deformation model that is suitable for a design optimization framework. Since the most common targets for design optimization are sheet metal surfaces, such as car bodies, aircraft wings, or ship hulls, we concentrate on *surface deformation* models first. We then extend the resulting

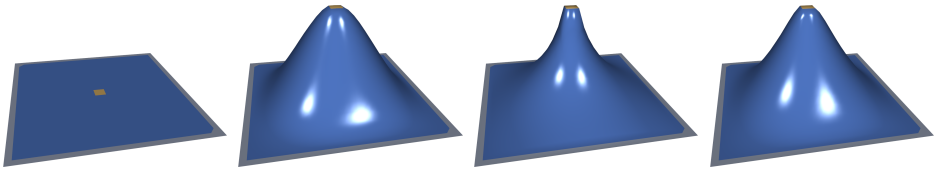


Figure 8.1: Handle-based surface deformation of a plane (1100 vertices). From left to right: Undeformed model, minimization of pure bending, pure stretching, and a mixture thereof with parameters $\gamma_b = 0.6$ and $\gamma_s = 1.0$. We choose $\gamma_f = 100$ in order to ensure that prescribed handle and fixed constraints are satisfied.

model to *subspace* surface deformations and true *volumetric space deformations* in the following sections. Note that the deformation model we present in this section is almost identical to the thin shell deformation approach described in section 3.1. However, for the sake of extensibility to volumetric space deformations we employ a slightly simplified formulation here.

Similar to the shape deformation methods presented in the preceding chapters, we use a handle-based direct manipulation interface to control the deformation: We distinguish three types of surface regions on the mesh: The handle region $\mathcal{H}$ is directly displaced by the user. The fixed region $\mathcal{F}$ stays in place. The deformable region $\mathcal{D}$ is updated according to the physical deformation method while satisfying the modeling constraints given by $\mathcal{H}$ and $\mathcal{F}$. We illustrate an example of this modeling metaphor in figure 8.1, with the handle region in gold, the fixed region in gray, and the deformable region in blue.

The deformable region $\mathcal{D}$ should behave in a physically plausible manner, i.e., it should deform like a thin shell based on stretching and bending energies. The deformations occurring in design optimization tasks typically are rather small. Therefore, a linear deformation model is sufficient, where stretching and bending are measured by first and second order partial derivatives of the displacement function $\boldsymbol{d}$, respectively.

In the continuous setting, the deformation $\boldsymbol{d} \colon \mathcal{S} \to \mathbb{R}^3$ of a surface $\mathcal{S}$ can be computed by minimizing the energy functional

$$E_{\text{shell}}[\boldsymbol{d}] = \gamma_s E_{\text{stretch}}[\boldsymbol{d}] + \gamma_b E_{\text{bend}}[\boldsymbol{d}] + \gamma_f E_{\text{fix}}[\boldsymbol{d}], \tag{8.1}$$

consisting of weighted energy contributions for bending, stretching, and constraint deviation (Botsch and Sorkine 2008):

$$E_{\text{stretch}}[\boldsymbol{d}] = \int_{\mathcal{D}} \|\nabla \boldsymbol{d}(\boldsymbol{x})\|^2 \, \mathrm{d}\boldsymbol{x}, \tag{8.2}$$

$$E_{\text{bend}}[\boldsymbol{d}] = \int_{\mathcal{D}} \|\Delta \boldsymbol{d}(\boldsymbol{x})\|^2 \, \mathrm{d}\boldsymbol{x}, \tag{8.3}$$

$$E_{\text{fix}}[\boldsymbol{d}] = \int_{\mathcal{H}\cup\mathcal{F}} \left\|\boldsymbol{d}(\boldsymbol{x}) - \bar{\boldsymbol{d}}(\boldsymbol{x})\right\|^2 \, \mathrm{d}\boldsymbol{x}, \tag{8.4}$$

where $\nabla \boldsymbol{d}$ denotes the Jacobian of $\boldsymbol{d}$, $\Delta \boldsymbol{d} = \nabla \cdot \nabla \boldsymbol{d}$ its Laplacian, $\|\cdot\|$ the Frobenius matrix norm or the Euclidean vector norm, and $\bar{\boldsymbol{d}}$ the prescribed Dirichlet constraints for the fixed and handle regions.

If we assume that the surface $\mathcal{S}$ is discretized by a proper two-manifold triangle mesh $\mathcal{T}$ with only non-degenerate triangles, then the most flexible discretization of the above thin shell deformation energies is one whose degrees of freedom are the individual vertex positions $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, or the vertex displacements $\boldsymbol{\delta}_1, \ldots, \boldsymbol{\delta}_n$:

$$\boldsymbol{d}_h(\boldsymbol{x}) = \sum_{i=1}^{n} \boldsymbol{\delta}_i \psi_i(\boldsymbol{x}), \tag{8.5}$$

where ψ_i are the piecewise linear shape functions on the triangulation $\mathcal{T}$. Based on this discretization we can approximate the above energies (Botsch and Sorkine 2008; Botsch et al. 2010) as

$$E_{\text{stretch}}[\boldsymbol{d}_h] = \sum_{t\in\mathcal{D}} A_t \, \|\nabla \boldsymbol{\delta}_t\|^2, \tag{8.6}$$

$$E_{\text{bend}}[\boldsymbol{d}_h] = \sum_{\boldsymbol{x}_i\in\mathcal{D}} A_i \, \|\Delta \boldsymbol{\delta}_i\|^2, \tag{8.7}$$

$$E_{\text{fix}}[\boldsymbol{d}_h] = \sum_{\boldsymbol{x}_i\in\mathcal{H}\cup\mathcal{F}} A_i \, \left\|\boldsymbol{\delta}_i - \bar{\boldsymbol{\delta}}_i\right\|^2, \tag{8.8}$$

where A_i denotes the Voronoi area of vertex i, and A_t is the area of triangle t. We use the well-established discrete differential operators proposed by Meyer et al. (2003), which allows us to write the discrete gradient $\nabla \boldsymbol{\delta}_t$ and discrete Laplacian $\Delta \boldsymbol{\delta}_i$ as a linear combination of neighboring vertices.

For implementation convenience and easier extensibility in the following sections, we write the discrete shell energy of equations (8.6)–(8.8) as

$$E_{\text{shell}}[\boldsymbol{d}_h] = \gamma_s \, \|\boldsymbol{G}\boldsymbol{D}\|^2 + \gamma_b \, \|\boldsymbol{L}\boldsymbol{D}\|^2 + \gamma_f \, \left\|\boldsymbol{F}(\boldsymbol{D} - \bar{\boldsymbol{D}})\right\|^2, \tag{8.9}$$

where $\boldsymbol{D} = [\boldsymbol{\delta}_1^{\mathrm{T}}, \ldots, \boldsymbol{\delta}_n^{\mathrm{T}}]^{\mathrm{T}}$ is the $(n \times 3)$ matrix of per-vertex displacements, and $\boldsymbol{G}$ and $\boldsymbol{L}$ are gradient and Laplacian matrices containing the required cotangent weights in each row and having their rows weighted by $\sqrt{A_i}$, respectively (see Botsch and Sorkine (2008) and Botsch et al. (2010) for details). $\boldsymbol{F}$ is a diagonal matrix with $\boldsymbol{F}_{i,i} = \sqrt{A_i}$ if $\boldsymbol{x}_i \in \mathcal{F} \cup \mathcal{H}$ and $\boldsymbol{F}_{i,i} = 0$ otherwise. The minimization of the shell energy of equation (8.9) then requires us to solve the normal equations of the linear least squares system

$$\left[\gamma_s \boldsymbol{G}^{\mathrm{T}}\boldsymbol{G} + \gamma_b \boldsymbol{L}^{\mathrm{T}}\boldsymbol{L} + \gamma_f \boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}\right]\boldsymbol{D} \;=\; \gamma_f \boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}\bar{\boldsymbol{D}}, \tag{8.10}$$

which we solve efficiently using sparse Cholesky factorization (Chen et al. 2008). Note that in case the conditioning of the normal equations becomes a problem, we could also solve the system directly using a sparse QR factorization method (Davis 2011). In order to ensure proper satisfaction of the Dirichlet boundary constraints, we typically choose γ_f to be one or two orders of magnitude larger than the smoothness weights γ_s and γ_b. This mesh-based surface deformation approach, depicted in figure 8.1, is our ground truth technique, which we try to reproduce using more robust and general space deformation methods.

8.3 Subspace Surface Deformation

The deformation model described in the previous section offers high flexibility, since it uses the degrees of freedom of the mesh as degrees of freedom for the surface deformation. As motivated above, we are aiming at a *space deformation* approach, which deforms not only the given surface $\mathcal{T}$, but the whole space Ω embedding the object.

In contrast to the surface deformation of the previous section, we are looking for a deformation function $\boldsymbol{d}\colon \Omega \subset \mathbb{R}^3 \to \mathbb{R}^3$ that deforms the embedding space Ω around the model, while at the same time offering a comparable flexibility and deformation quality:

$$\boldsymbol{d}_h(\boldsymbol{x}) \;=\; \sum_{j=1}^{k} \boldsymbol{w}_j \varphi_j(\boldsymbol{x})\,,$$

where $\varphi_1, \ldots, \varphi_k$ are coarser basis functions ($k \ll n$) and $\boldsymbol{w}_j \in \mathbb{R}^3$ their coefficients.

In the following, we analyze the modeling flexibility of different subspaces corresponding to different basis functions φ_j. In order to make the experiments more comparable to the mesh-based deformation, and to avoid any dependence on

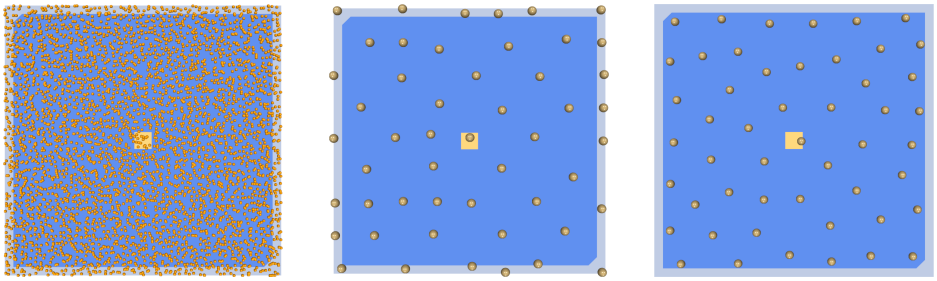


Figure 8.2: Surface sampling: Left: Dense random sampling of the mesh. From the dense samples we select a farthest point subset (center), perform iterative Lloyd relaxation on this subset (right), and use these points as RBF centers/MLS samples.

potentially insufficient numerical quadrature, we minimize the same vertex-based discrete shell energy of equation (8.9), but replace the per-vertex displacements $\boldsymbol{\delta}_i$ by $\boldsymbol{d}_h(\boldsymbol{x}_i)$. We can then express the $n \times 3$ matrix $\boldsymbol{D}$ of vertex displacements in terms of the coefficients $\boldsymbol{W} = [\boldsymbol{w}_1^\mathrm{T}, \dots, \boldsymbol{w}_k^\mathrm{T}]^\mathrm{T} \in \mathbb{R}^{k\times 3}$ using a $n \times k$ subspace matrix $\boldsymbol{\Phi}$:

$$\boldsymbol{D} = \boldsymbol{\Phi}\boldsymbol{W} \quad \text{with} \quad \boldsymbol{\Phi}_{i,j} = \varphi_j(\boldsymbol{x}_i)\,.$$

Inserting this into the discrete shell energy of equation (8.9) leads to the $k \times k$ least squares system

$$\boldsymbol{\Phi}^\mathrm{T}\left[\gamma_s \boldsymbol{G}^\mathrm{T}\boldsymbol{G} + \gamma_b \boldsymbol{L}^\mathrm{T}\boldsymbol{L} + \gamma_f \boldsymbol{F}^\mathrm{T}\boldsymbol{F}\right]\boldsymbol{\Phi}\,\boldsymbol{W} \;=\; \boldsymbol{\Phi}^\mathrm{T}\left[\gamma_f \boldsymbol{F}^\mathrm{T}\boldsymbol{F}\bar{\boldsymbol{D}}\right], \tag{8.11}$$

which has a drastically reduced complexity compared to the previous least squares system for surface deformation in equation (8.10).

In the following, we compare different choices for the basis functions φ_j. Motivated by the results obtained in the first two parts of this thesis, we focus on meshless, kernel-based discretizations, and start with globally supported triharmonic and biharmonic RBFs, which however have the drawback of high computational cost and limited scalability. We then analyze compactly supported Gaussians and Wendland RBFs (Wendland 2010) as well as moving least squares discretization (Fries and Matthies 2004).

For these kernel-based discretizations, we first need an efficient method to place the basis functions φ_j on the surface $\mathcal{T}$. To this end we employ a sampling strategy based on iterative Lloyd-relaxation (Lloyd 1982), which we illustrate in figure 8.2. Starting from the initial mesh, we create a dense sampling of the surface

by computing random points within each triangle of the mesh. We then select a subset of k samples from the dense sampling by means of farthest point selection: We start with a random sample and iteratively add new samples based on maximizing the minimum distance between the new and the previously chosen samples. Finally, in order to maximize uniformity of the sampling we perform Lloyd-relaxation, i.e., we iteratively move each sample to the barycenter of the dense sample points being closest to the sample (Lloyd 1982). We perform the search for closest points efficiently through parallelization (Dagum and Menon 1998) and space partitioning (Samet 1990).

8.3.1 *Global RBFs*

In the preceding chapters, we successfully employed global triharmonic RBFs for high quality mesh deformation. Following this approach, we can construct a subspace by using basis functions

$$\varphi_j(\boldsymbol{x}) \;=\; \left\|\boldsymbol{x} - \boldsymbol{c}_j\right\|^3$$

located at centers $\boldsymbol{c}_j$. In figure 8.3 we provide a comparison between the purely surface-based deformation and a subspace deformation using global triharmonic RBFs. While triharmonic RBFs work well for minimizing bending (which they do by construction), they fail to model stretching-dominant materials. Furthermore, due to their global support the matrix $\boldsymbol{\Phi}$ is dense, posing a serious limitation in terms of scalability. Even though biharmonic basis functions $\varphi_j(\boldsymbol{x}) = \|\boldsymbol{x} - \boldsymbol{c}_j\|$ (see also section 3.5) yield improved results for stretching minimization, they still suffer from the same scalability limitations as triharmonic basis functions.

8.3.2 *Compact RBFs*

An alternative to globally supported RBFs are compactly supported RBFs, such as the C^2-continuous Wendland functions

$$\begin{aligned}\varphi_j(\boldsymbol{x}) \;&=\; \varphi\left(\left\|\boldsymbol{x} - \boldsymbol{c}_j\right\|\right)\\ &=\; \varphi(r) \;=\; \begin{cases}(1-(r/\rho))^4((4r/\rho)+1)\,, & r<\rho\,,\\ 0\,, & \text{otherwise}\,.\end{cases}\end{aligned}$$

The choice of the support radius ρ is critical for the quality of the resulting subspace. In our implementation, we set support radii so that at least s basis functions φ_j

cover each point $\boldsymbol{x}_i$ of the geometry. As illustrated in figure 8.3, the results with compact RBFs heavily depend on the chosen support radius. A small radius of $s = 5$ leads to artifacts in the deformation. Only with a large radius of $s = 50$ the subspace produces results comparable to the surface deformation. In this case, however, the resulting linear system is not sufficiently sparse anymore, so that the compact RBFs are not an alternative in terms of scalability. Similar limitations apply to Gaussian RBFs

$$\varphi_j(\boldsymbol{x}) = \mathrm{e}^{-\frac{\|\boldsymbol{x}-\boldsymbol{c}_j\|^2}{\rho^2}},$$

which also lead to a certain amount of smoothness artifacts in the deformed meshes as we show in figure 8.3.

8.3.3 *Moving Least Squares*

An alternative to RBFs is the meshless moving least squares (MLS) approximation method, which allows for the construction of high quality and scalable subspaces, as we illustrate in figure 8.3. In contrast to compact RBFs, MLS yields high quality results already with a cover of $s = 5$. Since a reasonably comprehensive introduction to MLS is beyond the scope of this thesis we refer the reader to the detailed introduction of Fries and Matthies (2004) and only provide the required basic facts. The MLS basis functions $\varphi_j(\boldsymbol{x})$ are defined as

$$\varphi_j(\boldsymbol{x}) = \boldsymbol{p}(\boldsymbol{x})^{\mathrm{T}} \boldsymbol{M}^{-1}(\boldsymbol{x}) \boldsymbol{p}(\boldsymbol{c}_j) w(\boldsymbol{x} - \boldsymbol{c}_j),$$

where $\boldsymbol{p}(\boldsymbol{x})$ is the vector of monomials $\boldsymbol{p}(x, y, z) = [1, x, y, z]^{\mathrm{T}}$ and the spatially varying matrix $\boldsymbol{M}(\boldsymbol{x}) \in \mathbb{R}^{4\times4}$ is the so-called *moment matrix*

$$\boldsymbol{M}(\boldsymbol{x}) = \sum_{j=1}^{k} w(\boldsymbol{x} - \boldsymbol{c}_j) \boldsymbol{p}(\boldsymbol{c}_j) \boldsymbol{p}(\boldsymbol{c}_j)^{\mathrm{T}}.$$

The weighting function $w(\cdot)$ is *compactly supported* and of sufficient smoothness. In our implementation, we use $w(r) = \frac{1}{2}\cos(r/\rho \cdot \pi) + \frac{1}{2}$, with $w(r) = 0$ for $r > \rho$. Other choices of smooth weight functions work equally well, see Fries and Matthies (2004) for details. Unlike RBFs, the MLS basis functions do not have a simple analytic form, but require the inversion of the moment matrix for function evaluation. Note that the moment matrix becomes singular if the MLS samples $\boldsymbol{c}_j$ lie in the kernel of a linear polynomial (coplanar samples). In contrast to Martin et

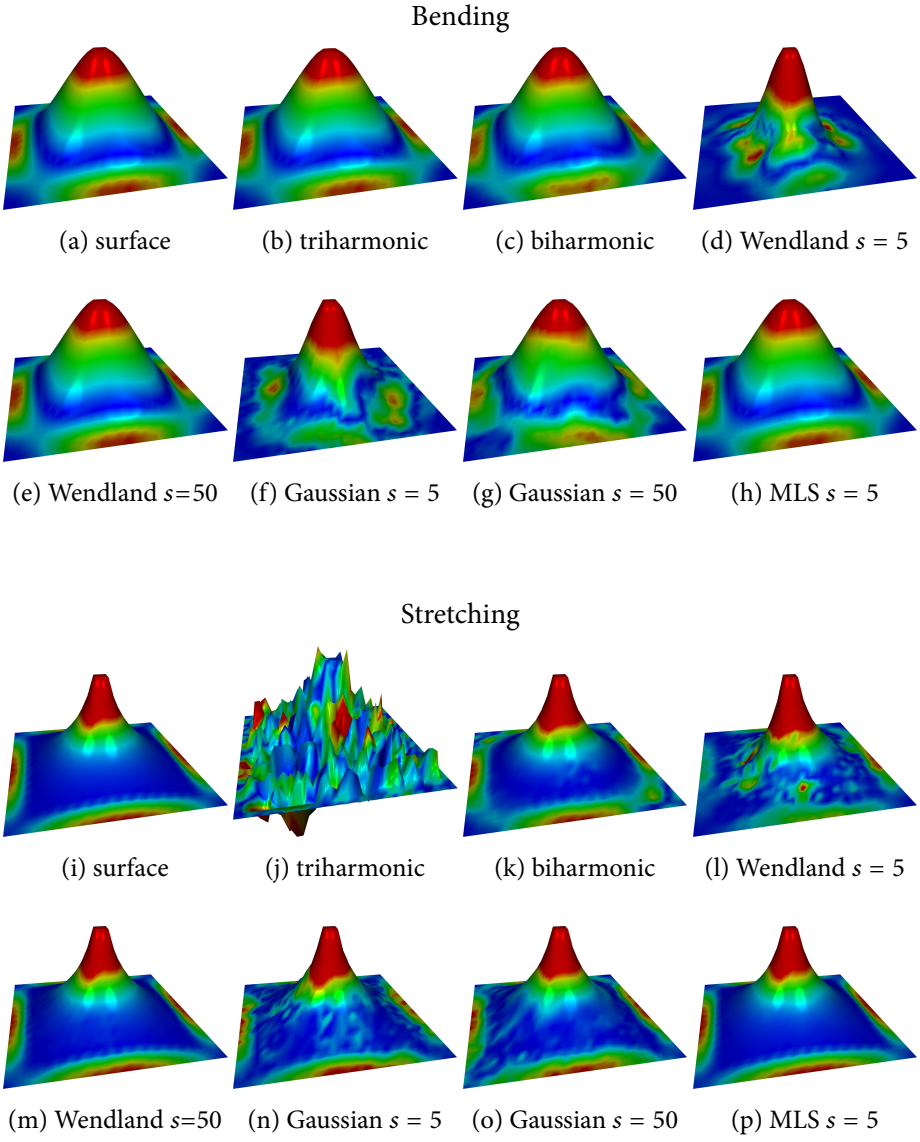


Figure 8.3: Subspace deformation of a plane (1100 vertices) minimizing bending (top rows) and stretching (bottom rows) energies. For each energy type we compare the mean curvature plot of the ground truth surface deformation, global triharmonic and biharmonic RBFs, compact Wendland and Gaussian RBFs with small ($s = 5$) and large ($s = 50$) support, MLS with small ($s = 5$) support. RBFs and MLS use 1000 basis functions.

