Visualization Research Center (VISUS)

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diploma Thesis Nr. 3442

Comparative Visualization of Electrostatic Fields

Katrin Scharnowski

Course of Study:	Computer Science
Examiner:	Prof. Dr. Thomas Ertl
Supervisor:	Dipl.-Inf. Michael Krone Dr. Guido Reina
Commenced:	28 January 2013
Completed:	30 July 2013
CR-Classification:	I.3.5, I.3.7, I.3.8, I.4.8

Abstract

When dealing with large amounts of complex data, such as the results of Molecular Dynamics simulations, comparative visualization techniques are a useful tool to demonstrate connections, differences, or similarities of different data sets. In order to facilitate comparative visualization of the molecular electrostatic surface potential, a shape correspondence framework for molecular surfaces is derived. Given two particle-based input data sets, an implicit molecular surface representation is defined by a Gaussian density volume. A triangulation is extracted from the volume using the Marching Tetrahedra method. A mapping relation between the two molecular surfaces is then established using a deformable model approach in combination with rigid alignment. To this end, the source surface is represented by an elastic shape that is locally deformed to match the target surface. The deformation of the model is driven by an adaptive external force and by an internal force that is tangential to the Gaussian volume. Based on the mapping relation, both a comparative visualization and a difference metric are developed. The surface generation and the surface mapping approach are implemented in a highly parallel manner using CUDA. The method is finally applied to several real-world data sets obtained by Molecular Dynamics simulations.

Kurzfassung

Im Kontext großer Mengen komplexer Daten, z.B. aus Molekulardynamik-Simulationen, können vergleichende Visualisierungstechniken wichtige Einblicke in die Zusammenhänge, Unterschiede und Gemeinsamkeiten verschiedener Datensätze geben. Zum Zweck einer vergleichenden Visualisierung des elektrostatischen Oberflächenpotentials von Molekülen wurde ein Framework zur Verknüpfung von Moleküloberflächen entwickelt. Hierbei wird zunächst aus den Partikeldatensätzen ein Dichtefeld zur impliziten Oberflächenrepresentation erstellt. Anschließend wird mit Hilfe des Marching Tetrahedra Algorithmus eine Triangulierung der Startoberfläche berechnet. Mit Hilfe einer Starrkörper-Abbildung und eines 'Deformable Model'-Ansatzes wird eine Relation zwischen den beiden Flächen hergestellt. Die Startfläche wird dafür durch ein elastisches Modell representiert und wird lokal deformiert um sich der Zielfläche anzupassen. Das Modell wird mit Hilfe einer adaptiven externen Kraft und einer projezierten internen Kraft verformt. Basierend auf der dadurch hergestellten Relation wird neben einer vergleichenden Visualisierung auch eine Differenz-Metrik hergeleitet, die zur Quantifizierung der Potentialunterschiede beider Flächen verwendet wird. Sowohl die Flächentriangulierung als auch die Deformation werden mit CUDA implementiert. Der entwickelte Ansatz wird schließlich auf Partikeldatensätze angewendet, die durch Molekulardynamik-Simulationen erstellt wurden.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The availability of increasingly powerful computational resources leads to fast-growing amounts of complex data in different fields of science and engineering. Scientific data often is of *multi-faceted* nature. The term multi-faceted refers to the fact that the data consists of the output of two or more different sources. According to Kehrer *et al.*, multi-faceted data can be categorized further into spatiotemporal data, multi-variate data, multimodal data, multirun data, and multimodel data [KH13]. Spatiotemporal data represent dynamic processes captured in time-varying measurements and simulations. Multi-variate data consist of different attributes, representing e.g. different physical phenomena, such as temperature or pressure. Multimodal data stems from different acquisition methods. One example would be a medical scan of a human body that is obtained both by computed tomography (CT) and magnetic resonance imaging (MRI). Multirun data refers to the data obtained by several runs of a simulation, each of which is computed with varied input parameters. Multimodel data are the result of of simulations that combine different physical models for interacting phenomena. Another problem where multi-faceted data plays a role is the validation of numerical models, based on inaccuracies or artifacts, by comparing their results with ones obtained in experiments [PP95]. In biochemistry, it is common practice to compare different molecules to gain insight in their functionality. The underlying assumption is that molecules that are similar with respect to a certain criterion have similar functionality. Here, the three-dimensional structures of the molecules play a big role and are often used to establish correspondence [KN03]. *Comparative visualization* techniques integrate several aspects of multi-faceted data in one representation while avoiding visual clutter and occlusion. They can, therefore, help reducing information density and facilitate the understanding and analysis of complex heterogeneous scientific data.

As stated before, multi-faceted data often stems from simulations. *Molecular Dynamics (MD) simulation* is a technique that uses computational methods to retrieve information about molecules on a macroscopic level [Sch10, FS01, SLH$^+$10]. In MD simulations, molecules are represented by many-body systems whose development over time is computed by numerically solving Newton's equations of motion. This allows investigating dynamic properties of many-body systems that can often not be computed analytically. Some of those properties include molecular geometries and energies, mean atomic fluctuations, local fluctuations (like formation/breakage of hydrogen bonds, water/solute/ion interaction patterns, or nucleic-acid backbone torsion motions), but most importantly large-scale deformations such as protein folding [Sch10]. The increasing availability of high-performing computational resources allows for more complex simulations consisting of large numbers of particles. This is especially true, since the use of *General Purpose Computation on Graphics Processing Unit (GPGPU)* in MD simulations and molecular modeling has become more common (see e.g. [SPF+07, LSVMW07]). This also leads to larger amounts of information that need to be analyzed. Dealing with the growing complexity and size of the input data can be very challenging. It is, therefore, crucial to find appropriate visual representations to facilitate exploratory analysis of the data.

In biology, understanding electrostatic properties of proteins plays an important role for the investigation of various processes. This includes inter-molecular interactions [SNH00, NPP03, KN10], protein folding, and structure-based drug design [HN95]. The electrostatic potential of a many-body system in a MD simulation is often approximated using the potential defined in *Coulomb's law*, which was formulated by the french physicist Charles Augustin de Coulomb (1736-1806). The Coulomb potential describes interactions between non-bonded atom pairs and decays only slowly with distance [Sch10]. Direct computation of the Coulomb potential, also known as *direct Coulomb summation*, is computationally heavy, since it involves the interactions of all atom pairs, which yields a complexity of $\mathcal{O}(n^2)$ (where n is the number of particles in the simulation). Alternative computational methods include spherical cutoff-schemes, multipole schemes, or Ewald summation. A further alternative are simplified solvent models, which only represent the solvent implicitly as a continuum function. There are various tools that compute the electrostatic potential of MD simulation results using one or more of the computational methods mentioned above. VMD (Visual Molecular Dynamics) [1] is a tool for molecular modeling and visualization. Here, the electrostatic potential can be computed using continuum models, direct Coulomb summation, or Ewald summation. GROMACS [2] is a free software package for MD simulations that can be used to compute electrostatic properties by spherical cutoffs, Ewald summation, or particle mesh Ewald (PME), a variation of Ewald summation [SLH+10]. Amber [3] stands for both a set of molecular mechanical force fields and a package for MD simulations. In terms of electrostatics computations, this package offers particle mesh Ewald computation, implicit solvent models, and cutoff-schemes.

The goal of this thesis was to develop a visualization that can be used to compare electrostatic properties of different MD simulation results. This includes e.g. the development of the electrostatic potential over time, the impact of punctual mutations on the electrostatic properties of proteins, or the comparison of different MD simulation configurations (e.g. with different solvation models, or different mixtures of solvations). Additionally, a metric should be developed that quantifies the differences and that facilitates comparative analysis. Furthermore, the metric should be used to develop an abstract summary of the variance in the input data. The implementation of the visualization should be done in C/C++ using OpenGL/GLSL, while emphasizing the use of GPGPU methods (with CUDA) and multi-core CPUs (with OpenMP). The visualization should be embedded in the visualization framework MegaMol, which is developed at the collaborative research center SFB 716 [4] at the University of Stuttgart.

[1]http://www.ks.uiuc.edu/Research/vmd/
[2]http://www.gromacs.org/
[3]http://ambermd.org/
[4]http://www.sfb716.uni-stuttgart.de

During this thesis, an approach to the comparative visualization of electrostatic properties of molecular surfaces has been developed. The underlying principle is a shape correspondence frame work for molecular surfaces that establishes a bijective mapping relation between two input shapes. Here, basic steps are the definition of an implicit molecular surface and a subsequent shape matching based on a combination of rigid and non-rigid alignment. The rigid alignment is done using the well-known RMSD minimization method. The non-rigid alignment is based on a deformable model approach. Furthermore, a difference metric has been derived that quantifies the electrostatic surface potential difference by computing an absolute mean error value. Surface properties are visualized by 3D renderings of the molecular surfaces combined with different texturing approaches. The computational result of the difference metric is visualized in both a 1D- and a 2D-plot, which allows analyzing more than one data set at a time. The visualizations developed in this work were finally applied to different particle-based data sets stemming from MD simulations.

The remainder of this document is structured as follows: In Chapter 2, the work done for this thesis is put into several contexts of related work. These contexts are comparative visualization, shape correspondence, and deformable models. In Chapter 3, the theoretical foundations for the mapping algorithm and its derivation are provided, as well as the mathematical formulation of the difference metric. Chapter 4 contains a detailed description of the GPU implementation of the surface generation, the mapping algorithm, and the computation of the difference metric. Furthermore, some aspects of the rendering are outlined. In Chapter 5, the visualizations developed during this thesis are applied to several real-world data sets and the visualization approach is discussed. Chapter 6 summarizes the thesis.

2

Related Work

The matter of this thesis can be put into context with respect to three fields of related work: *comparative visualization*, *shape correspondence*, and *deformable models*. The field of *comparative visualization* is a broad area of research that deals with various kinds of input data. The approach derived in this thesis can clearly be seen in the context of comparative visualization, since two or more data sets have to be compared with each other. A central requirement for the comparative visualization is to find a mapping relation between different shapes in order to compare their properties. This problem is extensively researched and known as *shape correspondence*. The method used in this work to establish the shape correspondence is based on a *deformable model* approach.

2.1 Comparative Visualization

The amount and complexity of scientific data is increasing, which makes comparative visualization techniques a suitable means of finding compact representations of heterogeneous input data. The visualization of multi-faceted data has been the topic of extensive research [PP95, VP04, FH09, KH13].

In [PP95], approaches to comparative visualization are classified according to two categories: *image level comparison* and *data level comparison*. Image level comparison refers to techniques in which images are generated for each data set in a separate visualization pipeline. This is done in a way that facilitates comparative analysis of the resulting images (e.g. by using the same view angle for both pictures). These images are either be shown side-by-side or a post-processing step is used to transform them into one single image. More sophisticated approaches of image level comparison use e.g. difference images [WS06] or superimposed images [LPPW95]. In [BBF+11], an image-based approach is used to show surface intersections. In contrast to image level approaches, data level comparison is done using only one visualization pipeline. A common representation of both data sets is generated on a data level and then visualized. Techniques that use data level comparison can be especially useful when more then two data sets have to be compared, since they allow reducing the information density significantly. In this thesis, a data level comparison for of two 3D surfaces is constructed by deriving a bijective mapping relation. In [PF96], data-level methods to compare attributes on 3D surfaces are discussed. In [KWP01], data level comparison is used to estimate errors during direct volume rendering caused by different gradient estimation methods. These methods, however, assume the surfaces to be identical, which is not the case in this work.

2.2 Shape Correspondence

The problem of establishing a meaningful relation between two or more given shapes is well known and there has been a rich history of extensive studies in various fields of research. The correspondence problem plays an important role in image analysis [AFP00, SCP12], computer graphics [LV98, Ale02, KZHCO11], and shape matching [VH01, IJL+05, BKS+05, TV08]. Finding and quantifying differences between two shapes is also a form of shape correspondence [CRS98, NSCE02].

In [KZHCO11], a classification of different correspondence methods is provided. Correspondence methods are classified according to the input data, the desired output, the kind of correspondence that is to be established, and the actual approach to obtain the correspondence. In this case, the input data contains of surfaces that are implicitly defined by a level-set in volume textures that are based on the data sets' particles. Additionally, the molecular structure can be used to align both surfaces in a way that is based on meaningful semantic information. The actual mapping, however, is established using a discrete approximation of the source surface. In addition to the rigid alignment, the source surface needs to be deformed locally to establish the mapping relation, since the molecular surfaces can be shaped rather differently, despite having a similar underlying structure. Therefore, the kind of correspondence that needs to be established after the initial (rigid) alignment is of non-rigid nature. Combinations of rigid and non-rigid alignment have been used before [HPM06, LSP08].

The subject of rigid alignment (often called registration) is the problem of finding a rotation and a translation to align one object with another. One group of algorithms that us often used in that context are *iterative closest point* (ICP) algorithms, which were first introduced by [BM92]. In this family of algorithms, a distance metric based on the surface vertices is defined to quantify the difference between the two surfaces. The rotation/translation is iteratively changed while minimizing the outcome of the distance metric. This approach does not need prior semantic information. However, in this case, the surface is implicitly defined by the underlying molecular structure, which serves as a means for semantic correspondence. Molecular structures are usually aligned using the *root-mean-square-deviation* (RMSD) between a subset of the particles of the structures. An algorithm to numerically solve this problem and obtain the according transformation is described in [Kab76].

Non-rigid shape correspondence often plays a big role in mesh morphing algorithms [Ale02]. Here, one of the main goals is to find a smooth transition between two given shapes. In this work, however, the shape correspondence serves the purpose of transferring texture values between two input surfaces. The subject of texture mapping is, in fact, closely related to the subject of shape correspondence. In [ACP03], it is demonstrated how shape correspondence, established by deforming a template model, can be used to transfer texture values. The work presented in [ZGVF98] deals with texture mapping of implicit surfaces. Their work resembles this work in that they distribute discrete particles on an implicit surface and let these particles travel to the target space on the basis of an external force that is derived from the image gradient of the volume. In their case, however, no internal forces are present and the target is a 2D texture loosely wrapped around the source shape. In [TW99], a definition for implicit skeletal surfaces is given. This definition is related to the approach used in this thesis, because molecular surfaces are also defined implicitly based on the underlying molecular structure. They, however, define surface attributes by the attributes of the defining skeletal primitives, whereas in this case, the surface attributes are defined by a solid 3D texture.

Shape correspondence between molecular geometry is often used to classify molecules according to their functionality. In [RLWN98], molecular shapes are compared in terms of patches that represent active sites (e.g. binding sites). Here, a similar location of these patches suggests a similar functionality of the proteins. In [KN03], a similarity search is used, where similarity is quantified according to a graph-based approach that takes the surface curvature and the electrostatic surface potential into account. Similar to the approach followed in this thesis, the method in [PS09] uses a deformable model approach. They establish shape correspondence between a sphere and a molecular surface and render surface features on the sphere.

2.3 Deformable Models

Deformable models are a curves or surfaces that seek to minimize a given energy functional, consisting of an internal energy that describes the properties of the physical model and an external energy that attracts them to a target shape. Deformable models are very common in medical image analysis [MT96], but also mesh reconstruction [MDA01].

The principle of deformable models was first described by Terzopoulos *et al.* [TPBF87]. Early work by Kass *et al.* in 1988 [KWT88] introduced deformable 2D contours called 'snakes' that minimize a given energy function in order to segment images. Terzopoulos *et al.* extended this approach the 3D case [TWK88] and introduced it to the computer graphics community [TF88]. One problem of the original approach by Kass *et al.* is that contours that are not close enough to the target shape are not attracted to them. A solution to this problem is proposed in [Coh91], where they use an additional external pressure force to push the contour towards the target shape.

Deformable models can broadly be grouped into explicit (parametric) models and implicit (geometric) models (see [MDA01] for an overview). Explicit deformable models provide an explicit representation of the deformable shape in each time step of the deformation process. In general, explicit models allow for fast computation times and the explicit representation facilitates the tracking of surface points over time. They, however, can not handle topology changes (such as splitting, merging, or the appearance of cavities) during the deformation process. Implicit deformable models are based on the level-set method first introduced by Osher and Sethian [OS88]. They represent the deformable shape implicitly as the zero level-set of a higher dimensional function. The deformation of the shape is then encoded in the deformation of the higher dimensional function. Since the deformation of the model does not depend on an explicit parameterization, these methods can handle topology changes. The approach of this thesis uses an explicit surface representation, since having an explicit representation of the deformable model is fundamental to be able to track surface points.

The deformable model chosen for this work uses a modified internal force that is projected to a plane tangential to the volume. In [KTZ92], it is proven that only the normal component of an internal force applied to a contour changes its actual shape. A more regular mesh can, therefore, be obtained without changing the shape by moving the vertices according to the tangential part of the internal force. There are other approaches that – similar to the internal force used in this work – are based on the decomposition of the classical internal force. In [DM00], a decomposition of the internal force into a tangential and a normal part is used to obtain an evenly spaced mesh. In [SHL^{+}11], an internal force described by a discrete Laplacian approximation is decomposed in a similar manner. Here, they associate different weights with both parts to control the smoothness of the model. The tension term of the internal force used in this work is identical to their decomposed internal force when associating a weight of zero to the part perpendicular to the surface.

The deformable model proposed in this thesis uses an adaptive external force term that is a combination of a distance field and a volume that is implicitly defined by the molecular structure. Combinations of several external force terms to ensure global attraction has been proposed elsewhere [CC93b, GR03].

3

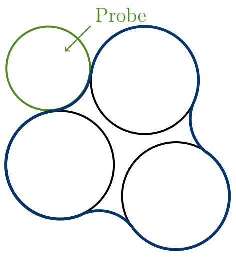
A Shape Correspondence Framework for Molecular Surfaces

In this chapter, the mapping relation needed for the comparative visualization is derived. Based on the given problem statement, a framework for establishing shape correspondence, consisting of several subsequent steps, is developed. The framework uses a deformable model approach to map a triangulated surface to a target shape, implicitly defined by a level set in a 3D volume texture. The initial triangulation is found using the Marching Tetrahedra method and regularized before the deformation. Finally, different metrics to quantify mapping errors and differences between the two input data sets are considered.

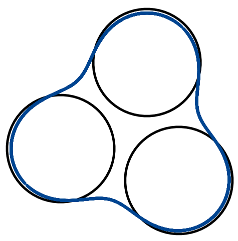
3.1 Input Data and Problem Statement

The input data consists of two or more volume textures in which a level-set is defined as the molecular surface. Additionally, the particles by which the volume is implicitly defined serve as a skeleton that can be used to obtain semantic correspondence. The overall goal is to find a bijective mapping function between two of the input shapes that facilitates the comparison of molecular surface properties. The mapping should have a deterministic outcome, an error quantification, and a fast GPU implementation should be possible. Furthermore, the mapping should establish an intuitively understandable semantic correspondence between two input data sets. Finally, a way has to be found to quantify the differences between the surface properties in a single metric to allow for comparing more then two input data sets at a time.

The bijective mapping is defined between two discrete surfaces. Thus, the source shapes are triangulated using the Marching Tetrahedra method. Here, different subdivision schemes have to be considered. The results of different runs of MD simulations are often differently bended or deformed despite having the same underlying structure. Thus, rigid alignment alone would not achieve a mapping of satisfactory accuracy and non-rigid alignment has to be applied in addition. To this end, the source shape is represented by an elastic deformable model that is iteratively deformed to fit the target shape. The mapping is then defined by the correspondence between the initial positions on the source shape and the mapped positions on the target shape. The following sections describe the computation of the initial vertex positions based on an implicitly defined molecular surface. Furthermore, the derivation of the deformable model approach and the formulation of a difference metric are explained.



(a) SES



(b) Gaussian approximation

Figure 3.1: Comparison of the solvent excluded surface and the Gaussian density approximation. The SES is shown in (a), the approximation in (b). The SES can be traced out by a probe rolling over all pairs of atoms. The particles have a radius of 0.45*, the probe has a radius of* 0.35*, and the factor* α *is* 0.35*. In comparison to the original SES, the approximation exhibits a slight over-smoothing effect, especially when the SES is computed using a rather small probe.*

3.2 The Implicit Molecular Surface

One possibility to model molecular surfaces is the implicit surface representation method proposed in [Bli82]. Each particle is associated with a Gaussian density distribution $\rho = e^{\frac{1}{\alpha r}}$, with r being the distance of the lattice position to the particle. A volume is computed by accumulating the density contributions of all particles at each lattice point. This method yields a smooth continuous density map that, when using an appropriate level set, can be seen as an approximation of the *solvent excluded surface* (SES) according to [Ric77]. The Gaussian distribution used in this work is slightly modified as described in [MGE07]. Every particle p_i is represented by a Gaussian density distribution ρ_i, which is defined by

$$\rho_i(\vec{x}) = e^{\frac{-|\vec{x}-\vec{x}_i|^2}{2\alpha^2}}, \tag{3.1}$$

where $\vec{x}_i$ is the position of the ith particle and α is a weighting factor. The weighting factor is the product of the radius associated with the particle and a user-defined scaling factor that affects the level of detail of the final volume. In this work, the *van der Waals radius* was used as the atom radius. Both the SES and the approximated surface are illustrated in Figure 3.1.

3.3 Rigid Alignment by Minimum Root-Mean-Square Deviation (RMSD)

Since a molecular surface can be defined implicitly by the underlying molecular structure, two molecular surfaces can be aligned by aligning the molecular structures. These structures are normally given as a set of particles, representing the individual atoms. In most cases (the same protein, similar proteins, mutant variants) it is possible to define a direct particle-to-particle relation for at least a subset of the particles. This relation can be used as a constraint for the

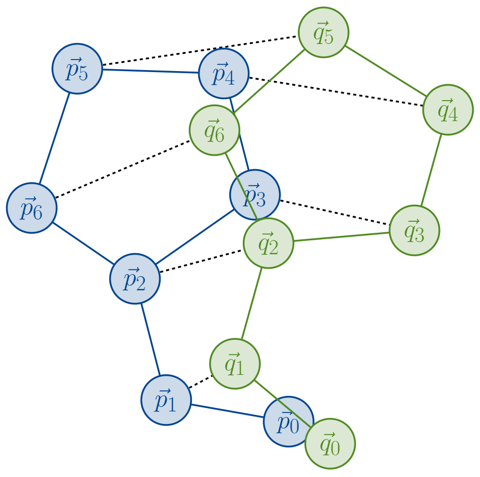


Figure 3.2: Illustration of the RMSD value of two molecules.

rigid alignment and essentially allows defining a semantic correspondence between the two shapes.

In bioinformatics, it is common practice to use the *root-mean-square deviation* (RMSD) of the backbone atoms (or only the C_α atoms) of two superimposed proteins as a dissimilarity measurement [RR73, MC94]. Given two sets of n points $P = \{\vec{p}_1, \ldots, \vec{p}_n\}$ and $Q = \{\vec{q}_1, \ldots, \vec{q}_n\}$, the RMSD is defined as

$$\mathrm{RMSD}(P, Q) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \|\vec{p}_i - \vec{q}_i\|^2}. \tag{3.2}$$

The RMSD is illustrated in Figure 3.2.

The goal is to find an affine transformation consisting of a rotation matrix R and a translation vector $\vec{t}$ that transforms the source molecule to the target molecule while minimizing the RMSD. The translation vector $\vec{t}$ can be found trivially by calculating the centroids of both particle sets. If the source molecule is represented by P and the target molecule is represented by Q and the centroids of both particle sets are $\vec{c}_p$ and $\vec{c}_q$, then the translation vector is $\vec{t} = \vec{c}_q - \vec{c}_p$. Finding the rotation matrix R is a more complex task. There are methods that use quaternions [CSD04] to find the optimal rotation matrix. However, the method most often used to compute R is the algorithm of Kabsch [Kab76, Kab78]. In order to apply the method of Kabsch, the source data set has to be translated to the target data set via the translation vector $\vec{t}$. The two particle sets are then represented as $n \times 3$ matrices that contain the three-dimensional coordinates of all particles.

$$P = \begin{pmatrix} p_{x1} & p_{y1} & p_{z1} \\ p_{x2} & p_{y2} & p_{z2} \\ p_{x3} & p_{y3} & p_{z3} \\ \vdots & \vdots & \vdots \\ p_{xn} & p_{yn} & p_{zn} \end{pmatrix} \quad \text{and} \quad Q = \begin{pmatrix} q_{x1} & q_{y1} & q_{z1} \\ q_{x2} & q_{y2} & q_{z2} \\ q_{x3} & q_{y3} & q_{z3} \\ \vdots & \vdots & \vdots \\ q_{xn} & q_{yn} & q_{zn} \end{pmatrix} \tag{3.3}$$

Subsequently, the covariance matrix A of P and Q is computed, which is defined by

$$A = P^T Q \tag{3.4}$$

The covariance matrix can be seen as a measurement for variance in multidimensional data. Next, the *singular value decomposition* of A is obtained by

$$A = VSW^T. \tag{3.5}$$

Here, V and W^T are 3×3 unitary matrices and S is a rectangular diagonal matrix containing the *singular values* of A. The next step is to decide whether the rotation matrix needs to be changed to yield a right-handed coordinate system.

$$d = \operatorname{sign}(\det WV^T) \tag{3.6}$$

Finally, the optimal rotation matrix can be computed as

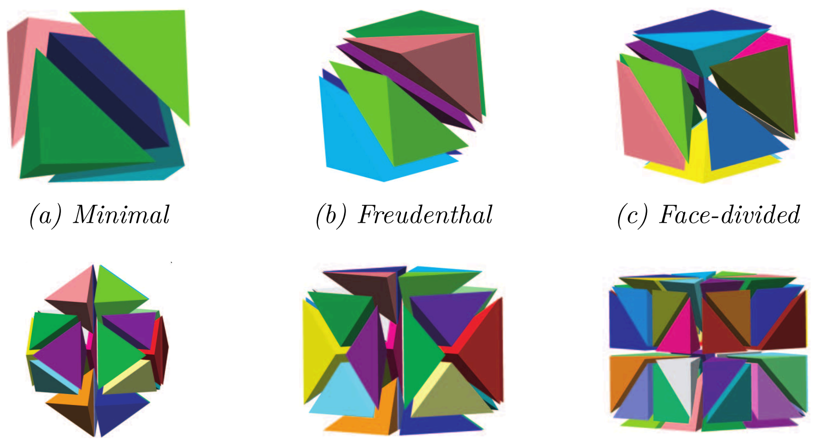
$$R = W \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & d \end{pmatrix} V^T \tag{3.7}$$

In this work, the algorithm of Kabsch was used to compute the initial rigid alignment. If the RMSD of the aligned structures is still very high after the rigid alignment, the mapping algorithm will not provide meaningful results. A sufficiently low RMSD value is, therefore, a requirement for the subsequent steps of the mapping algorithm.

3.4 Initial Triangulation

The initial triangulation of the implicitly defined source surface is now extracted from the volume texture generated by the approach described in Section 3.2. There are several requirements that the algorithm used for the triangulation should meet. The algorithm should be unambiguous, it should be suitable for a parallel GPU implementation, and, finally, there should be a fast way of extracting connectivity information between all vertices.

A widely used method for extracting a triangulation of an implicitely defined surface is the *Marching Cubes* algorithm proposed in [LC87]. The scalar field is discretized into voxels, which yields a set of cells. Each cell corner is then flagged depending on whether is lies outside (the sampled value is higher than the iso-value) or inside (the sampled value is below the iso-value) the implicit surface. These binary cell corner states can be combined into one state for every cell that describes if and how the surface crosses that cell. Since the number of possible states is finite, the according triangulation associated with every state can easily be stored in a lookup table. This easy LUT-based implementation is the main advantage of the Marching Cubes algorithm. The method, however, suffers from some substantial problems [HJ05]. The main problem is that some of the cases are ambiguous, which practically leads to disconnections and holes in the extracted triangulation. It is possible to resolve these ambiguities, e.g. by the use of *asymptotic deciders* [NH91]. Doing so, however, requires additional computational effort.


(a) Minimal *(b) Freudenthal* *(c) Face-divided*

(d) BBC *(e) Face-centered* *(f) Edge-centered*

Figure 3.3: Illustration of different subdivision schemes for the Marching Tetrahedra algorithm (from [CMS06]).

Another way of avoiding the ambiguities is to use a *Marching Tetrahedra* algorithm instead [HJ05]. Here, the cubic cells are further subdivided into tetrahedrons and the scalar field is then sampled at the vertices of each tetrahedron. The tetrahedra exhibit no ambiguous cases, which resolves the problem present in the Marching Cubes algorithm. The resulting triangulation, however, is much more irregular compared to the one generated by the Marching Cubes algorithm and contains a lot more triangles. Despite the irregular result, in this work, a Marching Tetrahedra algorithm is used in combination with a preprocessing step that regularizes the surface.

There are numerous ways of subdividing the cells into tetrahedrons. An overview is provided in [CMS06] (see also Figure 3.3). Each of these subdivision schemes has advantages and disadvantages to them. The actual choice depends on the goal that is to be achieved. One criterion for the choice in this work is that the scheme should be contained in a cube and not cross the edges of the cells. This is necessary to implement the parallel algorithm used in this work, since it fascilitates an easy computation of the connectivity information. From the subdivision schemes mentioned in [CMS06], only four fullfill this criterion: the *minimal* subdivision scheme, the *Freudenthal* subdivision scheme, and the *face-divided* subdivision schemes. Another important issue is the introduction of new vertices. The volume used in this work does not behave in a linear manner, but rather in an exponential way. However, every vertex introduced additionally would have to interpolate the necessary scalar value by using some interpolation scheme. Even when using cubic interpolation this would lead to artifacts. These artifacts are unnecessary, since there are subdivision schemes that do not introduce new vertices. This leaves the minimal and the Freudenthal subdivision scheme. In this work, the Freudenthal subdivision scheme was used since the GPU implementation is roughly based on an implementation that was already present in the MegaMol framework. However, both the minimal subdivision and the Freudenthal subdivision are valid choices.

One thing that has to be kept in mind when implementing the Marching Tetrahedra algorithm is that the goal is to process the extracted surface as a spring-mass model. The result should, therefore, be an array of vertices with the respective connectivity information. The actual triangles are not necessary for the mapping but are needed for the rendering of the surface. The same holds for the surface normals, which are needed for shading.

3.5 Derivation of the Deformable Model Approach

This section provides the mathematical foundations for the deformable model mapping approach used in this work. First, the underlying physical model for the continuous surface representation is formulated in terms of an *energy functional*. Then, the problem of minimizing this functional is reformulated using the notion of computing the equilibrium between two forces, the *internal forces* and *external forces*. This problem can then be transformed to the discrete case by using a discrete approximation of the necessary derivatives. This yields a representation of the surface that is comparable to a spring-mass-model. Finally, based on the discrete problem formulation, the actual iterative numerical solution of the deformation process is explained.

3.5.1 Mathematical Formulation

A 3D continuous elastic surface $\mathcal{S}$ is defined by a mapping v with

$$\begin{aligned} &v : \Omega \subset \mathbb{R}^2 \to \mathbb{R}^3 \\ &\qquad (s,r) \to v(s,r) = (v_1(s,r), v_2(s,r), v_3(s,r)) \end{aligned} \tag{3.8}$$

and an energy functional, decoding the properties the material's deformation, given by

$$\mathcal{E}(v) = \mathcal{E}_{int}(v) + \mathcal{E}_{ext}(v) \tag{3.9}$$

The external energy $\mathcal{E}_{ext}(v)$ is traditionally an image based potential function that causes the deformation of the input shape to the desired target shape. E.g. in [CC90], the external energy function is defined as $\mathcal{E}_{ext}(v) = -|\nabla I(v)|^2$. By using an energy function based on the negative gradient of the image intensity $I(v)$, the source shape is pulled towards sharp edges in the image. It is often derived from the image intensity by convolving it with a Gaussian and scaling it appropriately so that the deformation converges. The exact definition of the external energy term, however, depends on the overall purpose of the computation. The external energy function used in this thesis is explained in Section 3.5.3.

