ROBUST PRINCIPAL AXES DETERMINATION FOR POINT BASED SHAPES USING LEAST MEDIAN OF SQUARES

Jeremie DERAY

Sai Krishna PATHI

Prabhu Kumar CHALIKONDA

University of Burgundy
2013

ABSTRACT

In order to represent the 3D shape models there are many techniques, among them principal axes determination is the best one. Principal component analysis (PCA) is the most widely used technique for determining the principal axes of 3D shape representation by a set of points, possibly with noise. But there are some problems in using PCA, mainly it is sensitive to outliers. In order to overcome this, least median squares (LMS) is used, for outlier detection but it holds drawback due to lack of robustness. One of the advantages by using LMS algorithm is, without any extra segmentation procedure, disregard the outliers and distinguish the shapes as major and minor region during the principal axes determination. By using this method an outlier free major region of the shape is extracted, which ignores the effect of the other minor regions regarded as the outliers of the shape. The LMS algorithm is simple and effective, that gives good result for point-based shapes. Start from a small outlier free subset chosen as major region, where an Octree based approximation and sampling is used for good approximation and little computation time. By adding the samples, treating the points on minor region as outliers, can define the principal axes of the shape as one of the major region.

Acknowledgements

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals. We would like to extend our sincere thanks to all of them.

We would like to express our gratitude towards our professor Fougerolle Yohan who has been supportive and pushed us to complete the project and be on time.

We would also like to thank our special instructor Djamila Aouada.

At last, our team and classmates.

Table of Contents

Table of figures

Chapter 1

Introduction

1.1 Principal Component Analysis

The most common technique for determining the principal axes of 3D shapes is PCA. PCA is the technique used to reduce the multi dimensionality data set to lower dimensional data set for analysis. PCA analyses an object into 3 principal axes or Eigen vectors, and according to their Eigen values these vectors are mapped to three axes. PCA is good to work if the samples are uniformly distributed around their mean value but if some samples are outliers then principal axes changes. This is because of large magnitudes and large distances between the outliers and mean value.

1.2 Least Median of Squares

Classical least square regression consists of minimizing the sum of the squared residuals. There are many types of robust versions in least square. We are using least median squares. In this, the estimator provides smallest value for the median square residual computed for the entire data set. It is probably impossible to write down a straightforward formula for the LMS estimator. It must be solved by a search in the space of possible estimates generated from the data. Since this space is too large, only a randomly chosen subset of data can be analyzed. This method is very robust regarding false matches as well as outliers due to bad localization. The resulting estimator will resist the effect of 50% of contamination in the data.

1.3 Octree

The construction of a 3D object model from a set of images taken from viewpoint is an important problem in computer vision. One of the simplest ways to do this is to construct bounding volume (box) for the object. To efficiently represent this volume, we use hierarchical data structures; they are based on the principal of recursive decomposition (similar to divide and conquer methods). One such data structure is called Octree, which represents the object. Octree based approximation is used for the fast computation. Basically, Octree is the tree in which each node is divided into eight portions (children).Each child may be either another internal node or a leaf. For the partition of the 3 dimensional space recursively we subdivide into eight octants. In this, the model of the particular cell is divided into 8 sub cells. If we have weight (points) inside the cell then we will keep the weight if there is no weight then we will ignore that particular cell. By doing this we will get the approximate shape of the 3d model. In this

we have to consider the depth. If we divide the sub node again into 8 sub cells is at first depth. By doing this iteratively until it reaches ‘d’ depth, it will give us the accurate shape of the body. This depth is also considered for the accuracy of the model.

Chapter 2

Methodology

2.1 Second Algorithm

Step 1: Take three mesh’s C_N, Q, C_{rem} where C_N is the original mesh and the 'Q' and C_{rem} are empty meshes at the beginning.
Step 2: Give C_N as the input and construct an Octree of depth ‘d’ for C_N. The octree can also be constructed in the preprocessing step.
Step 3: Run a loop for ‘T’ iterations and randomly select 'K-Points’, which is a subset, C_{temp}.
Step 4: By using PCA, compute the first principal axis 'e1' and the origin 'o' for C_{temp}. Compute the residuals of 'C_N' as the distance between the feature point of each boundary cell and its orthogonal projection point onto the line through 'o' in the direction of 'e1'.
Step 5: By sorting these residuals select the median i.e. r_{half}.
Step 6: Check r_{half} is less than minimal residual r_{min} if its true then step 7 otherwise step 3.
Step 7: Change the minimal residual 'r_{min}' to ‘ r_{half}' and assign 'Q' with 'C_{temp}' which is 'k-points’.

In octree, Boolean operations are implemented easily. In this project we construct quadtree for C_N (Vertices) and divide them into two categories i.e. one containing the points of $C_N$and empty cell with no points. For each boundary cell, compute its gravity centre. This is taken as the featured point that characterizes all the vertices inside that particular cell.

For T-iteration (5000), k-points are selected randomly from the original set C_N and their principal axes are determined using PCA. The largest Eigen vector is chosen as first principal axis and the residual distance from these points to their orthogonal projection onto the first axis is computed. The problem lies, if k-points are chosen randomly using pseudo random generator. Due to this, there will be a chance in clustering of points (same points in the boundary cells). To avoid this, while doing quadtree, we calculate density of each cell then the cumulative density function of all the boundary cells. Here, density is nothing but counting the number of points inside the boundary cell. Boundary cells are selected proportionally to their densities. By this, the k random points are generated between 0 and the last value of the cumulative density function and also perform the binary search on the array of the cumulative densities.