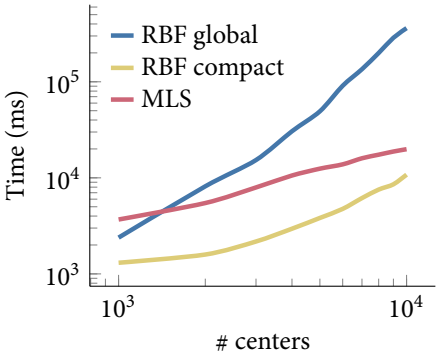


Figure 8.4: Performance comparison between global biharmonic RBFs, compact Wendland RBFs ($s = 50$), and MLS basis functions ($s = 5$). Computation time in milliseconds versus the number of basis function centers.

al. (2010), who switch to more complex generalized MLS basis functions in order to handle degenerate sampling, we robustly handle this case by replacing the inverse $\boldsymbol{M}^{-1}$ by the pseudo-inverse $\boldsymbol{M}^{+}$ (Golub and Van Loan 2013). We compute $\boldsymbol{M}^{+}$ as $\boldsymbol{V}\boldsymbol{\Sigma}^{+}\boldsymbol{U}^{\mathrm{T}}$ based on the singular value decomposition (SVD) $\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{T}}$, since this is the numerically most stable method (Golub and Van Loan 2013; Trefethen and Bau 1997). Using the SVD is also the computationally most expensive technique for computing the pseudo-inverse, but for our 4×4 matrices this turned out to not be crucial.

Even though MLS basis functions are more expensive to evaluate than RBFs, this is not a problem for their application in design optimization, since the MLS matrix $\boldsymbol{\Phi}$, and hence all pseudo-inverse computations, can be pre-computed and re-used throughout the design optimization loop. More importantly, the MLS discretization scales well to complex models due to the sparsity of $\boldsymbol{\Phi}$, and the evaluation of φ_j is trivial to parallelize.

We provide a performance comparison between global and compact RBFs as well as MLS basis functions in figure 8.4. In this test, we perform deformations with an increasing number of basis function centers ranging from 1k to 10k degrees of freedom. We can observe that with an increasing number of basis functions both compact RBFs and MLS outperform the globally supported RBFs. In terms of memory usage the globally supported RBFs result in dense system matrices and therefore have quadratically growing storage requirements. In contrast, both

compact methods have storage requirements growing only linearly with a constant depending on the support radius.

In summary, the MLS subspace yields a deformation that combines the strengths of the three approaches: the flexible energy minimization of mesh-based surface deformations, the high quality of global RBFs, and the scalability of compactly supported basis functions. The flexibility of MLS deformations was for instance demonstrated in (Martin et al. 2010), where MLS basis functions were used to deform solids, shells, and rods based on physical laws.

8.4 VOLUMETRIC SPACE DEFORMATION

The previous section motivated the use of MLS basis functions as a flexible subspace for high quality deformation. However, the above comparisons—while using a space deformation function—still employed the stretching and bending energies of equations (8.6)–(8.8) based on a surface mesh. In this section, we generalize the MLS deformation to true volumetric space deformations, which can then robustly process defect-laden, highly complex, and multi-component input meshes. To this end, we have to (i) place MLS kernels not only on the surface, but also in the embedding space Ω, and (ii) replace the vertex-based quadrature for integrating gradients and Laplacians over the surface $\mathcal{T}$ by a numerical cubature for integration over the embedding space Ω.

The volumetric sampling is a simple extension of the surface sampling shown in figure 8.2. We first perform a dense sampling of the volume elements and then choose a subset by means for farthest point sampling. We add this subset to the initial farthest point sampling of the surface $\mathcal{T}$ and then perform a combined Lloyd clustering of both the surface and volume samples, where we give a higher weight or density to the surface, leading to a slightly higher sampling density of the surface compared to the volume. As before, we denote the resulting MLS samples by $\boldsymbol{c}_j$, $j = 1, \dots, k$.

In order to determine the integration points $\boldsymbol{t}_i$, $i = 1, \dots, N$, we perform exactly the same sampling strategy, but we make sure that the sampling density of the integration points $\boldsymbol{t}_i$ is sufficiently larger than the density of the MLS samples $\boldsymbol{c}_j$ (we use $N \approx 4k$).

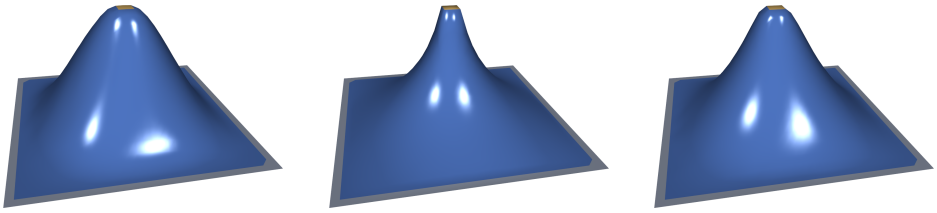


Figure 8.5: Handle-based deformation of a plane minimizing deformation energies using a spatial discretization based on MLS (1000 kernels). Pure bending (left), pure stretching (center), and a mixture thereof (right).

Discretizing the stretching energy of equation (8.2) in space amounts to evaluating the basis function derivatives at integration points:

$$E_{\text{stretch}}[\boldsymbol{d}_h] = \sum_{i=1}^{N} V(\boldsymbol{g}) \, \|\nabla \boldsymbol{d}(\boldsymbol{t}_i)\|^2 = \sum_{i=1}^{N} V(\boldsymbol{g}) \left\| \sum_{j=1}^{k} \boldsymbol{w}_j \nabla \varphi_j(\boldsymbol{t}_i) \right\|^2 = \|\boldsymbol{G}\boldsymbol{W}\|^2 , \quad (8.12)$$

where $V(\boldsymbol{g})$ is the (approximate) Voronoi volume of integration point $\boldsymbol{t}_i$, and $\boldsymbol{G}$ is a $3N \times k$ gradient matrix with

$$\begin{aligned} \boldsymbol{G}_{3i,j} &= \sqrt{V_i} \cdot \frac{\partial \varphi_j(\boldsymbol{t}_i)}{\partial x}, \\ \boldsymbol{G}_{3i+1,j} &= \sqrt{V_i} \cdot \frac{\partial \varphi_j(\boldsymbol{t}_i)}{\partial y}, \\ \boldsymbol{G}_{3i+2,j} &= \sqrt{V_i} \cdot \frac{\partial \varphi_j(\boldsymbol{t}_i)}{\partial z}. \end{aligned}$$

Similarly, discretizing the bending energy of equation (8.3) in space leads to

$$E_{\text{bend}}[\boldsymbol{d}_h] = \sum_{i=1}^{N} V(\boldsymbol{g}) \, \|\Delta \boldsymbol{d}(\boldsymbol{t}_i)\|^2 = \sum_{i=1}^{N} V(\boldsymbol{g}) \left\| \sum_{j=1}^{k} \boldsymbol{w}_j \Delta \varphi_j(\boldsymbol{t}_i) \right\|^2 = \|\boldsymbol{L}\boldsymbol{W}\|^2 , \quad (8.13)$$

with a $N \times k$ Laplacian matrix $\boldsymbol{L}_{i,j} = \sqrt{V(\boldsymbol{g})} \Delta \varphi_j(\boldsymbol{t}_i)$. For the computation of analytical basis function derivatives we refer to Fries and Matthies (2004).

For the prescribed modeling constraints we keep the subspace formulation $\|\boldsymbol{F}\boldsymbol{\Phi}\boldsymbol{W} - \boldsymbol{F}\bar{\boldsymbol{D}}\|^2$ of equation (8.11). Combining this with the above spatial energies, i.e., with the MLS version of the gradient matrix $\boldsymbol{G}$ and the Laplace matrix $\boldsymbol{L}$, leads to the final $k \times k$ linear least squares system

$$\left[\gamma_s \boldsymbol{G}^{\mathrm{T}}\boldsymbol{G} + \gamma_b \boldsymbol{L}^{\mathrm{T}}\boldsymbol{L} + \gamma_f \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}\boldsymbol{\Phi} \right] \boldsymbol{W} = \gamma_f \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}\bar{\boldsymbol{D}}. \quad (8.14)$$

Solving this system yields the desired MLS *space deformation*, which no longer depends on the complexity and quality of the input meshes. As demonstrated in figure 8.5, the MLS deformation based on spatial energies provides the same deformation quality and flexibility as the surface-based discretization of figure 8.3.

8.5 Constrained Space Deformation

A design prototype typically contains regions with important geometric properties such as planar components, characteristic feature lines, or circular and cylindrical couplings. Such geometric features are often essential for the design in order to fulfill its function or to meet production limitations. The classical approach to maintain such constraints during an optimization process is to penalize constraint violation by integrating additional penalty terms into the fitness or cost function. However, this approach has the severe drawback that infeasible designs are still created and *evaluated*, which is particularly unfavorable when the performance evaluation involves computationally expensive and time-consuming CFD or FEM simulations.

In contrast, we propose to maintain constraints right from the start by incorporating them directly into the deformation method, thereby preventing the evaluation of infeasible designs. Within our method the user marks a particular region—probably guided by some mechanism for automatic detection of geometric primitives—as being of a particular constraint type such as, e.g., planarity. Then, when deforming the shape by manipulating the handle region, our method automatically makes sure that the corresponding constraint is satisfied *while still minimizing the deformation energy* of equation (8.1).

As already noted in chapter 2, several constrained deformation approaches have been proposed during recent years (Mitra et al. 2013). Most of them, however, are purely surface-based in nature and therefore too limited for general design optimization tasks. In contrast, the Shape-Up technique of Bouaziz et al. (2012) allows to maintain constraints on arbitrary geometric data sets, making it the method of choice for our application area. In the following, we briefly describe the technique and show how we adopt it within our system. For a full treatment of the method, however, we refer the reader to the original paper (Bouaziz et al. 2012).

The key ingredients of Shape-Up are projection operators for different types of constraints. Let $\boldsymbol{X}$ be the vector of stacked point positions $\boldsymbol{x}$ of an arbitrary discrete geometric model $\mathcal{M}$. Modeling a constraint (e.g., planarity) for this model requires

the projection $P(\boldsymbol{X})$ of $\boldsymbol{X}$ onto the constraint set $\mathcal{C}$, i.e., the smallest change of $\boldsymbol{X}$ such that it satisfies the constraint. For a planarity constraint, for instance, $P(\boldsymbol{X})$ computes the projection onto a least squares fitting plane. For the most common constraints this projection can be computed rather straightforward, see Bouaziz et al. (2012) for a description of several projection operators.

Let m be the number of different constraint sets $\mathcal{C}_t, t = 1, \dots, m$. We can then measure deviation from the constraints as squared distance from constraint projections $P_t(\boldsymbol{X})$:

$$E_{\text{constr}}[\boldsymbol{X}] \;=\; \sum_{t=1}^{m} \|\boldsymbol{X} - P_t(\boldsymbol{X})\|^2 \,. \tag{8.15}$$

Since the projections $P_t(\boldsymbol{X})$ are typically nonlinear functions of $\boldsymbol{X}$, E_{constr} is minimized by *alternating optimization* (also called block coordinate descent): First $\boldsymbol{X}$ is kept fixed and all projections $\boldsymbol{X}_t = P_t(\boldsymbol{X})$ are computed (local step). Then the projected target positions $\boldsymbol{X}_t$ are held fixed and $\boldsymbol{X}$ is updated by a simultaneous least squares fit to the target positions $\boldsymbol{X}_t$ (global step). This process is iterated until convergence. Note that while Bouaziz et al. (2012) have shown that the energy is not increasing in each step, there is no formal proof that this process converges in a finite number of steps. We choose this type of minimization strategy for the sake of simplicity and robustness. However, other variants such as a non-linear conjugate gradient method or a trust-region variant of Newton's method (Nocedal and Wright 2006) are applicable as well.

In each iteration, the global step requires the solution of a linear least squares system of the form

$$\boldsymbol{C}^{\mathrm{T}}\boldsymbol{C}\boldsymbol{X} \;=\; \boldsymbol{C}^{\mathrm{T}}\bar{\boldsymbol{X}}, \tag{8.16}$$

where $\bar{\boldsymbol{X}}$ is the vector of the stacked projections $\boldsymbol{X}_t$. The matrix $\boldsymbol{C}$ contains the stacked constraint matrices $\boldsymbol{C}_t$, which combine the mean-centered positions of constrained points, i.e., for each constraint set $\mathcal{C}_t$ involving n_t points $\boldsymbol{C}_t$ is a $n_t \times n_t$ matrix with entries

$$\boldsymbol{C}_t(i,j) \;=\; \begin{cases} 1 - \frac{1}{n_t}\,, & i = j\,, \\ -\frac{1}{n_t}\,, & \text{otherwise}\,. \end{cases}$$

When combining the individual constraint matrices $\boldsymbol{C}_t$ into the global matrix $\boldsymbol{C}$, we adjust the columns of $\mathcal{C}_t$ such that they match the corresponding indices in the global point set $\boldsymbol{X}$. The mean-centering of positions allows for translation of

constraint sets during optimization, thereby improving the overall convergence rate of the iterative alternating optimization, see also section 2.2 and figure 10 in Bouaziz et al. (2012).

In order to integrate this approach into our framework, we add a constraint energy similar to equation (8.15) to our discrete shell energy equation (8.9) (weighted by γ_c) and also perform the above alternating optimization procedure. In each iteration, we first find the constraint projections $P(\boldsymbol{X})$ (local step) and combine them into the target vector $\bar{\boldsymbol{X}}$, which we rewrite in terms of displacements $\boldsymbol{\delta}$ instead of positions $\boldsymbol{x}$. The global minimization of constraint deviation is then integrated into the previous least squares system

$$\left[\gamma_s \boldsymbol{G}^{\mathrm{T}}\boldsymbol{G} + \gamma_b \boldsymbol{L}^{\mathrm{T}}\boldsymbol{L} + \gamma_f \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}\boldsymbol{\Phi} + \gamma_c \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{C}^{\mathrm{T}}\boldsymbol{C}\boldsymbol{\Phi}\right]\boldsymbol{W} = \boldsymbol{\Phi}^{\mathrm{T}}\left[\gamma_f \boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}\bar{\boldsymbol{D}} + \gamma_c \boldsymbol{C}^{\mathrm{T}}\bar{\boldsymbol{X}}\right], \quad (8.17)$$

which we again solve efficiently using sparse Cholesky factorization (Chen et al. 2008). We iterate this alternating optimization procedure until convergence, which typically happens after 1k–10k iterations in our examples—depending on the complexity of the constraints involved. Note that the convergence rate is independent from the chosen sampling density which only affects approximation accuracy.

8.5.1 *Alternative constraint formulation*

Applying the above constraint formulation in modeling setups with large constraint regions such as those obtained from automatic primitive detection (see section 8.5.3) reveals a severe limitation of the original Shape-Up formulation: Due to the mean-centering of constraints, the constraint matrix $\boldsymbol{C}$ can become dense, such that the resulting linear system can no longer be solved efficiently. The reason for this is the following: Let n_t be the number of points involved in a particular constraint set $\mathcal{C}_t$. Then in the original formulation this leads to $n_t \times n_t$ non-zero entries in the constraint matrix $\boldsymbol{C}_t$, leading to a fully dense matrix in the worst case scenario of a constraint covering the whole mesh, e.g., a planarity constraint on a planar mesh.

However, the primary reason for mean-centering is the improved convergence rate due to translation invariance of each constraint set. We propose an alternative constraint formulation sharing the same improved convergence behavior while

ensuring sparsity of the constraint matrix. Instead of mean-centering the points involved in a constraint set $\mathcal{C}_t$, we may also subtract an arbitrary point $\boldsymbol{x}_p \in \mathcal{C}_t$, e.g., the point closest to the mean, thereby leading to n_t rows with only two entries

$$\boldsymbol{C}_t(i,j) \;=\; \begin{cases} -1\,, & j = p\,, \\ 1\,, & j = i\,. \end{cases}$$

Using the above formulation effectively ensures that the constraint matrix $\boldsymbol{C}$ is sparse even in case of large constraint regions $\mathcal{C}_t$.

