The Cactus Project

Inès Potier & Manon Romain

August 28, 2018

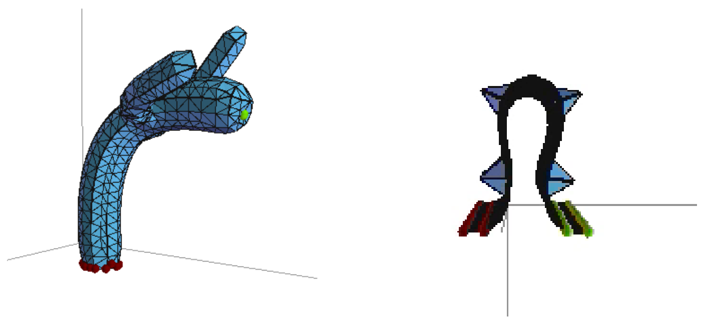

Figure 1: Example of results

1 Description of the method

The method presented in the paper *As-Rigid-As-Possible Surface Modeling* - by Olga Sorkine and Marc Alexa [SA07] - aims at constructing a non-rigid deformation framework that endeavor to locally preserve the shape of the object. This implies trying to keep small deformations as rigid as possible.

The problem considered is the following : starting from a predefined 3D-shape (comprising vertices, edges and cells), the user manually chooses a group of points he would like to fix - static anchors - and another group he wishes to move - handle anchors. The set of vertices selected corresponds to the problem's constraints. The framework then tries to modify the positions of the remaining points in order to obtain the most realistic deformation possible.

The whole approach of the paper is based on an energy measure, that is minimized at each iteration of the algorithm :

$$E(S') = \sum_{i=1}^{n} w_i E(C_i, C_{i'}) = \sum_{i=1}^{n} w_i \sum_{i \in N(i)} w_{ij} ||(p'_i - p'_j) - R_i(p_i - p_j)||^2$$

where w_i, w_{ij} are fixed cell and edge weights. The deformed vertex positions are represented by **p**, **p'**. R_i is the cell rotation.
This energy corresponds to the sum of the deviations from rigidity per cell.

Per-edge weights w_{ij} and per-cell weights w_i are present to ensure independence between the energy and the shape considered (to prevent discretization bias). They are computed as follow :

$$w_{ij} = \frac{1}{2}(\cot \alpha_{ij} + \cot \beta_{ij})$$

where α_{ij} and β_{ij} are the opposite angles of the edge (i,j).

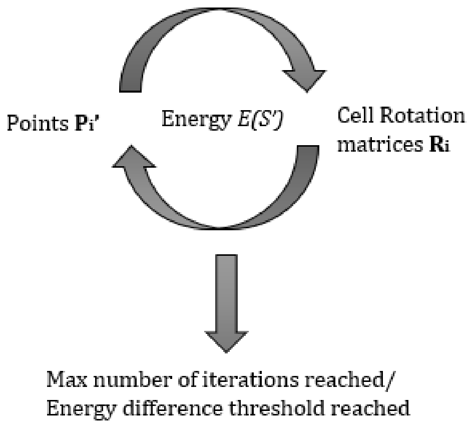
Points $\mathbf{P_i}$'
Energy $E(S')$
Cell Rotation matrices $\mathbf{R_i}$
Max number of iterations reached/ Energy difference threshold reached

Figure 2: Scheme of the algorithm operating principle

Two distinct sets of variables are alternatively optimized : the cells' rigid transformations matrices $R_i, i \in \{1, ..., n\}$ and the updated points of the shape. The algorithm takes into account the modeling constraints brought by the points previously fixed by the user (control handle(s)).

After a several iterations, the energy levels off, and a satisfying solution is reached.

2 Implementation

2.1 Algorithm

Github link : here.

We decided to re-use the `Jcg` library provided in the course and its `polyhedron3d` structure to manipulate the 3D-shape considered. The reason behind this choice lies in the easiness brought by this implementation when willing to move from a vertex to its neighbors. This is something that really came in handy in the algorithm we implemented. We also imported the external library `Jama` to facilitate matrices manipulation.

We agreed to use a unique class `Sorkine` that would comprise all the calculations - to each new deformation problem we associated an object of our class.

The implementation basically consists in three major stages. The first one calculates the rotations matrices (points fixed) that minimizes the energy presented above. The second computes the points (rotations fixed). The second step obviously abides by the modeling constraints of the control handles. Finally, the third one calculates the local rigidity energy and compares it to the previous computation's. The stop signal of the algorithm would therefore be either a maximum number of iterations or a threshold for the absolute value of the difference between two successive energy results ie when the energy stabilizes enough.