This quadtree based sampling method gives statistically k-points which belong to the main region.

The basic idea of PCA is to seek a projection in that best represents the data in least square sense. In this PCA assumes origin is at simple mean point 'o'. By using PCA, find the vector 'e' through origin such that the sum of the square distance between point (belongs to the mesh) and its corresponding orthogonal projection point p* onto e is as small as possible. The Eigen vector corresponding to the largest Eigen values gives the first principal axis.

2.2 First Algorithm

Step 1: Compute PCA of 'Q'.

Step 2: Compute residual distances from C_{rem} to e1

Step 3: Get the m points of C_{rem} whose residual distance to e1 is less than r_{max} and add them to 'Q'

Step 4: Redo the three first steps until the points of ' C_{rem}' whose residual distance to 'e1' is less than 'r_{max}' are less than m.

The outline of an algorithm for robust principal axes determination, called Robust Principal Axes, is given in Algorithm 1. First, provide the given point set C_N with 'N' points as input and get the output as the origin 'o', and the three principal axes 'e1, e2 and e3' of the reference frame. This is achieved through an iterative procedure with the aid of a variable 'Q' that is a working subset of 'C_N'. Run a loop, in this, add lowest residuals to 'Q' and gradually increase 'Q' by 'm' points. The increased 'Q' is considered as major region in the forward search. If the residuals of the remaining points i.e. C_N-Q are more than a threshold r_{max} then the procedure is terminated. Now, 'Q' is the major region and 'C_{rem}' is the minor regions.

Chapter 3

Implementation

The implementation presented here is based on the project 'Mesh2012' [2]. The update is divided into four parts (Figure 1). The first part is used to load a mesh from a 3D file. Second and third parts are the implementation of 'RPCA'. And finally the fourth part is the result displaying.

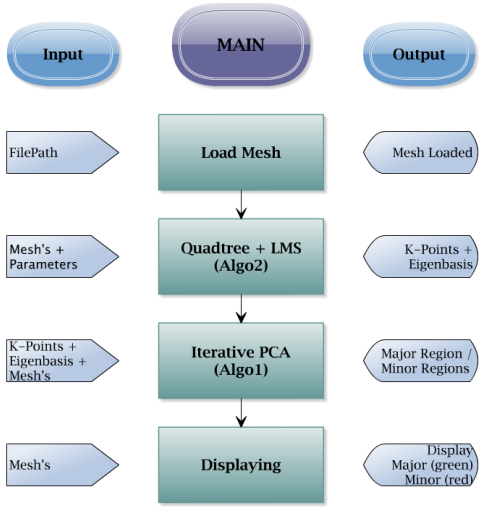


Figure 1

3.1 Load Mesh:

This part is divided into two parts (Figure 2). Firstly a mesh is constructed and filled with vertices data contained into a 3D file and then its other members are set up.

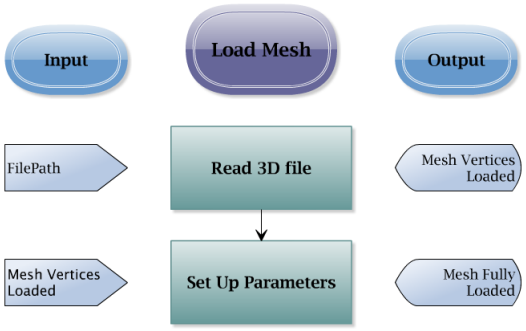


Figure 2

Original project contained a function to read 3D file of extension ' .VRML ' named 'ReadIv()' called by the class mesh member 'ReadFile()'. To work with 'SHREC2012' meshes, a function has been implemented to read ' .OFF ' 3D file.

' ReadOff() ' function :

This function, mainly based on ' ReadFile() ', is also called by ' ReadFile() ' and takes as entry parameters a mesh by reference, a file name (path file) and a boolean. It also returns a boolean as success or not of the reading.

```
// Read SHREC 2012 OFF 1.0 file (update of 'ReadIv')
bool ReadOff( Mesh& mesh, const string& file_name, bool disp ) {
```

Before going to explain that function, we have to give few words about 'SHREC2012' '.OFF ' files. These files are constructed as following:

```
OFF
VerticesNumber FacesNumber EdgesNumber
Vertex.X Vertex.Y Vertex.Z
...      ...      ...
...      ...      ...
...      ...      ...
FaceVerticesNumber FacesVertex.1 FacesVertex.2 FacesVertex.2
...                ...           ...           ...
...                ...           ...           ...
...                ...           ...           ...
EdgeVertex.1 EdgeVertex.2
...          ...
...          ...
...          ...


exemple : 'SHREC2012' -> D00025

OFF
11823 23007 0
0.099255 0.506081 -0.022015
0.097009 0.507335 -0.021039
0.096603 0.507512 -0.021947
0.092952 0.509216 -0.019728
...      ...      ...
0.729073 -0.238144 0.09275
3 11425 11424 11423
3 11424 11427 11426
3 11427 11425 11428
3 11425 11427 11424
...     ...   ...
3 1114 1121 6620
```

As seen in this example, first line indicates the extension file. Second one indicates numbers of vertices, faces and edges. Then comes vertices coordinates, followed by faces. For a face, its number of vertices is firstly indicated then comes point indexes, which compose that face. Finally edges should come, but 'SHREC2012' do not use edges.