The internal energy $\mathcal{E}_{int}(v)$ refers to the energy in the surface, indicating the amount of deformation of an elastic surface. Consequently, the energy term $\mathcal{E}_{int}(v)$ increases when the surface is deformed with respect to its initial minimum energy state. The term for internal energy can further be decomposed into two terms (see [CC93a] for more details):

$$\mathcal{E}_{int}(v) = \int_\Omega \underbrace{\tau \left(\left| \frac{\partial v}{\partial s} \right|^2 + \left| \frac{\partial v}{\partial r} \right|^2 \right)}_{\text{tension}} + \underbrace{\rho \left(2 \left| \frac{\partial^2 v}{\partial s \partial r} \right|^2 + \left| \frac{\partial^2 v}{\partial s^2} \right|^2 + \left| \frac{\partial^2 v}{\partial r^2} \right|^2 \right)}_{\text{rigidity}} ds\, dr. \tag{3.10}$$

Those terms are used to implement tension ('rubber skin aspect') and rigidity ('thin plate aspect') of the model. Both terms are controlled by the two parameters τ and ρ (see [MT96] for a similar notion deriving the energy function in the 2D case or [LKE00] for the 3D case). The tension term makes the surface behave like rubber skin seeking to minimize the surface area. The thin-plate term is needed to yield a smooth surface and to prevent self-intersection. It hinders the model from bending and should be weighted carefully, not to over-smooth the surface. For simplicity, in this case, τ and ρ are interpreted as weighting coefficients with

$$\tau = 1.0 - \rho. \tag{3.11}$$

The rigidity coefficient ρ can, thus, be used to steer the properties of the material representing the model.

The actual deformation of the model is achieved by minimizing the energy functional given in Equation 3.10 with respect to the external energy function. Consequently, the surface is pulled towards the target surface, while at the same time a certain surface texture is maintained since the internal deformation energy is minimized simultaneously. The surface geometry has, therefore, to be changed in a way that minimizes $\mathcal{E}(v)$. It should be noted that the target geometry is not entirely unique, since the deforming model can get stuck in local extrema of the external energy function.

The energy minimizing functional as a model is often used in image registration because here, the deformation energy of the model is used as a measurement to determine shape similarity. The energy functional is minimized by deriving a system of partial differential equations. This, however, can lead to complex computations and is hard to implement numerically [MDA01]. An alternative to the energy minimization problem is the formulation as a dynamic force problem. Here, the vertex displacements are obtained by computing a force equilibrium between external and internal forces. This allows for the definition of more general external forces, hence, external forces that can not be written as the negative gradient of a scalar function [XPP00]. External forces that have been proposed e.g. in [CC93a, XPP00] include distance fields, pressure forces or forces derived from additional user input.

When the energy function is in a minimum, an Euler-Lagrange equation which brings external and internal forces into equilibrium is fulfilled. Using the internal energy term and the external energy term mentioned before, this yields [CC93b]:

$$\underbrace{\tau\Delta v - \rho\Delta^2 v}_{\mathcal{F}_{int}} = \mathcal{F}_{ext}, \tag{3.12}$$

where

$$\Delta v = \frac{\partial}{\partial s}\left(\frac{\partial v}{\partial s}\right) + \frac{\partial}{\partial r}\left(\frac{\partial v}{\partial r}\right) \tag{3.13}$$

and

$$\Delta^2 v = 2\frac{\partial^2}{\partial s\partial r}\left(\frac{\partial^2 v}{\partial s\partial r}\right) + \frac{\partial^2}{\partial s^2}\left(\frac{\partial^2 v}{\partial s^2}\right) + \frac{\partial^2}{\partial r^2}\left(\frac{\partial^2 v}{\partial r^2}\right) \tag{3.14}$$

Using the dynamic force based formulation, the evolution of the deformable surface can be computed iteratively in discrete time steps, until it becomes stationary [CC93b, SHL+11]. The surface position at time step $t+1$ can then be computed by

$$v^{(t+1)}(s,r) = v^{(t)}(s,r) + \sigma((1.0-\mu)\mathcal{F}_{int} + \mu\mathcal{F}_{ext}), \tag{3.15}$$

where μ is used to steer the influence of internal and external forces and σ is a general scaling factor to ensure convergence.

3.5.2 Discretization

In Section 3.5.1, the mathematical formulation of the deformable model approach was derived. The goal is now to discretize this approach. The triangle mesh Λ representing the discretized version of the continuous surface $\mathcal{S}$ is defined as a graph (X,E). Here, $X = \{\vec{x}_0,\ldots,\vec{x}_n\}$ is a set of vertices in $\mathbb{R}^3$ and $E \subset X \times X$ is a set of edges connecting the vertices. The deformation method needs to be discretized in space to be applied to the triangle mesh. To this end, discrete formulations of the internal and the external force have to be derived.

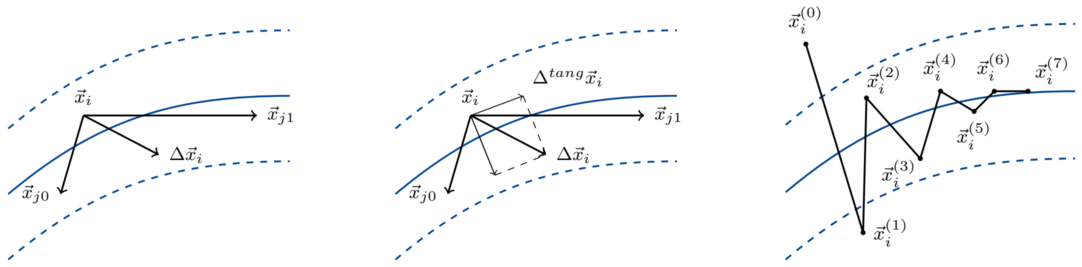
The external force on every vertex $\vec{x}_i$ can be obtained by sampling a pre-computed external energy volume and computing the gradient of its image intensity at every iteration step. This gradient is tweaked to make the surface deform towards the target surface. In this work the gradient is normalized and scaled in the following way: First, it is determined whether the initial vertex position is inside or outside the volume. The sign is then changed based on the underlying external energy volume so that the gradient always points in the direction of the isosurface. When a vertex crosses the isovalue, the sign of the gradient is reconsidered and the gradient is scaled by a factor of 0.5 (see Figure 3.4c).

In [TF88] central differences are used to discretize the computation of the internal forces for every vertex. They, however, discretize the surface by a rectangular grid with a (u,v) parameterization. Since, in this case, a simplex mesh is used, other methods have to be considered. A discrete formulation of the internal force can be otained by a discrete approximation for the Laplacian in triangle meshes. In [RBG+09], several approximations for the discrete Laplacian in arbitrary meshes are proposed. In this work, the rather simple approximation of a uniformly weighted Laplacian is used. Here, the Laplacian for a vertex $\vec{x}_i$ is computed as the sum of all the vectors pointing from $\vec{x}_i$ to its adjacent neighbors. Therefore,

$$\Delta\vec{x}_i = (1.0-\rho)\,\frac{\sum_{j\in N(i)}(\vec{x}_j - \vec{x}_i)}{|N(i)|}, \tag{3.16}$$

where $N(i)$ denotes the set of all vertices in X that are adjacent to $\vec{x}_i$, i.e. $(\vec{x}_i,\vec{x}_j) \in E$. This approximated Laplacian has been used before in a similar context [LKE00, SHL+11] and is illustrated in Figure 3.4a. As stated in [LKE00], a discrete version of the Δ^2 operator, which is needed for the rigidity term, can be computed by recursively using the same computation method. Thus, it is be obtained by

$$\Delta^2\vec{x}_i = \rho\,\frac{\sum_{j\in N(i)}(\Delta\vec{x}_j - \Delta\vec{x}_i)}{|N(i)|}. \tag{3.17}$$



(a) Discrete Laplacian *(b) Decomposition* *(c) External force*

Figure 3.4: Illustration of different forces used in the deformable model approach. (a) shows the computation of internal force by using a discrete approximation of the Laplacian $\Delta\vec{x}_i$. The projection of the force is illustrated in (b). The development of the external force over several iteration steps is shown in (c) (exaggerated for clarity).

In this work, the internal force is further modified. As stated before, the external force used in this approach is scaled down adaptively. This leads to fast convergence, if no internal forces are present. If, however, the internal force is present, vertices could get pulled away from their positions on the target surface. The scaling would then have to be reversed, which would be a complex task, or else the vertex would need a lot more iteration steps to once again reach the target surface. Therefore, the internal force is modified so that it can only move the vertices orthogonal to the volume gradient. That way, the deformation converges while a regular surface is maintained. The projection of the internal force vector $\vec{f}_{int}$ onto the plane tangential to the volume is done by decomposing it into two terms [SHL+11]: one term $\vec{f}_{int}^{tang}$ that is tangential to the target volume and one term $\vec{f}_{int}^{perp}$ that is perpendicular to the target volume. The term tangential to the volume, which is used in this case, is calculated by

$$\begin{aligned} \vec{f}_{int}^{tang} &= \vec{f}_{int} - \vec{f}_{int}^{perp} \\ &= \vec{f}_{int} - (\vec{f}_{ext} \cdot \vec{f}_{int}^{perp})\vec{f}_{ext}, \end{aligned} \tag{3.18}$$

where $\vec{f}_{ext}$ is perpendicular to the volume since it defers from its gradient only by a scalar factor.

Using this modified internal force in combination with the scaled external force causes the surface vertices that reach the target to be held strongly on the surface, allowing them to move only slightly. The tension term of the internal force causes the deformable model to behave similar to a spring mass model with all springs having equilibrium length of zero. This ensures that the mesh tends towards evenly distributed triangles. The rigidity term makes the model behave like a bendable thin plate and prevents the vertices from self-intersection while they are moving on the surface.

The deformation of the surface can now be computed by applying the discrete external and internal forces in a number of successive iterative steps until a stationary solution is reached. Figure 3.5 shows the mapping between the two discrete surfaces. Given initial vertex positions $\vec{x}^{(t)}$ one iteration of the deformation algorithm can be summarized by the following steps:

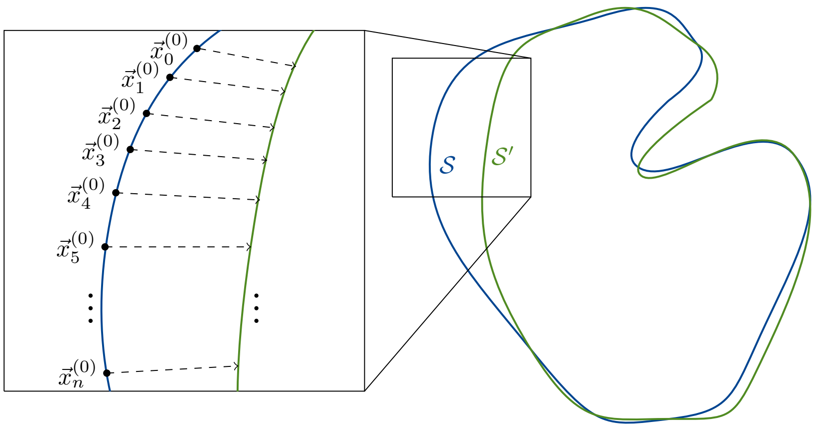


Figure 3.5: The mapping between the two discrete surfaces. $\mathcal{S}$ is the source surface and $\mathcal{S}'$ is the target surface. The deformation moves the discrete vertices of $\mathcal{S}$ towards the target shape.

1. Sample and scale the external force $\vec{f}_{ext}$
2. Compute the discrete Laplacian $\Delta\vec{x} = \frac{\sum_{j\in N}(\vec{x}_j - \vec{x})}{|N|}$
3. Compute $\Delta^2\vec{x} = \frac{\sum_{j\in N}(\Delta\vec{x}_j - \Delta\vec{x})}{|N|}$
4. Compute the internal force $\vec{f}_{int} = (1.0-\rho)\Delta\vec{x} - \rho\Delta^2\vec{x}$
5. Compute the tangential component of the internal force $\vec{f}_{int}^{tang}$
6. Update the vertex position $\vec{x}_i^{(t+1)} = \vec{x}_i^{(t)} + \sigma(\mu\vec{f}_{ext} + (1.0-\mu)\vec{f}_{int})$

3.5.3 Definition of the External Energy Potential $\mathcal{E}_{ext}$

In this section, different approaches of defining an external force are discussed and the approach used in this work is explained. There are a number of properties that the external energy volume should have. One of the problems deformable model approaches often have to deal with are local extrema. Local extrema in the external potential function are induced by noise in the underlying images and can cause the model to get stuck. The generated volume should, therefore, be smooth and should not contain any substantial noise. Another issue is that shapes that start out too far away from the target shape are not reached by the energy function and are, therefore, not attracted to the target. The volume should, thus, have a wide enough range to ensure attraction for all parts of the source surface. Finally, the volume should represent the target shape as accurately as possible.

Using the particle-based Gaussian volume texture (see Section 3.2) as external energy function seems viable, but has two major disadvantages. First, the Gaussian volume exhibits local extrema inside the molecular surface. These local extrema are caused by the overlapping of

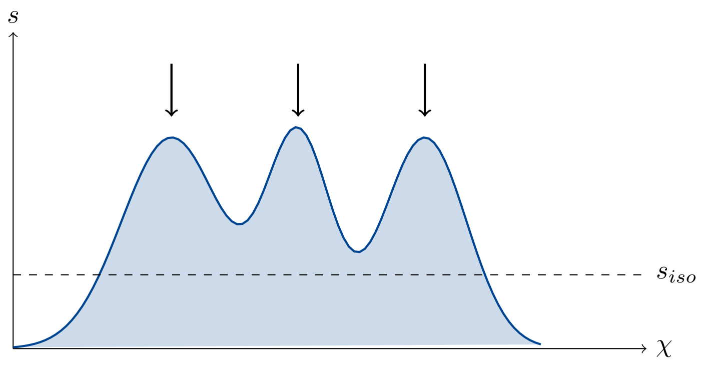


Figure 3.6: Accumulated Gaussian distribution for three particles as described in [Bli82]. The horizontal axis represents the spatial variation, the vertical axis is the scalar value of the Gaussian volume. s_{iso} is the isovalue defining the isosurface. Local extrema in the Gaussian volume, located at the positions of the particles, are denoted by arrows. If vertices of the source shape start out inside the isosurface beyond one of the local maxima, they will get pulled inside the molecule and, therefore, away from the surface.

several of the Gaussian distributions (see Figure 3.6 for an illustration). This does not cause problems when the two shapes are quite similar, which is often the case when comparing molecular surfaces. If, however, one of the structures has e.g. a cavity, then there is a certain chance that the vertices in this cavity will get stuck in a local extremum. Problems with local extrema in the context of deformable models are often dealt with by tweaking the internal or external forces. In [Coh91], e.g., an additional pressure force is introduced to push the shape out of local extrema. In that context, the deformable model is used for image segmentation, therefore, the additional distortion of the vertices is not an issue. In this work, however, a mapping relation is defined based on the mapped vertices and unpredictable and chaotic movements should be avoided. The second disadvantage is the fast convergence to zero of the density field. If vertices do not start out close enough to the target shape, they might not be attracted to it, since the local attraction potential is zero. In the following, several alternatives for the particle-based Gaussian volume are discussed that avoid issues with local extrema and the missing attraction of vertices.

In the context of medical images, a common method to increase the range of the external energy is the convolution of the original picture with a Gaussian kernel [Coh91]. In a similar fashion, in this case, a binary image could be extracted from the Gaussian volume, flagging lattice points inside the volume with '0' and outside with '1'. The Gaussian kernel then would have to be chosen so broad that every starting vertex is reached by the resulting energy function. One advantage of this approach, in addition of having no unwanted local extrema, would be that it is very flexible and basically every implicitly represented surface could be processed by the same approach. The disadvantage is that using a broad Gaussian kernel would necessarily introduce an error to the computation and cause the loss of details. Another error is introduced by turning the volume into a binary function. Furthermore, even when choosing a very broad Gaussian, there would be no guarantee that all vertices are attracted.

Another possibility is using a distance field as the target volume. This distance field would be based on a regularized discrete representation of the target surface. Distance fields have been used in the context of mesh morphing [COSL98] or collision detection [Eri05]. The distance field of a discrete shape can be obtained by computing the distance to the nearest vertex for every lattice point. More sophisticated approaches determine the nearest triangle and compute the orthogonal projection of the lattice point to that triangle [Eri05]. Distance fields have an infinite range, which solves the attraction problem. A drawback to using a distance field of a discrete mesh is that it transfers the discretization error to the volume. This error is most noticeable when approaching the target surface. Here, the Gaussian volume is superior, since it accurately pulls the vertices towards the centers of the particle locations, rather then towards certain surface locations.

The alternative used in this work is, therefore, the combination of the distance field approach with the Gaussian volume to combine the advantages of both methods. The distance field is used when vertices are far away from the target surface. This makes sure that the energy is reaching all of the surface points and additionally avoids the local extrema inside the molecule. When a certain distance threshold is overcome, the vertex uses the Gaussian volume instead. During the experiments done for this work, a threshold that is about half of the smallest atom radius seemed to be a good value.

3.5.4 Parameters

There are four parameters in total that can be exposed to the user to allow steering the deformation process. Choosing those parameters carefully is crucial for a satisfying outcome of the mapping algorithm. In the following, the parameters and their influence are explained.

- σ – This parameter serves as the global scaling factor for all forces. Increasing this parameter leads to faster convergence. Setting it too high, however, can cause the vertices to immediately jump over the target surface and get stuck at very distorted positions.
- μ – This parameters serves as a weighting function for the external and internal forces. It can obtain values in the interval of $[0, 1]$. The weighting of the internal forces is implicitly defined by $1.0 - \mu$. Setting a high weight on the external forces (and implicitly a low weight on the internal forces) hinders vertices from moving once they have reached the target surface. If the internal forces have higher weight, the vertices can move freely on the surface (while minimizing the internal energy). This, however, leads to higher distortion and slower convergence.
- ρ – This parameter defines the internal physical properties of the deformable model. It describes the amount of rigidity of the elastic surface. The tension of the surface, i.e. the pursuit of the surface to minimize its surface area, is implicitely defined by this parameter, since $\tau = 1.0 - \rho$. Giving ρ a high value prevents self-intersection of the surface, but can also cause over-smoothing since it prevents the surface area from bending in sharp angles.

- f_{min} – The minimum displacement parameter f_{min} serves as an ending condition for the iteration process. Especially when the internal force is weighted very high, minor displacements will keep appearing, since one small displacement is propagated through the entire mesh by the application of the discrete Laplacian. Thus, the average displacement of all vertices in every iteration step is computed and the iteration is stopped if the average displacement falls under the value of f_{min}.

3.5.5 Improving the Mapping through Prior Regularization

A criterion for a good mapping is that the deformation needed for the mapping is minimal. The amount of deformation can be quantified by the internal energy present in the model after the mapping. The more a shape has to be deformed to match a target, the more internal energy is present in the model after the deformation. Consequently, if the source and the target shape are identical, the difference of the internal energy in the source shape before and after the deformation process is zero. For this assumption to hold, the internal energy at the beginning of the iteration has to be minimal. If a source shape starts out in a state with non-minimal internal energy, the deformation process seeks to minimizes this initial internal energy, simultaneously to the actual deformation. As a result, the mapping relation gets less accurate, since the deformation of the final shape contains the additional deformation caused by the non-minimal internal energy. Depending on how the internal springs are defined, the minimum energy condition can be met in different ways, each of which has advantages and disadvantages to it.

One possibility is to define the initial length of the internal springs to be their equilibrium length. This approach, however, would require a different energy function, since the tension term assumes the equilibrium length of the springs to be zero. Furthermore, the final mesh is likely to be very irregular, since the triangle mesh created by the Marching Tetrahedra method is very irregular to begin with. Diminishing the negative effects of this irregularity would then require a higher grid resolution.

The second approach that can be used to ensure minimal internal energy at the beginning of the iteration is to use a preprocessing step in which the surface is regularized until its internal energy is minimized. Here, the equilibrium length of the springs is zero. The external force can be derived from the volume representing the same surface implicitly. The external force's length then converges to zero very quickly, since the target volume and the source mesh are based on the same shape and, thus, the starting positions are close to the target. The internal force, which should get a higher weighting in this case, however, takes long to converge to a minimum. In this work, the second regularization approach is used, since it is more flexible and the resulting mesh has a higher quality.

Figure 3.7 shows a synthetic surface as produced by the marching tetrahedra algorithm using the Freudenthal subdivision and the same mesh after regularization with the approach described above.

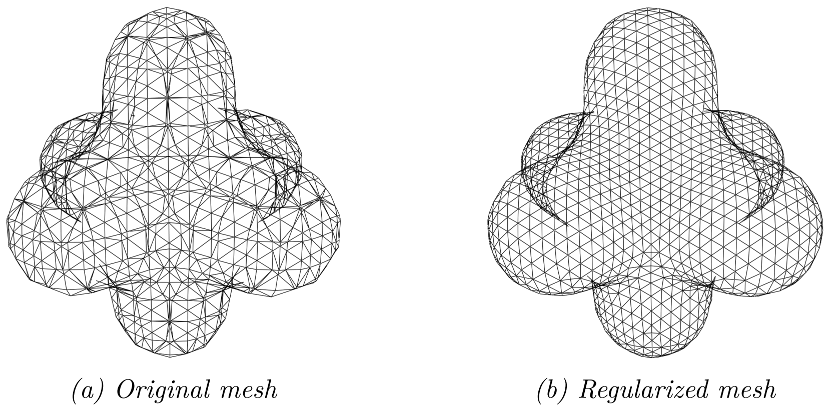


Figure 3.7: Comparison of original and regularized surface of a synthetic data set (the back-faces have been culled for clarity). (a) clearly exhibits a high number of small triangles, whereas in (b), the mesh consists of evenly distributed regular triangles.

3.5.6 Error Quantification

Since the mapping is essentially an approximation, one of the desired properties of the mapping relation is the possibility of error quantification. This is necessary to evaluate the quality of the surface mapping with respect to different input parameters. There are several criteria classifying the quality of a mapped surface (i.e a fully converged shape with zero net force). The first criterion is the absence of surface intersection, since surface intersection results in the mapping not being bijective anymore. Another criterion is the accuracy of the spatial mapping, i.e. how well the mapped shape fits the actual target shape. The last criterion is the overall semantic correspondence achieved by the mapping. The deformation approach is also used to regularize the initial mesh. Therefore, for the regularization process, additionally, there is the criterion of regularity.

The first criterion for the quality of the mapping is the absence of self intersection in the fully converged surface. A way to compute the number of intersecting triangles is described in [Mö97]. Here, in order to test two triangles against each other, each triangle is represented by a plane. A line segment is defined by the intersection of those planes. If the both triangles intersect the line segment and the resulting line intervals overlap the triangles are intersecting. This test is computationally expensive, since it tests all triangles against all other triangles. Other possibilities are mentioned in [Eri05]. The percentage of triangles of one surface that intersect with other triangles of the same surface could be used as a measurement for self-intersection.

The accuracy by which the mapped shape fits the target shape could be quantified in different ways. One common way of quantifying geometric differences of surfaces is to compute the *Hausdorff distance* [NSCE02, CRS98]. Let $\mathcal{S}$ and $\mathcal{S}'$ be two surfaces. The distance of an arbitrary point $\vec{p}$ of $\mathcal{S}$ to the surface $\mathcal{S}'$ is defined by

$$d(\vec{p},\mathcal{S}') = \min_{\vec{p}' \in \mathcal{S}'} \|\vec{p} - \vec{p}'\|_2 \tag{3.19}$$

and, based on that definition, the Hausdorff distance is given by

$$d(\mathcal{S}, \mathcal{S}') = \max_{\vec{p} \in \mathcal{S}} d(\vec{p}, \mathcal{S}'). \tag{3.20}$$

Since this metric only uses extreme values, it is only a very vague estimation of the actual difference. A variation of this metric, described in [NSCE02] that takes more details into account computes the average $d(\vec{p}, \mathcal{S}')$ of all surface points of $\mathcal{S}$. This yields a mean distance value for the shape.

Another criterion for the fitting of the mapped surface to the target surface is the amount of surface area of the mapped surface that consists of corrupt – i.e. badly mapped – triangles. Finding corrupt triangles could be done by flagging triangles which are acute-angled or oversized. It would, however, be hard to make sure that non-corrupt triangles are not flagged as well using a purely geometric approach like this. Thus, the method to identify corrupt triangles used in this work samples the target volume at the centroid of the triangle and flags the triangle as corrupt if the sample differs greatly from the isovalue. A criterion for the overall quality of the mapping is then defined by the percentage of the surface area that consists of corrupt triangles. The mean Hausdorff distance and the area percentage of corrupt triangles can, in some situations, deliver very different results. Hence, it is advisable to use both metrics to quantify the fitting of the mapped surface to the target shape.

The third criterion is that the final mapping maintains a certain amount of semantic correspondence. The only kind of semantic correspondence used in this work is the RMSD minimization which is based on paired atoms. A low RMSD value of the underlying molecular structures could, therefore, be seen as a measurement for semantic correspondence. This semantic correspondence, however, is not influenced by the deformation and can, thus, not be used as a quality criterion for the non-rigid alignment. In general, the amount of semantic correspondence depends heavily on the user and the subjective of the mapping. This impression is reinforced by the fact that most approaches to shape correspondence have some kind of user input in order to define feature points on the surfaces that need to be connected (see e.g. [ZSCO$^+$08]). Error quantification on the level of semantic correspondence is, therefore, beyond the scope of this thesis.

The success of the regularization process can be quantified by how regular the distance of one vertex to all its neighbors is. The internal force as defined in Section 3.5 seeks to minimize the overall surface area, which leads ultimately to each vertex having the same distance to all its neighbors. Thus, the length of the non-weighted internal force acting on each vertex can be used as a measurement for the irregularity. This value is averaged over all surface vertices to yield a metric for irregularity.

In this work, the corrupt triangles' area percentage and the mean Hausdorff distance are used to quantify the fitting of the mapped surface to the target surface. The regularity is measured by the mean internal force length present in the surface mesh after the shape is fully converged. A measurement for self-intersection has not been implemented due to time constraints.

3.6 A Metric to Quantify Differences in Molecular Surface Attributes

In Section 3.5, a method to yield a bijective mapping relation between two input shapes was derived. This mapping relation is used to compare arbitrary surface attributes of both input molecules. Though, in this work, special attention is placed on the electrostatic surface potential. Here, both the absolute value of the potential and the potential sign are taken into account, since mixing those values could yield misleading results. The absolute difference is used to compute a *mean potential difference*. The potential sign is used to compute the percentage of the mapped surface on which the potential has a different sign in both input shapes. In both cases a single metric value is yielded, suitable to compare arbitrary numbers of variants at once. In the following, the mean potential difference and the metric derived from the potential sign are defined. Afterwards, issues when transforming both metrics to the discrete case are considered. Finally, a measurement for uncertainty of the result is proposed.

Let $\mathcal{S}$ and $\mathcal{S}'$ denote two continuous surfaces with $(r,s) \rightarrow \mathcal{S}(r,s)$ and $(u,v) \rightarrow \mathcal{S}'(u,v)$. Let, furthermore, a bijective mapping between the two surfaces be defined by $m : (r,s) \rightarrow (u,v)$. The surface potential is denoted by $\phi : [0,1][0,1] \rightarrow \mathbb{R}$ for both surfaces. The absolute surface potential difference d_ϕ on one of the surfaces is given by

$$d_\phi(r,s) = |\phi(r,s) - \phi(m(r,s))|. \tag{3.21}$$

A simple function d_{sign}, taking the potential sign into account, can be defined by

$$d_{sign}(r,s) = \begin{cases} 1, & \text{if } \operatorname{sign}(\phi(r,s)) \neq \operatorname{sign}(\phi(m(r,s))) \\ 0, & \text{otherwise} \end{cases}. \tag{3.22}$$

Both the potential difference and the sign difference can be used in a comparative visualization for surface texturing. In order to compare an arbitrary number of variants, a function that summarizes that error for one pair of input data has to be defined. To this end, d_ϕ and d_{sign} are integrated over the surfaces and then normalized by dividing the value by the total surface area. This yields a mean potential difference $\tilde{d}_\phi$, defined by

$$\tilde{d}_\phi(\mathcal{S},\mathcal{S}') = \frac{1}{|\mathcal{S}|} \int_\Omega d_\phi(r,s) d\mathcal{S}. \tag{3.23}$$

The area percentage with different potential sign $\tilde{d}_{sign}$ is computed the same way.

Since the input surfaces are given as discrete triangle meshes, the approach is now defined for the discrete case. Let X and X' be two triangle meshes representing the discretized versions of $\mathcal{S}$ and $\mathcal{S}'$. Here, X' was obtained by using the mapping approach described in Section 3.5. Thus, there is a direct vertex-to-vertex relation available for X and X'. This relation is now used to compute $d_\phi(r,s)$ and $d_{sign}(r,s)$ for all vertex pairs. In order to discretize the integral in equation 3.23, the values of $d_\phi(r,s)$ and $d_{sign}(r,s)$ are first integrated over each triangle, and finally, the values are accumulated over all triangles. In the following, only the potential difference d_ϕ is referred to, the computations for d_{sign} are identical.

There are different ways of integrating values over simplexes differing in computational complexity and accuracy [HMS56]. If it is assumed that a continuous representation of d_ϕ can be obtained by linear interpolation, the integration can be computed by the affine invariant approach described in [HMS56]. They suggest evaluating the function to be integrated at the center and weighting it with the total area of the triangle. Since the barycentric coordinates of the centroid are $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})^T$, d_ϕ can be evaluated at the centroid by averaging over the values sampled at the triangle corners. Let the averaged value be d_{avg}, then the integrated potential difference of one triangle becomes

$$\int_\Omega d_\phi = A\, d_{avg}\,, \tag{3.24}$$

In [NSCE02] a similar approach is used to integrate the Hausdorff distance over a triangle mesh.

The potential difference is then accumulated over all triangles and finally divided by the total surface area to yield the mean potential difference

$$\tilde{d}_\phi(X, X') = \frac{1}{|X|} \sum_{t=1}^{|T|} A(t) d_{avg}(t), \tag{3.25}$$

where T is the set of all triangles in X, $d_{avg}(t)$ denotes the potential difference sampled at the centroid of the triangle t, and $A(t)$ is the area of the triangle t.

Since the error caused by the mapping approximation cannot be quantified exactly, a measure for uncertainty of the final computation is proposed. The idea is that the more the surface has to be deformed to yield the mapping, the higher the uncertainty of the outcome is. There are several ways of quantifying the deformation of the surface. The approach used in this work is the Hausdorff distance mentioned before. The Hausdorff distance is computed for every vertex and the distance value is averaged over all vertices to yield a mean distance value. The higher that mean distance value is, the more uncertain the outcome of the mapping becomes.

In this work, the bijective mapping relation is used to compute the potential difference and the potential sign flag on a per-vertex level. Those values are then integrated over the whole surface area to compute the mean values. These mean values serve as a metric to compare two or more variants of molecules and can be used to visualize an overview of a number of variants.