8.5.2 *Constraint Types*

In our current system, we implement five basic geometric constraint types of fundamental nature and general use: Planarity, circularity, cylinder, feature lines, and rigid shape constraints. For planarity and circularity constraints we employ projection operators described by Bouaziz et al. (2012). For the cylinder constraint, we use the cylinder fitting method described by Schneider and Eberly (2002) and project the points back onto the fitted cylinder. Our feature line constraint is modeled as a conformal matching of the points on the initial feature line, which therefore might translate, rotate, and uniformly scale. Similar to the feature line constraint, we also support shape constraints based on rigid matching, allowing for translation and rotation only. This constraint can be useful in a number of settings, e.g., in order to keep complete components of a design rigid, or to rigidly maintain the shape of selected mesh elements—such as boundary layer cells in a CFD mesh—which are of particular importance for accurate physical performance calculation. In figure 8.6, we show synthetic examples demonstrating the effect of each constraint type. Note that due to the iterative nature of the alternating optimization, the resulting meshes do indeed minimize the deformation energy while satisfying the geometric constraints.

8.5.3 *Automatic Constraint Detection*

The manual selection of geometric primitives in a mesh can be a tedious and time-consuming task. In order to speed up the setup process and guide the designer towards meaningful constraints, we employ a procedure for the automatic detection of primitives in the surface mesh which then can be used as constraints during deformation. Note that even though in some design optimization scenarios it is

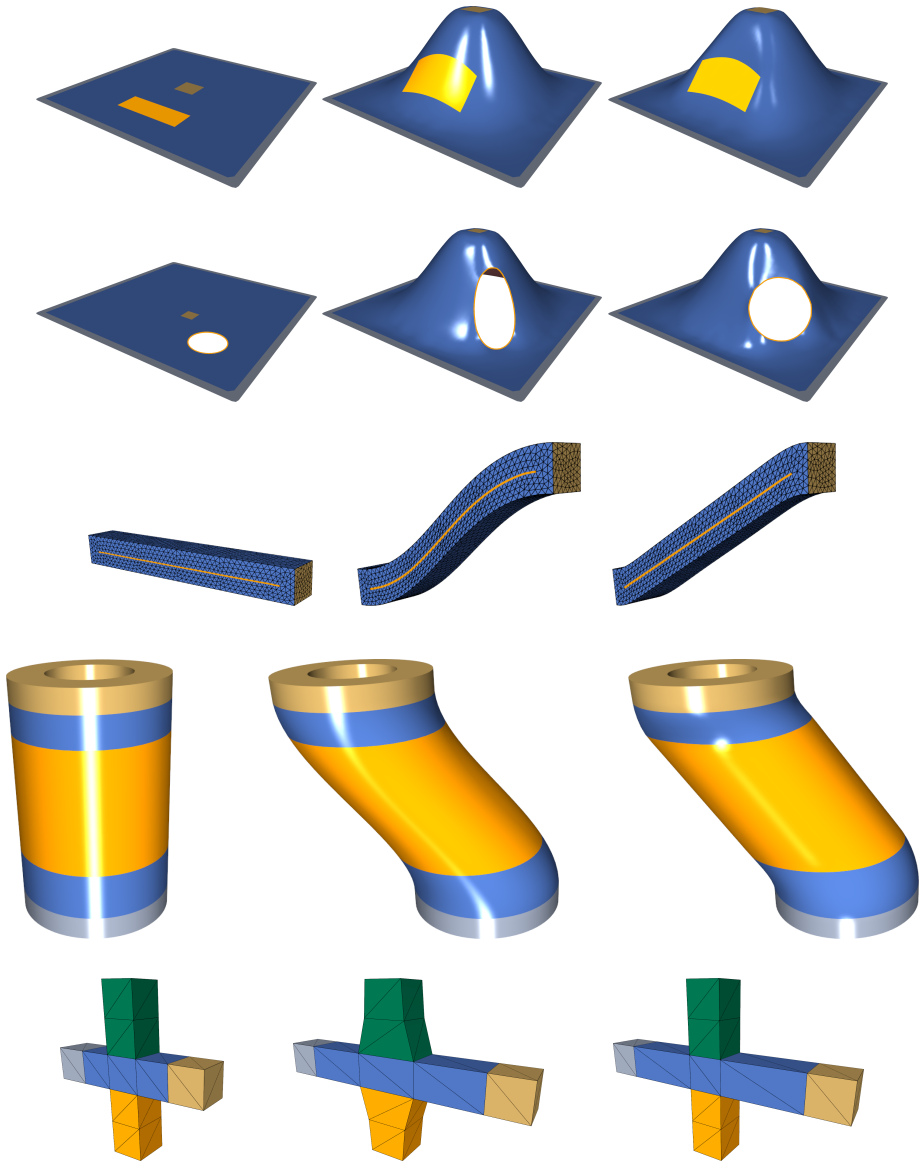


Figure 8.6: Synthetic constraint examples. For each constraint type (planarity, circularity, feature line, cylinder, rigidity) we show the original mesh, the deformation without constraint, and the deformation minimizing bending and constraint energies using $\gamma_b = 1.0$ and $\gamma_c = 10$ as weights.

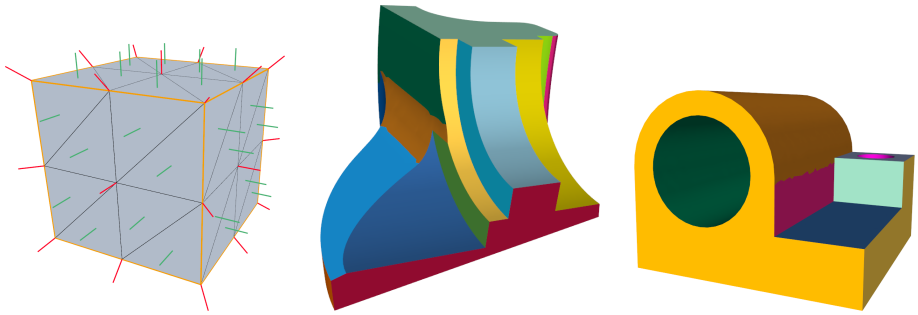


Figure 8.7: Left: Cube with sharp feature edges (orange), face (green) and vertex normals (red). Center/right: Automatically detected primitives in the Fandisk and Joint models. Each colored region corresponds to a plane or cylinder primitive.

possible to transfer information about geometric primitives from a corresponding CAD model, this is not necessarily the case in purely mesh-based modeling or reverse engineering scenarios.

The automatic detection of geometric primitives in point sets is a well-known problem in the context of surface reconstruction and segmentation. One of the most widely used techniques is the random sample consensus (RANSAC) algorithm introduced by Fischler and Bolles (1981). The core idea of this technique is to repeatedly draw random samples defining a geometric primitive from given point data and then to evaluate how well the primitive approximates the rest of the data set, see Roth and Levine (1993) for details. Our primitive detection is based on a combination of the efficient RANSAC algorithm described by Schnabel et al. (2007) and a forward search technique (Atkinson et al. 2004; Fleishman et al. 2005) to further refine the estimated primitives.

A key question when using the above techniques on a surface mesh is the choice of input data. Simply using mesh vertices and vertex normals leads to unsatisfactory results, since the normals at vertices on sharp feature edges are not aligned with the normal direction of the (multiple) geometric primitives such a vertex belongs to, see figure 8.7 left. We resolve this issue by using face barycenters and face normals as input instead. The detected primitives are then assigned to all vertices belonging to the corresponding faces, thereby allowing a vertex to belong to multiple primitives. Our system currently supports plane, cylinder, sphere, and cone primitives, see figure 8.7 for examples. From the set of detected primitives the user then selects those to be preserved during deformation.

8.6 RESULTS

In this section, we present different deformation results using our constrained space deformation technique. We use the Eigen (Guennebaud 2010) library for efficient matrix operations and the sparse Cholesky decomposition of CHOLMOD (Chen et al. 2008) for solving linear systems. We parallelize the evaluation of MLS basis functions and their analytical derivatives using OpenMP (Dagum and Menon 1998). In a typical modeling scenario satisfying the prescribed fixed and handle constraints is of highest importance and geometric constraints satisfaction is typically more important than smoothness minimization. Therefore, we select the weights balancing the individual constraint contributions such that $\gamma_f > \gamma_c > \gamma_{s/b}$, where $\gamma_f \approx 1000,\ \gamma_c \approx 10,\ \gamma_{s/b} \approx 1$. We also normalize the different weights by the number of constraints prescribed for a given type and region.

8.6.1 *Surface Deformation*

In this section, we present examples for constrained deformations on surface models of typical mechanical parts, such as they occur within design optimization scenarios. We begin with example deformations of the Fandisk model in figure 8.8. In this setup, we keep the bottom part of the model fixed and translate the handle region to the left. We select a subset of the sharp edges of the model as feature lines, and an additional planar constraint area in the upper left area. As becomes clear from the illustration, deforming the model without constraints distorts both feature lines and the planar region, whereas with constraints both of them are nicely preserved.

We illustrate example deformations of the Joint model in figure 8.9. We keep the bottom fixed, lift the top handle region, and impose a circularity constraint on the pipe-like opening. Without the constraint the opening would no longer fit with connecting parts, with the constraint, it does. The rightmost image in figure 8.9 shows the use of an additional planarity constraint. In this case, the initially already planar region deforms in such a way that the resulting mesh minimizes both the smoothness and planarity energies. In figure 8.10 we present a deformation example of the Joint model using constraints determined through our automatic geometric primitive detection. We keep the bottom fixed again and use the top cylinder region as a handle.

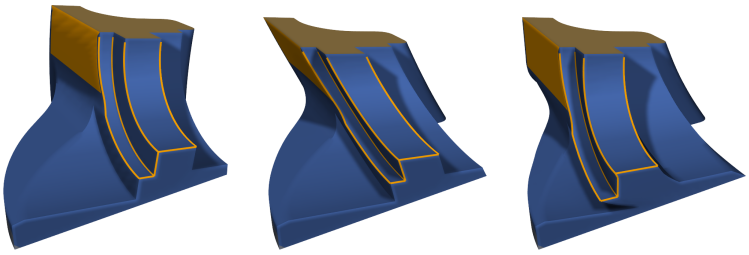


Figure 8.8: Deformation of the Fandisk model. From left to right: Original model, deformation without constraints, with feature line and planarity constraint.

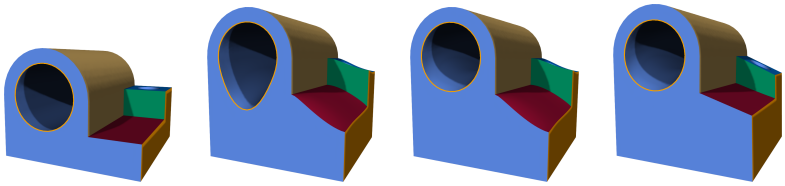


Figure 8.9: Deformation of the Joint model. From left to right: Original model, deformation without constraints, with a circularity constraint, and with a additional planarity constraints.

Figure 8.10: Deformation of the Joint model using automatically detected geometric primitives. Left: Original setup keeping the bottom fixed and using the top cylinder region (golden) as a handle. Right: Deformed model.

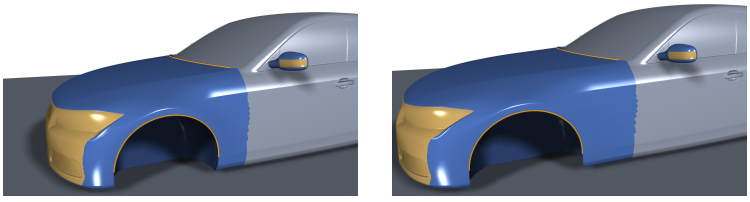


Figure 8.11: Deformation of the DrivAer model. Left: Original setup. Right: Stretching the front while keeping the wheelhouse circular.

Finally, as a more complex and application-oriented example, we include a deformation of the DrivAer (Heft et al. 2012) reference shape for car body aerodynamics in figure 8.11. The mesh contains 465k vertices, and we use 4k MLS samples to discretize the displacements and about 16k cubature points to discretize the deformation energies. As can be seen from the illustration, the circular shape of the wheelhouse is nicely preserved. We also note that deformations using global RBFs would not be easily applicable to this scenario: The number of user-prescribed handle and fixed constraints is much too high to be feasible for dense linear systems solvers. However, advanced RBF methods such as specialized incremental QR solvers (Botsch and Kobbelt 2005) or fast multipole methods (Carr et al. 2001; Wendland 2010) might still be applicable.

8.6.2 *Volume Deformation*

In this section, we compare the quality of our new volumetric mesh deformation method to that of our previously proposed RBF technique. We show an example deformation of a tetrahedral volume mesh containing 13k vertices in figure 8.12. In this setup, we keep the outer boundary fixed and use the interior sphere-shaped boundary as handle. We can see that both techniques allow for large deformations without resulting in inverted mesh elements. In order to provide a quantitative comparison to our previous results of chapter 7, we analyze mesh quality in terms of minimum scaled Jacobian. Our new method results in even slightly increased mesh quality (0.05) compared to our previous RBF deformations (0.03). Both methods produce similar results without inverted elements, as indicated by their still positive minimum scaled Jacobians. We refer to chapter 7 for a quantitative evaluation of mesh quality and element inversion of the RBF technique as well as other state-of-the-art deformation methods.

As an additional comparison to our previous results of chapter 7, we include a deformation example of the hexahedral Pipe model containing 11k vertices and 8.5k cells. We show the original and deformed meshes in figure 8.13. The original mesh has a minimum scaled Jacobian of 0.98. After performing one step of absolute deformation to the full parameter change—see chapter 7 for a description of absolute and relative deformation—the RBF deformation results in a mesh quality of 0.951, and our new method yields 0.954.

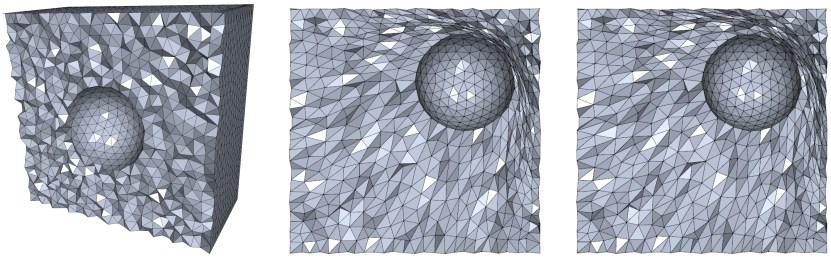


Figure 8.12: Comparison of volume mesh deformation quality in terms of min. scaled Jacobian. From left to right: The original mesh (0.12), a triharmonic RBF deformation (0.03), and our technique (0.05).

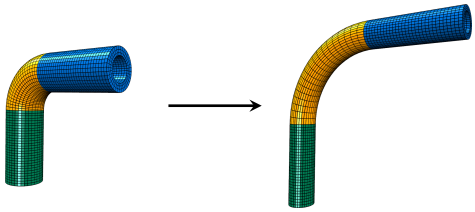


Figure 8.13: Deformation of the Pipe model, comparison in terms of minimum scaled Jacobian. Left: Original model (0.98). Right: Deformed model (0.954).

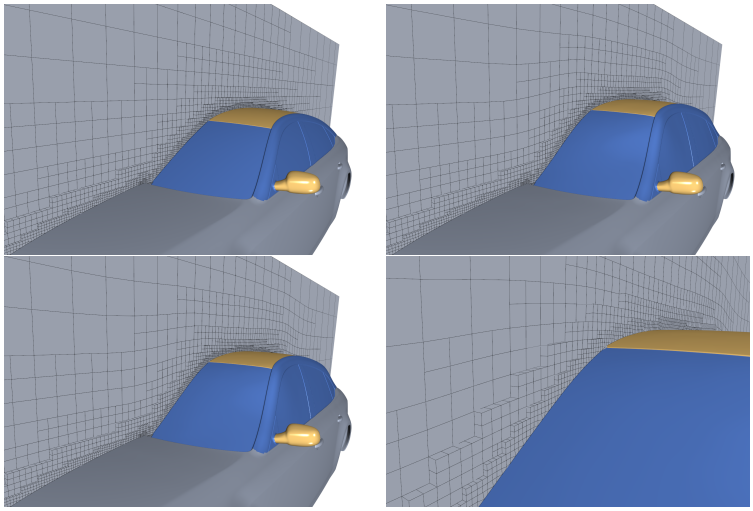


Figure 8.14: Combined volume and surface deformation of the DrivAer model. Top row: Original setup and deformation without constraints. Bottom row: Deformation with rigid shape constraints, close-up of boundary layer elements.

8.6.3 *Combined Surface and Volume Deformation*

As already noted in the introduction, an important feature of space deformation methods is the ability to deform an existing volumetric simulation mesh *along* with a surface. Therefore, our final example is a combined surface and volume mesh deformation of the DrivAer model, as illustrated in figure 8.14. In this case, we use a hex-dominant polyhedral volume mesh generated using OpenFOAM's `snappyHexMesh` utility, containing 1.3M vertices and 970k cells. We use 5k MLS samples on the surface as well as 2k samples in the volume to discretize displacements. Correspondingly, we use about 28k integration points to discretize our deformation energies. In order to better preserve the shape of boundary layer cells, we apply rigid shape constraints on those elements. For the deformation we lift the roof of the car model, which is a standard deformation in optimizing car body aerodynamics. After deformation the overall volume mesh quality is nicely preserved and the mesh is still usable for simulation, as we evaluated using OpenFOAM's `checkMesh` utility. The different meshes yield the following results for the important cell orthogonality check: The original mesh has a maximum value of 53.81, the deformed mesh without constraints has 54.52, and the mesh with rigid shape constraints enabled yields a value of 54.42 (smaller value indicating higher mesh quality). The less critical aspect ratio and faces skewness checks do not report significant differences.

8.7 Summary and Conclusion

In this chapter, we presented a novel space deformation technique based on MLS methods for its use in design optimization scenarios. Our method offers similarly high quality deformations as our previous RBF deformation technique, but with significantly increased scalability. Our space-based energy discretization allows for flexible modeling operations typically only provided by mesh-based deformation techniques. Even though our technique provides increased flexibility and scalability compared to RBF deformation, the implementation complexity increases as well. While RBF deformations simply require the solution of a dense linear system, our new technique involves Lloyd-relaxation in 3-space, numerical integration, more complex basis functions and derivatives, as well as the selection of several parameters such as the constraint weights, the number of sample points, or the basis function support radii.

Finally, by adapting and extending the projection-based constraint processing technique of Bouaziz et al. (2012), we also provide a concrete answer to the third research question set out for this thesis: *How can we incorporate additional constraints into the deformation*? Our choice of a constraint processing technique that is able to maintain constraints on arbitrary geometric data sets allows for a seamless integration into our space deformation framework. Our alternative formulation of projective constraints increases the scalability of the technique to modeling scenarios involving large constraint regions, such as required within practical design optimization tasks. Last but not least, our integration of a technique for the automatic detection of geometric primitives aids the designer or engineer during setup, thereby making our method more easily applicable.

