Interactive Constraint-Based Modeling

Rolf Weilenmann

Master Thesis
Swiss Federal Institute of Technology Zürich
August 2009

Advisor: Michael Eigensatz
Professor: Prof. Dr. Mark Pauly
Abstract

In this thesis we extend the surface methodology developed by Eigensatz [1][2] and co-workers to be able to modify surface properties in a more interactive manner. Opposed to traditional handle or flow based modeling metaphors, where one would solve for the deformed surface, approximating the positional constraints while preserving important properties of the original shape, the work of Eigensatz allows direct access to first and second order properties of a surface. Among several filters [3] for the principal curvatures, the constraint-based modeling approach allows also to prescribe properties along curves embedded in surfaces, such as path-lengths or prescribing normal curvatures along curves. Additional constraints also deal with the area of surface paths or simple positional constraints defined anywhere on the surface. Solving for the intuitively deformed shape is still computationally very expensive, which we counteract by heavily exploiting multi-core parallelism. While these optimizations are boosting the performance of the underlying non-linear optimization pipeline from minutes to seconds, and for smaller problem sizes (<5k meshes) even below a second, we permit to modify target values and optimization weights of the constraints interactively. The advantage of an immediate feedback is also accompanied by smaller deformations and better convergence as the user adjusts the parameters to fit the modeling intent. Lastly the entire setting is mapped in a 2D scenario, where we apply and extend the constraint-based modeling metaphor for manipulating images. The implementation is incorporated into the software Dolly developed by Eigensatz.
Interactive Constraint-Based Modeling

Advisor: Michael Eigensatz, Professor: Mark Pauly

Motivation

Effective algorithms for surface deformation are of central importance in digital geometry processing. One of the most popular interaction metaphors allows the user to select subsets of the model as a control handle and specify an affine transformation for each handle region. The deformed surface is then computed such that the resulting positional constraints are satisfied, while preserving important properties of the original shape. While intuitive and easy-to-learn, certain tasks remain difficult to achieve with handle-based interaction. For example, preserving or explicitly modifying first or second order properties of the surface, such as lengths, areas, or curvature, is cumbersome when deforming a shape by specifying positional constraints only. In our research we investigate techniques to enable surface editing by specifying constraints on such derived surface properties. There are usually many possible ways to satisfy these constraints. Thus we are particularly interested in finding intuitive parameters for shape preservation to provide active and flexible control on the deformation outcome. A major problem of existing techniques is that they are computationally too expensive to be interactive.

Tasks

The goal of this project is to improve and simplify the constraint-based modeling techniques presented in [1] such that comparable operations can be performed in an interactive manner. The thesis will consist of the following tasks:

- Understand the algorithmic framework described in [1].
- Perform an exhaustive performance and memory profiling to evaluate the parts with the highest potential for improvement.
- Improve the efficiency of the surface preservation energies using parallelisation.
- Apply multi-resolution strategies to increase the robustness and efficiency of surface preservation and curvature-domain shape processing.
- If the above measures to reduce computation time are sufficient to enable interactive deformation rates, editing constraints (e.g. curve lengths) can be gradually added to build an interactive system for surface deformation. If time allows, novel constraints can be explored to broaden the constraint-based modeling toolset. If, on the other hand, the optimization is still too slow, new techniques to describe the shape preservation will have to be investigated.

Schedule

Begin Date: 23.2.2009, Submission Date: 22.8.2009

References

# Contents

List of Figures ix

1. Introduction 1
   1.1. Related Work 2
   1.2. Overview 4

2. Constraint-Based Modeling 5
   2.1. Notation 5
   2.2. Problem Formulation 6
   2.3. Nonlinear Mesh Energy Formulations 7
      2.3.1. Metric Energies 8
      2.3.2. Principal Curvature Energy 9
      2.3.3. Positional Energy 10
      2.3.4. Path Length Energy 10
      2.3.5. Areal Energy 11
      2.3.6. Normal Curvature Energy 11
   2.4. Minimization Process 13
   2.5. Overview with an Editing Example 14

3. Parallelization and Optimizations 19
   3.1. Overview 19
   3.2. Terminology 20
   3.3. OpenMP 21
      3.3.1. Introduction 21
      3.3.2. Usage 21
      3.3.3. Performance 22
## 4. Parallelization and Optimization for Constraint-Based Mesh Modeling

4.1. Porting the Global Energies to CUDA ................................. 27
4.2. Assemble the Linearized System of Equations ............................. 31
   4.2.1. Sparse Matrix Representations ..................................... 31
   4.2.2. Jacobi-Matrix ....................................................... 33
   4.2.3. Building the Normal Equation ..................................... 33
   4.2.4. Solving the Linearized System of Equations ........................ 35
4.3. Performance Results ......................................................... 35
   4.3.1. Comparison with the Single-threaded Framework ...................... 36
   4.3.2. Detail Investigations ............................................... 40
4.4. Discussion and Outlook ..................................................... 40

## 5. 2D-Setting

5.1. New 2D-Constraints .......................................................... 43
   5.1.1. Path positional constraints ......................................... 43
   5.1.2. Path Angle Constraints ............................................. 44
   5.1.3. Path Segment Length ............................................... 45
5.2. Interactive Weight Control .................................................. 45
5.3. 2D-Constraint-Based Mesh Modeling ....................................... 45
   5.3.1. 2D-Shape Modeling .................................................. 46
   5.3.2. 2D-Embedded Shape Modeling ...................................... 47
5.4. Discussion and Outlook ...................................................... 51

## 6. Conclusion And Outlook

A. Appendix

A.1. Computer System ............................................................ 55
A.2. OpenMP ................................................................. 55
   A.2.1. Performance Analysis .............................................. 55
A.3. CUDA ................................................................. 58
   A.3.1. Used Hardware ..................................................... 58
   A.3.2. Performance Analysis .............................................. 58
   A.3.3. CUDA Memory Layout .............................................. 60
   A.3.4. CUDA-Energies Detail Analysis ................................... 61
A.4. Path Angle Derivatives ..................................................... 62
A.5. Gui Components ............................................................ 62
   A.5.1. Interface ............................................................ 62
   A.5.2. Mesh Energy Visualization ......................................... 64
   A.5.3. Texture ............................................................. 64
   A.5.4. Tools .............................................................. 65
A.6. Miscellaneous ............................................................... 66
   A.6.1. Generation of an Equilateral Triangulation ........................ 66
List of Figures

1.1. Horse multiple constraints ................................................. 2
1.2. Dining Room ................................................................. 2

2.1. Crude system overview ..................................................... 7
2.2. Local Parametrization ...................................................... 8
2.3. Curvature editing ............................................................ 11
2.4. Weak positional constraint editing ..................................... 12
2.5. Path length editing .......................................................... 12
2.6. Area constraint editing .................................................... 12
2.7. Normal curvature editing ................................................. 13
2.8. Coarse Sphere Example .................................................... 15
2.9. Sphere jacobian .............................................................. 16
2.10. Sphere j^t and chol. decomp. .......................................... 16

3.1. CPUGPU ................................................................. 20
3.2. OpenMPModel ............................................................... 21
3.3. Chip architecture .......................................................... 24
3.4. CUDA Architecture ....................................................... 24

4.1. Host-Device computation paths ........................................ 30
4.2. Objective function performance plot .................................. 37
4.3. Objective function performance detail plot ......................... 37
4.4. Assembly and solver performance plot .............................. 38
4.5. Assembly performance plot detail .................................... 38
4.6. Overall performance plot .............................................. 39
4.7. Overall performance plot detail ....................................... 39
### List of Figures

5.1. Path angle description ................................................. 44  
5.2. Path angle gradients ................................................. 44  
5.3. Path straightness ..................................................... 45  
5.4. As-rigid-as-possible man ............................................ 46  
5.5. As-rigid-as-possible man non-uniform weights .................. 46  
5.6. Dining Room wide-angle example ................................... 48  
5.7. Dining Room from mercator projection ............................ 48  
5.8. Living Room example .................................................. 48  
5.9. 2D-Area constraints .................................................. 50  
5.10. 2D-Path angle constraints .......................................... 50  
5.11. 2D-Path path length constraint .................................... 50  
5.12. 2D-Path path positional constraint ................................. 50  

A.1. OpenMP times of simple functions .................................. 57  
A.2. OpenMP speedups of simple functions .............................. 57  
A.3. CUDA times of simple functions .................................... 59  
A.4. CUDA speedups of simple functions ................................. 59  
A.5. CUDA Memory Layout ............................................... 60  
A.6. 3D CBM Interface ..................................................... 63  
A.7. 2D CBM Interface ..................................................... 64  
A.8. Point Dragger Interface ............................................... 65  
A.9. Interactive Weights Interface ....................................... 65  
A.10. Equilateral Triangulation ............................................ 66
Introduction

Mesh deformation and generation tools are extremely useful in the computer graphics field. Usually one has to model a 3D object with well known modeling software or use a scanner of some sort to acquire a model. Knowing that developing a fairly complex shape from scratch is often cumbersome, especially for people not using these tools everyday, it is often more convenient to edit and deform a given mesh or patch to suit ones needs. That is where surface-based mesh editing, and the constraint-based modeling approach [2] in particular, comes into play. In contrast to other editing metaphors, shortly presented in the related work section (1.1), the framework lets the user define constraints on the surface of a given mesh, and solves a global non-linear optimization problem in the least squares sense. Allowing direct access to first and second order properties of a mesh, such as lengths, areas and curvatures enables the user to modify surface properties that are usually hard to manipulate or preserve in an other setting. Another application of curvature-domain shape processing is in the area of curvature filtering. Similarly to image processing, filters are defined (e.g. mean- or bilateral filtering) for curvatures and will so optimize the surface with respect to the filtered curvatures.

Continuing the work from my advisor Michael Eigensatz [1][2] and Sandro Löschhorn [3], a former master student of his, allowed me to directly start from a working system. Besides the striking and very broad abilities of the constraint-based modeling metaphor, the modeling experience is greatly affected by the responsiveness of the system. Overall optimization times ranging from seconds up to minutes were common. In the following, we investigated approaches to speed up the computationally very involved optimization process. One way to achieve interactive performance at least for small meshes (<5k vertices) is the extensive usage of multi-core architectures and other optimizations concerning the assemblage of the sparse (linearized) system of equations resulting from the constraints. We achieve drastic performance gain from these parallelizations and optimizations, also for larger meshes. But mainly the non-linear formulation with its iterated use of the in-core direct solver, as the new bottleneck, impedes real interactivity.
1. Introduction

Fast visual feedback is also important since the shape is deformed based on some weighted constraints; controlling their influence interactively (also locally) in the optimization process therefore allows for better control of the desired output. While the user modifies the parameters on-the-fly, the deviation of the deformations will most likely be small compared to the previous obtained shape. The effects of using the previous result as a start-vector in the optimization pipeline thus leads to a faster, more stable convergence and fewer iterations needed.

As a practical extension we show how to map the formulation to a 2D planar mesh. The planar setting has the advantage, that the energy formulations are simpler and will at least reduce the complexity by a third. Attaching a texture rigidly to the original vertex texture coordinates will directly link to interesting new image manipulation tools. Conceptually we are interested in an intuitive mapping which warps the image approximating the modeling constraints such as straightness of lines, or other constraints already present in the more general 3D scenario under retention of locally definable rigidity.

Figure 1.1.: A cartoon horse with increased belly, hips and transformed head.

Figure 1.2.: 2D image manipulation: straighten out the shown lines.

1.1. Related Work

Since mesh manipulation and editing are important operations to model a shape, there is a long list of related work. An attempt to give a short overview of the related geometry-based mesh deformation techniques follows, not covering physically-based deformations, which are beyond the scope of this thesis.

One of the earliest approaches is the Free Form Deformation technique [4], which embeds a surface into a volume lattice. The (often hard to control) deformation is archived by manipulating the control points of the volume lattice. More recent improvements have been made by replacing the lattice by a radial basis function (RBF) [5]. While fixing the handles to pre-defined regions, most of the involved computations can be precomputed and can therefore run interactively even for large meshes. More intuitive detail preserving approaches compute a multi-resolution hierarchy by decomposing the mesh in a (often coarsened) base mesh, encoding the low frequencies of a shape, and a detail vector, encoding the high frequencies details.
1.1. Related Work

By editing the base mesh at an appropriate resolution allows the details to move accordingly; the interested is referred to [6][7][8] for further details. Further research by [9][10][11][12][13] have shown ways to archive similar results not revealing the multi-resolution setup. Since computation complexity is the limiting factor in editing large meshes, there are several methods to overcome this bottleneck, such as multilevel [12] or multi-grid methods [14]. Methods for mesh deformations often rely on differential coordinates (such as laplacian or gradient coordinates), because they are among other features affine transformation invariant. The deformations are expressed in modifications of the differential coordinates from which the vertex positions are reconstructed. A method to achieve intuitive and robust deformations is presented by Botsch et al. [15], by embedding a surface mesh in a layer of volumetric prisms and minimizing (non-linear) elastic energy (non-linear). Similar to this approach the formulation has been extended to coupled volumetric cells, where scalability is achieved by an adaptive spatial discretization [16]. The work of of Sumner et al. [17] addresses embedded deformations, also using non-linear positional constraint energy minimization, where the space deformation is represented by a collection of affine transformations organized in a graph structure. Their embedded approach allows to deform objects, such as (animated) meshes, polygon soups or particles with comparable results to the previous mentioned deformation methods.

In the context of the related surface optimization field, the most common methods are based on energy minimizing flows. A more complete overview and relation to the Curvature-Domain Shape Processing is found in the publication of Eigensatz et al. [1].

Considering the 2D setting, the existing related and recent publications can roughly be split into two categories. One deforms the space in which a shape is embedded (e.g. a shape on a squared image), and the other deforms a 2D representation of the shape itself (e.g. an triangle mesh enriched with a texture). While being closer to the 3D deformations, we will first consider the second case (less general; usually a specialization of the 3D case) and request similar to above that the deformation is performed in an intuitive manner. The concept of “as-rigid-as-possible” transformations introduced by Alexa et al. [18] has become popular in the work from Igarashi et al. [19]. While the user selects some handle vertices, their algorithms performs a two-step (rotation/scale) closed-form minimization satisfying the constraints while minimizing the distortion of each triangle. An extension to this work is presented by Schaefer et al. [20] where they extend the handle interface to sets of points and line segments. To construct the deformations they are using a moving least squares approach, while restricting the amount of local shearing and scaling to similarity and rigid-body transformations. The resulting deformation function is applied to a grid, warping the embedded shape (with possible “fold-backs”). This more general concept of warping the image space is for instance also used in the work of Wang et al. [21] in the context of image resizing (scale and stretch), where they preserve visually prominent features using a automatically computed gradient- and saliency-map. Content dependent non-linear constraints are also found in the recent work of Carroll et al. [22], where they apply a smooth, conformal mapping on the viewing sphere to address distortion in wide-angle images. Another interesting approach in the field of image editing is the work of Fang and Hart [23]. By decoupling the deformation from the feature-aligned texture synthesis, they achieve very nice detail preserving shape deformation without the usual distortion around the deformed silhouette.

