Digital 3D Geometry Processing - Project Final Deliverable

August 22, 2017

Raphaël Steinmann
Thomas Batschelet
Alain Milliet
Department of Computer Science, EPFL, Switzerland

I. Introduction

In this paper, we explain the process of creating an interesting lamp from a 3D mesh given at the beginning of the project for the course "Digital 3D Geometry Processing" given in Fall 2016 at EPFL.

II. Algorithms

In this section, we describe every algorithm we implemented to obtain our final result. The section is split in two parts to distinguish the algorithms used on Geralt's head from those used on the skull.

A. Geralt mesh

For the Geralt mesh we implemented 3 new algorithms to obtain the result we were looking for.

1) Separate Head (*MeshProcessing::separate_head_log* & *MeshProcessing::separate_head_melting*)
We want to separate the head by cutting it between the eyes, so we decide to find this x-coordinate using the nose of Geralt. The nose is easily found by looking at the vertex having the largest z-coordinate and when we have this point we just look at its x-coordinate to find the middle point we are looking for. We also compute the height where we want the cut to start by adding the threshold to the y-coordinate of the nose. From now we can go through the vertices of the mesh and test if it is higher than the height computed before and on which side of the nose it is. From there we implemented two different functions. In the log function, we update its x-coordinate by adding (respectively subtracting) $\gamma * log((vertex_x - height) + 1)$ to its old x-coordinate. In the melting function, we update not only the x-coordinate but also the y-coordinate the following way:
$x_{updated} = \gamma * ((y - height)/6)^2$
$y_{updated} = \lambda * ((y - height)/7)^2$
with $height$ being the height where we start the cutting and λ / γ being parameters we can tweak.

2) Create fracture (*MeshProcessing::create_fracture*)
When we just separate the head, boundaries of the fracture are not realistic and we wanted it to look more like a scar. That is what this algorithm is build for. It finds the vertices that are in and around the fracture and then compute a modulo value on the y- and z-coordinate of those vertices to modify their x-coordinate.

3) Delete big faces(*MeshProcessing::delete_long_edges_faces*)
The last algorithm we implemented for Geralt mesh is an algorithm that goes through the edges of the mesh and test their length. If this length is bigger than a certain size then we delete the faces it is part of.

B. Skull mesh

For the Skull mesh we implemented one main algorithm to apply onto our mesh the desired Voronoï pattern. The algorithm uses another two functions with the mean to ensure the printability of our mesh.

1) Create dual pattern
(*MeshProcessing::skull_dual_graph_pattern*)
Since Delaunay triangulation is complex and time consuming, we opted for a simplified Voronoï pattern implementation. The trick is to remesh the skull before computing the dual graph so that we have a regular triangulation. This allows us to bypass Delaunay triangulation while ensuring the quality of the Voronoï pattern. Note that we chose a height-based remeshing algorithm for aesthetic reasons but any other remeshing algorithm providing a more or less regular triangulation is also fine.

Concerning the dual graph computation, we tried several possible implementations before finding the one that worked for us. First we thought of using the *add_edge* method, but for some reason it was not implemented in Surface_mesh. We then tried to implemented it ourselves based on CGAL's implementation, but it turned out to be quite complicated as Surface_mesh and CGAL showed some compatibility issues.

Then, considering that we could not add an edge we came up with a dual graph implementation working with *add_face*. Unfortunately, the *add_face* method did not work as well as expected. We also noticed that the viewer was designed to display only triangle faces (After creating a quad, the viewer displayed a triangle but when we exported the mesh and opened it in Meshmixer it displayed a quad). Also the program crashed when we created faces with more than 4 vertices, which was problematic.

Finally, following one of the TA's advice, we went back to our original idea with the difference that instead of computing the dual mesh with vertices and edges, we did it directly with spheres and cylinders. The first advantage is that these primitive shapes can be constructed with only a few triangles. This solves the problem of displaying quad faces and computes in a reasonable time. The second asset of this method is that we do not have to remove the faces and thicken the edges afterwards, because the "hole pattern" is already being taken care of.

