UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ROSÁLIA GALIAZZI SCHNEIDER

View-Consistent Meshes

Prof. Dr. Manuel M. de Oliveira Neto
Advisor

Prof. Dr. Marc Alexa
Coadvisor

Porto Alegre, dezembro 2010

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Pró-Reitor de Coordenação Acadêmica: Prof. Rui Vicente Opermann
Pró-Reitora de Graduação: Profª. Valquíria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do Curso de Ciência da Computação: Prof. João César Netto
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in creative leaps, no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss."

— Scott Aaronson, MIT

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 6

LIST OF FIGURES 7

ABSTRACT 9

RESUMO 10

1 INTRODUCTION 11
1.1 Thesis Organization 12

2 RELATED WORK 14
2.1 Geometry of an object from multiple views 14
2.1.1 Estimating a shape from multiple views 14
2.1.2 View-Dependent Geometry 15
2.2 Detail as Intrinsic to the Mesh 17
2.2.1 Laplacian Surface Editing 17
2.2.2 Coupled Prisms for Intuitive Surface Modeling 19
2.3 Summary 21

3 VIEW-CONSISTENT MESHES 22
3.1 View-Consistent Meshes 22
3.2 Constructing the Linear System of Equations 23
3.3 Summary 24

4 SKETCHING CONTOURS 25
4.1 Modified Framework for Single-View 25
4.1.1 Silhouette Detection 26
4.1.2 Handle Estimation 27
4.1.3 Handle-Target Correspondences 28
4.1.4 Region of Interest Estimation 28
4.1.5 Mesh Deformation 29
4.2 Consensual Mesh Generation 30
4.2.1 Generating the Consensus Mesh 30
4.2.2 A Note on Implementation 31
4.3 View Interpolation 31
4.4 Results 32

5 GENERATING IMPOSSIBLE GEOMETRY 35
5.1 The Algorithm 35
5.1.1 Optimizing point position 36
5.1.2 Optimizing string curvature 36
5.1.3 Creating the mesh from the strings 37
5.2 Limitations 37
5.3 Results 38
6 CONCLUSIONS AND FUTURE WORK 39
REFERENCES 40

LIST OF ABBREVIATIONS AND ACRONYMS

GPU: Graphics Processing Unit

LSE : Laplacian Surface Editing

SFS : Shape From Shading

LIST OF FIGURES

Figure 1.1: **View-specific distortions in reference sketches for modeling.** The ears and the tail of the bunny are in different positions at each view. This may lead to unfeasible geometry. 11
Figure 1.2: **Restrictions a view imposes to the geometry of the object.** (a) All points in the projective line l going through vertex p project to the same screen pixel. (b) A bad estimation of p's relation to its neighbors. (c) A better estimation of p's relative coordinates. 12
Figure 1.3: **Impossible objects.** (a) PenRose Triangle. (b) Escher's Cube. 13

Figure 2.1: **Shadow Art.** A shape is optimized to project the shadows specified by the user. (a) Input shadows (b) Optimized shape, as projected in the x-, y- and z-axis. 14
Figure 2.2: **Sphere-Hull Equivalence.** The convex hull of the key views is homeomorphic to the sphere and thus can replace it. (From [18]) 15
Figure 2.3: **Barycentric Coordinates of a Triangle.** (a) $v_{new} \approx$ (0.33, 0.33, 0.33) (b) v_{new} = (0.5, 0.0, 0.5) (c) v_{new} = (1.0, 0.0, 0.0). . . 16
Figure 2.4: **Multi-Resolution Hierarchies.** Four resolutions of the Stanford Bunny, varying from fine to coarse. (From [13]) 17
Figure 2.5: **Laplacian Coordinates.** (a) Polygonal Mesh. (b) Adjacency Matrix A (c) Diagonal of Matrix D^{-1} (d) Laplacian Operator . . 18
Figure 2.6: **Results of Laplacian Surface Editing.** Deforming Dragon's upper lip. (From [19]). 19
Figure 2.7: **PriMo: Coupled Prisms for Interactive Surface Modeling.** (a) Prisms configuration around an object (From [5]). (b) As objects are deformed, elastic forces stretch. 20
Figure 2.8: **Results of PriMo: Coupled Prims for Intuitive Surface Modeling .** The Dragon was updated by fixing the feet (in white) and moving the head up (yellow) (From [5]). 21

Figure 4.1: **Pipeline of Sketching Contours.** (a) Automatically detect silhouettes. (b) User provides handle. (c) User provides Region of Interest. (d) Sketch. (e) Vertex Correspondences. (f) Deformation. 26
Figure 4.2: **Silhouette Detection.** Results of silhouette detection algorithm applied to Stanford Bunny, Camel and Tweety. 27
Figure 4.3: **Matching two lines.** (a) Result. (b) Tuple of edge lengths and turning angles. (From [22]) 28

Figure 4.4: **Handle Growth.** Top row - deformation without growing the handles; bottom row - improved handle set using the region growth. (From [22]) 29
Figure 4.5: **Problems estimating the Region of Interest.** (a) Result using [22]. (b) Result using our modification, asking the user for input. 30
Figure 4.6: **Region of Interest and Anchor Vertices.** To deform the ear of the bunny, we fix the base so that only the positions of the ear need to be calculated. 31
Figure 4.7: **Results of applying View-Consistent Meshes to Sketching Contours Framework.** Top row - each separated view; Bottom row - representation using a single mesh. 33
Figure 4.8: **Results of the Interpolation between Views.** Top - Key meshes. Bottom - Interpolation result and respective position in the face. 34

Figure 5.1: **Setting up a Penrose Triangle.** (a) Add points (b) Connect point faces by strings (c) Select view to be optimized 35
Figure 5.2: **Mapping a String to 2D.**(a) String in R^3 (b) Projective Space of the String (c) Projective space mapped to a plane. 36
Figure 5.3: **Creating a mesh from the strings.** (a) Square to be rotated. (b) First and last edges of a connection are used to calculate the orientation of the first and last squares. (c) Quaternions are used to linearly interpolate the two orientations. 37
Figure 5.4: **Problem using Laplacian = 0 to minimize curvature.** (a) Straight line (b) Laplacian coordinate of vertex v_1 (difference from v_1 to centroid c) is not zero 38
Figure 5.5: **Problem minimizing curvature.** The purple lines represent the projective lines while the blue line represents the "minimal curvature". The ideal solution would be a straight line (not moving the cube above along its projective line). 38
Figure 5.6: **Two views of our PenRose Triangle.** Left - Optimized View. Right - Another view. 38

ABSTRACT

Modern games and animation films use hundreds of extremely complex models. At the beginning of the modeling pipeline, artists sketch the object as seen from different points of view. These sketches normally do not comply with the 3D geometry of the objects and are often inconsistent.

The purpose of our work is to investigate the problem of creating meshes that respect every 2D view of an object as much as possible. In our algorithm, each view imposes some restrictions to the final geometry, which will be found through the solution of an optimization problem. Our main goal is to generate meshes that approximate sets of views that would be intuitively classified as conflictuous.

Our results show that many apparently inconsistent sets of views can be represented by a single mesh. Even in cases where the representation using one mesh is not adequate, we believe that a mesh that maximizes the similarity to all views can be used to improve interpolation between views (which is used to render the mesh, when seen from arbitrary points of view).

Another interesting application of our technique is the creation of realizable shapes that approximate impossible objects, at least when observed from a specific point of view. Sometimes artists will impose restrictions that are topologically and/or geometrically unfeasible. Two examples of that are the PenRose Triangle and Escher's Impossible Cube. In these cases, we generate objects that look like the provided shapes as much as possible, when seen from a particular point of view.

Keywords: Mesh Editing Preserving Detail, View-Dependent Geometry, Impossible Figures.

Malhas de Polígonos Consistentes com Vistas 2D

RESUMO

Jogos e filmes de animação utilizam centenas de modelos geométricos extremamente complexos. O início do processo de criação desses modelos passa pelo esboço da aparência dos objetos em 2D, a partir de vários pontos de vista. Como resultados, não é incomum a ocorrência de inconsistências entre as várias vistas.

O objetivo do nosso trabalho é investigar a relação entre uma ou mais projeções de um objeto no plano e sua geometria. Formalizamos essa relação de modo que cada vista impõe restrições à forma 3D, que pode então ser calculada como a solução de um problema de otimização. Nosso principal foco é a criação de geometrias que aproximem, tanto quanto possível, conjuntos de vistas que intuitivamente seriam classificados como inconsistentes entre si.

Os resultados obtidos demonstram que muitos conjuntos de vistas aparentemente conflitantes podem ser aproximados por uma única malha de polígonos, sem que o resultado final sofra perdas significativas. Mesmo em casos onde a representação por um só objeto não é adequada, acreditamos que a criação de uma malha que maximiza a similaridade com todas as vistas pode ser usada para melhorar a qualidade da interpolação (utilizada para renderizar o objeto a partir de pontos de vista arbitrários).

Uma segunda aplicação da técnica desenvolvida é a criação de modelos geométricos realizáveis, mas que aproximam, pelo menos a partir de uma vista, a aparência de objetos que são geométrica e/ou topologicamente impossíveis. Em alguns casos, o artista impõe propositalmente restrições que não podem ser satisfeitas, como, por exemplo, no Triângulo de PenRose e no Cubo Impossível de Escher. Nesses casos, criamos objetos que, quando observados a partir de um ponto específico, se parecem tanto quanto possível com as formas impossíveis desejadas.

Palavras-chave: Geometria Dependente de Vistas, Edição de Malhas Preservando Detalhes, Figuras Impossíveis.

1 INTRODUCTION

Three-dimensional modeling has become very popular in the last decade. Modern games and animation films include hundreds of extremely complex models and the industry provides advanced software to allow artists to keep up with this demand. The very beginning of the modeling pipeline is, however, still the same as in the 1920s when Walt Disney first created Mickey Mouse: sketching how an object is supposed to look like.

When constructing 3D shapes, artists typically begin by approximating a set of drawings from different views of the target object. These drawings are meant to capture the look and feel of the object and frequently include some distortions that cannot be represented by static geometry. An example of this can be seen in Figure 1.1, where the ears of the bunny are in different positions at each view.