CONCLUSION

We conclude this thesis by summarizing our main results and contributions, drawing conclusions from the results obtained, and by outlining promising directions for future research.

The primary goal of the first part of this thesis was to gain a better understanding of the specific requirements a deformation technique should satisfy in order to be successfully applicable in the context of design optimization. To this end, we presented a variety of state-of-the-art deformation methods and analyzed them in a series of both synthetic as well as application-oriented benchmarks. The major result of this investigation is a set of specific requirements for shape deformation methods in design optimization. In summary, the method should be a space deformation in order to be able to deal with a wide variety of geometry representations and to be robust against potential defects in the input data. It should perform high quality, smooth deformations based on the minimization of physically inspired energies in order to preserve the resulting mesh element quality. Finally, the method should be able to satisfy user-specified displacements exactly while allowing for the flexible placement of degrees of freedom without the need to generate and maintain complicated control structures. Based on these requirements, we gave a concrete recommendation for kernel-based space deformations based global triharmonic radial basis functions as the shape deformation technique of choice.

In the second part, we concentrated on the application and extension of RBF deformations to the specific needs within design optimization tasks. To this end, we introduced a unified framework for RBF deformation that allows for the simultaneous deformation of surface and volume meshes based on parameter changes in an initial CAD-based design prototype. A key idea of this framework is to exploit the flexibility of kernel-based scattered data approximation methods in order to warp the parametric coordinates of the surface nodes of a volumetric simulation mesh. In consequence, this component allows for the implementation of fully automatic CAD-based design optimization processes without the need for costly remeshing. It also confirms our previous hypothesis that the flexibility of kernel-based space deformations are highly beneficial for the implementation of

effective design optimization processes. We evaluated our framework by comparing it to other state-of-the-art methods and were able to show that our method more reliably achieves higher quality results. Finally, we extended our deformation framework by investigating the use of advanced linear solvers for the sake of improved performance and scalability as well as the use of a simple splitting technique in order to prevent inverted elements in the deformed meshes.

In the final third part of this thesis, we introduced a space deformation technique based on moving least squares approximation that improves upon our previous RBF deformations in crucial aspects: The explicit minimization of user-defined stretching and bending resistance provides a drastically increased modeling flexibility. The compactly supported MLS basis functions only require the solution of sparse linear systems and thereby significantly increase the scalability to complex modeling scenarios. The most important contribution, however, is the combination of our shape deformation technique with a method for maintaining geometric constraints, as well as the extension of the latter method to larger constraint sets. This combination allows for the creation of more feasible, meaningful, and producible designs during the optimization process. Furthermore, it prevents the costly creation and evaluation of infeasible designs, thereby also improving the general performance of the optimization process. The integration of an automatic method for discovering geometric primitives in the design further simplifies the setup procedure of the deformation method.

The majority of work presented in this thesis is the result of a research cooperation between Bielefeld University and the Honda Research Institute Europe. Therefore, the results we obtained are not only of academic or theoretical interest, but also support fundamental method research for practical applications. During the course of this cooperation, both our RBF shape deformations as well as parts of our constrained deformation technique have been integrated into in-house research software of Honda Research Institute Europe and is used for studying methods on practical design optimization problems.

Within each of the topics addressed within this thesis there are multiple directions for future work. As for the deformation methods investigated in our comparison we initially limited ourselves to linear deformation methods only—generally a reasonable assumption for design optimization. However, since in our constrained deformation technique we now already solve a non-linear optimization problem it might be worthwhile to extend this investigation to non-linear

methods as well, especially since a comprehensive comparison of such methods is currently missing from the literature.

The application-oriented benchmark of chapter 5 shows that the setup procedure of a deformation method is still a manual and tedious task. Therefore, research towards the automatic setup of the deformation method is a natural direction for future work (Richter et al. 2016). This could also include the incorporation of expert knowledge or statistical information such as sensitivity values obtained from previous optimization runs (Graening and Sendhoff 2014). Similarly, the adaptation of the deformation setup *during* the optimization procedure provides a promising direction for future research.

Although our unified deformation framework effectively enables CAD-based design optimization loops, there are still open questions regarding the more tight integration of CAD-based designs into the optimization loop. In the context of constrained deformation the direct transfer of geometric primitive information from the CAD model to the discrete model would be an obvious next step. In case no CAD-based initial prototype is available, e.g., in reverse engineering or rapid prototyping settings, the transfer of the optimization results back to a CAD model is a challenging task of its own. Our constrained deformation method with its integrated constraint detection constitutes a first starting point to re-create a CAD model based on geometric primitives.

A natural direction for future work is the integration of additional constraint types such as the maintenance of mutual distances between parts or the adherence to maximal or minimal widths and heights into our constrained deformation method. More advanced constraints could include relations between multiple parts, such as symmetry, orthogonality, or co-planarity, including methods for the automatic analysis of constraints using an approach similar to (Li et al. 2011). Additional constraints on the simulation meshes such as our rigidity constraint on boundary layer elements provide additional opportunities for further research. Finally, while we thoroughly evaluated our constrained deformation technique by comparing it to representative tests of our initial benchmarking and evaluation, we propose to further investigate the performance and effectiveness of our technique within an industrial-scale design optimization scenario such as the aerodynamic performance optimization of a passenger car.

APPENDIX

APPENDIX A

THE SURFACE MESH DATA STRUCTURE

In this appendix, we present the design, implementation, and evaluation of an efficient and easy to use data structure for polygon surface meshes. We systematically investigate the design choices arising during development and we give detailed reasons for choosing one alternative over another. We describe our implementation and compare it to other contemporary mesh data structures in terms of usability, computational performance, and memory consumption. Our evaluation demonstrates that our new `Surface_mesh` data structure is easier to use, offers higher performance, and consumes less memory than other state-of-the-art mesh data structures.

A.1 INTRODUCTION

Polygon meshes, or the more specialized triangle or quad meshes, are the standard discretization for two-manifold surfaces in 3D or solid structures in 2D. The design and implementation of mesh data structures therefore is of fundamental importance for research and development in as diverse fields as mesh generation and optimization, finite element analysis, computational geometry, computer graphics, and geometry processing.

Although the requirements on the mesh data structure vary from application to application, a generally useful and hence widely applicable data structure should be able to (i) represent vertices, edges, and triangular/quadrangular/polygonal faces, (ii) provide access to all incidence relations of these simplices, (iii) allow for modification of geometry (vertex positions) and topology (mesh connectivity), and (iv) allow to store any custom data with vertices, edges, and faces. In addition, the data structure should be easy to use, be computationally efficient, and have a low memory footprint.

Since it is hard to implement a mesh data structure that meets all these goals, many researchers and developers in both academia and industry rely on publicly available `C++` libraries like the Computational Geometry Algorithms Library (Fabri

et al. 1998), the Mesquite (Brewer et al. 2003) toolkit for mesh optimization, or OpenMesh (Botsch et al. 2002) for computer graphics.

However, even these highly successful data structures have their individual deficits and limitations, as we experienced during years of research and teaching in geometry processing. In this appendix, we systematically derive the design choices for our new `Surface_mesh` data structure and provide an analysis and comparison to the widely used mesh data structures of CGAL, Mesquite, and OpenMesh. These comparisons demonstrate that `Surface_mesh` is easier to use than these implementations, while at the same time being superior in terms of computational performance and memory consumption.

A.2 Related Work

Due to their fundamental nature, a wide variety of data structures to represent polygon meshes have been proposed. Some are highly specialized to only represent a certain type of polygons, such as triangles or quadrilateral elements. Others are designed for specific applications, e.g., parallel processing of huge data sets. In general, mesh data structures can be classified as being either *face-based* or *edge-based*. We refer the reader to Kettner (1999) and Botsch et al. (2010) for a comprehensive overview of mesh data structures for geometry processing.

In its most basic form a face-based data structure consists of a list of vertices and faces, where each face stores references to its defining vertices. However, such a simple representation does not provide efficient access to adjacency information of vertices or faces. Hence, many face-based approaches additionally store the neighboring faces of each face and/or the incident faces for each vertex. Examples for face-based mesh data structures include CGAL's 2D triangulation data structure (Pion and Yvinec 2015), Shewchuck's Triangle (Shewchuk 1996), Mesquite (Brewer et al. 2003), and the Visualization and Computer Graphics Library (VCGLib 2011).

In contrast to face-based approaches, edge-based data structures store the main connectivity information in edges or halfedges (Baumgart 1972; Guibas and Stolfi 1985; Campagna et al. 1998; Mäntylä 1987). In general, edges store references to incident vertices/faces as well as neighboring edges. Kettner (1999) gives a comparison of edge-based data structures and describes the design of CGAL's halfedge data structure. Botsch et al. (2002) introduce OpenMesh, a halfedge-based data structure widely used in computer graphics. Alumbaugh and Jiao (2005)

describe a compact data structure for representing surface and volume meshes by halfedges and half-faces.

Furthermore, a fairly large number of publications describe more specialized mesh representations. For instance, Blandford et al. (2005) introduce a compact and efficient representation of simplicial meshes containing triangles or tetrahedra. Other works focus on data structures for non-manifold meshes (De Floriani and Hui 2005; De Floriani et al. 2010), highly compact representations of static triangle meshes (Gurung, Laney, et al. 2011; Gurung, Luffel, et al. 2011), or mesh representations and databases for numerical simulation (Garimella 2004; Tautges et al. 2004; Seegyoung Seol and Shephard 2006; Edwards et al. 2010).

A.3 Design Decisions

While virtually all of the publications cited above describe the specific design decisions made for a particular implementation, a comprehensive and systematic investigation of the design choices available is currently lacking from the literature. We therefore try to provide such an analysis in this section.

As mentioned in the introduction, the typical design goals for mesh data structures are computational performance, low memory consumption, high flexibility and genericity, as well as ease of use. Since these criteria are partly contradicting, one has to set priorities and make certain compromises.

Based on our experience in academic research and teaching as well as in industrial cooperations, our primary design goal is *ease of use*. An easy-to-use data structure is learned faster, allows to focus on the main problem (instead of on the details of the data structure), and fosters code exchange between academic or industrial research partners. The data structure should therefore be just as flexible and generic as needed, but should otherwise be free of unnecessary switches and parameters. At the same time, however, we have to make sure not to compromise computational performance and memory consumption. Otherwise the data structure would be easy to use, but not useful, and hence would probably not be used at all. In the following, we systematically analyze the typical design choices one is faced with when designing a mesh data structure. Driven by our design goals we argue for choosing one alternative over another for each individual design criterion. We begin with high-level design choices and successively focus on more detailed questions.

A.3.1 *Element Types*

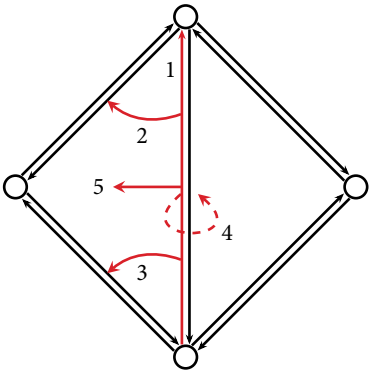
The most fundamental question is which types of elements or faces to support. While in computer graphics and geometry processing triangle meshes still are the predominant surface discretization (Botsch et al. 2010), quad meshes are at least as important as triangle meshes for structural mechanics. For many applications, restricting the supported element types to pure triangle or quad elements is not an option, though. Polygonal finite element methods (Sukumar and Malsch 2006) decompose their simulation domain into arbitrary polygons. In discrete exterior calculus many computations are performed on the dual mesh (Hirani 2003). In computational geometry, computations on Voronoi diagrams also need arbitrary polygon meshes (Kettner 2015a). Since we want our data structure to be suitable for an as wide as possible range of applications we choose to support *arbitrary polygonal elements*.

A.3.2 *Connectivity Representation*

As discussed in section A.2 there are at least two ways to represent the connectivity of a polygon mesh: a face-based or an edge-based representation.

Face-based data structures store for each face the references to its defining vertices. While this is sufficient for, e.g., visualization or setting up a FEM stiffness matrix, it is inefficient for mesh optimization, since vertex neighborhoods cannot be accessed easily. Some implementations therefore additionally store all incident faces per vertex (e.g., Brewer et al. (2003) and VCGLib (2011)), but even then it is still inefficient to enumerate all incident *vertices* of a center vertex—a query frequently required for many algorithms, such as mesh smoothing, decimation, or remeshing. Furthermore, since for a general polygon mesh the number of vertices per face and the number of incident faces per vertex are not constant, they have to be stored using dynamically allocated arrays or lists, which further complicates the data structure. Edges are typically not represented at all, making it difficult to implement algorithms operating on such entities.

In contrast, storing the main connectivity information in terms of edges or halfedges naturally handles arbitrary polygon meshes. The data types for vertices, (half-)edges, and faces all have constant size. The vertices and face-neighbors of a face can be efficiently enumerated, as well as the vertices or faces incident to a center vertex. Attaching additional data to vertices, halfedges, and faces is simple, since all



1. Target vertex
2. Next halfedge
3. Previous halfedge
4. Opposite halfedge
5. Adjacent face

Figure A.1: Connectivity relations within a halfedge data structure.

entities are explicitly represented. Finally, a halfedge-based data structure allows for simple and efficient implementation of connectivity modifications as required by modern approaches to interleaving mesh generation and optimization (Tournois et al. 2008; Sieger et al. 2010) or simulation (Wicke et al. 2010). We therefore choose a *halfedge data structure* to store the connectivity of a polygon mesh. The basic connectivity relations within a halfedge data structure are shown in figure A.1.

A.3.3 *Storage*

On an implementation level one has to decide whether to store the mesh entities in either doubly-linked lists or simple arrays.

Lists have the advantage that they allow for easy removal of individual vertices, edges, or faces, which is required, e.g., when collapsing edges or removing vertices in a mesh decimation algorithm. However, this flexibility comes at the price of higher memory consumption and less coherent memory layout compared to array-based storage, both resulting in considerable performance loss. We evaluated this on the halfedge data structure (Kettner 1998) of CGAL (2015b), which allows to switch between a list-based and an array-based implementation. Our benchmarks in section A.5 show that the list-based implementation is up to twice as slow as the array-based version.

Array-based storage on the one hand is more compact and faster, but on the other hand the removal of mesh entities is more difficult. Typically mesh entities are first marked as deleted and later removed by some form of garbage collection.

However, the advantages in terms of performance and memory consumption clearly outweigh the additional effort needed to support removal. For these reasons we choose an *array-based storage* scheme.

A.3.4 *Entity References*

When using array-based storage for mesh entities, references (or handles) to entities can be represented either as pointers or indices.

Pointers have three important drawbacks: First, they become invalid upon a relocation of the array, which happens if the array has to allocate more memory (e.g., for refinement or subdivision algorithms). While the data structure can automatically update all *internally* stored pointers, references that are stored externally by the user will inevitably become invalid. Second, on 64-bit architectures pointers consume twice as much memory as 32-bit indices. For larger meshes, however, one has to use 64-bit addressing, since complex meshes easily exceed the 2GB limit for 32-bit architectures. Finally, pointers cannot be used to access additional properties of mesh entities that are stored in additional synchronized "property arrays" (see the next section). We therefore choose *indices* as entity references.

A.3.5 *Custom Properties*

Additional information about the mesh entities can be stored either by extending the mesh entities themselves or by using additional arrays. For instance, vertex normals can be incorporated either by adding a member variable `normal` to the class `Vertex`, or by having an additional array `vertex_normals` where the `i`'th entry is the normal of vertex `i`.

The first approach, as e.g. chosen by CGAL, is more elegant from an object-oriented point of view, but has the following drawbacks: Since the class types of mesh entities are extended at compile-time, all custom properties are allocated over the whole running time of the application, even if the properties are used for a short time only. This does not only waste memory, it also slows down the algorithms due to a less compact memory layout: Just adding vertex and face normals to the CGAL mesh by extending the `Vertex` and `Facet` types slowed down our benchmarks (section A.5) by about 25% on average. This can be a significant drawback for larger mesh processing applications, where many individual algorithms need some custom data at some point in time.

In contrast, additional arrays can be dynamically allocated at run-time, such that custom properties are just allocated when needed and deleted afterwards (as implemented in OpenMesh and Mesquite). Keeping all property arrays synchronized upon resize and swap operations can easily be implemented. Furthermore, computations on the property arrays are also more cache-friendly, thereby increasing performance compared to extended mesh entities. Finally, if the model is meant to be visualized in an interactive application, property arrays can also be used in conjunction with OpenGL vertex arrays (normals, colors, texture coordinates), which speeds up rendering performance considerably. We therefore store custom properties in additional *synchronized arrays*.

A.3.6 *Ease of Use*

Up to this point, the OpenMesh data structure (Botsch et al. 2002), follows most of the design decisions made so far. From our experience in research and teaching, however, the level of genericity offered by OpenMesh is not needed in practice. For instance, custom properties can be allocated both by extending mesh entities as well as using additional arrays, where due to the former the mesh entities (and hence the whole mesh) become template classes. Furthermore, the large (template-parametrized) inheritance hierarchy makes the code unnecessarily hard to document and understand. In terms of C++ sophistication, the polyhedral data structure of CGAL requires an even higher level of template expertise, which makes it hard to use this data structure with students or inexperienced programmers, too.

To reduce the negative effect that heavy use of templates and complicated inheritance hierarchies have on the ease of use of the data structure, we made our design *as simple as possible* while maintaining maximum applicability.

A.4 Implementation

In the following, we highlight the most important aspects of our implementation. We first describe the fundamental organization of our new data structure and successively proceed to higher-level functionality.

Since OpenMesh already satisfies all design choices except simplicity, we started our implementation from a massively stripped-down and simplified version of OpenMesh. In contrast to other implementations, ours is concentrated within

a single class, namely `Surface_mesh`. While the core of `Surface_mesh` (without file I/O) is implemented in three files using about 2250 lines of code, the part of OpenMesh that implements the same functionality requires 41 files or 8400 lines of code. In contrast to CGAL and OpenMesh, `Surface_mesh` is not a class template, i.e., it does not require so-called traits classes as template parameters. However, the fundamental types `Scalar` and `Point` can still be defined by simple `typedef`s.

`Surface_mesh` implements an array-based halfedge data structure. The basic entities of the mesh, i.e., vertices, (half-)edges, and faces are represented by the types `Vertex`, `Halfedge`, `Edge`, and `Face`, respectively, all of which are basically 32-bit indices. Edges are represented implicitly, since two opposite halfedges (laid out consecutively in memory) build an edge.

The connectivity information is stored in form of custom properties (i.e., synchronized arrays) of vertices, faces, and halfedges: Each vertex stores an outgoing halfedge, each face an incident halfedge. Each halfedge stores its incident face, its target vertex, and its previous and next halfedges within the face. Since opposite halfedges are laid out consecutively in memory, the opposite halfedge can be accessed by simple modulo operations on the `Halfedge` indices and therefore does not have to be stored explicitly.

