Isotropic Surface Remeshing

I- Introduction

The objective of this project is to implement an Isotropic Surface Remeshing based on the ideas presented in the paper proposed by Mr. Alliez, Colin de Verdière, Devillers, and Isenburg. This remeshing is used to better fit a user specifications. For example when more information is needed on high-curvature parts of a mesh, or on the contrary when it's need to be uniform. The task is divided into eight sub-steps, and I have managed to implment only 6th of them.

This topic has already been extensively documented and studied, with advanced tips for solving various technical difficulties that may arise. The first difficulty is to parameterize the mesh into a 2D domain. Some meshes (with arbitrary genus surfaces or holes) need to be pre-processed, as done in a Local Parameterization Approach [1], this simple pre-processe is already a fully-fledged research field.

There are already some implementations available on GitHub that implement the eight steps that I mention before of Isotropic Remeshing, as described in the document [2] similar to the one shown in class.

II- Isotropic Surface Remeshing in 8 steps, current state of research

1) Cutting

To perform next steps, we need to parameterize the mesh in a 2D domain; here in particular, it should be parameterized in a disc-like domain. We therefore cut the surface to obtain something homoeomorphic to a disc.

Some recent work in Computational Geometry expresses interest in the problem of cutting a surface along a set of curves to obtain a topological disk. The latter is called a polygonal schema. Researchers have proved that this problem is NP-hard, yet they describe a greedy algorithm that outputs an $O(log^2 g)$ - approximation of the minimum schema [3].

2) Conformal parameterization

The conformal parameterization is needed to perform the steps that follow in an angle-preserving and locally isotropic parameterized space, which is essential for our remeshing, as it needs to be isotropic. Some algorithms already exist and are implemented in widely used libraries like libigl [4].

3) Spanning tree over adjacent triangle graph and feature edges

In some cases, mesh has some specific form and angle which we want to preserve. Like a mechanical part where we wouldn't want to flatten the sharp edges. A simple way to do so is to calculate the angle between to adjacent triangles and compare it to a threshold. These sharp edges, similarly as the boundaries of the mesh will be stored as feature edges. The spanning will be performed over triangle firstly and then over feature. It's the preprocess of the new idea that the paper describes to perform this isotropic remeshing.

4) Error propagation

A density function is provided by the user over the triangles of the mesh and over the feature edges. But for the uniform sampling, the density and sampling rate can be computed automatically as it is explain in the paper. The idea is to place a budget of V number of vertices on the parametrized domain. Each triangle 'i' will have a local budge of $R_s \times d_s(t_i) \times air_{t_i}$ and each feature edges j $R_f \times d_f(e_j) \times lenght_{e_j}$, which are float numbers. Rs and Rf are calculated to have (C represent the number of corner vertices, which doesn't move at all):

$$R_s \times \int_{surface} d_s(u,v)dudv + R_f \times \int_{features} d_f(u)du + C = V \qquad (1)$$

And to have a proper adjustment because Rs and Rf haven't the same dimension and doesn't represent the same thing we should have $R_s = 2\frac{R_f{}^2}{\sqrt{3}}$. Combined with (1) we have a simple second-degree equation. The float budge of triangle and edges will be rounded to the nearest integer and the rest will be propagated following the spanning tree that we have created before. The error propagation technique is simple and efficient and has received much attention [5], and it is especially used in image halftoning.

5) Place new vertices and compute the Delaunay triangulation

Now that we have a budge of vertices per triangle and per feature edges, we sample randomly these vertices. With this new set of vertices, we perform a Delaunay triangulation. Again, this is implemented in CGAL library which is largely used by the community.

6) Voronoi tessellation

From this Delaunay triangulation, we take the dual to create the Voronoi tessellation. Implemented in CGAL.

7) Lloyd relaxation

This algorithm moves the centroid of the Voronoi regions to its centre of masse, computed thanks to the density function over triangles. Do this until the centroid doesn't move anymore.

8) pull back up the parameterized new mesh.

At the end you keep the last position of the centroid, we again calculate the Delaunay triangulation and that new parameterized mesh will be push back up to give the final result.

III- Detail about algorithm

1) Error propagation for faces

We suppose that we have the spanning tree which will be used as a path to propagate the error. The algorithm proceeds by beginning from the deepest leaf's which will be the first current face and perform the following steps:

1. read the total amount of density on f ($d_s(t_i) \times air_{t_i}$);

2. deduce the number of samples to distribute on the current face ($R_s \times d_s(t_i) \times air_{t_i}$). This number is rounded to the nearest integer value, such a rounding generates a signed error e.