Sparse matrix-vector multiplication has been mapped to the GPU [24], with some impressive
results. This may also make iterative solvers (e.g. conjugated gradient) even more attractive. A general efficient solution for the BLAS-3 sparse matrix-matrix product is however still a missing piece. The recent trend to utilize newer “many-core” architectures is very attractive to increase the performance of many problems, which exhibits a dominant parallel structure. This includes feature tracking (and other visual computing algorithms), simulations such as “Monte-Carlo” or “N-Body” (particle simulations) and even ray-tracing seems to be within reach to do in real-time. Since this field is growing so rapidly, a more detailed overview of related work is beyond the scope of this thesis.

As mentioned in the introduction, this thesis does build directly upon the work from Eigen-satz et al. [1][2], where the basic formulations for the non-linear least squares optimization are presented and applied to edit mesh-surfaces.

1.2. Overview

In chapter 2 we will introduce the basic formulations, capabilities and application of the constraint-based modeling metaphor, as presented in [2] also including the previous work in curvature-domain shape processing [1][3]. This lays out the foundation for the optimization and parallelizations employed. In the following chapter 3 we will show how to employ parallelization on multi-core architectures in a general view, before going into a detailed look on how to employ the optimizations in the mesh-editing formulation (chapter 4). Chapter 5 gives some examples for the 2D scenario, followed by a conclusion and outlook (chapter 6).
Constraint-Based Modeling

This chapter gives an overview of the constraint-based modeling metaphor established by Eigen-satz et al. [1]. The following summary should help readers with limited or no prior knowledge about the topic to see what kind of computations are involved. For the sake of compactness, not all derivations are repeated here; please review [1][2].

2.1. Notation

A discretized continuous surface $S$ is given by a triangle mesh $M = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ where the vertices are given by $\mathcal{V} = \{v_1..v_n\}$ for $n$ vertices $n = |\mathcal{V}|$. Edges, representing connected vertices $(v_i, v_j)$ are given by $\mathcal{E} = \{e_{ij}\}$ and faces containing 3 connected vertices $(v_i, v_j, v_k)$ is denoted as $\mathcal{F} = \{f_{ijk}\}$ with the face-area $A_f$.

Embedded curves (or paths) $C$ on the surface $S$ are given by a polyline $\mathcal{P} = \{P\}$, consisting of connected polyline vertices $p_i$. These polyline-vertices are parametrized by the affine hull of the underlying face-vertices as a linear combination (barycentric coordinates) $p_i = \sum_{i=1}^{3} b_i \ast v_i$.

Other common conventions are: a vector $\in \mathbb{R}^3$ or $\in \mathbb{R}^2$ is denoted by $\vec{v}_i$, and has euclidean 2-norm (squared-length) given by $||\vec{v}_i||$. A position of a vertex in euclidean space is written bold $v_i$. An Edge $e_{ij}$ from vertex $v_i$ to vertex $v_j$ equals $e_{ij} = v_j - v_i$. 
2. Constraint-Based Modeling

In the framework we are using the principal curvatures $\kappa_1$ and $\kappa_2$ (maximum and minimum curvature) which are computed at the vertices $v_i$. An intuitive understanding about the curvature $\kappa \in \mathbb{R}^3$ is often visualized by constructing an osculating circle at a point $p$ (figure on the right 1). The surface is cut at the point $p$ by a plane constructed from the normal and an tangent vector at that point, leading to a curve $C$. The approximation of the curve $C$ at an infinitesimal neighborhood around $p$ with the osculating circle (with radius $r$) will correspond to the normal curvature $\kappa_n(p) = \frac{1}{r}$.

The principal curvatures are then found by choosing the tangent vector, which maximizes/minimizes the normal curvature. Additional concepts and classifications of configurations expressed in the curvature domain can be found in other literature.

When editing a mesh, the target values for the constraints are given by a hat symbol (e.g. prescribing principal curvatures $\hat{\kappa}_i$). They are approximated during the optimization by a deformed surface $S'$ and corresponding values (e.g. $\kappa'_i$) with a prime symbol.

2.2. Problem Formulation

The constraint based modeling metaphor is formulated in terms of an energy-functional which is minimized as explained in the following sections. The fairly common setup incorporating an optimization of an energy-functional can be found in many areas, and not just in computer science. What can be considered special in this framework is the fact that the minimization process is non-linear (meaning the dependency between the input parameters and the parameters to be optimized for) and the appliance for surface deformation. The user has the possibility to manipulate metric-, positional- and curvature-properties, as well as integrated measurements over curves or surface patches, anywhere on the 3D shape. The modeling intent is translated into a set of constraints, specifying the target values for the properties. Among the target value, the influence of a constraint is weighted by an additional user parameter. Since the deviation from the desired target values will encode an energy-functional, we will employ a non-linear minimization for finding a deformed shape $S'$, which approximates the desired constraints as well as possible. The process will solve for the deformed vertex positions, the (non-linear dependent) variables of the problem. In the next section 2.3, a set of meaningful energy measurements is presented. Among constraints used to model, a large set of constraints will be specified to preserve important metric- and curvature properties. In section 2.4 the minimization process is explained, which uses information from the first order partial derivatives of the energies. A crude graphical illustration of the involved steps is given in figure 2.1.

In the following, we will assume that an input triangle mesh $\mathcal{M}$ embedded in $\mathbb{R}^3$ is sampled in a desired resolution. Since finer meshes are more faithful representations of an input surface $S$, it is generally desirable to select a dense mesh. This enables the deformation process to approximate the desired constraints more accurately and will produce smoother results. However, there is a trade-off between resolutions: Finer meshes also imply many more constraints and degrees of freedom, thus boosting the computational complexity, accompanied by slower propagation of large, global deformations. A coarser mesh however can be used to propagate large deformations in early iterations or to edit interactively, due to the smaller number of variables.

1Graphic from S. Löschhorn: “Curvature-domain shape processing” [3]
2.3. Nonlinear Mesh Energy Formulations

Furthermore, we can assume that the triangle faces do not have a very small inner angle, or that the mesh is generally a fair surface (average valence 6), since such a mesh can be re-sampled in a preprocessing step.

The mesh energy formulations come in various forms and have been redesigned and enhanced by Eigensatz and co-workers [1][2] over time. The formulations are partially repeated here and illustrated by images, as they give the necessary insight how constraints can be formulated as energy-functionals. Important characteristics of a surface are incorporated by a metric measurement on triangle-faces (conformal and equiareal) and by the principal curvatures. Such formulations can be used to penalize large deviations from the original measurements during the optimization process. Those shape preservation energies, also commonly referred to as regularization constraints. Conceptually, there is however no difference if an energy is used to retain characteristics of the initial shape (e.g. is used as regularization with the initial values as target) or if the energy is used to manipulate the shape by prescribing target values (modeling constraints). Global regularization energies such as conformal and equiareal and principal curvature energies should however always be enabled, since only then the solution is well defined (largely overdetermined system) and deformations are propagated accordingly. The steering of the influence of an energy follows from the corresponding weights $k_i$. An important characteristic of these regularization terms is also their invariance under rigid transformations (translation and rotation). While being desired properties for editing, not having positional constraints (via. hard or soft-constraints) will result in global transformations. It is sort of equivalent to defining boundary conditions in differential equations.

Generally speaking, we will introduce a linear combination of energy terms defined on the continuous surface $S$ with the implementation in the discretized piecewise linear triangle mesh $M$.

$$E_{combined} = k_{\text{metric}}E_{\text{metric}} + k_{\text{curvature}}E_{\text{curvature}} + k_iE_i \quad (2.1)$$
2. Constraint-Based Modeling

2.3.1. Metric Energies

Faces \( f_{ijk} \) of a mesh encode metric properties, such as a conformal (related to the angles of a face) and equiareal (related to the triangle area) measurements. These concepts are often used in surface parametrization. Refer to figure 2.2.

For any point \( p \) on the surface \( S \), there exists a local parametrization which approximates the surface at that point \( \text{par}_p : (u,v) \rightarrow \mathcal{Q} \subset S \) for a local region \( \mathcal{Q} \). Now we consider a mapping \( m : S \rightarrow S' \) of a point \( p \) from the initial surface \( S \) to the target surface \( S' \): \( p' = c(p) \). Under the same local parametrization we can now define the parametrization of \( p' \) as \( \text{par}_{p'} : (s,t) \rightarrow (c(\mathcal{Q}) \subset S') \). Expressing the mapping between the local parameterizations can thus be written as: \( \text{par}_{p'} = \alpha_p(\text{par}_p) \).

Referring to [25] the jacobian derivatives of this mapping \( \alpha \), offers information about metric distortions. The two typical measurements for metric distortion are related to angles (conformal mapping, which has no angle distortion) and the area change induced by the parametrization (equiareal mapping, which has no local area distortion). Preserving angles or areas can now be expressed as linear combination of energy functionals.

\[
E_{\text{metric}} = k_{\text{conformal}} E_{\text{conf}} + k_{\text{equiareal}} E_{\text{metric}} \quad (2.2)
\]

Integrating over the whole surface will give us a global continuous energy functional.

\[
E_{\text{metric}} S = \frac{1}{A_S} \int_S E_{\text{metric}} dA \quad (2.3)
\]

Note that equal weights for \( k_{\text{conformal}} \) and \( k_{\text{equiareal}} \) will result in an isometric mapping.

For further connection to the first fundamental form, please refer to [1]. For clarity we will shortly repeat the discrete energy formulation:

\[
E_{\text{metric}} M = \frac{1}{A_M} \sum_{f_{ijk} \in \mathcal{F}} A_{f_{ijk}} E_{\text{metric}} \quad (2.4)
\]

Conformal Energy

Expressing an energy over the face’s inner angles is a very important concept. Since conformal mapping aims at angle preservation, we will reduce the amount of shearing a triangle face will

1Graphic from S. Löschhorn: “Curvature-domain shape processing” [3]
2.3. Nonlinear Mesh Energy Formulations

...suffer on the deformed shape. Converse to the equiareal term, manipulating the area of a face, while retaining the inner angles, will not increase the energy at all (scale invariance). As larger deformations will have areal distortion at least in some region, it is most often useful to provide enough influence for the conformal energy term.

A former realization of the per face constant conformal energy was defined as follows (compare also to [25]):

$$E_{\text{conf}, f_{ijk}} = \frac{\cot(\alpha_i) \| e'_{ji} \| + \cot(\alpha_j) \| e'_{ik} \| + \cot(\alpha_k) \| e'_{ij} \|}{2A_{f_{ijk}}}$$  \tag{2.5}

Outlining the concept of weighted cotangents, this formulation is also insensitive to varying tessellations (compared to conformal energy formulation used in [1]). This definition has also successfully been used in the field of mesh parametrization [25]. However, the most limiting factor is convergence, when used in an iterative process. An alternative formulation has been developed by Eigensatz [2]. Splitting the energy into a sum of rational quadratic terms aids a gauss-newton type of solver dramatically, since the objective function is locally approximated with a quadratic [26]. That novel conformal energy formulation incorporates a virtual subdivision to archive a split of the energy term into a sum of six positive rational quadratics. The minor deficit of the elevated computations involved and the increased memory consumption are easily outweighed by the faster convergence property.

**Equiareal Energy**

To provide control to global area distortion, we also use a per face constant energy formulation (compare to [25]). Giving a larger weight to this energy will enforce the solution to evolve towards a solution with as few areal distortion as possible.

$$E_{\text{areal}, f} = \frac{A_f}{A_f'} + \frac{A_{f'}}{A_f}$$  \tag{2.6}

2.3.2. Principal Curvature Energy

In section 2.1 the basic notion of principal curvatures is explained in terms of a parametrization into the tangent plane at a vertex. On a triangle mesh the principal curvatures are estimated at the vertices $v_i$ using the method of Cohen-Steiner and Morvan [27]. The energy formulation itself is again straight forward; by penalizing curvature distortion for the signed minimum and maximum curvatures and appropriate weights we can force the solution to approximate the target curvatures. The discrete version is:

$$E_{\text{pc}, M} = \sum_{v_i \in M} A_{v_i} \left[ (\kappa'_{1,i} - \kappa_{1,i})^2 + (\kappa'_{2,i} - \kappa_{2,i})^2 \right]$$  \tag{2.7}

While being able to use this energy as a regularization, when setting the target curvatures to their values of the undeformed shape, quite a big effort has been invested to provide filtering algorithms to edit a shape by prescribing different target principal curvatures. Please refer to the curvature-domain shape processing paper [1] for filters defined for curvatures. Shortly summarized we have these filters which are typically applied to all vertices, either for $\kappa_1$ and/or $\kappa_2$: 

\[\]
2. 

Constraint-Based Modeling

- prescribe target curvatures
- add a value to all curvatures
- multiply by a value
- mean value filtering
- bilateral filtering
- histogram clamping
- histogram equalization

It should be obvious that for certain configurations of target curvatures, we can get a lot of metric or curvature distortions, corresponding to the modeling intent and deformations. We can only give a least squares solution, even for cases were there is no such an embedding. Consider a genius-0 surface (e.g. a sphere), which we want to deform into a shape with zero maximum and minimum curvatures (deform to planarity). This modeling intent is ill-posed because the target curvatures cannot be reached by all the constraints. The output in such cases depends on the tessellation or numerical issues, and can thus not be considered a valid solution.

### 2.3.3. Positional Energy

Positional constraints are important because they provide constraints which resolve the invariance under rigid transformations that the previous energies incorporate. Hard positional constraints define a set of fixed vertices (partitioning the set of all vertices into disjoint sets of fixed and free vertices). Since they are rigidly fixed to their position in space they will also not contribute to any change of the energy term. The terms are therefore completely ignored in the optimization process, and thus reduce the number of variables in the system.