3.7 Summary

In this chapter, a framework to establish shape correspondence between two molecular surfaces was derived. The implicit surfaces are first defined by a Gaussian density distribution and an appropriate isovalue. Next, the source shape is triangulated using the Marching Tetrahedra method. Since the triangulation that the Marching Tetrahedra method yields can be very irregular, a regularization step has to be applied to the mesh, in order to ensure minimal internal energy. Before the non-rigid alignment, a rigid alignment step is performed to increase semantic correspondence. To this end, the transformation needed to minimize the RMSD is computed. The transformation is then applied to the regularized mesh. Please note that

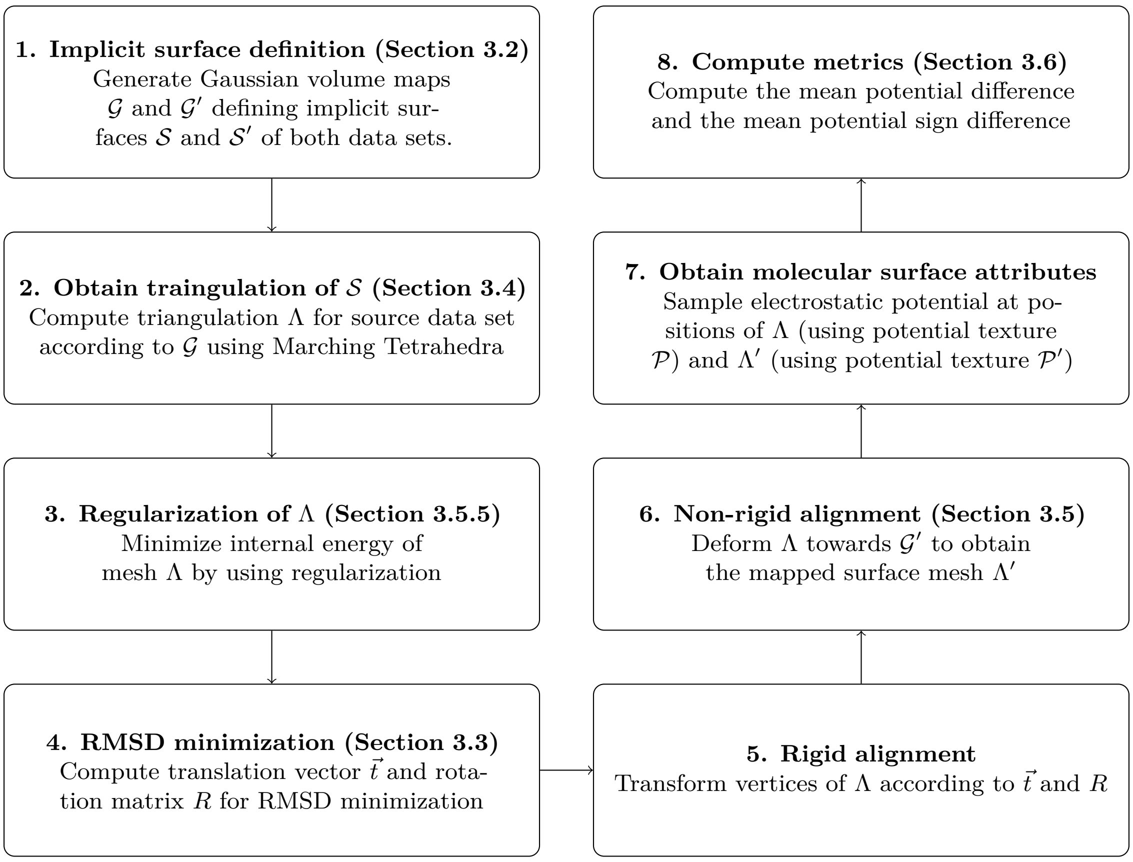

1. Implicit surface definition (Section 3.2)
Generate Gaussian volume maps $\mathcal{G}$ and $\mathcal{G}'$ defining implicit surfaces $\mathcal{S}$ and $\mathcal{S}'$ of both data sets.
2. Obtain traingulation of $\mathcal{S}$ (Section 3.4)
Compute triangulation Λ for source data set according to $\mathcal{G}$ using Marching Tetrahedra
3. Regularization of Λ (Section 3.5.5)
Minimize internal energy of mesh Λ by using regularization
4. RMSD minimization (Section 3.3)
Compute translation vector $\vec{t}$ and rotation matrix R for RMSD minimization
5. Rigid alignment
Transform vertices of Λ according to $\vec{t}$ and R
6. Non-rigid alignment (Section 3.5)
Deform Λ towards $\mathcal{G}'$ to obtain the mapped surface mesh Λ'
7. Obtain molecular surface attributes
Sample electrostatic potential at positions of Λ (using potential texture $\mathcal{P}$) and Λ' (using potential texture $\mathcal{P}'$)
8. Compute metrics (Section 3.6)
Compute the mean potential difference and the mean potential sign difference

Figure 3.8: Summary of the shape correspondence framework for molecular surfaces.

the source mesh is regularized before the RMSD transformation, since the regularized vertex positions are needed to sample the potential texture of the source data set. The transformed mesh is then deformed according to internal and external forces, until a stationary solution is found. Afterwards, the potential texture values can be sampled at both the old (regularized, non-RMSD-transformed) positions and the new positions to compute the mean potential difference and the mean potential sign difference. A summary of the steps involved in the shape correspondence approach can be found in Figure 3.8.

4 GPU Implementation & Rendering

The GPU implementation of the surface mapping are now described with the focus being on the CUDA kernel implementations. The notation used for different indexing schemes is explained, as well as the most important lookup-tables, which are used heavily in the implementation. The implementation of the individual steps of the mapping framework is demonstrated, including the generation of the initial triangle mesh, the regularization, the actual mapping, and the computation of the metric. Here, the notation used for pseudo-code is similar to the actual CUDA code, however, some parts of the code are summed up for clarity. Next, the rendering of the results is outlined. Both a 3D surface rendering and a 2D plot have been implemented. Finally, the modules and calls are described in the context of the visualization framework MegaMol by providing the respective call graphs.

This chapter does not include a description of how to compute the potential textures. For this thesis, the potential textures were obtained using the PME plugin in the visualization software VMD[1]. This chapter, furthermore, does not contain descriptions of implementations that were already present in the MegaMol framework. This includes the implementation of the RMSD minimization (see Section 3.3) and the generation of the Gaussian volume (see Section 3.5.3). The RMSD minimization is implemented using a CPU implementation of the the algorithm by Kabsch [Kab76, Kab78]. The implementation of the volume generation is the one described in [KSES12], which implements the accumulation of the Gaussian distributions as a parallel gathering algorithm on the GPU.

4.1 Graphics Pipeline & GPU Architecture

Before the actual implementation is explained, a short overview of the graphics pipeline and the GPU architecture is given. Additionally, libraries and APIs used in the implementation are listed.

[1] http://www.ks.uiuc.edu/Research/vmd/

4.1.1 OpenGL (Open Graphics Library)

OpenGL (Open Graphics Library) is an API for 2D and 3D real-time computer graphics. The function definitions associated with the API have numerous language bindings including C, C++, Fortran, Pascal or Ada. It offers hardware-accelerated rendering and is used in different applications like CAD, virtual reality, scientific visualization, simulations, and computer games. Furthermore, in contrary to the widely used DirectX API, OpenGL is platform-independent (supported platforms are Unix, Windows and OSX).

In the following, the typical graphics pipeline for OpenGL and its components are outlined. The pipeline described here is a simplified version of the one found in [BSW+07]) and is illustrated in Figure 4.1, while highlighting the programmable units avilable in OpenGL 2.1. Vertex data, representing the geometry of the scene, is sent from the application memory to the graphics pipeline. Vertex data, in this context, refers to spatial coordinates, normals, colors, texture coordinates, or other attributes that can be defined on a per-vertex basis. During the vertex processing step, the spatial vertex coordinates are transformed to clip space using the modelview and the projection matrix. The vertex processor can be programmed using a *vertex shader*. The primitive assembly step uses connectivity information of the vertices to form primitives like e.g. triangles. The vertices are then clipped against the viewport and projected from the three-dimensional clip space to the two-dimensional screen space. In the rasterization step, the primitives are transformed into fragments (where each fragment square corresponds to one pixel in the framebuffer). In the fragment processor, which can be programmed by a *fragment shader*, the fragments are assigned a color and a depth value. A fragment consists of different attributes like a depth value, a stencil flag or a color. Next, per-fragment operations like the depth test or blending are performed to yield the pixel groups. In contrast to fragments, pixels represent the final color value of the respective framebuffer position. As the final step, the pixel groups are written to the framebuffer.

4.1.2 GLSL (The OpenGL Shading Language)

GLSL (The OpenGL Shading Language) is a high-level shading language whose syntax is based on ANSI C. It allows the programmer to replace certain parts of the OpenGL fixed function graphics pipeline with shaders. As mentioned before, in OpenGL 2.1, these parts are the vertex processing step and the fragment processing step.

In GLSL, variables can be declared in different ways, depending on what the purpose is and on how frequently the variable is changed [RLKG+09]:

- Variables of type `uniform` are written by the application and are a means to provide data to the graphics pipeline.
- Variables of type `varying` are output variables for the vertex shader, the fragment shader receives a linearly interpolated value of that variable as input.
- Variables of type `attribute` can be used by the application to store vertex attributes or other data, that is frequently changed.

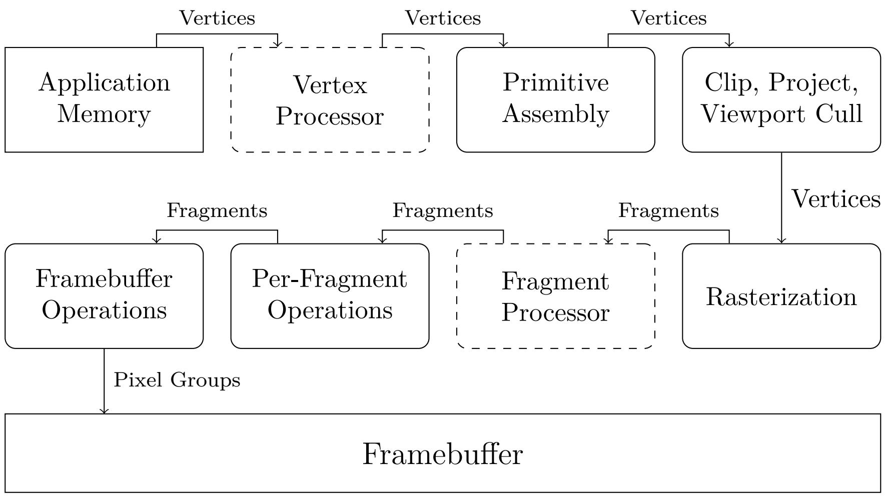


Figure 4.1: The OpenGL 2.1 rendering pipeline. Programmable units that can be changed by the use of GLSL shaders (see Section 4.1.2) are highlighted by a dashed box.

The *vertex processor* is a programmable unit that takes vertices as input and can be used to transform vertices and change their associated attributes. The functionality of this processing unit can be changed using *vertex shaders*. The vertex processor operates on one vertex at a time and has no knowledge about vertex topology. Intended functions mentioned in [RLKG+09] include vertex transformation (modelview and projection matrices), normal transformation and normalization, texture coordinate generation, texture coordinate transformation, lighting and color material application.

The *fragment processor* can execute *fragment shaders*. It operates on fragment units and can change their associated attributes. Some typical tasks for fragment shaders, mentioned in [RLKG+09] include processing of interpolated values, texture access, alpha testing or texture application. Fragment shaders can be used to implement per-pixel-lighting, which replaces the per-vertex-lighting that is implemented in the fixed function pipeline of OpenGL. The fragment shader receives interpolated vertex attributes that have been initialized in the vertex shader using the `varying` keyword.

4.1.3 CUDA (Compute Unified Device Architecture)

CUDA (Compute Unified Device Architecture) is a programming model that has been created by NVIDIA and is being implemented in their GPUs. CUDA comes with a set of functions that allow developers to use high-level programming languages such as C or FORTRAN. In this case, *CUDA C/C++* is used with the provided compiler `nvcc`. CUDA allows using the parallel GPU architecture to facilitate heterogeneous programming. Thus, serial parts of the computation are executed on the CPU, whereas parallel parts can be moved to the GPU.

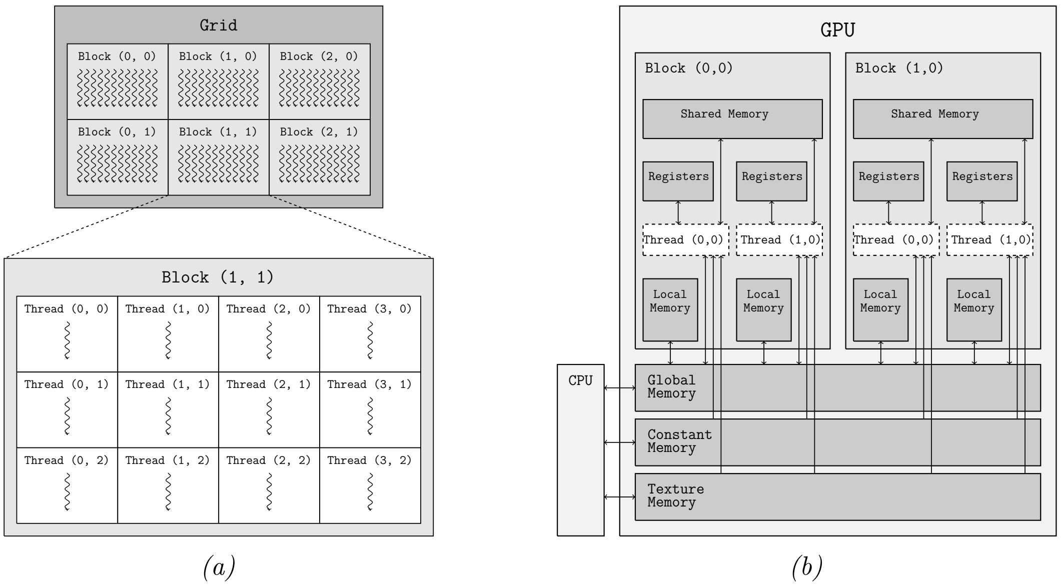


Figure 4.2: CUDA memory architecture (b) and CUDA grid layout (a) (c.f. [Nvi12]).

Code that is executed on the CUDA device is defined in so-called *kernels*. A kernel is executed by a large amount of threads in parallel with each thread handling a small part of the overall computation. In order to ensure lasting compatibility with future hardware, the threads are organized in a hierarchical manner, where a certain number of threads are grouped into a *block* and blocks are further grouped in a *grid*. Each thread in one block has a unique ID to it. The same holds for each block in a grid. The hierarchical layout is illustrated in Figure 4.2a.

The programming model introduced by CUDA is an abstraction of the underlying GPU architecture. This architecture is built around a scalable array of multi-threaded *Streaming Multiprocessors (SMs)*. When a kernel grid is invoked, the blocks of the grid are enumerated and distributed to the SMs. This distribution is based on the available execution capacity of the SMs. Every SM organizes and executes threads in groups of 32 parallel threads called *warps*. The threads inside one warp have consecutive thread IDs (i.e. the threads of the first warp have the IDs 0 to 31 and so on). Before a SM executes a thread block, it partitions it into warps, which then get scheduled by a warp scheduler for execution. Here, the way conditional statements are used can have a huge impact on the overall computation cost. All threads in one warp execute one common instruction at a time. Thus, all threads in the same warp should execute the same set of instructions to minimize computation time. Conditional statements in the kernel code can lead to different branches consisting of different sets of instructions, which are then executed in serial. Threads in different warps, however, can execute different conditional branches independently without the need for serialization.

CUDA threads can access data from different memory spaces, which have different scopes (Figure 4.2b). *Registers* and *local thread memory* are only accessible for one thread. *Shared memory* can be accessed by all threads in one block and serves as a means of communication between all threads in that block. *Global device memory*, *constant device memory*, and *texture memory* can be accessed by all threads of a grid and by the host application. Global device memory, constant device memory, texture memory, and local memory are ’off-chip’ and have much slower access then shared memory or local registers. A common programming strategy is, therefore, to tile the input data and store it to shared memory. The result can then be computed in a divide and conquer manner by distributing the work between different blocks.

Besides the *CUDA Runtime API*, the *CUDA Driver API*, and the *CUDA Math API*, there are several libraries with high-level implementations that come with the CUDA toolkit to facilitate efficient programming. THRUST is a library consisting of C++ templates that is based on the Standard Template Library (STL). The implementation done in this work makes heavy use of the THRUST library, mainly using the sorting algorithms, but also for reduction and summation of device arrays.

4.2 Indexing Notation & Lookup-Tables

The implementation of the surface generation and the surface mapping approach derived in Chapter 3 uses numerous different indexing schemes for the entities involved in the computation. All of these entities are defined with respect to the volume texture that defines the implicit surface. The first entities are grid cells in the volume texture and the respective vertices at the grid cell corners. Furthermore, entities that need to be defined are tetrahedrons, vertices at tetrahedron corners, tetrahedron edges, and, finally, the vertices that are extracted with the Marching Tetrahedra algorithm for the explicit surface representation. All of these entities are referenced using different indexing schemes. The implementation done in this work makes heavy use of lookup-tables, which are mainly used to transform entities from one index scheme to another. In order to simplify the description of the implementation in the following sections, the notation used for all the entities and their respective indices are defined in this section. Additionally, the most important lookup-tables are explained for later reference.

4.2.1 Notation

The basis for most of the definitions is the volume texture computed by the approach described in Chapter 3. The size of the volume texture is given by $dim = (dim_x, dim_y, dim_z)^{\mathrm{T}}$. The lattice points of the grid are denoted by $x = (x_x, x_y, x_z)^{\mathrm{T}}$.

The elements in the set of all cells C of the volume texture are referenced in two different ways. The global cell index c_i can be computed by the lattice position of the cell in the volume. Let $c = (c_x, c_y, c_z)^{\mathrm{T}}$ be the lattice position of the cell origin. Then, the global cell index c_i is given by

$$c_i = (dim_x - 1) * ((dim_y - 1) * c_z + c_y) + c_x \tag{4.1}$$

The other way of indexing is the active cells’ index a_i, which enumerates the set of all active cells, denoted by $A \subset C$. In this context, the term ’active’ refers to cells which contain parts

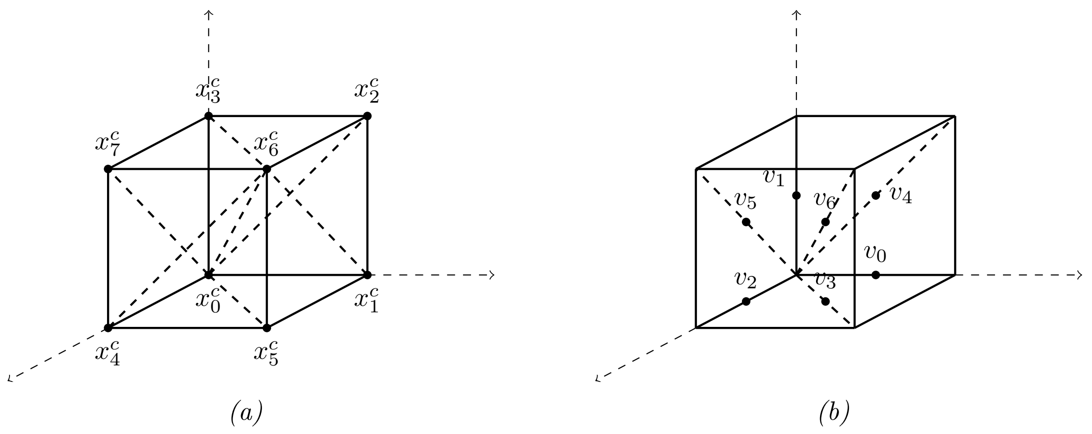


Figure 4.3: The vertex indexing scheme for the calculation of vertex positions. x_0^c *to* x_7^c *are the local indices of the cell corners.* v_0 *to* v_6 *are the local indices of the possible surface vertices associated with this cell.*

of the explicit surface representation. The ordering of the elements described by the index a_i is the same as in the global cell index. Hence, for active cells' indices a_i and a_j and their indices with respect to C, c_k and c_l, following statement is true:

$$\forall i, j \in A : (c_k < c_l) \rightarrow (a_i < a_j) \tag{4.2}$$

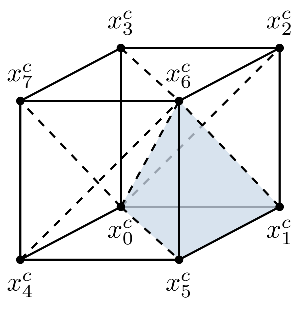
The eight corners of one cell are basically lattice points with a local index. They are denoted by $X^c = \{x_0^c, \ldots, x_7^c\}$. Their exact location with respect to the cell origin is illustrated in Figure 4.3a.

The tetrahedrons that subdivide the cubic cells, have a global index and a local index. As stated before, the Freudenthal subdivision scheme is used in this implementation. Thus, there are six tetrahedrons per cell, which are denoted by their local index $t_0, \ldots, t_5$ (see Figure 4.4). The global tetrahedron index is defined with respect to the set of active grid cells A, since only tetrahedrons in active grid cells are interesting for the implementation. For a tetrahedron with the local index t_i, located in the active cell a_j, the global index u_k is given by

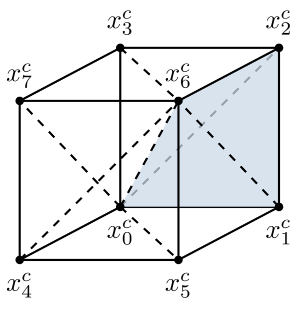
$$u_k = 6a_j + t_i \tag{4.3}$$

The four corner vertices of a tetrahedron are denoted by $X^t = \{x_0^t, \ldots, x_3^t\}$. Since the Freudenthal-subdivision does not add any vertices to the original lattice grid, those vertices are all effectively lattice points of the volume.

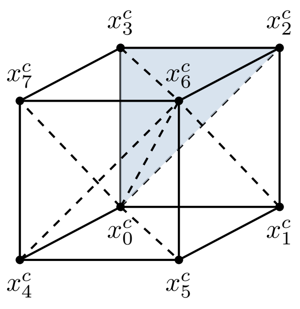
Furthermore, every tetrahedron has six edges, which are denoted by $E^t = \{e_0^t, \ldots, e_5^t\}$. The enumeration of the tetrahedron edges with respect to the tetrahedron corner vertices is illustrated in Figure 4.5.



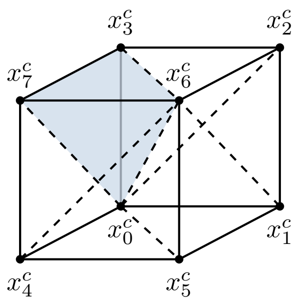
(a) Index t_0



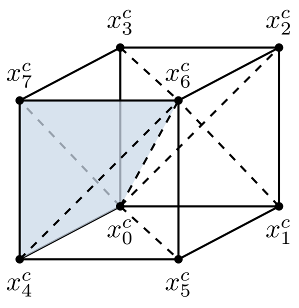
(b) Index t_1



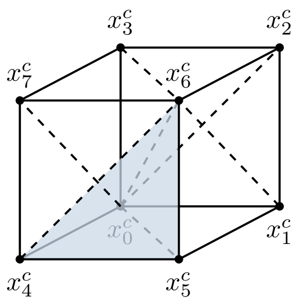
(c) Index t_2



(d) Index t_3



(e) Index t_4



(f) Index t_5

Figure 4.4: Illustration of the indexing for the tetrahedrons in every grid cell. The subdivision follows the Freudenthal scheme. The individual tetrahedrons are indexed as $t_0 \dots t_5$.

The most important entity for this implementation are the vertices extracted by the Marching Tetratedra algorithm as the explicit surface representation. They have both a global and a local index. In theory, each cell can be associated with 19 possible surface vertices. This straight-forward indexing scheme, however, associates most vertices with two cells (since most possible surface vertices are adjacent to two grid cells). In order to facilitate an implementation that deals with all grid cells in parallel, each possible surface point is only associated with one cell. Thus, seven possible surface vertices are uniquely associated with each cell and described by a local index $v_0, \dots, v_6$. Their location is illustrated in Figure 4.3b. The global index is defined with respect to the set of active grid cells A. The set of all vertices associated with active grid cells is W. For a vertex with local index v_i located in grid cell a_j, the global index w_k is

$$w_k = 7a_j + v_i \tag{4.4}$$

Finally, since not all of the vertices associated with an active cell are necessarily contributing to the surface, an index that enumerates active vertices only is needed. Let the set of all active vertices be $S \subset W$. The elements in S are denoted by s_i. Their ordering is the same as for the respective elements in W.

Table 4.1 sums up the notation used in this chapter.

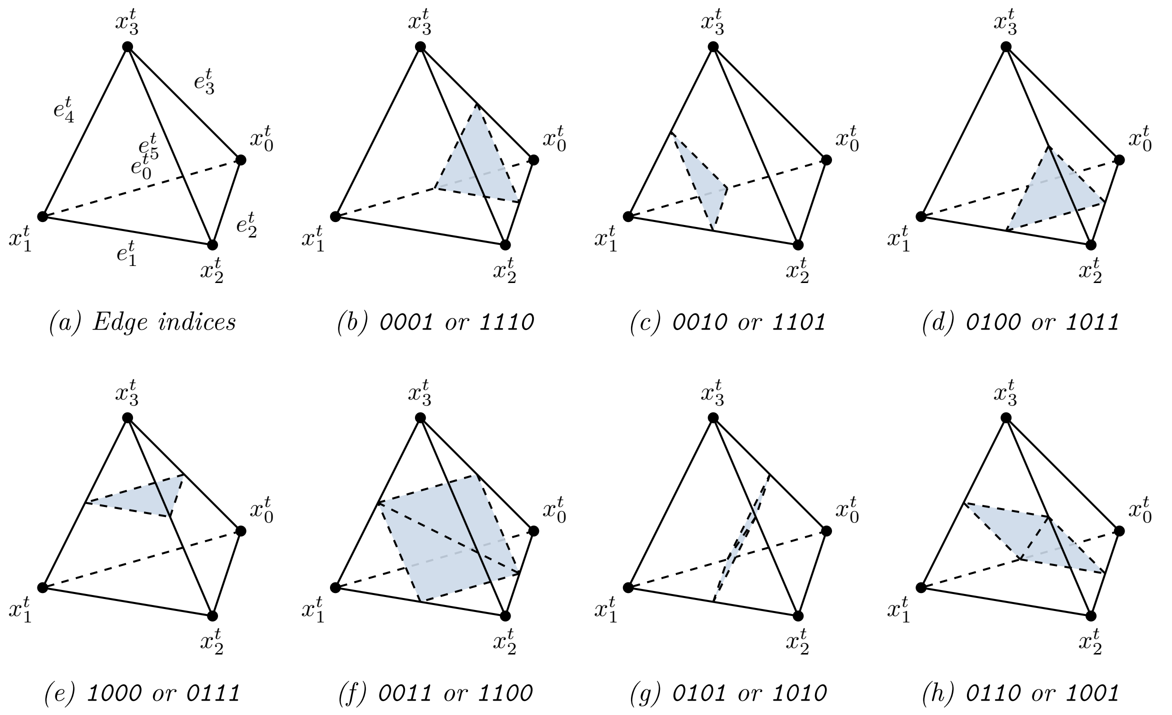


Figure 4.5: The edge indexing scheme and possible triangulations for one tetrahedron. The tetrahedron corner vertices are denoted by $x_0^t \dots x_3^t$. The tetrahedron edges ((a)) are defined with respect to the corner vertices and are denoted by $e_0^t \dots e_5^t$. All possible triangulations of one tetrahedron (without the cases `1111` *and* `0000`*) are shown in (b) to (h).*

4.2.2 Lookup-Tables

The most important lookup-tables used in the CUDA implementation of both the surface generation and the surface mapping are now listed and explained. The main purpose of this section is to simplify the description of the implementation later on. All the lookup tables are declared in constant device memory and, if viable, loaded into shared memory at the beginning of the respective kernel function.

- `cellVertexOffsets` defines the mapping from $x_0^c, \dots, x_7^c$ to the respective lattice points by providing an offset that has to be added to the cell origin (the left, bottom, back corner). These offsets are illustrated in Figure 4.3a.

```
__constant__ __device__ uint cellVertexOffsets[8][3] = {
    {0, 0, 0}, {1, 0, 0}, {1, 1, 0}, {0, 1, 0},
    {0, 0, 1}, {1, 0, 1}, {1, 1, 1}, {0, 1, 1}
};
```

Table 4.1: Summary of the notation used in Chapter 4.