3. the error e is diffused to the parent face in the spanning tree.

4. flag f as processed;

5. pick the deepest leaf not processed in the sub tree of root the parent of the current face, and apply the algorithm again.

2) Lloyd relaxation

We suppose now that we have the vertices of the new mesh lying on the parameterize space. The Lloyed algorithm consist on the following steps:

1. build the Voronoi tessellation corresponding to the V vertices (sites of the Voronoi cells);

2. compute the centroids of the V Voronoi regions with respect to the density function expressed in parameter space, and move the V sites onto their respective centroid;

3. repeat steps 1 and 2 until convergence criteria is achieved.

There are three difficulties in this algorithm. First one is that if a cell is near a feature edge and cross it we need to cut it to repulse the sites from this feature edge. Second one is to compute the centroid of each Voronoi regions, we need to compute the intersection between a Voronoi cell and the original mesh and computed a weighted contribution of the polygons coming from this intersection. Third is for vertices lying on feature edges, they need to move along the edge and not directly to the centroid of the Voronoi cell.

IV- Implementation

This part is the one that took the longest. I will first talk about the libraries I used, for which purpose and what difficulties I've faced using these libraries. Then I will talk about the function that I've implemented and what are their objective, in parallel of what did I added and what supposition I made to code these functions.

1) External libraries

How to install and link libraries is explain in the README.md file of the project. Same thing for issues that I faced when installing and linking the libigl and CGAL library.

- Libigl

This library is used to implement the conformal parameterization (step 2) and to show the result. I used the example project as a template for a correct linkage of the library and I used the tutorial 501_HarmonicParam [6] to perform the parameterization.

- CGAL

CGAL was a bit harder to use, I needed to install vcpkg and how the linkage of the library was done (manuel for cgal [7]), was really strange in the example project. They used a function calls create_single_source_cgal_program in the CMakeLists.txt file which wasn't wall documented.

I use this library to compute the Delaunay triangulation and to compute the Voronoi tessellation.

- Delaunator

I first struggling using CGAL, so i initialy used Delaunator to compute the Delaunay triangulation. This library was just copied in the /dep folder as we done it for glm and over libraries during the tp.

2) Function implemented in the project

The first assumption is that the mesh used is already homeomorph to a disc like domain. It is actually the case for the camel head and for the David head meshes.

The first function used is *createFeature*() which identify feature edges and corners. Feature edges come from the boundaries of the parameterized mesh, but also when the angle between two neighbours' triangles exceeds a 50 degrees threshold. As for the corners, they are Vertices at the end of a sequence of feature edges, vertex belonging to three feature edges, or when the angle between two consecutive feature edges also exceeds 50 degrees.

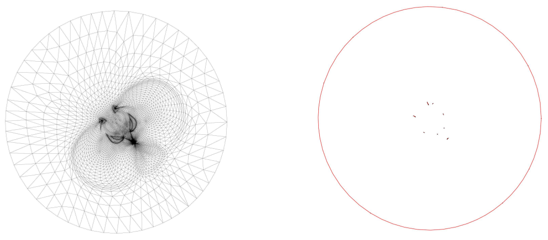
Figure 1: *to the left the original parameterized mesh, to the right the borders and feature edges*

Then there is the igl::harmonic function which compute the original parameterized mesh. Libigl works with Eigen matrices and vertices which constrains us to work with these objects for the rest of the project.

We perform a spanning tree of the parameterized mesh whith *arbreCouvrant*(). The root face and root feature edge are chosen randomly and both trees are built using a Breadth-First Search (BFS) algorithm. This function compute a "spanning tree" over feature edges, which may not be connected to one another.

For the next step, I have assumed that the density is defined on faces rather than on vertices. It is also important to note that we are applying density on faces in the parameterized space because we need to account for the distortion introduced when mapping the original 3D mesh to the 2D parameterized mesh. The density that will be use is $d_{s\,uv}(f_i) = \frac{area(f_i)}{area_{uv}(f_i)} \times d_s(f_i)$. the function *uniformeDensity*() return $d_{uv}(f_i)$ for each faces in parameterized space for a uniform sampling. If feature edges where to be incorporated, we should have computed the weighted average of the density of the two adjacent triangles. To calculate the simple rates Rs and Rf used in (1) we would calculate the sum over all triangle of $area_{uv}(f_i)$ times $d_{s\,uv}(f_i)$ then the sum of $lenght_{uv}(e_j)$ times $d_{f\,uv}(e_j)$ and then resolve the second degree equation (1).

