Delft University of Technology

Geometric Data Processing
IN4255

Practical Assignment 2

(Differential Coordinates and Shape Editing)

Authors:
Cristian Rosiu (5632226),
Maxmillan Ries (5504066),
Nafie El Coudi El Amrani (4771338)
June 7, 2022

1 Task 1 (Differential Coordinates)

1.1 Implementation

In this first task we were requested to compute matrix S containing gradients of triangles in a mesh. To achieve this, our team used the information in the slides, aswell as following the steps presented in the tutorial:

1. Compute gradient matrix G of each triangle
2. Use G to compute final matrix S

For computing a 3x3 gradient matrix **G'** we devised a function which takes as input the normal of a triangle, its vertices and area. The function then yields the correct representation of matrix **G'**. Inside the function we first compute the vectors corresponding to the edges opposite to all 3 vertices. Finally, we set each column of matrix **G'** to be the cross product between the normal and the previously computed edges. Final pseudocode:

Algorithm 1 GetGradientMatrix

Input: N, V, area
Output: G
$e_0 \leftarrow V.p_2 - V.p_1$
$e_1 \leftarrow V.p_0 - V.p_2$
$e_2 \leftarrow V.p_1 - V.p_0$
for i in 0..2 **do**
    $G.setColumn(i, Cross(N, e_i))$
end for
$G \leftarrow \frac{G}{2*area}$

We use this implementation to compute the final matrix G with shape (3F, V), where F is the number of faces/triangles and V number of vertices. To do so, a matrix G' is generate for each triangle in the mesh and its columns are appended to the final matrix G in the order of the global vertex indices of that respective triangle. Final implementation:

Algorithm 2 GetMatrixG

Input: F, V
Output: G
$i \leftarrow 0$
for t in 0..F **do**
    $p \leftarrow m\_geom.getElement(t).getEntries()$
    $N \leftarrow getTriangleNormal(p)$
    $g \leftarrow getGradientMatrix(N, p, m\_geom.getAreaOfElement(t))$
    for j in 0..2 **do**
        for k in 0..2 **do**
            $G.setEntry(i + k, p[j], g.getEntry(k, j)))$
        end for
    end for
    $i \leftarrow i + 3$
end for
return G

From the lectures and tutorial we know that matrix S is composed in the following way:

$$S = G^T M_v G \tag{1}$$

As it can be observed, the final piece needed to compute matrix S is the computation of mass matrix M_v. The mass matrix is just a diagonal matrix in which, each diagonal element is represented by a 3x3 diagonal matrix containing the area of a specific triangle in the mesh:

$$M_v = \begin{bmatrix} A_{T_1} & ... & 0 \\ ... & ... & ... \\ 0 & 0 & A_{T_m} \end{bmatrix} \tag{2}$$

The algorithm for computing such a matrix was quite easy to implement. In the beginning we generate a matrix with a shape of (3F, 3F). Then, the diagonal elements are filled with the respective areas.

Lastly, S is computed using formula (1). We first compute the transpose of G and multiply the rest.

1.2 Usage

One can call the function getGradients() which returns a custom data structure called Gradients. We designed this structure such that it can store all the computed matrices from task 1 (i.e S, G and $G^T M$). After retrieval, the user has access to all data needed for further computations:

Algorithm 3 GetGradients

Output: gradients
$gradients \leftarrow newGradients()$
$V \leftarrow m\_geom.getNumVertices()$
$F \leftarrow m\_geom.getNumElements()$
$G \leftarrow getMatrixG(F, V)$
$M \leftarrow getMatrixM(F)$
$GTM \leftarrow G^T \cdot M$
$gradients.setG(G)$
$gradients.setS(GTM \cdot G)$
$gradients.setM(M)$
$gradients.setGTM(GTM)$
return gradients

1.3 Tests

Since it is difficult to verify the results of our method on a normal mesh, we opted to create tests with some dummy values (4 vertices and 2 faces) to be able to calculate the results by hand. The unit tests can be found in: *DifferentialCoordinatesTest*. The test compare the results found by our implemented methods and what we calculated by hand. As you can see in the image below, all the test we have created pass.