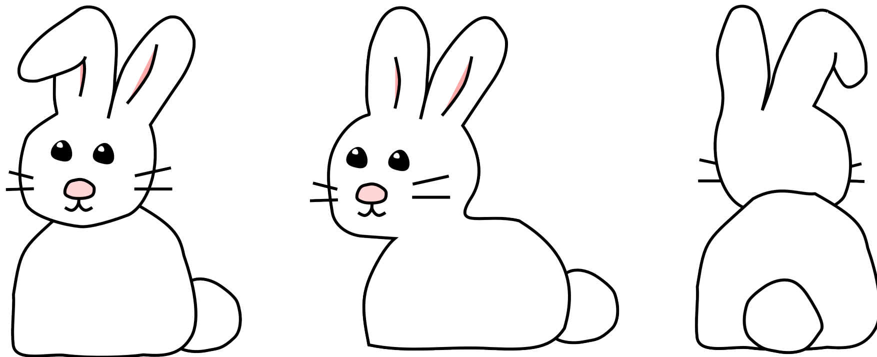

Figure 1.1: **View-specific distortions in reference sketches for modeling.** The ears and the tail of the bunny are in different positions at each view. This may lead to unfeasible geometry.

The purpose of our work is to investigate the problem of modeling shapes that are consistent with 2D views. We use equations on the vertices of the mesh to formalize the way the object should look like, when seen from a specific point of view. Then we solve a linear system, trying to match the equations provided by every view as closely as possible.

Our results show that, frequently, views that would be intuitively classified as conflicting can be represented without significant loss by one consensus mesh. Even in the cases where views cannot be satisfyingly represented by a single mesh, we believe the consensus mesh can be used to improve the interpolation between two different views of the object.

Our approach is based on projective geometry principles, where a point on the screen maps to a line in R^3, going through the point and the center of projection of the camera, as shown in Figure 1.2(a). Thus, we can move every point along its projective line without changing the projection.

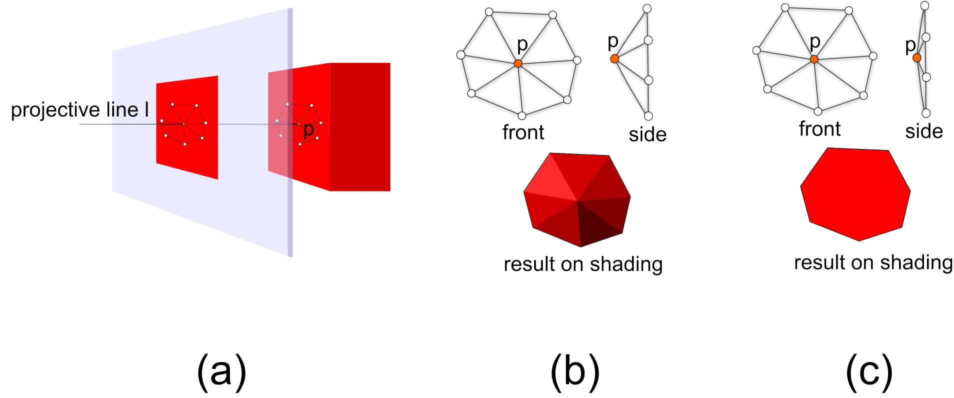

Figure 1.2: **Restrictions a view imposes to the geometry of the object.** (a) All points in the projective line l going through vertex p project to the same screen pixel. (b) A bad estimation of p's relation to its neighbors. (c) A better estimation of p's relative coordinates.

Because we are dealing with shaded views of the object, and not only shadows, some extra care must be taken. The shading of the image gives us information about the curvature of the object. We formalize that as the position of a vertex in relation to its neighbors. We can see a vertex in Figure 1.2(b) and (c), in the same position, surrounded by neighbors shifted along the projective line. We can clearly notice the distortion caused by this change in the angle of the faces, and consequently, in the shading of the object.

It is important to note that small errors in the relative position of a vertex to its neighbors do not significantly affect the final result. Respecting a line constraint and the exact relative coordinates for every vertex would leave little room for optimization. More specifically, we would only be able to translate the unchanged mesh in the view direction of the camera. We will elaborate on this when discussing Laplace Surface Editing [19], in chapter 2.

Another interesting application of our technique is to investigate cases where an artist intentionally generates a view which cannot be realized. Two examples of this can be seen in Figure 1.3. We construct three-dimensional shapes that approximate these impossible objects when seen from one specific view.

1.1 Thesis Organization

The remaining of this thesis is organized as follows. Chapter 2 reviews some related work. We will discuss the latest advances in acquiring a shape from multiple views and representing the details of a model as intrinsic characteristics.

Chapter 3 introduces our novel technique. We will explain the basic ideas that motivated the work and give more details on the construction of the linear system of equations.

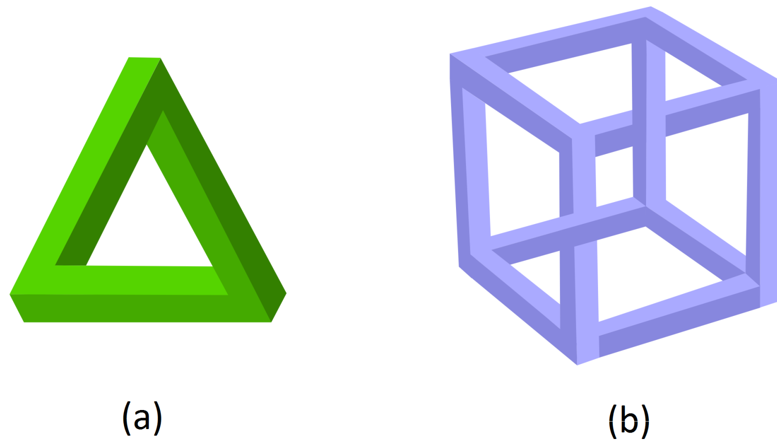

Figure 1.3: **Impossible objects.** (a) PenRose Triangle. (b) Escher's Cube.

Chapters 4 and 5 show two applications where we used our technique. In the former, we used an already existent pipeline where sketches are used to deform three dimensional shapes. The latter discusses the approximation of impossible objects when seen from an specific viewpoint.

Finally, Chapter 6 discusses the achieved results and possible directions for future exploration.

2 RELATED WORK

This chapter discusses existent research related to our work. We divided this review into two main sections: (1) calculating the geometry of an object from multiple views, and (2) editing objects preserving its details (which is the same as saying that the details are an intrinsic part of the object).

2.1 Geometry of an object from multiple views

There is some significant amount of research devoted to investigate the relation between the shape of an object and its views. In this section, we will review the works that are most closely related to ours.

2.1.1 Estimating a shape from multiple views

Laurentini [15] introduced the concept of visual hull. The visual hull of an object is the maximal volume that can generate the exact same silhouette as the object itself, when projected onto any plane. Kutulakos [14] discussed the intrinsic ambiguities when characterizing an object by its silhouettes.

More recently, Mitra [17] introduced a system that generates objects from given shadows. The user provides a small number of images that he wants to be the projected shadows of the object, when illuminated from specific points. The images are deformed such that they can all be represented by the same geometry, while still preserving their main features. Figure 2.1 shows some results achieved by this algorithm, obtained using the demo provided by the authors.

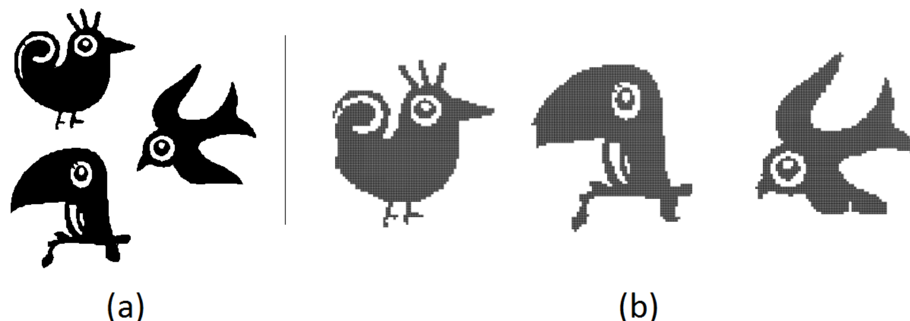

Figure 2.1: **Shadow Art.** A shape is optimized to project the shadows specified by the user. (a) Input shadows (b) Optimized shape, as projected in the x-, y- and z-axis.

Another branch of research is Shape-from-Shading (SFS), first implemented by Horn [11] in the early 1970s. The idea of the technique is to create shape based on the shading of one or more images of an object. Shading refers to smooth variations of color in a surface and can provide information about the curvature of the object (or, as we mentioned before, the relative position of a vertex to its neighbors). There are many approaches to calculate SFS, yet two groups seem to have received most of the attention:

- **Minimization.** The geometry is calculated using minimization of an energy function;

- **Propagation.** The geometry is estimated at some points, and these results are propagated to the rest of the shape.

The works contained in these two categories vary from assuming a fixed model of reflection to finding parts of the image where the shape can be determined. The entire body of research dedicated to this problem is too extensive to discuss in this section. Please see Zhang et al. [21] for a comprehensive discussion of these techniques.

2.1.2 View-Dependent Geometry

Rademacher [18] addressed the problem of creating models from conflicting views. He introduced a technique called View-Dependent Geometry to cope with the view-specific differences. The idea is to create multiple geometries, one for each view, and choose the geometry to be rendered based on the angle from which the object is observed. The deformations would be inherent to one object, instead of different instances of an object. The contributions of the work are how to represent and render this kind of model.

This approach is different from the ones we mentioned in Section 2.1.1 in two major points: First, the shape of the object at each view is created by the user, not estimated by the system. The user himself will deform the mesh to achieve the view-specific look. Second, the technique is not concerned in resolving the inconsistencies in the views, but in representing them and rendering the object in a way that approximates user expectation.

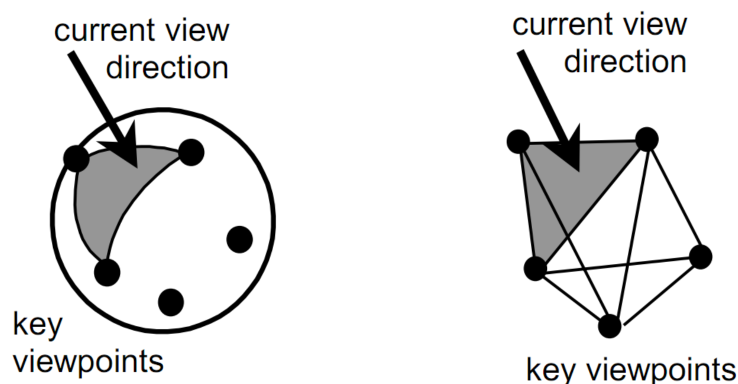

Figure 2.2: **Sphere-Hull Equivalence.** The convex hull of the key views is homeomorphic to the sphere and thus can replace it. (From [18])

The first steps of the algorithm can be seen as setting up the view-dependent geometry. The initial input is a small number of views, that the user will position in space. These initial views of the object will be referred as *keyviews*. The user will create a mesh for each key view, and deform it to respect the view-specific constraints. At the end of this phase, we will have a small number of models and their respective positions in space.

