INF555 - Digital Representation and Analysis of Shapes

Project report

As-Rigid-as-Possible Surface Modeling

Students:
Lucas Lugão Guimarães
Roger Leite Lucena

Professors:
Maks Ovsjanikov
Luca Castelli

Figure 1: X-shaped mesh deformed using the ARAP method.

1 Introduction

Surface deformation and editing, while preserving details and letting the deformations as intuitive and natural as possible, is an important challenge in the field of computer graphics. The approach to tackle this problem chosen by the paper (Sorkine O., Alexa M.: **As-Rigid-As-Possible Surface Modeling** - 2007) is to try to keep local transformations of the surface as rigid as possible, it means, local parts of the shape should change as rigidly as possible to satisfy the global asked deformation imposed by the user and under the required modelling constraints.

To measure how well this rigidity is kept the surface is divided into small covering overlapping cells and a non-linear, but simple, formulation of energy is introduced. The final deformation framework suggested by the paper is effective and well suited for practical applications.

2 Methods

The method proposed by Sorkine and Alexa consists in minimising a quadratic deformation energy in order to obtain the as-rigid-as-possible deformation of a given mesh subject to several constraints. The method proceeds in a alternate iterative fashion minimising the same energy using two different steps.

2.1 Deformation energy

The deformation energy is obtained by considering a weighted sum of local deformation energies. These local energies are calculated for each cell $\mathcal{C}_i$ of the triangle mesh $\mathcal{S}$ composed of the vertex i and its neighbours $\mathcal{N}(i)$. The local deformation energy $E(\mathcal{C}_i, \mathcal{C}'_i)$ is then deduced by first looking at the transformation $\mathcal{C}_i \to \mathcal{C}'_i$ as a rigid transformation where exists a rotation matrix $\mathbf{R}_i$ such that:

$$\mathbf{p'}_i - \mathbf{p'}_j = \mathbf{R_i}(\mathbf{p_i} - \mathbf{p_j}), \forall j \in \mathcal{N}(i). \tag{1}$$

Then, for the non-rigid case the local energy is defined in way to approximate the above equations in a weighted least square sense:

$$E(\mathcal{C}_i, \mathcal{C}'_i) = \sum_{j \in \mathcal{N}(i)} w_{ij} \|(\mathbf{p_i}' - \mathbf{p_j}') - \mathbf{R_i}(\mathbf{p_i} - \mathbf{p_j})\|^2, \tag{2}$$

Finally, if each cell contributes with a weight w_i to the overall energy the deformation energy $E(\mathcal{S}, \mathcal{S}')$ can be written as:

$$\begin{aligned} E(\mathcal{S}, \mathcal{S}') &= \sum_i w_i \, E(\mathcal{C}_i, \mathcal{C}'_i) && (3) \\ &= \sum_i w_i \sum_{j \in \mathcal{N}(i)} w_{ij} \|(\mathbf{p_i}' - \mathbf{p_j}') - \mathbf{R_i}(\mathbf{p_i} - \mathbf{p_j})\|^2, && (4) \end{aligned}$$

2.2 Weights

The choice of the weights w_{ij} and w_i is important in order to make the energy as mesh independent as possible. In this way, as explained in the paper, the cotangent weight formula is used for the triangular mesh. This weight is then defined as:

$$w_{ij} = \frac{1}{2} (\cot \alpha_{ij} + \cot \beta_{ij}) \tag{5}$$

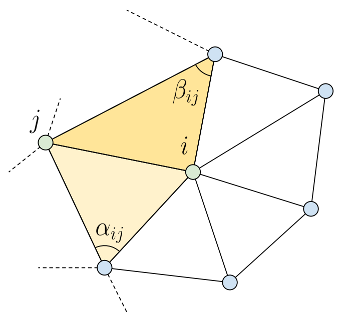

Figure 2: Cotangent weight diagram.

In addition, the weights of each cell can be set to 1 since the cotangent weights already take the area of the cell into account.