A weak positional energy measures the distortion from the target position. Typically weak global positional constraints are defined as a regularization energy (also referred to as damping energy) with a fairly low optimization weight $k_{pos,M}$. Constraining the target position $\hat{v}_i$ to the initial vertex location thus offers a way to have all variables in the system (no fixed vertices), while the deformed shape remains near the original.

$$E_{pos,M} = \frac{1}{A^2_M} \sum_{v_i \in V} A_{v_i} \| v_i' - \hat{v}_i \|^2$$ (2.8)

Additionally a handle based metaphor is incorporated, which allows to prescribe the target positions of a subset of free vertices (local weak positional constraints).

### 2.3.4. Path Length Energy

To include a tool to manipulate the lengths along curve, a path length energy was introduced by Eigensatz [1].

$$E_{length_P} = \frac{1}{l_P} (l_P' - \hat{l}_P)^2$$ (2.9)

where $l_P$ denotes the total length of the piecewise linear mesh path $P \subset M$. 
2.3. Nonlinear Mesh Energy Formulations

2.3.5. Areal Energy

Manipulation to the area of a surface patch (represented by a set of faces) is presented in [2], denoting another patch global energy. Similarly to other energy constraints it measures the quadratic deviation from the target area.

\[ E_{\text{area}_p} = \frac{1}{A_p} (A_{p'} - \hat{A}_p)^2 \]  

(2.10)

Note that the triangle face-global (a patch being only 1 face) areal energy can be seen as a bunch of local constraints. While being similar for small deformations, we can see that the global area prescription offers more degrees of freedom to distribute the target area to faces where other energies are less dominant.

2.3.6. Normal Curvature Energy

Manipulating principal curvature as described in section 2.3.2 does not implement any directional control in the curvature domain. Since intuitive detail control of curvature constraints is rather a question of the user interface, Eigensatz and co-workers [2] proposed to manipulate the normal curvature of the mesh along a user-drawn path as path segment constraints. The signed normal curvature is measured as the change of the surface normal along the curve. For any point \( p_i \) on the curve the normal is barycentrically interpolated from the per vertex normals of the corresponding face. Together with the tangential direction of the path, the osculating plane is defined. The change in normal curvature is then interpolated on small path segments and projected into the osculating plane. For details, please refer to [3][2].

\[ \text{Figure 2.3.: } \text{Prescribing the same values for the principal curvatures on the top of the model.} \]

Original image is always in the middle.
2. Constraint-Based Modeling

Figure 2.4.: Editing weak positional constraints. Left: translation / Right: rotation.

Figure 2.5.: Editing a path length constraint. Left: decreased length / Right: increased length.

Figure 2.6.: Editing a area constraint. Left: decreased area / Right: increased area.
2.4. Minimization Process

Defining constraint-based energies, as outlined above, is only the first step. The much harder part is to analytically derive the partial derivatives with respect to the free vertices, which are needed for the subsequent calculations. Depending on the abstraction used this can be rather involved using the chain-rule extensively. Since the optimization is performed on a mesh discretization, the free vertices, also showing up in the energy formulations, are the optimization variables of the system. Due to this non-linear relationship, we are required to employ a sequence of optimization steps to reach a reasonable minimum. Typically this is done with schemes which use informations of the derivatives.

More formal: We state that the minimum of the total Energy:

$$E_{\text{tot}} = \frac{1}{2} \sum_i k_i E_i$$ (2.11)

is found by:

$$v'_1 \ldots v'_n = \arg\min_{v'_1 \ldots v'_n} E_{\text{tot}}$$ (2.12)

Where the factor $\frac{1}{2}$ is induced in subsequent equations for convenience and the energies are defined as proposed in the previous section. This minimization is solved by employing a Gauss-Newton type solver. See [26] for an overview. The use of the proposed [1] Levenberg-Marquart algorithm, which is a damped version of the Gauss-Newton method, has been replaced by the plain use of the Gauss-Newton solver. Reasons for this are mainly the minimized functional, which can be for certain settings composite non-smooth. And observations showed that the damped version, which incorporates the step size control indirectly; we get slower convergence for small local deformations (which in contrast should not be damped too much).

The Gauss-Newton algorithm linearizes the non-linear problem around $x$ using the Taylor expansion:

$$f(x + \delta) = f(x) + J\delta$$ (2.13)

Where the jacobi-matrix $J$ of $f(x)$ entails all first-order partial derivatives used in the linear approximation near $x$. Details about the derivations of the derivatives in the jacobi-matrix can be found in [1] and [2], additionally we will investigate the structure of the jacobian in 4.2.2.
2. Constraint-Based Modeling

We request that the objective function \( f(x) \), depending on the free vertices \( x \), holds the total energy:

\[
f(x)^T * f(x) = 2E_{tot}
\]  

(2.14)

\( f(x) \) thus stacks the signed energy distortions of all the constraints.

Each iteration step (k) in the Gauss-Newton procedure solves in the linearized setting for a displacement vector \( \delta_k \) minimizing the overall energy.

\[
\delta_k = \text{argmin}_\delta \| f(x_k) + J\delta \|^2_2
\]  

(2.15)

Using the update rule: \( x_{k+1} = x_k + \delta_k \).

A practical implementation uses the normal equation

\[
(J^TJ)^{-1} * \delta = -J^T * f(x_k)
\]  

(2.16)

which solves the system in the least squares sense. Since this system of equations is sparse, symmetric and positive definite, we can solve this with an in-core sparse cholesky decomposition.

Alternatively, a numerical more stable method (in terms of the matrix’s condition number) can be derived with the QR-decomposition, solving the normal equation through the “pseudo-inverse” as follows:

\[
\delta = (J^TJ)^{-1} * J^T * f(x)
\]  

(2.17)

Since the jacobian is now decomposed into an orthogonal matrix \( Q \) and an upper triangular matrix \( R \) (i.e. \( J = Q * R \)), we can simplify the formula to:

\[
R * \delta = Q^T * -F
\]  

(2.18)

An investigation of the solver performance is found in section 4.2.4.

Having solved the system in the least squares sense, we have to probe for an admissible step-size \( \alpha \) (where we start with \( \alpha = 1 \)). Updating the vector \( x \) with \( \alpha * \delta \) has to be evaluated using the total energy \( E_{tot} \). If the current step did not decrease the total energy, we will decrease the step-size \( \alpha \) by half and repeat the evaluation. The entire optimization process stops if the gradient of the objective function or the displacement vector \( \delta \) is below an user defined threshold [28].

2.5. Overview with an Editing Example

Based on the outlined framework, an editing example on a coarse sphere is presented. Besides the visual results we get from applying the optimization pipeline, we shall look graphically at the characteristic layout of the sparse jacobi-matrix and the usage in the gauss-newton optimization. The presented example will also be referred to in later chapters, when the individual parts are investigate more detailed.

Given a sphere-mesh with the following statistics: [vertices: 642 / edges: 1920 / faces: 1280], we will mark a path \( \mathcal{P} \) on the surface. The path in this setting will form a path-length constraint \( E_{length, P} \). Besides this modeling constraint, we will also use the metric-/curvature- and damping-energies for regularization (\( E_{metric}, E_{curvature} \) and \( E_{damping} \)). From this setting a vast
possibility of deformed shapes arises, each a least squares solution, depending on the weights used for the constraints as well as the target path length \( l_p \). Figure 2.8 shows some possible results. Since the user can interactively influence these parameters any other combination will produce different deformed meshes \( M' \).

Figure 2.8: Example: Shorten the path length. The visualization corresponds to isometric distortion. (a) original; (b) dominant conformal weight (shrinks shape); (c) dominant curvature weight; (d), (e) increase equiareal weight.

Since the jacobi-matrix \( J \) is the central ingredient for the overdetermined system of equations, it is suitable to take a first look at the layout 2.9. For details in the context of sparse matrices we refer to section 4.2. It should be noted that as we preserve the connectivity of the underlying mesh throughout the optimization, the non-zero structure of both \( J \) and \( J^TJ \) will also only change their values but not their position in the matrices. While this approach makes the algorithm simpler and efficient, adaptively subdivisions in regions of large deformations are not allowed.

For a first overview, we will state some more observations: The order of individual constraints (rows) in the jacobian is not relevant but must of course correspond to the entries in the objective function. The columns of the jacobian will entail the partial derivatives of a single constraint w.r.t. the free vertices (our variables). The number of entries in a column is known a priori and will be discussed in section 4.2.2. In this example all the vertices are free and therefore the number of variables is exactly \( 3 \times 612 \), as we deal with 3 components (x,y,z) per vertex (3D case). Thus fixing a vertex rigidly to a position equals a removal of 3 columns (reducing the number of variables). Fixing vertices will also lead to empty rows in \( J \), since the variables are
removed from the system. Thus, special care has to be taken to build only a reduced jacobian, where in each row there are at least 3 entries. Without further going into details, the number of constraints can, in this example at least, easily be calculated.

\[
\text{#constraints} = \frac{2}{[1]} \cdot \#\text{vertices} + \frac{6}{[2]} \cdot \#\text{faces} + \frac{2}{[3]} \cdot \#\text{faces} + 3 \cdot \#\text{vertices} + 1
\]

(2.19)

Where there are two per vertex constraints \([1]\) of \(E_{\text{curv}}\) encoding the energy distortion for \(\kappa_1/\kappa_2\) in the objective function and in the corresponding rows in the jacobian the partial derivatives. Similar for the metric energies, where the virtual subdivision of \(E_{\text{conf}_{ijk}}\) has 6 rows per face \([2]\) and 2 for \(E_{\text{areal}_f}\) respectively \([3]\). The derivatives of \(E_{\text{damping}}\) is a diagonal matrix with the three orthogonal derivatives per vertex. \(E_{\text{length}_p}\) is an integrated measurement only spec-
ifying one constraint. From these observations, it is already obvious, that our target level of parallelism should exactly reflect the individual computations on the mesh primitives (vertices / faces), except for the integrated measurements, which are advantageously handled individually when multiple modeling constraints are defined.

As a last prerequisite, we will visualize the steps that are needed to solve the normal equation (2.16) in figure 2.10. The matrix product \((J^T J)\) is symmetric positive definite, hence we only need to calculate either the lower or upper triangular part. Given the sparsity structure of \((J^T J)\) we can see that e.g. the path length constraint does effect the amount of non-zeros dramatically (from the way the matrix multiplication works). When continuing with the cholesky decomposition (without any reordering or symbolic factorization) the sparsity will get even worse, because a lot of fill-in will occur. Moreover we can not expect that we can significantly reduce the amount of fill-in given the specific layout of the normal equation.

In the context of how the gauss-newton steps are chosen, we already outlined in 2.4, that the displacement vector \(\delta\) resulting from the least squares solution will have to be multiplied by a valid step-size \(\alpha\). As the search for a valid step size includes the evaluation of the objective function on the displaced mesh \(M'(x) = M(x + \alpha \cdot \delta)\), the objective function computation should be very fast. It is difficult to give a general estimate about how often we have to just probe the objective function to find a valid step size and when we need to objective function together with the jacobian to get the next least squares solution. Most often at the beginning of the optimization the magnitude of the step size is large (≈large deformations) and has to be at most once shortened. For a successful gauss-newton step this corresponds to: twice evaluating the objective function, and once the assemblage and solve of the normal equation (with the current jacobi-matrix). As the minimization process continues and locally contradicting constraints will influence the convergence of the energy formulation, the efficient probing of the objective function becomes crucial. In this state, the ratio between searching a valid step size (probing the objective function alone) and solving the expensive normal equation can easily surpass 5:1.

The overall convergence of the entire gauss-newton procedure is very much dependent on the modeling intent and how good all the constraints can be satisfied. While this is a typical characteristic of a non-linear optimization problem, where a minima/maxima is found by iterative refinement, we notice that the first steps are the important ones for global deformations. Given enough weight to the modeling constraint, we can achieve good results with even few (5-9) successful steps. Problematic in the evaluation of the convergence is also the weighted influence of the constraints. In this example for instance the shortened path-length \(\hat{l}_p\) is unlikely to be exactly \(l_p\), considering the other energy distortions, and can be approximated through a lower \(\hat{l}_p\) or an increased weight \(k_{\text{length}}\).

For some editing operations (mainly in curvature filtering) it has also proven to be useful to restart the optimization with the current metric properties which avoids global drastic changes is the triangle shapes over runs, while at the same time other constraints can be approximated better from the current deformed shape.

On the other hand we can support the interactivity in the following practical way: supposing that the constraints are defined and the user wants to adjust the weight or target values on-the-fly, we can reuse the obtained deformed mesh as start vector. While the deformation for slightly increased weights or manipulated modeling constraints will be very near the old solution, the linearized system is good approximation for the current mesh \(M'\) and will thus save us some of the gauss-newton steps. Independent of whether the different start vector is used, the system
2. Constraint-Based Modeling

will not change in its structure under these minor modifications. Reusing every information and
datastructures setup at a previous run will decrease the initialization costs to almost zero. In a
practical implementation the application will observe the changes in the parameters and request
an update, which will eventually restart the optimization after a user specified minimum number
of gauss-newton steps.
Parallelization and Optimizations

3.1. Overview

In the previous chapter we have laid out the basic formulation of the various constraints and the optimization pipeline. It is not surprising that a lot of computations can be done in parallel as will be discussed more detailed in the next chapter. Since we cannot expect that future systems can increase the clock-speed and minimize the memory latency, at the same rate as they did in the past years, we will get less performance boost for free. Exploiting current hardware trends clearly shows a way out of this misery by using parallelization. Since the number of processing cores on a shared memory architecture is still relatively limited (on a decent desktop computer one should expect about 4 cores) and we do not want to migrate to more specialized and dedicated systems, we propose the use of newer graphics hardware, allowing the “embarrassingly” parallel constraint computations to run on the GPU. While asynchronous computations can still be done on the CPU, we believe this is an ideal combination of doing distributed computing between the main CPUs and attached graphics-card GPUs, while still holding the data relatively close to each other, using the high-performance pci-express bus. The parallel portion of the code has been significantly increased, leaving few computations not benefiting from more processing units. The next section 3.2 provides some terminology about parallel computing, leading to the actual realizations using OpenMP for the shared-memory multi-processing and the “Compute Unified Device Architecture” (CUDA) for the parallel computing on the graphics-card.
3. Parallelization and Optimizations

Figure 3.1: Schematic coupling between a SMP-CPU die and a parallel multi-processor GPU architecture.