With the error propagation algorithm, we manage to place new vertices according to the density and spanning tree calculated before. This task is harder for feature edges as long as the spanning tree for feature edges is more difficult to construct. Here we perform a "local" budge of vertices for each triangle by calculating $R_f \times area_{uv}(f_i) \times d_{s\,uv}(f_i)$. It is a float value that we take the nearest round int, and the rest that remain is propagated to the parent triangle and so on like it is described in error propagation algorithm.

Now that we have a integer number that correspond to the budget of each triangle we place Vertices randomly on the triangle with *randomBarycentric*(), same thing for vertices on feature edges and cornre vertices. This first sampling is not described in the paper, but since we will rearrange the sampling during the Lloyd relaxation, I took the assumption that sampling the vertices for the first time could be done randomly.

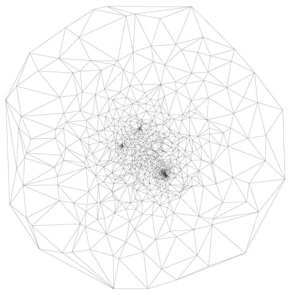

Figure 2 : *Delaunay triangulation*

Then from this initial sampling we perform a Delaunay triangulation (figure 2). That task took me a lot of time because I tried to implement it be my self but it was too slow, then I tried to do it with CGAL and initially I wasn't able to visualize the result so I wasn't sure that it was working, so I have done it with a third library which is called Delaunator.

Next step was from this Delaunay triangulation to perform a Voronoi tessellation (figure 3). I tried and didn't manage to do it by hand using the Delaunator result. I turn back to CGAL which finally worked, even if the iterator to extract the edges of the Voronoi cells wasn't working well, reding the hall manual of CGAL finally worked.

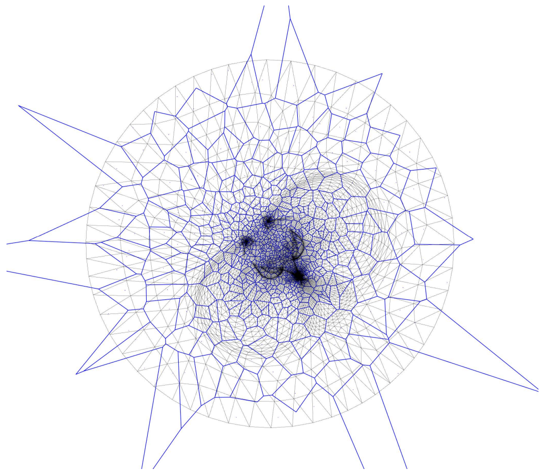

Figure 3 : *Voronoi Tessellation lying on the original parameterize mesh*

V- Improvement and further work

We it can be done to go further, perform the weighted centroid of the Voronoi cell, including the clipping of the cell when it comes to cross a feature edge, to finally apply the Lloyd algorithm.

To put back up the new parameterized mesh here again there is a giant work that's need to be done since i didn't find a external library that implement a transformation and the inverse transformation. We can for example take in consideration the local curvature of surface interpolate inside the triangle and rase the vertices when it's not on a vertices from the original mesh (like it is the case for corner vertices).

VI- Bibliographie

[1] Reserche on arbitrary genus surfaces https://www.researchgate.net/publication/29611859_Isotropic_Remeshing_of_Surfaces_a_Local_Parameterization_Approach

[2] A implementation on git witch the result are near from what we have been shone in course https://github.com/huxingyi/isotropicremesher/tree/main and https://graphics.stanford.edu/courses/cs468-12-spring/LectureSlides/13_Remeshing1.pdf

[3] ERICKSON, J., AND HAR-PELED, S. Optimally cutting a surface into a disk. In *Proceedings of the 18th Annual ACM Symposium on Computational Geometry* (2002), pp.244–253.

[4] DESBRUN, M., MEYER, M., AND ALLIEZ, P. Intrinsic parameterizations of surface meshes. In *Proceedings of Eurographics* (2002), pp.209–218.

[5] OSTROMOUKHOV, V. A Simple and Efficient Error-Diffusion Algorithm. In *Proceedings of SIGGRAPH* (2001), pp.567–572.

[6] https://github.com/libigl/libigl-example-project and https://github.com/libigl/libigl/tree/main/tutorial

[7] https://doc.cgal.org/6.0/Manual/windows.html