Université de Bourgogne

Software Engineering Project

Laplacian Mesh Processing

Students:
Flávia Dias Casagrande
Luis Miguel Zapata
Marcel Sheeny de Moraes

Professor:
Yohan Fougerolle

A project submitted in fulfilment of the requirements for the subject of Software Engineering

in the

VIBOT - Computer Vision and Robotics Master Programme
Centre Universitaire Condorcet

January 2015

UNIVERSITE DE BOURGOGNE

Abstract

VIBOT - Computer Vision and Robotics
Centre Universitaire Concorcet

Software Engineering

Laplacian Mesh Processing

by Flavia Dias Casagrande, Luis Miguel Zapata & Marcel Sheeny de Moraes

The field of application of surface processing has increased in the past years and recent developments ensure that the number of applications will be even greater. The project here described has the main application in mesh processing by using the Laplacian operator. A user interface was implemented, providing applications such as spectral decomposition, mesh smoothing and editing. Besides that, a basic code was provided, which was improved by employing C++ best practices seen in class.

Contents

List of Figures

Chapter 1

Introduction

The representation and processing of 3D models have been presenting a great development for the past years in the field of computer graphics. The Laplacian operator can be pointed out as one of the main factors that improves the 3D mesh processing, taking in account all the applications its properties can provide.

For instance, the spectral decomposition taken from the Laplace matrix: it has a large field of applications, including for example seismic data processing [4] and magnetic resonance imaging [5]. Besides that, applications such as mesh smoothing [6] and mesh editing [3].

In this project, the objective is to understand all the theory behind mesh processing and apply it to a provided code. The code given should be improved in the sense of applying best practices of C++ studied in class and besides that, the Laplacian operator should be implemented as well as its applications.

This report is organized in the following Chapters. The current Chapter gathers all the basic definitions and theory necessary for the complete understanding of the problem and solution implemented. In Chapter 2 there is the description of the project management to execute the project. Chapter 3 contains the improvement in the code, including new classes and rearrangement of methods and classes. Chapter 4 describes all the code implemented. Chapter 5 describes the user interface developed. Chapter 6 shows the results for each application. Finally, in Chapter 7 the conclusion and future works proposed.

1.1 Basic Definitions

1.1.1 3D Mesh

A 3D mesh is composed by a set of vertices, edges and faces. The connection of two vertices forms an edge and connections of edges form a face. Faces can be defined by either triangles or any other convex polygons. For this project, triangular meshes are considered.

The use of 3D meshes in many fields over the past years created the need of development in several applications such as smoothing, simplification, compression and parametrization of the model under study, easing their manipulation.

One of the approaches to implement these applications to a mesh is to perform its spectral decomposition, in which the mesh data is expressed as a linear combination of a set of orthogonal basis functions, each one characterized by a "frequency" [7].

The classical Fourier transform of a signal is the decomposition of the signal into a linear combination of the eigenvectors of the Laplacian operator [6]. Therefore, it is necessary to extend this concept to a 3D mesh in order to get the mesh frequencies.

1.1.1.1 The Laplacian Operator

Let a triangular mesh be characterized by (V, E, F), which represents the set of vertices, edges and faces, respectively.

It is possible to find the differential coordinates of the mesh by the vertices. They keep the local surface shape, by gathering information such as the normal direction and the mean curvature (Figure 1.1).

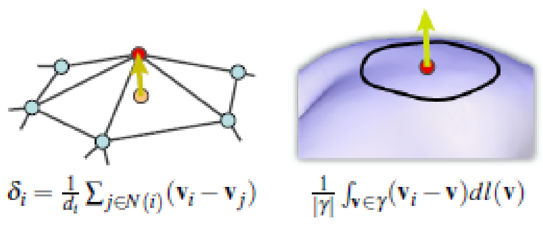

$\delta_i = \frac{1}{d_i}\sum_{j\in N(i)}(\mathbf{v}_i - \mathbf{v}_j)$
$\frac{1}{|\gamma|}\int_{\mathbf{v}\in\gamma}(\mathbf{v}_i - \mathbf{v})dl(\mathbf{v})$

FIGURE 1.1: Differential coordinates.[1]

The differential coordinates (or so-called δ-coordinates) are defined by Equation 1.1, which represents the difference between the coordinates of $\mathbf{v}_i$ (v_i, y_i, z_i) and the center of mass of its neighbors in the mesh.

$$\delta_i = (\delta_i^{(x)}, \delta_i^{(y)}, \delta_i^{(z)}) = \mathbf{v}_i - \frac{1}{d_i} \sum_{j \in N(i)} v_j \tag{1.1}$$

where (d_i is the number of neighbors N).

The equation 1.1 can also be represented in a matrix form, which makes the calculations simpler. Therefore, let the adjacency matrix A be defined as in 1.2 and a diagonal matrix D as $D_{ii} = d_i$.

$$A_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

Then the graph (or topological) Laplacian matrix can be calculated as in Equation 1.3 or 1.4, which is a symmetric version. Figure 1.2 shows an example.

$$L_s = D - A \tag{1.3}$$

or even

$$(L_s)_{ij} = \begin{cases} d_i & i = j \\ -1 & (i,j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{1.4}$$

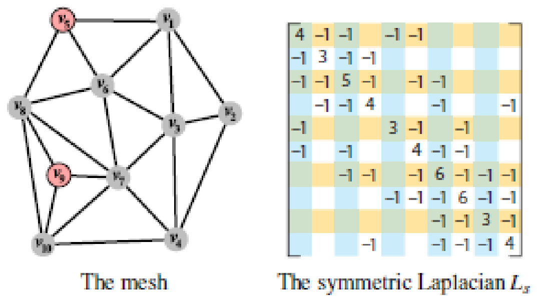


FIGURE 1.2: Mesh example and symmetrical Laplacian matrix. [1]

A better approximation for the calculation of the differential coordinates was employed by [8], considering non-uniform weights, as in Equation 1.5.

$$\delta_i^c = \frac{1}{|\Omega_i|} \sum_{j \in N(i)} \frac{1}{2}(cot\alpha_{ij} + cot\beta_{ij})(v_i - v_j) \tag{1.5}$$

where $|\Omega_i|$ is the size of the Voronoi cell of i and α_{ij} and β_{ij} are the two angles opposite of edge (i, j) (Figure 1.3).

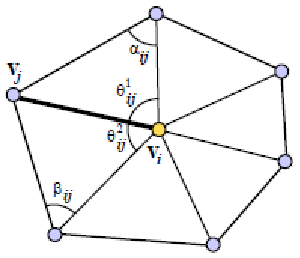


FIGURE 1.3: Voronoi cell and the Geometric Laplacian angles. [1]

1.2 Applications

The different Laplacian matrices calculated respond better to different applications.

The geometric differential coordinates are useful for applications such as filtering and editing, when the mesh geometry is known.

For compression applications though, the differential coordinates provided by the topological Laplacian are enough, as the meshes will not require extra information such as the surface shape.

In this section the applications implemented for the program are explained.

1.2.1 Spectral Decomposition

As mentioned before, the spectral decomposition of a mesh is done by the eigenvectors of the Laplacian operator. Then, from either the topological or geometrical Laplacian matrices calculated, the eigenvectors are computed.

Therefore, choosing a frequency corresponds to selecting the eigenvectors until that frequency.