Managing internal mesh data as well as dynamically allocated user-defined properties within the same framework for synchronized arrays on the one hand simplifies implementing and maintaining the data structure. On the other hand the performance of the data structure then crucially depends on efficient access to these properties. Our property mechanism deviates from Mesquite and OpenMesh in that it (i) avoids inefficient virtual function calls, (ii) does not require error-prone casting of `void`-pointers, (iii) avoids unnecessary indirections, and (iv) offers a cleaner interface. From a user's point of view, working with a custom property is as simple as shown in listing A.1.

In addition to access to all incidence relations and custom properties, `Surface_mesh` also offers higher-level topological operations, such as adding vertices and faces, performing edge flips, edge splits, face splits, or halfedge collapses. Based on these methods typical geometry processing algorithms such as smoothing, decimation, subdivision, or remeshing can be implemented conveniently. Since `Surface_mesh` uses an array-based storage special care has to be taken when removing items from the mesh. Such operations do not delete mesh entities immediately, but instead mark them as being to be deleted. The function `garbage_collection()`

```
Surface_mesh mesh;

// allocate property storing a point per edge
Surface_mesh::Edge_property<Point> edge_points
= mesh.add_edge_property<Point>("property-name");

// access the edge property like an array
Surface_mesh::Edge e;
edge_points[e] = Point(x,y,z);

// remove property and free memory
mesh.remove_edge_property(edge_points);
```

Listing A.1: Working with a custom edge property.

eventually deletes those items from the arrays, while preserving the integrity of the data structure.

In order to sequentially access mesh entities, we provide iterators for each entity type, namely `Vertex_iterator`, `Halfedge_iterator`, `Edge_iterator` and `Face_iterator`. Each iterator stores a reference to the current entity and to the mesh. The latter is used to automatically detect and skip deleted entities, for instance when the user collapsed some edges but did not yet clean-up using `garbage_collection()`. We decided for these "safe iterators" despite their small performance penalty, since "unsafe" iterators turned out to be a frequent source of errors for novice OpenMesh users.

Similar to iterators, we also provide circulators for the ordered enumeration of all incident vertices, halfedges, or faces around a given face or vertex. Since there is no clear begin- and end-circulator, we follow the CGAL convention and use `do-while` loops for circulators. The traversal of the one-ring neighborhood of a vertex—which corresponds to a `Vertex_around_vertex_circulator`—is shown in figure A.2. An example usage of iterators and circulators is demonstrated in the smoothing example in listing A.2.

```cpp
#include <Surface_mesh.h>

int main(int argc, char** argv)
{
    Surface_mesh mesh;

    // read mesh from file
    mesh.read(argv[1]);

    // get (pre-defined) property storing vertex positions
    Surface_mesh::Vertex_property<Point> points
      = mesh.get_vertex_property<Point>("v:point");

    // iterators and circulators
    Surface_mesh::Vertex_iterator vit, vend = mesh.vertices_end();
    Surface_mesh::Vertex_around_vertex_circulator vc, vc_end;

    // loop over all vertices
    for (vit = mesh.vertices_begin(); vit != vend; ++vit)
    {
        if (!mesh.is_boundary(*vit))
        {
            // move vertex to barycenter of its neighbors
            Point  p(0,0,0);
            Scalar c(0);
            vc = vc_end = mesh.vertices(*vit);
            do
            {
                p += points[*vc];
                ++c;
            }
            while (++vc != vc_end);
            points[*vit] = p / c;
        }
    }

    // write mesh to file
    mesh.write(argv[2]);
}
```

Listing A.2: A simple smoothing program implemented using `Surface_mesh`.

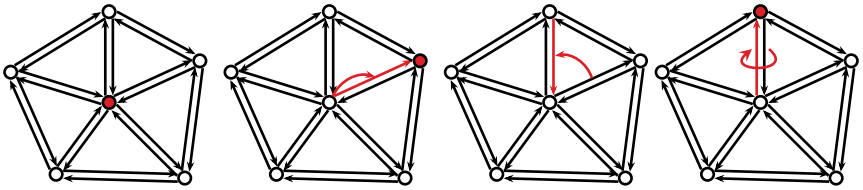


Figure A.2: Traversal of one-ring neighbors of a center vertex. From left to right: 1: Start from center vertex. 2: Select outgoing halfedge, access its target vertex. 3: Move to previous halfedge. 4: Move to opposite halfedge, access its target vertex. Steps 1 and 2 correspond to the initialization of a `Vertex_around_vertex_circulator` using `mesh.vertices(Vertex)`, steps 3 and 4 to the `++`-operator of the circulator.

A.5 Evaluation and Comparison

In this section, we evaluate our mesh data structure and compare it to three other widely used data structures: OpenMesh, CGAL, and Mesquite. Our evaluation criteria are ease of use, run-time performance, and memory usage.

All tests were performed on a Dell T7500 workstation with an Intel Xeon E5645 2.4 GHz CPU and 6GB RAM running Ubuntu Linux 10.04 x86_64. All libraries and tests were compiled with `gcc` version 4.4.3, optimization turned on (using `-O3`) and debugging checks disabled (`-DNDEBUG`).

For each of the mesh libraries in our comparison we used the latest version available at the time of performing our benchmarks, i.e., OpenMesh 2.0.1, CGAL 3.8, and Mesquite 2.1.4. To achieve comparable results, we chose double-precision floating point values for scalars, vertex coordinates, and normal vectors for all benchmarks and data structures. Since one benchmark requires vertex and face normals, all data structures allocate these properties, either by extending vertex and face types (CGAL) or using property arrays (Mesquite, OpenMesh, `Surface_mesh`).

Note that regarding CGAL we compare to both the list-based and the vector-based version of the `Polyhedron_3` mesh data structure, denoted as CGAL_list and CGAL_vector, respectively. Furthermore, following Shiue et al. (2004), we removed the storage for the plane equation from face entities in order to increase performance.

In contrast to the halfedge data structures of CGAL, OpenMesh, and `Surface_mesh`, Mesquite employs a face-based data structure that stores both downward adjacency (vertices of a face) and upward adjacency (faces of a vertex).

A.5.1 *Ease of Use*

Being our primary design goal, we begin our evaluation by comparing the ease of use of `Surface_mesh` to the other libraries.

As already outlined in section A.3.6, simplicity is a key criterion for the ease of use of a software library. By design, `Surface_mesh` is as simple as possible while maintaining high applicability. In contrast, both OpenMesh and CGAL offer a higher level of genericity. While this enables the customization of the mesh data structure for specialized applications, it also makes the library less accessible for students and inexperienced programmers. The differences in complexity are demonstrated best by example. In listing A.3 we show how to declare a custom halfedge property in CGAL, which is roughly equivalent to listing A.1 showing the usage of properties in `Surface_mesh`. It should become clear from comparing these examples that our implementation is considerably easier to grasp. Mesquite offers a similar level of simplicity, but a significantly reduced functionality (no connectivity modifications).

Compared to OpenMesh, our increased simplicity (and decreased genericity) is due to the definition of basic types (e.g., use `float` or `double` as scalar type, 2D or 3D vertex coordinates) through `typedef`s instead of through template parameters. While this allows `Surface_mesh` not to be a class template, it restricts each application to use a single `Surface_mesh` definition. In contrast, OpenMesh and CGAL allow for several custom-tailored template instances in a single application.

Comparing listings A.1 and A.3 not only serves as an example for evaluating simplicity, but also demonstrates the differences between CGAL's extended entities and `Surface_mesh`'s synchronized arrays for property handling. While the declaration of the former is rather involved and bound to compile-time properties, the latter is easy to use and dynamically allocated at run-time. Both OpenMesh and Mesquite also support dynamic property arrays. In case of OpenMesh however, the interface is slightly more complicated. Mesquite's implementation of properties relies on casting `void`-pointers, a practice generally discouraged and also relevant to our next evaluation criterion.

Especially for less experienced programmers protection against common sources of errors is a crucial aspect of usability. The use of `void` pointers in Mesquite mentioned above can be considered harmful in this context, since this practice essentially circumvents the static type-safety of the programming language. The use of pointers as entity references for CGAL's array-based mesh data structure is

```
typedef CGAL::Simple_cartesian<double> Kernel;
typedef Kernel::Point_3                Point_3;

template <class Refs>
struct My_halfedge : public CGAL::HalfedgeDS_halfedge_base<Refs>
{
    Point_3 halfedge_point;
};

class Items : public CGAL::Polyhedron_items_3
{
public:
    template <class Refs, class Traits>
    struct Halfedge_wrapper
    {
        typedef My_halfedge<Refs> Halfedge;
    };
};

typedef CGAL::Polyhedron_3<Kernel, Items>  Mesh;
```

Listing A.3: Declaring a custom halfedge property in CGAL.

prone to errors, since the pointers (and iterators) become invalid upon resizing. While OpenMesh uses safe, index-based entity references, its iterators by default do not skip deleted items, which turned out to be a common source of errors. In contrast, `Surface_mesh`'s implementation of safe iterators protects the user from iterating over deleted entities.

Finally, compilation time is a usability factor frequently overlooked. While the times to compile the individual programs in our test suite are relatively short, compilation time becomes a significant factor for the speed and efficiency of the development process in more complex projects. As can be seen from table A.1, `Surface_mesh` offers the fastest compilation times, mostly due to minimizing the use of templates.

We evaluated the usability of `Surface_mesh` in a user study among the participants of a two-day course of mesh processing (involving lectures and programming exercises) held at the Symposium on Geometry Processing 2011. The attendees had a varying degree of programming experience and exposure to other mesh libraries. After the two-days the participants were asked anonymously if `Surface_mesh`

	Time
Mesquite	1.57
CGAL_list	2.83
CGAL_vector	2.75
OpenMesh	3.37
Surface_mesh	1.13

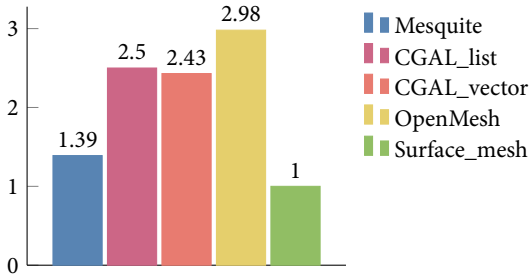


Table A.1: Compilation times (in seconds) of our benchmark program for the different mesh data structures. The chart shows timings relative to `Surface_mesh`.

was easy to use and understand for them. Out of 18 participants seven strongly agreed to this statement (5/5 points), another seven agreed (4/5 points). On average, `Surface_mesh` received 4.1/5 points. While this is not a representative survey, the results are still encouraging.

A.5.2 *Performance*

In order to compare the efficiency of our implementation with other mesh data structures we designed several benchmarks, which either evaluate a fundamental functionality of a data structure (e.g., iterators or adjacency queries) or test the performance in common application domains (e.g., mesh smoothing or subdivision). The benchmark tests are described below and their pseudo-code is shown in algorithms 1–6:

1. Circulator Test: For each vertex enumerate its incident faces. For each face enumerate its vertices. This test measures the efficiency of iterators and circulators.

2. Barycenter Test: Center the mesh at the origin by first computing the barycenter of all vertex positions and then subtracting it from each vertex. This test evaluates the performance of iterators and of the access to and basic computations on the vertex coordinates.

3. Normal Test: First compute (and store) face normals, then compute vertex normals as the average of the incident faces' normals. This test measures

the performance of iterators, circulators, vertex computations, and custom properties (storing face and vertex normals).

4. Smoothing Test: Perform Laplacian smoothing by iteratively moving each (non-boundary) vertex to the barycenter of its neighboring vertices. This test requires (and evaluates) the enumeration of incident vertices of a vertex.

5. Subdivision Test: Perform one step of $\sqrt{3}$-subdivision (Kobbelt 2000) by first splitting all faces at their centers, smoothing the old vertices, and then flipping all the old edges. This test mainly evaluates the performance of the face split and edge flip operators.

6. Edge Collapse Test: First split all faces at their center and then collapse each newly introduced vertex into one of its (old) neighbors, thereby restoring the original connectivity. This test evaluates the operators face split and halfedge collapse.

These benchmarks were performed on the Imp model, consisting of 300k vertices and 600k triangles, and the Dual Dragon model, a dualized triangle mesh consisting of 100k vertices and 50k polygonal faces. The models are shown in figure A.4. All tests were iterated sufficiently many times in order to get more reliable accumulated timings. The results are listed in tables A.2 and A.3. Note that we also performed the tests with other models and setups (CPU, compiler, operating system). While the results quantitatively vary to a certain extent, they were qualitatively equivalent to those we report here.

It can be observed that for some tests the performance varies significantly between different libraries. While it is hard to track down the reasons in detail, we point out the most important issues we identified.

For Mesquite, a significant performance penalty comes from the large number of virtual functions (e.g., to access incidences or vertex coordinates), as well as from memory fragmentation due to dynamically allocated arrays for storing per-vertex and per-face incidences. Moreover, enumerating incident *vertices* of a center vertex is not directly supported by this face-based data structure and therefore has to be implemented less efficiently by looping over the vertices of the incident faces. Since Mesquite does not support connectivity modifications, the subdivision and collapse test were not implemented.

The performance difference between CGAL_list and CGAL_vector is due to the higher memory consumption and fragmentation of the list-based version. Both

Algorithm 1: Circulator Test

```
Initialize counter = 0;
for each vertex v do
    for each face f incident to v do
        counter = counter + 1;
    end
end
for each face f do
    for each vertex v incident to f do
        counter = counter − 1;
    end
end
```

Algorithm 2: Barycenter Test

```
Initialize p = [0, 0, 0];
for each vertex v do
    p = p + point(v);
end
p = p/number_of_vertices();
for each vertex v do
    point(v) = point(v) − p;
end
```

Algorithm 3: Normal Test

```
for each face f do
    Compute the face normal of f;
end
for each vertex v do
    n = [0, 0, 0];
    for each face f incident to v do
        n = n + face_normal(f);
    end
    vertex_normal(v) = normalize(n);
end
```

Algorithm 4: Smoothing Test

```
for each vertex v do
    if v is not a boundary vertex then
        p = [0, 0, 0];
        c = 0;
        for each vertex w incident to v do
            p = p + point(w);
            c = c + 1;
        end
        point(v) = p/c;
    end
end
```

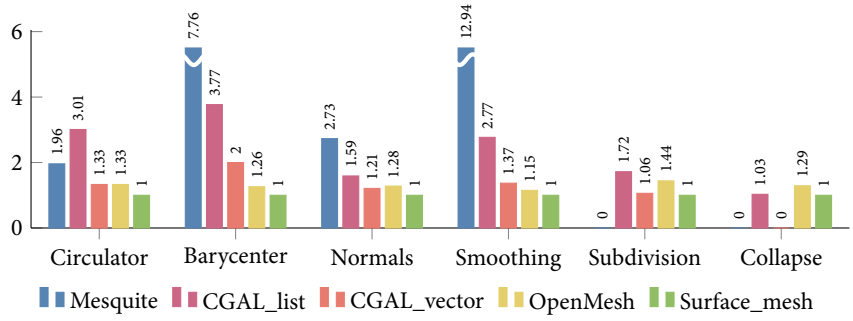
Algorithm 5: Subdivision Test

```
for each face f do
    Compute centroid c;
    Split f at centroid c;
end
for each old vertex v do
    Smooth vertex position;
end
for each old edge e do
    Flip e;
end
```

Algorithm 6: Collapse Test

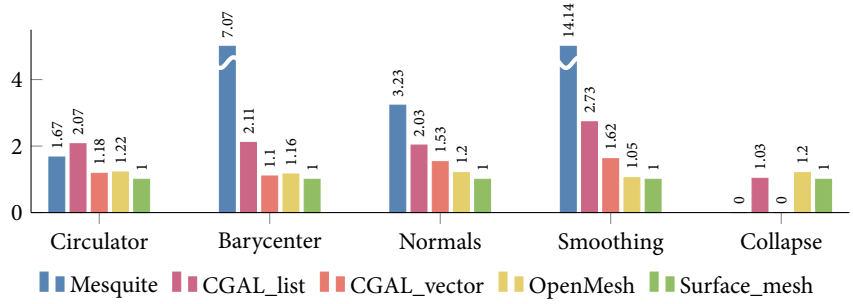
```
for each face f do
    Split f;
end
for each new vertex v do
    Collapse v into one of its neighbors;
end
```

Figure A.3: The six benchmark tests used to evaluate and compare the run-time performance of `Surface_mesh` to Mesquite, CGAL, and OpenMesh.



	Circulator	Barycenter	Normals	Smoothing	Subdivision	Collapse
Mesquite	3479	15039	11406	23228	—	—
CGAL_list	5329	7298	6642	4976	506	1582
CGAL_vector	2358	3879	5064	2467	312	—
OpenMesh	2359	2443	5356	2071	423	1987
Surface_mesh	1673	1412	4181	1757	294	1547

Table A.2: Timings for performing Algorithms 1–6 on the Imp model. The table lists timings in ms, the chart visualizes the performance relative to `Surface_mesh`.



	Circulator	Barycenter	Normals	Smoothing	Collapse
Mesquite	650	4632	2234	5554	—
CGAL_list	804	1381	1403	1070	74
CGAL_vector	460	718	1057	636	—
OpenMesh	475	760	830	410	86
Surface_mesh	388	655	690	392	71

Table A.3: Timings for performing Algorithms 1–4 and 6 on the Dual Dragon model. The table lists timings in ms, the chart shows performance relative to `Surface_mesh`.

Figure A.4: The three models used in the evaluation. From left to right: Imp model, 300k vertices, 600k triangles. Lucy model, 10M vertices, 20M triangles. Dual Dragon model, 100k vertices, 50k polygons.

CGAL mesh data structures store 64-bit references, vertex positions, and normal vectors in extended mesh entities, leading to a less compact memory layout, which in turn results in performance penalties. Note that the array-based version does not support removal of entities, so that the collapse test could be implemented with the slower list-based version only.

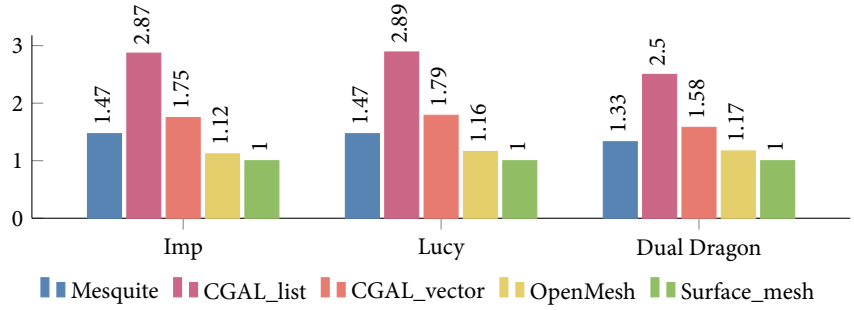
Since OpenMesh is closest to `Surface_mesh` in terms of design and implementation, it also is close in terms of performance. The differences of about 20–30% are due to our more efficient mechanism for accessing custom properties, which requires fewer indirections. Furthermore, our `do–while` circulators are slightly more efficient than the `for` circulators of OpenMesh, which use a rather complex test for detecting the end of the loop.