Based on these information's, function ' ReadOff() ' first verifies the file extension. If something else than ' OFF ' is read, the function stops there and returns 'false ', otherwise it goes to the next line to get the number of vertices, faces and edges. To do so, that line is stored into a string then read character by character within 'for' loop. Until a space is read, characters are stored into another string. Once a space is

read, the second string is converted into integer and stored into its corresponding variable.

The same principle is used to read vertices and faces except that for faces, the first value (face vertices) is ignored.

The second part of 'Load Mesh' is simply the set up of two mesh members ; member 'isProjected ', a boolean set to ‘false’ if vertices are the original ones and set to ‘true’ if vertices are projected onto an Eigenspace. Second member is an array of booleans, same size as vertices array and whose indexes correspond to vertices indexes. If a vertex belong to the major region then its corresponding boolean is set to ' true ', otherwise it is set to ' false '.

These both parts are called from the main program through a function called 'LoadMesh()'. It takes as entry the same variables as ' ReadOff() '.

```
bool LoadMesh(Mesh &myMesh, string &myMeshFile){
```

3.2 Quadtree + LMS:

This part is actually the algorithm2 [1]. Its aim is to select a first set of k-points which belong to the major region. It is divided into five parts and one conditional test.

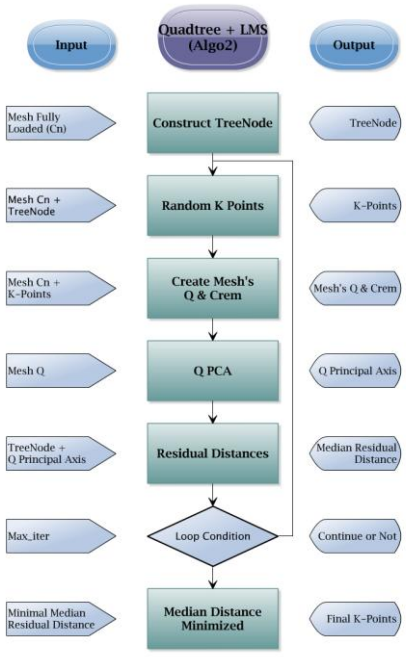


Figure 3

Construct TreeNode:

For this project, we tried to implement our own tree node. Instead of an octree, we did a quadtree, and to do so a class, 'class Node ', has been created. This class has ten members and ten functions defined as following:

```
class Node{

    public :

    //Constructor
    Node();
    Node(Vector3d ,Vector3d , int);

    //Destructor
    ~Node();

    //Node points
    vector<Vector3d> pts;

    //Pointers to subNodes
    Node *subNode[8];

    //List of points which belong to Node
    multimap<int, int> indCenter2pts;

    //Node gravity center
    Vector3d center;

    //Leafs cdf
    map<int,int> nodeCdf;

    //Node leaf
    bool leaf;

    //Node first root
    bool root;
```

Where 'pts' is the node vertices

'subNode' an array of pointers to the sub nodes

'indCenter2pts' a multimap containing the node indexes as key and their associated mesh vertices

'center' the node gravity center

'nodeCdf' a map containing the Cdf as key and the node indexes as value

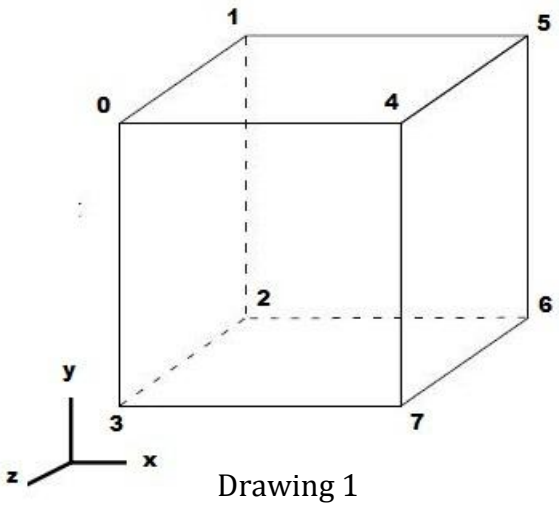
'Leaf' a boolean which indicates if the node is the last level of the tree node

'root' a boolean which indicates if a node is the original root of the tree node

```
//Compute surrounding box diagonal
vector<Vector3d> TreeNodeDim(const Mesh&);

//Initialise octree
vector<Vector3d> initNode(const Mesh& ,int);

//Get number of leafs
int size();

//Draw leafs
void Draw(bool);

//Display leafs center value
void Disp();

//Draw boundaries nodes
void DrawBound(bool,vector<Vector3d>& ,multimap<int, int>&);

//Reset treenode
void ClearAll();

//Exctract leafs centers
void GetCenters(vector<Vector3d>&);

//Compute distances from each point to each node center
//and store node index and points indexes that it contains
bool SetWeight(const Mesh&, vector<Vector3d>&);

//Compute the cumulative density function
bool SetCdf();
```

At the beginning, a quadtree is a cube surrounding the mesh, so it needs some features from the mesh to construct our cube around it. To do so, firstly the mesh mean data is computed, then the distances from that mean to all vertices are computed and finally the greater is kept. Starting from the mean we add that distance to each mean coordinate, that gave us the point 4 (drawing 1). From the mean subtract the distance to each coordinate and that gave point 2, they represent the cube diagonal. This operation is done through the function 'TreeNodeDim()'