The first question we had to answer when implementing the paper's method concerned our class parameters. Obviously, the polyhedron had to be one of them, as well as its original version, which we decided to keep as an dictionary (easier to use and requires less memory space). We also defined the original lengths of the polyhedron's edges as a class parameter in order to facilitate the calculation of the RMS error (introduced in the paper) at the end of the execution. To avoid any unnecessary and costly repetitions, we decided to add the weights and the discrete Laplace-Beltrami operator (used for the points computation). This way, they only had to be computed once in the constructor.

The modeling constraints were as well an invariant of the problem and therefore, the set of fixed points became our sixth parameter - meaning it became useless to include them in each of our functions' input variables. Finally, in order to facilitate the use of the previous energy value and rotations at each iteration we also added them to the class' parameters.

A major encouragement for so many class parameters was that we decided to allow the user to visualize the problem resolution iteration by iteration if he wanted to. This forced us to find an easy way to keep in memory all the results of one iteration. To add that functionality, we separated the "one iteration `Sorkine` computation function" from the "whole problem resolution `Sorkine` function", which permitted to call the first one from the user's interface and separate each iteration if desired.

One difficulty we had to face involved the weights w_{ij}. We first tried to use the course formula and calculate the opposite triangles' areas but soon understood that something was odd when we compared the results to the ones obtained using the original formula with the cotangents. We realized that there was a sign error in the first formula that distorted the results.

We also took some time understanding and adapting our algorithm to the concept of boundaries. The shapes studied in TDs had none, therefore we did not expect this characteristic before testing our algorithm on the shapes provided by the paper, like in Fig. 1.

In order to better visualize the energy variations, we added a Python code to our algorithm. Using the external library `matplotlib.pyplot` this code returns the graphic representation of the energy function at each iteration. We used a text input file containing all the energy values to ease the transition from Java to Python (Java writes in the file, Python reads from the file).

An odd observation we made was that the energy curve was not only decreasing. The typical shape of the curve was a huge drop à the first iteration followed by a slight increase before leveling off. Although we believe this particularity does not result from an error in our implementation, we have not found the right explanation yet.

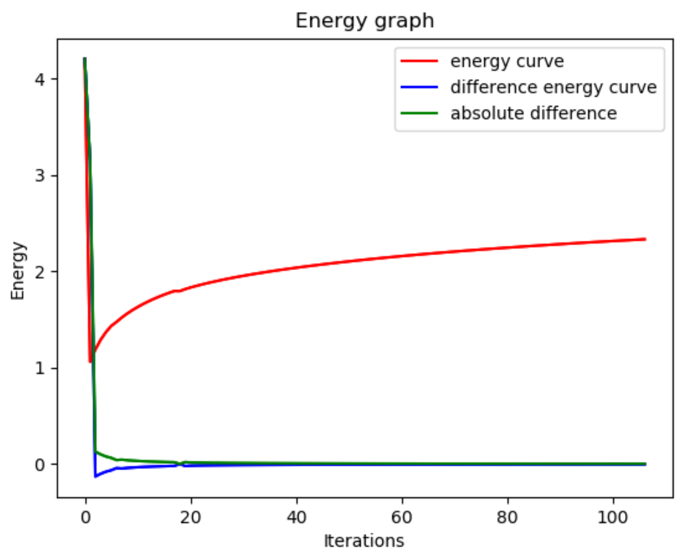

Figure 3: Local Rigidity Energy

2.2 User Interface

In order to best present our results, we wanted to provide the user with some intuitive commands. We thus started from the code of TD5 on mesh subdivision to adapt it to our problem. Our goal was to be able to provide you with a demo as Sorkine and Alexa on their video.

The first step was to provide the user with an easy way to select handle and static anchors. Adding member indicators to the `MeshViewer` class, we managed to come up with three distinct selection modes. Clicking on the screen launches a linear and exhaustive nearest neighbor search. Considering that the advances provided by the paper remained our priority and that we were working on small numbers of vertices, we didn't try to optimize this part, using KD-trees for instance. While working on the implementation of the paper, we notice we needed some more options (like zooming, resetting the selection, etc.).

To move the « handle » points, we resorted to use the arrows of the keyboard, rather than interacting the mouse, that already manages the `ArcBall`. The arrows allow movement in the XY-plane, while 'f' / 'b' keys (stands for front and back) manage translation on the Z axis. The problem is it allows translation but not rotation. We thought this issue was too peripheral.

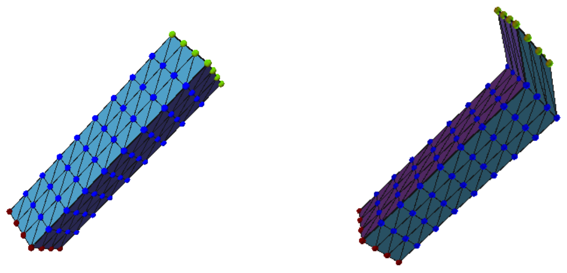
Figure 4: Example of vertices manipulation

3 Results

We obtained very satisfying results, as you can see on Fig. 5. We reproduced the best we could images shown in the original paper. We also tried to calculate indicators like error on edge lengths.