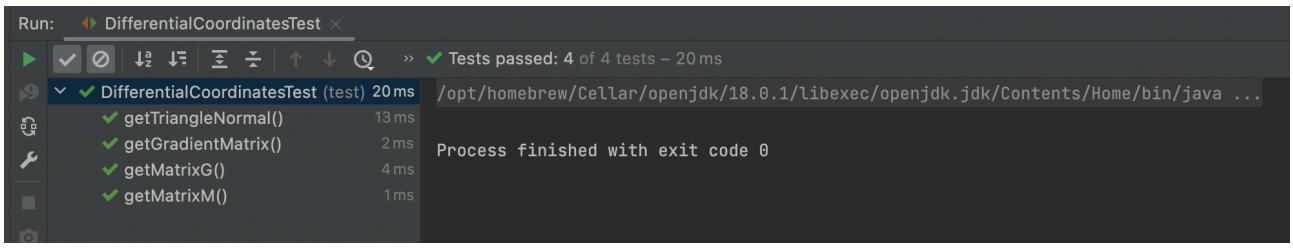


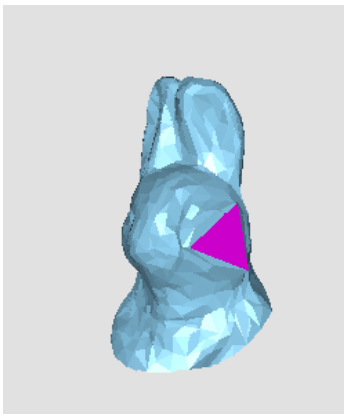
Figure 1: Test suite results

2 Task 2 (Shape Editing using differential coordinates)

2.1 Examples of results

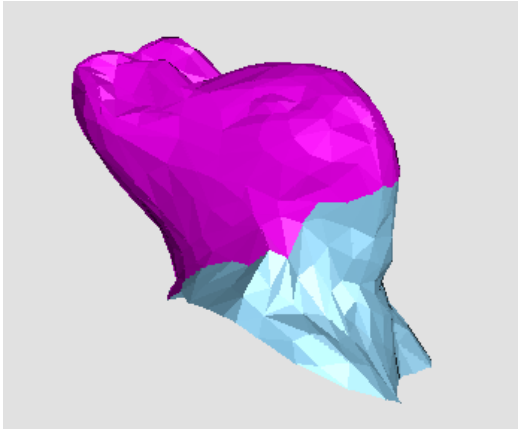
(a) Ears scaled in all directions

(b) Nose scaled along y-axis

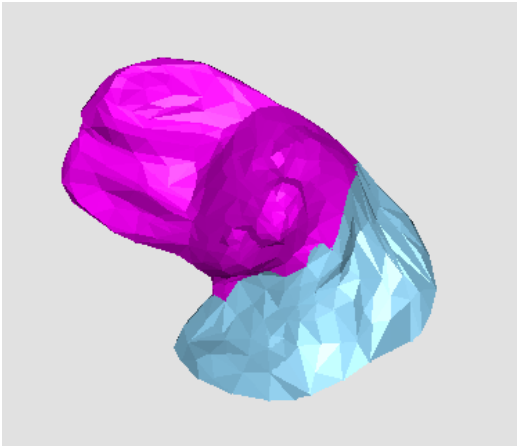


(c) Only one triangle scaled up

Figure 2: Experiments on rabbit head mesh. Note in (c) you can see how well the triangles next to the selected one move to allow the selected triangle to transform.

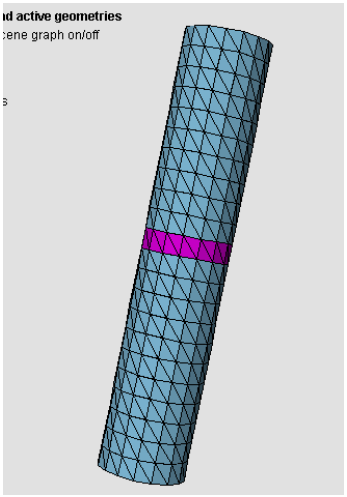


(a) Entire head rotated by 90 degrees on x-axis

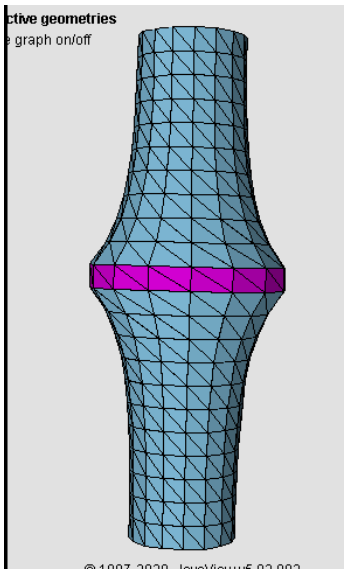


(b) Entire head rotated by 90 degrees on z-axis

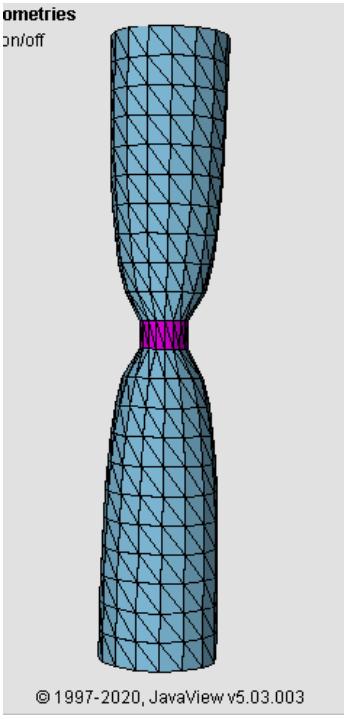
Figure 3: Apply rotation matrices on rabbit head.