With that diagonal, construct the cube with the following table:



Drawing 1

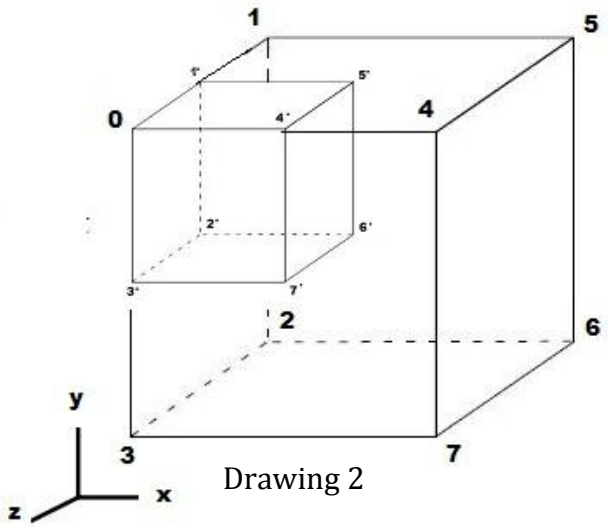
Vertex	X coordin	Y coordi	Z coordi
0	min	max	max
1	min	max	min
2	min	min	min
3	min	min	max
4	max	max	max
5	max	max	min
6	max	min	min
7	max	min	max

```
Dim = TreeNodeDim(myMesh);

Vector3d xyzMax = Dim[0] + 0.001*Dim[0];
Vector3d xyzMin = Dim[1] + 0.001*Dim[1];

Vector3d pts0(xyzMin[0],xyzMax[1],xyzMax[2]);
pts.push_back(pts0);
pts0 << xyzMin[0],xyzMax[1],xyzMin[2];
pts.push_back(pts0);
pts0 <<xyzMin[0],xyzMin[1],xyzMin[2];
pts.push_back(pts0);
pts0 <<xyzMin[0],xyzMin[1],xyzMax[2];
pts.push_back(pts0);
pts0 <<xyzMax[0],xyzMax[1],xyzMax[2];
pts.push_back(pts0);
pts0 <<xyzMax[0],xyzMax[1],xyzMin[2];
pts.push_back(pts0);
pts0 <<xyzMax[0],xyzMin[1],xyzMin[2];
pts.push_back(pts0);
pts0 <<xyzMax[0],xyzMin[1],xyzMax[2];
pts.push_back(pts0);
```

This table is the architecture of the quadtree, each cell follow it for different values of vertices 2 and 4.
Once the main cube is defined, its feature point is defined as its gravity center (its diagonal bisectrix) then a node constructor is called recursively to create the sub nodes, still based on the diagonal. So first the sub diagonal must be defined for each sub nodes. For instance, the sub node '0' share the vertex '0' with the main cube (drawing 2), its diagonal is defined as follows:

```
if (depth>0){

    leaf = false;

    xyzMax << ((pts[0][0]+pts[4][0])/2),pts[0][1],pts[0][2];
    xyzMin << pts[0][0],((pts[0][1]+pts[3][1])/2),((pts[0][2]+pts[1][2])/2);

    subNode[0] = new Node(xyzMin,xyzMax,depth-1);
```

The constructor is called for the new diagonal, with a depth value decreased and a pointer to that new node.



Drawing 2

Since the constructor is called recursively, if the depth didn't reach zero, this sub node will be sub divided and its boolean 'leaf' set to 'false'.

Once the quadtree is built, the function 'GetCenters()' is called is order to store within its entry variable, the center coordinates of all the leafs of the quadtree. That function, called recursively, follows the same path as the construction step.

```
void Node::GetCenters(vector<Vector3d> &myCenters){//Get leafs c

    if (leaf == true){
        myCenters.push_back(center);
    }else{
        for (int i=0;i<8;i++) subNode[i]->GetCenters(myCenters);
    }
}
```

Next step is to associate leafs with mesh vertices. If a vertex is contained into a leaf then the leaf index will be associated with the vertex index. That operation is done within the function 'SetWeight()'. To associate a vertex with its node, the distance from that point to each leaf center is computed, and for the smaller distance, the node index is associated with the vertex index within a map.

```
ptsCent.insert(make_pair((int)indCen,(int)pts));
```

The final step is to compute the cumulative density function. Through the function 'SetCdf()' previous map is used to compute the Cdf as follows :

```
int tempWeight = 0;
int indMove = 0;

map<int, int>::iterator myIt;
myIt = indCenter2pts.begin();

while (myIt!=indCenter2pts.end()){

    tempWeight += indCenter2pts.count(myIt->first);
    indMove = indCenter2pts.count(myIt->first);

    nodeCdf.insert(make_pair((int)tempWeight,(int)myIt->first));

    advance(myIt,indMove);
```

'tempWeight' represents the sum of leaf weights where that weight is the number of vertex indexes associated to the leaf. So for each key (node index), count how many values are associated (point indexes) and sum the result with 'tempWeight' then go to the next key which is different to the one analyzed until the procedure reachs the end of the map. The output is a new map which has for key the Cdf values and for values the node indexes.

From the tree node dimension definition, to the Cdf computation, all the steps are called within a unique function named 'initTreeNode()'. It takes as input the mesh and returns a vector containing leaf centers.

```
vector<Vector3d> Node::initNode(const Mesh &myMesh , int depth){
```

Two other tree node functions:

Size() :

This iterative function return simply the number of leafs.