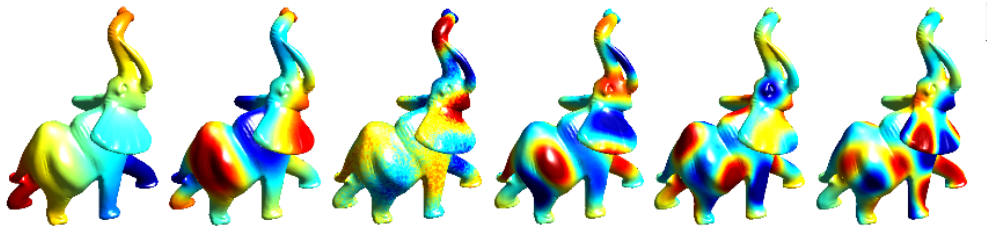


FIGURE 1.4: Spectral decomposition.[2]

1.2.2 Smoothing

The “low frequency” coefficients of a mesh contribute more to the mesh data than the “high frequency” ones, which represent mainly the details.

Therefore, in order to smooth a mesh, only a certain number of spectral coefficients are used, setting the others to zero. This procedure is equivalent to a low-pass filtering. Figure 1.5 presents the smoothing performed by [7].

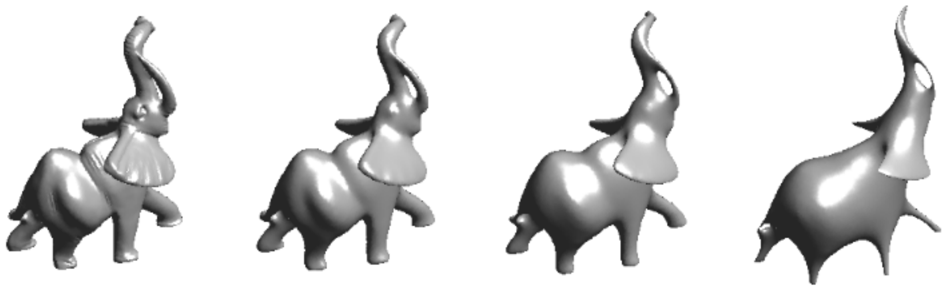


FIGURE 1.5: Smoothed mesh. [2]

1.2.2.1 Frequency Removal

There is also the possibility of removing a specific frequency, so the eigenvector corresponding to the frequency should be set to zero. This procedure is equivalent to a band-reject filtering.

1.2.3 Mesh Editing

As mentioned before, the mesh editing procedure requires the use of the geometrical Laplacian, which employs the differential coordinates. They will retain the details of the surface, after applying the desired modifications [3].

The edition is implemented in few steps (Figure 1.6). Firstly, it is necessary to define a region of interest (ROI) which means the points there will be moved. Then, inside the ROI a point will be chosen to be a handler and all the other will be anchors.

The surface is reconstructed with respect to the final position of the handler.

Regarding the calculations of the new vertices, selecting anchors means adding lines to the symmetrical Laplacian matrix. which will form $\tilde{L}$. This is useful since matrix L_s is singular, and therefore not possible to calculate its inverse to solve $v_i = L_s^{-1}\delta_i$.

Therefore, the reconstruction of the surface under edition is then completed by solving the linear system by using the least squares method and Equation 1.6.

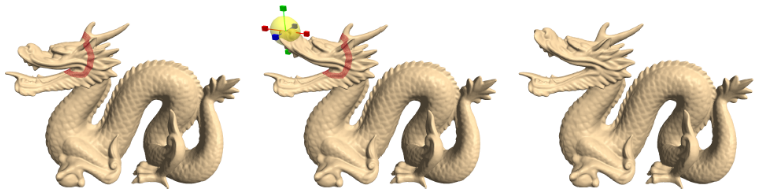
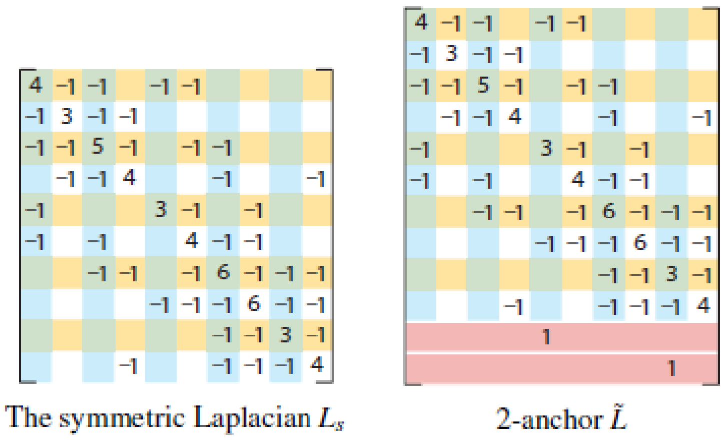


FIGURE 1.6: Mesh editing. Selection of region of interest (red line), anchor (yellow sphere) and result. [3]



$$\begin{bmatrix} 4 & -1 & -1 & & -1 & -1 & & & & \\ -1 & 3 & -1 & -1 & & & & & & \\ -1 & -1 & 5 & -1 & & -1 & -1 & & & \\ & -1 & -1 & 4 & & & -1 & & & -1 \\ -1 & & & & 3 & -1 & & -1 & & \\ -1 & & -1 & & & 4 & -1 & -1 & & \\ & & -1 & -1 & & -1 & 6 & -1 & -1 & -1 \\ & & & & -1 & -1 & -1 & 6 & -1 & -1 \\ & & & & & & -1 & -1 & 3 & -1 \\ & & & -1 & & & -1 & -1 & -1 & 4 \end{bmatrix}$$

The symmetric Laplacian L_s

$$\begin{bmatrix} 4 & -1 & -1 & & -1 & -1 & & & & \\ -1 & 3 & -1 & -1 & & & & & & \\ -1 & -1 & 5 & -1 & & -1 & -1 & & & \\ & -1 & -1 & 4 & & & -1 & & & -1 \\ -1 & & & & 3 & -1 & & -1 & & \\ -1 & & -1 & & & 4 & -1 & -1 & & \\ & & -1 & -1 & & -1 & 6 & -1 & -1 & -1 \\ & & & & -1 & -1 & -1 & 6 & -1 & -1 \\ & & & & & & -1 & -1 & 3 & -1 \\ & & & -1 & & & -1 & -1 & -1 & 4 \\ & & & & 1 & & & & & \\ & & & & & & & & 1 & \end{bmatrix}$$

2-anchor $\tilde{L}$

FIGURE 1.7: Anchor matrix.

$$\tilde{\mathrm{x}} = (\tilde{\mathrm{L}}^T \tilde{\mathrm{L}})^{-1} \tilde{\mathrm{L}}^T b \tag{1.6}$$

where b is the δ with added lines which should correspond to the new vertices position.

Chapter 2

Project Management

2.1 Methodology

The first task of the project was the reading and complete understanding of the main paper and all other references provided, including all additional material for the Laplacian operator and the spectral decomposition.

Then, the code given for the assignment was studied and understood.

After the materials provided were all studied, the programming phase started. There were two main tasks for the programming. First one was to improve the code given, which was really challenging. As will be mentioned in Chapter 3, GLUT user interface was replaced by the Qt one, which required a long time of work. Besides that, some classes were rearranged and some created. The second part was to implement the Laplacian operator and the spectral decomposition of meshes.

With all the basic work finished, the group decided to implement some of the applications described in the main paper. First smoothing and frequency removal were done. Then, a simpler version of mesh editing was also implemented.

Regarding the report writing, the group started it since the material's reading, in order to gather all the references.

2.2 Schedule

Figure 2.1 shows the time schedule planned for the project's execution and the real schedule.