Once the key meshes are placed, we want to use them to render the object from every view point. Suppose we have three main views a, b and c, placed respectively at (1, 0, 0), (0, 1, 0), (0, 0, 1). Obviously, when observing the object from (1, 0, 0), the user will see the object as deformed for view a (and analogously for b and c), but we need a way to cover every direction.

The most intuitive geometry to parameterize all views is a sphere of radius 1 (for simplicity, in fact any sphere would work) centered at the origin. Each point in the surface of the sphere represents a viewing direction. We would then search the sphere for the key deformations that are nearest to the desired point of view: the three nearest points in the surface of the convex hull of the key views, as shown in Figure 2.2.

To find the key points to be used in the arbitrary view creation, we first find the intersection of the convex hull of the views and the desired view direction. The key views selected are the vertices incident to the face intersected by the ray. The last step in rendering the mesh is to interpolate the three views. We need to find weights that vary according to the distance, so that there are no abrupt changes in the mesh deformation, which would look unintuitive for the user. An elegant solution is to use the barycentric coordinates of the intersection point in the selected face. The barycentric coordinates are the proportion between the area of the face and the area of the triangles formed by the point, as shown in Figure 2.3.

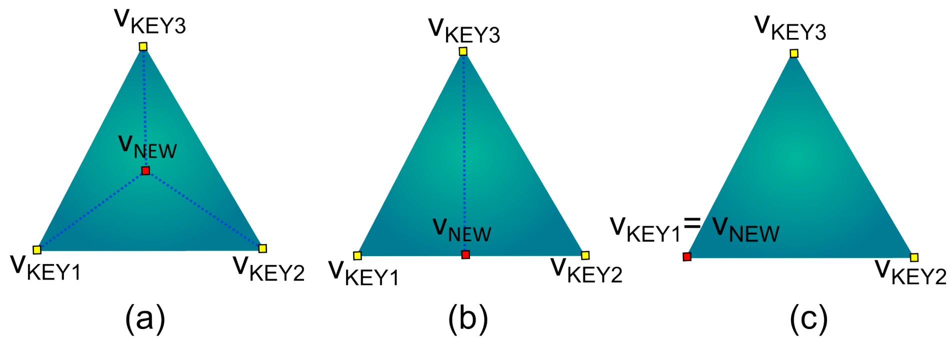

Figure 2.3: **Barycentric Coordinates of a Triangle.** (a) $v_{new} \approx$ (0.33, 0.33, 0.33) (b) v_{new} = (0.5, 0.0, 0.5) (c) v_{new} = (1.0, 0.0, 0.0).

The author used the barycentric coordinates (α, β, γ) associated with the desired view v_{new} as the weights of the contribution of each key mesh. The position of each vertex would then be interpolated as $v_{new} = \alpha \times v_{key1} + \beta \times v_{key2} + \gamma \times v_{key3}$. This provides a smooth variation between the views.

There are some cases where the view direction ray will not intersect the convex hull. This may happen because the key views cover only one side of the object. One of the strategies suggested by Rademacher is to use one of the existent meshes in this situation.

2.2 Detail as Intrinsic to the Mesh

A very important part of our technique is based in representing the details of a mesh as an intrinsic part of it. Our goal is to keep the details imposed by the views, while allowing the vertices to move. One approach that tries to separate the object transformation from its intrinsic geometry is to use several levels of refinement, introduced by Kobbelt [13]. In this paper, the author uses a hierarchy of different resolutions, where the coarse resolution is the base shape and the finer resolutions will add details to the mesh. We can see four resolutions of the Stanford Bunny in 2.4.

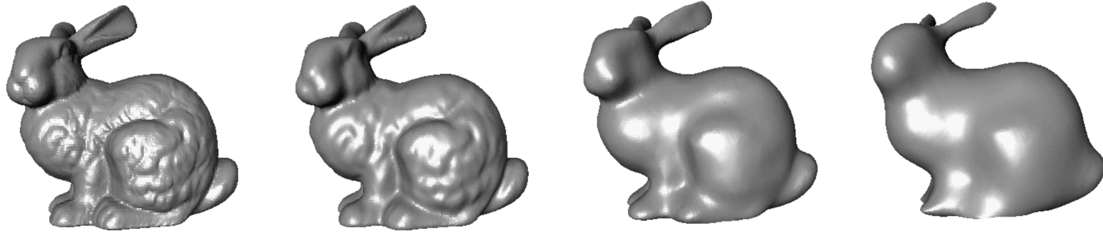

Figure 2.4: **Multi-Resolution Hierarchies.** Four resolutions of the Stanford Bunny, varying from fine to coarse. (From [13])

There were many other works in the same direction. In the next two Sections we will review the one most closely related to our approach, and the one we consider to be the state-of-art in the subject.

2.2.1 Laplacian Surface Editing

Sorkine et al. [19] represent detail by replacing the coordinates of each vertex v_i by the difference from its position to the average of the positions of its neighbors. The difference to the coordinate of a vertex to the centroid of its neighbors is called Laplacian coordinate of v_i [12], [2].

Let A be the matrix that encodes the adjacency of the mesh

$$A_{ij} = \left\{ \begin{array}{l} 1,\ if\ v_i\ and\ v_j\ are\ connected\ by\ an\ edge \\ 0,\ otherwise \end{array} \right\} \tag{2.1}$$

and D be a diagonal matrix where each element D_{ii} is the degree of the vertex v_i, we can encode the calculation of the Laplacians of a mesh as $L = I - D^{-1}A$. The matrix L is called the Laplacian operator.

In Figure 2.5 we show a triangle mesh and the respective matrices. Note that the inverse of a diagonal matrix is the matrix with the non-zero elements replaced by their multiplicative inverses ($x \to 1/x$).

The matrix L has rank $n-1$, where n is the number of the vertices. This means that the linear system of equations that solves the geometry of a mesh from its Laplacian coordinates, *i.e.*, $Lx = b$, is underdetermined. The geometrical interpretation for that is straightforward: Laplacian coordinates are invariant to translation, so we can only reconstruct the mesh from its Laplacians if we specify where the mesh will be (fixing at least one vertex to a static position, $v_i = u_i$).

We can even have additional constraints that model, for example, editions applied to an object by the user. In this case, we would convert user modifications into

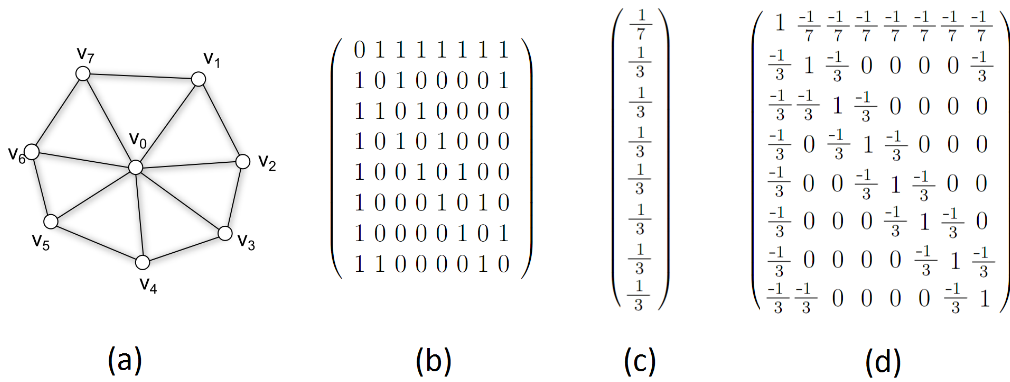

Figure 2.5: **Laplacian Coordinates.** (a) Polygonal Mesh. (b) Adjacency Matrix A (c) Diagonal of Matrix D^{-1} (d) Laplacian Operator

constraints of the type $v'_i = u_i$, where v'_i is the position of vertex v_i to be optimized and u_i is the user provided position. The coordinates of the object would be solved by minimizing the following error function on V' (the set of all vertices)

$$Error(V') = \sum_{i=1}^{n} (L(v'_i) - L(v_i))^2 + \sum_{j=1}^{k} (v'_j - u_j)^2 \tag{2.2}$$

where $L(v'_i) - L(v_i)$ is the difference between the Laplacians before (v) and after the optimization (v'), and $v'_j - u_j$ is the distance from the optimized vertex position to the user provided one. Note that the first sum goes from 1 to n, the number of vertices in the mesh, and the second one from 1 to k, the number of provided constraints.

This is the base of the edition of surfaces preserving details: to respect constraints given by editions while keeping the local geometry.

To represent detail as an intrinsic part of a model, it would be necessary to expand Laplacians' invariance to rotations (a rotated model is clearly still the same object and we would like the representation of its details to stay the same). The solution presented by [19] is to calculate a transformation T_i for each vertex v_i based on the position of its neighbors after the optimization. Using the notation we applied in the last equation, we would find a transformation that maps v_i and its neighbors to v'_i and its neighbors. The equation would be modified as follows:

$$Error(V') = \sum_{i=1}^{n} (T_i \times L(v'_i) - L(v_i))^2 + \sum_{j=1}^{k} (v'_j - u_j)^2 \tag{2.3}$$

The authors restrict the transformations to rotations and isotropic scaling. That way, the T matrix can be represented as $T = s \times exp(H)$, where s is a scalar and H is restricted to be a skew-symmetric matrix. The exponential of a matrix is the matrix analogous to the ordinary exponential function, and can be calculated by expanding a power series. Deeper mathematical background is out of the scope of this work (refer to [3]). The authors rely on some properties of skew-symmetric matrices (refer to [19] for more information) to derive the following approximate representation for T.

$$T = \begin{bmatrix} s & -h_3 & h_2 & t_x \\ h_3 & s & -h_1 & t_y \\ -h_2 & h_1 & s & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

The t_x, t_y and t_z values encode the translational part, while s and h encode the scaling and rotation. These variables do not have a meaningful interpretation when analyzed individually. This matrix is a good approximation for isotropic scaling and small rotations. It was necessary to use an approximate matrix, since the exact matrix would not depend linearly in V'. Now we can the write the linear dependency of T_i (the T matrix, calculated for vertex v'_i) on v'_i as minimizing the equation

$$(A_i(s_i, h_i, t_i)_i^T - b_i)^2 \tag{2.5}$$

where A_i contains v_i and its neighbors and b_i contains v'_i and its transformed neighbors. Matrix A is structured as follows, in order to express the transformation $T_i v_i$.

$$A_i = \begin{bmatrix} v_{k_x} & 0 & v_{k_z} & -v_{k_y} & 1 & 0 & 0 \\ v_{k_y} & -v_{k_z} & 0 & v_{k_x} & 0 & 1 & 0 \\ v_{k_z} & v_{k_y} & -v_{k_x} & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

Once we have the values for s, h and t, we can add them to our initial system, as in Equation 2.2. We can see some results from the technique in Figure 2.6

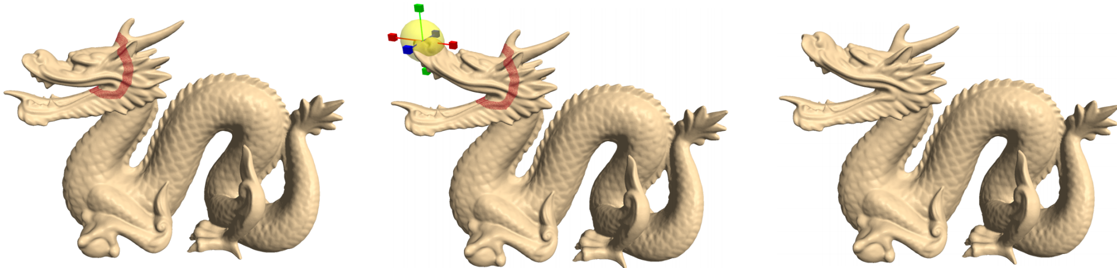

Figure 2.6: **Results of Laplacian Surface Editing.** Deforming Dragon's upper lip. (From [19]).