The following pseudo-code explains how the dual-graph pattern algorithm works:

```
fs_and_ps = empty vector of tuples (Face, Point)
dual_vertices = empty vector of Point
for Face f ∈ primal_faces do
    create Point p in the middle of f
    add p to dual_vertices
    add tuple (f,p) to fs_and_ps
end
for Edge e ∈ primal_edges do
    get f1 and f2 the two faces adjacent to e
    find p1 and p2 the middle points in f1 and f2
    create a cylinder between p1 and p2
end
remove the entire primal graph
for Point p ∈ dual_vertices do
    build a sphere centered on p
end
```

Algorithm 1: Dual Graph hole-pattern

2) build cylinder
(*MeshProcessing::build_cylinder*)
To ensure printability, we wrapped each edge with a rectangular shape cylinder to link each pair of vertices. The function takes three parameters, namely the two end points of an edge and the length between one end-point to one corner of the squares base around it. We first find the direction vector between the two end points. Then we compute a first orthogonal vector A and its opposite vector C. This gives us two corner points. To get the two others, we compute the cross product between the first direction edge with vector A, which gives us a third vector C orthogonal to all other ones. After computing its opposite and rescaling all vectors to their desired length, we create 4 vertices by adding the 4 computed vectors to each end point. The last step, is to link the new surrounding vertices of an end- point to its corresponding vertex at the other end-point and adding two faces to each side of the rectangle.

3) Build spheres at vertices
(*MeshProcessing::create_spheres_on_vertices*)
This function has the same purpose than the build_cylinder, namely to ensure printability in inter-connecting every cylinder at each vertex with a sphere between them. Again, because of time complexity, the function is hard coded for creating a icosahedron but there is a a nbIteration variable that can be changed to yield a more spherical polygon.
The function takes two parameters, namely a Point where the sphere should be computed and its radius. We first need to find the first 12 vertices around the point given. To do this, we compute a special number called the golden ratio that is used to compute the platonic form that is the regular icosahedron. The function will then go through an iteration algorithm which will split each triangle of a face in 4 sub triangles. To do this it finds the middle of each edges, create a vertex and push it out to match the norm of the radius. Each step will multyply the number of triangles four time, this explains our choice to not use spheres as it would compute a long time for the whole mesh.

III. Pipeline

In this section, we will give a precise pipeline with the transformations we did on our meshes. The first part will be about the pre-processing we did on our meshes and then as we transform the two meshes separately we will give a different pipeline for each of them.

A. Pre-processing

First of all, we had to clean the meshes used for this project. To do so, we used different tools. First of all we used Meshlab to fix some artifacts :

1) Remove zero area faces
2) Remove duplicate faces
3) Remove duplicated vertex
4) Remove faces from non manifold edges
5) Remove unreferenced vertex

Then we wanted to get a smaller amount of elements (vertices, edges and faces) in the meshes so we used 'Quadric Edge Collapse Decimation' with these values :

1) Target number of faces : 50000 for Geralt and 27000 for the skull
2) Quality threshold : 1.0
3) Preserve boundary of the mesh
4) Boundary preserving weight 1
5) Preserve normal
6) Preserve topology

7) Optimal position of simplified vertices
8) Post-simplification cleaning

For Geralt, we decided to cut his hair as we think the result of separating his head is not interesting if we keep his ponytail. (still with Meshlab)

Concerning the skull, we noticed that the mesh was composed of several connected components (mainly because of the teeth). This was causing some issues when remeshing the graph, so we merged the skull in a single connected component using Meshmixer's "make solid" functionality. This changed the number of vertices to 54'000, which is still fine.

B. Geralt

Here is a description of all the transformations done on Geralt. (see Figure 1 to see the result after each transformation)

1) Remeshing & smooth: At first we want to have a regular model before working on the other transformations. We decided to do an average remeshing using the code we wrote for lab 6 so that the faces contained in Geralt have nearly all the same size. As the mesh doesn't look really good after this remeshing we decide to do 1 iteration of uniform laplacian smoothing.

2) Separate head: We use the algorithm 'Separate Head' (using Log) described above with $\gamma = 3$ and $threshold = 20$.

3) Create fracture: We use the algorithm 'Create fracture' described above.

4) Delete big faces: We use the algorithm 'Delete big faces' described above.

5) Thicken the mesh: Before exporting the mesh into Meshmixer to thicken our mesh and make it printable, we go back to Meshlab and do the same cleaning we did at the beginning to avoid some artifacts. Then we go on Meshmixer and in the 'print' section, we repair the mesh. It will make it stronger for the printing by thickening it a bit.