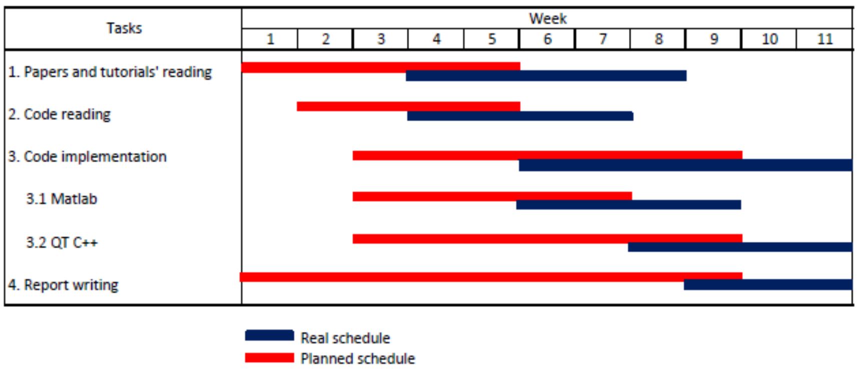


FIGURE 2.1: Planned and real schedule.

The project was executed relatively late. It is due mainly to the fact that the group was not really familiarized with all the C++ and OpenGL [9] syntaxes and the object-oriented programming.

Besides that, it is required to understand completely the theory and properties of the Laplacian operator before moving to coding.

However, the group could completely finish the project assignment, even including some other applications.

Chapter 3

Improvement in the Code

3.1 Software given

The basic code for the development of this project was given to the group by Professor Yohan Fougerolle. It can be used for general purposes in mesh processing applications, such as mesh smoothing, segmentation and animation.

It is a framework in C++ and OpenGL [9] which display meshes and can easily access the information about their vertices, edges and faces. This software was implemented using the GLUT [10], which is a simple toolkit to render OpenGL.

There are basically two classes: *Mesh.h* class that stores the list of vertices, faces, vertices normals, face normals and colors; and *NeighbourMesh.h* class that builds a list of the neighbours, so that we know for each vertex its neighbour's vertices. It also builds all the neighbour's faces for each vertex and builds all neighbour faces for each face. With this information it is possible to apply the mentioned computer graphics applications.

However the program presents some software engineering problems in the code, for instance only public methods in *NeighbourMesh.h* and *Mesh.h* classes, many global variables, no pattern, a poor objected-oriented approach and poor software architecture.

Letting all variables and methods as public provides a faster development, however it can let the software more suitable to errors. For instance modifying variables that can change internally the implementation of some algorithm and corrupting the software.

Using global variables is also a way of fast development, although it is much more difficult to control how the variable changes in the software. In many cases it is used because of the GLUT, since it uses callbacks in C (no object-oriented approach), which forces us to use global variables.

The software also put many parameters of the rendering together, such as the camera and light parameters, making it difficult to read and understand the code. Therefore, it is also a good practice to make the as object-oriented. Then, it will be more readable and easier to update the code.

3.2 Refactoring

Our first task was to try to take out the public methods and variables from *Mesh.h* and *NeighbourMesh.h* classes. Most of the interface to access the attributes in the mesh class were already implemented, then we just continued to put all those methods as public and as protected the list of vertices, faces, etc.

They were declared as protected because the *NeighbourMesh.h* is inherited from *Mesh.h*, so it would be more convenient to allow the inherited classes to have access those variables.

The same was done in the *NeighbourMesh.h* class. Although, as there was not much get and set functions, then we implemented those methods.

We also put the list of point to points, point to faces and face to faces as protected variables, since we created a new class *LaplacianMesh.h* that contains our Laplacian implementation. This new class is inherited from *NeighbourMesh.h*.

In the Figure 3.1 we can see the class diagram of the *Mesh*, *NeighbourMesh* and *LaplacianMesh* with the main methods using the Unified Modeling Language (UML) approach.

More details about the implementation of the *LaplacianMesh* will be discussed in the Chapter 4.

In the Codes 3.1, 3.2, 3.3 it can be seen the header files of the *Mesh.h*, *NeighbourMesh.h* and *LaplacianMesh.h*.

Only the main attributes are shown, except the *LaplacianMesh* that describes the whole class. As mentioned before, the details regarding the implementation will be discussed in the Chapter 4.

```
class Mesh
{
protected:
    // Original Vertex array
    vector<Vector3d> originalVertices;
```

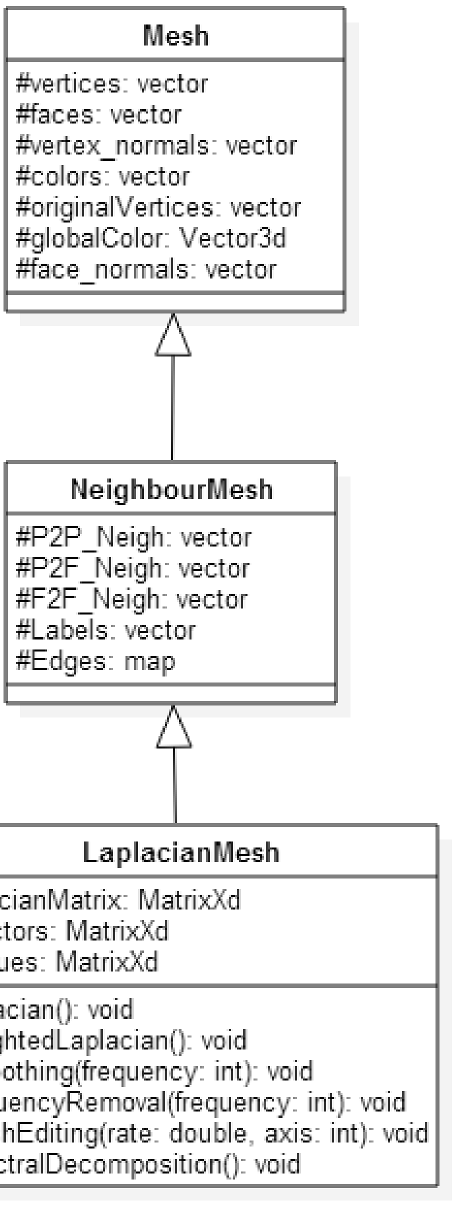


FIGURE 3.1: Class diagram of the *Mesh*, NeighbourMesh and *LaplacianMesh* with the main attributes and methods.

```
// Vertex array
vector<Vector3d> vertices;
// Face array
vector<Vector3i> faces;
// Color array
vector<Vector3d> colors;
// Global color (paint the whole object with the same color using this
value)
Vector3d globalColor;
// Texture coordinate array ... probably useless for you in this project
vector<Vector2d> textures;
```

```
    // Face normal array
    vector<Vector3d> face_normals;
    // Vertex normal array
    vector<Vector3d> vertex_normals;
};
```

Listing 3.1: Header of the Mesh (just the main attributes)

```
class NeighbourMesh : public Mesh
{
protected:
    //attributes
    vector < set<int> > P2P_Neigh;
    vector < set<int> > P2F_Neigh;
    vector < set<int> > F2F_Neigh;
    vector<double> Labels;
    map < pair <int,int>,  set<int> > Edges;
};
```

Listing 3.2: Header of *NeighbourMesh* (Just main attributes)

```
class LaplacianMesh : public NeighborMesh
{
public:
    //Default constructor
    LaplacianMesh();
    //Compute laplacian matrix
    void laplacian();
    //Compute weighted laplacian
    void weightedLaplacian();
    //Smooths the mesh according to the frequency
    //It puts all the eigenvectors equal to zero from the frequency given
    void smoothing(int frequency);
    //remove only one frequency of the mesh
    void frequencyRemoval(int frequency);
    //Computes the weight for the weightedLaplacian implementation
    MatrixXd computeWeight();
    //Mesh editting according to the size and the axis
    //0 = x, 1 = y, 2 = z
    //it uses the extreme points of the mesh (lowest and highest points)
    //as anchors, and move one of them to strech to mesh
    void meshEditing(double rate, int axis);
    //Calculate the spectral decomposition
    void spectralDecomposition();
    //Set laplacian Matrix
    void setLaplacianMatrix(MatrixXd laplacianMatrix);
    //Get EigenVectors
    MatrixXd getEigenVectors() {return eVectors;}
    //Build spectral labels
    void BuildSpectralLabels(int i);
private:
    //Computes the cotangent
    double cot(double i);
    //Eigen values
    MatrixXd eValues;
    //Eigen Vectors
    MatrixXd eVectors;
    //Laplacian Matrix
    MatrixXd laplacianMatrix;
};
```

LISTING 3.3: Header of *LaplacianMesh*

3.3 New Classes

Regarding the rendering part in the *Scene.h* class of the original code, we could see that would be convenient to put the camera and light parameters in a new class, since the variables for it were well designed.