Draw() :

This iterative function, which follows the quadtree construction path draw the leaf vertices.

Random K-points:

The following is done K-times. In this function, firstly an integer is randomly selected between one and the number of vertices. This number is an index for the Cdf multimap of the tree node. Using that number an operator is set to the first cell of the Cdf whose key is equal or superior to that index.

```
randomInd = rand() %myMesh.VertexNumber() +1; //
myItCdf = myNode.nodeCdf.lower_bound(randomInd);
```

```
nodeInd = myItCdf->second; //Get node index (ke
nbOfPts = myNode.indCenter2pts.count(nodeInd);

myItNode = myNode.indCenter2pts.find(nodeInd);
advance(myItNode,rand() %nbOfPts); //Select ra
myVec.push_back(myItNode->second); //Store that
```

Once that cell is found, the number of point indexes associated to its key is used to select randomly a new number between zero and that number of vertices in order to select randomly a point index using this new random number. Finally, these K random points (indexes) are returned.

Create Meshes Q and C_{rem} :

Since we have our K-points, now create the subset Q which will become the major region and the remaining subset 'C_{rem}' which will be the minor region through the function 'RemainingPoints()'. For this, these K points are copied from mesh 'C_{rem}' (which is, initially, the complete mesh) to mesh Q and then delete from 'C_{rem}'. To keep a correspondence between point indexes stored into the entry vector and point indexes into the mesh during the deleting step, the vector is sorted in descending order. At the end of this step, Q is of size K-points and 'C_{rem}' is of size total of vertices minus K (C_N- k).

Q PCA:

Next step is to compute the Principal Component Analysis of Q. To do so, use the OpenCV library. OpenCV is the reference library for computer vision programming. A brief installation tutorial is provided within appendix (3). That library provides a class PCA which takes as input the mesh member ' vertices ', a second array which contains data mean values, an option to indicate that it should consider data as rows and finally the number of Eigenvectors it should compute. Provide an empty array as data mean values, doing that, it will compute itself these mean values.

```
PCA myPca(myMat,myMean,CV_PCA_DATA_AS_ROW,3);
```

Since that class doesn't work with ' vector<> ' containers, data have to be copied into an OpenCV matrix to process them.

```
for (int i=0;i<NBrows;i++) //Copy c
{
    myMat.row(i)(0)=vertices[i][0];
    myMat.row(i)(1)=vertices[i][1];
    myMat.row(i)(2)=vertices[i][2];
}
```

Once PCA has been computed, the object ' myPCA ' provides three members and a function. The three members are EigenValues, EigenVectors, and data Mean. The function is to project data onto the EigenSpace 'project()'. Since these PCA members are OpenCV containers, copy them into ' Eigen' containers before to store them in their corresponding mesh members.

```
//Eigen vectors
vector<Vector3d> eigenVec;

//Eigen values
Vector3d eigenVal;
```

```
//Mean values from PCA
Vector3d eigenMean;

// Vertex array into eigen subspace
vector<Vector3d> verticesProjected;
```

The whole PCA computation has been implemented into a mesh function called 'Meshpca()'.

Residual Distances:

Once the main EigenVectors of Q are known, compute the residual distances between leaf node centers and that vector. This is done by using an overloaded function 'Distance2Eigen1()' which takes as input the quadtree, a vector containing leaf centers and a structure containing the main EigenVector and the mesh mean values.

```
multimap<double, int> Distance2Eigen1(const Node& myNode,const vector<Vector3d> &centers, const GenEigen& genEig){
```

Results of this computation are kept into a multimap container whose key is the distance and its associated value the leaf nodes index (regarding the vector of centers).

```
dist.insert(make_pair((double)resDist, (int)nodeInd));
```

Since multimaps are automatically sorted regarding their keys, the leaf nodes, which have the smallest distance to the main EigenVector, are on the top of the multimap.

Once multimap is constructed, it is simple to set an operator to its median key (median value) and to extract it in order to minimize the median residual distance.

```
medDist = dist2eig.size()/2;

myIt = dist2eig.begin();

advance(myIt, medDist);

rHalf = myIt->first;
```

Loop Condition:

These five previous steps are executed within a ' while ' loop in order to ensure that statistically we minimized the median residual distance.

This whole algorithm 2 returns, mesh Q with its K-points for which the median residual distance is minimized, and mesh 'C_{rem}' which is the original mesh minus Q's K-points. Algorithm 2 has been completely implemented within a function called 'First_Part()'.

```
void First_Part(const Mesh &Original, Mesh &Crem, Mesh &Q, Node &myNode, const int max_iter, const int k, const int depth){
```

3.3 Iterative PCA:

Aim of this part, which is actually algorithm1 [1], is to find the major region, based on previous K-points. It is divided into four parts and one conditional test (Figure4).