The results clearly demonstrate the performance of `Surface_mesh` to be (in most cases) superior to or at least on par with the other data structures.

A.5.3 *Memory Efficiency*

Besides run-time performance, memory consumption is a key criterion to measure the efficiency of a library, especially when it comes to applications dealing with highly complex data sets. We compare the memory consumption of the data structures on three different models: the Imp model (300k vertices, 600k triangles) and the Dual Dragon (100k vertices, 50k polygons) already used in the performance comparison and the complex Lucy model (10M vertices, 20M triangles). We present

the results in table A.4, both in terms of absolute numbers as well as relative difference between the data structures.



	Imp	Dragon	Lucy
Mesquite	88M	16M	2.8G
CGAL_list	172M	30M	5.5G
CGAL_vector	105M	19M	3.4G
OpenMesh	67M	14M	2.2G
Surface_mesh	60M	12M	1.9G

Table A.4: Memory usage for the Imp, Lucy, and Dual Dragon models. The table lists resident size memory usage after reading the meshes, without performing any further tests or processing. The chart visualizes the relative difference to `Surface_mesh`.

Although a face-based data structure in general consumes less memory than a halfedge data structure, Mesquite requires more memory than `Surface_mesh` because (i) of the overhead of the dynamic arrays used to store incidences, (ii) the use of 64-bit references, and (iii) the storage of helper data per face and vertex. In addition to the memory overhead due to the doubly-linked list of the CGAL list-kernel, both CGAL data structures use 64-bit pointers as references, which consume twice as much memory than the 32-bit indices employed by OpenMesh and `Surface_mesh`. Our slight performance advantage with respect to OpenMesh comes from the different storage of the information whether a vertex, edge, or face is deleted. We store this information in custom `bool` property arrays, which in a `std::vector<bool>` require approximately 1 bit per entity. In contrast, OpenMesh uses one status byte per entity, similar to Mesquite. These results show that `Surface_mesh` is superior to other data structures in terms of memory consumption.

A.6 SUMMARY AND CONCLUSION

Our results show that the design decisions made during the development of a mesh data structure have a crucial impact on both the usability and the efficiency of the library. By systematically analyzing the design questions we derived design decisions that—if carefully implemented—result in a mesh data structure that is more usable, offers higher performance and consumes less memory than several other mesh data structures publicly available.

Considering the sometimes drastic differences in performance and memory consumption between the individual libraries, it is important to keep in mind that some of them have originally been designed and implemented with a strong focus on a given application domain, such as computational geometry in case of CGAL and mesh optimization in case of Mesquite. As a consequence, both libraries provide significantly more functionality that goes beyond a pure surface mesh data structure, e.g., Mesquite supports the optimization of surface and volume meshes within a single framework.

While we are confident with the tests and results achieved thus far, we feel that our benchmark tests should be expanded to a wider variety of different setups (i.e. different hardware, operating systems, compilers and mesh models). Furthermore, additional algorithms and additional mesh data structures, for instance VCGLib (2011), `libigl` (Jacobson, Panozzo, et al. 2016), or VTK (Kitware 2010) could be included in future evaluations.

Our performance and memory benchmarks are a first step towards a general benchmark for mesh data structures. We made the source code and the results of the benchmarks publicly available. Furthermore, in order to facilitate wide adoption of our new data structure, we also made `Surface_mesh` freely available under an Open Source license allowing for both academic and commercial usage. Following our original publication of the data structure it has since been integrated into the CGAL library as a more efficient alternative to the original halfedge data structure (Botsch et al. 2015).

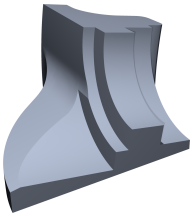
While our current work is focused on surface meshes only, we are aware that applications such as physical simulations require volume meshes. We feel that a systematic approach as presented in this appendix equally applies to the design and implementation of volume mesh data structures. In particular, design decisions such as array-based storage, indices as entity references and custom properties as synchronized arrays should carry over to such a data structure seamlessly.

APPENDIX B

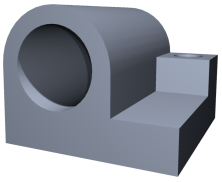
DATA SOURCES

This appendix acknowledges the origins of models used throughout this thesis that were neither created by the author nor one of his colleagues of the Computer Graphics and Geometry Processing Group at Bielefeld University:

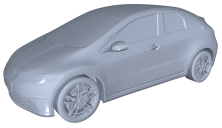
The Fandisk model is courtesy of Hughes Hoppe, Microsoft Research, USA.



The Joint model is courtesy of Pierre Alliez, Inria Sophia-Antipolis, France.

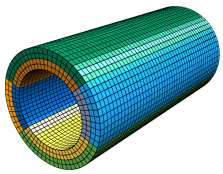


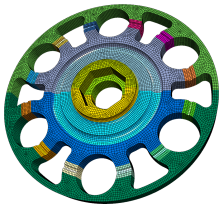
The Civic model is courtesy of the Honda Research Institute Europe GmbH, Germany.

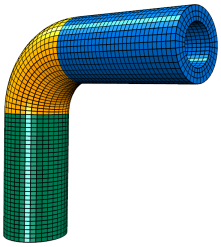


The DrivAer model is courtesy of the Institute for Aerodynamics and Fluid Mechanics at TU Munich, Germany.

The Bore, Pipe, and Courier models are courtesy of Matthew Staten, Sandia National Laboratories, USA.







The Lucy model is courtesy of the Stanford University Computer Graphics Laboratory.

The Dragon model is courtesy of XYZ RGB Inc.

LIST OF PUBLICATIONS

The results and ideas presented in this thesis are partially based on the following previous publications of the author:

Sieger, Daniel, Sergius Gaulik, Jascha Achenbach, Stefan Menzel, and Mario Botsch. 2016. "Constrained Space Deformation Techniques for Design Optimization." *Computer-Aided Design* 72:40–51. doi:10.1016/j.cad.2015.07.004.

Sieger, Daniel, Stefan Menzel, and Mario Botsch. 2015. "On Shape Deformation Techniques for Simulation-Based Design Optimization." In *New Challenges in Grid Generation and Adaptivity for Scientific Computing,* edited by Simona Perotto and Luca Formaggia, 5:281–303. SEMA SIMAI Springer Series. Springer International Publishing. doi:10.1007/978-3-319-06053-8_14.

Sieger, Daniel, Stefan Menzel, and Mario Botsch. 2014a. "Constrained Space Deformation for Design Optimization." Proceedings of the 23rd International Meshing Roundtable. *Best Technical Paper. Procedia Engineering* 82:114–126. doi:10.1016/j.proeng.2014.10.377.

Sieger, Daniel, Stefan Menzel, and Mario Botsch. 2014b. "RBF Morphing Techniques for Simulation-based Design Optimization." *Engineering with Computers* 30 (2): 161–174. doi:10.1007/s00366-013-0330-1.

Sieger, Daniel, Stefan Menzel, and Mario Botsch. 2012a. "A Comprehensive Comparison of Shape Deformation Methods in Evolutionary Design Optimization." In *Proceedings of the 3rd International Conference on Engineering Optimization,* edited by José Herskovits.

Sieger, Daniel, Stefan Menzel, and Mario Botsch. 2012b. "High Quality Mesh Morphing Using Triharmonic Radial Basis Functions." In *Proceedings of the 21st International Meshing Roundtable,* edited by Xiangmin Jiao and Jean-Christophe Weill, 1–15. Berlin: Springer-Verlag. doi:10.1007/978-3-642-33573-0_1.

Sieger, Daniel, and Mario Botsch. 2011. "Design, Implementation, and Evaluation of the `Surface_mesh` data structure." In *Proceedings of the 20th International Meshing Roundtable,* edited by William Roshan Quadros, 533–550. Berlin: Springer-Verlag. doi:10.1007/978-3-642-24734-7_29.

Sieger, Daniel, Pierre Alliez, and Mario Botsch. 2010. "Optimizing Voronoi Diagrams for Polygonal Finite Element Computations." In *Proceedings of the 19th International Meshing Roundtable,* edited by Suzanne M. Shontz, 335–350. Berlin: Springer-Verlag. doi:10.1007/978-3-642-15414-0_20.

NOMENCLATURE

A	area. 100, 101
$\boldsymbol{a}$	displacement of a deformable vertex. 41
$\boldsymbol{A}$	matrix of deformable point displacements. 41, 42
$\boldsymbol{B}$	matrix of prescribed displacements. 23, 27, 28, 37, 41, 42
$\boldsymbol{b}$	prescribed displacement. 23, 26, 27, 34–37, 41
$\boldsymbol{b}_{\text{max}}$	maximum point of the bounding box. 58
$\boldsymbol{b}_{\text{min}}$	minimum point of the bounding box. 58
C^2	function space of twice differentiable functions. 36, 38, 47, 72, 103
$\boldsymbol{c}$	control point, kernel, center, sample, cage vertex. 14, 25, 29, 35, 36, 41, 71, 72, 90, 103, 104, 107
C^∞	function space of infinitely differentiable functions. 35
$\boldsymbol{C}$	constraint matrix. 110–112
$\mathcal{C}$	constraint set. 110–112
$\boldsymbol{d}$	deformation function. 9, 10, 12, 14, 22, 23, 26, 34, 36, 41, 70–72, 80–82, 99–102, 108
$\boldsymbol{d}_{\text{cages}}$	cage-based deformation function. 29, 30
$\boldsymbol{D}$	matrix of displacements. 23, 27, 28, 41, 100–102, 108, 111
$\mathcal{D}$	deformable region of a model. 10, 99
$\boldsymbol{\delta}$	displacement vector. 9, 12, 23, 25, 27–29, 41, 100–102, 111
$\boldsymbol{d}_{\text{ffd}}$	FFD deformation function. 25, 27
$D(\boldsymbol{g})$	aerodynamic drag of a parameter vector $\boldsymbol{g}$. 57
$\boldsymbol{d}_{\text{rbf}}$	RBF deformation function. 34, 35, 41
ϵ	shape parameter of a radial basis function. 35
E_{bend}	bending energy. 99, 100, 108
E_{constr}	constraint energy. 110
E_{fix}	fixed energy. 99, 100

E_{stretch}	stretching energy. 99, 100, 108
E_{shell}	thin shell energy. 99, 100
$\mathcal{F}$	fixed region on a model. 10, 23, 26, 99
f_{fit}	fitness function of a population. 57
$\boldsymbol{f}$	parametric face $\boldsymbol{f}: \Gamma \to \mathbb{R}^3$. 68–71
$\boldsymbol{F}$	fixed constraints matrix. 100–102, 108, 111
$\boldsymbol{g}$	parameter vector in evolutionary optimization. 57, 58, 108
Γ	parameter domain of a parametric face $\boldsymbol{f}$. 68, 70
γ_b	bending weight. 22, 23, 99–102, 108, 111, 113
γ_c	constraint weight. 111, 113, 115
γ_f	fixed weight. 99–102, 108, 111, 115
γ_s	stretching weight. 22, 23, 99–102, 108, 111
$\mathcal{G}$	continuous geometry description such as a spline-based representation of a CAD model. 67–69, 71, 89
∇	gradient operator. 100, 108
$\boldsymbol{G}$	gradient matrix. 100–102, 108, 111
$\boldsymbol{h}$	handle point $\in \mathcal{H} \cup \mathcal{F}$. 23, 26, 27, 34, 36, 37, 41, 70
$\mathcal{H}$	handle region selected for manipulation. 10, 23, 26, 99
H	mean curvature. 45
$\mathbf{I}$	first fundamental form of a parametric surface $\mathcal{S}$. 22
$\mathbf{II}$	second fundamental form of a parametric surface $\mathcal{S}$. 22
κ	principal curvatures. 45
$\boldsymbol{L}$	Laplacian matrix. 23, 41, 100–102, 108, 111
Δ	Laplace operator. 23, 36, 45, 100, 108
$\mathcal{M}$	discrete geometric model such as a point set or mesh. 12, 25, 27–30, 34, 37, 109
$\boldsymbol{M}$	moment matrix. 104, 106
$\boldsymbol{n}$	curve node. 68
Ω	embedding space around an object. 12, 101, 107
P	projection onto a constraint set $\mathcal{C}$. 110, 111
$\boldsymbol{p}$	vector of monomials $\boldsymbol{p}(x, y, z) = [1, x, y, z]^{\mathrm{T}}$. 104
$\boldsymbol{\Phi}$	basis function matrix. 27, 37, 40–42, 72, 91, 102, 103, 106, 108, 111

φ	basis function such as RBF, MLS, or splines. 25, 27, 29, 35–37, 40–42, 71, 72, 79, 88, 91, 101–104, 106, 108
π	polynomial basis. 35, 37, 70, 72
$\boldsymbol{\Psi}$	handle basis function matrix. 41, 42
ψ	piecewise linear shape functions on a triangulation $\mathcal{T}$. 100
$\boldsymbol{Q}$	orthogonal matrix from QR factorization. 91
$\boldsymbol{q}$	polynomial coefficients. 35, 37, 70, 72
$\mathcal{R}$	control cage. 28, 29
$\boldsymbol{r}$	sample point. 69, 70
ρ	support radius of a basis function. 103, 104
$\boldsymbol{R}$	upper triangular matrix from QR factorization. 91
s	minimum number of basis functions covering each point $\boldsymbol{x}$ of a geometry during cover construction. 103, 104
$\boldsymbol{\Sigma}$	diagonal matrix of singular values σ obtained from SVD. 27, 40, 106
σ	singular value. 27, 44
$\boldsymbol{S}$	matrix of prescribed surface node displacements. 72, 91
$\boldsymbol{s}$	the boundary surface nodes of a volume mesh. 68, 69, 71, 72, 90
$\mathcal{S}$	smooth parametric surface. 22, 99, 100
$\boldsymbol{t}$	integration points. 107, 108
$\mathcal{T}$	triangular surface mesh representing a continuous surface $\mathcal{S}$. 9, 10, 22, 23, 45, 100–102, 107
$\boldsymbol{\Theta}$	basis function matrix. 41, 42
$\boldsymbol{U}$	orthogonal matrix resulting from SVD. 27, 40, 106
$\boldsymbol{u}$	vector of local coordinates of a point $\boldsymbol{x}$ with regards to a control lattice. 25, 27, 41
$\boldsymbol{V}$	orthogonal matrix resulting from SVD. 27, 40, 106
v	vertex of a mesh. 9, 45
$\mathcal{V}$	volume mesh. 67, 68, 71, 72, 80–82, 89
$V(\boldsymbol{g})$	volume term for a parameter vector $\boldsymbol{g}$. 57, 58, 108
$\boldsymbol{v}$	interior nodes of a volume mesh $\mathcal{V}$. 71, 72
$\boldsymbol{W}$	weight matrix. 37, 42, 72, 91, 102, 108, 111

$\boldsymbol{w}$	weight vector. 35, 37, 41, 70, 72, 101, 102, 108
w_d	aerodynamic drag weight. 57, 58
w	compactly supported weight function. 104
w_v	volume weight. 57, 58
$\boldsymbol{x}$	arbitrary point in 3D space. 9, 10, 12, 25, 29, 35, 37, 41, 42, 45, 71, 72, 80–82, 100–104, 109, 111, 112
$\boldsymbol{X}$	matrix of stacked point positions $\boldsymbol{x}$. 109–111
$\boldsymbol{y}$	parametric curve $\boldsymbol{y}\colon \mathbb{R} \to \mathbb{R}^3$. 68
$\mathbf{0}$	zero vector. 23, 37, 72

ACRONYMS

CAD	Computer-aided Design. 1, 2, 4, 5, 7, 9, 63, 65–71, 73, 76, 77, 79, 84, 87, 89, 92, 114, 121, 123
CAM	Computer-aided Manufacturing. 1
CFD	Computational Fluid Dynamics. 2, 55, 57, 58, 83, 84, 109, 112
CMA-ES	Covariance Matrix Adaptation Evolution Strategies. 55, 56
DM-Cages	Direct Manipulation Cage Deformation. 43, 45, 46, 48, 49, 51
DM-FFD	Direct Manipulation Free-form Deformation. 26, 40, 41, 43–46, 48, 49, 51, 57, 58, 83, 86
DoF	Degree of Freedom. 14, 47–50
ES	Evolution Strategies. 55
FEM	Finite Element Method. 2, 55, 109, 130
FEMWARP	FEM-based deformation method of Baker (2002). 12, 77, 92
FFD	Free-form Deformation. 24–30, 38–43, 45–47, 50–52, 56–58, 60, 83
GPU	Graphics Processing Unit. 89–92
IQR	Incremental QR. 90, 91
LBWARP	Smoothing-based deformation method of Shontz and Vavasis (2003). 11, 77, 79, 92
MAGMA	Matrix Algebra on GPU and Multicore Architectures. 89–91
MLS	Moving Least Squares. 98, 102, 104–109, 115, 117, 119, 122
RBF	Radial Basis Function. 17, 34–38, 40–53, 57–61, 65, 66, 70–73, 75, 77, 79, 83–87, 89–92, 97, 98, 102–107, 117–119, 121, 122
SVD	Singular Value Decomposition. 27, 29, 40, 106

BIBLIOGRAPHY

Alumbaugh, Tyler J., and Xiangmin Jiao. 2005. "Compact Array-based Mesh Data Structures." In *Proceedings of the 14th International Meshing Roundtable,* 485–503. Berlin: Springer. doi:10.1007/3-540-29090-7_29.

Anderson, Edward, Zhaojun Bai, Christian Bischof, Laura Susan Blackford, James Demmel, Jack Dongarra, Jeremy J. Du Croz, et al. 1999. *LAPACK Users' Guide (Third Ed.)* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Anderson, George R., Michael J. Aftosmis, and Marian Nemec. 2012. "Constraint-based Shape Parameterization for Aerodynamic Design." In *Proceedings of the Seventh International Conference on Computational Fluid Dynamics.*

Angelidis, Alexis, and Karan Singh. 2006. "Space Deformations and Their Application to Shape Modeling." In *ACM SIGGRAPH 2006 Courses,* 10–29. SIGGRAPH '06. Boston, Massachusetts: ACM. doi:10.1145/1185657.1185672.

Atkinson, Anthony C., Marco Riani, and Andrea Cerioli. 2004. *Exploring Multivariate Data with the Forward Search.* Springer Series in Statistics. Springer. doi:10.1007/978-0-387-21840-3.

Bäck, Thomas. 1996. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press.

Bäck, Thomas, and Hans-Paul Schwefel. 1993. "An Overview of Evolutionary Algorithms for Parameter Optimization." *Evolutionary Computation* 1 (1): 1–23. doi:10.1162/evco.1993.1.1.1.

Baker, Timothy J. 2002. "Mesh Movement and Metamorphosis." *Engineering with Computers* 18 (3): 188–198. doi:10.1007/s003660200017.