2.3 Rotations step

The energy defined in the equation (3) can be interpreted as a function of both the rotations $\mathbf{R}_i$ and the deformed positions $\mathbf{p}'_i$. In this case, by fixing the deformed positions $\mathbf{p}'_i$ it is possible to minimise for the rotations $\mathbf{R}_i$ in a per-cell basis.

First, let us denote the edge $\mathbf{e}_{ij} = \mathbf{p}_i - \mathbf{p}_j$ then we can rewrite the local energy (2) as

$$E(\mathcal{C}_i, \mathcal{C}'_i) = \sum_{j \in \mathcal{N}(i)} w_{ij} \left(\mathbf{e}'_{ij} - \mathbf{R}_i \mathbf{e}_{ij}\right)^T \left(\mathbf{e}'_{ij} - \mathbf{R}_i \mathbf{e}_{ij}\right) \tag{6}$$

$$= \sum_{j \in \mathcal{N}(i)} w_{ij} \left(\mathbf{e}'^T_{ij} \mathbf{e}'_{ij} - 2\mathbf{e}'^T_{ij} \mathbf{R}_i \mathbf{e}_{ij} + \mathbf{e}^T_{ij} \mathbf{R}^T_i \mathbf{R}_i \mathbf{e}_{ij}\right) \tag{7}$$

$$= \sum_{j \in \mathcal{N}(i)} w_{ij} \left(\mathbf{e}'^T_{ij} \mathbf{e}'_{ij} - 2\mathbf{e}'^T_{ij} \mathbf{R}_i \mathbf{e}_{ij} + \mathbf{e}^T_{ij} \mathbf{e}_{ij}\right) \tag{8}$$

In order to minimise the above expression with respect to the rotation $\mathbf{R}_i$ it is possible to drop the terms not containing it:

$$\underset{\mathbf{R}_i}{\text{argmin}} \sum_{j \in \mathcal{N}(i)} -2w_{ij} \mathbf{e}'^T_{ij} \mathbf{R}_i \mathbf{e}_{ij} = \underset{\mathbf{R}_i}{\text{argmax}} \sum_{j \in \mathcal{N}(i)} 2w_{ij} \mathbf{e}'^T_{ij} \mathbf{R}_i \mathbf{e}_{ij} \tag{9}$$

$$= \underset{\mathbf{R}_i}{\text{argmax}}\ Tr\left(\sum_{j \in \mathcal{N}(i)} w_{ij} \mathbf{R}_i \mathbf{e}_{ij} \mathbf{e}'^T_{ij}\right) \tag{10}$$

$$= \underset{\mathbf{R}_i}{\text{argmax}}\ Tr\left(\mathbf{R}_i \sum_{j \in \mathcal{N}(i)} w_{ij} \mathbf{e}_{ij} \mathbf{e}'^T_{ij}\right). \tag{11}$$

In order to obtain the rotation matrix maximising the above expression let us denote $\mathbf{S}_i$ the covariance matrix

$$\mathbf{S}_i = \sum_{j \in \mathcal{N}(i)} w_{ij} \mathbf{e}_{ij} \mathbf{e}'^T_{ij} = \mathbf{P}_i \mathbf{D}_i \mathbf{P}'^T_i \tag{12}$$

where $\mathbf{D}_i$ is a diagonal matrix containing the weights w_{ij}, $\mathbf{P}_i$ is the $3 \times |\mathcal{N}(v_i)|$ matrix containing the edges $\mathbf{e}_{ij}$ as columns and similary for $\mathbf{P}'_i$. As explained in the paper it is known that the rotation matrix maximising the trace $Tr(\mathbf{R_i S_i})$ is attained when $\mathbf{R_i S_i}$ is symmetric positive semi-definite. Thus, using the singular value decomposition of $\mathbf{S}_i = \mathbf{U}_i \mathbf{\Sigma}_i \mathbf{V}^T_i$ the rotation R_i is given by:

$$\mathbf{R}_i = \mathbf{V}_i \mathbf{U}^T_i \tag{13}$$

It is also important to rectify the orientation of the rotation matrix by changing the sign of the smallest eigenvalue column in the case where $\det(\mathbf{R}_i) < 0$.