We just needed to get the parameters of the *Scene.h* class and create two new classes, *Camera.h* and *Scene.h*. With those classes it became much easier to understand the code, and also to set up the user interface.

Creating a new class for the light makes easier, for example, the creation of multiple lights in the scene with different parameters.

The header files of those new classes can be seen in the Codes 3.4 and 3.5.

```
//class that contains the camera parameters
class Camera
{
public:
    //Constructor
    Camera();
    //get camera target
    Vector3d getTarget();
    //get camera position
    Vector3d getPosition();
    //Init camera parameters
    //This function also sets the extreme points in the mesh
    void initCamera(NeighborMesh& mesh);
    //Get Z near
    double getZnear ();
    //Get Z far
    double getZfar ();
    //Get distance from the mesh
    double getDistance();
private:
    //Position of the camera
    Vector3d position;
    //Target of the camera
    Vector3d target;
    //z near plane
    double znear;
    //z far plane
    double zfar;
    //distance from the mesh
    double distance;
};
```

Listing 3.4: Header of *Camera* class

```
class Light
{
public:
    //Constructor
    Light();
    //Set position of the light
    void setPosition(float posX, float posY, float posZ);
    //Set position of the light passing the camera position
    void setPositionByCamera(float posX, float posY, float posZ);
    //Set direction of the light
    void setDirection(float posX, float posY, float posZ);
    //Get specular parameters
    float* getSpecular() {return specular;}
    //get ambient light parameters
    float* getAmbient() {return ambient;}
    //get diffuse light parameters
    float* getDiffuse() {return diffuse;}
    //get shininess parameter
    float getShine() {return shine;}
    //get direction of the lights
    float* getDirection() {return dir;}
private:
    //specular parameters
    float specular[4];
    //shininess
    float shine;
    //ambient light
    float ambient[4];
    //diffuse light
    float diffuse[4];
    //light position
    float position[3];
    //light direction
    float dir[3];
};
```

Listing 3.5: Header of *Light* class

3.4 Qt OpengGL Integration

One of the tasks proposed by the group was to use the Qt with OpenGL instead of the *GLUT* of the original version. Using Qt provides many advantages over *GLUT*.

The Qt is a general user interface framework, which we can design the user interface easily by using the Qt Creator IDE. Therefore, by using the Qt we could design our program in a professional layout and also use and test the Laplacian algorithms easier.

Another great advantage of the Qt over *GLUT* is that with Qt everything is in object-oriented approach, making it simpler to program. With Qt we could take out all the global variables of the original software.

To make this integration, first we needed to create a *QWidget*, that is a general widget from Qt, in order to promote this widget into a *QGLWidget*.

We created a class called *GLWidget* that is inherited from *QGLWidget*. With the *QGLWidget* we just needed to find equivalent functions. For example, the GLUT uses *void glutDisplayFunc(void (*func)(void))* as a callback to display, and the equivalent function in *QGLWidget* is *void paintGL ()*. So we just needed to find equivalent Qt functions to integrate the software given into the Qt approach.

All the implementation of the *GLUT* was previously in the *Scene.h* class, then we placed all the new implementation of the rendering in the *GLWidget.h* class and we could remove the *Scene* class.

Some functions such as draw axis use *GLUT* functions, then when we remove the *GLUT* we could not call those functions any more. Therefore, we decided to remove the functions that use the *GLUT*.

Another task proposed by the group was to remove all the *OpenGL* functions in all the classes that were not the *GLWidget* class. Then we can have a software architecture based on the Model-View approach, where all the implementation is on the model and all the rendering and display is on the View. There were many functions for drawing in the *Mesh* class, for example. We simply put those functions in the *GLWidget* class.

It was basically how we refactored the code. In summary, we could get a much improved version from the original code, with many applications and an easy user interface to use.

Chapter 4

Program Implemented

In order to familiarize with meshes' coding and understand each application, [?] was a great source to help in the implementation in Matlab of some of the applications that were afterwards implemented in Qt using C++.

In this chapter, the Matlab codes are presented as well as the codes in C++.

4.1 Matlab Code

A toolbox for mesh manipulation is available in [?], which was used in the codes provided in this section.

4.1.1 Spectral Decomposition

After the mesh is loaded, the vertices and faces are stored. By using the function *compute_mesh_laplacian* and choosing the desired options (such as the type of Lapacian matrix to be calculated), a Laplacian matrix is returned.

The eigenvectors and eigenvalues are then calculated by the Matlab function *eig* and the mesh is plotted by selecting the eigenvectors that corresponds to the frequency wanted.

```
% -------------------------------Spectral
    Decomposition---------------------------------

% -------------------------------Load mesh---------------------------------
name = 'mushroom';
[vertex,face] = read_mesh(name);
```

```
% ---------------------------Compute laplacian------------------------------
laplacian_type = 'conformal';
options.symmetrize = 1;
options.normalize = 0;
Ls = compute_mesh_laplacian(vertex,face,laplacian_type,options);

% ------------------------Compute Spectral Decomposition--------------------
[E,lambda]=eig(full(Ls));
v=vertex';
subplot(1,5,1);
plot_mesh(v',face,options);
```

Figure 4.1 presents the results for the spectral decomposition implemented.

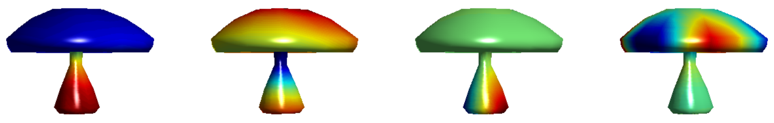


FIGURE 4.1: Spectral decomposition with Matlab code.

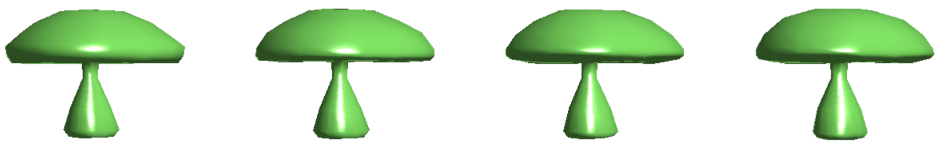
4.1.2 Mesh Smoothing and Frequency Removal

Following the same steps as for the spectral decomposition, until the eigenvectors are calculated, the mesh smoothing will set to zero all the eigenvectors which frequencies are not wanted. By eliminating the "high-frequencies" (eigenvectors corresponding to the smallest eigenvalues), the details will be erased, so that the mesh will be smoothed.

The frequency removal uses the same code, although instead of setting to zero all the unwanted frequencies, it is possible to eliminate specific ones.