Symbol	Meaning
x_i	The ith lattice point
c_i	The ith grid cell
a_i	The ith active grid cell
x_i^c	Local cell corner index ($i \in 0, \ldots, 7$)
t_i	Local tetrahedron index ($i \in 0, \ldots, 5$)
u_i	Global tetrahedron index (defined with respect to active grid cells)
x_i^t	The ith tetrahedron corner vertex ($i \in 0, \ldots, 3$)
e_i^t	The ith tetrahedron edge ($i \in 0, \ldots, 5$)
v_i	Local surface vertex index ($i \in 0, \ldots, 6$)
w_i	Global surface vertex index (defined with respect to active grid cells)
s_i	The ith active vertex

- As mentioned before, every active cell is associated with seven possible surface vertices $v_0 \ldots v_6$. `vertexIdxPerTetrahedronIdx` associates each of these vertices with one specific local tetrahedron index inside the cell. Every tuple represents one vertex by its index v_i and the edge index e_j^t on which the vertex lies in the respective tetrahedron. E.g., the two vertices v_0 and v_6 are associated with tetrahedron t_0, where they lie on the edges e_2^t and e_3^t. All other tetrahedrons are only related to one vertex. This is due to the fact that the Freudenthal scheme sudivides the cell into six tetrahedrons, but seven vertices are associated with each cell.

```
__constant__ __device__ int vertexIdxPerTetrahedronIdx[6][2][2] = {
        {{ 0,  2}, { 6,  3}}, /* Tetrahedron #0 */
        {{ 4,  2}, {-1, -1}}, /* Tetrahedron #1 */
        {{ 1,  2}, {-1, -1}}, /* Tetrahedron #2 */
        {{ 5,  2}, {-1, -1}}, /* Tetrahedron #3 */
        {{ 2,  2}, {-1, -1}}, /* Tetrahedron #4 */
        {{ 3,  2}, {-1, -1}}, /* Tetrahedron #5 */
};
```

- `tetrahedronEdgeFlags` relates the set of active corner vertices in a tetrahedron with the corresponding set of active tetrahedron edges. Active tetrahedron edges are edges on which a point of the final surface representation lies. Associating either `1` (vertex is inside the isosurface) or `0` (vertex is outside the isosurface) with the corner vertices yields a bit pattern $x_3^t x_2^t x_1^t x_3^t$ that can be interpreted as a number between zero and 15. The LUT contains the respective bit pattern $e_5^t e_4^t e_3^t e_2^t e_1^t e_0^t$, where `1` means that the respective edge is 'active' and `0` means that the respective edge is 'inactive'.

```
__constant__ __device__ unsigned char tetrahedronEdgeFlags[16] = {
    0x00, 0x0d, 0x13, 0x1e, 0x26, 0x2b, 0x35, 0x38,
    0x38, 0x35, 0x2b, 0x26, 0x1e, 0x13, 0x0d, 0x00
};
```

- `tetrahedronTriangles` defines the triangles inside a tetrahedron based on the tetrahedron flags obtained from the corner vertex activity flags. The triangles are defined with respect to the local tetrahedron edge index e_i^t.

```
__constant__ __device__ char tetrahedronTriangles[16][6] = {
    {-1, -1, -1, -1, -1, -1}, /* #0  */
    { 0,  3,  2, -1, -1, -1}, /* #1  */
    { 0,  1,  4, -1, -1, -1}, /* #2  */
    { 1,  4,  2,  2,  4,  3}, /* #3  */
    { 1,  2,  5, -1, -1, -1}, /* #4  */
    { 0,  3,  5,  0,  5,  1}, /* #5  */
    { 0,  2,  5,  0,  5,  4}, /* #6  */
    { 5,  4,  3, -1, -1, -1}, /* #7  */
    { 3,  4,  5, -1, -1, -1}, /* #8  */
    { 4,  5,  0,  5,  2,  0}, /* #9  */
    { 1,  5,  0,  5,  3,  0}, /* #10 */
    { 5,  2,  1, -1, -1, -1}, /* #11 */
    { 3,  4,  2,  2,  4,  1}, /* #12 */
    { 4,  1,  0, -1, -1, -1}, /* #13 */
    { 2,  3,  0, -1, -1, -1}, /* #14 */
    {-1, -1, -1, -1, -1, -1}  /* #15 */
};
```

- `vertexNeighbouringTetrahedrons` contains all neighbouring tetrahedrons of a vertex v_i, defined by a global lattice offset (first three numbers) and a local tetrahedron index $t_0 \dots t_5$ (fourth number). This mapping is necessary to obtain neighboring vertices of v_i, which could be associated with adjacent cells. `99` indicates that there is no neighbor with the specified tetrahedron index.

```
__constant__ __device__ int vertexNeighbouringTetrahedrons[7][6][4] = {
        {{ 0,  0,  0,  0}, { 0,  0,  0,  1}, { 0, -1,  0,  2},
         { 0, -1, -1,  3}, { 0, -1, -1,  4}, { 0,  0, -1,  5}}, // v0
        {{-1,  0, -1,  0}, {-1,  0,  0,  1}, { 0,  0,  0,  2},
         { 0,  0,  0,  3}, { 0,  0, -1,  4}, {-1,  0, -1,  5}}, // v1
        {{-1,  0,  0,  0}, {-1, -1,  0,  1}, {-1, -1,  0,  2},
         { 0, -1,  0,  3}, { 0,  0,  0,  4}, { 0,  0,  0,  5}}, // v2
        {{ 0,  0,  0,  0}, {99, 99, 99,  1}, { 0, -1,  0,  2},
         { 0, -1,  0,  3}, {99, 99, 99,  4}, { 0,  0,  0,  5}}, // v3
        {{99, 99, 99,  0}, { 0,  0,  0,  1}, { 0,  0,  0,  2},
         {99, 99, 99,  3}, { 0,  0, -1,  4}, { 0,  0, -1,  5}}, // v4
        {{-1,  0,  0,  0}, {-1,  0,  0,  1}, {99, 99, 99,  2},
         { 0,  0,  0,  3}, { 0,  0,  0,  4}, {99, 99, 99,  5}}, // v5
        {{ 0,  0,  0,  0}, { 0,  0,  0,  1}, { 0,  0,  0,  2},
         { 0,  0,  0,  3}, { 0,  0,  0,  4}, { 0,  0,  0,  5}}  // v6
};
```

- As stated before, the vertices v_i are defined on tetrahedron edges, which are denoted by the indices $e_0^t \dots e_5^t$. Therefore, one vertex is associated with more than one edge index, since the edge it is laying on has different edge indices in the different adjacent tetrahedrons. `vertexNeighbouringTetrahedronsOwnEdgeIdx` contains the edge index that is associated with every vertex $v_0 \dots v_6$ inside its adjacent tetrahedrons. `-1` indicates undefined values, since not all of the edges have the same amount of adjacent tetrahedrons.

```
__constant__ __device__ int
    vertexNeighbouringTetrahedronsOwnEdgeIdx[7][6] = {
         { 2,  0,  1,  5,  4,  1}, // v0
         { 4,  1,  2,  0,  1,  5}, // v1
         { 1,  5,  4,  1,  2,  0}, // v2
         { 0, -1,  5,  4, -1,  2}, // v3
         {-1,  2,  0, -1,  5,  4}, // v4
         { 5,  4, -1,  2,  0, -1}, // v5
         { 3,  3,  3,  3,  3,  3}  // v6
};
```

- `edgeConnectionsByTetrahedronFlags` describes connections inside the tetrahedron for every tetrahedron edge based on the tetrahedron edge flags. E.g. if the tetrahedron flags are `0111` then for the vertex located on edge e_3^t the LUT yields `0x30` or `110000`, meaning that this vertex has a connection to the vertices laying on the edges e_4^t and e_5^t in the same tetrahedron.

```
__constant__ __device__ unsigned char
    edgeConnectionsByTetrahedronFlags[16][6] = {
/* edges  #0    #1    #2    #3    #4    #5                        */
         {0x00, 0x00, 0x00, 0x00, 0x00, 0x00}, /* vertices active 0000 */
         {0x0C, 0x00, 0x09, 0x05, 0x00, 0x00}, /* vertices active 0001 */
         {0x12, 0x11, 0x00, 0x00, 0x03, 0x00}, /* vertices active 0010 */
         {0x00, 0x14, 0x1A, 0x14, 0x0E, 0x00}, /* vertices active 0011 */
         {0x00, 0x24, 0x22, 0x00, 0x00, 0x06}, /* vertices active 0100 */
         {0x2A, 0x21, 0x00, 0x21, 0x00, 0x0B}, /* vertices active 0101 */
         {0x34, 0x00, 0x21, 0x00, 0x21, 0x15}, /* vertices active 0110 */
         {0x00, 0x00, 0x00, 0x30, 0x28, 0x18}, /* vertices active 0111 */
         {0x00, 0x00, 0x00, 0x30, 0x28, 0x18}, /* vertices active 1000 */
         {0x34, 0x00, 0x21, 0x00, 0x21, 0x15}, /* vertices active 1001 */
         {0x2A, 0x21, 0x00, 0x21, 0x00, 0x0B}, /* vertices active 1010 */
         {0x00, 0x24, 0x22, 0x00, 0x00, 0x06}, /* vertices active 1011 */
         {0x00, 0x14, 0x1A, 0x14, 0x0E, 0x00}, /* vertices active 1100 */
         {0x12, 0x11, 0x00, 0x00, 0x03, 0x00}, /* vertices active 1101 */
         {0x0C, 0x00, 0x09, 0x05, 0x00, 0x00}, /* vertices active 1110 */
         {0x00, 0x00, 0x00, 0x00, 0x00, 0x00}  /* vertices active 1111 */
};
```

- The final output for the computation of the vertex connectivity is an array of the size 18*`activeVertexCnt`, where each of the 18 associated indices either points to another vertex or is undefined. In order to make the computation thread-safe, it is necessary to define unique offsets for all possible neighbors. `tetrahedronToNeighbourIdx` defines the neighbor index for all possible connected edges for all vertices. The layout is $[v_i]$[adjacent tetrahedron idx][edgeIdx]. `-1` indicates that there is no connection possible. E.g. for a vertex with the local vertex index v_0 in active cell a_i, if there is a connection to the vertex laying in the adjacent tetrahedron two, on edge one, then the index of the neighbor vertex is stored at $18{*}a_i + 5$.

```
__constant__ __device__ int tetrahedronToNeighbourIdx[7][6][6] = {

        {{ 0,  1, -1,  2,  3,  4}, {-1,  5,  6,  2,  4,  7},
         { 8, -1,  9, 10,  1,  0}, {11, 12, 13, 14, 15, -1},
         {13,  9, 16, 14, -1,  8}, {12, -1, 15, 17,  6,  5}}, // #v0

        {{ 0,  1,  2,  3, -1,  4}, { 5, -1,  6,  7,  8,  9},
         {10, 11, -1, 12, 13, 14}, {-1,  9,  8, 12, 14, 15},
         { 4, -1,  1, 16, 11, 10}, {17,  5,  0,  3,  6, -1}}, // #v1

        {{ 0, -1,  1,  2,  3,  4}, { 5,  6,  7,  8,  9, -1},
         { 7,  1, 10,  8, -1,  0}, { 6, -1,  9, 11, 12, 13},
         { 4,  3, -1, 14, 15, 16}, {-1, 13, 12, 14, 16, 17}}, // #v2

        {{-1,  0,  1,  2,  3,  4}, {-1, -1, -1, -1, -1, -1},
         { 5,  1,  6,  7,  0, -1}, { 6,  8,  9,  7, -1, 10},
         {-1, -1, -1, -1, -1, -1}, { 8, 10, -1,  2, 11,  3}}, // #v3

        {{-1, -1, -1, -1, -1, -1}, { 0,  1, -1,  2,  3,  4},
         {-1,  5,  6,  2,  4,  7}, {-1, -1, -1, -1, -1, -1},
         { 8,  6,  9,  10, 5, -1}, { 9,  0, 11, 10, -1,  1}}, // #v4

        {{ 0,  1,  2,  3,  4, -1}, { 2,  5,  6,  3, -1,  7},
         {-1, -1, -1, -1, -1, -1}, { 5,  7, -1,  8,  9, 10},
         {-1,  4,  1,  8, 10, 11}, {-1, -1, -1, -1, -1, -1}}, // #v5

        {{ 0,  1,  2, -1,  3,  4}, { 2,  5,  6, -1,  4,  7},
         { 6,  8,  9, -1,  7, 10}, { 9, 11, 12, -1, 10, 13},
         {12, 14, 15, -1, 13, 16}, {15, 17,  0, -1, 16,  3}}, // #v6
};
```

4.3 CUDA Implementation of the Surface Generation

The surface generation includes the computation of the vertex positions, the computation of triangles based on these vertices, obtaining connectivity information for each vertex, computing vertex normals for later rendering, and, finally, the computation of texture coordinates.

4.3.1 Computing Vertex Positions

The goal of this step is to compute the vertex positions for the explicit surface representation based on the Gaussian volume distribution described in Section 3.5. No triangle indices or connectivity information is computed, yet. The outcome of this step is a `float`-array containing all the vertex positions.

Based on the indexing scheme introduced before, the algorithm for computing the vertex positions can be outlined as follows: First all cells that are crossed by the isosurface are marked as *active*. All active cells now have two indices, the global grid index c, which enumerates all grid cells in the volume, and the active-cells-index a, which enumerates only active cells.

It is necessary to keep track of the global grid cell index, because it is needed later on when computing the connectivity between vertices. A mapping function is defined to be able to switch between the global grid index and the active-cells-index. This mapping function can be seen as 'compacting' the grid cell array to only contain the active grid cells. Next, all the active grid cells are processed to determine which of the vertices associated with them contribute to the final surface representation. During this step, the actual vertex position is computed and written to an array. Finally, this array is compacted similar to the cell array before, which yields the final array containing the vertex positions.

The first step contains flagging the grid cells depending on whether the isosurface crosses them or not. To this end, the volume is sampled at all eight cell corners and every cell corner is flagged either with '`1`' (the value of the sample is smaller than the isovalue) or with '`0`' (the value of the sample is higher than the isovalue). Here, the LUT **`cellVertexOffsets`** is used to get the offsets. This yields a bit pattern, consisting of eight bits (the order of the bits being the same as the indices of the cell corners). This eight bit number can be reduced to a one bit cell flag by computing the modulo with 255. If either all flags are `1` or all flags are `0`, the modulo operator yields `0`. Otherwise the flag is set to `1`. The result is an array of the size of all grid cells with zeros at the indices of inactive cells and ones at the indices of active cells. The pseudo-code below (Listing 4.1) sums up the step.

Listing 4.1: CUDA kernel that flags active grid cells for the Marching Tetrahedra Algorithm.

```
__global__ void FindActiveGridCells_D(...) {

    /* Use local registers for vertex offsets */
    const float cellVertexOffsets[8][3] = {
        {0, 0, 0}, {1, 0, 0}, {1, 1, 0}, {0, 1, 0},
        {0, 0, 1}, {1, 0, 1}, {1, 1, 1}, {0, 1, 1}
    };

    /* Sample volume at the cell origin */
    // float volSample = ...

    /* Init cell flags */
    unsigned char cellFlags = uint(volSample <= isoval);

    /* Loop through all cell corners */
    for (int v = 1; v < 8; ++v) {

        uint3 pos = cellorg + cellVertexOffsets[v];

        /* Sample volume at pos */
        // volSample = ...

        /* Update cell flags */
        cellFlags |= uint(volSample <= isoval) * (1 << v);
    }

    /* Reduce vertex states to one cell state and write to global
     * device memory */
    activeCellFlag_D[cellIdx] = min(cellFlags % 255, 1);
}
```

Next, an index is obtained that enumerates all active grid cells. The goal here is to obtain a mapping between the indices c and a. One way to achieve this is to compute the prefix sum using the `THRUST` library[2]. The function `thrust::exclusive_scan` takes an array as argument and writes the prefix sum for every element in an output array. Let $D = \{d_0, \ldots, d_n\}$ be an array, then the prefix sum of an element d_j is defined as $\sum_{i=0}^{j-1} d_i$. The use of the prefix sum computation for compacting arrays is inspired by the marching cubes implementation provided within the CUDA Computing SDK by NVIDIA for CUDA 4.2 [3]. Below is illustrated how the prefix sum computation in an array with cell flags yields a alternative index enumerating only active cells.

```
Global cell index  : 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 ...
Cell flags         : 0  0  0  1  1  0  0  0  1  0  1  1  1  1  1  0  1  0 ...
Prefix sum         : 0  0  0  0  1  2  2  2  2  3  3  4  5  6  7  8  8  9 ...
Active cells index : -  -  -  0  1  -  -  -  2  -  3  4  5  6  7  -  8  - ...
```

This effectively yields a mapping of the two indices in both directions. It is, however, not bijective, since set of active-cells A has fewer elements then the set of all cells C. This compacted form is crucial for the following steps, since it reduces the input data a lot (about 97-99% in experiments done for this work) and enables more efficient implementations.

The array containing the prefix sum can be used to determine the overall number of active grid cells `activeCellCnt`. This is done by adding the last element of the prefix sum array to the last element of the flag array (which is either `1` or `0`).

Since the active cells were compacted to a concurrent array in the last step, the following computations can be reduced to active cells only. The next task is to find all actual surface vertices s amongst the seven vertices v associated with each active cell. To this end, every CUDA kernel processes one tetrahedron (i.e 6*`activeCellCnt` in total). The following pseudo-code (Listing 4.2) sums up the computation of the vertex positions.

Listing 4.2: CUDA kernel that computes vertex positions in the Marching Tetrahedra algorithm.

```
__global__ void CalcVertexPositions_D(...) {

    /* Load lookup-tables to shared memory */
    // ...

    __syncthreads();

    /* Get bitmap to classify the tetrahedron */
    // unsigned char tetrahedronFlags = ...

    /* Loop through the two possible vertices $v_i$ the tetrahedron is
     * associated with */
```

[2]https://developer.nvidia.com/thrust
[3]https://developer.nvidia.com/cuda-toolkit-42-archive

```
13      for (int i = 0; i < 2; ++i) {
14          if (vertexIdxPerTetrahedronIdx[localTetraIdx][i][0] < 0) {
15              continue;
16          }
17          /* Get local vertex index $v_i$ */
18          uint localVtxIdx = VertexIdxPerTetrahedronIdx[$t_k$][i][0];
19          /* Get the edge index $e^t_j$ associated with this vertex */
20          uint edgeIdx = VertexIdxPerTetrahedronIdx_S[localTetraIdx][i][1];
21          /* Check whether the edge is active in this tetrahedron */
22          if (tetrahedronEdgeFlags_S[tetrahedronFlags] & (1 <<
                static_cast<unsigned char>(edgeIdx))) {
23
24              /* Interpolate position of the vertex inbetween v0 and v1 */
25              // vertex = ...
26
27              /* Save vertex position and activity flag to global device
                    memory */
28              activeVertexIdx_D[activeCubeIdx*7+localVtxIdx] = 1;
29              activeVertexPos_D[activeCubeIdx*7+localVtxIdx] = vertex;
30          }
31      }
32  }
```

The kernel loops through all the vertices v_i associated with the tetrahedron represented by the thread. This association information can be obtained from the LUT `vertexIdxPerTetrahedronIdx`, based on the local tetrahedron index t_j. The LUT also allows to relate the vertex index v_i with the local tetrahedron edge index e^t_k. It is then tested whether this edge is active or not (line 15). If the edge is 'active' the exact position is computed by using linear interpolation between the two respective corner vertices of the tetrahedron. Additionally an 'active' flag is set for the particular vertex, which can subsequently be used to compact the vertex positions and yield the final array containing the positions of all active surface vertices s.

4.3.2 Computing Triangle Indices

The triangles of the surface mesh are computed based on the vertex positions obtained in the previous step. In the resulting array, all triangles are represented by triples of vertex indices. The calculation can be outlined as follows: First, the number of vertex indices necessary to describe the triangles in each tetrahedron is computed and stored. Processing this array with a prefix sum computation yields vertex index offsets for all tetrahedrons. This information is finally used to to obtain and store the actual triangle indices.

As an initial step, the number of vertices for all tetrahedrons is computed. Since each active tetrahedron contains either one or two triangles, the number of vertices is either three or six. It can be obtained based on the according LUT `tetrahedronVertexCount`. The number of vertices per tetrahedron is written to the device array `verticesPerTetrahedron_D`.

Similar to the steps described before, a prefix sum computation is now applied to the device array `vertexIdxOffs_D`. This yields a vertex index offset for all tetrahedrons containing triangles, similar to the following example:

```
Tetrahedron index         : 0  1  2  3  4  5  6  7  8  9 10 11 12 ...
Vertices per tetrahedron  : 0  6  3  3  6  0  0  0  6  6  3  0  0 ...
Vertex index offsets      : 0  0  6  9 12 18 18 18 18 24 30 33 33 ...
```

The vertex index offset is stored in the device array `tetrahedronVertexIdxOffs_D`.

The information obtained in the two previous steps is now used to compute the actual triangle index array. The pseudo-code for this step can be found in Listing 4.3.

Listing 4.3: CUDA kernel that computes the triangles indices in the Marching Tetrahedra algorithm.

```
__global__ void GetTrianglesIdx_D(...) {

    /* Load lookup tables to shared memory */
    // ...

    __syncthreads();

    /* Get bitmap to classify the tetrahedron */
    // unsigned char tetrahedronFlags = ...

    /* Skip inactive tetrahedrons */
    if (tetrahedronFlags == 0x00 || tetrahedronFlags == 0x0F) {
        return;
    }

    /* Shared memory array to hold the global vertex index $w_i$
     * associated with every active edge */
    __shared__ uint edgeVertexIdx[6 * GET_TRIANGLE_IDX_BLOCKSIZE];

    /* Test all edges of the current tetrahedron */
    for (int edgeIndex = 0; edgeIndex < 6; edgeIndex++) {
        if (tetrahedronEdgeFlags[tetrahedronFlags] & (1 <<
            static_cast<unsigned char>(edgeIndex)))  {
            /* Store $w_i$ to shared memory */
            // edgeVertexIdx[threadIdx.x * 6 + edgeIndex] = ...
        }
    }

    __syncthreads();

    /* Transform indices $w_i$ to indices $s_i$ and store them to
     * represent triangles */
    for (int triangleIndex = 0; triangleIndex < 2; triangleIndex++) {
        if (tetrahedronTriangles[tetrahedronFlags][3 * triangleIndex] >=
            0) {
            for (int cornerIndex = 0; cornerIndex < 3; cornerIndex++) {
                /* Get local edge index associated with this vertex */
                // localEdgeIdx = ...
37                 /* Compute global edge index */
38                 edge = threadIdx.x * 6 + localEdgeIdx;
39                 /* Get global vertex index w */
40                 w = edgeVertexIdx[edge];
41                 /* Compute vertex index offset in the triangle idx
42                  * array */
43                 uint vertexOffset =
                       vertexOffsets_D[id]+3*triangleIndex+cornerIndex;
44                 /* Store vertex index to triangle index array */
45                 triangleVertexIdx_D[vertexOffset] = vertexMapInv_D[w];
46             }
47         }
48     }
49 }
```

A kernel is invoked for every tetrahedron in every active cell, therefore, the number of threads is `6*activeCellCnt`. If the tetrahedron is entirely outside or entirely inside the surface (either `flags == 0x00`, or `flags == 0x0F`), the thread is terminated. If the tetrahedron represented by a thread contains active edges, the respective global vertex index w is computed for each active edge and stored in shared memory. Note that all edges have several adjacent tetrahedrons and, therefore, each of the indices is written by several threads. The precomputed indices w are needed in the next step, where the triangle triples are written. This is done by iterating through all triangles of the tetrahedron the thread is representing. For each triangle vertex, the local edge index is obtained from the LUT `tetrahedronTriangles`. The global edge index is computed and used to obtain the vertex index w from shared memory. In line 45, the global vertex index w is transformed to a surface vertex index s using the pre-defined mapping relation (stored in `vertexMapInv_D`) and the vertex index s is written to the device array `triangleIdx_D` using the index offset computed in the previous step. The resulting array contains all triangles defined by their respective triples of vertex indices s.

4.3.3 Computing Connectivity Information

Each vertex can have up to 18 neighbors. The resulting index array, therefore, has the size 18*`activeVertexCnt`, with `-1` denoting invalid indices. The pseudo-code in Listing 4.4 outlines the CUDA kernel used for the computation.

Listing 4.4: CUDA kernel that computes the vertex connectivity in a triangle mesh.

```
__global__ void ComputeVertexConnectivity_D(...) {

    /* Load lookup tables to shared memory */
    // ...

    __syncthreads();

    /* Loop through all adjacent tetrahedrons */
    for (int t = 0; t < 6; ++t) {
        /* Check whether the adjacent tetrahedron is valid */
11          if (VertexNeighbouringTetrahedrons_S[localVtxIdx][t][0] == 99)
                return;
12
13          /* Get tetrahedron flags of the adjacent tetrahedron */
14          // flags = ...
15
16          /* Edge index associated with this vertex in the adjacent
17           * tetrahedron */
18          ownEdgeIdx = VertexNeighbouringTetrahedronsOwnEdgeIdx[v][i];
19
20          /* Look up connections of the vertex in the adjacent
21           * tetrehadron */
22          connectionFlags =
                edgeConnectionsByTretrahedronFlags[flag][ownEdgeIdx];
23
24          /* Loop through possible connections */
25          for (int j = 0; j < 6; ++j) {
26              if (connectionFlags & (1 << unsigned char(j))) {
27                  /* Obtain index s_k of the connected vertex */
28                  // int vertexIdx = ...
29
30                  /* Store to global device memory */
31                  uint nIdx = 18*THREADIDX+
32                      tetrahedronToNeighbourIdx[localVtxIdx][i][j];
33                  vertexNeighbours_D[nIdx] = vertexIdx;
34              }
35          }
36      }
37  }
```

A kernel is invoked for every active vertex. It loops through all tetrahedrons adjacent to the edge the current vertex lays on. For each of these adjacent tetrahedrons the tetrahedron flags are computed and stored (line 9). Next, the index of the edge the vertex lays on with respect to the adjacent tetrahedron is obtained from the LUT `vertexNeighbouringTetrahedronsOwnEdgeIdx` (line 10). The kernel loops through all connections that the vertex has in this adjacent tetrahedron. The subject of the computations in lines 24-35 is basically to transform the local edge index of the neighboring vertices the adjacent tetrahedron into a surface vertex index s that can then be stored in the neighbor index array.

4.3.4 Computing Vertex Normals

In the following, it is described how the normals can be computed using the neighbor connectivity computed in the previous step. Listing 4.5 shows the necessary steps.

Listing 4.5: CUDA kernel that computes the vertex normals in a triangle mesh.

```
1  __global__ void ComputeVertexNormals_D(...) {
2
3      /* Load lookup tables to shared memory */
4      // ...
5
```

```
    __syncthreads();

    /* Get position from global device memory */
    float3 pos = vertexPos_D[THREADIDX];

    /* Loop through all adjacent tetrahedrons */
    for (int tetrahedronIdx = 0; tetrahedronIdx < 6; ++tetrahedronIdx) {

        /* Check whether =? 99 (is tetrahedron valid?) */
        if (VertexNeighbouringTetrahedrons[v][tetrahedronIdx][0] == 99) {
            continue;
        }

        /* Get origin of the cell containing the adjacent tetrahedron */
        // int3 cellOrgTemp = ...

        /* Get tetrahedron flags of the adjacent tetrahedron */
        // flags = ...

        /* Edge index of this vertex in the adjacent tetrahedron */
        // ownEdgeIdx = ...

        /* Loop both possible triangles of this tetrahedron */
        for(int triangleIdx = 0; triangleIdx < 2; ++triangleIdx) {
            if(/* triangleIdx not valid */) {
                continue;
            }

            for(int vtx = 0; vtx < 3; vtx++) {

                if(/* ownEdgeIdx takes part in the triangle */) {

                    /* Get the vertex indices of the two connected
                       vertices */
                    // vertexIdx0 = ...
                    // vertexIdx1 = ...

                    /* Get positions of the two connected vertices */
                    // pos0 = ...
                    // pos1 = ...

                    /* Get displacement vectors to this vertices */
                    float3 vec0 = pos0 - pos;
                    float3 vec1 = pos1 - pos;

                    /* Update vertex normal */
                    normal += cross(vec0, vec1);
                }
            }
        }
    }

    /* Normalize and write to global device memory */
    normal = normalize(normal);
    normals_D[THREADIDX] = normal;
```

```
}
```

A kernel is invoked for every surface vertex s_i. The kernel program loops through all adjacent tetrahedrons, similar to the connectivity computation described before. For every adjacent tetrahedron, the up to two triangles are used to compute a cross product between the two spanning vectors. This result of this cross product is accumulated for all triangles adjacent to the vertex. Finally, the summed up cross product is normalized and written to the output buffer in global device memory.

4.3.5 Computing Texture Coordinates

Computing texture coordinates is crucial for the comparative visualization, since two potential values are to be compared. The texture coordinates can be obtained trivially if the world space vertex positions and the location and dimensions of the texture are known. Let s be the world space position of a surface point. Furthermore, let the texture space be defined by an origin x_{org}, a spacing $\mu = (\mu_x, \mu_y, \mu_z)$, and the size $S = (S_x, S_y, S_z)$. The maximum coordinates x_{max} for the texture in each dimension is given by

$$x_{max} = x_{org} + \mu\,(S - 1) \tag{4.5}$$

Using the maximum coordinate x_{max}, the texture coordinates $tc = (tc_x, tc_y, tc_z)$ can be obtained by

$$tc = \frac{s - x_{org}}{x_{max} - x_{org}} \tag{4.6}$$

4.4 CUDA Implementation of the Surface Mapping Algorithm

The subject of this computations is to update vertex positions according to the external and internal forces. In the following, the two necessary pre-processing steps, i.e. initializing the external forces' scale factor and pre-computing the gradient at all lattice positions, are outlined. Then the computation of the updated vertex position is described. Additionally, the possibility of executing several iterations in one kernel evocation is discussed.

First, the gradient of all lattice points is pre-computed using central differences. This seams to be an overhead, since the computation is done for every lattice point and, as mentioned before, about 97% of the grid cells are not even active. However, this pre-computation is done only once per mapping and saves a number of global memory fetches in each iteration step. When the gradient is obtained during the computation of the mapping iterations, the interpolation of the pre-computed gradient only requires to fetch 8*3=24 float values from global device memory (assuming trilinear interpolation) and three interpolations (one for each component of the gradient), whereas the direct computation requires 6*8=48 fetches from global device memory and six interpolations. Additionally, the pre-computation is much faster than even a single iteration (see Chapter 5), therefore, the one-time effort of computing the gradient for all lattice points pays off in the total computation cost.

The combination of the Gaussian volume and the distance field (see Section 3.5.3) can be implemented by modifying the pre-computation of the gradient before the mapping in a way that involves the distance field. The value of the distance field is sampled at every lattice point and when its beyond a certain threshold the gradient is computed based on the distance field. If the distance sample is below the threshold, the gradient is instead computed based on the Gaussian volume. Additionally, when using the distance field, the sign of the gradient is changed to match up with the one defined by the Gaussian volume. This is necessary, since the gradient based on the distance field and the gradient based on the Gaussian volume point in opposite directions when being outside the molecular surface. Due to time constraints, the computation of the distance field was done in a non-optimized brute-force way. There are, however, possibilities to speed up the computation, e.g. by using a grid based neighborhood search similar to the one used in the 'particles' example provided by NVIDIA in the CUDA SDK. The pre-computed gradient is stored in the device array `gradient_D`.

As a second initialization step, the scale factor for the external forces is calculated depending on whether the vertex on the source surface starts outside or inside the target surface. If the starting position of a vertex is outside the target surface the scale factor is initialized with 1, since the gradient points towards the target surface. If the vertex starts out inside the target surface, the initial gradient points away from the target surface and the scale factor is, therefore, initialized with -1 to make the gradient point towards the target surface. The initial scale factors are stored in the device array `externalForcesScl_D`.

Below (Listing 4.6) is a simplified version of the algorithm implemented in the kernel to compute one iteration.

Listing 4.6: CUDA kernel that computes one iteration of the surface mapping algorithm.

```
__global__ void UpdateVertexPosition_D(...) {

    /* Retrieve input from global device memory */
    externalForcesScl = vertexExternalForcesScl_D[THREADIDX];
    pos = vertexPos_D[THREADIDX];

    /* Sample target volume at current position pos */
    // ...

    /* If surface has been crossed: switch sign of external forces
       and scale by 0.5 */
    bool negative = externalForcesScl < 0;
    bool outside = sampleDens <= isoval;
    int switchSign = int((negative && outside)||(!negative && !outside));
    externalForcesScl = externalForcesScl*((1-switchSign) - switchSign);
    externalForcesScl *= (1.0*(1-switchSign) + 0.5*(switchSign));

    /* Sample gradient grad at current position pos */
    // grad = ...

    /* Calculate external force $\color{spring_green}{\vec{f}_{ext}}$ */
    externalForce =  externalForcesScl*normalize(grad);

    /* Calculate discrete Laplacian */
25      for (int i = 0; i < 18; ++i) {
26          int isIdxValid = int(nIdx[i] >= 0); /* 1 if nidx[i] != -1 */
27          int tmpIdx = isIdxValid*nIdx[i];    /* Map -1 to 0 */
28          posNeighbour = vertexPos_D[tmpIdx];
29          displ = (posNeighbour - pos);
30          laplacian += displ*isIdxValid;
31          activeNeighbourCnt += 1.0f*isIdxValid;
32      }
33      laplacian /= activeNeighbourCnt;      /* Normalize */
34
35      /* Save to global device memory */
36      laplacian_D[idx] = laplacian;
37      __syncthreads();
38
39      /* Calculate laplacian^2 */
40      for(int i = 0; i < 18; ++i) {
41          int isIdxValid = int(nIdx[i] >= 0);
42          int tmpIdx = isIdxValid*nIdx[i];
43          laplacian2 += (laplacian_D[tmpIdx ]- laplacian)*isIdxValid;
44      }
45      laplacian2 /= activeNeighbourCnt;
46
47      /* Compute internal force $\color{spring_green}{\vec{f}_{int}}$ */
48      float3 internalForce = (1.0 - rigitiy)*laplacian -
            rigitiy*laplacian2;
49
50      /* Project internal force */
51      float3 normal = normalize(grad);
52      internalForce = internalForce - dot(internalForce, normal)*normal;
53
54      /* Compute final position */
55      pos = pos +
56          sclAll*(externalWeight*externalForce
                +(1.0-externalWeight)*internalForce);
57
58      /* Write position and scale factor back to global device memory */
59      vertexPos_D[THREADIDX] = pos;
60      vertexExternalForcesScl_D[THREADIDX] = externalForcesScl;
61  }
```

When invoking the kernel, both the external force scale factor and the current position are loaded from global device memory to local registers to avoid high latency when this information is accessed again (line 3). The next step is to get a sample of the target volume and use it to determine the necessary sign switching and/or rescaling for the external force. This can be done without conditional statements, and – while avoiding additional branching – without introducing a lot of additional computations. The integer `switchSign` denotes whether the vertex has crossed the isosurface with respect to the previous position. This can be determined by comparing the current volume sample with the current sign of the scaling factor in the following way:

`negative`	`outside`	`switchSign`
`false`	`false`	`1`
`false`	`true`	`0`
`true`	`false`	`0`
`true`	`true`	`1`

As a result, `switchSign` is 1 if the surface has been crossed. The `switchSign` flag is then used to yield the scaling (line 16) and the new sign of the external force (line 15). The gradient at the current position is then sampled and scaled with the factors determined before to yield the external force vector f. After that, the scaled force is used to compute the new vertex position with respect to external forces only. In lines 24-33, the discrete Laplacian is computed according to Equation 3.16 (page 16). Since not all of the neighbor indices saved in the array are valid, the non valid indices are mapped to `0` and their contribution to the Laplacian is weighted with zero. This way, additional branching is avoided while, however, additional memory fetches are performed. Conditional statements would worsen the overall performance in this case, since they would be inside a loop. Therefore, the additional access to global device memory is justified. The conditional behavior is implemented in a similar way as before, using a flag as weighting function. The discrete Laplacian obtained by this step is then normalized and stored to global device memory. This additional write access to device memory cannot be avoided since the Δ^2 term of a vertex can potentially depend on the Laplacian of every other vertex. Subsequently, the Δ^2 term is computed in the same manner as the Laplacian. The internal force can now be computed by combining the Laplacian and the Δ^2 term (line 48). Finally, the internal force is projected to be tangential to the volume as described in Section 3.5. The new vertex position is computed by combining the external and the internal forces with the appropriate weighting and both the new position and the external forces scaling factor are written back to global device memory.

