Variational Surface Remeshing Techniques

Marios Bikos
University College London
MSc Computer Graphics, Vision & Imaging
Email: mariosbikos@computer.org

***Abstract*—In digital geometry processing, remeshing is considered a fundamental part in order to improve the quality of a simplified mesh after the scanning of a 3D object. In this project, we were asked to write code in order to implement basic remeshing algorithms which will approximate the original model while improving the triangle properties.**

I. Introduction

3D models are used in a broad spectrum of applications ranging from scientific visualization to the entertainment industry. In order to represent an object in its digital form, a 3D acquisition process has to take place first. During this process, several inaccuracies are caused due to scanning devices and interactive software packages, that have as a result unsatisfactory 3D models. These models may consist of too many vertices that are actually redundant, so during the post-processing simplification process by software packages, the attempt to accurately approximate flat or curved regions leads to triangles with a really bad shape and long areas.

This is where the remeshing process emerges, in order to improve meshes with respect to their size and quality, while still keeping the mesh fidelity at a decent level, approximating the initial mesh features. For example, for easier mesh processing in modeling application and for numerical robustness in simulations, meshes that consist of nearly equilateral triangles are preferred. The final result of remeshing can have more quality related to the sampling, regularity, shape, or even size of triangles. A combination of these criteria is used according to the application in question. Taking everything into consideration, remeshing algorithms are therefore considered as a really important phase of the 3D mesh processing domain.

II. Aims

Over the years, several different techniques have been proposed for remeshing [1]. What we hope to achieve through this project is to implement surface remeshing algorithms that can be applied to different surface triangle meshes, producing remeshed surfaces with better quality, while also approximating the original mesh surface. In other words, "given a 3D mesh, we try to find a better discrete representation of the underlying surface" as remeshing process is usually defined.

In our case, for the basic part of this project, an isotropic surface remeshing approach was implemented, which is based on a global parameterization of the original mesh utilizing a weighted Centroidal Voronoi Diagram for the parameter space.

III. Tools & Libraries

For the remeshing algorithms proposed, we were advised to use the CGAL library [2] for specific parts of the implementation, since it contains state-of-the-art algorithms for geometry processing that would otherwise take too much time to implement by ourselves. For example, the CGAL library provides a plethora of Voronoi diagrams and Delaunay triangulations. It is worth mentioning that CGAL also provides ready algorithms to get a remeshed surface immediately, however, we didn't used any of these functions, since we needed to implemented that ourselves. Later on, at the description of each stage, it is described when a CGAL algorithm was used in order to derive a specific result. Apart from CGAL, MeshLab software was used for pre-processing simplification and graph cutting as described later on.

IV. Area-based Remeshing

For this technique we need as an input a 3D mesh surface and the output of our algorithm should produce a newly generated mesh, which will satisfy specific quality requirements, keeping the fidelity of the original mesh and the mesh complexity in specific levels. In order to get an isotropic remeshing as a result, our algorithm is based on Centroidal Voronoi Tessellation.

- Input: We downloaded the 3D models most often used in 3D geometry processing papers [3] and selected a few as input.
- Output: The simplified and remeshed 3D model.

The main stages of the algorithm are the following ones:

A. Preprocessing: Simplification

Although a simplification algorithm could have been implemented, we were advised to use MeshLab software instead, for the simplification of the 3D mesh to a simpler one with far fewer faces, so that our program could illustrate the remeshing process from the beginning to then end faster without waiting too long for the final outcome to be produced. In MeshLab this was easily done using the filter Quadratic Edge Collapse Detection. This filter allows us to get a simplified mesh after specifying either the percentage of the original mesh to be reduced, or the final number of faces wanted. A quality threshold of 0.3 was used for the simplification.