2.4 Positions step

For the "positions"-step we assume that the rotations matrices $\mathbf{R_i}$ are given or were already calculated by the previous step. In summary, then, the goal of this step is to compute optimal vertex positions $\mathbf{p_i}'$ minimising $E(S')$ taken the $\mathbf{R}_i$'s as given.

Calculating the derivatives of $E(S')$ with respect to the positions $\mathbf{p_i'}$ and using the fact that $w_{ij} = w_{ji}$ we get:

$$\frac{\partial E(S')}{\partial \mathbf{p_i'}} = \sum_{j \in \mathcal{N}(i)} 4w_{ij} \left((\mathbf{p_i'} - \mathbf{p_j'}) - \frac{1}{2}(\mathbf{R}_i + \mathbf{R}_j)(\mathbf{p_i} - \mathbf{p_j}) \right) \tag{14}$$

Setting these partial derivatives with respect to each $\mathbf{p_i'}$ to zero to reach the minimum of energy we get the following sparse linear system of equations:

$$\sum_{j \in \mathcal{N}(i)} w_{ij}(\mathbf{p_i'} - \mathbf{p_j'}) = \sum_{j \in \mathcal{N}(i)} \frac{w_{ij}}{2}(\mathbf{R}_i + \mathbf{R}_j)(\mathbf{p_i} - \mathbf{p_j}) \tag{15}$$

That can be compactly written as:

$$\mathbf{L}\mathbf{p}' = \mathbf{b} \tag{16}$$

Where $\mathbf{L}$ is the known Laplace-Beltrami operator obtained from the linear combination of the left-hand side of the equation (15), and $\mathbf{b}$ is the n-vector whose ith row is the right-hand side of the same equation.

In order to take into account the modelling constraints, we can directly substitute above the deformed positions $\mathbf{p}_j^{'}$ that were constrained:

$$\mathbf{p_j'} = \mathbf{c_k}, k \in \mathcal{F}, \tag{17}$$

where $\mathcal{F}$ is the set containing the indices of the constrained vertices. To incorporate them to the equation (16) it is sufficient just to substitute the corresponding variables, erasing the corresponding rows and columns from $\mathbf{L}$ (keeping its symmetric positive definite property - which will allow us to use a Cholesky sparse solver later) and updating the right-hand side accordingly. To detail this approach, as it is not very well explained on the paper, we made the following changes in the matrices $\mathbf{L}$ and $\mathbf{b}$.

For the matrix $\mathbf{L}$, when taking the row corresponding to the $\mathbf{p_j'}$ fixed, we replace this row by the identity row, changing the correspondent row of $\mathbf{b}$ to the value c_k fixed. But if we let it like this the matrix $\mathbf{L}$ would lose its symmetric positive definite property and then we could not use the Cholesky sparse solver in the following steps. The trick then was to change the entire j-th column of $\mathbf{L}$ for the fixed $\mathbf{p_j'}$ points, the non-zero values were all set to zero and then the correspondent row of $\mathbf{b}$ was updated accordingly (basically summing to it the $w_{ij}\mathbf{p_j'}$ factor that disappeared from the left-hand side of the equation (15) when changing the entire j-th column of $\mathbf{L}$ as described before, with $\mathbf{p_j'}$ set to $\mathbf{c_k}$). This way the SPD property was successfully preserved, as necessary for the next steps.

Using the same Cholesky factorisation for $\mathbf{L}$ we can solve for $\mathbf{p}'$ by doing three back substitutions, one for each column of $\mathbf{p}'$ - this is done and abstracted by the sparse solver. With these values for $\mathbf{p}'$ we can then re-initiate the loop with the positions as given, solving for the rotations again in a way to keep reducing the energy, coming back to the rotation step explained above.

3 Implementation

The method presented in the previous section was implemented from scratch as an interactive web application based on WebSockets. It uses Python/Flask for its back-end, where all the calculations are done, and a JavaScript/ThreeJS front-end, used to show and manipulate the model in any modern browser.