So far, only one iteration of the surface mapping has been executed per kernel invocation. A modified CUDA kernel that computes several iteration steps in a row can be found in the following Listing 4.7 (only statements that differ from the previous descriptions are contained).

Listing 4.7: CUDA kernel that computes multiple iterations of the surface mapping algorithm.

```
__global__ void UpdateVertexPositionMultiIt_D(...) {

    /* Retrieve input from global device memory */
    // ...

    /* Retrieve neighbour indices from global device memory */
    // nIdx[i] = nIdx_D[THREADIDX*18+i] ...

    for (int i = 0; i < UPDATE_VTX_POS_ITERATIONS_PER_KERNEL; ++i) {

        /* Check whether the displacement constraint has been reached */
        // ...

        /* Calc forces and update position */
        // ...

        /* Write position back to global device memory */
        vertexPos_D[THREADIDX] = pos;

        /* Syncronize all threads */
        __syncthreads();
    }

    /* Scale factor back to global device memory */
    vertexExternalForcesScl_D[THREADIDX] = externalForcesScl;
}
```

There are several changes with respect to the other kernel. First, the neighbor indices are retrieved from global device memory and written to local registers. Since the indices are static, they have to be acquired only once per vertex. Writing them to registers, therefore, is necessary to prevent unnecessary memory latency. Second, the vertex positions now have to be written back to global device memory after every iteration. This does not introduce additional writing operations, since now, several iterations are done in one kernel. Writing the positions is necessary, because the computation of the internal forces depends on the updated vertex positions. Third, after the writing operation, all threads have to be synchronized to make sure that all writing operations are completed.

There are two advantages to this approach. First, some time can be saved because there are lesser kernels to be invoked for the same result as before. The second advantage is that some writing operations to global device memory can be avoided, since the external forces scale factor is only used by one vertex and, therefore, does not need to be written back to global memory after each iteration step. The disadvantage to this approach is the additional thread synchronizing, which could introduce latency. However, when choosing the maximum number of iterations per kernel carefully, the approach is still faster, despite the thread synchronizing (see Chapter 5).

The final implementation for this work uses the CUDA kernel performing multiple iterations during one evocation. Both linear and cubic interpolation methods have been implemented for later comparison.

4.5 CUDA Implementation of the Metric

In this section, the computation of the metric according to Section 3.6 is described. The procedure consists of several steps. First, all corrupt triangles are found and flagged. Then, the potential difference and the potential sign flag are computed for every vertex. Subsequently, the area of all non-corrupt triangles is calculated. The triangles' areas are then used to integrate the potential difference and the potential sign flag over all non-corrupt triangles. Next, both the integrated values and the triangle areas are accumulated over all triangles. Finally, the accumulated values are used to compute the absolute mean value of the potential difference and the surface percentage exhibiting a sign change.

The first step is finding corrupt triangles according to the metric described in Section 3.5. Here, corrupt triangles are flagged with '`0`' and valid triangles are flagged with '`1`'. For each triangle, a kernel is invoked that computes the centroid of the three triangle vertices $c = (p_0 + p_1 + p_2)/3$ and then samples the volume at this centroid. If the difference between the sample and the midpoint is beyond a certain threshold (in this case, a threshold of 0.1 was used), then the triangle is flagged as 'corrupt'.

Next, the potential difference and the sign flag for each vertex are computed. Here, one kernel is invoked for each vertex. The kernel samples the potential texture of the source shape at the old unmapped vertex positions and the potential texture of the target shape at the new mapped positions. Subsequently, these values are integrated over the surface triangles using the method described in Section 3.6. Here, values of corrupt triangles are set to zero.

In order to compute the area for each triangle, one kernel per triangle is invoked which uses the three triangle vertices, $\vec{p}_0$, $\vec{p}_1$, and $\vec{p}_2$, and the precomputed triangle flag f to compute the total area of the triangle by

$$A = \|\vec{p}_1 - \vec{p}_2\| \, \|\vec{p}_2 - \frac{(\vec{p}_0 + \vec{p}_1)}{2}\| \, 0.5\, f \, . \tag{4.7}$$

Obviously, the triangle area is going to be zero if the triangle is flagged as 'corrupt'.

Assume that the areas of all triangles have been stored in a device array `triangleArea_D`. In order to get the total (non-corrupt) area of the surface, the values stored in this array need to be accumulated. This can be done using the THRUST library provided with the CUDA Toolkit. Here, the function `thrust::reduce(triangleArea_D, triangleArea_D + triangleCnt)` is used to compute the sum over all array elements.

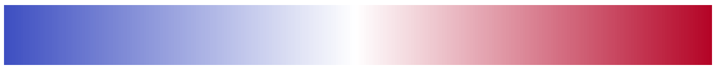
Finally, the absolute mean potential difference is now computed in the following way (the same for the surface area with switched sign): First, the integrated potential difference is accumulated over all triangles. This is done using the same THRUST function call as before. Assuming the total integrated potential difference is stored in `diffTotal`, the absolute mean potential difference `diffMean` can now be computed by `diffMean = diffTotal/areaTotal`, which is the desired value defined by the metric.

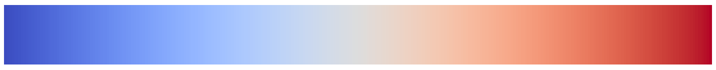
4.6 Rendering

The mapped surface and the computational results of the metrics are now rendered in both a shaded semi-transparent surface rendering and two 2D views (a matrix representation and a plot).

4.6.1 Color Map for Potential and Potential Differences

Finding a good and intuitive way of mapping the results of the previous computations to a colored representation is crucial for the understanding of the rendering. To this end, the electrostatic surface potential and the surface potential difference are encoded in different color


(a) Piecewise linear interpolation (RGB)


(b) Diverging color map (MSH)

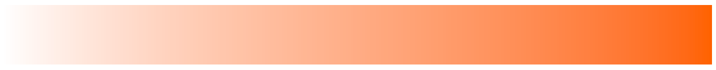

(c) Single hue variation

Figure 4.6: Comparison of interpolation in MSH color space and piecewise linear interpolation. The two base colors according to the in RGB color space definition are $c_{blue} = \{59, 76, 192\}$ and $c_{red} = \{180, 4, 38\}$. (a) shows the color map obtained by piecewise linear interpolation between the two base colors and white as intermediate color. The white banding artifacts are clearly visible. (b) shows the color map obtained by interpolating between the two base colors in MSH color space. (c) shows another colormap used in the surface rendering that is also interpolated in MSH space.

maps. The electrostatic surface potential is usually rendered using a diverging color map with blue for high potential and red for low potential (see e.g. [FBM02]). The potential difference is always greater then zero. It is, therefore, mapped to a single-hue color map with varying saturation.

OpenGL uses the *RGB (red green blue)* color space. It would, therefore, be straight forward to obtain a diverging color map by piecewise linear interpolation of the according colors. This, however, can lead to unwanted visual artifacts (match bands) since the interpolated color does not change in a perceptually uniform way [Sto03]. The match bands can be seen in Figure 4.6a. They visually separate regions of high potential and regions of low potential. However, this effect is unwanted for this implementation, since the color map is supposed to depict variations in the potential value in a perceptually uniform way. To avoid these issues, a different color space was used to represent the electrostatic surface potential. A good way of designing diverging color maps is to interpolate in *MSH (magnitude saturation hue)* space, as suggested in [Mor09]. The MSH color space is based on the CIELAB color space [WS00], which tries to approximate human perception better than other color spaces. The MSH space is a reformulation of the CIELAB color space with polar coordinates. A diverging color map can be obtained by piecewise linear interpolation of the the saturation s. Starting with the color (90, 1.08, 0.5) (red), the saturation is varied until is reaches zero. The hue is then switched to match the second color (90, 1.08, -1.1) (blue) and the saturation is increased again, until the second color has full saturation. The colors used in this thesis are the ones suggested in [Mor09]. The same cool-warm color scheme is also used in the Visualization

Toolkit[4]. The color map for the potential difference is obtained in a similar manner. However, only the saturation of one color (in this case orange) is varied. In both cases, the resulting colors have to be transformed to RGB color space as a final step, since this is the color space used by OpenGL. To this end, the color is first transformed to XYZ space and then to the RGB space. A summary of the color maps used in the rendering can be found in Figure 4.6.

4.6.2 3D Surface Rendering

For the surface rendering, the surface is represented by an array of triangles, together with the vertices and their according attributes, such as normals and texture coordinates. The texture coordinates are used to sample the electrostatic potential and, thus, to compute the potential difference between the original and the mapped surface.

All of the vertex position and attributes that were obtained as described in the previous sections are computed on the GPU. Hence, it would be an unnecessary overhead to copy the data back to the host, only to copy it back to the GPU for rendering. CUDA offers a possibility to overcome this overhead by letting the programmer map *vertex buffer objects (VBOs)* to a CUDA device array. After a VBO handle has been created, it can be registered with a `cudaGraphicsResource` and a mapped device pointer can be obtained. The data necessary for rendering can then be written inside a CUDA kernel by passing it the mapped pointer.

In the rendering routine, the VBO containing the vertex data is handed over to the rendering pipeline to draw the surface triangles. In the implementation used in this work, basically all information for the mapped surface is stored in one vertex buffer object with different offsets. Additionally, the vertex indices defining the triangles are stored in a separate VBO. For the semi-transparent rendering, the surface triangles have to be sorted by their depth values to ensure correct blending. This is done after the deformation process to ensure that the resulting depth values are correct. First, the depth value for the centroid of each triangle is computed using a CUDA kernel and stored in a device array. The sorting is done using the THRUST library by calling `thrust::stable_sort_by_key`. The depth values associated with the triangles are used as keys to sort the triangle index array. The rendering is finally initiated by binding both VBOs and calling `glDrawElements`.

Most of the rendering in this implementation is done using GLSL shaders. This includes the computation of the colormaps, but also the per-pixel lighting and the transparency of the surface. In the vertex shader, the vertex positions and attributes stored in the VBO are stored to the respective `varying` variables. Additionally, a view vector, an eye space normal, and the eye space light vector are computed for later use in the per-pixel lighting. The fragment shader uses the information stored in the varyings in the vertex shader to compute the final color and the lighting for every fragment. The lighting is done using the Blinn-Phong model [Bli77, Pho75], which combines an ambient term, a diffuse term, and a specular term to compute the lighting. The ambient term l_a is a constant factor. The diffuse term l_d is defined by the dot product of the surface normal N and the vector L, pointing from the surface point

[4] `http://www.vtk.org`

to the light source. The specular term l_s consists of the dot product of the normal N and the halfvector H. This yields

$$l_d = \frac{L \cdot N}{|L||N|} \qquad \text{and} \qquad l_s = \frac{N \cdot H}{|N||H|}, \tag{4.8}$$

where $H = \frac{L+V}{|L+V|}$.

In order to compute the potential difference, two kinds of texture coordinates are needed. First, the texture coordinates of the new positions with respect to the potential texture of the target shape. And second, the texture coordinates of the positions in the original surface with respect to the potential texture of the original shape. Here, it is important that the texture coordinates of the unmapped vertices are computed before the transformation based on the RMSD minimization is applied. In the fragment shader the transparency of the final color of the surface fragment is scaled by the uncertainty value mentioned in Section 3.6. To this end, the Euclidian distance between new and old fragment positions are computed and used to scale the alpha value of the final color. Furthermore, corrupt triangles are rendered completely transparent. An example for the semi-transparent rendering can be found in Figure 4.7.

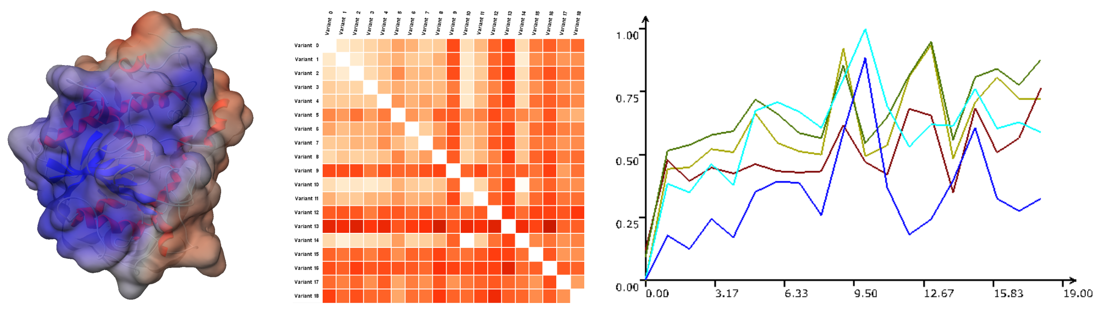
4.6.3 1D/2D-plot of the Metric

In addition to the 3D surface rendering, two plots showing the results of the metric were implemented. The 1D-plot shows different metrics (such as the Hausdorff distance or the mean potential difference) of one molecule variant with respect to all other molecule variants. A 2D heat-map facilitates a comparison between all the given variants. A renderer module that offers 1D-plots is already implemented in the MegaMol framework. The rendering of the 2D-plot is straightforward. The different metrics are computed for all combinations of given variants and stored in a matching matrix. Symmetric cases are computed in both directions, since it is not guaranteed that the mapping relation is symmetric. The values in the matching matrix are then stored in a texture object and sampled in the shader, using nearest neighbor sampling. Additionally, text labels are placed at the sides of the texture for better understanding. The value sampled from the texture is mapped to the same color map that was used in the 3D rendering. Examples of the renderings of both the 1D-plot and the 2D-plot can be found in Figure 4.7.

4.7 Implementation in the MegaMol Framework

MegaMol is a visualization framework used to visualize point-based molecular datasets [GRE12]. It is developed at the Visualization Research Center (VISUS) of the University of Stuttgart within the Collaborative Research Center 716.

MegaMol consists of three kinds of components: the core, a front-end, and plug-ins. The core comprises utility functions and basic functionality for the visualization of particle-based molecular data sets. The front-end serves as an interface for the end-user. It makes use of the core-functions for the purpose of data management and rendering. Currently, the only



(a) Surface rendering *(b) 2D-plot* *(c) 1D-plot*

Figure 4.7: A demonstration of different rendering methods used in this thesis. In (a), the semi-transparent surface rendering is shown. The surface is textured based on the surface potential. An example of the 2D-plot (showing the mean Hausdorff distance) can be found in (b). (c) shows the 1D-plot.

front-end available is a combination of GLUT [5] and the AntTweakBar [6]. Plug-ins can be used to extend the functionality of the core. Applications that are based on MegaMol normally consist of *modules* that can be connected with *calls* to form a rendering graph. The graph is specified in a configuration file and MegaMol connects the different modules during runtime.

In the remainder of this section, all modules and calls, as well as other classes that are essential for the visualization, are described. First, all classes are listed with a short description. Afterwards, the possible combinations of classes in the MegaMol framework are shown.

4.7.1 Summary of Implemented Classes

In the following, all the module and call classes used for the visualization are listed with a short description. Classes that are used in the visualization but were already present in the MegaMol frame work are listed separately.

Classes that have been implemented during this thesis:

- `VTILoader` – A data loading module, which loads the *.vti files used in the Visualization Toolkit. *.vti files are used to store the potential textures. The *.vti file format was chosen for compatibility with visualization software based on the visualization toolkit (such as Paraview [7]).
- `VTIDataCall` – A call that can be used by other modules to request data from the `VTILoader`.

[5] http://www.opengl.org/resources/libraries/glut/
[6] http://anttweakbar.sourceforge.net/doc/
[7] http://www.paraview.org/

- `ComparativeSurfacePotentialRenderer` – Render module that does the mapping and the 3D rendering of the mapped surface as described in Sections 4.3, 4.4, and 4.6.
- `ProteinVariantMatch` – Implements the mapping as described in Section 4.4 and the computation of the metric as in Section 4.5.
- `VariantMatchDataCall` – Call that can be used to request the matching matrix from the `ProteinVariantMatch` module.
- `VBODataCall` – Call to request a vertex buffer object handle containing vertex positions, vertex normals, and texture coordinates.
- `SharedCameraParameters` – A module to store camera parameters that are accessible for both read and write by a group of modules.
- `CallCamParams` – Call to request read/write access to the parameters stored in `SharedCameraParameters`.
- `LinkedView3d` – A view module that synchronizes its camera orientation with the one stored in `SharedCameraParameters` before doing any rendering. This module can be used to implement linked views.
- `SurfacePotentialRendererSlave` – A surface rendering module that does no computations on its own, but rather gets all the necessary vertex attributes from a `ComparativeSurfacePotentialRenderer` instance using `VBODataCall`.

Classes that have already been implemented in the framework:

- `View2d` – A basic view module for 2D renderings.
- `View3d` – A basic view module for 3D renderings.
- `CallRender3D` – A call that can be used to request 3D renderings from a render module.
- `CallRender2D` – A call that can be used to request 2D renderings from a render module.
- `PDBLoader` – A data source module that is used to load *.pdb files used in the RCSB Protein Data Bank [8].
- `MolecularDataCall` – A call to request data from an instance of `PDBLoader`.
- `DiagramRenderer` – A render module that provides 2D function plots.
- `MoleculeCartoonRenderer` – A render module that offers abstract rendering of the secondary structure of a protein.
- `SimpleMoleculeRenderer` – A render module offering atomistic molecular renderings using spheres or cylinders (e.g. a 'Ball-and-Stick' representation or the 'Spacefilling' model).
- `DiagramCall` – A call that is used to request diagram data to be plotted as a function.

[8] http://www.rcsb.org

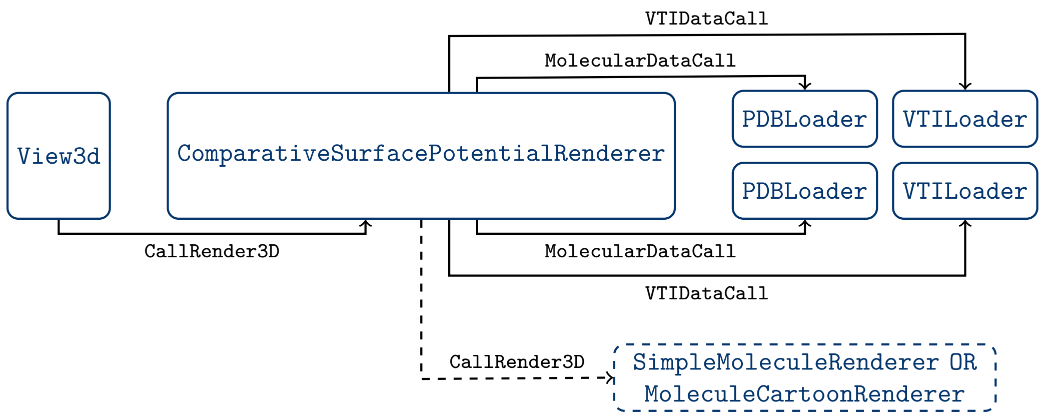


Figure 4.8: MegaMol call graph for a comparative surface potential rendering. The dotted parts are optional.

4.7.2 Visualizations using the MegaMol Framework

Using the notion of modules and calls, the different visualizations can be represented as call graphs. In the following, the different combinations of the classes mentioned before are combined to yield the final visualizations.

- The call graph shown in Figure 4.8 represents a visualization that implements the surface rendering described in Section 4.6. The `View3d` module requests the surface rendering from the renderer module through `CallRender3D`. The renderer module can request the data of both data sets to be compared from the `VTILoader` and the `PDBLoader` modules. There are two instances of each of the data loading modules since two data sets have to be loaded at interactive rates for comparison. Here, the `PDBLoader` loads the particle data and the `VTILoader` loads the potential texture of the respective data set. Additionally, a molecule renderer can be called for rendering. The result of the additional rendering is then blended with the semi-transparent surface rendering. The additional renderer, however, is optional (denoted in Figure 4.8 by dashed lines).
- The visualization described before can be extended by linked views, which can then be used to render not only the comparative visualization but also a surface potential rendering of the two original data sets. The `LinkeView3d` module inherits from `View3d` and extends its functionality by synchronizing its camera with the parameters provided by `SharedCameraParameters` before the rendering in every frame. Consequently, changes in the camera position or orientation that are e.g. caused by user inputs are applied to all three linked views. Figure 4.9 illustrates the call graph for this visualization.
- Besides the 3D surface rendering, a 1D -plot and a 2D-plot showing the results of the metric were implemented. Here, the renderer module requests the matching matrix from the `ProteinVariantMatch` module and renderes it either as a function plot or as

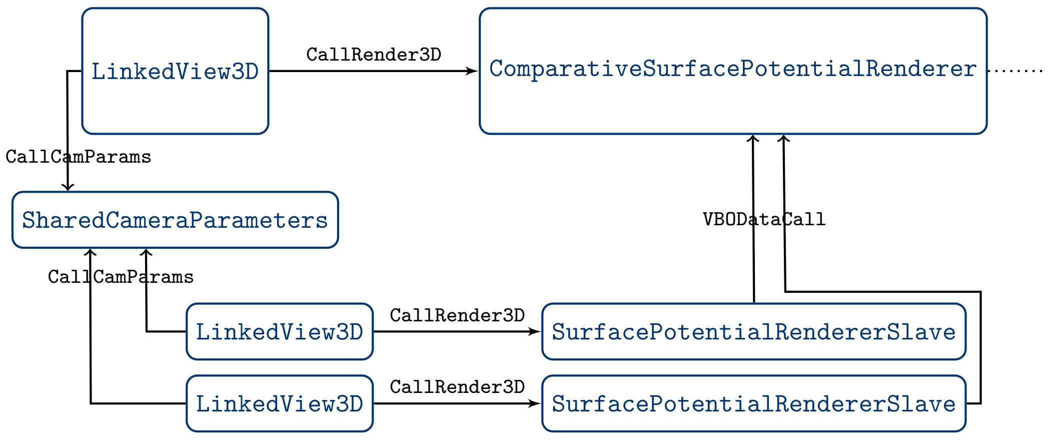


Figure 4.9: MegaMol call graph for a surface potential rendering using linked views. The data retrieval for the main renderer module is not included, but indicated by the dotted part. The left out modules and calls can be found in Figure 4.8.

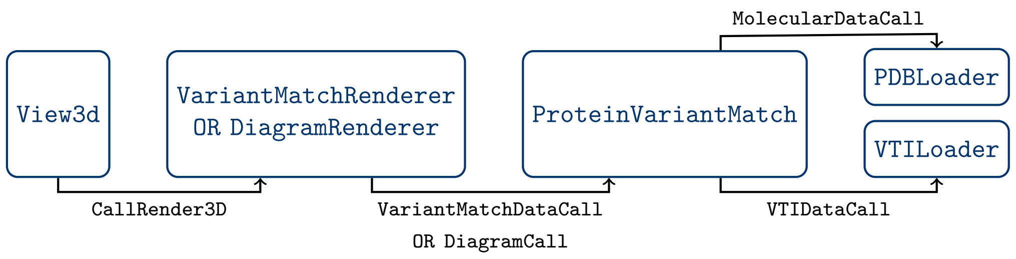


Figure 4.10: MegaMol call graph for a matrix-like 2D rendering and 1D function plot.

a heatmap-like 2D-plot. The module `ProteinVariantMatch` does all the computations regarding the mapping and the metric. It only uses one data call for all the data and, therefore, all the variants have to be stored in one data set. The renderer module can either be an instance of `DiagramRenderer` or `VariantMatchRenderer`. The call graph for both alternatives is illustrated in Figure 4.10.

5

Results & Discussion

The mapping algorithm is applied to different real-world particle-based data sets obtained from MD simulations. The mapping results are demonstrated by the 3D surface rendering and the 2D views. The overall performance of the implementation of the mapping and the rendering is tested with different input data sets. The impact of different parameters on the convergence of the mapping, the accuracy of the mapped surface, and the regularity of the mesh are investigated in a parameter study. Finally, strengths and weaknesses of the current mapping approach are discussed and possible future work is outlined.

5.1 Data Sets

First, the particle data sets to which the mapping approach is applied are described. All of the data used in the following are provided by the Institute of Technical Biochemistry (ITB) of the University of Stuttgart and were obtained by a MD simulation using the simulation package Gromacs 4.5.5 [HKSL08]. All simulations were executed over a time period of 70 ns, with discrete time steps of 2 fs.

5.1.1 Candida antarctica Lipase B (CALB)

The first data set is a file series of Candida antarctica Lipase B (CALB). The series consists of 19 variants of CALB in total (each with 4622 atoms). The variants have been simulated using different mixtures of solvent with varying percentages of water, methanol, toluol, and ethanol (summarized in Table 5.1). The comparative visualization is used to investigate whether the changes in functionality that come with using different solvents are related to a change in electrostatic properties. The method developed in this thesis is used to quantify and visualize differences in the electrostatic surface potential using both the 3D surface rendering and the 2D plot.

5.1.2 Triosephosphate Isomerase (TIM)

The second data set is a Triosephosphate Isomerase (TIM) mutant and its non-mutated parent structure. Investigating the mutated protein in terms of functionality is a task to the biovis

Table 5.1: List of different solvents simulated with the CALB. The different variants are organized in groups with similar solvent composition. In group A, the ration of water and toluol is varied slightly. In group B, the ratio of methanol and toluol is changed gradually. Variants in group C contain a small portion of water and a varying ratio of toluol and ethanol. Variants in groups D and E are similar to the ones in C, with a higher percentage of water.

	Variant	**Water**[%]	**Methanol**[%]	**Toluol**[%]	**Ethanol**[%]
Group A	00	4.7	0.0	95.3	0.0
	01	6.2	0.0	93.8	0.0
	02	7.6	0.0	92.5	0.0
Group B	03	5.7	12.5	81.8	0.0
	04	6.5	22.5	70.9	0.0
	05	5.7	39.5	54.9	0.0
	06	0.5	65.4	34.1	0.0
	07	1.3	64.9	33.8	0.0
Group C	08	0.3	0.0	99.7	0.0
	09	0.3	0.0	90.2	9.5
	10	0.3	0.0	81.6	18.1
	11	0.3	0.0	66.5	33.2
Group D	12	1.0	0.0	99.0	0.0
	13	0.9	0.0	89.6	9.4
	14	0.9	0.0	81.1	18.0
	15	0.8	0.0	66.1	33.1
Group E	16	3.2	0.0	96.8	0.0
	17	3.0	0.0	87.7	9.2
	18	2.7	0.0	64.9	32.4

contest 2013 [1]. The parent can be found in the RCSB Protein Data Bank[2] under the ID 2YPI and consists of 3811 atoms. The data set containing the mutant consists of 3759 atoms. In order to obtain the structure file of the mutant, the sequence was changed according to several punctual mutations. Aligned sequences of the original scTIM protein and the defective dTIM can be found in Figure 5.1. The alignment was obtained using the free alignment software ClustalX [3].

[1]http://www.biovis.net/contest
[2]http://www.rcsb.org
[3]http://www.clustal.org/

```
                **** *****:*:**:* . **:** *::    :   **  *..
dTIM            MARTPFVGGNWKMNGTKAEAKELVEALKAKLPDDVEVVVAPPAVYLDTAR    50
scTIM           MARTFFVGGNFKLNGSKQSIKEIVERLNTASIPENVEVVICPPATYLDYS    50

                 :*  .   ... :      ..     ..    .  .   ::   ..*:
dTIM            EALKGSKIKVAAQNCYKEAKGAFTGEISPEMLKDLGADYVILGHSERRHY   100
scTIM           VSLVKKPQVTVGAQNAYLKASGAFTGENSVDQIKDVGAKWVILGHSERRS   100

                : . *: .  . ::        :        *.: . .   *..:    :
dTIM            FGETDELVAKKVAHALEHGLKVIACIGETLEEREAGKTEEVVFRQTKALL   150
scTIM           YFHEDDKFIADKTKFALGQGVGVILCIGETLEEKKAGKTLDVVERQLNAV   150

                    .:*.***:*********** :****:**::** ***:**.::. :.
dTIM            AGLGDEWKNVVIAYEPVWAIGTGKTATPEQAQEVHAFIRKWLAENVSAEV   200
scTIM           LEEVKDWTNVVVAYEPVWAIGTGLAATPEDAQDIHASIRKFLASKLGDKA   200

                *..:*******.: :**  :  :.*:**************:*******
dTIM            AESVRILYGGSVKPANAKELAAQPDIDGFLVGGASLKPEFLDIINSRN   248
scTIM           ASELRILYGGSANGSNAVTFKDKADVDGFLVGGASLKPEFVDIINSRN   248
```

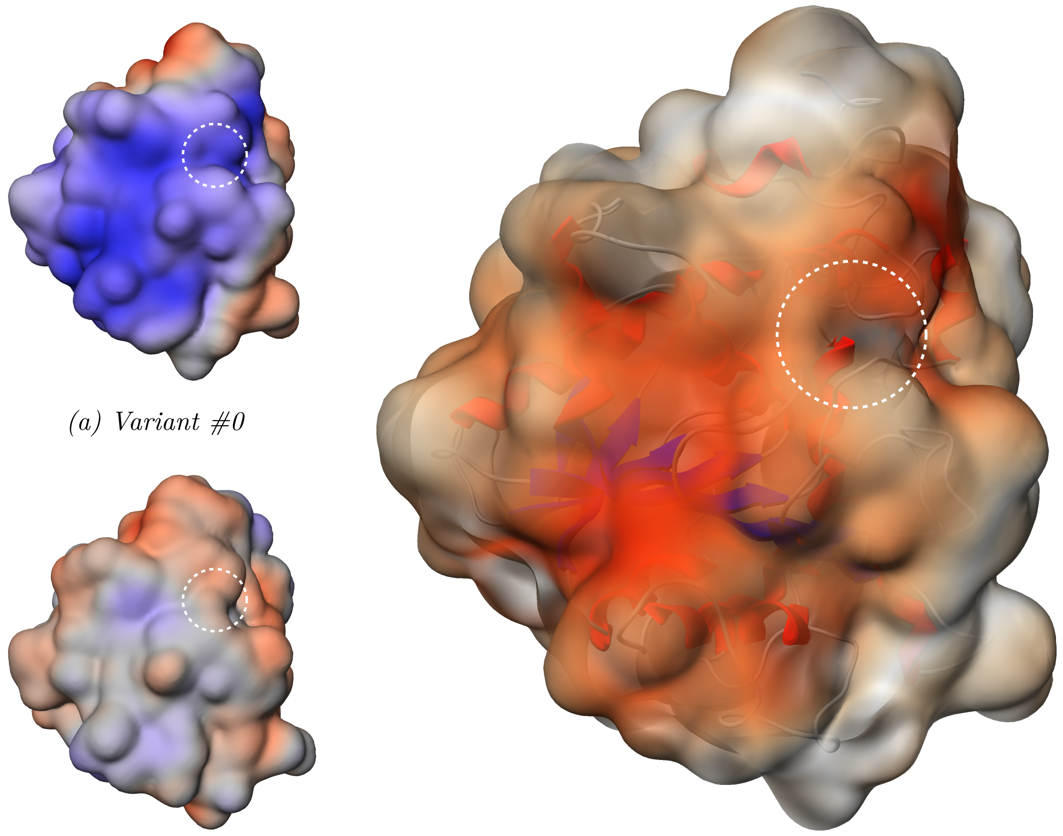
Figure 5.1: Aligned sequences of the parent scTIM and the defective dTIM.