B. Load Model and compute Normals

By pressing the M key user is able to load the simplified 3D .off model produced from Meshlab, creating a CGAL polyhedron mesh in our program. In order to display the 3D mesh correctly throught the framework used in previous coursework, we wrote a function which converts the CGAL polyhedron mesh to the frameworks class mesh. However,

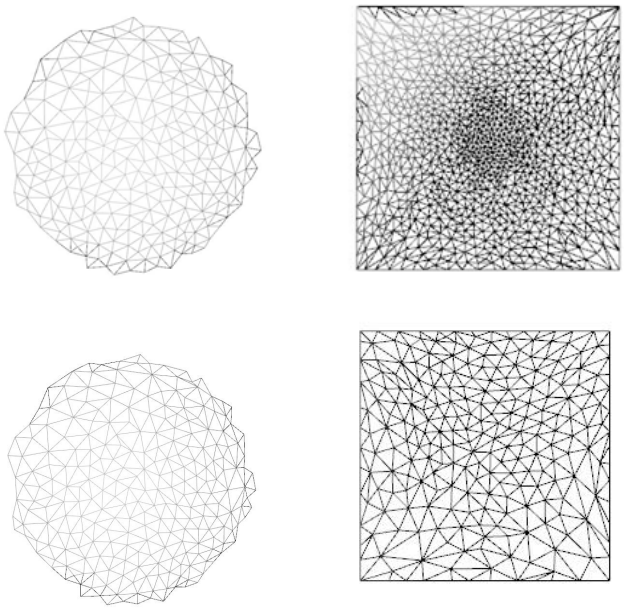

Fig. 1: The result of initial parameterization(top right) of our algorithm using a patch of a sphere as input model (top left) and the result triangulation of the parameterization after converged CVT(bottom right) together with the updated mesh(bottom left)

using a .off model as input, we don't have information about the normals. Therefore, we also coded a function which estimates the normals for the loaded mesh. This way we can accurately render the mesh which was created from CGALs polyhedron using the framework.

C. Parameterization

The next step is to estimate the global parameterization of the mesh, which can help us solve complex 3D domain problems, using a representation of the mesh in the 2D domain. With parameterization, we technically attach a geometric coordinate system to the 3D mesh. So, we have a 1-to-1 correspondence of the surface of 3D object with an image in 2D domain. Since the 3D mesh we use is a triangulated surface, the 2D parameterization created, will also have triangle pieces corresponding to triangles of the surface of 3D mesh. The parameterization mapping is done by representing the parameterization using a set of all (u,v) coordinates associated with each 3D vertex of the surface (x,y,z). The vertices of boundary are fixed on a convex polygon, while the coords of internal 2D vertices are found by solving 2 linear equations.

However, instead of doing this manually, we utilized CGAL in order to easily get the global planar parameterization of the input mesh. CGAL provides a range of surface parameterization methods, according to the distortion they minimize (angles vs. areas) and other relevant properties. In our case, we opted to use the discrete authalic parameterization method with square borders, which corresponds to a weak formulation of an area-preserving method, and locally minimizes the area distortion.

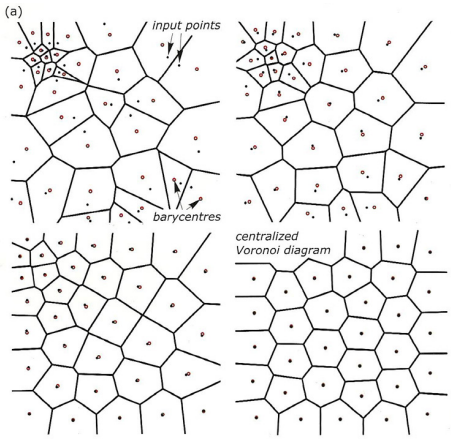

Fig. 2: An example of iterations until the final weighted centroidal voronoi tesselation [4]

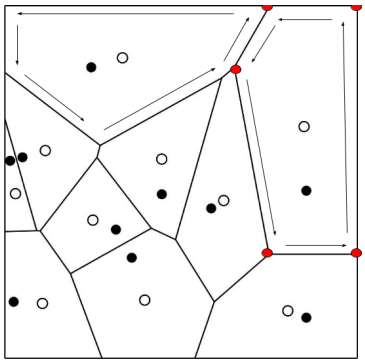

Fig. 3: An example of a bounded Voronoi diagram for the filled circles as points. displaying the centroids for each polygon cell as empty circles. Also the vertices the form a cell are colored red for the upper right polygon and the half-edges are displayed.