Baumgart, Bruce G. 1972. *Winged-Edge Polyhedron Representation.* Technical report STAN-CS320. Computer Science Department, Stanford University.

Bechmann, Dominique. 1994. "Space Deformation Models Survey." *Computers & Graphics* 18 (4): 571–586. doi:10.1016/0097-8493(94)90071-X.

Beyer, Hans-Georg, and Hans-Paul Schwefel. 2002. "Evolution Strategies—A Comprehensive Introduction." *Natural Computing* 1 (1): 3–52. doi:10.1023/A:1015059928466.

Biancolini, Marco Evangelos. 2015. *RBF-Morph.* http://www.rbf-morph.com.

Blandford, Daniel K., Guy E. Blelloch, David E. Cardoze, and Clemens Kadow. 2005. "Compact Representations of Simplicial Meshes in two and three Dimensions." *International Journal of Computational Geometry & Applications* 15 (1): 3–24. doi:10.1142/S0218195905001580.

Blender. 2015. *Blender—A 3D Modelling and Rendering Package.* Blender Foundation. `http://www.blender.org`.

Boer, Aukje de, Martijn van der Schoot, and Hester Bijl. 2007. "Mesh Deformation Based on Radial Basis Function Interpolation." *Computers & Structures* 85 (11–14): 784–795. doi:`10.1016/j.compstruc.2007.01.013`.

Botsch, Mario. 2008. *Lecture notes on Geometric Modeling Based on Polygonal Meshes.* Bielefeld University.

Botsch, Mario, David Bommes, and Leif Kobbelt. 2005. "Efficient Linear System Solvers for Mesh Processing." In *Proceedings of Mathematics of Surfaces XI,* edited by Ralph Martin, Helmut Bez, and Malcolm Sabin, 3604:62–83. Lecture Notes in Computer Science. Berlin: Springer. doi:`10.1007/11537908_5`.

Botsch, Mario, and Leif Kobbelt. 2004a. "A Remeshing Approach to Multiresolution Modeling." In *Proceedings of Eurographics Symposium on Geometry Processing,* 189–96. Eurographics Association. doi:`10.1145/1057432.1057457`.

Botsch, Mario, and Leif Kobbelt. 2004b. "An Intuitive Framework for Real-Time Freeform Modeling." *ACM Transaction on Graphics* 23 (3): 630–34. doi:`10.1145/1015706.1015772`.

Botsch, Mario, and Leif Kobbelt. 2005. "Real-Time Shape Editing using Radial Basis Functions." *Computer Graphics Forum* 24 (3): 611–621. doi:`10.1111/j.1467-8659.2005.00886.x`.

Botsch, Mario, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. 2010. *Polygon Mesh Processing.* AK Peters.

Botsch, Mario, Daniel Sieger, Philipp Moeller, and Andreas Fabri. 2015. "Surface Mesh." In *CGAL User and Reference Manual,* 4.7. CGAL Editorial Board.

Botsch, Mario, and Olga Sorkine. 2008. "On Linear Variational Surface Deformation Methods." *IEEE Transactions on Visualization and Computer Graphics* 14 (1): 213–30. doi:`10.1109/TVCG.2007.1054`.

Botsch, Mario, Stephan Steinberg, Stephan Bischoff, and Leif Kobbelt. 2002. "OpenMesh: A Generic and Efficient Polygon Mesh Data Structure." In *OpenSG Symposium 2002.*

Bouaziz, Sofien, Mario Deuss, Yuliy Schwartzburg, Thibaut Weise, and Mark Pauly. 2012. "Shape-Up: Shaping Discrete Geometry with Projections." *Computer Graphics Forum* 31 (5): 1657–1667. doi:`10.1111/j.1467-8659.2012.03171.x`.

Bouaziz, Sofien, Andrea Tagliasacchi, and Mark Pauly. 2014. "Dynamic 2D/3D Registration." In *EG 2014 - Tutorials.* Eurographics Association. doi:`10.2312/egt.20141021`.

Brewer, Michael, Lori Freitag, Patrick Knupp, Thomas Leurent, and Darryl Melander. 2003. "The Mesquite Mesh Quality Improvement Toolkit." In *Proceedings of the 12th International Meshing Roundtable,* 239–250.

Campagna, Swen, Leif Kobbelt, and Hans-Peter Seidel. 1998. "Directed Edges: A Scalable Representation for Triangle Meshes." *Journal of Graphics Tools* 3 (4): 1–11. doi:10.1080/10867651.1998.10487494.

Carr, Jonathan C., Richard K. Beatson, Jon B. Cherrie, Tim J. Mitchell, William R. Fright, Bruce C. McCallum, and Tim R. Evans. 2001. "Reconstruction and Representation of 3D Objects with Radial Basis Functions." In *Proceedings of SIGGRAPH 2001,* 67–76. New York: ACM. doi:10.1145/383259.383266.

Celniker, George, and Dave Gossard. 1991. "Deformable Curve and Surface Finite-elements for Free-form Shape Design." *Computer Graphics* 25 (4): 257–266. doi:10.1145/127719.122746.

Chen, Yanqing, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. 2008. "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate." *ACM Transactions on Mathematical Software* 35 (3): 1–14. doi:10.1145/1391989.1391995.

Coello, Carlos A. Coello. 1999. "A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques." *Knowledge and Information Systems* 1 (3): 269–308. doi:10.1007/BF03325101.

Coquillart, Sabine. 1990. "Extended Free-form Deformation: A Sculpturing Tool for 3D Geometric Modeling." *Computer Graphics* 24 (4): 187–196. doi:10.1145/97880.97900.

Cottrell, J. Austin, Thomas J.R. Hughes, and Yuri Bazilevs. 2009. *Isogeometric Analysis: Toward Integration of CAD and FEA.* John Wiley & Sons.

Dagum, Leonardo, and Ramesh Menon. 1998. "OpenMP: An Industry Standard API for Shared-Memory Programming." *Computational Science & Engineering* 5 (1): 46–55. doi:10.1109/99.660313.

Davis, Timothy A. 2011. "Algorithm 915, SuiteSparseQR: Multifrontal Multithreaded Rank-revealing Sparse QR Factorization." *ACM Transactions on Mathematical Software* 38 (1): 8:1–8:22. doi:10.1145/2049662.2049670.

de Berg, Mark, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry: Algorithms and Applications.* 3rd ed. Springer. doi:10.1007/978-3-662-04245-8.

De Floriani, Leila, and Annie Hui. 2005. "Data Structures for Simplicial Complexes: An Analysis and a Comparison." In *Proceedings of Eurographics Symposium on Geometry Processing,* 119–128.

De Floriani, Leila, Annie Hui, Daniele Panozzo, and David Canino. 2010. "A Dimension-independent Data Structure for Simplicial Complexes." In *Proceedings of the 19th International Meshing Roundtable,* 403–420. Springer. doi:10.1007/978-3-642-15414-0_24.

Delfour, Michel C., and Jean Paul Zolésio. 2011. *Shapes and Geometries: Metrics, Analysis, Differential Calculus, and Optimization.* 2nd ed. Society for Industrial / Applied Mathematics. doi:10.1137/1.9780898719826.

Deng, Bailin, Sofien Bouaziz, Mario Deuss, Juyong Zhang, Yuliy Schwartzburg, and Mark Pauly. 2013. "Exploring Local Modifications for Constrained Meshes." *Computer Graphics Forum* 32 (2): 11–20. doi:10.1111/cgf.12021.

Detroit Engineered Products. 2015. *DEP MeshWorks Morpher.* http://www.depusa.com.

Deuss, Mario, Anders Holden Deleuran, Sofien Bouaziz, Bailin Deng, Daniel Piker, and Mark Pauly. 2015. "Modelling Behaviour: Design Modelling Symposium 2015." Chap. ShapeOp—A Robust and Extensible Geometric Modelling Paradigm, 505–515. Springer. doi:10.1007/978-3-319-24208-8_42.

Edwards, H. Carter, Alan B. Williams, Gregory D. Sjaardema, David G. Baur, and William K. Cochran. 2010. *SIERRA Toolkit Computational Mesh Conceptual Model.* Technical report SAND2010-1192. Sandia National Laboratories.

Esturo, Janick Martinez, Christian Rössl, Stefan Fröhlich, Mario Botsch, and Holger Theisel. 2011. "Pose Correction by Space-Time Integration." In *Proceedings of Vision, Modeling, Visualization,* 33–40. doi:10.2312/PE/VMV/VMV11/033-040.

Fabri, Andreas, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. 1998. *On the Design of CGAL, the Computational Geometry Algorithms Library.* Technical report RR-3407. INRIA Sophia Antipolis.

Fasshauer, Greg E. 2007. *Meshfree Approximation Methods with MATLAB.* World Scientific Publishing.

FIA. 2016. *Formula One Technical Regulations.* Federation Internationale de l'Automobile.

Fischler, Martin A., and Robert C. Bolles. 1981. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography." *Communications of the ACM* 24 (6): 381–395. doi:10.1145/358669.358692.

Fleishman, Shachar, Daniel Cohen-Or, and Cláudio T. Silva. 2005. "Robust Moving Least-Squares Fitting with Sharp Features." *ACM Transaction on Graphics* 24 (3): 544–552. doi:10.1145/1073204.1073227.

Floater, Michael S., Géza Kós, and Martin Reimers. 2005. "Mean Value Coordinates in 3D." *Computer Aided Geometric Design* 22:623–631. doi:10.1016/j.cagd.2005.06.004.

Fonseca, Carlos M., and Peter J. Fleming. 1995. "An Overview of Evolutionary Algorithms in Multiobjective Optimization." *Evolutionary Computation* 3 (1): 1–16. doi:10.1162/evco.1995.3.1.1.

Fries, Thomas-Peter, and Hermann-Georg Matthies. 2004. *Classification and Overview of Meshfree Methods.* Technical report Informatikbericht 2003-3. Institute of Scientific Computing, Technical University Braunschweig.

Fu, Luoting, Levent Burak Kara, and Kenji Shimada. 2012. "Feature, Design Intention and Constraint Preservation for Direct Modeling of 3D Freeform Surfaces." *3D Research* 3 (2). doi:10.1007/3DRes.02(2012)3.

Funck, Wolfram von, Holger Theisel, and Hans-Peter Seidel. 2006. "Vector Field-based Shape Deformations." *ACM Transaction on Graphics* 25 (3): 1118–25. doi:10.1145/1141911.1142002.

Gabriel, Edgar, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, et al. 2004. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation." In *Proceedings, 11th European PVM/MPI Users' Group Meeting,* 97–104. Budapest, Hungary. doi:10.1007/978-3-540-30218-6_19.

Gain, James E., and Dominique Bechmann. 2008. "A Survey of Spatial Deformation from a User-Centered Perspective." *ACM Transaction on Graphics* 27 (4): 107:1–107:21. doi:10.1145/1409625.1409629.

Gain, James E., and Neil A. Dodgson. 2001. "Preventing Self-Intersection Under Free-form Deformation." *IEEE Transactions on Visualization and Computer Graphics* 7 (4): 289–298. doi:10.1109/2945.965344.

Gal, Ran, Olga Sorkine, Niloy J. Mitra, and Daniel Cohen-Or. 2009. "iWIRES: An Analyze-and-Edit Approach to Shape Manipulation." *ACM Transaction on Graphics* 28 (3): 33:1–33:10. doi:10.1145/1531326.1531339.

Garimella, Rao V. 2004. "MSTK - A Flexible Infrastructure Library for Developing Mesh Based Applications." In *Proceedings of the 13th International Meshing Roundtable,* 203–212.

Garland, Michael, and Paul S. Heckbert. 1997. "Surface Simplification Using Quadric Error Metrics." In *Proceedings of SIGGRAPH 1997,* 209–216. doi:10.1145/258734.258849.

Giannelli, Carlotta, Bert Jüttler, and Hendrik Speleers. 2012. "THB-splines: The Truncated Basis for Hierarchical Splines." *Computer Aided Geometric Design* 29 (7): 485–498. doi:10.1016/j.cagd.2012.03.025.

Golub, Gene H., and Charles F. Van Loan. 2013. *Matrix Computations.* 4th ed. Johns Hopkins University Press.

Graening, Lars, and Bernhard Sendhoff. 2014. "Shape Mining: A Holistic Data Mining Approach for Engineering Design." *Advanced Engineering Informatics* 28 (2): 166–185. doi:http://dx.doi.org/10.1016/j.aei.2014.03.002.

Griessmair, Josef, and Werner Purgathofer. 1989. "Deformation of Solids with Trivariate B-splines." In *Proceedings of Eurographics.*

Guennebaud, Gaël. 2010. *Eigen v3.* http://eigen.tuxfamily.org.

Guibas, Leonidas, and Jorge Stolfi. 1985. "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi." *ACM Transaction on Graphics* 4 (2): 74–123. doi:10.1145/282918.282923.

Gurung, Topraj, Daniel Laney, Peter Lindstrom, and Jarek Rossignac. 2011. "SQuad: Compact Representation for Triangle Meshes." *Computer Graphics Forum* 30 (2): 355–364. doi:10.1111/j.1467-8659.2011.01866.x.

Gurung, Topraj, Mark Luffel, Peter Lindstrom, and Jarek Rossignac. 2011. "LR: Compact Connectivity Representation for Triangle Meshes." *ACM Transaction on Graphics* 30 (3): 67:1–67:8. doi:10.1145/2010324.1964962.

Habbecke, Martin, and Leif Kobbelt. 2012. "Linear Analysis of Nonlinear Constraints for Interactive Geometric Modeling." *Computer Graphics Forum* 31 (2): 641–650. doi:10.1111/j.1467-8659.2012.03043.x.

Hansen, Nikolaus, and Andreas Ostermeier. 2001. "Completely Derandomized Self-Adaptation in Evolution Strategies." *Evolutionary Computation* 9 (2): 159–195.

Harmon, David, Daniele Panozzo, Olga Sorkine, and Denis Zorin. 2011. "Interference Aware Geometric Modeling." *ACM Transaction on Graphics* 30 (6): 137:1–137:10. doi:10.1145/2070781.2024171.

Heft, Angelina I., Thomas Indinger, and Nikolaus A. Adams. 2012. *Introduction of a New Realistic Generic Car Model for Aerodynamic Investigations.* SAE Technical Paper 2012-01-0168. SAE International. doi:10.4271/2012-01-0168.

Held, Harald. 2009. *Shape Optimization under Uncertainty from a Stochastic Programming Point of View.* Vieweg+Teubner. doi:10.1007/978-3-8348-9396-3.

Helenbrook, Brian T. 2003. "Mesh Deformation using the Biharmonic Operator." *International Journal for Numerical Methods in Engineering* 56 (7): 1007–1021. doi:10.1002/nme.595.

Hirani, Anil N. 2003. "Discrete Exterior Calculus." PhD diss., California Institute of Technology.

Hormann, Kai, and Natarajan Sukumar. 2008. "Maximum Entropy Coordinates for Arbitrary Polytopes." *Computer Graphics Forum* 27 (5): 1513–1520. doi:10.1111/j.1467-8659.2008.01292.x.

Hsu, William M., John F. Hughes, and Henry Kaufman. 1992. "Direct Manipulation of Free-form Deformations." *Computer Graphics* 26 (2): 177–184. doi:10.1145/142920.134036.

Igel, Christian, Verena Heidrich-Meisner, and Tobias Glasmachers. 2008. "Shark." *Journal of Machine Learning Research* 9:993–996.

Intel. 2013. *Intel Math Kernel Library version 11.0.* http://software.intel.com/en-us/intel-mkl.

Jacobson, Alec, Ilya Baran, Jovan Popović, and Olga Sorkine. 2011. "Bounded Biharmonic Weights for Real-Time Deformation." *ACM Transaction on Graphics* 30 (4): 78:1–78:8. doi:10.1145/2010324.1964973.

Jacobson, Alec, Zhigang Deng, Ladislav Kavan, and John Peter Lewis. 2014. "Skinning: Real-time Shape Deformation." In *ACM SIGGRAPH 2014 Courses.* doi:10.1145/2614028.2615427.

Jacobson, Alec, Daniele Panozzo, et al. 2016. *libigl: A simple C++ geometry processing library.* http://libigl.github.io/libigl.

Jakobsson, Stefan, and Olivier Amoignon. 2007. "Mesh Deformation Using Radial Basis Functions for Gradient-Based Aerodynamic Shape Optimization." *Computers & Fluids* 36 (6): 1119–1136. doi:10.1016/j.compfluid.2006.11.002.

Joshi, Pushkar, Mark Meyer, Tony DeRose, Brian Green, and Tom Sanocki. 2007. "Harmonic Coordinates for Character Articulation." *ACM Transaction on Graphics* 26 (3). doi:10.1145/1276377.1276466.

Ju, Tao, Scott Schaefer, and Joe Warren. 2005. "Mean Value Coordinates for Closed Triangular Meshes." *ACM Transaction on Graphics* 24 (3): 561–566. doi:10.1145/1073204.1073229.

Kazhdan, Michael, Matthew Bolitho, and Hughes Hoppe. 2006. "Poisson Surface Reconstruction." In *Proceedings of the Fourth Eurographics symposium on Geometry processing,* 61–70. doi:10.2312/SGP/SGP06/061-070.

Kettner, Lutz. 1998. "Designing a Data Structure for Polyhedral Surfaces." In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry,* 146–154. doi:10.1145/276884.276901.

Kettner, Lutz. 1999. "Using Generic Programming for Designing a Data Structure for Polyhedral Surfaces." *Computational Geometry – Theory and Applications* 13 (1): 65–90. doi:10.1016/S0925-7721(99)00007-3.

Kettner, Lutz. 2015a. "3D Polyhedral Surfaces." In *CGAL User and Reference Manual,* 4.7. CGAL Editorial Board.

Kettner, Lutz. 2015b. "Halfedge Data Structures." In *CGAL User and Reference Manual,* 4.7. CGAL Editorial Board.

Kitware. 2010. *The VTK User's Guide.* 11th ed. Kitware, Inc.

Knupp, Patrick. 2000. "Achieving Finite Element Mesh Quality via Optimization of the Jacobian Matrix Norm and Associated Quantities. Part I." *International Journal for Numerical Methods in Engineering* 48 (3): 401–420. doi:10.1002/(SICI)1097-0207(20000530)48:3<401::AID-NME880>3.0.CO;2-D.

Knupp, Patrick. 2008. "Updating Meshes on Deforming Domains: An Application of the Target-Matrix Paradigm." *Communications in Numerical Methods in Engineering* 24 (6): 467–476. doi:10.1002/cnm.1013.

Kobbelt, Leif. 2000. "$\sqrt{3}$-Subdivision." In *Proceedings of SIGGRAPH 2000,* 103–112. doi:10.1145/344779.344835.

Kobbelt, Leif, Swen Campagna, Jens Vorsatz, and Hans-Peter Seidel. 1998. "Interactive Multi-Resolution Modeling on Arbitrary Meshes." In *Proceedings of SIGGRAPH,* 105–114. doi:10.1145/280814.280831.