C. Skull

Here is a description of all the transformations done on the skull. (see Figure 2 to see the result after each transformation)

1) Remeshing: We first apply a height-based remeshing algorithm on the skull to obtain a regular triangulation, so that the holes in the Voronoï pattern are nice and bigger on the top of the head.

2) Dual graph: We compute the dual graph with the method *skull_dual_graph_pattern*. Each edge of the dual is represented by a cylinder.

3) Icosahedron: Then, we put a sphere on each vertex of the dual graph. The method *skull_dual_graph_pattern* takes care of that as well.

D. Add lamp base

Now that our meshes are ready we have to merge them in one mesh and scale it in function of the lamp base given. We do it using '3D Builder'. We decided to print the lamp base directly with our model to prevent a nasty surprise after the printing (not being able to put our mesh on the base for example). The lamp base given was really large at the bottom, we were supposed to put our mesh on it. In our case we want the base to be in the inside of Geralt and just under the skull but if we do it with the given base our final mesh would be of size (20x20x30 cm) and it would take 3 days to print. As we wanted it to take a reasonable amount of times to be printed we decided to create our proper base following the measures given in the project description. After using the custom base we just did to do a small hole at the bottom of the skull to let the lamp pass in it (done using meshmixer).

IV. Result

After creating these two different meshes we had to merge them in one mesh and we also add a custom base so that it will fit the lamp model given in the project. We used different software to obtain our final result.

1) Cinema 4D to create a new base for our model using the measures given in the project description.
2) 3D Builder to merge the 3 meshes together.
3) meshmixer to create the hole at the bottom of the skull so that the lamp can go inside.
4) meshlab to clean the final mesh after all these modifications (same cleaning as in III-A.

The final result (see Figure 3) takes around 14 hours to print (Cura approximation). Despite the difficulties we ran into we are really satisfied of the result and everything we learned within the realization of this project.