Figure **??** shows the mesh smoothing and frequency removal applications.

```
% ------------------------Mesh Smoothing-----------------------------------
% -------------------------------Load mesh---------------------------------
name = 'mushroom';
[vertex,face] = read_mesh(name);
% ---------------------------Compute laplacian------------------------------
laplacian_type = 'conformal';
options.symmetrize = 1;
options.normalize = 0;
Ls = compute_mesh_laplacian(vertex,face,laplacian_type,options);

% ------------------------Compute delta coordinates-------------------------
```


(A)

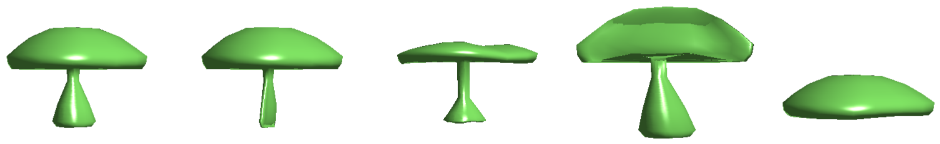

(B)

Figure 4.2: Mesh smoothing (A) and frequency removal (B).

```
[E,lambda]=eig(full(Ls));
v=vertex';
subplot(1,2,1);
plot_mesh(v',face,options);
d=E\v;
%------------------------------Apply Filter-------------------------------
frequency=100;
d(frequency:end,:)=0;
%----------------------------New Vertices---------------------------------
vn=E*d;
subplot(1,2,2);
plot_mesh(vn',face,options);
```

4.1.3 Mesh editing

The mesh editing was implemented in such way that the ROI is all the mesh.

The lowest point of the mesh regarding y-axis was chosen to be a fixed anchor and the highest point the handler.

Two lines were added in the end of the Laplacian matrix calculated (which should be the geometrical one in order to preserve the surface shape). The first, setting to one the element in the position of the vector corresponding to the lowest y values and the second setting the one corresponding to the and highest y value. All the other elements are set to zero.

To the δ vector two lines will be also added. The first will correspond to the coordinates of the fixed point and the second to the new vertice's position.

The new Laplacian with the anchors is inverted and it is possible to calculate then the new vertices.

This procedure enlarges the image, as shown in Figure 4.3.

```
% --------------------Mesh Enlargement with Laplacian----------------------
% -------------------------------Load mesh---------------------------------
name = 'nefertiti';
[vertex,face] = read_mesh(name);
% --------------------------Compute laplacian------------------------------
laplacian_type = 'conformal';
options.symmetrize = 1;
options.normalize = 0;
Ls = compute_mesh_laplacian(vertex,face,laplacian_type,options);
% -----------------------Compute delta coordinates-------------------------
v=vertex';
subplot(1,2,1);
plot_mesh(v',face,options);
d=Ls*v;
%------------------------Set anchors---------------------------------------
scale=1.5;%Scale for enlargement
[valuemin, indexmin]=min(v(:,2));%Find the lowest vertex in Y
[valuemax, indexmax]=max(v(:,2));%Find the highest vertex in Y
%Set the anchors to the laplacian for the lowest and highest vertex
Lanchor=[Ls;zeros(1,indexmin-1),1, zeros(1,length(Ls)-indexmin);
                zeros(1,indexmax-1),1, zeros(1,length(Ls)-indexmax)];
%Lowest vertex on its same position and move the highest in Y direction
danchors=[d;v(indexmin,1),v(indexmin,2),v(indexmin,3);
                        v(indexmax,1),v(indexmax,2)*scale,v(indexmax,3)];
%------------------------Least Mean Square---------------------------------
linv=pinv(full(Lanchor));
vn=linv*danchors;
subplot(1,2,2);
plot_mesh(vn',face,options);
```

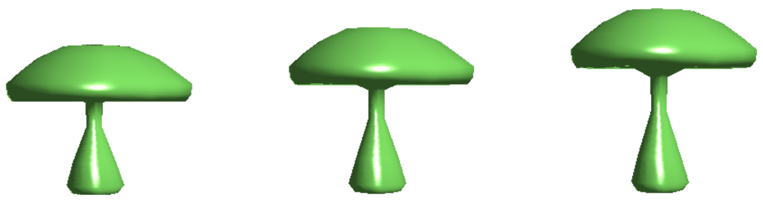


Figure 4.3: Spectral decomposition with Matlab code.

4.2 C++ Code

4.2.1 Graph Laplacian

For computing the Graphic Laplacian the main idea is to use the *P2P_Neigh* variable, which contains the neighbours for each vertex.

First the matrix A is calculated. It is created by setting to 1 the columns corresponding to the neighbours of the i vertex. Afterwards, D is calculated by setting the amount of neighbours that the i has. Finally the Laplacian Matrix is computed by $L = D - A$.

```
{
    //Generates the A and D matrices
    set<int>::iterator it;
    int N=P2P_Neigh.size();
    MatrixXd A;
    MatrixXd D;
    A=MatrixXd::Zero(N,N);
    D=MatrixXd::Zero(N,N);

    //variable that stores the current P2P set
    set<int> neighp;

    //Iterates over all P2P
    for(int i=0;i<P2P_Neigh.size(); i++)
    {
        neighp = P2P_Neigh.at(i);
        for(it=neighp.begin(); it != neighp.end();it++)
        {
            //Putting all weight equal to 1
            A(i,(*it))=1;
        }
        //Put the number of neighbors in the D
        D(i,i)=neighp.size();
    }
    laplacianMatrix = D-A;
}
```

4.2.2 Geometric Laplacian

As stated before the Geometric Laplacian is computed by calculating the weights of the i with the j vertex.

The formula is given by $Cot(\alpha_{ij}) + Cot(\beta_{ij})$.

Once the $weights'matrix$ is computed, the diagonal matrix is found by adding the weights of each row. Once again the Laplacian Matrix corresponds to $L = D - W$.

```
{
    //Compute weight
    MatrixXd W = computeWeight();
    int N = P2P_Neigh.size();
    MatrixXd D;
    D=MatrixXd::Zero(N,N);

    //Iterates for all P2P_neigh
    set<int>::iterator it;
    for(int i=0;i<P2P_Neigh.size(); i++)
    {
        //Sum all the weights to compute D matrix
        D(i,i)=W.row(i).sum();
    }
    laplacianMatrix = D-W;
}
```

In order to find the weights, different steps are performed.

By spanning the $P2P\_Neigh$ matrix the faces in common between the vertex i and j are found by using the $P2F\_Neigh$.

With this information and the $faces$ the two opposite points are found, which are called u and v. Using these new points, four vectors are computed $v1 = u - i$, $v2 = u - j$, $v3 = v - i$, $v4 = v - j$.

Finally the angles between *v1* and *v2* and *v3* and *v4* are calculated, which correspond to α_{ij} and β_{ij}.

```
MatrixXd LaplacianMesh :: computeWeight()
{
    //Initializes the W matrix
    int N = vertices.size();
    MatrixXd W = MatrixXd::Zero(N,N);
    int face[2];
    set<int> intersect;
    set<int>::iterator it;
    set<int>::iterator it2;

    //iterate for all vertices
```

```
for (int i = 0; i < N; i++)
{
    //get neighbors from the current vertex
    set<int> currentP2P = P2P_Neigh[i];

    //iterates for all neighbors
    for (it = currentP2P.begin(); it != currentP2P.end(); it++)
    {
        //get the current neighbor
        int j = (*it);

        //Get the faces intersection
        set<int> faces1 = P2F_Neigh[i];
        set<int> faces2 = P2F_Neigh[j];
        intersect.clear();

set_intersection(faces1.begin(),faces1.end(),faces2.begin(),faces2.end(),
                          std::inserter(intersect,intersect.begin()));

        //if it intersects different than 2 faces, it puts the W = 0
        if (intersect.size() != 2)
        {
            W(i,j) = 0;
        }
        else
        {
            //put the faces index into variable face[]
            int cnt = 0;
            for (it2 = intersect.begin(); it2 != intersect.end(); ++it2)
            {
                face[cnt++] = (*it2);
            }
            //gets the 2 correspondents vertices to compute alpha and beta
            int u = 0;
            int v = 0;
            for (int l = 0; l < 3; l++)
            {
                if (faces[face[0]][l] != i && faces[face[0]][l] != j)
                {
                    u = faces[face[0]][l];
                }
            }
            for (int l = 0; l < 3; l++)
            {
                    if (faces[face[1]][l] != i && faces[face[1]][l] != j)
                    {
                        v = faces[face[1]][l];
                    }
                }

                //compute the vectors in order to compute the angles
                Vector3d vec1 = vertices[u] - vertices[i];
                Vector3d vec2 = vertices[u] - vertices[j];
                Vector3d vec3 = vertices[v] - vertices[i];
                Vector3d vec4 = vertices[v] - vertices[j];

                //compute alpha and beta
                double alpha = acos(vec1.dot(vec2)/(vec1.norm() *
    vec2.norm()));
                double beta = acos(vec3.dot(vec4)/(vec3.norm() * vec4.norm()));

                //Store the weight
                W(i,j) = cot(alpha) + cot(beta);
            }
        }
    }
    return W;
}
```