The authors present other applications of their method, for example creating models with the base geometry of one object and the details of another. We will not discuss these applications, since they do not add much to the method itself, and are not relevant to our work.

2.2.2 Coupled Prisms for Intuitive Surface Modeling

Rather recently, Botsch et al. [5] introduced a system to create physically plausible deformations. The system is based on the usage of prisms connected by elastic forces. One of the main advantages is that the method will prevent degenerations (that the mesh intersects itself, for example) even in extreme cases. Also, while former methods often had problems handling substantial transformations, this system

can handle them elegantly. The most significant disadvantage is that it is computationally more expensive than other methods for the same problem.

The first step of the technique is to attach a set of prisms to the mesh, as shown in Figure 2.7a). Users' interaction will be applied to the prisms and the position and orientation of the prisms will be converted to mesh deformation.

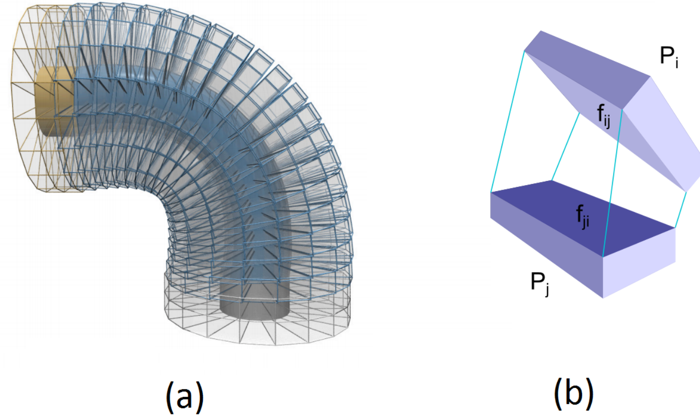

Figure 2.7: **PriMo: Coupled Prisms for Interactive Surface Modeling.** (a) Prisms configuration around an object (From [5]). (b) As objects are deformed, elastic forces stretch.

The configurations generated by user interaction are likely to be extreme, which often leads to the degeneration of the mesh. To prevent prisms from degenerating, they are made rigid, and connected by elastic joints, that will stretch as the user deforms the object. The deformation of the set of prisms is solved by minimizing an energy function, more specifically, the volume between the two prisms.

$$E_{ij} = \int_{u=0}^{u=1} \int_{v=0}^{v=1} \|f_{ij} - f_{ji}\| dvdu \tag{2.7}$$

where f_{ij} is the retangular bi-linear patch between $f_{ij}(u,v)$, for $(u,v) \in [0,1]^2$, that interpolates the four elastic forces in the face from P_i facing P_j. f_{ji} is the analogous construction in P_j. This can be seen in Figure 2.7(b). To create the global energy function, the authors sum the volumes pairwise, weighting each of them by the length of the correspondent faces F_i and F_j in the mesh and their shared $edge_{ij}$ (Equation 2.8). Note the difference between f_{ij} and F_i: while f_{ij} is the face of a prism, F_i is the correspondent face on the mesh.

$$E = \sum_{i,j} E_{ij} w_{ij}, w_{ij} = \frac{e_{ij}}{|F_i| + |F_j|} \tag{2.8}$$

The system works as follows: the user selects a subset of the prisms and determine their positions. The position of other prisms are calculated by the minimization of the Equation 2.7. Once we have minimized the energy and found the optimal position for the prisms, the position of an unconstrained vertex (as opposed to the vertices that are constrained by the user) is calculated by transforming it by the average of the transformations of its incident prisms. In Figure 2.8 we can see some results of the technique.

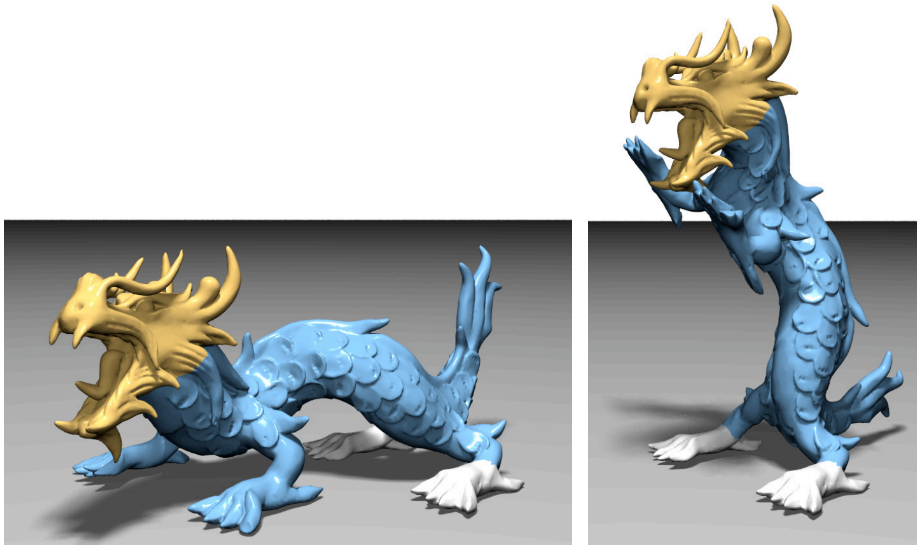

Figure 2.8: **Results of PriMo: Coupled Prims for Intuitive Surface Modeling .** The Dragon was updated by fixing the feet (in white) and moving the head up (yellow) (From [5]).

2.3 Summary

In this chapter we reviewed some works that are closely related to ours. We focused on View-Dependent Geometry and Laplacian Surface Editing, since the comprehension of these works is essential to understand our project.

3 VIEW-CONSISTENT MESHES

Views of a three-dimensional object are generated by projecting its vertices onto a two-dimensional view plane. When projecting, we lose information about depth. Thus, theoretically, a view of the object only gives us two-dimensional information about the position of its vertices (in practice shading will give us some information about the vertex position in relation to its neighbors). Our project attemps to solve conflicts between the views of the object and its geometrical representation, by adding a new degree of freedom to the optimization: each vertex is allowed to move along its projective line, as long as its relative position to the neighbors vertices stays roughly the same.

We will represent each point by the line that goes from the center of projection exactly through it. In order to add the curvature factor to the equation (needed to keep the shading of the projection unchanged), we will minimize the difference from the initial Laplacians to the Laplacians after the deformation.

3.1 View-Consistent Meshes

The basic idea that motivated our project is the fact that a position on the screen really only restricts the position of the vertex to the projective line going from the camera's center of projection through that screen point. In other words, a vertex would be constrained in space by an equation of the form:

$$v_i \approx e + s_i r_i, \|r_i\| = 1 \tag{3.1}$$

where e is the center of projection and r_i the direction of the ray going through vertex v_i. The variable s_i is a multiplier of the ray and will represent the distance of the vertex to the center of projection. This distance will be added to the optimization. When we discussed Laplacian Surface Editing, we had to minimize:

$$Error(V') = \sum_{i=1}^{n} (L(v'_i) - L(v_i))^2 + \sum_{j=1}^{k} (v'_j - u_j)^2 \tag{3.2}$$

where $L(v'_i) - L(v_i)$ is the difference between the Laplacians before (v) and after the optimization (v'), and $v'_j - u_j$ is the distance from the optimized vertex position to the user provided one. Note that n is the number of vertices in the mesh and k the number of provided constraints.

We will apply a very similar algorithm. The only difference will be that we are now using the projective lines as the constraints. The equation will be modified to:

$$Error(V') = \sum_{i=1}^{n} (L(v_i') - L(v_i))^2 + \sum_{j=1}^{k} (v_j - e - s_j r_j)^2 \qquad (3.3)$$

Minimizing the quadratic distance from a vertex to a ray, $\|v_j - e - s_j r_j\|^2$, is still a linear problem, not only in v_j but also in s_j. The usage of lines instead of points causes a problem, since for each additional constraint we are also adding a new variable s, which means that the Laplacian will continue to have rank $n-1$. We can solve that either by adding "anchor" constraints, a small number of constraints of the type $v_j \approx u_j$ or by using perspective projection (only in orthographic projection we can move the object along the projective lines without affecting the relation between neighbors). Also, if we set the equations for the x, y and z coordinates of the vertex in the same system, this will solve the problem, because every new line will add three equations (one for each dimension) and only one variable.

In our implementation, we used orthographic projection and solved the problem by having all dimensions in one system. We also needed the anchor variables for specific application purposes, in both applications, so linear dependency was not an issue.

3.2 Constructing the Linear System of Equations

As we mentioned in Chapter 2, the Laplacian operator of a mesh is a matrix of rank $n-1$, where n is the number of vertices of the mesh. Since we have n variables, we need at least one additional contraint, otherwise the system of equations is underdetermined. In our case, this is made worse by the fact the each additional constraint also adds a s variable. We solved this by having all three dimensions in the same linear system.

Additionally to the line constraints, we restricted some vertices in the old $v_i \approx u_i$ way. The use of these constraints was specific to the application. In the sketching contours application (Chapter 4), we used these restrictions to avoid having to solve the position of each vertex in the mesh. In the generation of impossible geometry (Chapter 5), we used these to add the topological features that make the object impossible. We will discuss this in more detail when explaining both applications.