3.2. Terminology

“Amdahl’s law” [29] is often cited in the context of parallel computations, as it gives an upper bound on the maximum expected speedup obtained by parallelization.

\[
\frac{1}{(1 - P) + (P/N)}
\]  

(3.1)

Where \( P \) is the portion of code that can be parallelized by \( N \) processors. It immediately follows that the speedup is bounded by the serial fraction \((1-P)\), as \( N \) goes to infinity. Consequently the performance/price ratio is greatly affected by the serial part, while only the parallel portion can increase performance due to investment in more resources. In “embarrassingly” parallel problems the \( P \) value is very dominant (almost 1) or even no communication/serialization among parallel tasks is required and thus scales with \( N \). Besides identifying regions with this structure, we try also to reduce the serial fraction wherever possible, leading to code that has to be protected from race condition artefacts resulting from careless parallelization. Whenever a problem is not “embarrassingly” parallel or just a simple task which does not affect other parts of the system, we need some synchronization concepts to control the order of execution, exchanging data among parallel threads (rendez-vous/barrier) or simply protect a shared resource against each other (critical section/locks). Depending on the application programming interface (API) they may differ in terms of cost from the underlying hard-/software implementation but the concepts is very similar; being a distributed or shared memory architecture. While the numbers of processing units with their characteristics (mainly clockspeed, pipelining, registers and on-chip cache) is central to perform fast computations, the most limiting factor in current systems has become the memory latency and transfer costs. In parallel computing this speed gap can become even worse while sharing the memory bandwidth with other threads. On the other hand, if the computations are not very global memory bound, the observed speedup can also be super-linear (\( \geq N \)), since for instance the data can be held local per processor in cache and thus reduce the memory latency.
3.3. OpenMP

3.3.1. Introduction

One of the easiest way to make use of parallelism in an existing software can be achieved with OpenMP, which stands for Open Multi-Processing. This API allows programmers to parallelize regions in the code with simple compiler directives. The master-thread then will fork a number of slave worker-threads and the tasks are divided among them.

Figure 3.2: OpenMP execution model.¹

A typical parallel section includes a short living parallelized task-set which will run as high prioritized threads among other system processes. Generally, there is no restriction on the number of threads created this way, but a 1:1 mapping of the threads to physical processing units (default; static distribution of work) will not cause additional cost for thread creation/deletion and thus is likely to be the best choice. The creation and termination of threads, as well as the data layout and synchronization among cores is completely hidden from the programmer, allowing in common cases only to insert one line of code and harvest the benefit of parallel execution. Since the modifications only influence are by means of compiler directives and environment variables, the same code will work with compilers not supporting OpenMP (by ignoring the directives) and run on any system without explicitly setting or knowing the amount of cores. Note that a stricter coupling between threads and physical processing units (PUs) can be enforced with a thread affinity construct (only intel processors) to exhibit better data locality and minimize thread migration costs.

3.3.2. Usage

By default there are two possible, related, ways to use parallelism: task parallelism and data parallelism. Since the amount of threads of a parallel region is controllable and each thread is having a unique ID, task parallelism is achieved by giving each thread a different task. Provided that a program can be divided into independent tasks (with few or none communication needed), this work sharing may be less common for the following reasons: totally independent tasks typically vary in their execution time and there might be few individual tasks, both characteristics do not allow to optimally utilize all the PUs. Depending on the duration and communication

¹<http://en.wikipedia.org/wiki/Openmp>
3. Parallelization and Optimizations

between the threads, it may be more suitable to use other constructs for thread creation and control than forcing the OpenMP-threads to implement a MIMD (multiple instruction, multiple data) characteristic (Flynn’s taxonomy [30]).

On the other hand, working on chunks of data concurrently (data parallelism), following more the SIMD (single instruction, multiple data) taxonomy, is typically better suited to parallelize with OpenMP. Most commonly we parallelize a “for loop”, distributing the work within a loop to the threads (outer loop parallelism). Currently OpenMP only runs efficiently on tight shared memory architectures, then opposed to the more general MPI (message passing interface) all the data is by default shared among all threads. This concept requires that global data is coherently visible to all threads; whenever a thread writes to global memory the operating system has to make sure that any old data (still present in an other processor cache) is no longer used, by marking the cache-line as invalid. In an extreme case when data parallelism is used for chunk-size below a cache-line size (8-512 Bytes), the effects of “false sharing” are considerable.

The goal is therefore to identify regions with high computational content, which can be treated in a thread-local context and only write the result to global memory at the end of the parallel section. Synchronization/communication among threads is achieved similarly by writing to shared global memory and synchronization concepts like barriers or locks will handle the control-flow of each thread. Critical sections implement a coarse grained locking against all other threads, allowing resources to be used exclusively. The best choice of synchronization construct is often problem dependent, but finer grained locks, only protecting a part of the shared memory/resources under mutual exclusion, does usually perform much better than the critical section approach, protecting the entire shared resources for a short time. Especially if the computations under mutual exclusion are longer, it is certainly recommended to use a limited number of locks and a suitable hash function to find the current lock. While the number of locks itself is secondary, the amount of performed locking determines the scalability.

3.3.3. Performance

Performance measurements typically suffer from severe drawbacks concerning their expressiveness: The testing environment is very much dependent on the computer system specifications and the compiler used to generate the optimized machine code. The complex computer system also mixes effects from the overall system load, threading overhead, cache effects and library calls together and makes the analysis of the effects on smaller problem sizes even harder. For larger problem sizes, which are sampled in sequence as with benchmarking applications, these restrictions may seem to be weakened and statistics about Gflops can give a hint about the peak performance. Due to this weakness and missing relevance with what is parallelized in a real world application, we renounce to give scientific relevant measurements.

A short overview about how to estimate the threshold of some basic vector manipulation algorithm is however shown in the appendix A.2. An estimate about the overhead of issuing a parallel section can also be found in ¹. This estimate is given with about 0.1 ms, which is not at all negligible for smaller problems. Much more important in real scientific computations is of course the issue with the memory traffic/latency, which arises if multiple threads load and store (even in scattered fashion) to RAM and cost from synchronization constructs. Since this is completely problem and implementation dependent, we refer to the next chapter for an illustration of an actual example.

¹<http://software.intel.com/en-us/articles/basic-openmp-threading-overhead/>
Graphics Processing Units (GPUs) have evolved significantly over the past years. Performing efficient operations on independent data in parallel has been the driving force behind current visually appealing graphics capabilities. From rasterization over programmable shaders towards a general purpose API, the GPU has become a de facto standard for efficient stream processing. A typical system has a parallel “many-core” architecture with a multi-processor (MP) layout that differs significantly from the SMP (symmetric multi processor) layout discussed before (see also figure: 3.3). Along with the “many-core” architecture, yielding extreme high Gflops, also the memory layout has changed to accommodate the need to keep the PUs busy, permitting memory intense (data parallel) problems to exploit the, by a multiple factor higher, memory bandwidth. All in all, with a suitable parallel problem, carefully engineering and respecting the properties of the hardware, the performance gain ranges up to several hundred times compared to a current general purpose SMP computer system.

Facilitating the access for non-graphics related computations, the two major graphics card vendors (ATI and NVIDIA) introduced their SDK, allowing the programmer to use a subset of the C-language. Most important changes involve scattered read/write to graphics memory and arbitrary loop and branching as defining the control flow. Thus, the capabilities have moved away from the pure SIMD concept towards a flexible model. Performance penalties from branching only affects performance significantly when divergent paths are taken and the execution is serialized. Another important issue, to be efficient, concerns the memory-type and -layout, which will be covered more detailed in section 3.4.1. In order to be more specific about the concept, we will in the following restrict ourselves to the “Compute Unified Device Architecture” (CUDA) SDK from NVIDIA, publicly released in 2007.

### 3.4.1. Introduction

A good introduction into the CUDA framework can be obtained by downloading the SDK and browsing the examples as well as from the programming guide [31]. For the sake of understanding we will introduce some more terminology. When a parallel function is called by the host processor (CPU) to instruct the launch of a bunch of threads on the device (GPU), we speak about a kernel call. To compute a result, a set of input data is first copied to the device global memory, loaded and processed with a number of kernel calls. The computed result, residing in the global device memory, is then transferred back to the host for further post-processing. The smallest unit of a parallel execution on the device is a lightweight thread, which is grouped into a thread block (up to 512 threads). Synchronization however is only possible among threads of the same block (via the fast per block shared memory of a multi-processor) or at the end of a kernel call (via the global device memory), with all the threads. The reason is that thread blocks are mapped onto arbitrary multi-processors and executed without any specific order. Also, the maximum of active blocks on a single multi-processor, is currently limited to 8. Considering the memory consumption of a thread multiplied by the number of threads per block, we often experience the shortage of registers. A schematic overview of the compute capability is given in the appendix of the programming guide [31]. Behind the scene, current multi-processors consist of 8 scalar processors (SP) which implements a new SIMT (single instruction, multiple threads) architecture. Threads are scheduled in groups of 32 parallel threads (called a warp) denoting the minimum of threads per block we should allow for efficient computations. When
3. Parallelization and Optimizations

considering the SIMT architecture, it becomes obvious that threads in a warp should also read from memory in an un-coalesced way, and thus avoiding individual address computations or bank conflicts (block shared memory).

It should be noted that if a CUDA device is still used for the graphical output (integrated/non-dedicated use of a CUDA capable device), the available resources are shared, degrading the overall performance.

![Figure 3.3: Chip architectures](image1)

![Figure 3.4: CUDA Architecture](image2)

3.4.2. Usage

Preferably individual kernel calls issued by the CPU exhibit long parallel computations with as few global memory read/write operations as possible, while doing as much floating point operations as possible. While the host returns asynchronously after a kernel call, we should use this opportunity to do other tasks before issuing the next transfer of the results back to the host or starting the next kernel call. Such a pipelined layout of CPU and GPU threads, each running their separate tasks, can be given by the problem formulation, or might be tailored by streaming partial results.

A memory read/write from a thread to the global memory is very costly (approximately 400 cycles) but can of course not always be avoided. To hide the latency it is preferably to have multiple blocks mapped to a multi-processor (if the shared memory and register per block usage allows this). As outlined, the only synchronization/communication during a kernel call can be achieved by a per block basis via the fast shared memory. If no communication is needed it is still adequate to use the shared memory as an extended register set. Typically the CUDA compiler (nvcc) will also follow this practice; but not to an amount one might desire.

---

1Graphics from the NVIDIA - CUDA Programming Guide [31].
A last ingredient concerns the host-device transfer of the data; significant faster transfers can be achieved if we use the specific host memory allocation construct (cuMallocHost) instead of new or malloc. The effect of disallowing paging (page-/pinnlocking) of the data can be used by the CUDA driver to (even asynchronosly for newer devices) stream from/to the physical adress of the host memory without the indirection via the virtual address-space. Despite the large pci-bus bandwidth, the amount of transferred data between host and device dictates the overall speedup drastically.

3.4.3. Performance

Currently limiting the use for hard scientific computations is the fact that the architecture (before compute capability 1.3 anyways) has only one double precision (DP) processing unit (opposed to 8 SP handling single precision computations) per MP. Thus the number of flops significantly differs; and the drastic speedup is only acquired when restricting to single precision.

As in the section 3.3.3 we shall look at the performance using a real-life example in the next chapter. Some measurements, giving a coarse view about the performance expectancy of some simpler examples is given in the appendix A.3.2.
3. Parallelization and Optimizations
Parallelization and Optimization for Constraint-Based Mesh Modeling

In chapter 2 we have briefly pointed out, that the large sized Jacobi-matrix assembly is, from the involved computations, inherently parallel. The framework thus offers a great opportunity to do a vast majority of computations in parallel. A straightforward attempt to use parallelism involves the computation of the objective function and derivatives of an individual energy, e.g. $E_{\text{curvature}}$, parallel to other energy formulations. The entire Jacobian matrix would then be assembled from locally stored matrices of the individual energies, given a suitable sparse matrix representation (section 4.2.1). This is a suitable approach if all the energies had approximately the same computational complexity and the number of PUs is about the same as individual types of energies (which are typically only a few). Thus the maximal speedup we might expect compared to a pure serial execution is at best in this order of magnitude. Not being enough, we have seen from the way the Jacobian is built that some of the energy formulations are perfectly suited to be parallelized on a per face or per vertex level. Section 4.1 describes a coalesced memory layout of the mesh data structure needed for the parallel computation on the graphics device, where the granularity of parallelism is much higher. Computations are performed entirely in parallel without any communication. The resulting objective function and Jacobian entries of each energy are transferred back to the host, ready to instruct the new asynchronous kernel call and carry on processing the partial matrices. Building the sparse system of equation from the partial matrices is then done in parallel on the CPU (section 4.2), before the solver gets called.

4.1. Porting the Global Energies to CUDA

Examining which energies are most suited to be offloaded to the graphics card should fulfill some criteria: since there are many threads (in the thousands) working in parallel, the energies
4. Parallelization and Optimization for Constraint-Based Mesh Modeling

that build a global energy by combining the influence of each primitive (mesh-path vertex or face) in one common constraint are not well suited, due to their data dependency. An example of these energies are usually found when working with a path. E.g. the integrated path-length constraint only forms one row in the jacobian (by summing up the per vertex partial derivatives), hence does also only have a single objective function value (namely the weighted target path length difference). The same holds for the area energy. Luckily, these constraints are rarely defined over the entire mesh and are much easier handled with OpenMP parallelism, on the host. Either fine-grained locking is used or parallelism is employed over multiple defined constraints (various paths or areas).

In contrast the energies from [1] are perfectly suited to be computed without any communication needed. In the following we will consider the following energy formulations:

i damping
ii conformal
iii equiareal
iv principal curvature

In an editing setting, these energies will contribute by far the most entries in the system of equations.

Computations on a triangular mesh require efficient access to the vertex position and the static connectivity defining the embedding in $\mathbb{R}^3$. In the appendix A.5 an schematic overview of the datastructures is outlined. First the half-edge datastructure of OpenMesh [32] is flattened and offloaded to the graphics card. Since efficient face computations are needed (for [ii] and [iii]), each relevant face (relevant for the computation of metric energies) has constant access to the vertices of a face, similar to a “indexed-face”-set mesh representation. For the principal curvature energy ([iv]), an additional edge traversal is needed, computing data from the 4 involved vertices of the two edge-adjacent triangles. Similar to common vertex-, face- and half-edge iterators, each thread is then assigned to handle exactly one element completely in parallel. Besides the mesh-datastructures and result arrays (objective function and derivatives), there are a few other lookup maps involved, to handle the computations and mapping to their free vertex index efficiently (e.g. to define the column-index of the jacobian).