Nevertheless, CGAL provides surface parameterization methods which only deal with meshes which are topologically equivalent to discs. In order to deal with that, since our mesh can have an arbitrary topology and number of connected components, we must manually specify a cut graph, which is actually an oriented list of vertices, as the border of a topological disc. Therefore, using MeshLab, we need to cut our mesh first, before being able to get a parameterization correctly through CGAL. This can be done using MeshLab, simply by selecting faces and then deleting them. This way we can have a valid input for the parameterization used in CGAL.

D. Deduce the border points

After we have found the 2D points (u,v) of the parameterization, we need to know the border points of the parameterization (the ones that belong to the square border) and the inside points, since they are not explicitly specified by CGALs parameterization methods. In order to do so, we wrote code which compares the (u,v) coordinates of the parameterization points with the 2 points (0,0) and (1,1) that are the

standard points of the parameterization, since parameterization is specified within a rectangle formed by these 2 points every time and the u,v points are normalized to fit within it. This way, we can get the border points and the inside points separately for later use. Also for each 2D point (either inside or border points) we get the index that shows the vertex of the polyhedron that it corresponds to specifically.

E. Centroidal Voronoi Tessellation

From parameterization and after discriminating between border and inside points, we estimated the weighted Centroidal Voronoi Tessellation. Generally, a 2D planar Voronoi diagram or tessellation is defined for a set of points called sites, that lie in some space R^2. It actually partitions a 2D space into specific regions according to the distance between points. Each region consists of all points that have distance closer to the site of the region than by any other site. Similarly, centroidal voronoi tessellation (CVT) is a special Voronoi tessellation where the site of each cell coincides with the mass center of the Voronoi cell. So the 2 properties that characterize a CVT are the isotropy and equal-mass enclosing. The mass center is defined as:

$$c_i = \frac{\int_{V_i} xp(x)dx}{\int_{V_i} p(x)dx} \tag{1}$$

Therefore, given an mesh as input, as well as a number of site points, which correspond to the mesh vertices on the 2D parameterization, we can iteratively minimize a CVT-based energy function to distribute the sites in space, getting an isotropic triangle mesh when we define its dual triangulation. The stages of a sample weighted CVT can be seen in Figure 2.

From the parameterization, we actually got an initial triangulation of the vertices, defining a 2D tessellation of vertices. Using this triangulation, we must deal with the 3 vertices of each triangle, which makes things complicated. However, utilizing the duality between voronoi diagrams and triangulations, we can get the Voronoi diagram, which simplifies the procedure into dealing with a one sample for each triangle. In our algorithm, we could choose to create the CVT either using both border and inside points, or just by using the inside points. We considered that we should do the latter, so that we can have a standard border rectangle and easily move points towards centroid without any more malfunctions or moving points outside the require territory of the parameterization, since this would require excessive exceptions.

In order to get the Voronoi diagram for the inside points of the parameterization, we also had to limit the Voronoi diagram according to a bounding box which is actually the rectangle formed by points (0,0) and (1,1). Since CGAL doesnt really have a helpful and easy way to accurately determine the voronoi diagram based on a bounding box, we searched alternatives and found out that in most similar questions in CGAL forums, most users decided used a set of functions (https://github.com/sloriot/cgal-voronoi-cropping) that allow this specific process. This way we managed to determine the voronoi diagram of a set of points given a bounding box, and get as an output the voronoi diagram as a half-edge data structure.

Using this half-edge data structure we can then get the half-edges that define each cell and therefore the points that form each cell of the voronoi diagram. This was done by traversing the halfedges of the cell. Also since we have the bounding box into consideration, when a voronoi diagram ray is supposed to be long and outside the bounding box, it is instead cut according to the bounding box and so this works perfectly as planned to get all the neighbors correctly within the parameterization limits. Since we can easily get all the neighbors, we can then estimate the centroid 2D point of each cell.