In the end, the linear system of equations (without the anchor constraints) looks as following:

$$\begin{pmatrix} & & L & & \\ & & & & \\ & & \text{-----------} & & \\ e_j.x & \dots\ 0 & \dots\ 0 & \dots & -r_{ij}.x \\ 0 & \dots\ e_j.y & \dots\ 0 & \dots & -r_{ij}.y \\ 0 & \dots\ 0 & \dots\ e_j.z & \dots & -r_{ij}.z \end{pmatrix} \begin{pmatrix} x_i \\ \vdots \\ y_i \\ \vdots \\ z_i \\ \vdots \\ s_j \\ \vdots \end{pmatrix} = \begin{pmatrix} L(v_i)_x \\ \vdots \\ L(v_i)_y \\ \vdots \\ L(v_i)_z \\ \vdots \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad (3.4)$$

where e_j is the eye position in the view j and r_{ij} is the ray going from the eye e_j through the vertex v_i. To change the contribution of each constraint to the final position of a vertex, we can multiply the whole equation by a constant.

We solved the system using the Cholesky Decomposition [9] provided in the TAUCS Library [20]. Solving the linear system of equations is the performance bottleneck of our framework, a transition to the state-of-art library Cholmod [7] or to an implementation that uses GPUs is a possible future improvement.

3.3 Summary

In this chapter, we described the main idea that motivated our technique. This idea is straightforward, yet it gives us a powerful tool to describe the look of an object, while restricting its geometry the least we can. In the next two chapters, we will see applications where we demonstrate the possibilities enabled by our technique.

4 SKETCHING CONTOURS

In this chapter, we discuss the first application of our method. We extended an existent system introduced by [22], where sketches are used to edit a three-dimensional model preserving its details. The user will make a sketch over a mesh, and the mesh will deform accordingly. We can see some results in Figure 4.1.

Since we are using an edition that preserves detail, we have to make an important distinction between two different kinds of optimizations: the one that optimizes single-view geometry with respect to user editions and the one that optimizes the consensus mesh in respect to every view. In the next section, we will discuss the pipeline we followed to perform the first one. The latter will be discussed in the following sections.

4.1 Modified Framework for Single-View

Most of the pipeline we will follow to edit meshes is derived from [22]. In the existent work, one of the main goals is to make the whole process transparent to the user, and efficient enough to be interactive. Our goal is to edit the view as closely as possible to user's intention, so we can focus on creating the consensual geometry and interpolating it. We replaced automatic estimation at points where it was not reliable enough. By not reliable, we mean that very similar inputs generated different results, which made it hard to evaluate the effect of changes in the second part of the algorithm. The overview of the modified pipeline can be seen in Figure 4.1.

According to [10], humans recognize objects by its feature lines. The sketch-based editing uses this assumption. Thus, the first step of the algorithm is to detect the silhouettes of the object. The user only has information about the projection of the object, so the most obvious approach is to detect the main lines in image-space. The silhouettes have to be segmented in meaningful parts, such that they have an adequate 3D correspondent. Even when two vertices that are part of the silhouette are neighbors in image space, they may be distant in the model.

After showing the user which feature lines the algorithm detected, we ask the user to choose the one he wants to deform. The user will also inform which is the region of interest, *i.e.* the region of the object that should be affected by the operation. The region of interest is necessary to prevent the object from deforming globally, as will be discussed in Section 4.4. These two steps were modified from the original framework because the automatic estimation often did not generate the results we expected.

The user will then make a sketch, to inform where he wants the selected feature line to go. The system will automatically find correspondences between the vertices

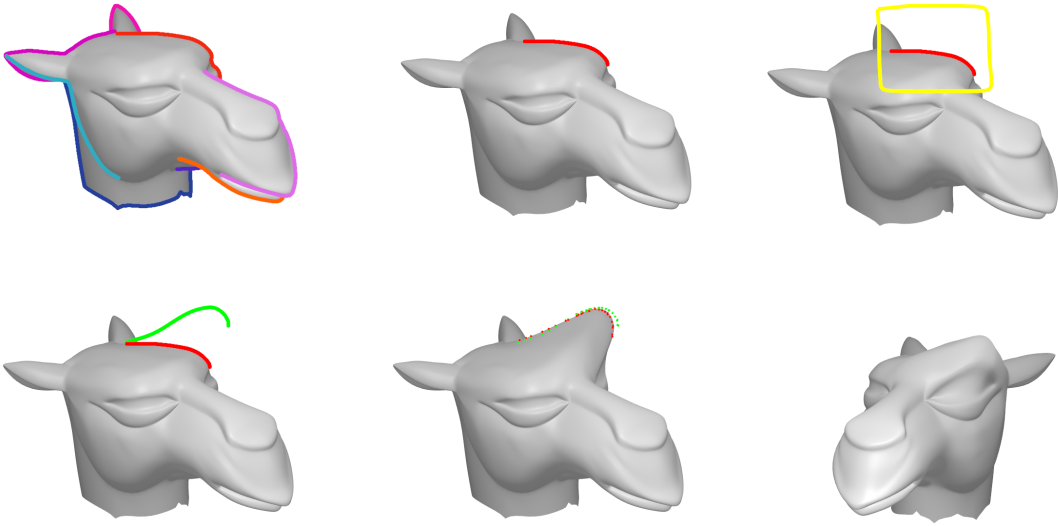

Figure 4.1: **Pipeline of Sketching Contours.** (a) Automatically detect silhouettes. (b) User provides handle. (c) User provides Region of Interest. (d) Sketch. (e) Vertex Correspondences. (f) Deformation.

in the mesh and positions in the sketch. Finally, the mesh will be deformed to respect the sketch, while trying to preserve the details of the shape as much as possible. In the next sections we will describe every step presented in more detail.

4.1.1 Silhouette Detection

To detect the silhouettes, we used the algorithm suggested by [22]. The final result we want to achieve is a set of disjoint lines, each one representing a meaningful part of the object.

To find the parts of the object that consist of feature lines, we simply detected the highest frequencies in the depth map (Z-Buffer), using a Laplacian filter (note the difference between a Laplacian filter, applied to 2D images, and the Laplacian operator, applied to 3D meshes). We use a 3x3 Laplacian filter, as we can see below.

$$L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{4.1}$$

A pixel is determined to be part of the silhouette if the difference from its depth to the depth of its neighbors (the result of the Laplacian) is greater than a small constant ϵ.

$$isSilhouette(p) = L_{xy}(depthMap[p]) > \epsilon \tag{4.2}$$

After determining which pixels are part of the silhouette, it is necessary to segment these pixels into lines that make sense in the mesh. The first condition used by the authors is the depth difference between elements - pixel paths that are continuous in 2D may have discontinuities in 3D, and vertex paths that are not continuous in 3D should not be part of the same silhouette segment. It is possible though,

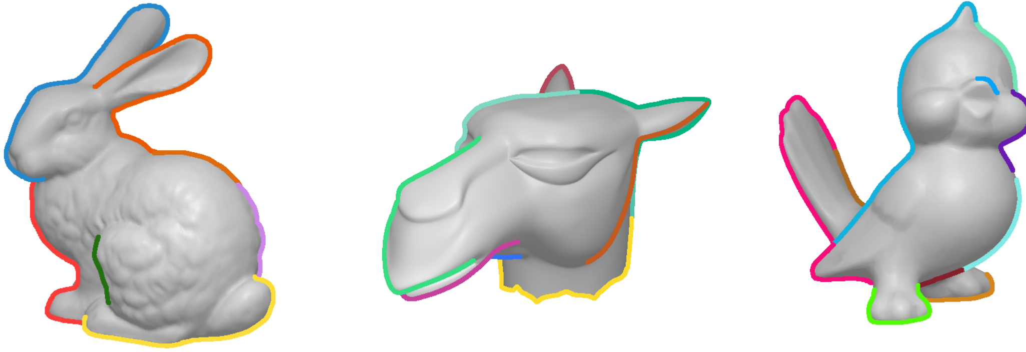

Figure 4.2: **Silhouette Detection.** Results of silhouette detection algorithm applied to Stanford Bunny, Camel and Tweety.

that more than one neighbor pixel obey this condition. To handle these situations, the pixels are inserted in a priority queue ordered by curvature, where pixels with less curvature (*i.e.* that form a straight line) come first. Some results of silhouette detection and segmentation can be seen in Figure 4.2.

4.1.2 Handle Estimation

At this point, we decided that the algorithm used in [22] often generates unexpected results. This happens mostly because the sketch can be very different from the silhouette (for extreme deformations). Since handle estimation is an activity where user interaction can quickly provide a good answer, we ask the user for input in our framework.

It is worth mentioning, however, that the algorithm we use is the same used by [22] (we will review the algorithm in this section). The authors tested the similarity from the handles to the user sketch, whereas we tested the similarity to the user input of which handle he desired. By asking the user to draw exactly over the feature line, we avoid having to decide between choosing handles that are similar to the sketch or handles that are near it. A user study conducted in the original work showed that there is no right answer for this choice.

We used the algorithm by Cohen and Guibas [6]. The method extracts the cumulative turning angle and the length of the edges from the two lines.

$$length_i = \|edge_i\|, tAngle_i = \left\{ \begin{array}{ll} 0, & i = 0 \\ edge_i - edge_{i-1} + t_{i-1}, & i \neq 0 \end{array} \right\} \quad (4.3)$$

We can see the graphical representation of two lines in Figure 4.3 (Turning Angle Summaries - TAS). The blue part of the red line is the most similar to the green line. We compute the similarity of two line segments by estimating the difference between a scaled, translated and/or rotated version of one of the lines and the other.

Before testing the similarity between the two lines, we simplified the user provided line using the algorithm by Douglas and Peucker [8]. The algorithm starts with a bad estimation of the line and iteratively improves it. At each step, the algorithm chooses the vertex v that is most distant to the line approximation, and divides the vertices in "right from v" and "left from v". The algorithm is then

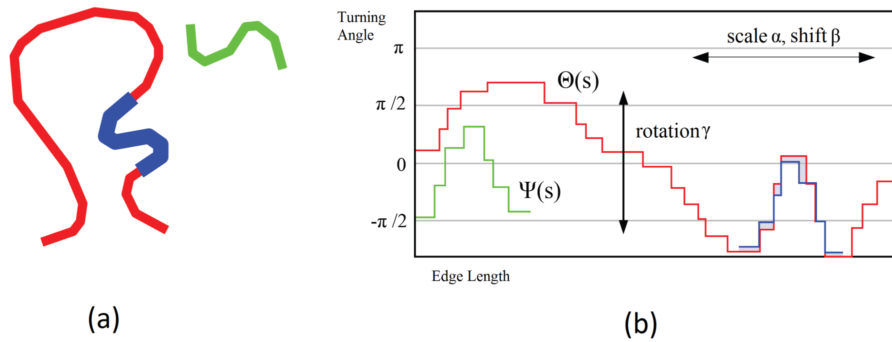

Figure 4.3: **Matching two lines.** (a) Result. (b) Tuple of edge lengths and turning angles. (From [22])

recursively applied to each side, until there are no more vertices more distant than a provided ϵ.