3.1 Running time and Complexity

As stated when explaining our implementation, the time complexity depends both on the number of vertices and number of edges.

Let n be the number of vertices and m the number of edges. The first step is initialization, where we store original points locations and the lengths of edges. We also computed weights in a constant number of operations per edges. Thus this step is in $O(n + m)$. However, we also need to compute the inverse of L. Jama doesn't detail their implementation of matrices inversion so we have to assume the complexity is $O(n^3)$[1]. Fortunately, we just have to run this once.

Let's consider one optimization iteration. The first step is to compute the rotation for each vertex. We need to iterate over neighbors hence the complexity will be in $O(\sum \deg(v_i)) = O(m)$. As the size of the matrix is always 3x3, the actual computation of the rotation matrix is in constant time. Then, we compute the new locations of points, we also need to loop over neighbors (so that's in $O(m)$) and then computes the matrix multiplication $L^{-1}b$ which is done in $O(n^2)$ time. Hence one iteration will take $O(n^2 + m)$ time.

Overall, the complexity of the algorithm is : $O(\text{num\_iter}(n^2 + m) + n^3)$.

Model	n	m	Running time per iteration
Armadillo	864	2586	$\sim$ 30ms
Cactus	620	1854	$\sim$ 23ms
Spiky plane	441	1240	$\sim$ 13ms

Table 1: Running time per iteration

[1]See this for time complexity of matrix operations

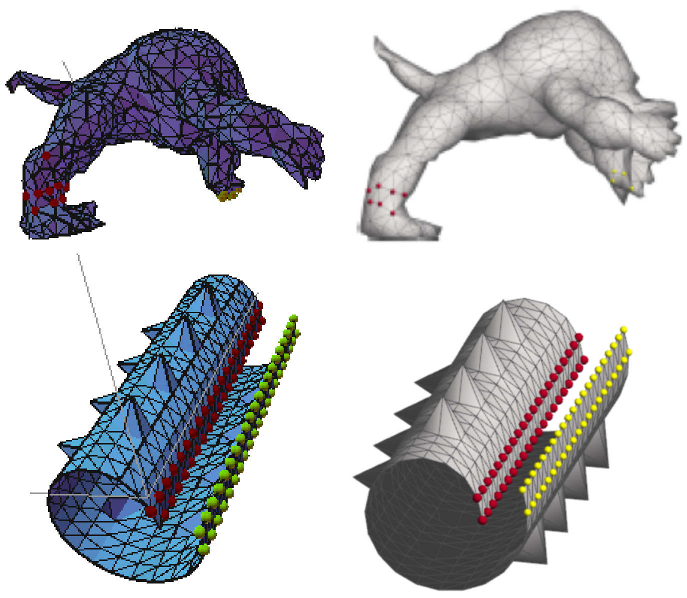
Figure 5: Example of results compared to results in the original paper

3.2 Relative Mean Square Error

As the Alexa and Sorkine's paper did in Table 1, we tried to compute the relative RMS error on edge lengths.

$$\text{Relative RMS error} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\frac{l_i - l'_i}{l_i}\right)^2}$$

Model	Figure	Relative RMS Error	Paper Relative RMS Error
Armadillo	Fig. 5	0.081	0.051
Spiky plane	Fig. 5 b.	0.014	0.034
Spiky plane	Fig. 1 b.	0.033	0.016
Cactus	Fig. 1 a.	0.052	NA

Table 2: Relative RMS error of edge lengths for various deformations. We compared it to the RMS error mentioned in the original paper (trying to replicate the deformations) and obtained similar results.

4 Conclusion

We managed to construct a satisfying deformation framework inspired from the paper's method. However, there are several axes of research that we did not have time to explore (yet). The first one concerns the initialization of the `p'` points. The paper proposes several different solutions on that topic : Previous frame (for interactive manipulation), Naive Laplacian editing, Rotation-propagation. We chose to do a user-interactive implementation of the algorithm and therefore focused on the Previous frame initialization. We also did not include the possibility to rotate handle points. We could however imagine to switch to a non-user-interactive algorithm, include rotation; and test the other possibilities mentioned by the paper.

Another point involves the "dino" OFF file obtained from the paper. We did not manage to test our algorithm on that shape because it was too heavy for Jama to manipulate. Since we are really fond of the dinosaur shape, we were thinking, for future development, about testing new matrix manipulation libraries, that handle sparse matrices, to solve that problem.

We strongly believe this research offers a smart and useful method to provide animation engineers with a convenient framework that accelerates shape deformation.

References

[SA07] Olga Sorkine and Marc Alexa. As-Rigid-As-Possible Surface Modeling. In Alexander Belyaev and Michael Garland, editors, *Geometry Processing*. The Eurographics Association, 2007.