4.2.3 Spectral Decomposition

Thanks to the *Eigen Library* the *Eigen Values* and *Eigen Vectors* of any of the Laplacian Matrix are computed. They correspond to the Spectral Decomposition of the mesh.

```
{
    //find the eigen values and eiven vectors and store them
    SelfAdjointEigenSolver<MatrixXd> e(laplacianMatrix);
    eValues = e.eigenvalues();
    eVectors = e.eigenvectors();
}
```

4.2.4 Smoothing

The Smoothing application corresponds to first process the δ coordinates by using the *Eigenvectors* as in $\delta = E^{-1}v$.

It has to be done only after computing one of the Laplacian Matrices and its corresponding *Spectral Decomposition*.

Then, the highest desired frequencies are set to 0 in the δ coordinates to finally obtain the new vertices by applying $v = E\delta$.

```
{
    //copy all the original vertices into the Eigen matrix structure
    MatrixXd verticesMat(originalVertices.size(),3);
    MatrixXd delta;
    for (int i = 0; i < originalVertices.size();i++)
    {
        for (int j = 0; j < 3;j++)
        {
            verticesMat(i,j)=originalVertices[i][j];
        }
    }

    //Compute the delta coordinates
    delta=eVectors.inverse()*verticesMat;

    //put the delta equal to 0 from the selected frequency
    int init = frequency;
    for (int i = init; i < delta.rows();i++)
    {
        for (int j = 0; j < delta.cols();j++)
        {
            delta(i,j) = 0;
        }
    }

    //compute the smoothed vertices
    MatrixXd smooth = eVectors*delta;

    //put the smoothed vertices into the vertices to be drawn
    for (int i = 0; i < vertices.size();i++)
    {
        for (int j = 0; j < 3;j++)
        {
            vertices[i][j] = smooth(i,j);
        }
    }

    //put the colors
    for (int i = 0; i < colors.size();i++)
    {
       colors[i]= globalColor;
    }
}
```

4.2.5 Frequency Removal

The Frequency Removal is quite similar to the Smoothing operation. The difference relays in the fact that instead of removing a set of frequencies from the δ coordinates, only one frequency is removed.

Once again, the δ coordinates have to be obtained by using $\delta = E^{-1}v$. One frequency of δ coordinates is set to 0 and the reconstruction of the normal coordinates is done by $v = E\delta$.

```
{
    //convert the vertices into the Matrix eigen structure
    MatrixXd verticesMat(originalVertices.size(),3);
    MatrixXd delta;
    for (int i = 0; i < originalVertices.size();i++)
    {
        for (int j = 0; j < 3;j++)
        {
            verticesMat(i,j)=originalVertices[i][j];
        }
    }

    //compute the delta coordinates
    delta=eVectors.inverse()*verticesMat;

    //set the given frequency to be 0
    for (int j = 0; j < delta.cols();j++)
    {
        delta(frequency,j) = 0;
    }

    //compute the real vertices
    MatrixXd removal = eVectors*delta;

    //put the computed values in the vertices
    for (int i = 0; i < vertices.size();i++)
    {
        for (int j = 0; j < 3;j++)
        {
            vertices[i][j] = removal(i,j);
        }
    }

    //paints the mesh with the global color
    for (int i = 0; i < colors.size();i++)
    {
       colors[i] = globalColor;
    }
}
```

4.2.6 Mesh Editing

Mesh Editing is done preferably after computing the Geometric Laplacian instead of the Graphic Laplacian, since this one contains more information regarding the characteristics of the surface of the mesh. In this algorithm an enlargement or a compression is performed.

As explained before, this algorithm consists in computing the δ coordinates by $\delta = Lv$, where v stands for the set of vertices and L is the Geometric Laplacian. The lower and upper vertices in the mesh in both x and y axis are found, but the user interface as well as the code allows only to do the mesh editing in one of the axis at the same time.

By using the previously found vertices, two anchors are added to the *Laplacian Matrix* obtaining the *laplacianAnchors* and in the other hand the desired location for those two points are set below the δ coordinates obtaining a matrix called *deltaAnchors*. The first point added to this matrix corresponds to an anchor while the other one is called a handler and it is moved along either the x or y axis.

For obtaining the new set of vertex, the Least Mean squares method is performed and the mesh is updated.

```
{
    //set axis to be used
    Vector3d sizes(1,1,1);
```

```
double size = vertices[maxs[axis]][axis] - vertices[mins[axis]][axis];

sizes[axis] = size*rate;

//get the maximum and minimum XYZ indexes
int getMin = 0;
int getMax = 0;
getMax = maxs[axis];
getMin = mins[axis];

//convert the vertices into the eigen Matrix structure
MatrixXd verticesMat(originalVertices.size(),3);
MatrixXd delta;
for (int i = 0; i < originalVertices.size();i++)
{
    for (int j = 0; j < 3; j++)
    {
        verticesMat(i,j)=originalVertices[i][j];
    }
}

//compute the delta coodinates
delta=laplacianMatrix*verticesMat;

//add more 2 rows in the laplacianMatrix and add the 2 points to be anchor
MatrixXd laplacianAnchors(laplacianMatrix.rows()+2,
laplacianMatrix.cols());
MatrixXd B;
B = MatrixXd::Zero(2,laplacianMatrix.cols());
B(0,getMin) = 1;
B(1,getMax) = 1;
laplacianAnchors << laplacianMatrix, B;

//add more 2 rows in the delta coordinates
MatrixXd deltaAnchors(delta.rows()+2, delta.cols());
MatrixXd C(2,3);

//fix the first point
C(0,0) = verticesMat(getMin,0);
C(0,1) = verticesMat(getMin,1);
C(0,2) = verticesMat(getMin,2);

//move the second point according to the parameter given
C(1,0) = verticesMat(getMax,0)+sizes[0];
```