4.1.3 Handle-Target Correspondences

Given the handle and the sketch, we need to decide which vertices are the three-dimensional correspondents to the feature line we selected and where we are going to position them.

We start by finding the vertices. Based on the 2D position of the handle pixels and on the depth map, we are able to create a 3D bounding volume that will contain some of the vertices that are near the projected pixel in 3D space. These are the handle vertices, that will be mapped to sketched positions. To create a more robust set of handle vertices, we also add every vertex v that respects the two following conditions:

- **Adjacency:** v is adjacent to the at least one of the vertices already classified as handle.

- **Image-Space Proximity:** when projected to the image-space, v is contained inside the 2D area formed by the projection of the 3D bounding volume.

In Figure 4.4 we can see the effect of the handle growth. The first handles are sparse and randomly placed over the region whereas the new handle set is more compact and will create smoother deformations.

To decide where in space the handle vertices should go, we need to map from their distance to the chosen feature line to a distance from the sketch. Specifically, we determine the coordinate of each vertex to be the position it occupies in relation to the total length of the line. We parametrized both the feature line and the sketch from [0,1], i. e. a point in the middle of the line will have the coordinate 0.5. In this way, we can map the middle point of the handle to the middle point of the sketch and so forth.

4.1.4 Region of Interest Estimation

The region of interest estimation used in the existent pipeline often had such an influence on the results, that it was hard to evaluate whether changes in other

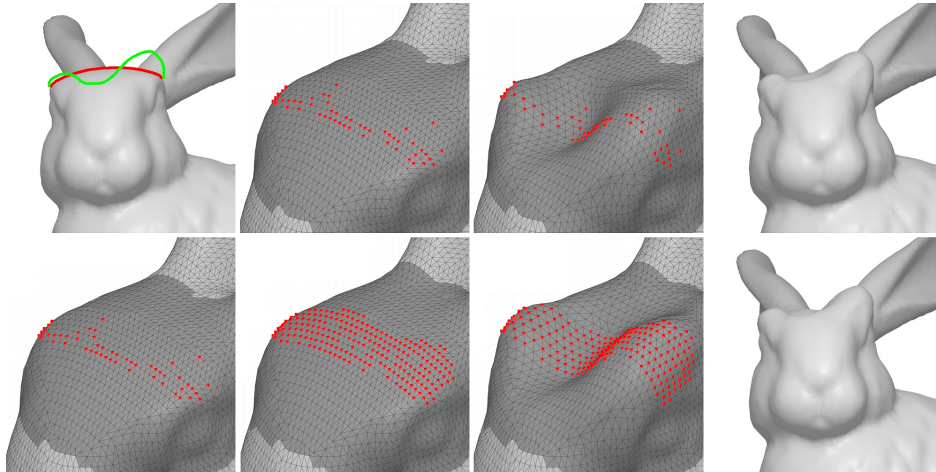

Figure 4.4: **Handle Growth.** Top row - deformation without growing the handles; bottom row - improved handle set using the region growth. (From [22])

parts of the technique were having the expected effect. Also, it only considered the vertices on the surface (and not their future positions), which generated bad results when the user deformed the vertices to go inside the mesh (see Figure 4.5).

We changed the pipeline to allow the user to provide the region of interest, by drawing a sketch around the desired region in the model. To test whether a vertex is part of the selected region, we create a polygon from the sketch segments and test whether the projection of the vertex in the screen is inside this polygon. We used the CGAL Library [1] to perform this operation.

It would be reasonable to use the depth of a vertex to check whether the user can see it, before adding it to the region of interest. We did not implement this check because handle estimation already makes sure we will not deform vertices that are not visible from our point of view.

The region of interest is important to prevent the mesh from deforming too much or in areas not intended by the user. Also, since Laplacians are invariant to translation, it may happen that moving the whole mesh toward the sketch is a better approach than deforming it, from the optimization point of view. This behavior would be unintuitive for the users and, as such, should be avoided.

To guarantee that the region of interest will be the only part of the mesh to be deformed, we add anchor constraints, *i.e.*, equations of the form $v_i = u_i$, that tell a vertex to stay where it is. The vertices constrained by these equations are inserted in the border of the region of interest forming a ring that separates the part of the object to be optimized and the part supposed to stay the same.

Having anchor vertices also allows us to improve performance significantly, since we do not need to add the vertices outside the ring to the linear system anymore. Note that this would not be possible if we did not have these constraints because the vertices in the system would not take the other ones into account, generating discontinuities. An example showing an anchors' ring can be seen in Figure 4.6.

4.1.5 Mesh Deformation

The mesh deformation is done in the same way as in Laplacian Surface Editing. We add a Laplacian matrix and use the handle-target the correspondence (handle

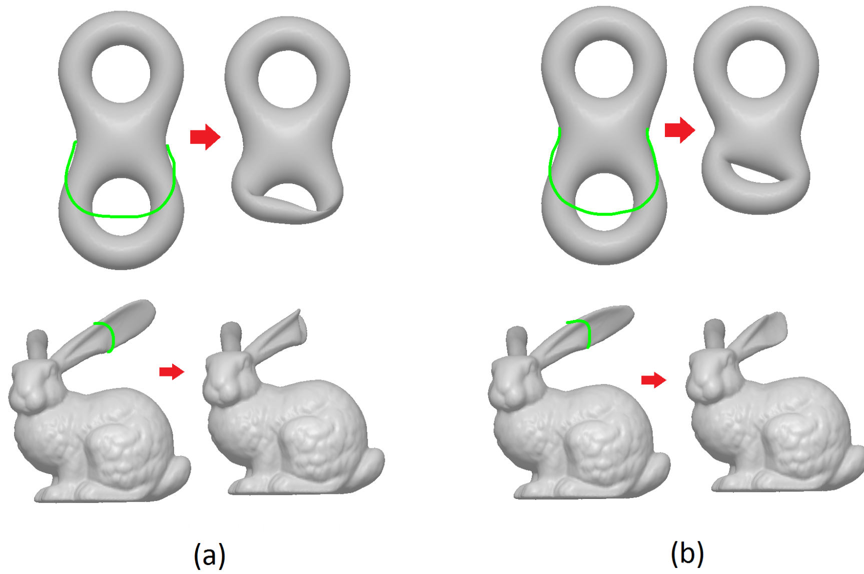

Figure 4.5: **Problems estimating the Region of Interest.** (a) Result using [22]. (b) Result using our modification, asking the user for input.

vertices' position should equal the target) and the anchors (anchor vertices' position should stay the same) as $v_i \approx u_i$ equations. In our framework, we did not use the rotation and scaling invariance presented in [19], but the results were satisfactory anyway.

4.2 Consensual Mesh Generation

In the second part of our algorithm, we assume we have many different meshes, already positioned in R^3. Note this is the same configuration as in View-Dependent Geometry.

We are now able to apply the concept we developed in Chapter 3, that each vertex position only constrains the vertex to a half-line. To calculate the position of a vertex at the consensual mesh, we compute a line going through the vertex at each view. These are the constraints that we are going to use when optimizing the mesh using the Laplacians.

4.2.1 Generating the Consensus Mesh

Generating the consensual mesh is not very different from applying user transformations to single-view meshes. We will create a system like the one presented in Chapter 3. The Laplacian matrix used will contain the Laplacian coordinates calculated in the mesh before doing any optimization. We will also restrict the vertices according to their positions in the views.

The first issue is how to decide which will be the region of interest of the consensual mesh. The first implementation was to use the whole set of vertices as the region to be deformed. This is possible since optimization between views differs from the one applied for a sketch. For example, many of the vertices will be asked to stay where they are, which prevents the mesh from respecting another view by translating in its direction.

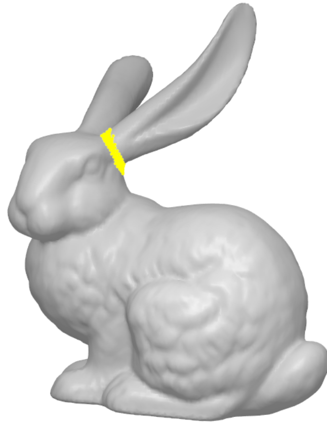

Figure 4.6: **Region of Interest and Anchor Vertices.** To deform the ear of the bunny, we fix the base so that only the positions of the ear need to be calculated.

The problem on using the whole set of vertices as the region of interest is performance, since we will be adding many vertices to the system that are not supposed to move at all. As an alternative, we decided to use the region of interest as the union of the regions of interest from each view. In the worst case, it will lead to the whole mesh, but we found that the performance gain was worth it. For a small number of deformations in the Stanford Bunny, we improve response time from 20 to 2 seconds. We added anchors in the border of the region of interest, as we did in single-view optimization. We are not worried about the mesh deforming too much anymore, but we still need to prevent discontinuities between vertices that are part of the system (and as such are going to be updated) and the ones that are not.

4.2.2 A Note on Implementation

There are many different options on how to represent a triangle mesh. Some data structures will prioritize small size, while others will focus on efficiently providing information about the mesh. We chose the implementation of the Half-Edge Data Structure [16] in the OpenMesh Library [4]. The Half-Edge structure provides very efficient queries about the topology of the mesh.

Another important observation is that the meshes that represent each view share the same topology. To save memory, we use a single half-edge structure and different geometries, *i.e.*, arrays containing the positions of vertices and normals.

4.3 View Interpolation

There are some cases where the representation using only one mesh is not adequate. In these situations, to render the object as observed from an arbitrary viewpoint, we need to calculate the interpolation between two views. Our first approach to the interpolation of views is the same as in [18].

We start with a set of key views (containing the respective meshes), and a point in space where the viewer is located. We calculate the convex hull of the points where the key views are located, and then find the intersection of the hull with a ray going from the viewer position in the direction of the object. For now, we will assume that the convex hull of the views covers all possible directions; if this is not

true, we must have an alternative for cases where the ray will not intersect it.