Even though a full real-time implementation is indeed possible using the presented method the current implementation focus on near real-time performance, where each deformation is individually calculated and displayed in the front-end in a matter of deciseconds. The following diagram shows the main actions and the data flow between the front and back-ends.

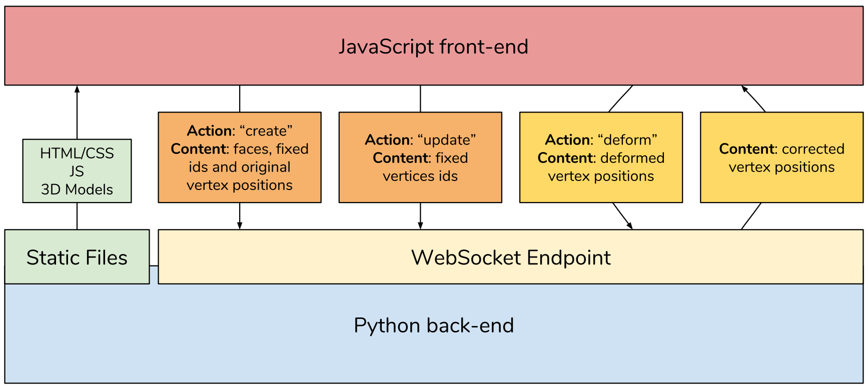
JavaScript front-end
HTML/CSS JS 3D Models
Action: "create" Content: faces, fixed ids and original vertex positions
Action: "update" Content: fixed vertices ids
Action: "deform" Content: deformed vertex positions
Content: corrected vertex positions
Static Files
WebSocket Endpoint
Python back-end

Figure 3: Implementation diagram.

As shown in the diagram, the Python back-end is responsible for serving the static files (scripts, pages, and css) and most importantly for calculating the deformation of the model. This is done by establishing a WebSocket connection between the front and back-end that remains active during all the user session. In addition, all the data transmitted via WS is encoded as a JSON payload containing only the needed information for each step.

3.1 Creation of the instance

As described in the methods section there are mainly two steps in the deformation calculation, the first one is the "rotations"-step and the second one the "positions"-step. The "rotations"-step depends only on the previous and new positions (**p** and **p'**) and thus it is always dependent on the previous iteration of the algorithm needing to be completely recalculated every time.

The second step on the other hand is done by solving the linear system of the equation (16) and, as explained, this can be done by precalculating the Cholesky factorisation of the matrix **L** only once in the moment of the creation of the instance or when the constraints change ("create" and "update" actions).

The Cholesky factorisation is done using a Python library called "`scikit-sparse`" a wrapper for the `CHOLMOD` library. This solver was used because of its known reliability and because of its good integration with other scikit libraries. This step is implemented in the main file of the back-end within the method named "`prefactorLMatrix`" where a compressed sparse column matrix (`scipy.sparse.csc_matrix`) is created and then factorised using the "`sksparse.cholmod.cholesky`" method.

3.2 Deformation Loop

To achieve a near real-time performance, the present implementation calculates the deformation every time the user requests. This is done by sending the positions of the vertices displayed in the front-end to the back-end so it can execute both the "rotations" and "positions"-steps and resend the results using the same pipe.

The "rotations"-step is then calculated by creating the $\mathbf{P_i}$, $\mathbf{D_i}$ and $\mathbf{P'_i}$ matrices described on the equation (12) and by finally decomposing the $\mathbf{S_i}$ matrix in order to obtain $\mathbf{U_i}$ and $\mathbf{V_i}$. With these two matrices the $\mathbf{R_i}$ rotation matrix is calculated as shown in the equation (13) taking into account the possible sign flipping if the determinant happens to be negative. The SVD decomposition was done by using the "`numpy.linalg.svd`" method and the process is carried in an iterative fashion for each vertex.

3.3 Front-end details

The front-end part of the implementation is focused to deliver in an easy to use sandbox experience and for this it relies heavily on the ThreeJS library. This library is responsible for loading the model, rendering it, and for giving the ability to deform the vertices. As a first approach the current version of the front-end is only capable of moving individual points at a time.