To visualize this process for readers, Figure 3 displays the centroids as empty circles and the points of the voronoi diagram as filled circles. In this figure, for the cell on the right of the image, we can easily traverse the cell with the half-edges and get all the points that form the cell. We coded a workaround and changed the initial code a bit here, in order to actually get back the correct centroid for the corresponding initial inside point of the 2D parameterization, since the half-edge data structure created, gives us no information on the order of how traversal will take place. Next, we can traverse through each face of the created Voronoi diagram and get the points that form each cell. Based on these points we can estimate the centroid coordinates of each cell.

In order to get the centroid of the cell, we need the area of the polygon/cell and then we can get the coordinates of each centroid:

For the area of the polygon we have:

$$A = \frac{1}{2}\sum_{i=0}^{N}{}^{1}(x_i y_{i+1} - x_{i+1} y_i) \tag{2}$$

Now for the centroid coordinates of the polygon we have:

$$x_{centroid} = \frac{1}{6A}[\sum_{i=0}^{N}{}^{1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i] \tag{3}$$

$$y_{centroid} = \frac{1}{6A}[\sum_{i=0}^{N}{}^{1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i] \tag{4}$$

In order to create the final weighted CVT we implemented Lloyds relaxation method. Given the density function according to user's and application's needs and the an initial set of the 2D points derived from parameterization, this deterministic algorithms consists of the following 3 steps:

- Generate a Voronoi diagram according to the n input points
- Calculate the centroid for each Voronoi region created based on the density function used and then move the n points to the position of the centroids.
- Repeat the previous steps until convergence takes place.

In our algorithm, convergence takes place utilizing 2 threshold values. The first threshold used is a distance threshold, measuring if centroids moved more than a threshold value distance from their previous corresponding position. We then estimated the percentage of points that exceed this distance threshold out of all points into consideration. If this percentage is above a second threshold value percentage, e.g 5%, then we repeat the centroid estimation procedure, otherwise we end the process and give as output the centroid points for each input inside-point. Also an alternative approach implemented as well is for the user to interactively select how many iterations will take place. So he can press a key and the iteration will take place, then feedback will be provided for him to see the result and then according to whether he likes it or not, the process may and or continue.

F. Get Barycentric Coordinates for each centroid

Now that we have all the 2D centroids, we need to actually manage to find the 3D corresponding movement that the initial 3D points of the mesh have to make in order to get the new positions of the vertices. To do that, we need to estimate the barycentric coordinates of the centroids, based on the triangle of the parameterization they belong to. However, in order to find the corresponding triangle for each centroid point, we cant just find the 3 closest points of the 2d parameterization, since we dont know if they will actually form a correct triangle as specified after the parameterization. Instead, we decided that we had to use a greedy method (this of course makes our program slower, but its the only solution we could think of). So we:

- Traverse all the facets of the 3D polyhedron mesh
- Get the 2D parameterization points that correspond to the face and form a triangle in 2D parameterization domain
- Check if the 2D centroid point in question is indeed within this 2D triangle using a point-in-triangle test

Once we find, the 2D triangle of parameterization which has the centroid inside it, then we can find the barycentric coordinates for this centroid point, since we know the 3 points that form the triangle and the centroid coordinates.

G. Relocation of 3D vertices of mesh

Finally, we need estimate the new 3D vertices position for the mesh, for those points whose parameterization makes them belong in the inside points and not the border points. This way we can relocate them from their previous position, creating a remeshed surface. This was easily done, using the barycentric coordinates already calculated, multiplied individually by each 3D point of the face, where the centroid corresponds to. So we can get a new 3D position for each of these 3D vertices that represent centroids in parameterization and therefore change the current scenes mesh vertices to a new position, getting eventually the new 3D remeshed surface. The results for a small patch of a sphere can be seen in figure 1.