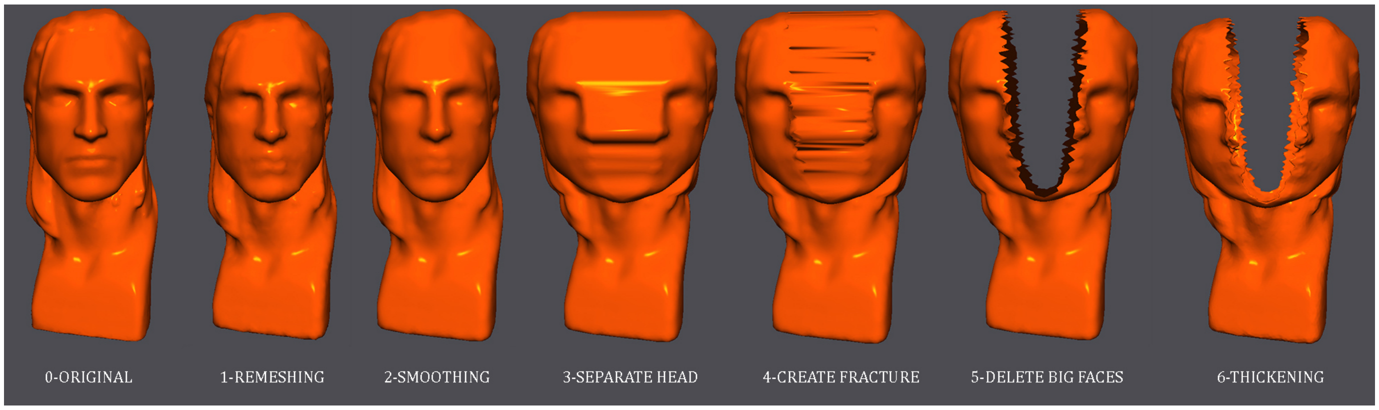

Fig. 1: Transformations on Geralt

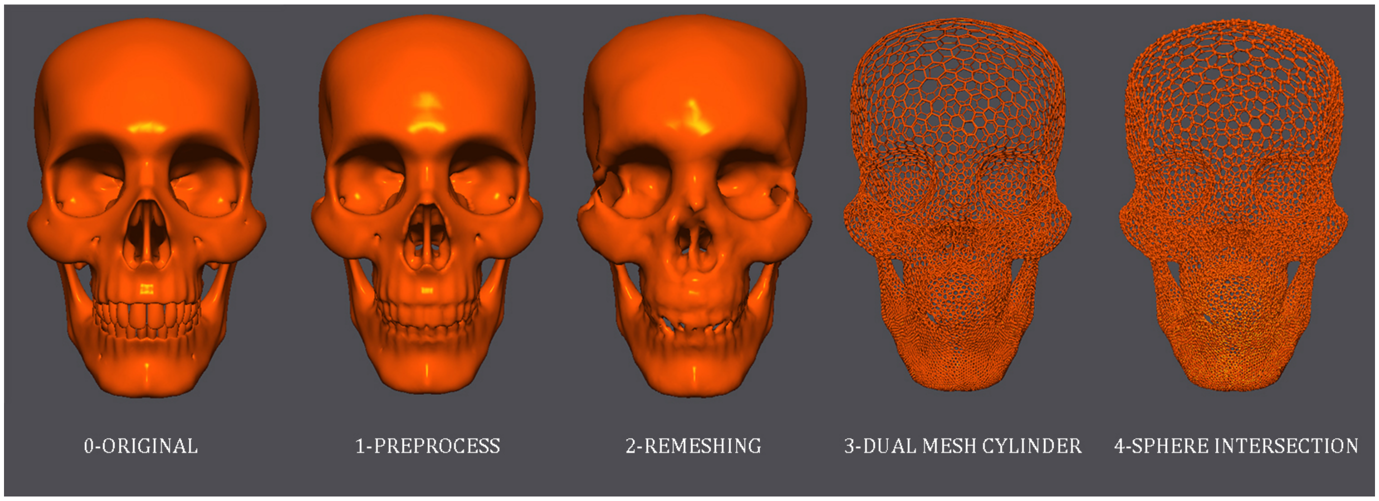

Fig. 2: Transformations on the skull

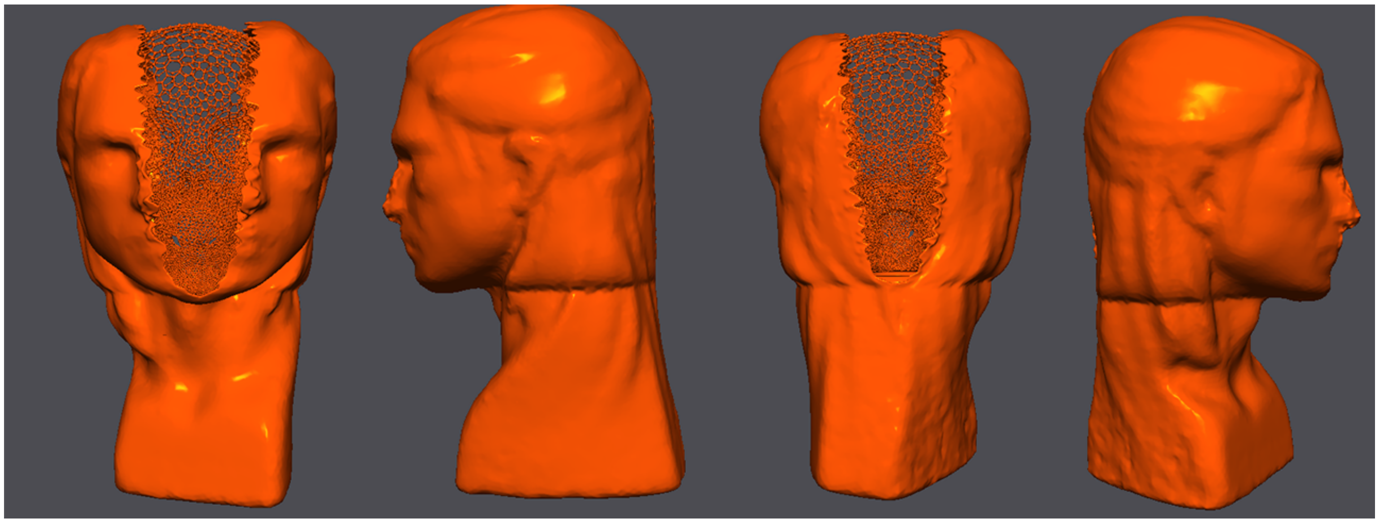

Fig. 3: Final mesh