There are three possible cases:

- If the ray intersects the hull exactly on a vertex, the key view at that point is the one to be rendered;
- If the ray intersects the hull on an edge, the two key views that are connected by the edge should be linearly interpolated;
- If the ray intersects the hull on a face, the three key views that are connected by the face should be linearly interpolated.

An elegant solution that generalizes these three cases is to use the barycentric coordinates of the face. The barycentric coordinates of a point are the proportion between the area of the face and the area of the triangles formed by the three edges and the point. This can be seen in Figure 2.3, in Chapter 2. As we can see, the barycentric coordinates provide a robust solution to weight the contribution of each view in the final position of the vertices.

A second approach, would be to use the direction of the constraints to measure how far a key view is from the arbitrary view. The dot product of the vectors, for example, could give this information. The linear system could be solved again using these weights. Due to time restrictions, we were not able to further investigate this alternative.

4.4 Results

We achieved positive results applying our technique. Many objects that have conflicting appearance may be represented using a single geometry. As we can see in Figure 4.7, we show two views of the Stanford Bunny (top row), and the correspondent consensual mesh (bottom row), when seen from the respective key positions. We believe this is an useful extension of the View-Dependent Geometry work [18].

In Figure 4.8, we can see some results for the interpolation of the views. Besides generating meshes for the arbitrary view, interpolation between meshes of same topology is useful in many other applications. In our result, for example, we can see the interpolation between different features of the object.

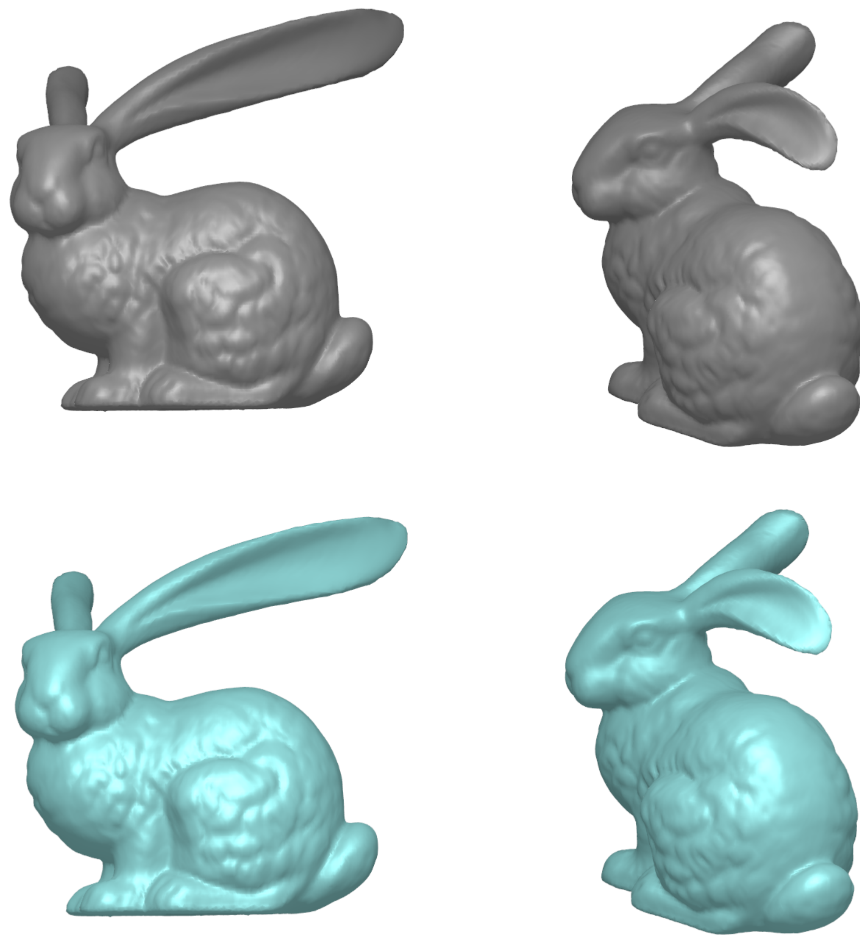

Figure 4.7: **Results of applying View-Consistent Meshes to Sketching Contours Framework.** Top row - each separated view; Bottom row - representation using a single mesh.

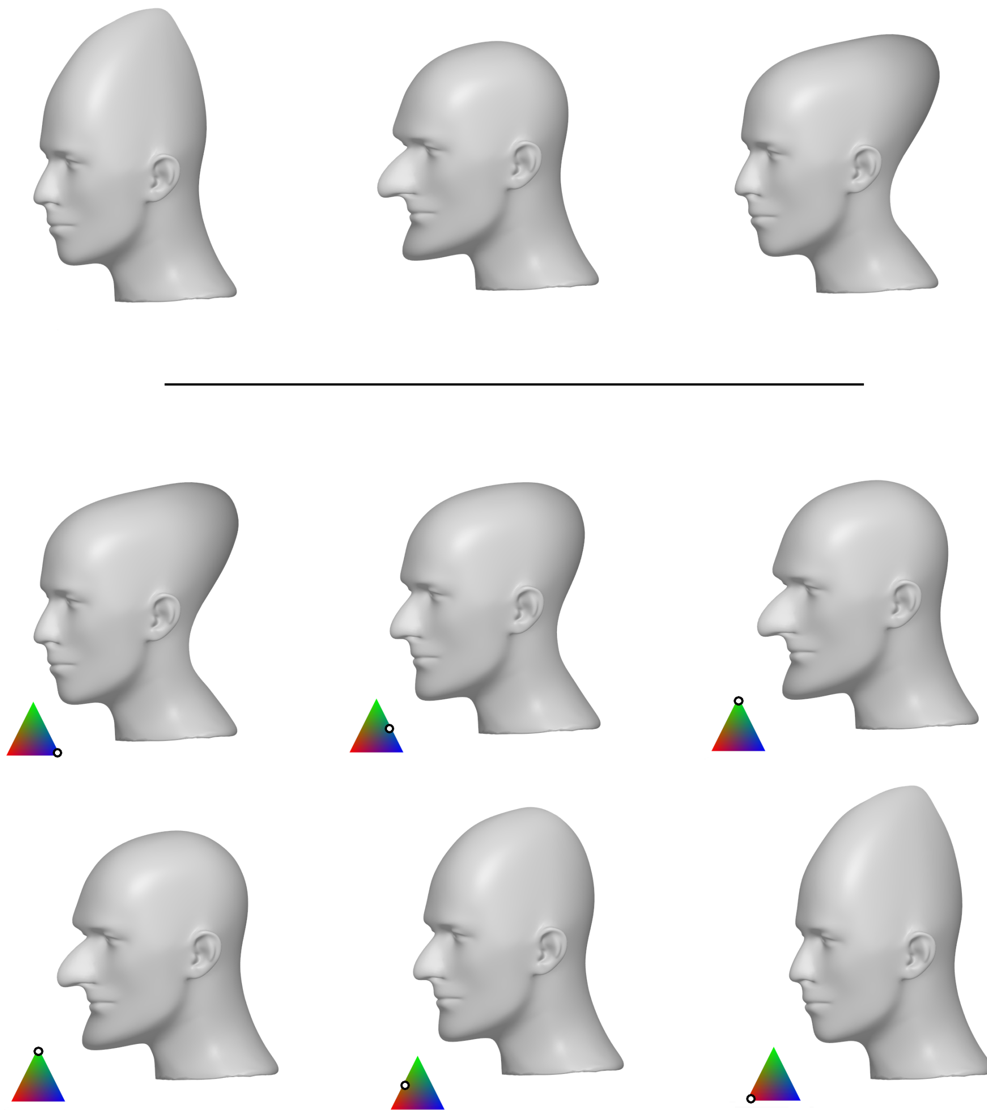

Figure 4.8: **Results of the Interpolation between Views.** Top - Key meshes. Bottom - Interpolation result and respective position in the face.

5 GENERATING IMPOSSIBLE GEOMETRY

An interesting application of our method is to study views that suggest impossible geometry, even when not conflicting with other views. Examples of that are the PenRose Triangle and Escher's Impossible Cube. To optimize one view to look as though generated by this kind of object we must change our pipeline.

Since the objects desired are not feasible, we cannot start from a mesh that respects the specified view. Instead the user will provide some points of the object and place them in space as they are projected on the image. There is no need to provide information about depth, the optimal depth of each point will be found by our algorithm. Each point provided will be represented by a cube. This representation is important for adding the topological constraints in the next step.

After placing the points, the user will connect the faces of the cubes by strings. These strings represent the topological constraints, *i.e.*, the connectivity of the points. Finally, the user will select which view should be optimized to look like the impossible object. Some views of the setup process for a Penrose Triangle can be seen in Figure 5.1.

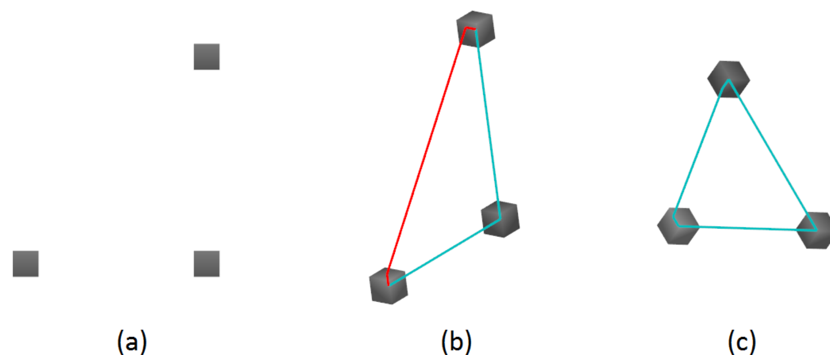

Figure 5.1: **Setting up a Penrose Triangle.** (a) Add points (b) Connect point faces by strings (c) Select view to be optimized

5.1 The Algorithm

The optimization will be done in two steps. At the first step, we will optimize the positions of the points along their projective lines. After that we will find the strings with least curvature that can be used without changing the projection. We will discuss each part separately in the next sections.

5.1.1 Optimizing point position

The first step is to optimize the positions of user provided points along the respective projective lines. The goal of this step is to create a better environment for the string optimization. Since we are creating “impossible objects”, topology constraints can conflict with the positions a user intuitively gives to the points. A bad positioning would diminish the quality of the final result.

To represent the constraints, we will use the tools we developed in Chapter 3. Each point will be restricted to a line going from the center of projection through it. Also, we want the points to be located where they minimize the curvature of hypothetical strings connecting the faces. To formalize this, we set the Laplacians of the string to zero. In Figure 5.5 we can see the result of a first step optimization. The purple lines represent the projective lines while the blue line represents the “minimal curvature”. The result is not optimal, as we will discuss in Section 5.2.