(a) Original mesh



(b) Blown up cylinder in the middle



(c) Shrunk cylinder in the middle

Figure 4: Experiments on cylinder mesh. Note in (b) & (c) you can see how our implementation allow the cylinder to deform while the mesh stays continuous.

2.2 Implementation

For this task we were in need of a function which can generate the correct embeddings v_x, v_y, v_z. Therefore, we design a method which iterates over all the vertices and adds each coordinate to its specific embedding. The pseudocode:

Algorithm 4 GetEmbeddings

```
Output: embeddings
for vertex in V do
    for i in 0..2 do
        embeddings[i].addEntry(vertex[i])
    end for
end for
```

After the computation of each embedding, we calculated their respective gradients (i.e. Gv_x, Gv_y, Gv_z). Next, our algorithm iterates over all triangles that are selected and modify the previously computed gradients using the transformation matrix A that is passed as a parameter. This yields the new gradient embeddings that will later be used in solving the following 3 systems:

$$S\tilde{v_x} = G^T M_v \tilde{g_x} \tag{3}$$

$$S\tilde{v_y} = G^T M_v \tilde{g_y} \tag{4}$$

$$S\tilde{v_z} = G^T M_v \tilde{g_z} \tag{5}$$

For solving the actual systems, we made use of the libraries mentioned in the instructions. We tested both of them (i.e. MUMPS and Biconjugate) on 2 different operating systems (macOs and Windows 10). After the solutions are computed we update the vertices of the mesh and shift the barycenter to its previous location.

Until this point, the method works perfectly and applies the transformation on the selected vertices as expected. However, the whole mesh gets translated to the center of the world. An easy solution to this problem is to compute the centroid of the mesh before and after the application of the brush. The vector between the new centroid and the original one is the translation vector that we can apply to all vertices of the mesh to put it back in its place in the world space.

2.3 Usage

To apply a brush effect using a transformation matrix A, one can just call the function applyBrushes(PdMatrix A).

Algorithm 5 ApplyBrush

```
Input: Matrix A (brush matrix)
Get matrices S, G, M
embeddings ← GetEmbeddings()
Compute gradients g of each embeddings
for each face f in Faces do
    if f is selected by user then
        multiply the corresponding gradient with A
    end if
end for
Solve S · ṽ = G^T · M_v · g̃
Update mesh using the new approximations
```

3 Discussion

The tool we built works surprisingly well on all input matrices. We have tested it on scaling, rotation and also translation matrices (see photos above for some examples). Since we did not use projective geometry in this

setting, and we are not sure if it is actually possible to be applied to this algorithm (we need to look it up), we cannot do translations in the z direction. However, we tried translations on the xy-plane. It works, but it doesn't look good.

The main benefit of the chosen method (Geometric Laplace coordinates) is the fact that it takes into account the irregularities of the triangles in the neighborhood, since information about the angles is used. The alternative would have been to use the combinatorial Laplace coordinates, but this alternative method has an important disadvantage. It doesn't take into account the irregularities of the neighborhood and assumes that all triangles are of similar sizes and have similar angles.

Even though the chosen method is theoretically good. Our implementation is quite slow. As you can see in the images, all our examples have meshes with low number of triangles. This was on purpose, since one operation on a triangulated mesh of above 40k triangles took over 10 minutes to calculate. We were forced to work with a low number of triangles, in the range of 1 to 5 thousand triangles per mesh.

We didn't do tests for this task, since it uses the same methods as the first task and methods coming from the framework. Also, calculating the values by hand will be time-consuming and can easily be wrong with the huge amount of calculations that were needed. Therefore, we opted to verify the results of our implementation visually by applying different types of matrices.

With more time, we can extend this work by optimizing our code. The current code is good, but does not follow coding best practices (for example: unnecessary memory allocation). Other than optimizing our implementation, we can speed up the computation of our method for large meshes by using approximation algorithms. Two methods were mentioned in the lectures. Either couple a coarse mesh to the fine mesh and to compute the deformations on the coarse mesh or restrict the computations to a subspace of vertex displacements.