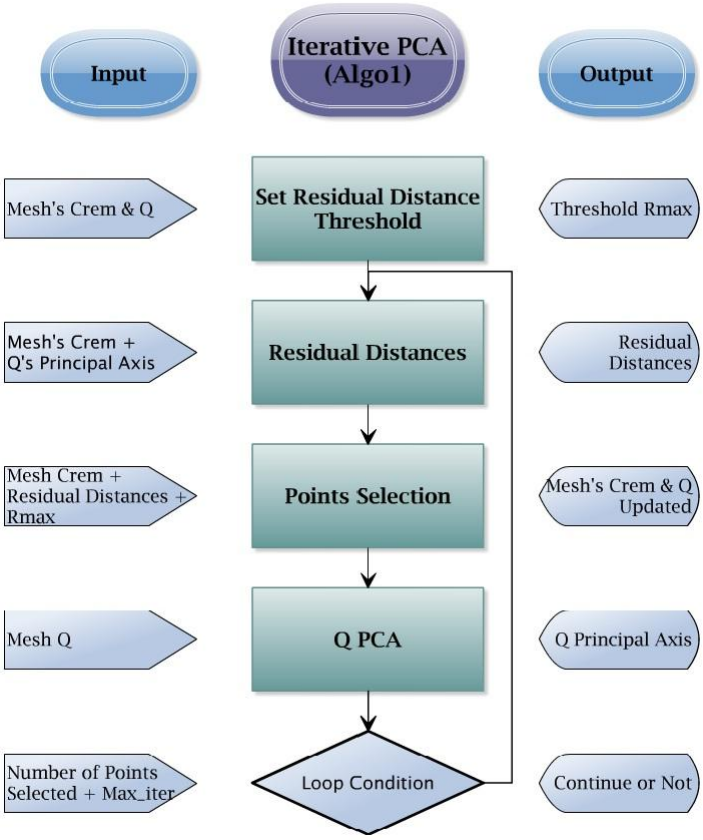


Figure 4

Set Residual Distance Threshold:

To know if the vertex belongs to the main region, set a threshold 'r_{Max}' based on the residual distance. This is done within the function 'SetRmax()' which takes as entry the mesh 'Q'. Within this function, the residual distance is computed for all 'Q' vertices using the overloaded function 'Distance2Eigen1()' which now takes as input the mesh 'Q' instead of quadtree and the structure containing the main Eigenvector and data

mean. Then the greater distance is returned. A coefficient ' alpha ' is applied to that distance and finally gives us 'r_{Max}'.

Points Selection:

To select 'm' points which belong to the main body, the residual distance between each point of 'C_{rem}' and the main Eigenvector of 'Q' is computed, once again using 'Distance2Eigen1()'. Then keep 'm' closest vertices through the function 'GetClosesrPOint()' which takes as entry the multimap containing the distances, 'm' and 'r_{Max}'. This function simply assures that the 'm' first distances (key) of the multimap are inferior to the threshold 'r_{Max}' and returns the associated point indexes sorted in a descending order.

```
vector<int> GetClosestPoint(multimap<double, int> myMap, int m, double rmax){

    vector<int> dum;

    multimap<double, int>::iterator myIt = myMap.begin();

    for (int i=0;i<m;i++){ //For m closest points

        if (myIt->first < rmax){            //If point distance is less than r
            dum.push_back(myIt->second);    //then get its index
        }else{
            sort(dum.rbegin(),dum.rend());  //Else, sort descending points ind
            return dum;
        }
        myIt++; //Go to next point
    }
    sort(dum.rbegin(),dum.rend()); //sort descending points indexes
    return dum;
}
```

Point indexes are next used with the function 'RemainingPoints()' to extract them from 'C_{rem}' and put them into 'Q'.

Q PCA:

Since the work set 'Q' has changed a bit, its main Eigenvector is computed again still using the mesh member 'Meshpca ()'.

Loop Condition:

The previous four steps are executed within a 'while' loop whose conditions are a certain amount of computation 'MAX_iter' and the number of closest points extracted from 'C_{rem}' to 'Q'. If that number of points is less than 'm', then the complete main region has been found and the computation can stop.

3.4 Displaying:

This part, divided into three subparts, is mainly based on pre-existing functions such as the mesh function 'Draw()' which has been updated to display major region and minor regions in two different colors (Figure 5).

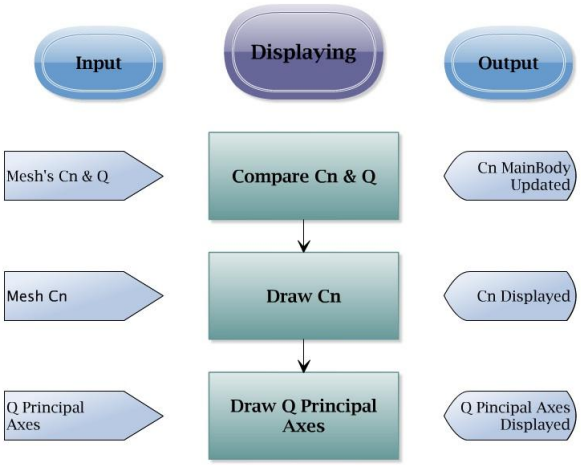


Figure 5

Compare C_N and Q:

Once the major region is obtained, the first step is to compare the original mesh vertices to the 'Q' vertices. If a vertex of the original mesh is also present into 'Q' it means that vertex belong to the major region, so its corresponding boolean 'mainBody' will be set to 'true'. To do so the function 'CompMesh()' searches each vertex of 'Q' into C_N extract the corresponding index of that vertex into C_N and set the corresponding boolean to 'true'.

```
for (int i=0;i<MainBody.VertexNumber();i++){ //For all 'MainBody' (Q) vertices

    it = find(orig.vertices.begin(), orig.vertices.end(), MainBody.vertices[i]);

    ind = it - orig.vertices.begin(); //Get its index

    orig.mainBody[ind] = true; //Set corresponding 'mainBody' bool to true
}
```

Draw C_N:

This step uses the mesh function 'Draw()' which has been updated in its option 'ONLY_VERTEX', such as it draws a vertex in green if that vertex corresponding 'mainBody' boolean is 'true' and in red otherwise.