König, Oliver, and Marc Wintermantel. 2004. "CAD-based Evolutionary Design Optimization with CATIA V5." In *Proceedings of the 1st Weimar Optimization and Stochastic Days.*

Kovalsky, Shahar Z., Noam Aigerman, Ronen Basri, and Yaron Lipman. 2015. "Large-scale Bounded Distortion Mappings." *ACM Transactions on Graphics* 34 (6): 191:1–191:10. doi:10.1145/2816795.2818098.

Lamousin, Henry J., and Warren N. Waggenspack. 1994. "NURBS-based Free-form Deformations." *Computer Graphics and Applications* 14 (6): 59–65. doi:10.1109/38.329096.

Li, Yangyan, Xiaokun Wu, Yiorgos Chrysathou, Andrei Sharf, Daniel Cohen-Or, and Niloy J. Mitra. 2011. "GlobFit: Consistently Fitting Primitives by Discovering Global Relations." *ACM Transaction on Graphics* 30 (4): 52:1–52:12. doi:10.1145/2010324.1964947.

Lipman, Yaron, David Levin, and Daniel Cohen-Or. 2008. "Green Coordinates." *ACM Transaction on Graphics* 27 (3): 78:1–78:10. doi:10.1145/1360612.1360677.

Liu, Tiantian, Ming Gao, Lifeng Zhu, Eftychios Sifakis, and Ladislav Kavan. 2016. "Fast and Robust Inversion-Free Shape Manipulation." *Computer Graphics Forum* 35 (2).

Lloyd, Stuart P. 1982. "Least Square Quantization in PCM." *IEEE Transactions on Information Theory* 28 (2): 129–37. doi:10.1109/TIT.1982.1056489.

Loop, Charles Teorell. 1987. "Smooth Subdivision Surfaces Based on Triangles." Master's thesis, University of Utah, Department of Mathematics.

MacCracken, Ron, and Kenneth I. Joy. 1996. "Free-form Deformations with Lattices of Arbitrary Topology." In *Proceedings of SIGGRAPH 1996,* 181–188. doi:10.1145/237170.237247.

Mäntylä, Martti. 1987. *An Introduction to Solid Modeling.* Computer Science Press.

Manzoni, Andrea, Alfio Quarteroni, and Gianluigi Rozza. 2012. "Shape Optimization for Viscous Flows by Reduced Basis Methods and Free-form Deformation." *International Journal for Numerical Methods in Fluids* 70 (5): 646–670. doi:10.1002/fld.2712.

Martin, Sebastian, Peter Kaufmann, Mario Botsch, Eitan Grinspun, and Markus Gross. 2010. "Unified Simulation of Elastic Rods, Shells, and Solids." *ACM Transaction on Graphics* 29 (4): 39:1–39:10. doi:10.1145/1778765.1778776.

Masuda, Hiroshi, Yasuhiro Yoshioka, and Yoshiyuki Furukawa. 2007. "Preserving Form Features in Interactive Mesh Deformation." *Computer-Aided Design* 39 (5): 361–368. doi:10.1016/j.cad.2007.02.010.

McDonnell, Kevin T., and Hong Qin. 2007. "PB-FFD: A Point-based Technique for Free-form Deformation." *Journal of Graphics, GPU, and Game tools* 12 (3): 25–41. doi:10.1080/2151237X.2007.10129242.

Menzel, Stefan, Markus Olhofer, and Bernhard Sendhoff. 2005. "Application of Free Form Deformation Techniques in Evolutionary Design Optimisation." In *Proceedings of the 6th World Congress on Structural and Multidisciplinary Optimization.*

Menzel, Stefan, Markus Olhofer, and Bernhard Sendhoff. 2006. "Direct Manipulation of Free Form Deformation in Evolutionary Design Optimisation." In *International Conference on Parallel Problem Solving From Nature,* 352–361. doi:10.1007/11844297_36.

Menzel, Stefan, and Bernhard Sendhoff. 2008. "Representing the Change - Free Form Deformation for Evolutionary Design Optimization." In *Evolutionary Computation in Practice,* edited by Tina Yu, Lawrence Davis, Cem Baydar, and Rajkumar Roy, 88:63–86. Studies in Computational Intelligence. doi:10.1007/978-3-540-75771-9_4.

Meyer, Mark, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. 2003. "Discrete Differential-Geometry Operators for Triangulated 2-Manifolds." In *Visualization and Mathematics III,* edited by Hans-Christian Hege and Konrad Polthier, 35–57. Springer-Verlag. doi:10.1007/978-3-662-05105-4_2.

Michler, Andreas K. 2011. "Aircraft Control Surface Deflection using RBF-based Mesh Deformation." *International Journal for Numerical Methods in Engineering* 88 (10): 986–1007. doi:10.1002/nme.3208.

Mitra, Niloy J., Michael Wand, Hao Zhang, Daniel Cohen-Or, and Martin Bokeloh. 2013. "Structure-aware Shape Processing." In *EG 2013 STARs,* 175–197. The Eurographics Association. doi:10.1145/2614028.2615401.

Moccozet, Laurent, and Nadia Magnenat Thalmann. 1997. "Dirichlet Free-form Deformations and Their Application to Hand Simulation." In *Computer Animation '97,* 93–102. IEEE Computer Society. doi:10.1109/CA.1997.601047.

Mohammadi, Bijan, and Olivier Pironneau. 2010. *Applied Shape Optimization for Fluids.* 2nd ed. Oxford University Press.

Nealen, Andrew, Matthias Mueller, Richard Keiser, Eddy Boxerman, and Mark Carlson. 2006. "Physically Based Deformable Models in Computer Graphics." *Computer Graphics Forum* 25 (4): 809–836. doi:10.1111/j.1467-8659.2006.01000.x.

NHTSA. 1998. *Federal Motor Vehicle Safety Standards and Regulations.* National Highway Traffic Safety Administration.

Nieto, Jesús R., and Antonio Susín. 2013. "Cage Based Deformations: A Survey." In *Deformation Models,* edited by Manuel González Hidalgo, Arnau Mir Torres, and Javier Varona Gómez, 7:75–99. Lecture Notes in Computational Vision and Biomechanics. Springer. doi:10.1007/978-94-007-5446-1_3.

Nocedal, Jorge, and Stephen Wright. 2006. *Numerical Optimization.* Springer. doi:10.1007/978-0-387-40065-5.

Olhofer, Markus, Thomas Bihrer, Stefan Menzel, Michael Fischer, and Bernhard Sendhoff. 2009. "Evolutionary Optimisation of an Exhaust Flow Element with Free Form Deformation." In *Proceedings of the 4th European Automotive Simulation Conference.*

Open CASCADE. 2012. *Open CASCADE Technology, 3D Modeling & Numerical Simulation.* `http://www.opencascade.org/`.

OpenFOAM. 2012. *Open Source Field Operation and Manipulation C++ libraries.* `http://www.openfoam.org`.

Optimal Solutions. 2015. *Sculptor.* `http://www.optimalsolutions.us`.

Paillé, Gilles-Philippe, Nicolas Ray, Pierre Poulin, Alla Sheffer, and Bruno Lévy. 2015. "Dihedral Angle-based Maps of Tetrahedral Meshes." *ACM Transaction on Graphics* 34 (4): 54:1–54:10. doi:`10.1145/2766900`.

Pavić, Darko, and Leif Kobbelt. 2008. "High-Resolution Volumetric Computation of Offset Surfaces with Feature Preservation." *Computer Graphics Forum* 27 (2): 165–174. doi:`10.1111/j.1467-8659.2008.01113.x`.

Pion, Sylvain, and Mariette Yvinec. 2015. "2D Triangulation Data Structure." In *CGAL User and Reference Manual,* 4.7. CGAL Editorial Board.

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing.* 3rd ed. Cambridge University Press.

Rechenberg, Ingo. 1973. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution.* Frommann-Holzboog.

Rechenberg, Ingo. 1994. *Evolutionsstrategie '94.* Frommann-Holzboog.

Richter, Andreas, Jascha Achenbach, Stefan Menzel, and Mario Botsch. 2016. "Evolvability as a Quality Criterion for Linear Deformation Representations in Evolutionary Optimization." In *Proceedings of the IEEE Congress on Evolutionary Computation.* To appear.

Roth, Gerhard, and Martin D. Levine. 1993. "Extracting Geometric Primitives." *CVGIP: Image Understanding* 58 (1): 1–22. doi:`10.1006/ciun.1993.1028`.

Ruprecht, Detlef, Ralf Nagel, and Heinrich Müller. 1995. "Spatial Free-form Deformation with Scattered Data Interpolation Methods." *Computers & Graphics* 19 (1): 63–71. doi:`10.1016/0097-8493(94)00122-F`.

Sacht, Leonardo, Etienne Vouga, and Alec Jacobson. 2015. "Nested Cages." *ACM Transactions on Graphics* 34 (6): 170:1–170:14. doi:`10.1145/2816795.2818093`.

Samareh, Jamshid A. 2001. "Survey of Shape Parameterization Techniques for High-Fidelity Multidisciplinary Shape Optimization." *AIAA journal* 39 (5): 877–884. doi:`10.2514/2.1391`.

Samareh, Jamshid A. 2004. "Aerodynamic Shape Optimization Based on Free-form Deformation." In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference.*

Samet, Hanan. 1990. *The Design and Analysis of Spatial Data Structures.* Addison Wesley. doi:`10.1109/MCG.1990.10031`.

Sapidis, Nicholas S. 1994. *Designing Fair Curves and Surfaces: Shape Quality in Geometric Modeling and Computer-Aided Design.* Society for Industrial and Applied Mathematics.

Schaback, Robert, and Holger Wendland. 2000. "Adaptive Greedy Techniques for Approximate Solution of Large RBF Systems." *Numerical Algorithms* 24 (3): 239–254. doi:10.1023/A:1019105612985.

Schnabel, Ruwen, Roland Wahl, and Reinhard Klein. 2007. "Efficient RANSAC for Point-Cloud Shape Detection." *Computer Graphics Forum* 26 (2): 214–226. doi:10.1111/j.1467-8659.2007.01016.x.

Schneider, Philip, and David H. Eberly. 2002. *Geometric Tools for Computer Graphics.* Morgan Kaufmann.

Schüller, Christian, Ladislav Kavan, Daniele Panozzo, and Olga Sorkine-Hornung. 2013. "Locally Injective Mappings." *Computer Graphics Forum* 32 (5): 125–135. doi:10.1111/cgf.12179.

Sederberg, Thomas W., and Scott R. Parry. 1986. "Free-form Deformation of Solid Geometric Models." *Computer Graphics* 20 (4): 151–160. doi:10.1145/15886.15903.

Sederberg, Thomas W., Jianmin Zheng, Almaz Bakenov, and Ahmad Nasri. 2003. "T-splines and T-NURCCs." *ACM Transaction on Graphics* 22 (3): 477–484. doi:10.1145/882262.882295.

Seegyoung Seol, E., and M. S. Shephard. 2006. "Efficient Distributed Mesh Data Structure for Parallel Automated Adaptive Analysis." *Engineering with Computers* 22 (3): 197–213. doi:10.1007/s00366-006-0048-4.

Shewchuk, Jonathan R. 1996. "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator." In *Applied Computational Geometry: Towards Geometric Engineering,* 1148:203–222. Springer-Verlag. doi:10.1007/BFb0014497.

Shiue, Le-Jeng, Pierre Alliez, Radu Ursu, and Lutz Kettner. 2004. "A Tutorial on CGAL Polyhedron for Subdivision Algorithms." In *Symposium on Geometry Processing Course Notes.*

Shontz, Suzanne M., and Stephen A. Vavasis. 2003. "A Mesh Warping Algorithm Based on Weighted Laplacian Smoothing." In *Proceedings of the 12th International Meshing Roundtable,* 147–158.

Shontz, Suzanne M., and Stephen A. Vavasis. 2010. "Analysis of and Workarounds for Element Reversal for a Finite Element-Based Algorithm for Warping Triangular and Tetrahedral Meshes." *BIT Numerical Mathematics* 50 (4): 863–884. doi:10.1007/s10543-010-0283-3.

Sibson, Robin. 1980. "A Vector Identity for the Dirichlet Tessellation." *Mathematical Proceedings of the Cambridge Philosophical Society* 87 (1): 151–155. doi:10.1017/S0305004100056589.

Sibson, Robin. 1981. "A Brief Description of Natural Neighbor Interpolation." *Interpreting Multivariate Data* 21:21–36.

Sieger, Daniel, Pierre Alliez, and Mario Botsch. 2010. "Optimizing Voronoi Diagrams for Polygonal Finite Element Computations." In *Proceedings of the 19th International Meshing Roundtable,* edited by Suzanne M. Shontz, 335–350. Berlin: Springer-Verlag. doi:10.1007/978-3-642-15414-0_20.

Sieger, Daniel, Stefan Menzel, and Mario Botsch. 2012. "A Comprehensive Comparison of Shape Deformation Methods in Evolutionary Design Optimization." In *Proceedings of the 3rd International Conference on Engineering Optimization,* edited by José Herskovits.

Simon, Dan. 2013. *Evolutionary Optimization Algorithms.* John Wiley & Sons.

Song, Wenhao, and Xunnian Yang. 2005. "Free-form Deformation with Weighted T-spline." *The Visual Computer* 21 (3): 139–151. doi:10.1007/s00371-004-0277-8.

Sorkine, Olga, and Mario Botsch. 2009. "Interactive Shape Modeling and Deformation." In *Eurographics 2009 Tutorials.* doi:10.2312/egt.20091068.

Sorkine, Olga, and Daniel Cohn-Or. 2004. "Least-Squares Meshes." In *Proceedings of Shape Modeling International,* 191–199. doi:10.1109/SMI.2004.38.

Staten, Matthew L., Steven S. Canann, and Steven J. Owen. 1999. "BMSweep: Locating Interior Nodes During Sweeping." *Engineering with Computers* 15 (3): 212–218. doi:10.1007/s003660050016.

Staten, Matthew L., Steven J. Owen, Suzanne M. Shontz, Andrew G. Salinger, and Todd S. Coffey. 2011. "A Comparison of Mesh Morphing Methods for 3D Shape Optimization." In *Proceedings of the 20th International Meshing Roundtable,* edited by William Roshan Quadros, 293–311. Berlin: Springer-Verlag. doi:10.1007/978-3-642-24734-7_16.

Sukumar, Natarajan. 2004. "Construction of Polygonal Interpolants: A Maximum Entropy Approach." *International Journal for Numerical Methods in Engineering* 61 (12): 2159–2181. doi:10.1002/nme.1193.

Sukumar, Natarajan, and Elisabeth Anna Malsch. 2006. "Recent Advances in the Construction of Polygonal Finite Element Interpolants." *Archives of Computational Methods in Engineering* 13 (1): 129–163. doi:10.1007/BF02905933.

Tang, Chengcheng, Xiang Sun, Alexandra Gomes, Johannes Wallner, and Helmut Pottmann. 2014. "Form-finding with Polyhedral Meshes Made Simple." *ACM Transactions on Graphics* 33 (4). doi:10.1145/2601097.2601213.

Tautges, Timothy J., Ray Meyers, Karl Merkley, Clint Stimpson, and Corey Ernst. 2004. *MOAB: A Mesh-Oriented Database.* Technical report SAND2004-1592. Sandia National Laboratories.

Terzopoulos, Demetri, John Platt, Alan Barr, and Kurt Fleischer. 1987. "Elastically Deformable Models." *Computer Graphics* 21 (4): 205–214. doi:10.1145/37402.37427.

Tomov, Stanimire, Rajib Nath, Hatem Ltaief, and Jack Dongarra. 2010. "Dense Linear Algebra Solvers for Multicore with GPU Accelerators." In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum,* 1–8. doi:10.1109/IPDPSW.2010.5470941.

Tournois, Jane, Pierre Alliez, and Olivier Devillers. 2008. "Interleaving Delaunay Refinement and Optimization for 2D Triangle Mesh Generation." In *Proceedings of the 16th International Meshing Roundtable,* 83–101. doi:10.1007/978-3-540-75103-8_5.

Trefethen, Lloyd N., and David Bau. 1997. *Numerical Linear Algebra.* Society for Industrial and Applied Mathematics.

VCGLib. 2011. *The Visualization and Computer Graphics Library.* http://vcg.sourceforge.net/.

Wachspress, Eugene L. 1975. *A Rational Finite Element Basis.* Elsevier.

Welch, William, and Andrew Witkin. 1992. "Variational Surface Modeling." *Computer Graphics* 26 (2): 157–166. doi:10.1145/142920.134033.

Wendland, Holger. 2010. *Scattered Data Approximation.* Cambridge University Press.

Wicke, Martin, Daniel Ritchie, Bryan M. Klingner, Sebastian Burke, Jonathan R. Shewchuk, and James F. O'Brien. 2010. "Dynamic Local Remeshing for Elastoplastic Simulation." *ACM Transaction on Graphics* 29 (4): 49:1–49:11. doi:10.1145/1778765.1778786.

Xian, Chuhua, Hongwei Lin, and Shuming Gao. 2009. "Automatic Generation of Coarse Bounding Cages from Dense Meshes." In *Proceedings of Shape Modeling and Applications,* 21–27. doi:10.1109/SMI.2009.5170159.

Xian, Chuhua, Hongwei Lin, and Shuming Gao. 2012. "Automatic Cage Generation by Improved OBBs for Mesh Deformation." *The Visual Computer* 28 (1): 21–33. doi:10.1007/s00371-011-0595-6.

Yamazaki, Wataru, Sylvain Mouton, and Gerald Carrier. 2010. "Geometry Parameterization and Computational Mesh Deformation by Physics-based Direct Manipulation Approaches." *AIAA journal* 48 (8): 1817–1832. doi:10.2514/1.J050255.

Yang, Yong-Liang, Yi-Jun Yang, Helmut Pottmann, and Niloy J. Mitra. 2011. "Shape Space Exploration of Constrained Meshes." *ACM Transaction on Graphics* 30 (6): 124. doi:10.1145/2070781.2024158.

Zhang, Juyong, Bailin Deng, Zishun Liu, Giuseppe Patanè, Sofien Bouaziz, Kai Hormann, and Ligang Liu. 2014. "Local Barycentric Coordinates." *ACM Transaction on Graphics* 33 (6): 188:1–188:12. doi:10.1145/2661229.2661255.

Zheng, Youyi, Hongbo Fu, Daniel Cohen-Or, Oscar Kin-Chung Au, and Chiew-Lan Tai. 2011. "Component-wise Controllers for Structure-Preserving Shape Manipulation." *Computer Graphics Forum* 30 (2): 563–572. doi:10.1111/j.1467-8659.2011.01880.x.

COLOPHON

This thesis was typeset using LaTeX. The body text is typeset at 11pt using the *Minion Pro* font designed by Robert Slimbach. All math was typeset using Johannes Küster's *Minion Math* font. The `typewriter` text uses the *Consolas* font designed by Lucas de Groot.