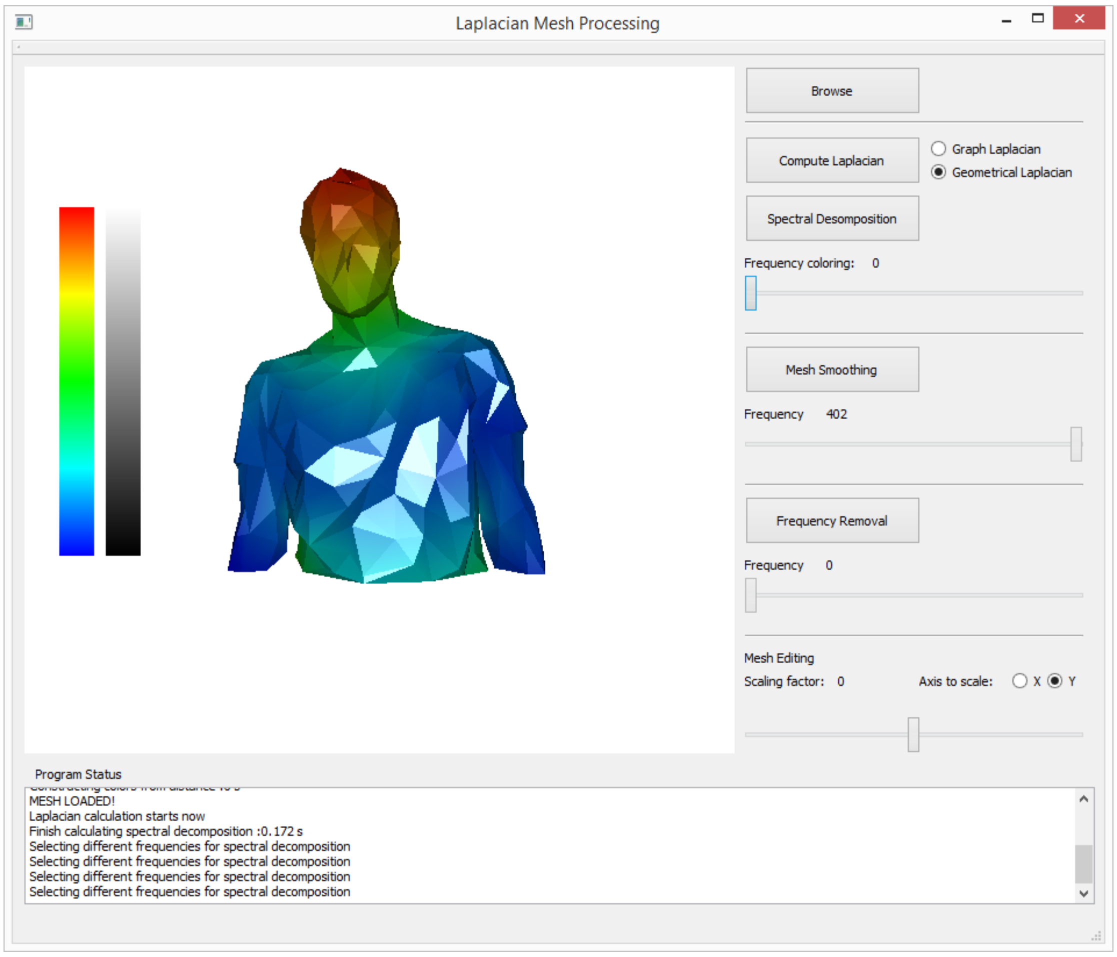
```
    C(1,1) = verticesMat(getMax,1)+sizes[1];
    C(1,2) = verticesMat(getMax,2)+sizes[2];

    deltaAnchors << delta,C;

    //compute the newvertices using the least squares method
    MatrixXd newVertices =
    (laplacianAnchors.transpose()*laplacianAnchors).inverse()*laplacianAnchors.transpose()*del

    //put the vertices computed in vertices list
    for (int i = 0; i < vertices.size();i++)
    {
        for (int j = 0; j < 3;j++)
        {
            vertices[i][j] = newVertices(i,j);
        }
    }
}
```

Chapter 5

User Interface

5.1 Interface Layout

Figure 5.1 shows the interface developed for the Laplacian Mesh Processing application.



Graph Laplacian [✓ Geometrical Laplacian]

Frequency coloring: 0

Frequency 402

Frequency 0

Scaling factor: 0

Axis to scale: X [✓ Y]

```
MESH LOADED!
Laplacian calculation starts now
Finish calculating spectral decomposition :0.172 s
Selecting different frequencies for spectral decomposition
Selecting different frequencies for spectral decomposition
Selecting different frequencies for spectral decomposition
Selecting different frequencies for spectral decomposition
```

FIGURE 5.1: User interface developed.

The interface provides to the user the following manipulations:

Browse button

It opens a browse window so the user can select the correct path of the mesh to be loaded.

Compute Laplacian button

It computes the Laplacian matrix, that can be either the topological (graph) or the geometrical. There are no outputs for the user.

Spectral Decomposition button

It allows the spectral decomposition of the mesh and the choice of the frequency to be displayed.

Mesh Smoothing button

It allows the mesh smoothing of the mesh and the choice of the frequencies to be displayed.

Frequency Removal button

It allows the removal of a specific frequency of the mesh.

Mesh Editing label

It resizes the mesh, getting the scaling factor chosen and the axis that the image will be modified.

Program Status Log:

The log provides information regarding the current activity the program is running.

5.2 How to use it

When initialized, the window will only let enabled for the user the *Browse button*.

As soon as the path is settled and the mesh is loaded, the *Compute Laplacian button* will be enabled.

After the calculation is completed, the buttons *Spectral Decomposition*, *Mesh Smoothing* and *Frequency Removal* and the slider of *Mesh Editing* are enabled.

The sliders of *Spectral Decomposition*, *Mesh Smoothing* and *Frequency Removal* are only enabled when the corresponding buttons are clicked.

When a new mesh is loaded, all the buttons will become unable again.

Chapter 6

Results

Several Mesh Editing Tools could be implemented and the resulting meshes can be observed below. The different Laplacian Operators have different properties and fit better to different tasks, but regardless their good properties, these algorithms have a high computational cost. Operations with bigger meshes than around the 1500 could not be performed.

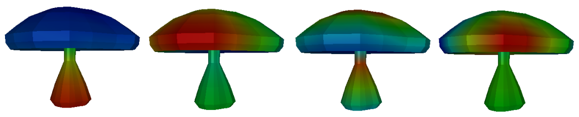
6.1 Spectral Decomposition

By performing the spectral decomposition and colouring the meshes according to a certain *Eigen Vector*, it is easy to see the influence of it in the different regions of the mesh.

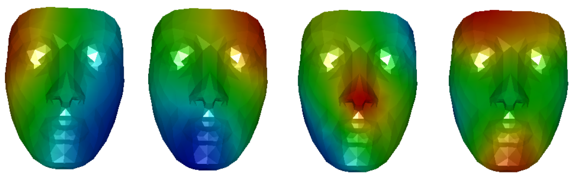
The red color stands for the highest values and the blue for the lowest. Figure 6.1 shows the results reached for the spectral decomposition implemented.

6.2 Smoothing

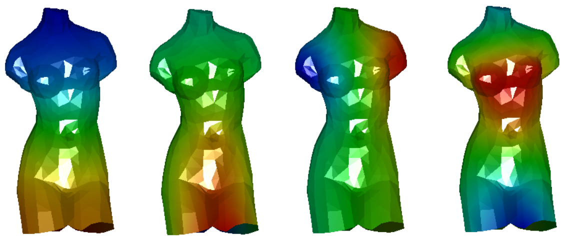
As the smoothing corresponds to a low pass filtering it can be easily seen that the sharp edges in the mesh are removed and if the filter is performed from a frequency close to the first *Eigen Vectors* the mesh fundamental information could be lost. The results of the smoothing Operation can be seen in the figure 6.2.



(A)

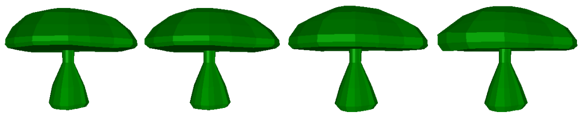


(B)

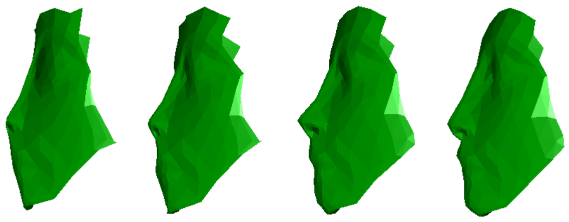


(C)

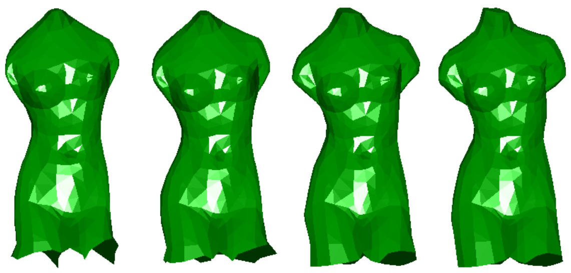
Figure 6.1: Spectral decomposition by using the geometrical Laplacian. Frequencies (from left to right): (A)0,2,5,8; (B)1,2,4,5; (C)1,2,3,4.