Draw Q Principal Axes:

Finally, this function displays the three EigenVectors as three lines starting at the mean value. Main axis is drawn in red, second in green and third in blue. Using entry parameters, they can be scaled regarding their corresponding EigenValue, an integer or both.

```
void Mesh::Draw_EigenBasis(const int &i, const bool &EigenValScale)
```

During the execution of the program some information is displayed such as time to load each part, or mesh shape properties etc. That may look like this:

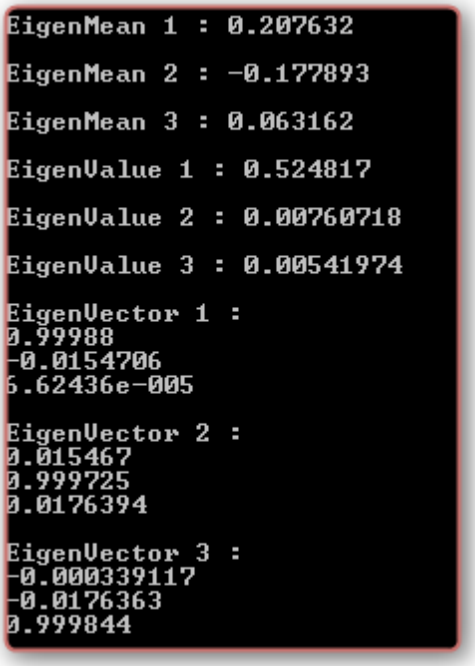
```
READ FILE
Extension = OFF
Reading OFF

// nbVertices : 11823 // nbFaces : 23007 // nbEdges : 0

OFF/SHREC2012/D00025.off has been loaded

Loading Off file time :0.057 s

Point selected index : 10629 8852 7476 3186

Loading First_Part time :18.803 s

rMax : 0.264576

Loading Second_Part time :1.024 s

Vertices displayed = 11823

Q size : 8465

Displaying loading time :0.147 s
```

Nonessential functions:

Some nonessential functions have been implemented in order to help during the programming. These functions are briefly presented here:

DispEig():

This function takes as entry a mesh and simply display within the console its data mean values then its three EigenValues and finally its three Eigenvectors. This looks as following:



Stop() :

Since for an unknown reason can't use the debugger, all the debugging has been done in a 'noob' way using the so called 'system(''pause'')'. That system function has been included into the function 'stop()' which displays a comforting sentence and an entry integer.

Chapter 4

Results

4.1 Difficulties:

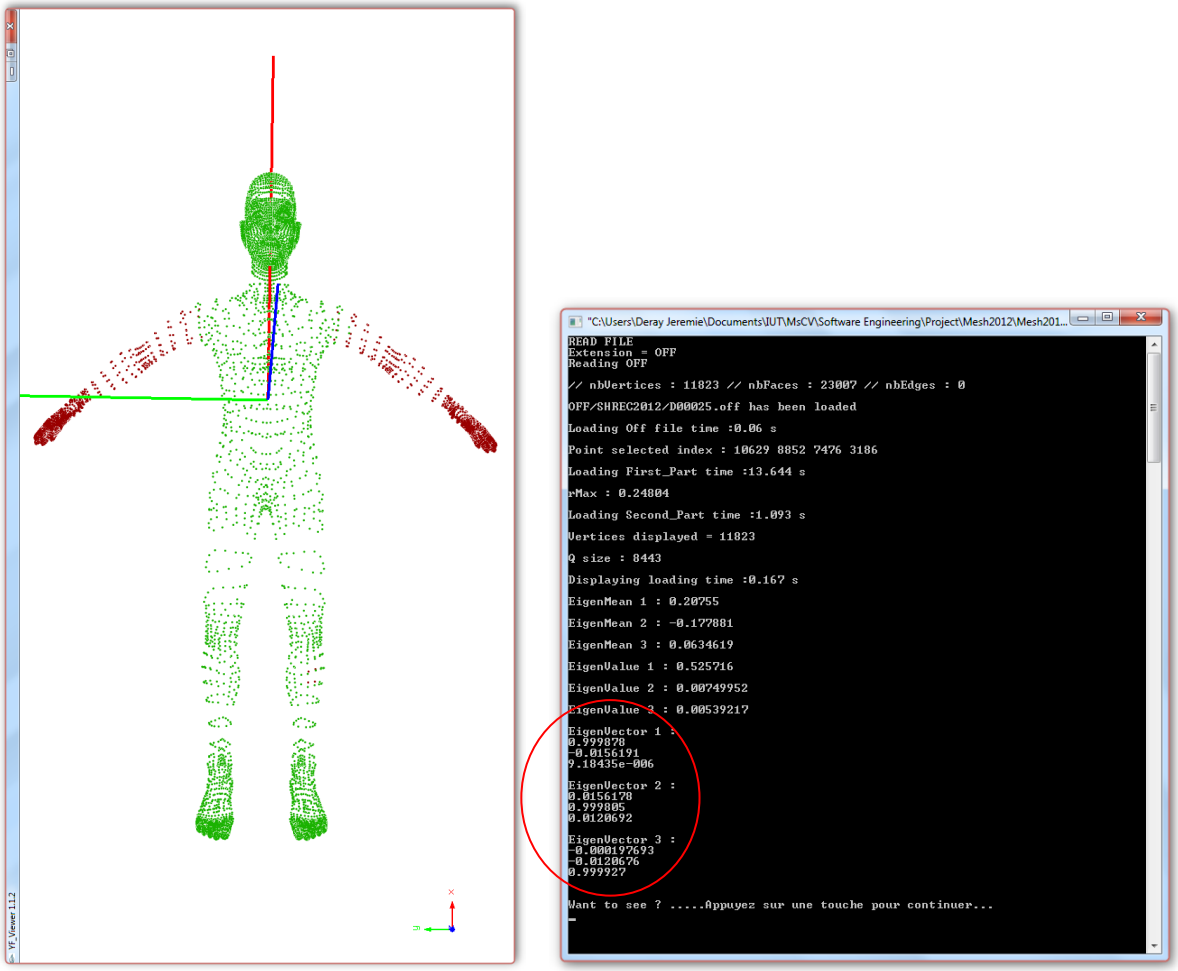
- Octree: It is complicated structure. That's why we did an experimental quadtree.