The use of the ThreeJS library made it easy to access all the WebGL capabilities of the modern browsers in a seamless way, letting us focus on usability and clearness. In addition, the WebSocket connection is established using the native class `WebSocket` described in the `HTML Living Standard`. The main methods of the front-end include the loading of the model, the creation of the handles and the input/output methods such as `sendObject`, `updateObject`, and `requestDeformation`.

The scene and the lighting setup was heavily inspired on two of the ThreeJS examples available in their online catalogue such as the spline editor and the hemispherical light example.

4 Results and comments

By the end of the implementation the framework was working smoothly for applying the global changes asked by the user (in the form of fixed positions of some points). The code was submitted to Github here and then a Heroku application was generated. To start the application using the cube model the link is available here, for the cactus model the link is this one. The instructions for the user are specified at the bottom-left region of the screen.

As-rigid-as-possible deformations were obtained after following the instructions on the paper. The method involved is a pretty robust one, converging to a final state of lower energy. It gives birth to final very intuitive deformation effects as it is possible to verify in the images below and the 1-min video resuming the main results we obtained, available here.

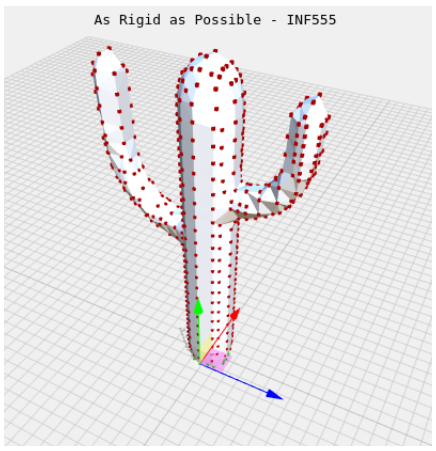

Figure 4: Original cactus model

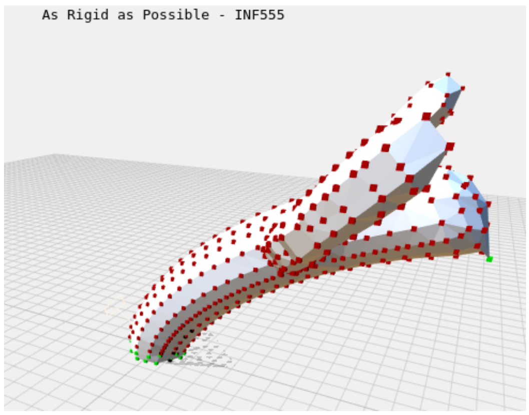

Figure 5: Cactus deformed - 16 iterations

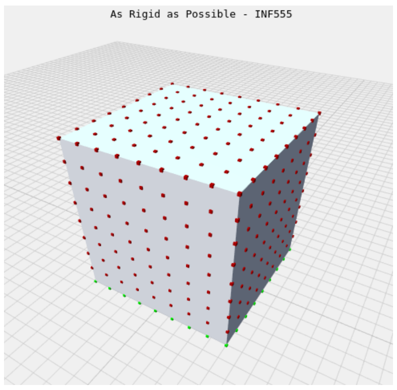

Figure 6: Original cube model

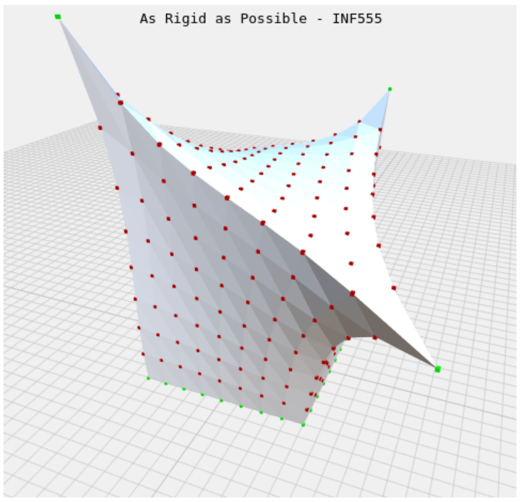

Figure 7: Cube deformed - 4 iterations

When implementing the inclusion of the constraints of the equation (17) into the equation (16), we first tried to redefine the corresponding row of $\mathbf{L}$ as to give a trivial solution for that $\mathbf{p}'_\mathbf{j}$, and to change the corresponding row of $\mathbf{b}$ accordingly with the fixed value $\mathbf{c_k}$, but not changing the other rows of $\mathbf{L}$, letting the solver do its job for the other variables. Unfortunately, this approach did not work out as expected. We first thought that the solver was not conceived to deal with systems having trivial solutions for some variables like this.

However, after some tests and also after discussing with the professors about the problem, it became clear that the real reason for the problem was that, when changing the matrix $\mathbf{L}$ as explained in the previous paragraph the matrix was losing its symmetric positive definite property, and so it was expected that the Cholesky sparse solver would not work for it. It was then necessary to make different changes to the matrices $\mathbf{L}$ and $\mathbf{b}$ in order to take the constraints correctly into account.

After, proceeding with the second approach explained in the methods section, being careful to keep the SPD property as it was suggested but not very well detailed on the paper, the solver worked with no problems.

Also, after our first implementation we observed that, for the cactus model for example, when setting a high number of iterations for the algorithm, it seemed that it was switching between two final convergence states of lower energy, instead of just one. To solve this problem we had two look carefully at the details of our implementation, looking for small errors, since the general idea and overall result of the algorithm seemed right.