V. Alternative Remeshing Approach

We also implemented an alternative remeshing technique which is based on the area-based remeshing with some changes. In this case, after having estimated the weighted CVT and the centroids of the cells, as well as the new 3D positions of the vertices, we actually create a new mesh. More specifically, we connect the centroids creating a new form of triangulation, projecting it back in 3D space as a whole new mesh, instead of keeping the original triangulation connections which were based on the connectivity of the input mesh. We found our that this method gives a really good result and so evaluation is mainly based on this approach. The only difference with the previous technique is mentioned is the final stage where we create a whole new triangulation and a whole new mesh.

VI. Experimental Results & Evaluation

The remeshing algorithms we implemented can actually take as an input a simplified cut mesh and generate a new mesh with more isotropic triangles in order to efectively represent the initial mesh. The final mesh is produced after convergence of the weighted CVT. However, we noticed that the threshold values may affect the final outcome and the visual result of the first method implemented may not be obvious if the model has vertices with bad valence. Using the second method, the results are more obvious since with the new triangulation the visual results displayed indeed givesus a good feedback as expected after the weighted CVT convergence and the re-triangulation of the centroids for the creation of the new mesh.

For the evaluation of our algorithm, we decided to search past approaches for quantitative measures used in order to assess the remeshed results. As we found out, apart from the obvious visual results comparisons, traditional ways for measuring the quality of the remeshing algorithm involve deducing geometric properties of the triangles of the final mesh [5]. More specifically, usually authors measure the minimal angle of all the triangles of the final mesh, which is obviously anywhere between 0 and 60 . Moreover, in order to have mesh with good quality we need a minimum angles with value over 10 , while the average angle metric should be over 45 .

Therefore, we created extra evaluation functions that measure the irregularity percentage of vertices of the mesh, the minimum of all the angles of all triangles of the mesh, as well as the average angle of all the minimum angles of each triangle of the mesh. For the irregularity calculation, we used an approximation, since the border points are not known, so we considered that since most vertices do not belong to border, we can consider each vertex of the mesh as irregular if its valence is not 6.

For experimenting with our algorithms, we decided different methods so that user can either see the result after 1 iteration by pressing a key, or wait to get the final result after convergence of CVT. Also user can use either of the 2 methods developed for remeshing. For testing our algorithm, we used 2 different models as input. The first model is a beetle car model, while the second one is a mask-face model. Both models are used quite often in papers related to 3D geometry.

Apart from the quantitative metrics mentioned above, we also decided to run experiments using 3 different types of the