5.2 3D Surface Rendering

Figure 5.2 shows a comparative rendering for CALB variants 0 and 10. The uncertainty caused by strongly varying geometry is indicated by a high transparency value. Additionally, a cartoon rendering of the molecular structure is rendered in combination with the semi-transparent surface. The cartoon rendering provides additional clues about the underlying molecular structure and helps to distinguish transparent parts from opaque parts of the surface. The highlighted area shows how a difference in the surface geometry leads to a transparent patch in the comparative surface rendering. In the upper part of the highlighted area, the higher potential difference is depicted by a higher saturation of the surface coloring in the difference rendering.

A similar rendering can be found in Figure 5.3. Here, the comparative visualization of the scTIM and the defective dTIM is shown. The potential sign difference rendering in Figure 5.3d shows that the potential sign varies strongly through out the whole molecular surface. The potential difference, however, has several peeks in a couple of distinguishable areas. The different results of the two metrics indicate that it is justified to include both of them in the visualization.

Figure 5.5 shows the potential difference for variant 0 with respect to all other variants. For comparison, a 1D-plot of the different metrics and the mean Hausdorff difference of variant 0 with all other variants is shown in Figure 5.4. Although the 3D rendering shows only one specific camera angle, there is clear correspondence between the visible patches containing areas of high surface potential difference in the surface texture and the values shown in the 1D-plot. The mean potential difference with respect to variants 1 to 8 is low to medium, whereas it has a peak at variants 9 to 11. The high uncertainty in some areas of the comparative rendering with variants 9 and 13 is also denoted in the 1D-plot by the high mean Hausdorff distance



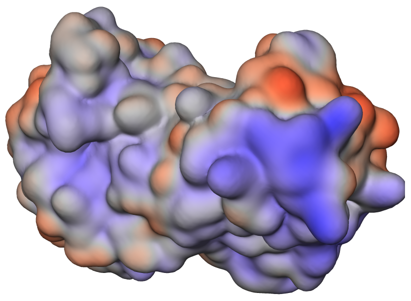
(a) Variant #0

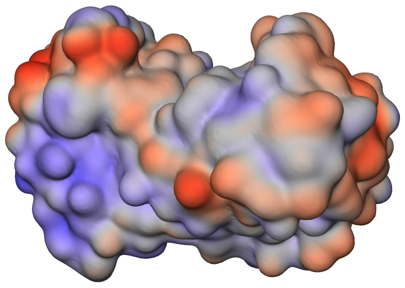
(b) Variant #10

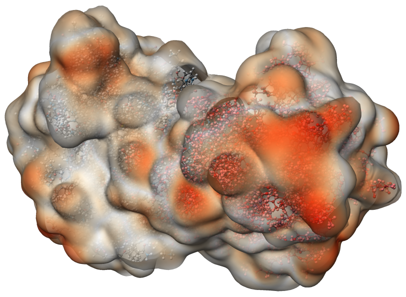
(c) Potential difference

Figure 5.2: 3D surface rendering of the potential difference. Uncertainty is symbolized by increased transparency. The highlighted area shows how variation in the surface shapes of the input data increases uncertainty of the potential difference. Variant 0 is convex whereas variant 10 has a concave bump. The different surface geometry leads to the area being drawn with a high transparency value. The additional cartoon rendering of the molecular structure improves the perception of the transparent areas.

values. In the 3D rendering, variant 14 seems to have a similar amount of potential difference as variant 15, although there is a clear difference visible in the 1D plot. This is due to the fact that only part of the surface is visible in the image. A similar series of comparative renderings can be found in Figure 5.6. Here, the potential sign difference of variant 0 with respect to all other variants is shown. The amount of potential sign difference clearly defers from the mean potential difference. However, a similar development becomes visible when using the 1D plot. The peaks at variants 10 and 15 can e found in both functions. The same holds for the local minima at variants 8 and 12. The rather low differences of variant 0 and variants 8 and 12 actually comply with their similar solvent composition (see Table 5.1).


(a) scTIM


(b) dTIM


(c) Potential difference

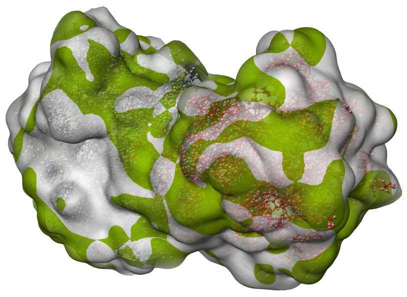

(d) Potential sign difference

Figure 5.3: 3D surface rendering of the potential difference and the potential sign difference. The pictures show a comparison of the two TIM variants. Uncertainty is symbolized by increased transparency. Areas in which both input molecules have a different potential sign are shown in green. Here, the molecular structure was added using a ball-and-stick (c) and a line representation (d).

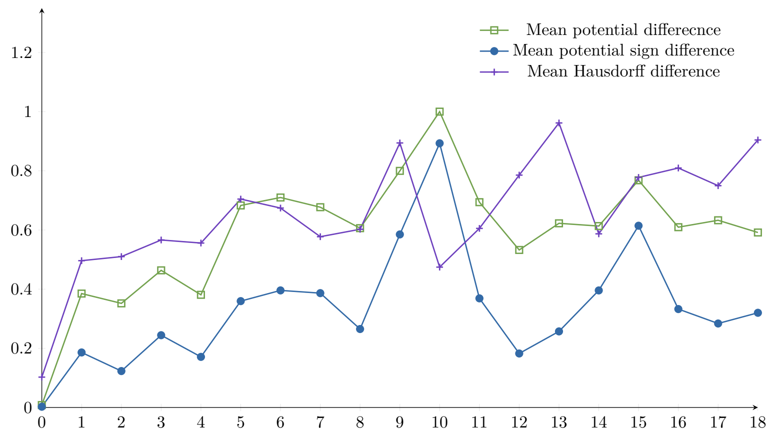


Figure 5.4: Hausdorff distance, mean potential difference, and mean potential sign difference of CALB variant 0 with respect to all other variants. All values have been normalized to a range of [0,1] to make them comparable.

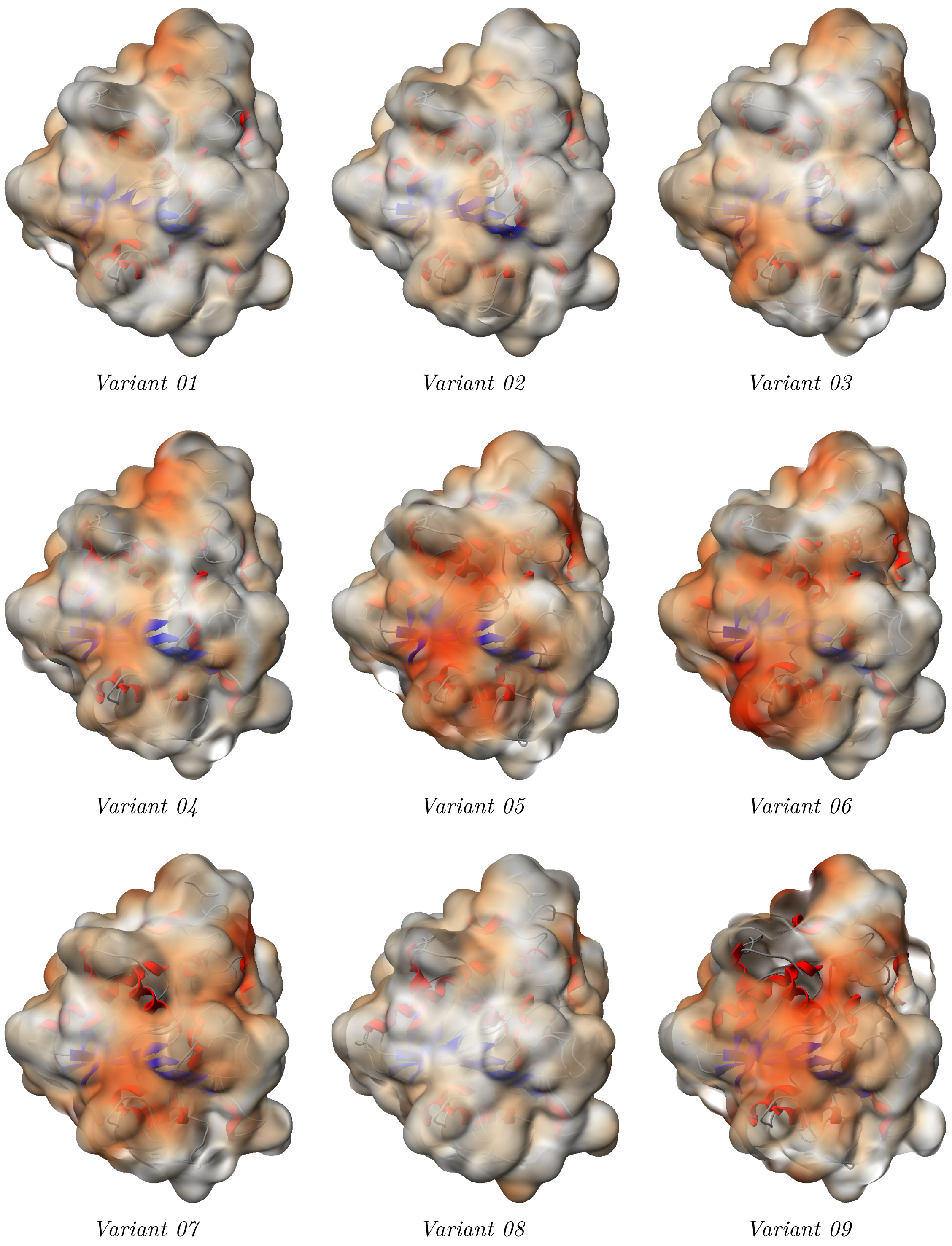
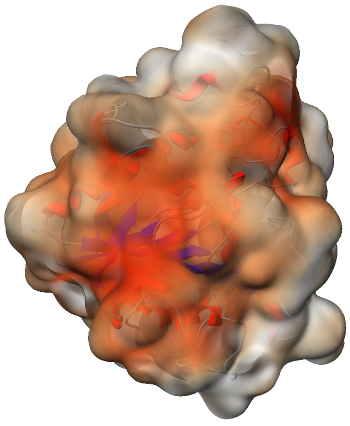
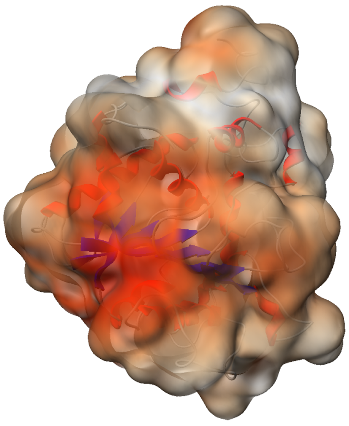
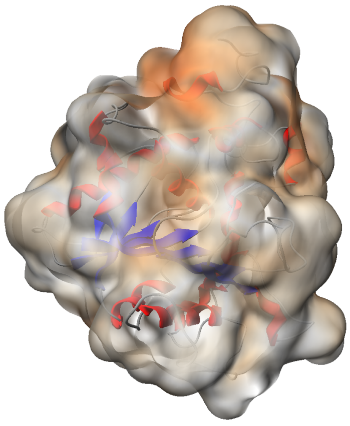
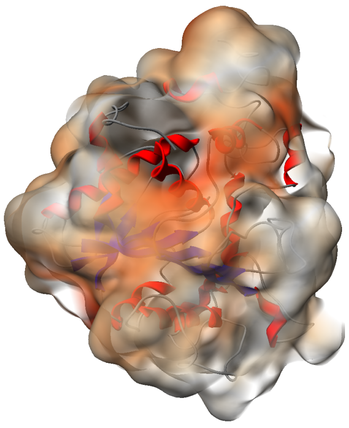


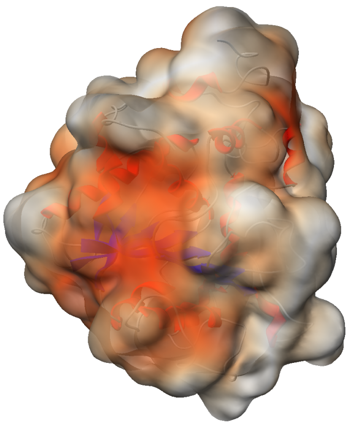
Figure 5.5: 3D surface rendering showing the potential difference. Variant 0 of the CALB file series is compared with all other variants.

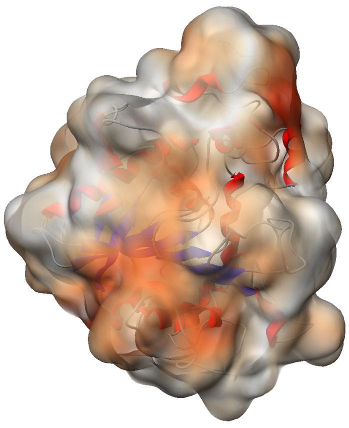

Variant 10

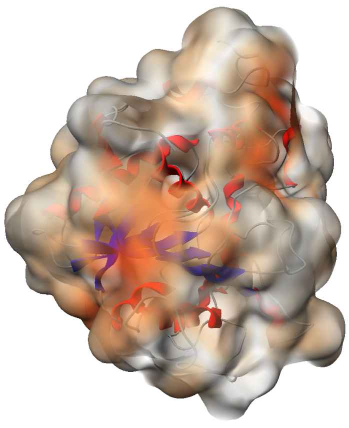

Variant 11

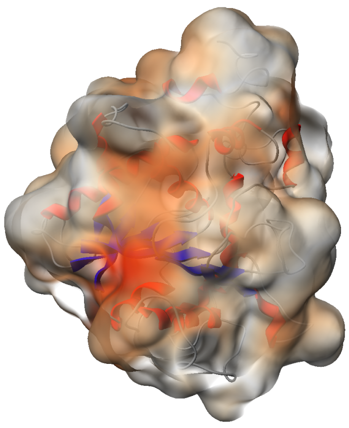

Variant 12

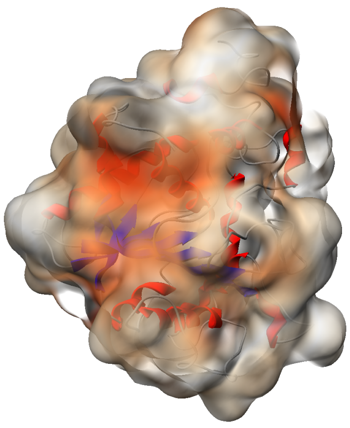

Variant 13


Variant 14


Variant 15


Variant 16


Variant 17


Variant 18

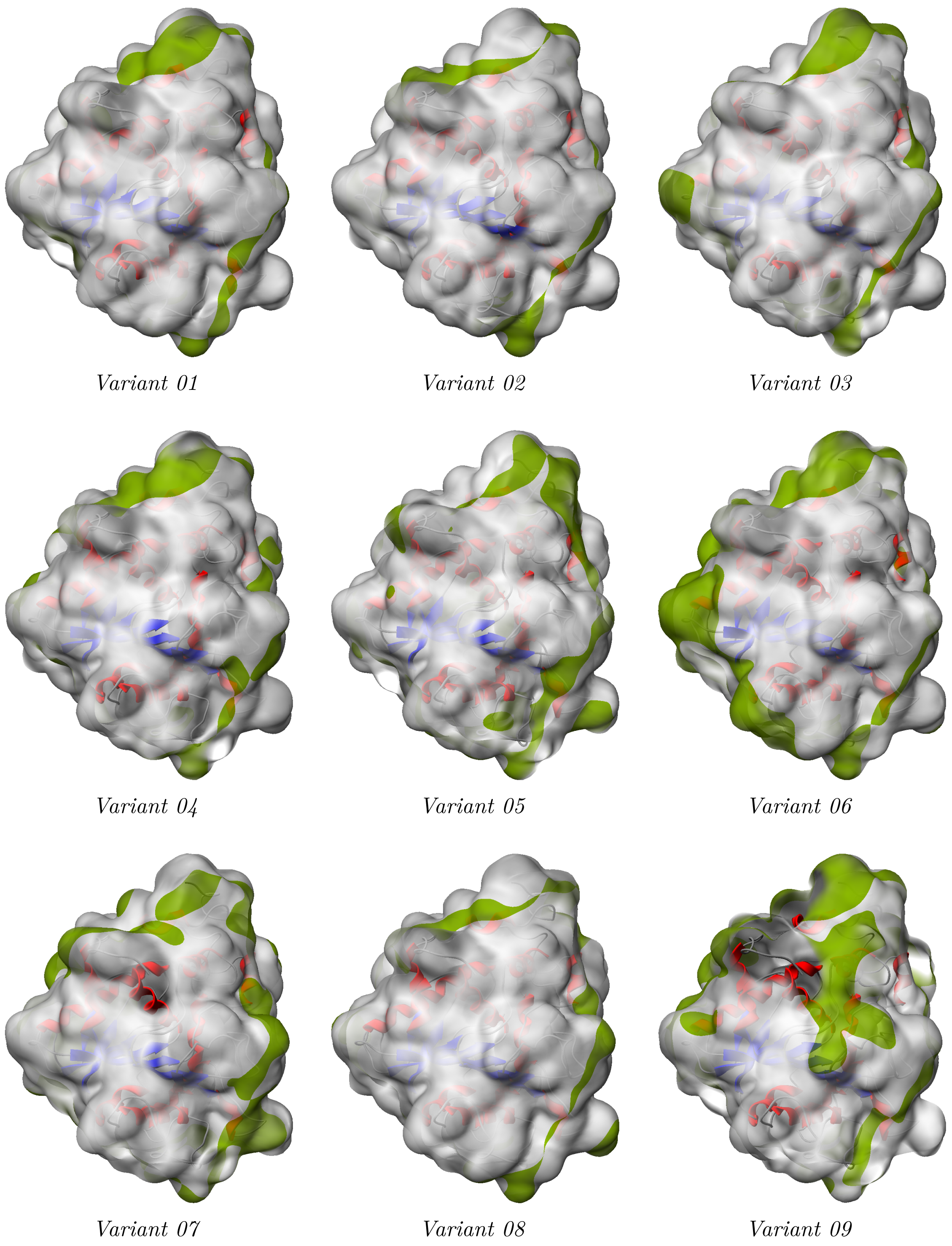
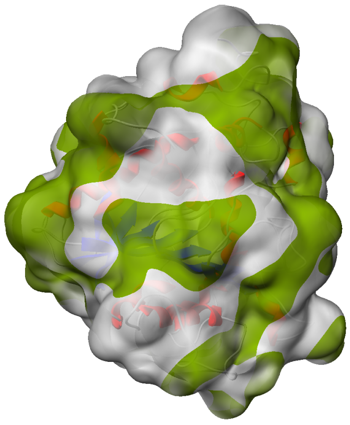
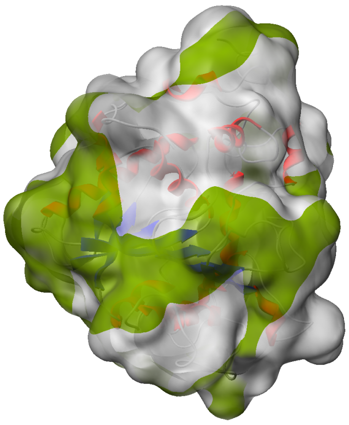
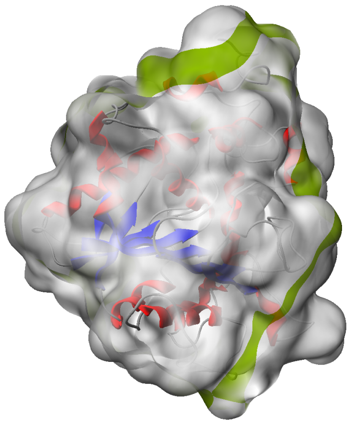
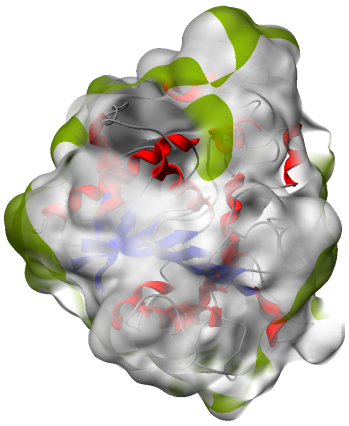


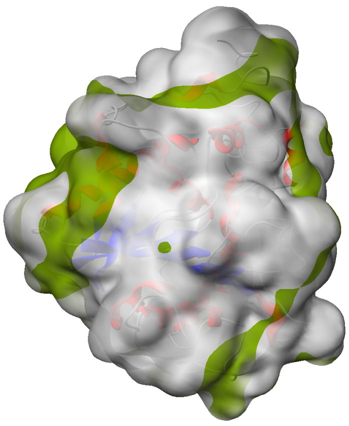
Figure 5.6: 3D surface rendering showing area the potential sign change. Variant 0 of the CALB file series is compared with all other variants.

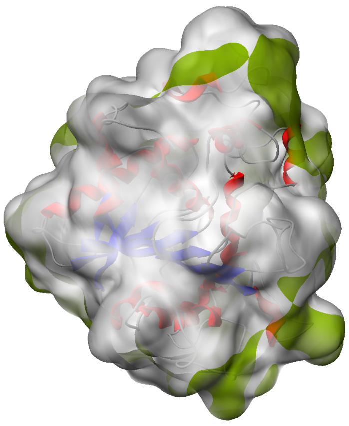

Variant 10

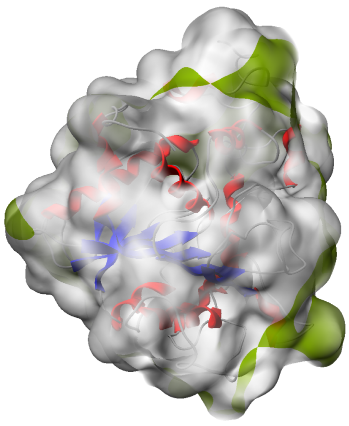

Variant 11

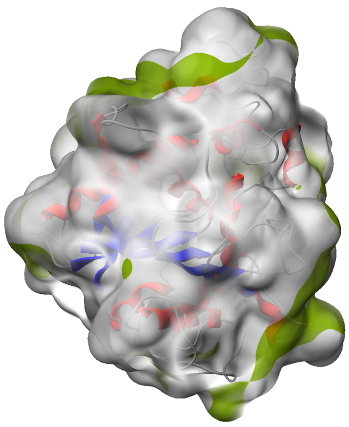

Variant 12

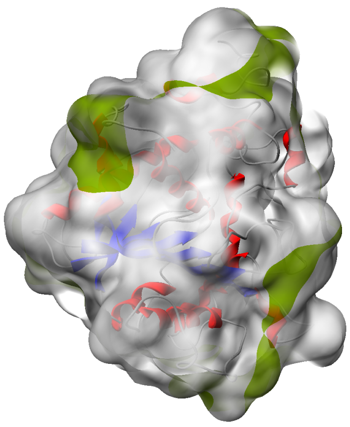

Variant 13


Variant 14


Variant 15


Variant 16


Variant 17

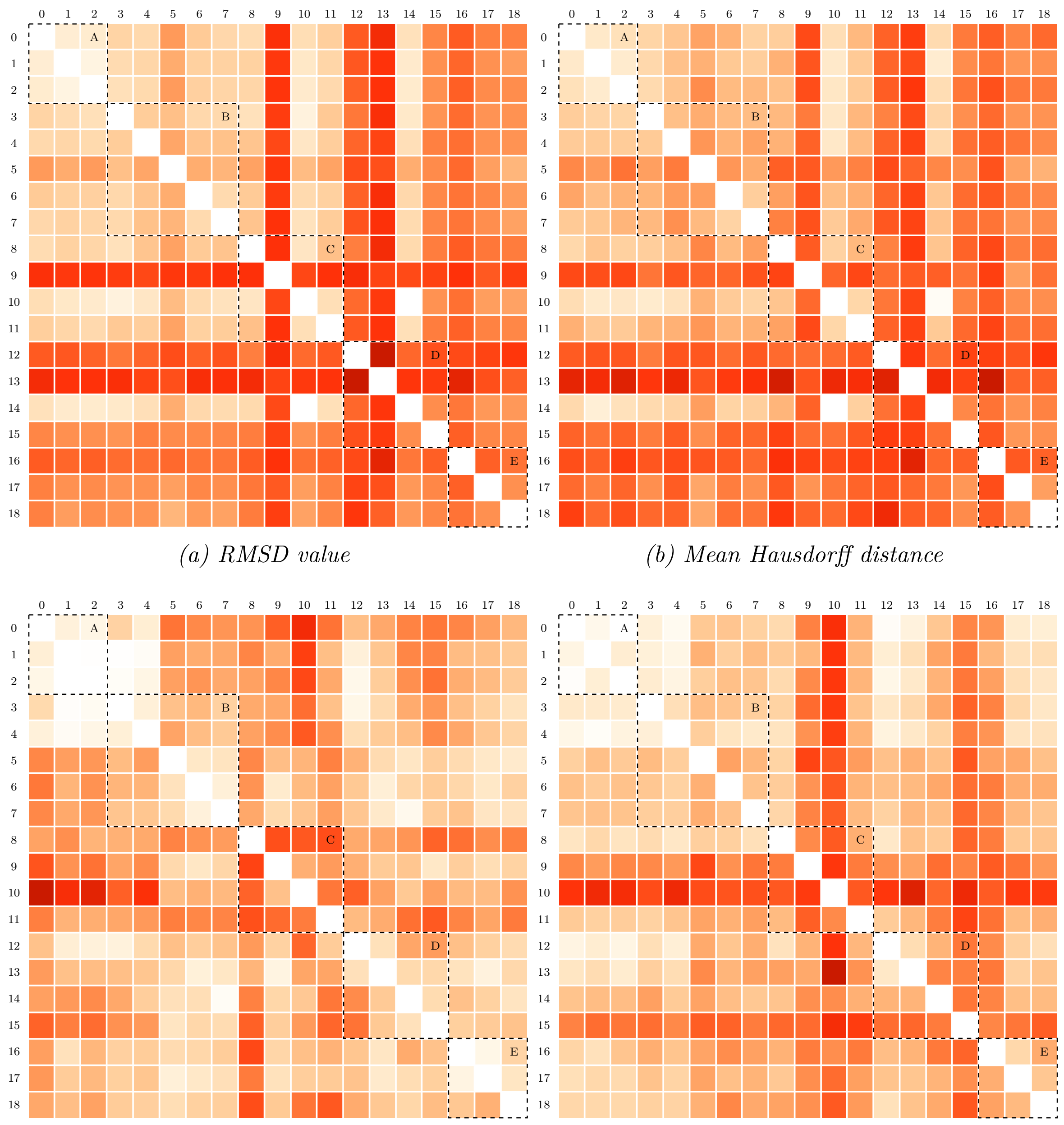

Variant 18

5.3 2D Plot

The 2D-plot for the CALB variants is now rendered using the mean potential difference, the potential sign switch area, the mean Hausdorff difference, and the RMSD value. All values are visualized using a color map that is interpolated in MSH space based on minimum and maximum values. Note that the relation of each variant to itself is not included in the color mapping for clarity. It is, however near zero in all four cases.

The plots showing the RMSD value (Figure 5.7a) and the mean Hausdorff distance (5.7b) are clearly corresponding. In both plots, the variation of the metric and the location of extrema is similar. This is most likely due to the fact that the molecular surfaces are defined implicitly based on the underlying molecular structure. Therefore, a change in the molecular structure, which is measured by the RMSD value, also implies a change in the molecular surface, which is quantified by the mean Hausdorff distance. The differences in both metrics are caused by the fact that not every atom is associated with the same radius and, therefore, the molecular surface behaves slightly differently. Furthermore, the RMSD value contains changes inside the molecule, whereas, the mean Hausdorff distance is reduced to the molecular surface. Both metrics can be seen as a measurement for uncertainty. However, since the main target of the visualization is the molecular surface, the mean Hausdorff distance has a higher correspondence to the final result.

Figures 5.7c and 5.7d show 2D-plots of the pair-wise mean potential difference and the percentage of the surface area in which the potential sign is different. The different groups of variants are depicted by squares. Both metrics show only weak changes in group A. The variants in group B show a slightly varying sub-grouping. In terms of the mean potential difference, variants 3 and 4 and variants 5 to 7 show similar behavior. In variants 3 and 4, the solvent has a significantly higher percentage of toluol, which seems to lead to an even more similar surface potential. Likewise, variants 5 to 7 in which the percentage of toluol is lesser or equal the percentage of methanol show a low value in the mean potential difference metric. The similarities for variants 5 to 7, however, are not present in the plot of the mean potential sign change. Significant changes in both the absolute value of the surface potential and the sign of the surface potential are shown for group C. This also holds for groups D and E, although the differences seem to decrease with increasing water percentage.



(a) RMSD value

(b) Mean Hausdorff distance

(c) Mean potential difference

(d) mean potential sign change

Figure 5.7: Comparison of different pairwise metrics of all variants of CALB. The pictures are screenshots of the 2D rendering in MegaMol. (a) shows the pair-wise RMSD value, (b) is the mean Hausdorff distance, (c) is the mean potential difference, and (d) represents the percentage of the surface area that has an opposite sign with respect to the source shape. Groups of variants with similarly composed solvents (see Table 5.1) are depicted by dashed squares (not part of the actual rendering).

Table 5.2: Performance of the surface generation. One variant of the CALB and the TIM mutant are compared. All values are approximate and in ms. In all computations, the grid spacing was set to 1.0Å *in all dimensions, the atom radius was the van der Waals radius, and the isovalue was 0.5.*

CUDA kernel	**Data set**	
Grid layout: 256 threads per block	CALB variant #0 Vertices: 42338 Grid: 126×109×113 Triangles: 84124	TIM mutant Vertices: 66818 Grid: 91×110×91 Triangles: 133636
`FindActiveGridCells_D`	0.55 ms	0.33 ms
`CalcCubeMap_D`	0.13 ms	0.09 ms
`CalcVertexPositions_D`	0.09 ms	0.13 ms
`CompactActiveVertexPositions_D`	0.05 ms	0.07 ms
`GetTrianglesIdx_D`	0.15 ms	0.24 ms
`ComputeVertexConnectivity_D`	0.42 ms	0.67 ms
`ComputeVertexNormals_D`	1.18 ms	1.87 ms
`ComputeVertexTexCoords_D`	0.05 ms	0.09 ms
`SortTriangles`	1.55 ms	2.00 ms
$\sum$	4.17 ms	5.49 ms

5.4 Performance of the Implementation

In order to facilitate exploratory analysis, it is important to maintain interactivity and avoid long computation times. Therefore, the performance of the individual CUDA kernels used in the surface generation and surface mapping step are measured. Furthermore investigated is the frame rate that can be achieved with the semi-transparent surface rendering. The implementation was tested on an Intel Core i3-3220 (4× 3.3 GHz) with 8 GB RAM and a NVIDIA GeForce GTX 660 with 2GB VRAM. The CUDA device is of compute capability 3.0 and has 5 multiprocessors. The maximum shared memory per block is 48 KiB and the maximum number of threads per block is 1024. The implementation was tested with one variant of the CALB files series and the TIM mutant. For the measurements, the CUDA kernels are grouped in three sets. First, the surface generation framework is tested. Then, the surface mapping for one iteration is tested using both linear and cubic interpolation. Finally, the frame rate achieved with the semi-transparent surface rendering is measured. As in the implementation chapter, only implementations done for the thesis are considered. The performance of the individual CUDA kernels is measured using the GPU timing functionality provided by the CUDA runtime API.