After discussing with the professors we decided to analyse the energy value for every iteration and also for each step inside every iteration, in order to verify if it was really decreasing. Then, we realised that only the overall energy seemed to be the decreasing. In fact, it was always decreasing for the "positions"-step, but the "rotations"-step was sometimes increasing the value of the energy. In general, the global sum was decreasing as the contribution of the "positions"-step was usually greater than the other one. But, with a high number of iterations we had a final situation of refined loops around the convergence state, where the problem with the "rotations"-step became visible as its contributions started to make the overall energy of the iteration increase noticeably - effect translated as a switching cactus between two different positions.

At this point, after refining the scope of the problem, we looked carefully at the code of the "rotations"-step and concluded that there was a small error on the way we were implementing the covariance matrix $\mathbf{S_i}$ of the equation (12) shown previously. Now, after fixing this small problem the code works smoothly as expected.

5 Conclusion and next steps

The experience of implementing a real research paper and having the opportunity to look at the practical results of all the theoretical structure, constructed through the equations and ideas of the article, is a pretty interesting one. It is great to understand how the authors thought to tackle the problem, the strategy and the original ideas they developed, step by step, and how it was possible to go the practical side coding all the mathematics they proved that works, robustly and efficiently. Looking at the results that were born from the equations explained on the paper was a really motivating experience.

For the future of the project there are possibilities and approaches to increase the speed and reach real-time dynamic interaction with the application, as the authors of the paper did. Some ideas that could help increasing the speed would be using C++ as the back-end, for example, instead of Python.

Also, switching to a monolithic C++ program with an integrated OpenGL visualisation instead of using a web front-end could dramatically improve the speed of the program by compromising portability and easiness of use of the web-based approach. In this new approach it would also be possible to parallelize the "rotations"-step since each rotation matrix is calculated independently. Finally, in this new approach the memory used to represent the mesh could be shared as opposed to the current implementation, in which there is no memory sharing between the back-end and front-end.

To sum up, it was good to study a little what was being developed as the high-new results on surface modelling, deformation and edition, by 2007. It has just made us more curious about the results and state-of-the-art approaches that are being discovered and discussed today, in this dynamic field of Computer Graphics, 11 years later.

6 References

Sorkine O., Alexa M.: **As-Rigid-As-Possible Surface Modeling** (2007). *The Eurographics Association.*

Three.js library. Retrieved on January 26th from https://threejs.org/

HTML Living Standard - **Web Sockets**. Retrieved on January 26th from https://html.spec.whatwg.org/multipage/web-sockets.html#the-websocket-interface