Once the connectivity has been off-loaded to the graphics card, some allocated memory can be freed again (one-way host to device upload). In contrast, data which are transfered from the device to the host and vice-versa, need to be in the same internal memory layout in both memory spaces. To reduce the transfer cost between the two memory locations, the allocation of page-locked host memory is recommended. This is effectively a way to share physical memory with the host processors, suitable for an integrated graphics-devices.

Precomputations and initializations of the energies is then done as expected: the target values are computed in parallel and stored:

- per vertex areas (barycentric subdivision) and the face areas ([iii]) (along with the summed original total mesh area), which will also be used by other energies on the host.
- original free vertex positions are stored for [i]. Whereas the same array also serves to reset the shape to the initial state, if desired.
- inner angles and the edge-lengths (of the virtual subdivision, as outlined in 2.3.1) are computed, as well as the derivative-indices, indicating the column indices of the jacobian for every second row ([ii] and [iii]).
- energy [iv] first computes the estimated per vertex minimum and maximum curvature; loads them to the host to perform the filtering (based on the user parameters (GUI) using
OpenMP parallelism) and sends the target curvatures back to the device. Additionally a derivatives lookup-table is computed, similar to the metric energy. This table contains the sorted indices of the vertices in the one-ring (plus the current vertex), and thus specifies the column index for every second row in the jacobian.

Once these properties have been computed, these values will remain constant during the iteration process. While no additional constraints are introduced or changes in target values are requested, the only thing that might change are the optimization weights between individual runs, which are given to the kernel as parameter anyhow.

A schematic overview about the interaction between the host and device shows the use of asynchronous computation on the individual systems (figure: 4.1).

It should be noted that since we have to probe during a gauss-newton step if the solution vector from the solver did in fact decrease the overall energy norm, we have to evaluate the objective function more often (depending on the convergence) than assembling and solving the system of equations. Since this step should therefore be made as cheap as possible, we suggest to decouple the computation of the objective function and the computation of the entire jacobian matrix as much as possible. The trade-off of having the jacobian ready without recomputations in a next step and the cost of a on-the-fly calculations has to be weighted for each energy individually. Typically the computations of the objective function and jacobian do so much overlap, that there is a smaller gain from decoupling. A mixture of the trade-off is employed for the CUDA-energies: for [i] the jacobian is computed, but not transferred to the host unless needed. For the metric energies ([ii] and [iii]), we employ the separation because filling the jacobian matrix will cause a lot of costly global memory writes, which will be hidden by the asynchronous kernel computation. For energy [iv], the separation is again not that obvious. Since the eigenvalue decomposition of the curvature tensor (3x3 SPD-matrix) has to be performed already for the objective function, the cost of the additional device-memory writes of partial results for the derivatives is smaller than repeating the kernel on the primitives, we are not splitting the computations for the objective function and the involved derivatives. Again only the actual necessary data is transferred to the host.

The size of the data will of course have influence in the transfer cost between device and host, and these results will have to be used at some later time (memory latency to fill cache on host). For [ii] and [iii] a compact representation is used exploiting that there are at most 9 entries per row (max. 3 free face-vertices times 3 components per vertex). An indexing for these entries is computed once as a precomputation, respecting also the fact that components of a free vertex are consecutive and that 2 rows have the same column indices.

The curvature energy [iv] and its derivatives (respecting also cylindrical and spherical configurations), which have to be computed with respect to the one-ring vertices (and the current vertex) are assembled into the jacobian using an additional kernel launch. This will lead to at most, 3 (components) times the free vertices in the 1+one-ring, values in a row of the jacobian. Again, the table (column indices of each row in $J$) is precomputed. Considering that again 2 rows are coupled (for $\kappa_1$ and $\kappa_2$) and component derivatives are computed consecutively, we can reduce the lookup structure by 83\%, opposed to a general coordinate oriented sparse representation. Problematic in the derivatives computation is the fact that a vertex can have an arbitrary number of neighbouring vertices. Since the derivatives computation is quite memory intense (involving also searches in arrays), we have to use shared memory to be efficient. For this reason we have bound the maximal admissible valence of a vertex to a fixed number. Currently we support up to valence 10. If this limit is exceeded, the computation of the jacobian has
4. Parallelization and Optimization for Constraint-Based Mesh Modeling

<table>
<thead>
<tr>
<th>Host (CPUs)</th>
<th>Device (GPUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>init mesh datastructures</td>
<td>offload mesh datastructures</td>
</tr>
<tr>
<td></td>
<td>↓</td>
</tr>
<tr>
<td></td>
<td>initial computations</td>
</tr>
<tr>
<td></td>
<td>↓</td>
</tr>
<tr>
<td></td>
<td>share static properties</td>
</tr>
<tr>
<td>↓</td>
<td>↓</td>
</tr>
</tbody>
</table>

iterations:

| Probe Energy Norm: | fill damping obj. func. and jacobian |
| ↓ | ↓ |
| | obj. func. |
| | ↓ |
| | fill metric obj. func. |
| ↓ | ↓ |
| | obj. func. |
| | ↓ |
| | fill curvature obj. func. and jacobian ingredients |
| ↓ | ↓ |
| | sum squared obj. func. |
| ↓ | ↓ |
| | sum squared obj. func. |
| ↓ | ↓ |
| | sum squared obj. func. |

| Fill system of equations: |
| ↓ | ↓ |
| damping jacobian | fill metric jacobian |
| ↓ | ↓ |
| | metric jacobian |
| | ↓ |
| | fill curvature jacobian |
| ↓ | ↓ |
| | sum jacobian into jtj and RHS |
| ↓ | ↓ |
| | sum jacobian into jtj and RHS |
| ↓ | ↓ |
| | sum jacobian into jtj and RHS |

| ↓ | ↓ |
| Host computations |
| | Device computations (kernels) |
| | host-device memory transfer |
| | instruct async. kernel call |

**Figure 4.1:** Host-Device computation paths.
4.2. Assemble the Linearized System of Equations

In this section we will refer to the editing example introduced in chapter 2.5. Since we deal with inherently sparse matrix structures, we will introduce some well known, basic representations of sparse matrices and discuss some properties of the chosen implementation in the next subsection 4.2.1. By considering the characteristic layout of the jacobi-matrix (subsection 4.2.2), we can formulate efficient parallel algorithms for the assembly of the linearized system of equations. As the performance of the solver (subsection 4.2.4) is crucial, we will focus on the least squares solution obtained from the normal equations. This however involves sparse matrix operations, which are discussed in subsection 4.2.3.

4.2.1. Sparse Matrix Representations

Sparse matrix representations are crucial when dealing with large matrices, containing a certain number of non-zeros \((nnz)\). Sparse representations successfully reduce the amount of memory the matrix needs (to the complexity of \(O(nnz)\)) by storing only the non-zero values. The main goal, however, is to reduce the computational effort for BLAS-2,-3 operations compared to the general algorithms applied to dense matrices. The difference is that these operations greatly depend on the structural form of the matrix, influencing the sparsity criteria.

A simple yet computationally inefficient representation for a sparse matrix \(M\) (dimension: \(n = m = 3\) and \(nnz = 5\)):

\[
M = \begin{pmatrix}
a & b & 0 \\
0 & c & d \\
0 & 0 & e
\end{pmatrix}
\]
is given by the coordinate storage format (COO):

\[
\begin{align*}
\text{value} &= \begin{pmatrix} a & b & c & d & e \end{pmatrix} \\
\text{row} &= \begin{pmatrix} 1 & 1 & 2 & 2 & 3 \end{pmatrix} \\
\text{column} &= \begin{pmatrix} 1 & 1 & 2 & 2 & 3 \end{pmatrix}
\end{align*}
\]

Since there is no restriction on the ordering the representation is not unique and requires \(O(\text{nnz})\) time to access an element. This format is therefore barely used in practice. But when considering only a row or a column at a time and requesting that the entries are sorted, we get a efficient representation for a sparse vector embedded in column or row of \(M\). A compressed application of the sorted row/column requirement for sparse matrices is given by the compressed row storage (CRS), or compressed column storage (CCS) respectively. We will focus on the CRS representation, as this is suited to be built row-wise and is also the solver’s input format. The decision whether to use column or row storage is largely dependent on the access pattern and the the distribution of non-zeros along the two dimensions. The CRS representation of \(M\) is

\[
\begin{align*}
\text{value} &= \begin{pmatrix} a & b & c & d & e \end{pmatrix} \\
\text{colInd} &= \begin{pmatrix} 1 & 2 & 2 & 3 & 3 \end{pmatrix} \\
\text{rowPtr} &= \begin{pmatrix} 1 & 3 & 5 & 6 \end{pmatrix}
\end{align*}
\]

with the following connection:

The compression, not clearly visible in this example, comes from the fact that \(\text{rowPtr}\) only requires \(n+1\) space and defines the row indirectly, denoting an index for the \(\text{value}\) and \(\text{colInd}\) arrays. By convention the last \(\text{rowPtr}[n + 1]\) is equal to \(\text{nnz} + 1\). Since the values in the matrix are typically double precision, we get from a dense matrix representation (storage: \(O(n^2 \times \text{sizeof(double)})\)) to a memory efficient sparse representation (storage space: \(O(\text{nnz} \times \text{sizeof(double)} + \text{nnz} \times \text{sizeof(int)} + (n + 1) \times \text{sizeof(int)})\)).

The trouble comes when building the matrix in a scattered fashion, without having a fixed non-zero structure (equals predefined \(\text{colInd}\) and \(\text{rowPtr}\)). To overcome this obstacle, a combination of the presented approaches is used for the central assembly of the sparse linear system. Similar to the CRS format we use a (sorted) row-wise representation, which is beneficial if operations are performed in parallel on a row basis. A sparse row, contains the \(\text{values}\) and \(\text{colInd}\) of the non-zeros for this row as a dynamic vector of structs (containing \(\text{val}\) and \(\text{ci}\) of an entry). The coupling of the the \(\text{values}\) and column-indices into a common struct has the disadvantage that modifications to the \(\text{value}_{ij}\) alone are not as efficient as only modifying the \(\text{value}\) array at a known location. On the other hand, the struct is compact and most often the two members are read in a common memory read. Also this compact representation is simpler to handle, because the vector of sparse row entries is again wrapped into a dynamic array pointing to the individual rows. (yielding something like: vector<vector<CEntry»). While the access of a row \(i\) is constant, the access of row-element \(ij\) is not. Even if a row is ordered (which is advantageous and required for certain operations on the sparse matrix), we have \(O(\log(\text{entries per row}_i))\)
access time. Further investigation on the sparsity structure of the individual matrices used in the constraint-based modeling framework give raise for other optimizations, like exploiting the component (x,y,z) nature and optimized block multiplications using an additional hash-map to access the elements of the “not so sparse” normal equations (section 4.2.3).

### 4.2.2. Jacobi-Matrix

In the section 2.5 we have already inspected the structure of the jacobi-matrix graphically (figure 2.9) and discussed some properties, whereas we provide the mathematical background here. Since the jacobian is defined as all the first order partial derivatives or a vector valued function, we will only get non-zero entries for the variables (the free vertices with their 3 components (x,y,z)) directly involved in an energy formulation from an individual constraint.

\[
J = \begin{pmatrix}
\frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n}
\end{pmatrix}
\]  

(4.1)

For mesh algorithms this corresponds to the partial derivatives w.r.t. to the free vertices of a face (energies [ii] and [iii]) or in the one-ring of a vertex as used in the barycentric curvature estimation ([iv]). Vertex-based positional constraints ([i]) are the easiest, as the only affect the current vertex. Similar simple local definitions can also be derived for edges or other quantities defined on a mesh. Integrated functionals are likewise summed, combining the effects of the composited constraint in a variable, as for instance for the path length constraint.

### 4.2.3. Building the Normal Equation

The normal equation of a gauss-newton step are given by \((J^T J) \ast \delta = -J^T \ast f(x)\), containing a sparse matrix-matrix product (with itself) and a matrix-vector product. The matrix-matrix product is defined as:

\[
(AB)_{i,j} = P_{i,j} = \sum_{r=1}^{n} A_{i,r}B_{r,j}
\]

(4.2)

setting \(A = J^T\) and \(B = J\) yields:

\[
P_{i,j} = \sum_{r=1}^{n} J_{r,i}J_{r,j}
\]

(4.3)

In this case it is directly visible that \(P_{i,j}\) is a sum from each row of \(J\), having an entry (not zero product) at \(ij\). The adding up of product seen over all \(ij\)'s is essentially an outer-product method where the inner products are dot products of row: \(J_i\) and \(J_j\). See 1 for a graphical illustration and relation to general block-matrices.

Translated to a given row \(r\) of the jacobian \(J\): a scalar multiplication \(J_{r,i} \ast J_{r,j}\) for \(\forall i, j\) has to be performed and added up into the sparse product matrix \(P_{i,j}\). Notice how the row index of

\[^1\text{http://en.wikipedia.org/wiki/Matrix_multiplication}\]
J is completely ignored letting multiple rows be handled as block matrix.

Compared to the traditional way of multiplying matrices \((AB)_{i,j}\) with a dot product between row and column \(A_{ir} \ast B_{rj}\) vector for \(\forall r\), this procedure cannot be mapped 1:1 to the sparse case. The accessing pattern is column-wise for \(J\) and the row access of \(J^T\) are the same columns (as we look at the special \(J^T J\) matrix product), thus we would naturally choose a column-wise sparse representation. But because we have to identify the corresponding non-zero entries, this naive approach will be very slow. To reduce the effort, one could use a lookup table, defining a director accessing. Due to the bandwidth bound nature of this implementation, usually another approach is taken. Namely by building the \(J^T\) explicitly (which also is an other accessing map for \(J\) at the same time) and then use of the much more efficient sparse matrix-vector product in a sequence. The cost of building the transposed matrix explicitly is not at all negligible, followed by the multiplications, which could then at least be done in parallel without any communication nor locking. Unfortunately is the large dimensionality of the jacobian (un-symmetric; much more rows) not favoring the presented algorithm. Additionally, in our case at least, we would only need either the lower or upper triangular part of the symmetric product matrix \(P\).

Another disadvantage arises when the assembly of the jacobian should (preferably) be done in parallel.

- The sorting criteria often required for efficient sparse matrix representation would imply that for all constraints handled in parallel, an ordering must be employed, guaranteeing that in a column-based representation columns are filled with increasing row index.
- The ordering cannot be guaranteed (closed surface) or is at least not trivial to find for all constraints; effectively decreasing the parallel portion of the code.
- Any artificially imposed order is less intuitive.