Table 5.2 shows a summary of the performance of the surface generation. The CALB data set produces a significantly larger grid then the TIM mutant, since the bounding box is initialized to contain all 19 variants and there is some variation in the positioning of the different variants.

Table 5.3: Performance of the surface mapping. One variant of the CALB and the TIM mutant are compared. All values are approximate and in ms. In all computations, the grid spacing was set to 1.0*Å in all dimensions, the atom radius was the van der Waals radius, and the isovalue was 0.5. The vertex positions were updated using one iteration per kernel.*

CUDA kernel	**Data set**	
Grid layout: 256 threads per block	CALB variant #0 Vertices: 42338 Grid: 126×109×113 Triangles: 84124	TIM mutant Vertices: 66818 Grid: 91×110×91 Triangles: 133636
`InitExternalForceScl_D`	0.07 ms	0.10 ms
`CalcVolGradient_D`	0.50 ms	0.30 ms
`CalcVolGradientWithDistField_D`	0.56 ms	0.34 ms
`UpdateVertexPositionTrilinear_D`	1.01 ms	1.83 ms
`UpdateVertexPositionTricubic_D`	1.86 ms	3.47 ms
`FindCorruptTriangles_D`	0.21 ms	0.32 ms

The TIM, however, produces more vertices, since the protein is larger, which leads to a larger surface area. This configuration of input data allows identifying the main factors for the different CUDA kernels' performance. `FindActiveGridCells_D` and `CalcCubeMap_D` obviously depend on the grid size and the resulting number of grid cells. The rest of the kernels depend on the number of vertices and, therefore, the surface area of the molecule. Overall, the time for the computation of the surface generation is in the scope of milliseconds. Thus, when using dynamic data sets, interactive frame rates could be maintained when doing surface generation only.

The next set of functions that are tested are the kernels that compute one iteration of the mapping algorithm. Also measured are preprocessing steps, such as the computation of the distance field, the external force scale factor initialization and the computation of the gradient using the target volume and the distance field. The mapping iteration is computed using both trilinear and tricubic interpolation. CALB variant 0 is mapped to variant 1 of the same file series and the TIM mutant is mapped to the regular TIM. Furthermore, the computation time of the search for corrupt triangles after the mapping is investigated. Table 5.3 shows a summary of the computation times for the surface mapping. Clearly, the time needed for one iteration step (i.e. `UpdateVertexPositionTrilinear_D` or `UpdateVertexPositionTricubic_D`) is crucial to the overall runtime since it is potentially executed several hundred times. It is also the most costly of the CUDA kernels needed for the surface mapping. Using trilinear interpolation is, not surprisingly, significantly faster than tricubic interpolation. In order to maintain a frame rate of 60 fps, the mapping could only execute up to 9 iterations per frame (with linear interpolation). However, a realistic number of iterations would be about 300-400 iterations (see Section 5.5). Hence, it is hardly possible to maintain interactivity, when the mapping is computed in every frame.

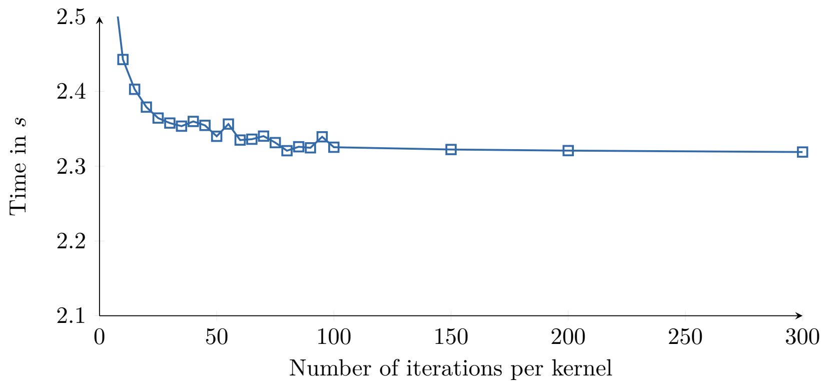


Figure 5.8: Running time of 3000 iterations using the multi-iteration kernel. The number of iterations computed in one kernel invocation is varied from 1 to 300. The running time converges quickly towards ∼2.32 s.

As mentioned in Chapter 4, a multi-iteration kernel was implemented to speed up the computation by saving a number of kernel invocations and some read/write-operations to global device memory. In the following, the impact of the multi-iteration approach on the overall performance is tested. To this end, the mapping of CALB variant 0 to CALB variant 1 was investigated while changing the number of iterations computed in one kernel invocation. Figure 5.8 shows the running time for a run of 3000 iterations with varying numbers of iterations computed per kernel while using linear interpolation for all sampling operations. Computing multiple iterations in one kernel clearly reduces the overall running time, despite the additional thread synchronization. The running time converges quickly to a certain value, which it keeps from about 100 iterations on. Overall, the running time is reduced by about 10%, which is beneficial, since the iterations make up the main part of the costly computations.

Finally, the frame-rate of 3D surface rendering is measured. A resolution of 1024×1024 was used with the camera being fully zoomed in and the molecular surface being fully visible. All renderings were additionally tested in combination with the abstract cartoon rendering of the secondary structure. The measurements can be found in Table 5.4. Both the semi-transparency and the cartoon rendering lower the frame-rate. The linked view also shows a slight decrease in performance, since three surfaces are rendered and shaded. However, overall, interactive frame rates can be maintained in all combinations.

5.5 Parameter Study

The result of the mapping algorithm can vary greatly depending on how the input parameters are chosen. The impact of different parameters on the mapping results is, therefore, investigated. The mapping result is evaluated according to three criteria. The first criterion is the convergence of the mapping, i.e. how many iterations are necessary until a stationary solution is reached. The second criterion is the accuracy of the fully converged deformed model. This is quantified

Table 5.4: Performance of the surface rendering. The surface rendering was tested using different color maps, with and without transparency, with and without additional rendering of the molecular structure.

Rendering	**Data set**	
Resolution: 1024 × 1024	CALB variant #0 Vertices: 42338 Triangles: 84124	TIM mutant Vertices: 66818 Triangles: 133636
Opaque surface (RGB space color map)	∼ 380 fps	∼ 285 fps
Opaque surface (MSH space color map)	∼ 320 fps	∼ 260 fps
Semi-transparent surface	∼ 285 fps	∼ 245 fps
Semi-transparent surface (with cartoon rendering)	∼ 111 fps	∼ 77 fps
Linked view (semi-transparent surface and 2× opaque MSH color mapping)	∼ 235 fps	∼ 210 fps
Linked view (the same as above, but with cartoon rendering)	∼ 83 fps	∼ 47 fps

by the percentage of the surface area that consists of corrupt triangles. The last criterion is the regularity of the fully converged triangle mesh. This criterion is important for the regularization step.

This parameter study is limited to the testing of one parameter at a time, combinations of parameters are not tested. The minimum displacement parameter f_{min} is set to 10^{-4}Å, which is one thousandth of the chosen grid spacing of the density volume. For each criterion, the three remaining parameters are varied: the force scaling σ, the rigidity ρ, and the force weighting μ. The weighting parameters ρ and μ are each varied from 0.05 to 0.95. The force scale factor σ is varied from 0.01 to 0.17, since for larger values, the mapping fails to converge according to the chosen value for the minimum displacement f_{min}. The convergence and the accuracy of the mapping are tested based on a fully regularized source shape. The regularization is tested on a non-regular triangle mesh, as delivered by the Marching Tetrahedra algorithm. All computations are done using both linear and cubic interpolation. The CALB file series serves as input data. The computational results are averaged over all 19 variants.

5.5.1 Convergence of the Deformable Model

First, the convergence of the deformable model algorithm is tested. To this end, the algorithm is executed until the average vertex displacement falls below the threshold defined by f_{min}. One parameter at a time is varied, while the rest of the parameters have fixed values. The fixed value for ρ is 0.3, the external forces μ weighting is set to 0.75, and the scaling factor σ is 0.1. Figure 5.9 shows the average number of iterations until convergence with respect to different parameters with linear and cubic interpolation, respectively. Setting a high weight on the external forces (by increasing μ) leads to very fast convergence. This is probably due to the fact that the external force is determined locally by sampling the target volume, whereas the internal force is propagated through the whole mesh and, therefore, takes much longer to fall under a certain threshold. The rigidity ρ has a slight impact on the number of iterations, however, not as strong as μ. A high value on ρ, and, therefore, a high weight on the rigidity term slightly improves the convergence of the algorithm. By far the biggest impact on the convergence has the global force scaling factor σ. When choosing values beyond 0.1, the number of necessary iterations quickly rises to several thousand. In general, the choice of a cubic interpolation scheme rather then a linear one only slightly improves the convergence of the method. An improvement when using linear interpolation can only be seen when using a (too) high global scaling factor σ.

5.5.2 Accuracy of the Mapped Surface

The next criterion to be tested is the accuracy of the fully converged mapped surface. Again, the algorithm is executed until the average vertex displacement falls below the threshold defined by f_{min}. The portion of the surface area that consists of corrupt triangles is then calculated as described in Section 3.5.6. As before, only one parameter at a time is varied, while the rest of the parameters have fixed values. The fixed value for ρ is 0.3, the external forces μ weighting is set to 0.75, and the scaling factor σ is 0.1. The impact of different parameters on the portion of the surface area that consists of corrupt triangles can be found in Figure 5.10. A high value for μ clearly leads to higher accuracy, which is not surprising, since the external forces pull the source shape towards the target shape. A higher rigidity value ρ leads to a slight increase of corrupt triangles. This is the result of the over-smoothing that the rigidity term causes when being weighted too highly. The global force scaling factor σ has a rather interesting impact. The amount of corrupt triangles increases if it is chosen either too low or to high. An ideal value according to chosen grid spacing of 1Å seems to lie somewhere around 0.1. In order to achieve the best possible result, the value of this parameters should, therefore, be chosen with respect to the grid spacing. In all three cases, choosing cubic interpolation leads to a decrease of corrupt triangles.

5.5.3 Regularity of the Triangle Mesh

The last criterion that is investigated is the regularity of the triangle mesh. Here, the regularization process is investigated, since a regular mesh is not the main goal in the mapping algorithm. As before, σ is set to 0.1 and ρ is set to 0.3. However, the external forces μ weighting is set to 0.5 because during regularization, the source mesh already starts out near the target shape. The regularity is measured with respect to the non-weighted, non-scaled internal force acting on each vertex in the fully converged mesh. The length of this internal force displacement vector quantifies the 'desire' of the triangle mesh to be regularized and is, therefore, interpreted as a measurement for the mesh distortion. The computational results can be found in Figure 5.11. Using a higher value for σ leads to a less regular grid. This is expected since the purpose of the internal forces is to maintain a regular, smooth mesh.

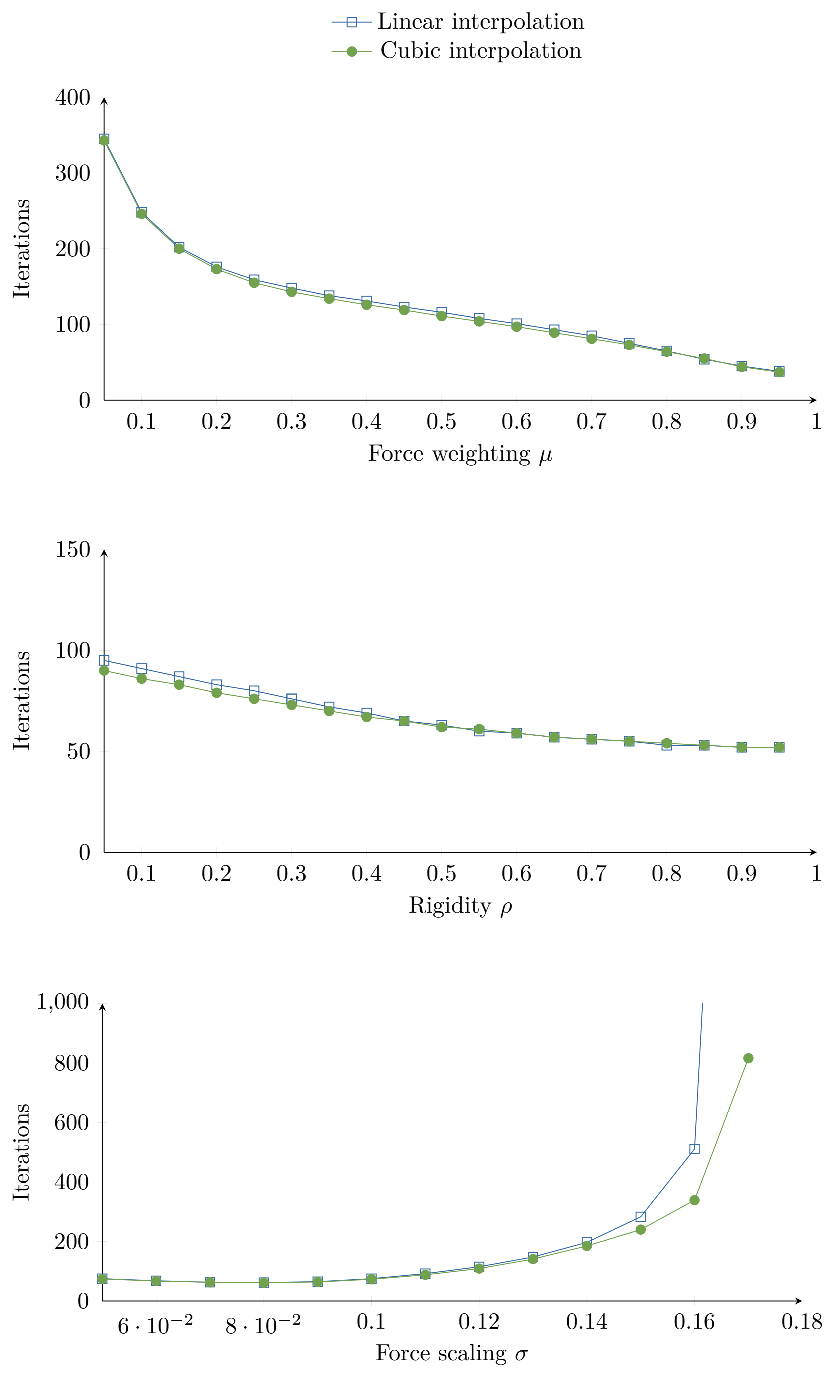


Figure 5.9: The impact of different parameters on the convergence of the deformable model. The plots shows the average number of iterations necessary for the CALB file series to reach a stationary solution.

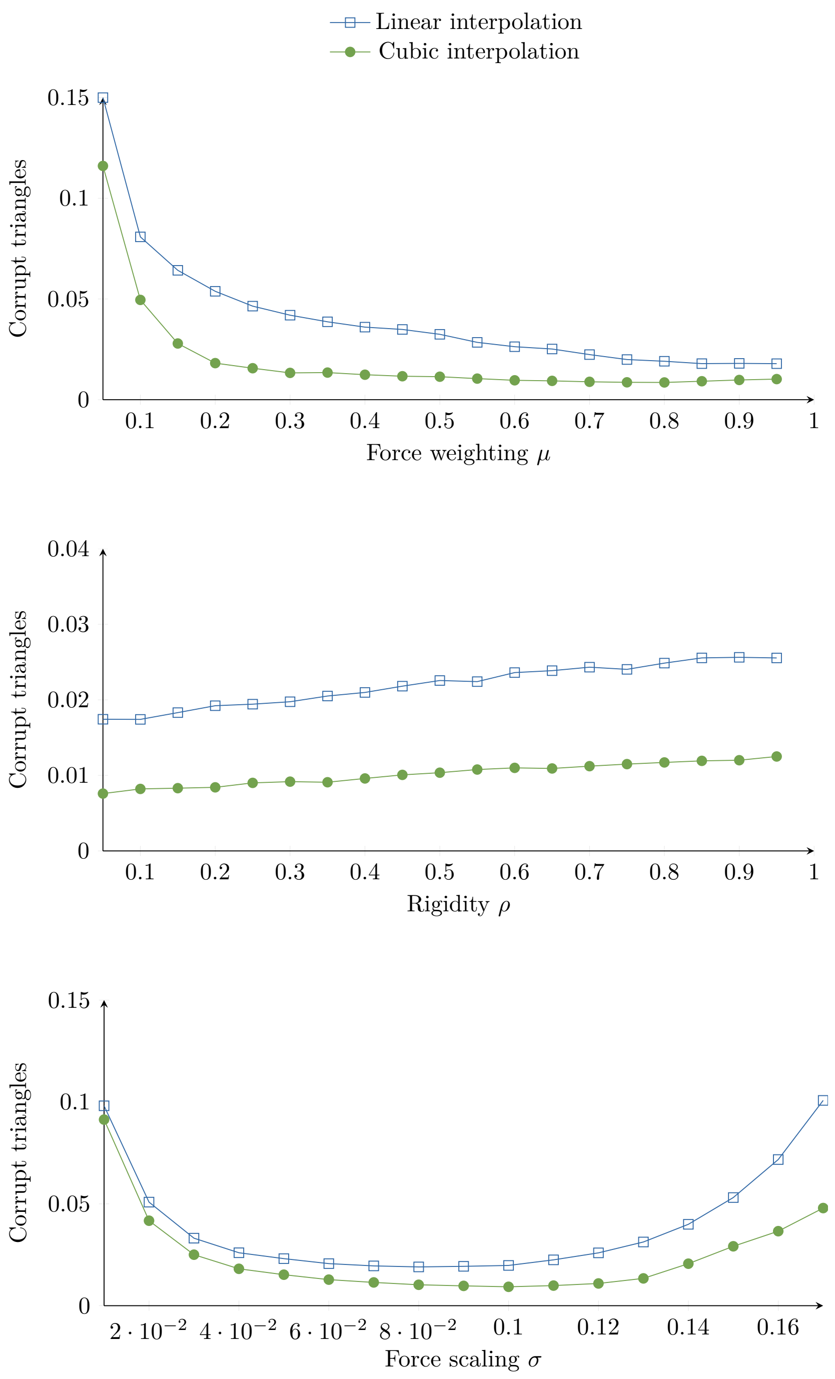


Figure 5.10: The impact of different parameters on the amount of corrupted triangles. The plots shows the average area portion covered by corrupt triangles for the CALB series.

The rigidity term has almost no impact on the mesh regularity. The distortion only slightly decreases with a higher rigidity value. In contrast, the impact of the force scaling on the mesh regularity is clearly noticeable. As the plot shows, a higher force scaling leads to less mesh distortion. Interestingly, the choice of the interpolation method, has only a minor impact on the outcome.

5.6 Discussion & Future Work

In this section, the results are evaluated and possible solutions for the encountered problems are discussed. The main challenges are the improvement of the performance and the stability of the mapping algorithm. Hence, issues concerning slow convergence are discussed and the matter of robustness is reconsidered. Finally, general improvements and possible extensions to the surface mapping and the rendering are proposed.

5.6.1 Convergence of the Mapping Algorithm

The parameter study in Section 5.5 has shown that the force weighting parameter μ has a strong impact on the overall convergence. The reason for that is the slow convergence of the movements caused by the internal forces. The subtle movement of one vertex is propagated through the entire mesh by the computation of the discrete Laplacian, causing even more subtle displacements that are, themselves, further propagated. However, the parameter study has also shown that putting a high weight on the internal forces is mainly necessary during the regularization process. Thus, alternatives for the initial regularization step should be considered. One possibility would be to generate the initial regular grid by using an mesh refinement approach, as e.g. suggested in [SP97]. Here, interleaved steps of displacement and refinement are executed to transform a simple initial shape (e.g. a sphere) to the target shape.

As mentioned before, the main reason for the slow convergence of the internal force is its global character. However, another issue is the discrete time steps in combination with the decomposition in a perpendicular and a tangential part. The main motivation for the tangential internal force was to keep the vertex from being pulled away from the target surface once it has reached it. However, when moving a vertex that is laying on the target surface according to the tangential internal force on a surface with a certain curvature, the vertex is necessarily moved away from the surface. When using a high global scale factor σ, this basically disrupts the convergence of the external force, since the vertex now has to move to the target surface once again. Since the external force is scaled down permanently, it becomes more and more difficult for the vertex to achieve this. Consequently, when computing the mapping over a large number of iterations, the result becomes more and more unstable. The effect can be diminished by using a very small value for the force scaling parameter σ (about 0.01). The instability of the mapping for different values of σ is illustrated in Figure 5.12. A scaling that small, however, leads to slow general convergence. Another possibility to deal with this issue

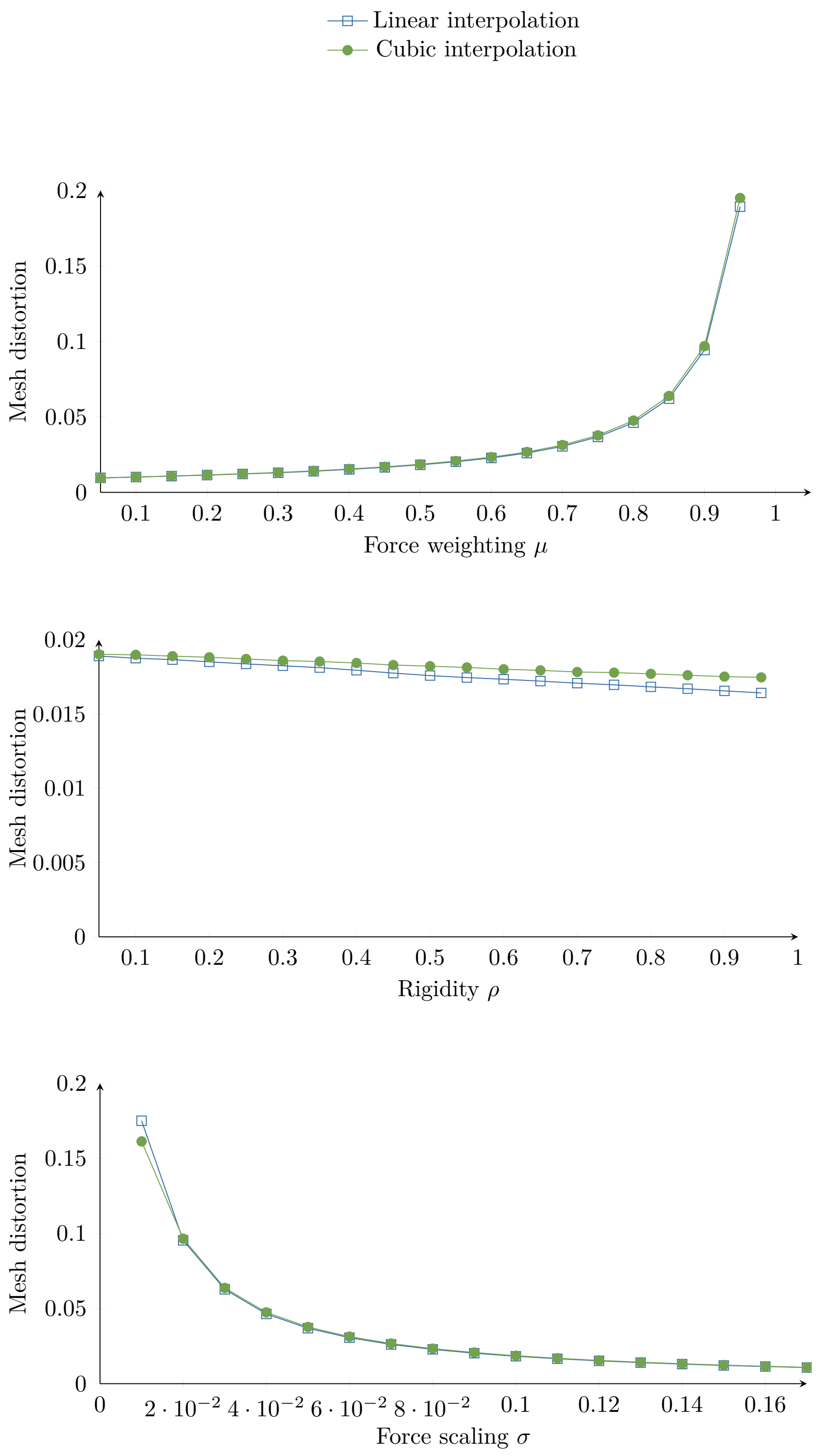


Figure 5.11: The impact of different parameters on the mesh regularity. The plots shows the average internal force present in the mesh after convergence all variants of the for the CALB series.

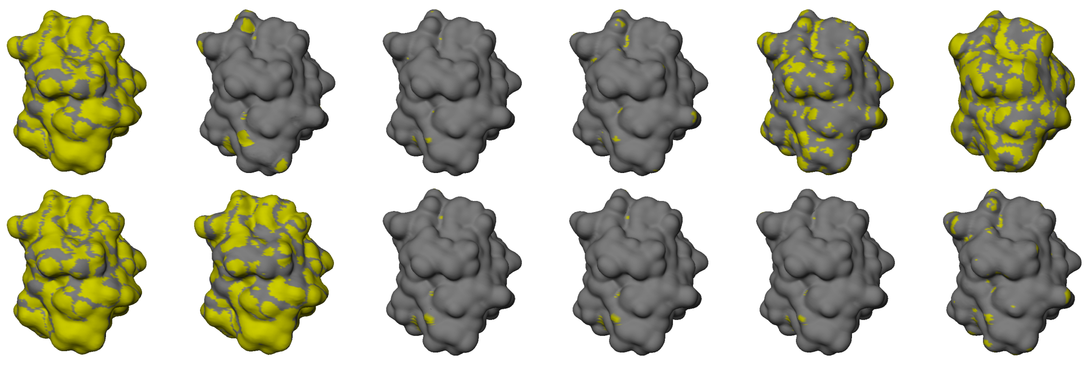
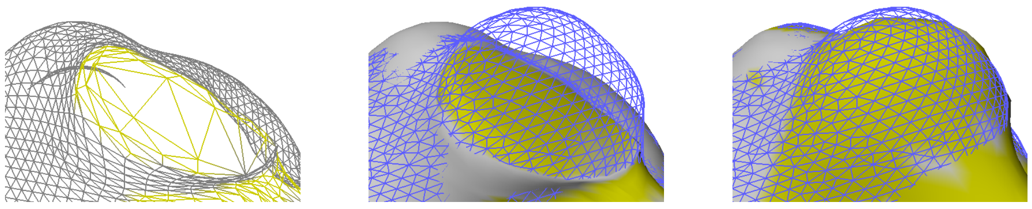


Figure 5.12: Instability of the mapping algorithm. The algorithm was executed over the respective number of iterations without applying the f_{min} criterion to stop the computation. The upper row shows the development of the surface for $\sigma = 0.2$, the lower row for $\sigma = 0.01$.

would be to take the local geometry (e.g. the curvature) of the surface into account to compute the displacement of the vertex, so that its new position lays accurately on the surface.

5.6.2 Robustness of the Mapping Algorithm

The mapping sometimes fails in cases where the target shape has large or complex concave regions and, at the same time, the source shape is very flat. In this situation, occasionally, the source shape converges to a state in which the bump or the cavity is not fully covered. This is a well-known issue of deformable models [YXSN09]. Although the external force in general leads to higher accuracy (as shown in Section 5.5), in this case, it locally leads to over-sized corrupted triangles. The reason for this issue is that the adaptive external forces seek to fixate the vertices at their current position as soon as they have reached the target surface. The result of this is the manifestation of the corrupt triangles in concave target regions, as illustrated in Figure 5.13. The problem can be diminished by using a lower weighted external force. This, however, will also generally lead to more corrupted triangles as shown in the parameter study. Not to treat the corrupt triangles would introduce an error to the metric described in Section 3.6, since the potential difference is normed by the surface area and the surface area, in this case, would be incorrect. In this thesis, the problem is dealt with in both the metric computation and the rendering. In the metric computation, corrupt triangles are omitted when integrating the scalar value over the surface area. In the 3D rendering, corrupt triangles are rendered transparent, since they are a source of uncertainty. Another option would be to use an adaptive mesh refinement during the mapping procedure. Here, a temporal subdividing of the huge triangles that got stuck in the concave region could be performed. The additional vertices would then pull the mesh into the concave region. As soon as the triangle is not stuck anymore, the additional vertex could be dismissed. Since no additional vertices would be introduced to the actual mesh, this would not affect the mapping relation.



(a) Over-sized triangles *(b) High external force* *(c) Low external force*

Figure 5.13: Failure of the mapping algorithm creating corrupt triangles. Back faces are culled for clarity. The target shape (shown in blue) contains a large concave region to which the source shape (grey/yellow) is not mapped correctly. The triangles in yellow are flagged as 'corrupt' by the algorithm. Lowering the weight of the external force causes the source shape to be pulled inside the concave region, but, at the same time, causes less accuracy in general (as can be seen by the increased number of corrupt triangles in (c)).

The additional vertices could, in fact, be seen as an additional temporal external force, similar to the global pressure force proposed in [Coh91].

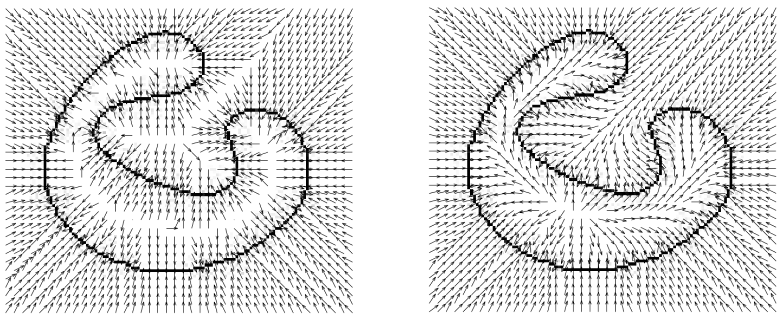
The second issue that occurs in the mapping is self-intersection of the mapped surface. The rigidity term in the internal energy function usually prevents self intersection of the surface. Nonetheless, during the mapping algorithm, self-intersection might occur if the source shape has a complex deep cavity and the target shape has not. The vertices associated with the cavity travel to the target surface according to the volume gradient. However, they are not distributed regularly, since the area they are mapped to is not sufficiently large. Depending on which way they take towards the target surface they can get 'stuck', which then leads to the self-intersection. Another case of self-intersection can happen during both the regularization process and the mapping algorithm. Here, the problem is the discretization by the Marching Tetrahedra in combination with very narrow cavities that can barely be captured by the used grid resolution. Sometimes, the vertices in such cavities the Marching Tetrahedra method extracts are closer to the opposite side then to the side they actually belong to. The external force then pulls them towards that side, which leads to the self-intersection of the surface. Here, using a smaller grid resolution clearly helps, but substantially increases the overall computation time since more vertices are produced. Using an adaptive refinement method mentioned for the initial triangulation could help as well.