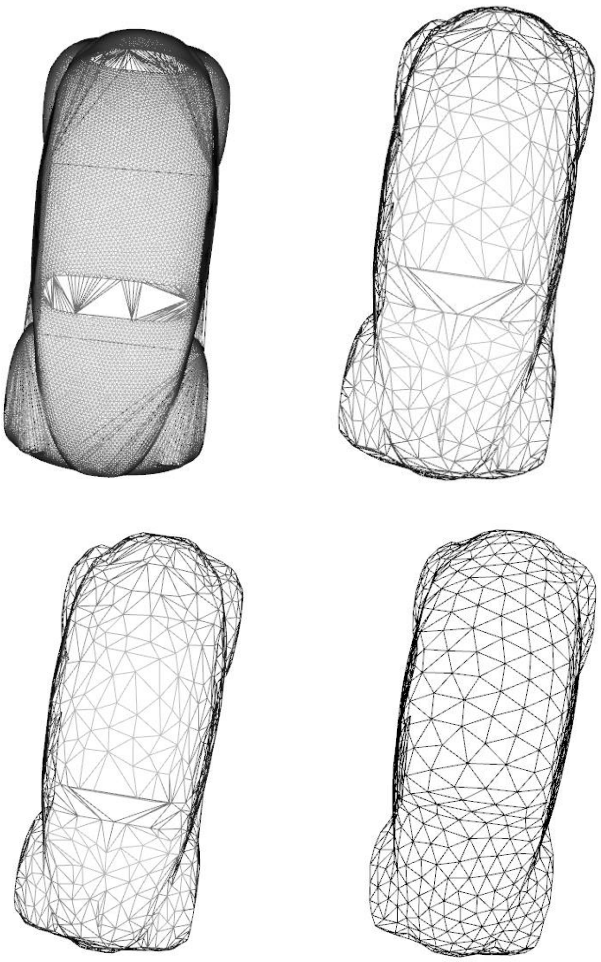

Fig. 4: Results of the Remeshing algorithms for the beetle car model:On the top left-corner we can see the initial model. On the top right, the model after simplification done via MeshLab. On the lower left corner, the result of remeshing using the area-based remeshing technique. Finally, on the lower left-corner the result of the alternative approach for remeshing.

each of the 2 models. So we decided to simplify each model to 2000,1000 and 500 vertices and find the iterations until convergence and the duration of the execution of our program. The thresholds used were set at 0.05 for the percentage threshold and 1e-06 for the distance threshold.

The results for the iterations and the duration of the algorithm can be seen in Table I. The results were tested on the computer of my partner in this project, therefore are the same for both our reports (since my laptop runs Windows on MacOS greatly affecting performance). As we can see, more iterations are needed until convergence for models with lower number of vertices. On the other hand, we found out that the execution time of course depends on the number of iterations, but again higher-vertices triangle may take less time to complete for the final result.

The statistics for irregularity and angles using the second method and the medium-poly models can be seen in Table

Model	Vertices	Faces	Iterations until conv.	Time (sec)
Beetle (Low)	547	1024	57	19.52
Beetle (Medium)	1025	1944	50	33.34
Beetle (High)	2042	3956	21	30.47
Mask (Low)	465	870	97	29.57
Mask (Medium)	937	1782	58	36.54
Mask (High)	1924	3736	14	19.91

TABLE I: The number of vertices and faces for each model used for evaluation, as well as the iteration times until convergence and the total duration of program execution.

Model (size)	Beetle Original (Medium)	Beetle Result (Medium)	Face Original (Medium)	Face Result (Medium)
Irregularity (%)	70.6	35.6	64.4	34.3
Min Angle (Deg)	0.1	10.4	1.1	8.2
Avg. Angle (Deg)	29.4	43.6	33.7	43.2

TABLE II: The quantitative measures used in order to compare the initial and final meshes to see if we have a better quality.

II. As we can see, indeed the irregularity is reduced by a percentage of 49% for the beetle model and 46% for the mask model. The result is indeed lower as expected but not as good as it should be generally, since about a third of the vertices are irregular. A lower value would be better to achieve in future approaches. Regarding the angles, we can see that the minimum angle is increased above 10 degrees for the beetle which is considered as a good result, however for the face model we reach 8.2 degrees which still shows that our algorithm still needs improvement. On the other hand, the average angle results are quite good since we have an increase of about 45%, so the angle values are generally increased in the whole mesh.

VII. Discussion-Extensions

Taking everything into consideration, we can conclude that the algorithms developed gave us an insight of the basic remeshing techniques, as well as the technical background on how to utilize CGAL library and how to process 2D parameterization in order to effectively change a 3D mesh surface. Of course, more advanced techniques could be utilized such as an adaptive remeshing, which take into account the curvature of the mesh. In this case, more curved regions of the model will contain smaller triangles in comparison with flat areas which would contain larger triangles. In order to do that, we would need to change the density function from the uniform one used to a new one which would be defined based on the mean and gaussian curvature of each vertex of the mesh. More specifically, just like in [5] we could define for each vertex a value:

$$(v) = \frac{1}{|K(v)| + \quad H^2(v)} \tag{5}$$

using different weights , to achieve different results according to the desired application, where H and K are the discrete Gaussian and Mean curvatures per vertex v. This method would allow us to achieve more efficient results and would be more robust and accurate. However due to limited time we weren't able to provide an implementation.