On the other hand, we can use a row-based representation as proposed in 4.2.1 for the jacobimatrix. In combination with the outlined outer-matrix product, it is sufficient to compute a row of the jacobian and sum the multiplied elements of a row into the product matrix \(P\), as well as into the right hand side (RHS). The product matrix is much smaller in size (squared-sized: \#variables), and due to the symmetry, only the upper triangular part is built. When applying this to the mesh energies [ii],[iii] and [iv] we can additionally make use of the fact that there are always two rows with the same column indices and will thus be added at the same position. While doing these operations in parallel, we need to guarantee mutual exclusion during the update of the entries. Pure mutual exclusion with a critical section construct is too restrictive, leading to the next fine grained locking: the rows of \(P\) (while the same lock is also used for the RHS). The amount of locking \(O(\#nnz(jacobian))\) required is by far the biggest disadvantage of this method. Less dramatic are the scattered updates of the products, which are relatively small compared to what can be done with optimizations to this innermost function.

Clearly the damping energy [i] is simply a diagonal, hence does neither need any locking, and will find it’s place in \(J^T J\) in the first entry (diagonal). The energies of [ii],[iii] and [iv] reduce the locking effort by half, due to their 2-row coupling. An efficient way to sum the jacobian rows in groups of at least two constraints will also make use of tailored thread private memory, which will be reused for each iteration of a worker-thread. Additionally, again due to the block processing, calculations can be made before summing the combined product into \(J^T J\) and the RHS, effectively giving the opportunity to use vectorization.
4.2.4. Solving the Linearized System of Equations

Most of the performance gain in the solving of the sparse linear system is obtained if the non-zero structure between runs is preserved. As long as the constraints only change their weights or target values and no new constraint is added or removed, this can be guaranteed and the solver’s internal memory stays allocated. By considering the needs for the in-core parallel direct solver, we are currently using the PARallel DIrect SOLver “PARDISO” [33][34], available in the Intel Math Kernel Library (MKL). “PARDISO” uses a combination of left- and right-looking Level-3 BLAS supernode technique, including supernodal pivoting. This increases the numerical stability and scales very well on SMP architectures. Besides the low memory footprint, also the factorization times are lower compared to other solvers (see [35][36]).

As mentioned in section 2.4 we can also bypass the normal equation (avoiding the expensive sparse BLAS-3 product) using a QR-factorization. Performance measurements however showed that the SuiteSparseQR multi-frontal algorithm [37][38] is an order of magnitude slower and sooner (around 900000 constraints (a model with approx. 40000 vertices)) gets out of memory, compared to the cholesky-factorization. For instance the path-length editing of the horse \((3006v/9006e/6000f)\) takes 1000 ms with the QR-Decomposition and around 100 ms with “PARDISO”. For the sphere \((10242v/30720e/20480f)\) the sparse-QR takes already 10 seconds, whereas the cholesky factorization is around 600 ms. Since we deal with a largely overdetermined system, the reduction to the squared matrix \(J^TJ\) (symmetric, positive definite) drastically reduces the dimension to the number of variables. In the following we are favoring the speed of the “PARDISO” solver together with the outlined assembly of the normal equation over the better conditioned solution obtained from the QR-decomposition, while the difference in convergence is in practice not observable.

Updating the CRS-formatted value array can be accomplished directly and in parallel from our sparse row-formatted matrix. Whereas the reuse of the fill-in reduction analysis and symbolic factorization from the initial run, which is roughly as costly as the two following steps: the numerical factorization and forward/backward solve including iterative refinements, serves to halve the expensive solver costs.

4.3. Performance Results

In this section we shortly summarize the performance gain compared to the single threaded framework of Eigensatz [2], acknowledging that speed was not the main focus. With the parallelizations employed as presented in the previous sections and changing the internal sparse matrix representation as in 4.2 to allow parallel assembly, we achieve speedups which exceeds the expectancy of what could have been done on a common multi-core processor architecture. A more detailed examination of the execution times, made from a series of error quadric simplification steps applied to a mesh, will be given along with diagrams. As the decimation algorithm does collapses in flat regions, the mesh connectivity can change drastically, also yielding configurations that cannot be done entirely on the GPU (concerns only the principal curvature). Moreover, the connectivity along with the integrated constraints also changes the sparsity structure of the normal equations, yielding different request on the storage space and factorization effort needed.

For a detailed investigation into the individual contributions of the energies, we will assemble the individual energies without exploiting the asynchronicity of the kernel calls. Note however
that the time used for the kernel computations are most likely completely hidden. Of course the
times involving the CUDA computations involve also the transfer cost of the results to the host.
Measurements separating the effects of the transfer cost can be found in A.3.2. In the test suite
we also neglected smaller optimizations concerning visual feedback, calculating the stopping
criteria etc., as they contribute a really tiny fraction to the overall times.

4.3.1. Comparison with the Single-threaded Framework

First we look at the evaluation of the total energy norm $E_{tot}$, which is used by far the most
frequently (figure: 4.2). Since the evaluation of the total energy norm did always assemble
the global jacobian together with the objective function in the former implementation, we get a
tremendous speedup from separating the two ingredients.

Unfortunately, the previous part is not useful by itself as it evaluates only a functional. But
already from this measurements, we get an idea of how a massively parallel architecture can
scale compared to a single threaded system. The steepest increase in the old timing plot is in
the region, where the sheer size of the data exceeds the cache size. Surely the parallelization of
the energy formulations has successfully removed this bottleneck from the system.

For the second step, we investigate the assembly of the normal equation (4.4). In the orig-
inal version, the global jacobian was built and processed in a column-oriented sparse matrix
representation as outlined in 4.2.3. Building $J^T J$ thus involved the assembly of the transposed
explicitly, trashing the cache and causing costly dynamic allocation of memory for larger matri-
ces (especially visible when considering measurements over runs (not used for further plots)).

Once the linearized system of equations is set up, the timings are the same for both methods.
Considering the former implementation, we must however add the cost of the symbolic analysis
phase to the total times, as the non-zeros structure was not fixed between runs.

Adding up the costs for a successful gauss-newton step sums up all the individual parts and
is visualized in figure 4.6.
4.3. Performance Results

**Figure 4.2.** Performance plot of the evaluation of the objective function compared to the former implementation.

**Figure 4.3.** Performance detail plot for the CUDA energies and their averaged influence for the total cost.
4. Parallelization and Optimization for Constraint-Based Mesh Modeling

**Figure 4.4.** Performance plot of the assembly of the normal equation (left).
Performance of the "PARDISO"-solver (right).

**Figure 4.5.** Performance plot of the assembly for the CUDA energies and their averaged influence for the total cost.
4.3. Performance Results

Figure 4.6.: Performance plot of a complete gauss-newton step (left) and the speed-up factor (right).

Figure 4.7.: Performance plot of a complete gauss-newton step (enlarged).
4. Parallelization and Optimization for Constraint-Based Mesh Modeling

4.3.2. Detail Investigations

While the previous subsection contained a comparison with the original framework, we will summarize the performance of the parallelized components for two distinct examples:

i Sphere: \(1024^2 v/30720 e/20490 f\)

ii Horse: \(3006 v/9006 e/6000 f\)

<table>
<thead>
<tr>
<th>Times in milliseconds (ms)</th>
<th>Sphere (i)</th>
<th>Horse (ii)</th>
</tr>
</thead>
<tbody>
<tr>
<td>objective Func.</td>
<td>5.6</td>
<td>1.8</td>
</tr>
<tr>
<td>assembly LSE</td>
<td>291.2</td>
<td>51.4</td>
</tr>
<tr>
<td>solve</td>
<td>660</td>
<td>65</td>
</tr>
</tbody>
</table>

Both models are relatively small, but the difference in complexity is evident. Foremost for the solver. An investigation on how the individual optimizations steps builds up the total execution times is found in the appendix A.3.4.

4.4. Discussion and Outlook

By using the high bandwidth and “many-core” architecture of the graphics device, we have successfully removed data intense parts of the computations, which were inherently parallel tasks and have been among the most expensive parts (super-linear speedup). The remaining energies computed on the host are parallelized on a suitable level (mostly over their individual constraints). The computational complexity of the normal equation assembly is reduced well below the complexity of the solver, leaving the part which is the most difficult to optimize. While the linearized system could, with some more effort, also be composited directly on the device, we still would have to solve the system. Moreover, as we need a series of gauss-newton steps, we look in fact at a problem which is a serial iteration with parallel subtasks. Given this structure, the problem size for interactive examples is currently still very limited (around 5k vertices), for which we can do at least some steps per second. Of course, a further converged solution can only improve the result, but in the outlined interactive setting we are in a first step primarily interested in fast feedback from the given constraint parameters, which are then adjusted on-the-fly.

In consideration of the fact that more recent hardware can increase the speed of the solver using more CPUs (Intel-MKL claims a scaling of factor 7 on a eight-core machine). Similarly we can expect the assembly to scale. The additional use of designated “many-core” devices (e.g. in a cluster) could additionally reduce the timing for the computation of the derivatives and objective function, since the kernels calls itself could be executed in parallel. Due to the fact that the computations on the GPU scale that well and more designated hardware is even less usual in a common desktop computer, using clusters is not a realistic scenario. What is however useful is the steadily increasing amount of multi-processors, registers and shared memory in the “many-core” architecture. Currently, we are using the first generation of the CUDA-architecture (compute capability 1.0), guaranteeing at least the execution on all CUDA-capable devices.
4.4. Discussion and Outlook

While testing the constraint-based modeling framework (also the 2D setup discussed in the next chapter) the following observations concerning the strengths and weaknesses of the involved energies in combination with the gauss-newton procedure shall be briefly discussed.

The strength of the approach is obviously that we can solve non-linear energy formulations by the iterated use of gauss-newton steps. On the other hand, this general approach is not unconditionally stable. Issues such as small local deformations or large displacements (especially when using positional constraints) can exhibit poor convergence due to a combination of the following reasons:

- the influence of the constraint affecting a variable may be contradicting or small (related to the optimization weights).
- in the following gauss-newton step, the surrounding configuration may have changed, partially annihilating the optimizations made in a previous step.
- numerical stability (limited floating point arithmetic) is always an issue and there is no robust mechanism incorporated to avoid that triangles get flipped or converge to a single point.
- once metric distortion, for instance, occurs strongly locally in one step, the process will find over multiple runs a least squares solution minimizing the global distortion. This however makes no statement about local minima and the solution may not be converged to a state where subsequent on-the-fly deformations affecting again the unconverged region are propagated correctly. In such a (extreme, with large un-smooth constraint modifications) setting this can yield a solution with few faces having large distortion and the rest practically none (global least squares solution).
- the number of iterations needed for advanced convergence differs severely (type of constraint, weight, target value).
- the mesh-connectivity has an additional minor involvement in these reflections.
- matrix factorization results (with possible super-nodal or graph partitioning techniques) may not produce an equally well result, when reusing the symbolic factorization from a previous run.

In an investigation how to handle larger meshes, several approaches are thinkable, each with its strengths and weaknesses. Depending on the constraints, defined on the original, dense, mesh we run into various problems. More concretely we tried to reduce the complexity of the mesh by edge-collapsing simplification steps, while recording the simplification steps in local-frames on the coarser level. The collapse informations are represented similar to Hoppe’s progressive mesh [39], with the extension to address dynamic meshes. The coordinates of the collapsed vertices are represented in a local frame as in the work of Kobbelt et al. [7]. While this allows to reduce the complexity of the mesh to a desired degree, it also means that the low-frequency modifications on the coarse representation is computed on the coarse-scale vertices (variables), and thus reduces the employability of the (also global) defined constraints in the global minimization. Very hard, if not impossible is the usage of the curvature-domain shape processing tool-set in this combination. For constraints defined over a path or a surface patch, a simplification algorithm should be used which respects the locality to the modeling constraints and thus only decimates the mesh in “irrelevant” regions. While this restriction suppresses the need to map the paths or areas into the coarser resolution, the solution can not be considered general enough. An other issue, besides the missing high frequency properties, is the quality
4. Parallelization and Optimization for Constraint-Based Mesh Modeling

of the simplified mesh incurred from the various choices of simplification algorithms (error quadric metrics [40], maximal independent set or based on the curvature are just some possible candidates). The intention in this setting is to run some steps of the non-linear optimization on the coarse mesh, propagating the deformations on the next finer level and repeat until we have converged on the finest level. Global and large deformations are thus propagated faster (from the coarser mesh), and in subsequent runs hopefully only refined in this half-multigrid (multilevel) setup. In consideration of these drawbacks from the original framework, that approach does not seem to fit well into the general non-linear optimization.

An other approach is to use a multi-resolution setup, where the shape is decomposed into a base-shape and a detail-vector \( M = B \oplus D \), as in the work from Botsch et al. [41][42]. Deformations are only applied to the base shape \( B' \), which is a low-frequency representation of the input surface. Since the low-frequency representation can be sub-sampled with fewer errors, we could again try to use a hierarchical mesh to propagate deformations from the coarsest level. Once more, these coarse representations are not ideally suited to be used with intrinsic surface properties constraints such as the principal curvature filtering or other modeling constraints. These representations have however proven there entitlement is in the domain of positional constraints (affine transformation), where different surface deformations setups are more suited.
2D-Setting

A practical application of the constraint-based modeling metaphor is also found in the 2D domain. Given a planar mesh, we will attach an image texture rigidly to the undeformed mesh, using linear interpolated texture coordinates. From this initial setting, we can now reuse the energy formulations presented in chapter 2 to define easy to use image operations. The general non-linear framework subsume some interesting image manipulations outlined in the related work, especially since we are using the stable metric energies [2], we can implement an adaptable “as-rigid-as-possible” characteristic. To enhance the control of the metric distortion, we allow to define not just a single global optimization weight, but face local individual weights (section 5.2). Practically the 2D domain reduces one dimension, so that we can formulate new constraints exhibiting the planar embedding (section 5.1). Besides the reduction in terms of variables by a third, the linear algebra calculations are simplified as well, so that we can handle mesh-sizes with approximately up to 10'000 vertices interactively. Another good thing is that the visual information is in the texture, making the need for denser meshes less stringent. While the framework is general and quite flexible, we will present some limitations and relations to other computer vision tools in section 5.3 followed by a discussion (section 5.4).