The deformable model approach used in this thesis does not handle input shapes with different topology. In general, Lagrangian deformable models need extra modifications to be able to deal with complex or changing topology. One way to circumvent the problems associated with Lagrangian models is to use so-called level-set methods, which implicitly formulate the curve evolution as an Eulerian problem over the whole image domain. Those methods handle topology changes naturally, but are computationally more complex since the dimensionality of the problem statement is increased. Additionally, there are explicit deformable model approaches that can handle topology changes [MT00, MT99]. In this case, an implicit representation is no option, since it does not allow for the tracking of surface points. But even using a topology-adaptive explicit deformable model approach would not solve the issues caused by the different topology. After all, the purpose of the deformable model is to establish a mapping relation. The question, therefore, is how to establish this mapping between molecules whose surfaces have a different topology. There are several cases which have to be considered here. First, the input data sets contain a different number of molecules, e.g. caused by the presence of ligands. Here, the problem is basically solved by applying the RMSD computation and the subsequent mapping steps only to the subset of particles that is present in both data sets. The second case occurs, if one of the shapes contains one or more fully enclosed cavities that are not present in the other shape. Here, one solution would be to simply ignore the vertices forming the cavities not present on both molecules in the deformation process. There is, however, the possibility that cavities are present in both data sets while being closed in one data set and open in another data set. In this case, the current approach would actually deliver proper results, if the shape with the closed cavity would be mapped to the shape containing the open cavity. The triangles spanning over the opened cavity would be marked as corrupt and would, therefore, not be included in the final computation of the metric. If the source shape has an open cavity and the target has a closed cavity, this would most probably lead to self-intersection. Due to the complexity of the issue, it should be considered to handle cavities in the molecular surfaces separate from the rest of the surfaces.

One substantial problem with the approach used in this work is the underlying external energy function. The volume generation method used to generate the implicitly defined surfaces causes local extrema in the energy function. Additionally, the potential based on the Gaussian density distribution converges to zero very quickly, making it impossible to attract source shapes that start out far away from the target shape. In this work, both issues are dealt with by using a distance field in addition. Distance fields, however, cause problems in large cavities as stated in [XP98, GR03]. They solve some of the problems by introducing a new external force called *gradient vector flow (GVF)* (see Figure 5.14). Using a GFV would solve some problems encountered in this case, but would require additional computational effort, since it has to be computed iteratively.

5.6.3 General Improvements and Extensions to the Mapping

A possible matter for future work is to improve the semantic correspondence between the input shapes. This could be done e.g. by involving surface features like bumps or cavities in the initial rigid alignment. Furthermore, the RMSD could be computed solely based on the location of those features. This would be beneficial if molecules are compared that have a different structure to them but exhibit a similar surface shape and have similar surface features. Another possibility that is often used in shape registration is to involve additional user input to the computations. The user could for example execute the rigid alignment by hand or provide a connection between surface features that are semantically corresponding.



(a) Distance field *(b) GVF*

Figure 5.14: Illustration of problems caused by distance fields (from [GR03]). (a) shows that when using a distance field as external force, vertices cannot be pulled inside concave regions. This problem is solved when using the GVF instead.

At the moment the mapping method can not be executed if the RMSD value of both shapes too high. It can, however, happen that two input data sets containing the same molecule lead to a high RMSD value, if one of the structures is deformed or differently folded. A better treatment of such cases could be achieved by segmenting the structures into different parts and perform the computations involved in the mapping pairwise for those segments.

Several modifications to the computation of the difference metric could be thought of. The energy present in surface after the deformation could be seen as a measurement for uncertainty, since the more the source shape has to be deformed the higher the energy gets. Furthermore, it would be possible to concentrate all the computations involved in the metric on 'active' sites, e.g. possible binding sites for ligands, rather then the whole molecular surface.

5.6.4 General Improvements and Extensions to the Rendering

When using the surface rendering proposed in this work, the electrostatic potential can vary depending on where exactly the molecular surface is defined to be located. The visualization of the surface properties (such as the potential difference) could be extended to a certain area around the actual molecular surface rendering, e.g. by a ray casting approach. Then, however, a way would have to be found to deform the whole potential volume rather then just the surface, which would require substantial changes in the mapping approach. Another possibility is to render and blend a certain number of layers produced by the current approach to represent the area around the molecular surface (similar to [LV02]).

Another extension to the current visualization would be to render additional information about the data and the electrostatic field. This could for example be ligands or solvent molecules within a certain distance. In addition, interaction between the molecules could be represented by selected sparse field lines of the electrostatic field. An early prototype showing a reduced view of the electric field lines can be found in Figure 5.15. Here, streamline bundles are

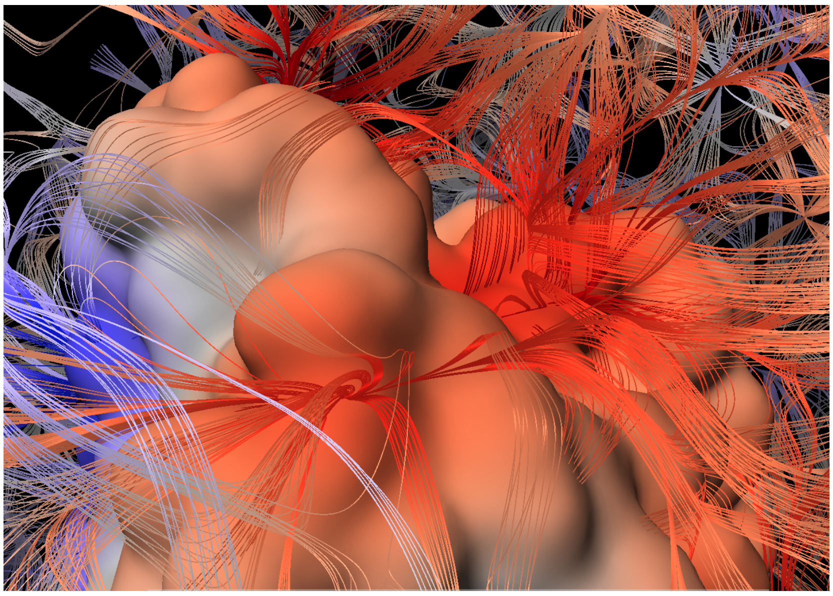


Figure 5.15: An early prototype showing field lines in combination with the surface rendering.

seeded based on saddle points that were extracted from the gradient field of the electric field magnitude.

6 Summary

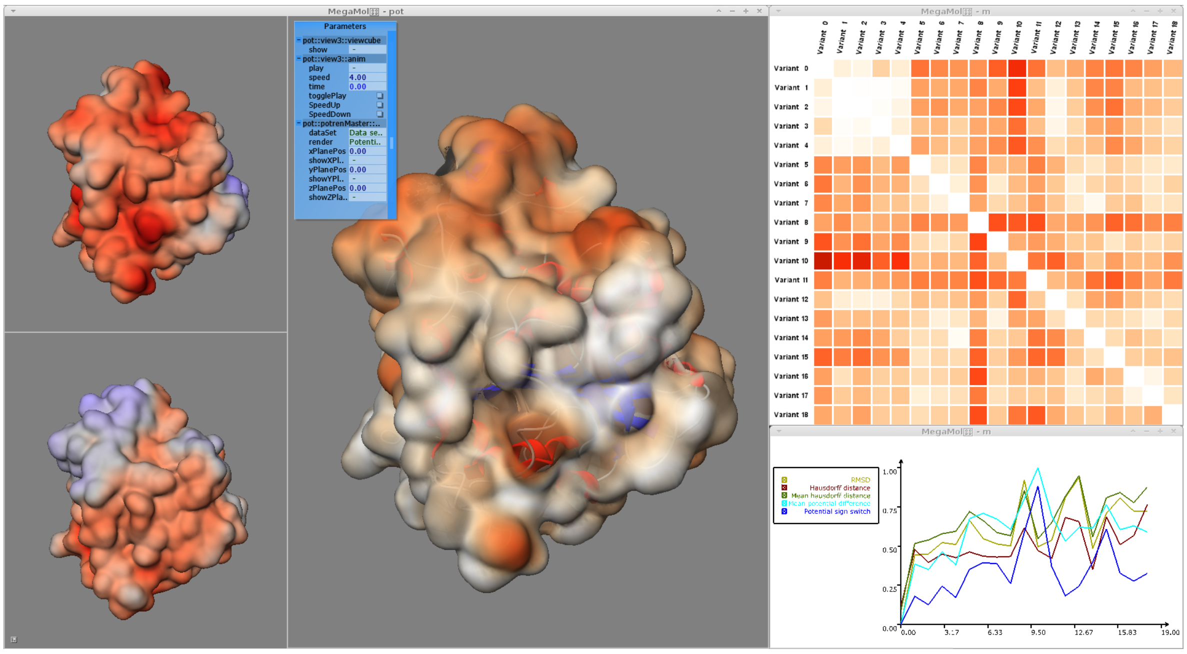
In this thesis a shape correspondence framework for molecular surfaces was developed. Using a deformable model approach, a mapping relation between two input molecular surfaces was established. This mapping relation was used for a comparative visualization of the electrostatic surface potential of two input molecules and to derive a difference metric that facilitates fast comparison of the electrostatic surface potential of a number of variants.

The input data consisted of particle data sets obtained from MD simulations. An implicit molecular surface representation was computed by representing each particle by a Gaussian density distribution, which – together with an appropriate isovalue – yielded an approximation to the molecular SES.

The mapping framework was applied to a source shape and a target shape and consists of several steps. First, the source shape is triangulated using the Marching Tetrahedra method. In order to minimize its internal energy, the resulting triangle mesh is regularized using internal spring forces and external forces based on its own underlying volume representation. The regularized mesh is then used as a basis for the actual mapping deformation. Based on internal and external forces, the source mesh is deformed until a stationary solution is found. Consequently, the discrete locations on the mapped source shape and respective locations on the original source shape define a mapping relation. The mapping relation allows comparing molecular surface features on a per-vertex basis. Here, the changes in electrostatic potential are quantified not only by their absolute difference, but also by their sign. Two difference metrics are computed describing the mean potential difference and the mean potential sign difference.

The framework was implemented using C/C++ and CUDA. To this end, a modified parallel version of the Marching Tetrahedra was implemented that – besides the triangulation – computes the connectivity information needed for the deformation process. The deformation was implemented in a multi-iteration CUDA kernel that executes several iterations of the dynamic force computation in a row, allowing for faster computation.

Based on the potential difference and the potential sign difference, a semi-transparent 3D surface rendering was implemented using OpenGL/GLSL. The mapped surface is rendered using a color map to represent differences in the electrostatic surface potentials of both shapes while changing the transparency according to the uncertainty of the mapping. This surface rendering is combined with an abstract cartoon-like rendering of the underlying molecular



Parameters

pot::view3::viewcube	
show	-
pot::view3::anim	
play	-
speed	4.00
time	0.00
togglePlay	
SpeedUp	
SpeedDown	
pot::potrenMaster::..	
dataSet	Data se..
render	Potenti..
xPlanePos	0.00
showXPl..	-
yPlanePos	0.00
showYPl..	-
zPlanePos	0.00
showZPla..	-

Figure 6.1: Summary of the application developed in this thesis.

structure to give additional visual clues. In addition to the 3D surface rendering, a heatmap-like 2D-plot rendering was implemented that summarizes the differences of an arbitrary number of input molecules and that can be used to give first hints about which pairs of molecules could be investigated further in the comparative 3D rendering. The screen-shot in Figure 6.1 summarizes the different rendering techniques used in this thesis.

Finally, the mapping algorithm and the comparative rendering were applied to real-world data sets. This revealed strengths and weaknesses of the current approach and provided a perspective about possible future improvements.

Bibliography

[ACP03] B. Allen, B. Curless, Z. Popović. The space of human body shapes: reconstruction and parameterization from range scans. *ACM Trans. Graph.*, 22(3):587–594, 2003. doi:10.1145/882262.882311. 6

[AFP00] M. A. Audette, F. P. Ferrie, T. M. Peters. An algorithmic overview of surface registration techniques for medical imaging. *Medical Image Analysis*, 4(3):201–217, 2000. doi:10.1016/S1361-8415(00)00014-1. 6

[Ale02] M. Alexa. Recent Advances in Mesh Morphing. *Computer Graphics Forum*, 21(2):173–198, 2002. doi:10.1111/1467-8659.00575. 6

[BBF$^+$11] S. Busking, C. P. Botha, L. Ferrarini, J. Milles, F. H. Post. Image-based rendering of intersecting surfaces for dynamic comparative visualization. *Vis. Comput.*, 27(5):347–363, 2011. doi:10.1007/s00371-010-0541-z. 5

[BKS$^+$05] B. Bustos, D. A. Keim, D. Saupe, T. Schreck, D. V. Vranić. Feature-based similarity search in 3D object databases. *ACM Comput. Surv.*, 37(4):345–387, 2005. doi:10.1145/1118890.1118893. 6

[Bli77] J. F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, 1977. doi:10.1145/965141.563893. 53

[Bli82] J. F. Blinn. A Generalization of Algebraic Surface Drawing. *ACM Trans. Graph.*, 1(3):235–256, 1982. doi:10.1145/357306.357310. 10, 19

[BM92] P. J. Besl, N. D. McKay. A Method for Registration of 3-D Shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):239–256, 1992. doi:10.1109/34.121791. 6

[BSW$^+$07] O. A. R. Board, D. Shreiner, M. Woo, J. Neider, T. Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1*. Addison-Wesley Professional, 6th edition, 2007. 28

[CC90] L. Cohen, I. Cohen. A finite element method applied to new active contour models and 3D reconstruction from cross sections. In *Computer Vision, 1990. Proceedings, Third International Conference on*, pp. 587–591. 1990. doi:10.1109/ICCV.1990.139601. 14

[CC93a] L. Cohen, I. Cohen. Finite-element methods for active contour models and balloons for 2D and 3D images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(11):1131–1147, 1993. doi:10.1109/34.244675. 14, 15

[CC93b] L. D. Cohen, I. Cohen. Finite-Element Methods for Active Contour Models and Balloons for 2-D and 3-D Images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(11):1131–1147, 1993. doi:10.1109/34.244675. 8, 15, 16

[CMS06] H. Carr, T. Moller, J. Snoeyink. Artifacts caused by simplicial subdivision. *Visualization and Computer Graphics, IEEE Transactions on*, 12(2):231–242, 2006. doi:10.1109/TVCG.2006.22. 13

[Coh91] L. D. Cohen. On active contour models and balloons. *CVGIP: Image Underst.*, 53(2):211–218, 1991. doi:10.1016/1049-9660(91)90028-N. 7, 19, 80

[COSL98] D. Cohen-Or, A. Solomovic, D. Levin. Three-dimensional distance field metamorphosis. *ACM Trans. Graph.*, 17(2):116–141, 1998. doi:10.1145/274363.274366. 20

[CRS98] P. Cignoni, C. Rocchini, R. Scopigno. Metro: Measuring Error on Simplified Surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998. doi:10.1111/1467-8659.00236. 6, 22

[CSD04] E. A. Coutsias, C. Seok, K. A. Dill. Using quaternions to calculate RMSD. *Journal of Computational Chemistry*, 25(15):1849–1857, 2004. doi:10.1002/jcc.20110. 11

[DM00] H. Delingette, J. Montagnat. New Algorithms for Controlling Active Contours Shape and Topology. In *Proceedings of the 6th European Conference on Computer Vision-Part II*, ECCV '00, pp. 381–395. Springer-Verlag, London, UK, UK, 2000. doi:10.1007/3-540-45053-X_25. 8

[Eri05] C. Ericson. *Real-Time Collision Detection*. Number Bd. 1 in Real-time Collision Detection. Elsevier/Morgan Kaufmann Publishers, 2005. 20, 22

[FBM02] F. Fogolari, A. Brigo, H. Molinari. The Poisson–Boltzmann equation for biomolecular electrostatics: a tool for structural biology. *J. Mol. Recognit.*, 15(6):377–392, 2002. doi:10.1002/jmr.577. 52

[FH09] R. Fuchs, H. Hauser. Visualization of Multi-variate Scientific Data. *Computer Graphics Forum*, 28(6):1670–1690, 2009. doi:j.1467-8659.2009.01429.x. 5

[FS01] D. Frenkel, B. Smit. *Understanding Molecular Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2001. 1

[GR03] D. Gil, P. Radeva. Curvature Vector Flow to Assure Convergent Deformable Models for Shape Modelling. In *In EMMCVPR*, pp. 357–372. 2003. doi:10.1007/978-3-540-45063-4_23. 8, 81, 82

[GRE12] S. Grottel, G. Reina, T. Ertl. MegaMol™: A Visualization Middleware for Point-based Molecular Data Sets, 2012. `http://www.vis.uni-stuttgart.de/megamol`. 54

[HJ05] C. Hansen, C. Johnson. *The Visualization Handbook*. Referex Engineering. Elsevier-Butterworth Heinemann, 2005. 12, 13

[HKSL08] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. *Journal of Chemical Theory and Computation*, 4(3):435–447, 2008. doi:10.1021/ct700301q. 59

[HMS56] P. C. Hammer, O. P. Marlowe, A. H. Stroud. Numerical integration over simplexes and cones. *Math. Tables Aids Comp.*, 10(55):130–137, 1956. doi:10.2307/2002483. 24, 25

[HN95] B. Honig, A. Nicholls. Classical electrostatics in biology and chemistry. *Science*, 268(5214):1144–1149, 1995. doi:10.2307/2888372. 2

[HPM06] X. Huang, N. Paragios, D. Metaxas. Shape registration in implicit spaces using information theory and free form deformations. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(8):1303–1318, 2006. doi:10.1109/TPAMI.2006.171. 6

[IJL+05] N. Iyer, S. Jayanti, K. Lou, Y. Kalyanaraman, K. Ramani. Three-dimensional shape searching: state-of-the-art review and future trends. *Comput. Aided Des.*, 37(5):509–530, 2005. doi:10.1016/j.cad.2004.07.002. 6

[Kab76] W. Kabsch. A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A*, 32(5):922–923, 1976. doi:10.1107/S0567739476001873. 6, 11, 27

[Kab78] W. Kabsch. A discussion of the solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A*, 34(5):827–828, 1978. doi:10.1107/S0567739478001680. 11, 27

[KH13] J. Kehrer, H. Hauser. Visualization and Visual Analysis of Multifaceted Scientific Data: A Survey. *Visualization and Computer Graphics, IEEE Transactions on*, 19(3):495–513, 2013. doi:10.1109/TVCG.2012.110. 1, 5

[KN03] K. Kinoshita, H. Nakamura. Identification of protein biochemical functions by similarity search using the molecular surface database eF-site. *Protein Science*, 12(8):1589–1595, 2003. doi:10.1110/ps.0368703. 1, 7

[KN10] P. Kukić, J. E. Nielsen. Electrostatics in proteins and protein-ligand complexes. *Future Med Chem*, 2(4):647–66, 2010. doi:10.4155/fmc. 2

[KSES12] M. Krone, J. E. Stone, T. Ertl, K. Schulten. Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories. In *EuroVis 2012 Short Papers*, volume 1. 2012. 27

[KTZ92] B. B. Kimia, A. Tannenbaum, S. W. Zucker. On the evolution of curves via a function of curvature. I. The classical case. *Journal of Mathematical Analysis and Applications*, 163(2):438 – 458, 1992. doi:10.1016/0022-247X(92)90260-K. 8

[KWP01] K. Kim, C. M. Wittenbrink, A. Pang. Data level comparison of surface classification and gradient filters. In *Proceedings of the 2001 Eurographics conference on Volume Graphics*, VG'01, pp. 17–33. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2001. doi:10.2312/VG/VG01/017-033. 5

[KWT88] M. Kass, A. Witkin, D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1988. doi:10.1007/BF00133570. 7

[KZHCO11] O. van Kaick, H. Zhang, G. Hamarneh, D. Cohen-Or. A Survey on Shape Correspondence. *Computer Graphics Forum*, 30(6):1681–1707, 2011. doi:10.1111/j.1467-8659.2011.01884.x. 6

[LC87] W. E. Lorensen, H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987. doi:10.1145/37402.37422. 12

[LKE00] C. Lürig, L. Kobbelt, T. Ertl. Hierarchical Solutions for the Deformable Surface Problem in Visualization. *Graphical Models*, 62:2000, 2000. doi:10.1006/gmod.1999.0515. 15, 16

[LPPW95] W. C. D. Leeuw, H.-G. Pagendarm, F. H. Post, B. Walter. Visual Simulation of Experimental Oil-Flow Visualization by Spot Noise Images from Numerical Flow Simulation. In *In Visualization in Scientific Computing '95*, pp. 135–148. Springer Verlag, 1995. doi:10.1007/978-3-7091-9425-6_13. 5

[LSP08] H. Li, R. W. Sumner, M. Pauly. Global correspondence optimization for non-rigid registration of depth scans. In *Proceedings of the Symposium on Geometry Processing*, SGP '08, pp. 1421–1430. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2008. 6

[LSVMW07] W. Liu, B. Schmidt, G. Voss, W. Müller-Wittig. Molecular dynamics simulations on commodity GPUs with CUDA. In *Proceedings of the 14th international conference on High performance computing*, HiPC'07, pp. 185–196. Springer-Verlag, Berlin, Heidelberg, 2007. 2

[LV98] F. Lazarus, A. Verroust. Three-dimensional metamorphosis: a survey. *The Visual Computer*, 14(8-9):373–389, 1998. doi:10.1007/s003710050149. 6

[LV02] C. H. Lee, A. Varshney. Representing Thermal Vibrations and Uncertainty in Molecular Surfaces. In *Proceedings SPIE Conference on Visualization and Data Analysis*. January 20 - 25, 2002. doi:10.1117/12.458813. 82

[MC94] V. N. Maiorov, G. M. Crippen. Significance of Root-Mean-Square Deviation in Comparing Three-dimensional Structures of Globular Proteins. *Journal of Molecular Biology*, 235(2):625–634, 1994. doi:http://dx.doi.org/10.1006/jmbi.1994.1017. 11

[MDA01] J. Montagnat, H. Delingette, N. Ayache. A review of deformable surfaces: topology, geometry and deformation. *Image and Vision Computing*, 19(14):1023–1040, 2001. doi:10.1016/S0262-8856(01)00064-6. 7, 15

[MGE07] C. Müller, S. Grottel, T. Ertl. Image-space GPU metaballs for time-dependent particle data sets. In *In Vision, Modelling and Visualization (VMV '07) (2007*, pp. 31–40. 2007. 10

[Mor09] K. Moreland. Diverging Color Maps for Scientific Visualization. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II*, ISVC '09, pp. 92–103. Springer-Verlag, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-10520-3_9. 52

[MT96] T. Mcinerney, D. Terzopoulos. Deformable models in medical image analysis: A survey. *Medical Image Analysis*, 1:91–108, 1996. doi:10.1016/S1361-8415(96)80007-7. 7, 15

[MT99] T. McInemey, D. Terzopoulos. Topology adaptive deformable surfaces for medical image volume segmentation. *Medical Imaging, IEEE Transactions on*, 18(10):840–850, 1999. doi:10.1109/42.811261. 81

[MT00] T. McInerney, D. Terzopoulos. T-snakes: Topology adaptive snakes. *Medical Image Analysis*, 4(2):73–91, 2000. doi:10.1016/S1361-8415(00)00008-6. 81

[Mö97] T. Möller. A Fast Triangle-Triangle Intersection Test. *Journal of Graphics Tools*, 2:25–30, 1997. doi:10.1080/10867651.1997.10487472. 22

[NH91] G. M. Nielson, B. Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *Proceedings of the 2nd conference on Visualization '91*, VIS '91, pp. 83–91. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991. doi:10.1109/VISUAL.1991.175782. 12

[NPP03] M. T. Neves-Petersen, S. B. Petersen. Protein electrostatics: A review of the equations and methods used to model electrostatic equations in biomolecules – Applications in biotechnology. *Biotechnology Annual Review*, 9:315–395, 2003. doi:10.1016/S1387-2656(03)09010-0. 2

[NSCE02] A. N, D. Santa-Cruz, T. Ebrahimi. MESH: measuring errors between surfaces using the Hausdorff distance. In *Multimedia and Expo, 2002. ICME '02. Proceedings. 2002 IEEE International Conference on*, volume 1, pp. 705–708. 2002. doi:10.1109/ICME.2002.1035879. 6, 22, 23, 25

[Nvi12] Nvidia. *CUDA C Programming Guide*, 2012. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`. 30

[OS88] S. Osher, J. A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988. doi:10.1016/0021-9991(88)90002-2. 7

[PF96] A. Pang, A. Freeman. Methods for Comparing 3D Surface Attributes. In *In SPIE Vol. 2656 Visual Data Exploration and Analysis III*, pp. 58–64. 1996. doi:10.1117/12.234691. 5

[Pho75] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975. doi:10.1145/360825.360839. 53

[PP95] H.-G. Pagendarm, F. Post. Comparative Visualization - Approaches and Examples. In *Proceedings of 5th Eurographics Workshop on Visualization in Scientific Computing*, pp. 95–108. 1995. 1, 5

[PS09] N. Postarnakevich, R. Singh. Global-to-local representation and visualization of molecular surfaces using deformable models. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pp. 782–787. ACM, New York, NY, USA, 2009. doi:10.1145/1529282.1529446. 7

[QZS$^+$04] H. Qu, N. Zhang, R. Shao, A. Kaufman, K. Mueller. Feature preserving distance fields. In *Volume Visualization and Graphics, 2004 IEEE Symposium on*, pp. 39–46. 2004.

[RBG$^+$09] M. Reuter, S. Biasotti, D. Giorgi, G. Patanè, M. Spagnuolo. Discrete Laplace–Beltrami operators for shape analysis and segmentation. *Computers & Graphics*, 33(3):381–390, 2009. doi:http://dx.doi.org/10.1016/j.cag.2009.03.005. 16

[Ric77] F. M. Richards. Areas, Volumes, Packing, and Protein Structure. *Annual Review of Biophysics and Bioengineering*, 6(1):151–176, 1977. doi:10.1146/annurev.bb.06.060177.001055. 10

[RLKG$^+$09] R. J. Rost, B. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, M. Weiblen. *OpenGL Shading Language*. Addison-Wesley Professional, 3rd edition, 2009. 28, 29

[RLWN98] M. Rosen, S. L. Lin, H. Wolfson, R. Nussinov. Molecular shape comparisons in searches for active sites and functional similarity. *Protein Engineering*, 11(4):263–277, 1998. doi:10.1093/protein/11.4.263. 7

[RR73] S. Rao, M. G. Rossmann. Comparison of super-secondary structures in proteins. *Journal of Molecular Biology*, 76(2):241–256, 1973. doi:http://dx.doi.org/10.1016/0022-2836(73)90388-4. 11

[Sch10] T. Schlick. Molecular Dynamics: Basics. In *Molecular Modeling and Simulation: An Interdisciplinary Guide*, volume 21 of *Interdisciplinary Applied Mathematics*, pp. 425–461. Springer New York, 2010. 1, 2

[SCP12] A. Sotiras, D. Christos, N. Paragios. Deformable Medical Image Registration: A Survey. Research Report RR-7919, INRIA, 2012. doi:10.1109/TMI.2013.2265603. 6

[SHL+11] T. Shen, X. Huang, H. Li, E. Kim, S. Zhang, J. Huang. A 3D Laplacian-driven parametric deformable model. In D. N. Metaxas, L. Quan, A. Sanfeliu, L. J. V. Gool, editors, *ICCV*, pp. 279–286. IEEE, 2011. doi:10.1109/ICCV.2011.6126253. 8, 16, 17

[SLH+10] D. van der Spoel, E. Lindahl, B. Hess, A. R. van Buuren, P. J. M. E. Apol, D. P. Tieleman, A. L. T. M. Sijbers, K. A. Feenstra, R. van Drunen, H. J. C. Berendsen. *Gromacs User Manual version 4.5.4*, 2010. `www.gromacs.org`. 1, 2

[SNH00] F. B. Sheinerman, R. Norel, B. Honig. Electrostatic aspects of protein–protein interactions. *Current Opinion in Structural Biology*, 10(2):153–159, 2000. doi:10.1016/S0959-440X(00)00065-8. 2

[SP97] I. A. Sadarjoen, F. H. Post. Deformable Surface Techniques for Field Visualization. *Comput. Graph. Forum*, 16(3):109–116, 1997. doi:10.1111/1467-8659.00147. 77

[SPF+07] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007. doi:10.1002/jcc.20829. 2

[Sto03] M. Stone. *A Field Guide to Digital Color*. Ak Peters Series. Peters, 2003. 52

[TF88] D. Terzopoulos, K. Fleischer. Deformable Models. *The Visual Computer*, 4(6):306–331, 1988. doi:10.1007/BF01908877. 7, 16

[TP96] J. Trapp, H.-G. Pagendarm. Data level comparative visualization in aircraft design. In *Visualization '96. Proceedings.*, pp. 393–396. 1996.

[TPBF87] D. Terzopoulos, J. Platt, A. Barr, K. Fleischer. Elastically deformable models. *SIGGRAPH Comput. Graph.*, 21(4):205–214, 1987. doi:10.1145/37402.37427. 7

[TV08] J. W. Tangelder, R. C. Veltkamp. A survey of content based 3D shape retrieval methods. *Multimedia Tools Appl.*, 39(3):441–471, 2008. doi:10.1007/s11042-007-0181-0. 6

[TW99] M. Tigges, B. Wyvill. A field interpolated texture mapping algorithm for skeletal implicit surfaces. In *Computer Graphics International, 1999. Proceedings*, pp. 25–32, 240. 1999. doi:10.1109/CGI.1999.777896. 7

[TWK88] D. Terzopoulos, A. Witkin, M. Kass. Constraints on deformable models: recovering 3D shape and nongrid motion. *Artif. Intell.*, 36(1):91–123, 1988. 7

[VH01] R. Veltkamp, M. Hagedoorn. State of the Art in Shape Matching. In M. Lew, editor, *Principles of Visual Information Retrieval*, Advances in Pattern Recognition, pp. 87–119. Springer London, 2001. doi:10.1007/978-1-4471-3702-3_4. 6

[VP04] V. Verma, A. Pang. Comparative flow visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 10(6):609–624, 2004. doi:10.1109/TVCG.2004.39. 5

[WS00] G. Wyszecki, W. S. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae (Wiley Series in Pure and Applied Optics)*. Wiley-Interscience, 2 edition, 2000. 52

[WS06] J. Woodring, H.-W. Shen. Multi-variate, Time Varying, and Comparative Visualization with Contextual Cues. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):909–916, 2006. doi:10.1109/TVCG.2006.164. 5

[XP98] C. Xu, J. Prince. Snakes, shapes, and gradient vector flow. *Image Processing, IEEE Transactions on*, 7(3):359–369, 1998. 81

[XPP00] C. Xu, D. L. Pham, J. L. Prince. Image segmentation using deformable models. In *Handbook of Medical Imaging. Vol.2 Medical Image Processing and Analysis*, pp. 175–272. 2000. doi:10.1117/3.831079.ch3. 15

[YXSN09] S. Y. Yeo, X. Xie, I. Sazonov, P. Nithiarasu. Geometric Potential Force for the Deformable Model. In *BMVC*. British Machine Vision Association, 2009. doi:10.5244/C.23.99. 79

[ZGVF98] R. Zonenschein, J. Gomes, L. Velho, L. H. de Figueiredo. Controlling texture mapping onto implicit surfaces with particle systems. In *Proceedings of 3rd Eurographics Workshop on Implicit Surfaces*, pp. 131–138. 1998. 6

[ZSCO$^+$08] H. Zhang, A. Sheffer, D. Cohen-Or, Q. Zhou, O. Van Kaick, A. Tagliasacchi. Deformation-Driven Shape Correspondence. *Computer Graphics Forum*, 27(5):1431–1439, 2008. doi:10.1111/j.1467-8659.2008.01283.x. 23

All links were last followed on 29 July 2013.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

____

(Katrin Scharnowski)