(A)

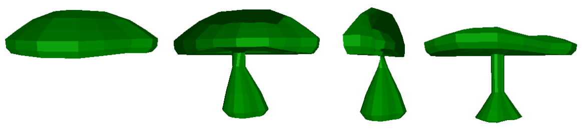


(B)

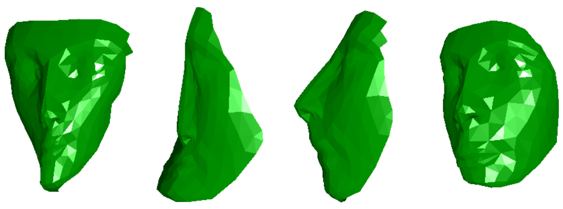


(C)

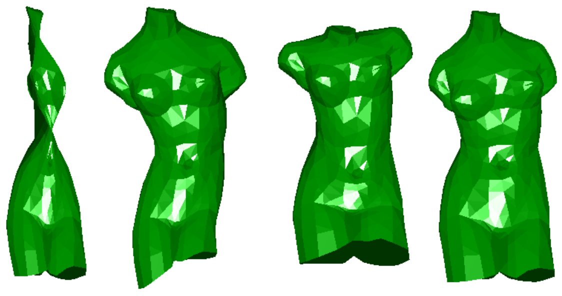
Figure 6.2: Smoothing by using the geometrical Laplacian. Frequencies (from left to right): (A)40,60,99,133; (B)22,38,61,102; (C)25,53,107,327.



(A)



(B)



(C)

FIGURE 6.3: Frequency removal by using the geometrical Laplacian. Frequencies (from left to right): (A)1,2,4,5; (B)3,4,5,8; (C)3,4,7,9.

6.3 Frequency Removal

Frequency Removal corresponds to a *Band Reject Filter* and it can be observed what happens to the mesh once a certain frequency is removed. For instance in figure 6.3 some of the fundamental frequencies are removed and several information of the original mesh is lost.

6.4 Mesh Editing

In this case the *Mesh Enlargement* and *Compression* presented here seems to be a really good tool with a huge set of applications.

The obtained meshes fit a good solution for the desired constrains and visually the meshes do not lose too much information. Some results of the obtained meshes enlarged along the y axis can be seen in figure 6.4.

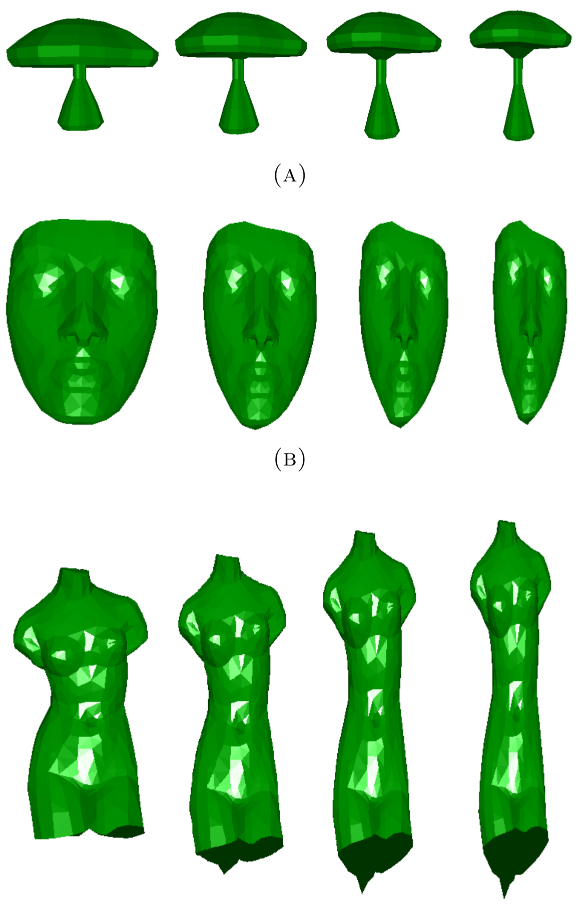


Figure 6.4: Mesh editing by using the geometrical Laplacian. Frequencies (from left to right): (A)1,2,3,4; (B)1,2,3,4; (C)1,2,3,4.

Chapter 7

Conclusion and Future Works

7.1 Conclusion

The project allowed us to work with the Laplacian Mesh Processing, a really powerful framework in the field of computer graphics with many interesting applications, specially by using the Laplacian operator with its many properties.

In this paper we could see the two main applications, Mesh Smoothing and Mesh Editing.

The Mesh Smoothing can be used for smoothing meshes acquired with really accurate 3D sensors in order to compress the number of vertices without losing visual quality, and putting those 3D models in movies or video games.

The Mesh Editing is a really powerful technique for animation, making much easier for artists to create a more realistic animation.

Just by those two applications we can confirm how powerful this technique is.

Besides that, by executing this project we could put in practice software engineering techniques in order to develop a real software. It is evident how a well-written code and a clear implementation is important in order to provide a great quality software for the user.

It was a great experience in the sense we could do a research and software development, preparing us for new researches and thesis.

7.2 Future Works

During the development of this projects we had many ideas to implement that could not be implemented on time.

Our first idea was to provide a mouse interaction in the mesh editing part, that we could select the anchors to be fixed and others to move with mouse interaction to create a new mesh.

We were also planing to implement the rendering by using shaders in GLSL, so that we could take more advantage of the graphics card making it faster then the approach used.

Another work would be to implement the Laplacian matrix in parallel by using GLSL, CUDA or OpenCL, since many of those computations can be done in parallel, then we could create a faster implementation of the Laplacian.

Those are basically some ideas for future works, in order to improve this project and continue researching the possible applications the Laplacian mesh processing can provide.

Bibliography

[1] O. Sorkine. Spectral mesh processing. EUROGRAPHICS '05, 2005. ISBN 3-905673-13-4.

[2] Gabriel Peyré. Data processing using manifold methods, 2007. URL https://www.ceremade.dauphine.fr/~peyre/teaching/manifold-sci/.

[3] O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rössl, and H.-P. Seidel. Laplacian surface editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '04, pages 175–184, New York, NY, USA, 2004. ACM. ISBN 3-905673-13-4. doi: 10.1145/1057432.1057456. URL http://doi.acm.org/10.1145/1057432.1057456.

[4] Douglas E. Meyer. Use of seismic attributes in 3-D geovolume interpretation. *Geophysics*, 20, 2001. doi: 10.1190/1.1486768.

[5] Yong M. Ro1 and Zang-Hee Cho1. Susceptibility magnetic resonance imaging using spectral decomposition. Magnetic Resonance in Medicine, 1995.

[6] Gabriel Taubin. A signal processing approach to fair surface design. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 351–358, New York, NY, USA, 1995. ACM. ISBN 0-89791-701-4. doi: 10.1145/218380.218473. URL http://doi.acm.org/10.1145/218380.218473.

[7] Zachi Karni and Craig Gotsman. Spectral compression of mesh geometry. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 279–286. ACM Press/Addison-Wesley Publishing Co., 2000.

[8] Ulrich Pinkall, Strasse Des Juni, and Konrad Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental Mathematics*, 2:15–36, 1993.

[9] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009. ISBN 0321552628, 9780321552624.

[10] Mark J. Kilgard. The opengl utility toolkit (glut) programming interface, 1996.