5.1. New 2D-Constraints

5.1.1. Path positional constraints

To enhance the toolkit from weak positional constraints defined on mesh-vertices, which supports translation, rotation and uniform scaling through a handle interface, an additional path-positional constraint is defined, supporting the same affine transformations. While the energy formulation is essentially the same, replacing the mesh-vertices \( v_i \) with the polyline vertices \( p_i \) of a path \( \mathcal{P} \), the new formulation is a natural extension, incorporating an interface for finer control, direct transformations and combinations with other path related constraints.
5. 2D-Setting

5.1.2. Path Angle Constraints

The signed angles of the path’s inner vertices (excluding the first and last vertex) embedded in \( \mathbb{R}^2 \) can be seen as a measurement of the path’s piecewise deviation from straightness (curvature).

\[
E_{\text{angle}} = \sum_{p_i \in P} (\alpha'_p - \tilde{\alpha}_p)^2
\]  
(5.1)

The inner angle is given in the interval \([0..2\pi)\), where we start from the “from” to the “to”-edge in counter-clockwise order, and the input of the path denotes the global orientation. Refer to figure 5.1. The partial derivatives calculation is given in the appendix A.4. Two basic filters: namely prescribe (absolute) and add (relative) target angles are similarly defined as in the curvature domain filtering.

Please note, that the convergence of this energy is not too good, since the angles are local to the path-vertices \( p_i \) where the angular deviation from \( p_i \) influences both \( p_{i-1} \) and \( p_{i+1} \). As an example when defining the target angle \( \tilde{\alpha}_p \) to be \( \pi (180^\circ) \) for every path-vertex, the gradients will partially annihilate the effects on a path-vertex (see figure 5.2).

A better way to describe the straightness (but only straightness) should be implemented in a further extension, to provide a more stable energy. The way this constraint should be defined follows the recent work of Carroll et al. [22], where the straight line constraint is very central energy. In figure 5.3 the deviation for \( p_i \) from the straight line between \( p_{\text{start}} \) and \( p_{\text{end}} \) is given by the distance:

\[
d_{p_i} = (p_i - p_{\text{start}})^T \vec{n}(p_{\text{start}}, p_{\text{end}})
\]

\[
\vec{n}(p_{\text{start}}, p_{\text{end}}) = R_{90} \frac{p_{\text{end}} - p_{\text{start}}}{\|p_{\text{end}} - p_{\text{start}}\|}
\]

Where the \( R_{90} \) is \( 90^\circ \) givens-rotation matrix to derive the normal vector \( \vec{n}(p_{\text{start}}, p_{\text{end}}) \) of the line. This can be directly expressed as non-linear energy:

\[
E_{\text{straight-line}} = \sum_{p_i \in P} (d_{p_i})^2
\]  
(5.2)

For this energy, we should provide the partial derivatives involving all variables \( (p_i; p_{\text{start}}, p_{\text{end}}) \) differentiated w.r.t. the free vertices of the face. The formulation would also allow to incorporate prescribed normals \( \vec{n} \) as well as constraining lines to be parallel, horizontal or vertical etc. Unfortunately, this is left as a future extension, given the short time left to complete the thesis.
5.2. Interactive Weight Control

Instead of using global weights to control the influence of an energy term, a more fine-grained control of the weights is employed. We allow the user to modify the weights, for the metric energies on a per face- and for the damping energy on a per vertex-basis, by offering a surface brush tool. The weights are then visualized on the faces of the mesh, indicating the current distribution. While the adapted weights are interactively updated, the manipulation of the metric energies in particular serves as an effective manipulation tool to weight the influence of other modeling constraints or to define regions where certain metric distortion is less desired. For larger selections on a dense mesh it is however advantageous to turn off the automatic update, which would otherwise degrade the coloring experience when interleaved with the optimization, as they do not run in completely independent threads.

5.3. 2D-Constraint-Based Mesh Modeling

In this section we investigate the capabilities of the system by considering some examples. At first the connection to the “as-rigid-as-possible” metaphor is explained by examining the manipulation of a triangulated shape (subsection 5.3.1). A more general setting with regard to
5. 2D-Setting

embedded image manipulation, where the semantic shape-contents of an image are not triangulated as a individual meshes, is then outlined in the next subsection 5.3.2.

5.3.1. 2D-Shape Modeling

When considering a triangulated shape, we typically do not want to be concerned about the underlying representation. Assuming a regular sized mesh, we have fewer triangles in a thinner area (e.g. the extremities in figure 5.4.

![Figure 5.4.](image)

*Figure 5.4.: From left to right (1): original model with 5 handles (head and extremities) move head handle down with: (2): dominant equiareal weight (3): dominant conformal weight. (4): near isometric; only when both metric energies are enabled we get a intuitive reasonable, rigid result.*

In the least squares solution these less denser regions are thus also the less rigid ones, when considering uniform metric weights. While this corresponds quite well to the intuitive understanding, other “material” parameters can only be achieved with non-uniform metric weights (see figure 5.5).

![Figure 5.5.](image)

*Figure 5.5.: Visualizing the effects of different metric weights. (1): give the arms more weights, results in (2) when pulling the arms. The same manipulation with uniform weights (3).*

While this setting illustrates the involved energies quite well, we have to be careful, not to use only the simple positional constraints together with the metric energies, since this setting is more elegantly and gentle in terms of computer resources handled for instance in the work of Igarashi et al. [19]. On the other hand, positional constraints and the corresponding handles
(note that multiple points can without problem be moved at the same time given a suitable input device) are often used because the user is accustomed to this interface, and a lot of transformations are realizable. By modifying every single point, even all configurations are feasible. We will exhibit the strength of the general constraint-based modeling metaphor, which allows to describe combinations of various other constraints everywhere on the surface, in the next subsection, where we will not make use of the additional benefit of having triangulated shapes under deformation.

5.3.2. 2D-Embedded Shape Modeling

Embedded shape modeling defines a general image manipulation setup, where a texture is embedded into a surrounding space. The discretization of the ambient space is often secondary, but we keep the focus on triangle meshes. The underlying principle is a warping (mapping) function which translates the pixels of an image to the corresponding target positions.

Unfortunately the field of image manipulation is very broad, considering the various nature of images (drawings, computer generated-, camera-images etc.), a general approach is difficult to develop, and many tools have been introduced to handle a specific user modeling intent. Sometimes the use of simple cut, “manipulate” and paste operations seem even to do the trick. While on the other side, we have todays powerful computer systems, allowing to do much more elaborated image editings which relieve the user from tedious, (semi-) manual manipulations. Of course we cannot completely automatize tasks since an intent for post-processing an image must be given, but the way a desired target image can be created significantly differs when using designated tools. Given the vast amount of different tools (professional, commercial) or research (some mentioned in the related work section 1.1), an inspection highlighting some results from the constraint-based modeling toolset seems adequate.

Wide-Angle Lens Distortion

In photography the choice of an adequate lens for capturing a subject is very important. An artist may choose a wide-angle lens (short focal length) for capturing very near or large objects for several reasons, but mainly if the entire motif should fit on the image, while a further distance to the object is not possible. Whereas the perspective distortion from this scenery may not be unwanted for esthetic reasons, the induced distortion significantly differs from human perception. At least in our mind’s eye, objects retain their natural shape, e.g. a sphere is a circle and linear structures remain straight. A way to correct the distortion which arises from flattening the viewing sphere to an image can thus be achieved in a post-processing step, which tries to warp the image undoing the perspective distortion. The simplest correction algorithm uses a simplified mathematical model (a polynomial function \( f(r) \) which depends on the radius from the “center” of the image) to handle (radially symmetric) radial distortion.

Handling asymmetric distortion needs at least some features from the image. Quite correctly this is done in the recent work from Carroll et al. [22], where they constrain feature lines to be straight in a non-linear least-squares formulation, very similar to ours. The main difference is that they do the calculations on the viewing sphere (given by a camera-dependent variable) and thus retain conformality in that domain.

Since we can do similar constraint optimizations, but from a distorted input image, we will
5. 2D-Setting

Figure 5.6.: Straightened lines from a wide-angle input image.

Figure 5.7.: Straightened lines from mercator input image.

Figure 5.8.: Straightened lines from a wide-angle input image. Yielding not parallel lines (2), which is adjusted using path-positional constraints (3).
present some results from straightened line features, yielding more “natural” looking images, where the straight features are enforced to be straight in the result image. Since the mapping is a dominated conformal least-squares mapping (with the additional weaker equiareal constraints enabled), we can also take other local conformal projections (mercator or stereographic) as our input. The input pictures in figure 5.6, 5.7 and 5.8 are from the webpage of Carroll et al.[22].

Please note that the unpleasant bumpiness around the path constraints does not result from the underlying triangulation, but solely from the way the paths are entered into the framework. Currently it is only possible to draw free-hand lines or by stitching piecewise linear paths together, hence we do not get a smooth path. This should be corrected in a future release by either smoothening the paths (e.g. moving average filter and mapping to the faces) or gathering them from a spline representation (possible imported from professional vector graphics application). Extensions are manifold, the best way to enter a path on a feature line would be to incorporate a feature sensitive path generation (A*-algorithm).

Mixed Constraints

The broad band of image manipulation that can be done with the constraint-based modeling approach, is highlighted by some distinct examples, showing the nature and capabilities of the various modeling constraints (figure 5.9, 5.10, 5.11 and 5.12). Since the distortion has to go somewhere in the image, it is globally (in the least squares sense) distributed in the image. We can aid the modeling intent by defining different local weights to the metric energies, as was already done in example 5.5. For larger deformations an increasing of the rigidity along the constraint is often advantageous since the deformation will act at first very locally and will propagate globally through the gauss-newton iterations. Doing so thus increases the stability of the propagation and especially for positional constraints (when defined with a high weight to increase the effect) and avoids to some extent the danger of flipped triangles. The flipping occurs when the sign of the normal changes (also in the jacobian) because of a large positional deformations (the other energies are far more stable from how they act) which overshoots the metric regularization energies and since there is no 3rd dimension to evade to, the deformations are forced to planarity. Of course this is only a problem because we cannot do enough gauss-newton iterations until the global convergence is completely propagated before a further displacement is issued interactively. Diverting the local modeling constraints to the robust metric energies thus offers an effective way to avoid these situations from which the gauss-newton procedure may not recover. Alternatively, one could intercept the flipping when updating the displacement vector and move the vertex, invalidating the configuration, towards its one ring center. Other regularization strategies can also be introduced, but in practice the problem does only arise when we have a dense mesh and very strong deformations from one gauss-newton run to another.

Another issue that is worth mentioning is related to the above observation. When many (locally contradicting) modeling constraints are defined on the same variables, dissipating their effects, the convergence may get very poor. Again the problem is a bit artificially constructed and is eased by avoiding these situations in the modeling intent. There is no obvious example, where this is really a problem and other strategies as leaving a small gap between constraints (shift the effects again to the robust metric energies) or weighing one constraint higher in an alternating fashion, so that the different steps in the gauss-newton optimization are taken, leading to the global minimum.
5. 2D-Setting

**Figure 5.9.:** Increasing the highlighted area constraints.

**Figure 5.10.:** The eye-iris shape is retained by setting the target angles to their initial values, while the right path is made rounder and the upper side is given a curved shape.

**Figure 5.11.:** All marked paths form path-angle constraints (straightness) and the top and bottom path are constrained by their length. The marked regions are increased weights for conformal and metric energies defining more rigid areas, when manipulating the length.

**Figure 5.12.:** A path-positional constraint (on top of the car) together with path-angle energies.
5.4. Discussion and Outlook

We have shown that the constraint-based modeling framework offers some easy to use image manipulations. It demonstrates that the same toolset can be reused from the 3D-domain to generate a practical application. Since the solution cannot escape in the 3rd dimension, this application illustrates the behavior of the involved energies by example. Local adjustments are possible by weighting the metric energies on a per face basis, which offers a powerful mechanism to define regions with high importance or to correct e.g. the conformal distortion on a converged solution on-the-fly. From the outlined principle, conformality, allowing also for large deformations, is very important in image processing, even if the conformality is “only” maintained in the least squares sense. Explanatory this means that the distortion is distributed primarily in a region around the constraint and propagated through the face adjacency (metric energy). The place where higher distortions can generally be generously tolerated are uniform or poorly textured regions. Whereas line features or generally salient structures should at least obey a local conformality, so that the contours remain to some extent smooth. What is realistic when manipulating a photo to an artificial image (and why manipulating at all?) remains in the eyes of the observer anyways.

Concluding this chapter we can see that in the embedded deformation, the changes do involve the entire image, such that there is unavoidably distortion somewhere. As outlined, this is accomplished by non-uniform weighted metric constraints. An extension how to automatically generate such importance map is given in the work of Wang et al. [21] by combining gradient and salient information from the image content.

Another way to decouple the global deformations is when the underlying mesh is triangulated for a shape individually and thus removes the rigidness from unrelated shapes, similar to what was discussed in subsection 5.3.1. The alleviated deformation for an individual shape then is pasted into the target image followed by a re-synthesizing texturing step similar to the work of Fang et al. [23].
5. 2D-Setting
Conclusion And Outlook

In this thesis we have shown how the largest parts of the “embarrassingly” parallel energy formulation from the constraint-based modeling metaphor developed by Eigensatz et al. [1][2] can be mapped onto a computer device, which is suitable to handle (independent) parallel tasks very efficiently. In addition we are also utilizing the CPUs to their full extend. Obviously the degree of performance was not achievable some years ago and a similar statement will hold for future architectures. Furthermore, we have to consider the degree of abstraction and programming ease, where a return to plain “C” with these low level optimizations is not directly helpful, and neither is debugging on the device enjoyable. Together with the available hardware and possible flows in implementation it would therefore be impudent to claim that we have reached an optimal solution.

However, a remarkable speedup has been achieved making the very general mesh constraint-based modeling metaphor hopefully more attractive. Then besides the developed constraint definitions by Eigensatz et al. the framework is relatively easily extensible and offers alternate access to surface properties than just using e.g. positional constraints. The generality can also be a weakness since the modeling with the “uncommon” indirect access, foremost when considering curvature-domain manipulations, is often hard to control such that the shape is deformed with as few parameters and user-interactions as possible. Since the deformation is based on a weighted non-linear energy minimization, the process can not be liberated completely from the influence of the optimization weights, which have to be adjusted by the user to reflect the modeling intent. For an initial deformation however, the default settings (metric energies with higher conformal weight, allowing larger deformations, and appropriate weights for the modeling constraints, defining how strong the constraint is reflected in the output model) are providing a often sufficiently good “preview” of the deformation.