5.1.2 Optimizing string curvature

To optimize string curvature, we first need to map it to the projective plane. By doing this, we do not need to worry about the projection anymore, since we are working in the projective space of the string. The projective space of a curve that does not have self-intersections is homeomorphic to a plane (when using orthographic projection).

The first step is to convert the coordinates of the string to the coordinate system of the view we want to optimize. We mapped the 3D string to a 2D curve, using as (x,y) coordinates the arc-length of the projection($\sqrt{x^2+y^2}$) and the depth(z), respectively. The geometric interpretation would be to map the projective space of the string to a plane, as shown in Figure 5.2.

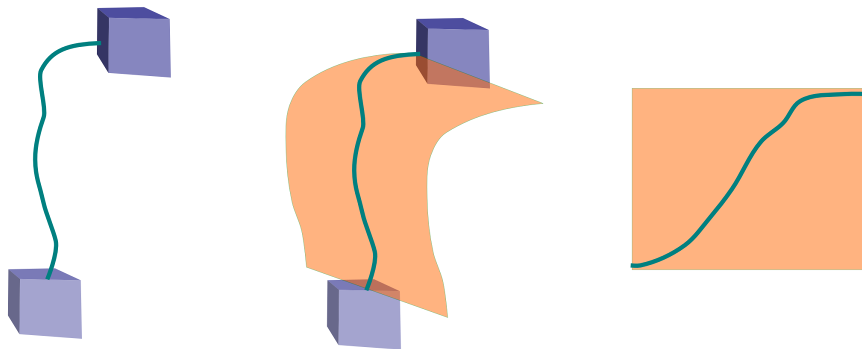

Figure 5.2: **Mapping a String to 2D.**(a) String in R^3 (b) Projective Space of the String (c) Projective space mapped to a plane.

Once again, we used Laplacians = 0 to minimize the curvature between two points. We could not have solved the problem in 3D, however, because our mapping is what avoids the necessity of additional constraints to respect the projection.

After optimizing the curve, we can map it back to R^3. We only need to update the z coordinate and map back to the coordinate system we were using before the optimization.

5.1.3 Creating the mesh from the strings

Once we have placed the strings, we need to create a mesh around it. We already have the vertices of the cubes. To complete the mesh, we go through each connection between faces and add vertices around the string.

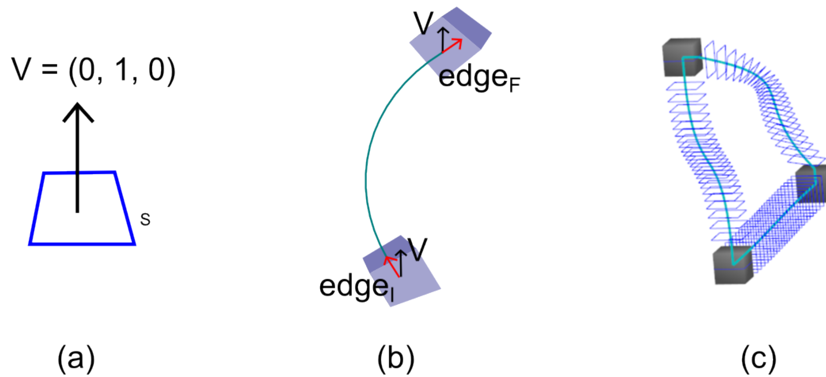

Figure 5.3: **Creating a mesh from the strings.** (a) Square to be rotated. (b) First and last edges of a connection are used to calculate the orientation of the first and last squares. (c) Quaternions are used to linearly interpolate the two orientations.

We start with a square S in the xz plane. Given the initial ($edge_i$) and the final edges ($edge_f$) of the connection, we compute quaternions $\hat{\mathrm{Q}}$ and $\hat{\mathrm{R}}$ with the rotation transform that would rotate a vector $v(0.0, 1.0, 0.0)$ into the direction of the edge. These are the transformations we will use to rotate the square along the string. If the quaternions are different, they can be interpolated along the string. We can see this operation in Figure 5.3.

5.2 Limitations

The most relevant problem we found in our technique is that the standard technique for minimizing curvature, setting Laplacians to zero, led to problems. When connecting four points by a string whose Laplacian is minimized, we expected that the result would always be the minimal curvature. The problem, however, is that even a straight line may have Laplacians different from zero, if the points are not evenly distributed.

As we discussed in Chapter 2, the definition of the Laplacian coordinate is the difference between the position of a vertex and the centroid of its neighbors. Even in a straight line, it is possible that the vertex does not lie exactly in the centroid of its neighbors, as shown in Figure 5.4.

In our technique, we have to fix the position of the point inside the cube and the one going through the face, when optimizing the strings. When we fix the distance between these two points, the only way to guarantee that all vertices on the line are evenly distributed would be if we knew the length of string. The length, however, is solved dynamically by our system, such that there is no way of calculating the ideal number of vertices before the optimization. Figure 5.5 shows the result of our "curvature minimization" to exemplify the problem. The result we would like to have is a straight line.

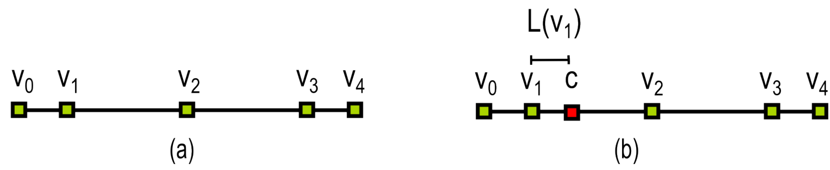

Figure 5.4: **Problem using Laplacian = 0 to minimize curvature.** (a) Straight line (b) Laplacian coordinate of vertex v_1 (difference from v_1 to centroid c) is not zero

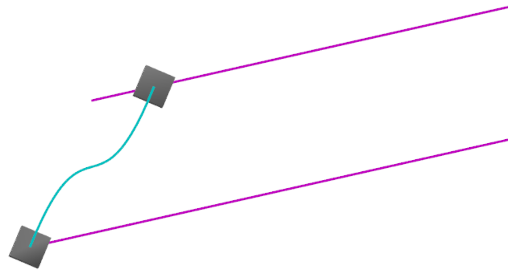

Figure 5.5: **Problem minimizing curvature.** The purple lines represent the projective lines while the blue line represents the "minimal curvature". The ideal solution would be a straight line (not moving the cube above along its projective line).

5.3 Results

In Figure 5.6, we can see the result of generating a PenRose Triangle using our algorithm. The results are still far from the desired object, but demonstrate the potential of the technique.

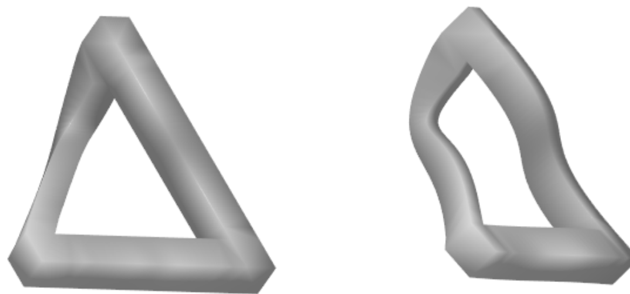

Figure 5.6: **Two views of our PenRose Triangle.** Left - Optimized View. Right - Another view.

6 CONCLUSIONS AND FUTURE WORK

We introduce a system to generate a mesh from possibly unconsistent projections of an object into two-dimensional planes. We believe that this is an advance, mainly because these models are not intuitive to create manually.

In the sketching contours application, we believe that the optimization we introduced can be used to improve the quality of the interpolation between different views of an object. Due to time restrictions, these possibilities were not explored. Also, performance could probably be improved with the use of newer libraries or even GPU implementations for the Cholesky Decomposition.

In the generation of realizable geometry that looks like impossible objects, there are still many possibilities to explore. We would like to extend our work to place complex meshes around the strings. Also, the optimization we used to minimize curvature needs to be improved.

REFERENCES

[1] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[2] Marc Alexa. Differential coordinates for mesh morphing and deformation. *The Visual Computer*, 19(2):105–114, 2003.

[3] Richard Bellman. *Introduction to matrix analysis (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[4] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. Openmesh - a generic and efficient polygon mesh data structure, 2002.

[5] Mario Botsch, Mark Pauly, Markus Gross, and Leif Kobbelt. Primo: coupled prisms for intuitive surface modeling. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, pages 11–20, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[6] Scott D. Cohen and Leonidas J. Guibas. Partial matching of planar polylines under similarity transformations. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 777–786, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[7] Tim Davis. CHOLMOD: sparse supernodal Cholesky factorization and update/downdate. http://www.cise.ufl.edu/research/sparse/cholmod, 2005.

[8] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10:112–122, 1973.

[9] James E. Gentle. *Numerical Linear Algebra for Applications in Statistics.* Springer, 1998.

[10] Donald D. Hoffman and Manish Singh. Salience of visual parts. *Cognition*, 63(1):29 – 78, 1997.

[11] B. K.P. Horn. Shape from shading: A method for obtaining the shape of a smooth opaque object from one view. Technical report, Cambridge, MA, USA, 1970.

[12] Zachi Karni and Craig Gotsman. Spectral compression of mesh geometry. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 279–286, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[13] Leif Kobbelt, Jens Vorsatz, and Hans-Peter Seidel. Multiresolution hierarchies on unstructured triangle meshes. *Comput. Geom. Theory Appl.*, 14(1-3):5–24, 1999.

[14] Kiriakos N. Kutulakos and Steven M. Seitz. A theory of shape by space carving. *Int. J. Comput. Vision*, 38(3):199–218, 2000.

[15] A. Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(2):150–162, 1994.

[16] Martti Mäntylä. *An introduction to solid modeling.* Computer Science Press, Inc., New York, NY, USA, 1987.

[17] N. J. Mitra and M. Pauly. Shadow art. *ACM Transactions on Graphics*, 28(5), 2009. to appear.

[18] Paul Rademacher. View-dependent geometry. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 439–446, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[19] O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rössl, and H.-P. Seidel. Laplacian surface editing. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 175–184, New York, NY, USA, 2004. ACM.

[20] Sivan Toledo. Taucs: A library of sparse linear solvers. http://www.tau.ac.il/ stoledo/taucs/, 2003.

[21] Ruo Zhang, Ping-Sing Tsai, James Edwin Cryer, and Mubarak Shah. Shape from shading: A survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21:690–706, August 1999.

[22] Johannes Zimmermann, Andrew Nealen, and Marc Alexa. Sketching contours. *Computers & Graphics*, 32(5):486–499, 2008.