Interactivity for medium or larger sized meshes (above 5k vertices) unfortunately has not been achieved and thus leaves the field of interactive applications to either smaller meshes or parts of the surface. By leaving the 3D surface modeling domain, we have introduced a practical ap-
6. Conclusion And Outlook

Application for image manipulation which reuses and extends the presented energy formulations. Besides the deeper insight in how good the convergence and propagations of the energy distortions are handled by the gauss-newton iterations, we have introduced an image manipulation toolset, which partially overlaps with current research in computer vision.

In future extensions we see various potential approaches to increase the appeal of the framework, especially in those domains where other software solutions do not offer these types of deformations. This primarily concerns the curvature-domain shape processing. While the deformations itself can be rather intuitive, it is the way the target principal curvatures are entered and manipulate that is limiting its use. While prescribing normal curvatures along a path offered at least some directional controls, the one parameter (prescribing or adding normal curvature) setting may not offer enough control. The same argument holds also for other constraints. Since this is rather a user-interface reflection, which in the case of constraints defined along a path could be alleviated through an extended GUI which offers for instance a editable profile view of the path in 2D, the general issue of manipulating other things than one can drag intuitively (as positional constraints) with an input device remains. Apart from that, a further investigation on how the intrinsic surface parameters could be employed for adaptivity or morphing approaches could be very interesting. In such a setting we could also deform a model based on some synthesized “template” curvatures or from other shapes.

In the 2D-domain the extensions are as well numerous. Besides the discussed “obvious” extensions in chapter 5, a further investigation in a pseudo-projective mapping could be considered. Already in the presented examples the visual stereoscopic effect changed quite a bit, when using e.g. path positional constraints to manipulate lines (e.g. in the example with the wide-angle room picture 5.6). The achieved effect, when moving, rotating and scaling, resembles in a changed viewing direction, when done adequately. In an extension, salient features could therefore be segmented automatically and used to build up different projections. Similarly, constraining features offer also the capability to stitch together panorama views. Finally, to close the gap to the 3D modeling domain again, the deformed textures can of course also be used in combination with a mesh under deformation or to fit more accurately to a given geometry.
Appendix

A.1. Computer System

<table>
<thead>
<tr>
<th>Intel Core2 Quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6600 @ 2.4 GHz (Kentsfield)</td>
</tr>
<tr>
<td>3.24GB RAM</td>
</tr>
<tr>
<td>2x4 MB L2-Cache</td>
</tr>
<tr>
<td>1066 MHz FSB</td>
</tr>
</tbody>
</table>

A.2. OpenMP

A.2.1. Performance Analysis

To see the effects of using multiple processors on a common task, we present some toy examples. Please note that such examples do often not reflect a real problem statement, where much more work per iteration/thread has to be done. The size of the tasks per thread can really determine if we want to use OpenMP and as can be seen, very soon (even for this lightweight functions) the overhead of creating a parallel region from the threadpool is compensated.

The testsuite setup is as follows:

- since the effects should be measured in a on-the-fly multiprocessing scenario, we want to minimize cache effects (by calling a memory intense function on all processors between measurements). This tends to be more realistic in our framework anyhow, since data is moved un-cached from the GPU to CPU memory and the various stages of the minimization process can easily exceed the cache-size.
A. Appendix

- each function is sampled once (times not averaged over runs) with the parallel directive and once without (on different memory locations).
- for the tests we use dynamic heap arrays with double precision (unless noted differently).

For this small examples, no significant difference is measured between arrays[] (allocated with new) and std::vector. There arises a small discrepancy penalizing the std::vector choice for larger sizes.

- the outer loop we parallelize does not have an inner loop, taking away the opportunity for automatic compiler vectorization. From this point of view, the speedups would even approach the number of processors even earlier for computationally more intense inner functions. A combination of symmetric multi-core processing and vectorization is the optimum we can hope to accomplish.
- the size of the outer loop is rather small (denoting also the size of the involved memory); however there is no reason why the setting should not scale.
- we are using the MS-VisualStudio2005 C++ compiler, with full optimization.

The prototyp for the setup:

2 replacements for foo(i):

<table>
<thead>
<tr>
<th>1. fill without flops:</th>
<th>2. fill with flops</th>
<th>3. fill from other</th>
</tr>
</thead>
<tbody>
<tr>
<td>array[i]=i;</td>
<td>array[i]=log(exp(i));</td>
<td>array1[i]=array2[i];</td>
</tr>
</tbody>
</table>

4. sum:                                      5. sum squared norm:                  6. locking overhead
| sum1+=testarray3[i];                        | sum1+(array1[i]*array2[i]);         | omp_set_lock(&locks[i]);           |
|                                            |                                    | dummyWork();                       |
|                                            |                                    | omp_unset_lock(&locks[i]);         |

7. displace simple:                          8. displace OMesh::Point:              9. displace float3:
| array1[i]+=array2[i];                      | array1[i]+=array2[i];               | array1[i]+=array2[i];              |

Most of these examples would allow for good vectorization with a proper memory alignment (loop unrolling without any data dependency; SIMD instructions for floating point operations etc.). While evaluating the intel c++ compiler (11.1), we observe that in most of these examples the automatic parallelizations for SMP along with an automatic vectorizer using the latest streaming extensions can do the same optimizations automatically (for these obvious examples at least). Manual effort will still be required for handling any non trivial function.
A.2. OpenMP

Figure A.1: OpenMP times for simple functions.

Figure A.2: OpenMP speedups for simple functions.
A. Appendix

A.3. CUDA

A.3.1. Used Hardware

<table>
<thead>
<tr>
<th>Nvidia GeForce 8800 GTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifications for Compute Capability 1.0</td>
</tr>
<tr>
<td>16 multiprocessors (MP)</td>
</tr>
<tr>
<td>16 KB fast shared memory per MP</td>
</tr>
<tr>
<td>8192 registers per MP</td>
</tr>
<tr>
<td>8 single prec. (SP) and 1 double prec. (DP) processing unit per MP</td>
</tr>
<tr>
<td>no asynchronous data streaming rest: see specification.</td>
</tr>
</tbody>
</table>

A.3.2. Performance Analysis

In this section some basic measurements are listed for transferring a page-locked float/float3 array of size N from the host to device and vice versa. See figure A.3 and A.4. To give a hint about the performance two simple kernel examples are considered: computing the square-root of each element (very global memory latency bound) and computing the eigenvalues/eigenvectors of a symmetric positive definite 3x3 matrix (as used for the curvature tensor). Of course the first BLAS-1 example only makes sense if the data is already on the device and is most often part of a larger kernel. The same applies for this simple EV decomposition, where the pci-bus bandwidth clearly dictates the overall speedup. Moreover this function is not extremely well suited for the GPU architecture (while loops, for loops without constant number of iterations etc.), but considering the small effort needed for porting existing algorithm to pure “C”-code, the performance gain is not too bad. The transport cost of the Eigenvalue test contain the whole EV-struct (72 Bytes times N, including: Eigenvalues, Eigenvectors and the upper triangular matrix for both ways (host-device). There are two versions, one does use shared memory for some internal memory and is therefore faster. Since the code is part of a larger pipeline the resources (shared memory) are not used to the maximal extend (in the real example all the data are in registers or in fast shared memory, and these partial results are post-processed on the device).

To be fair, it should be noted that the eigenvalue decomposition on the host uses double precision.

The measured bandwidth:

<table>
<thead>
<tr>
<th></th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>host-device</td>
<td>2.52 GB/sec</td>
</tr>
<tr>
<td>(page-locked)</td>
<td></td>
</tr>
<tr>
<td>host-device</td>
<td>2.07 GB/sec</td>
</tr>
<tr>
<td>( pageable)</td>
<td></td>
</tr>
<tr>
<td>device-host</td>
<td>1.97 GB/sec</td>
</tr>
<tr>
<td>(page-locked)</td>
<td></td>
</tr>
<tr>
<td>device-host</td>
<td>1.56 GB/sec</td>
</tr>
<tr>
<td>( pageable)</td>
<td></td>
</tr>
<tr>
<td>device-device</td>
<td>69.36 GB/sec</td>
</tr>
<tr>
<td>(global memory)</td>
<td></td>
</tr>
</tbody>
</table>
A.3. CUDA

Figure A.3.: CUDA times for simple functions.

Figure A.4.: CUDA speedups for simple functions.
A. Appendix

A.3.3. CUDA Memory Layout

The memory layout for the data on the device is shown in figure A.5.

A calculation for the memory usage and transferred amount of pinnlocked memory, assuming all vertices are free:

<table>
<thead>
<tr>
<th></th>
<th>avg. memory usage in Bytes per</th>
<th>total</th>
<th>objective fun.</th>
<th>jacobian (∀ on gpu / ∃ val. &gt; 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex</td>
<td>392</td>
<td>20</td>
<td></td>
<td>124/88</td>
</tr>
<tr>
<td>face</td>
<td>392</td>
<td>32</td>
<td></td>
<td>288/328</td>
</tr>
<tr>
<td>edge</td>
<td>352</td>
<td>0</td>
<td></td>
<td>0/328</td>
</tr>
</tbody>
</table>

**Figure A.5.:** CUDA Memory Layout.
A.3.4. CUDA-Energies Detail Analysis

For the sphere and horse example presented in section 4.3.2, a more detailed examination is presented in the following tables.

<table>
<thead>
<tr>
<th>metric energies:</th>
<th>principal curvature energy:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1. global Jac. &amp; serial</td>
<td></td>
</tr>
<tr>
<td>2. global Jac. &amp; parallel</td>
<td></td>
</tr>
<tr>
<td>3. interleaved &amp; parallel</td>
<td></td>
</tr>
<tr>
<td>4. interleaved &amp; parallel (2 rows at once)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1. global Jac. &amp; serial</td>
<td></td>
</tr>
<tr>
<td>2. global Jac. &amp; parallel</td>
<td></td>
</tr>
<tr>
<td>3. interleaved &amp; critical section</td>
<td></td>
</tr>
<tr>
<td>4. interleaved &amp; parallel</td>
<td></td>
</tr>
<tr>
<td>5. interleaved &amp; parallel (2 rows)</td>
<td></td>
</tr>
<tr>
<td>6. full deriv. on GPU &amp; interleaved &amp; parallel (2 rows)</td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**

| global Jac. | Transfer device data to host and assemble the global jacobian. |
| serial / parallel | How the normal equation is assembled from the global jacobian. |
| interleaved & parallel / parallel (2 rows) | Transfer device data to host and assemble directly the normal equation. Either taking the block matrix structure into account, or not. |
A. Appendix

A.4. Path Angle Derivatives

Equation 5.1 reflects \( n - 2 \) constraint for each inner angle of a path (with \( n \) path vertices). We differentiate the energy w.r.t. the variables (the free vertices \( v_i \) and their components). By employing the chain-rule, we get:

\[
\nabla_{v_i} \alpha = J_{v_j}(p_i) * \nabla_{p_i} \alpha \\
= b_j \left( \frac{\partial \alpha}{\partial p_{ix}} \frac{\partial \alpha}{\partial p_{iy}} \right)^T
\]

with \( p_i = \sum_{i=1}^{3} b_j * v_j \) defined on a face, as on figure 5.1. The angle derivatives w.r.t. \( p_i \) are defined similar as in [1] for an earlier implementation of the metric derivatives.

\[
\frac{\partial}{\partial p_i} \alpha = \frac{e_{to} \times \vec{n}}{\|e_{to}\|^2} \\
\frac{\partial}{\partial p_{i+1}} \alpha = \frac{e_{from} \times \vec{n}}{\|e_{from}\|^2} \\
\Rightarrow \frac{\partial}{\partial p_i} \alpha = -\frac{e_{to} \times \vec{n}}{\|e_{to}\|^2} - \frac{e_{from} \times \vec{n}}{\|e_{from}\|^2}
\]

A.5. Gui Components

A.5.1. Interface

Most of the graphical user interface is to a large extent self-explanatory, whereas a modeling intent is typically realized in this way:

- Load mesh (must be selected to begin optimization).
- Select faces, vertices or paths and assign them as corresponding constraints (Button: “New”).
- Start optimization (Button: “Start”).
- Changed parameters are set (via the “Modify” button, which is implicitly called when an update is issued. Button “Update”). The slider movements will call the update, either when the slider is moved or released.
- End optimization (Button: “End”) releases all memory, especially the one on the graphics-card (ensures lower memory footprint).

3D-CBM Interface

The 3D interface is shown in figure A.6.

2D-CBM Interface

The 2D interface is shown in figure A.7 together with the slider windows (2D and 3D).
Figure A.6: 3D constraint-based modeling interface.
A. Appendix

Figure A.7.: 2D constraint-based modeling interface and the slider dialogs for 2D (top-right), 3D (bottom-right) respectively.

A.5.2. Mesh Energy Visualization

Computes the energy distortions (evaluating the objective function) with respect to an initial rest state. The plugin is thought as independent component from the optimization, thus uses its own memory on the host and device. Visualization is done using a colormap and the histogram plugin [3].

A.5.3. Texture

The texture interface uses the Simple OpenGL Image Library (SOIL) [43] to load various image formats. The dialog is accessible from the mesh properties (double-click on the scene-graph object). Similar to the mesh visualization modes (available in the context-menu or via key shortcuts, if the scene-graph object is highlighted), the texturing is enabled/disabled using “T” and compositing is toggled with “CTRL-T”.

64
A.5.4. Tools

The toolbox (icon bar) holds some tools, among a new 2D-path generator tool has been added. Context-menus are shown by double-click on the icon. The dynamic items for the point-dragger (figure: A.8) or the interactive weights (figure: A.9) tools are added to the list while the CBM-system is active, allowing to switch between tools more easily.

The radial distortion box in the point-dragger interface was thought to handle symmetric radial distortion (barrel, pincushion), by first deforming the mesh according to the polynomial function. After this step paste the texture on the mesh and push the mesh back to a rectangle with the undeformed point constraints (or path-angle), which then results in a conformal warping (no example shown).

Figure A.8: Point-dragger interface for manipulating positional constraints.

Figure A.9: Interactive weights, called form the “Widget”-Buttons in the 2D-CBM-Plugin.
A. Appendix

A.6. Miscellaneous

A.6.1. Generation of an Equilateral Triangulation

Illustrates the prescription of metric energies, so that the planar grid has equilateral triangles: equal edge lengths \( l_e \), angles \( 60^\circ \) and Area \( A = \frac{l_e \sqrt{3}}{4} \). Potential applications are not further investigated. Figure A.10.

*Figure A.10.* Create a equilateral triangulation from the regular generated mesh (left) by prescribing metric target values (right).
Bibliography


Bibliography


Bibliography