- OpenCV: OpenCV is not as easy as Eigen library to install. There are many steps involved in installing OpenCV. Sometime due to linking problem the Code Blocks was crashed. Linking of OpenCV plays major role in installing. The tutorial in annexure [3] really helped us a lot to install the OpenCV.

- PCL: PCL library for Octree does not work on Windows. But it works on Linux. We checked all the stuff clearly but we did not figure out the problem exactly.

- We were working on meshes of shrec2012 which are not ideal for the algorithm RPCA. This algorithm needs well sampled 3D shapes.

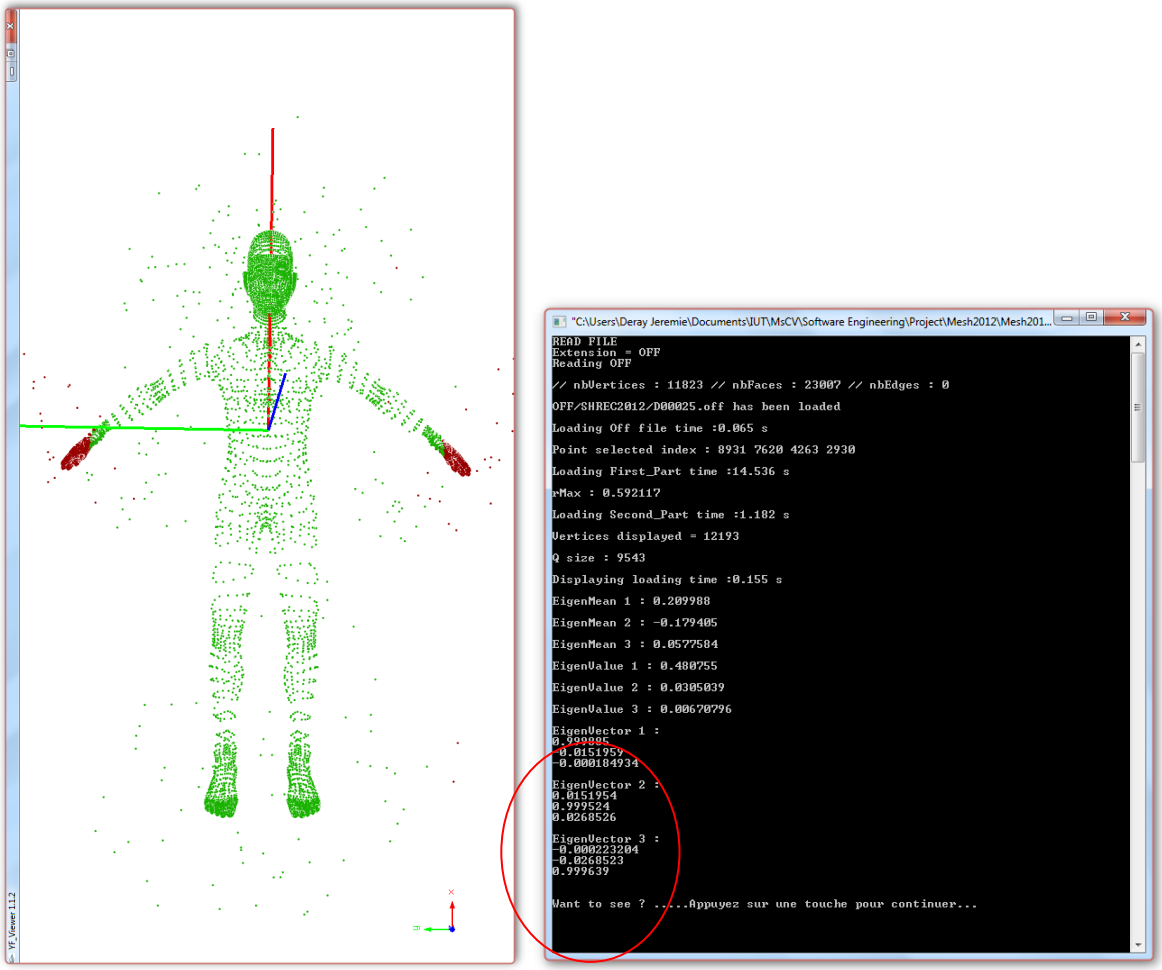
4.2 Results:



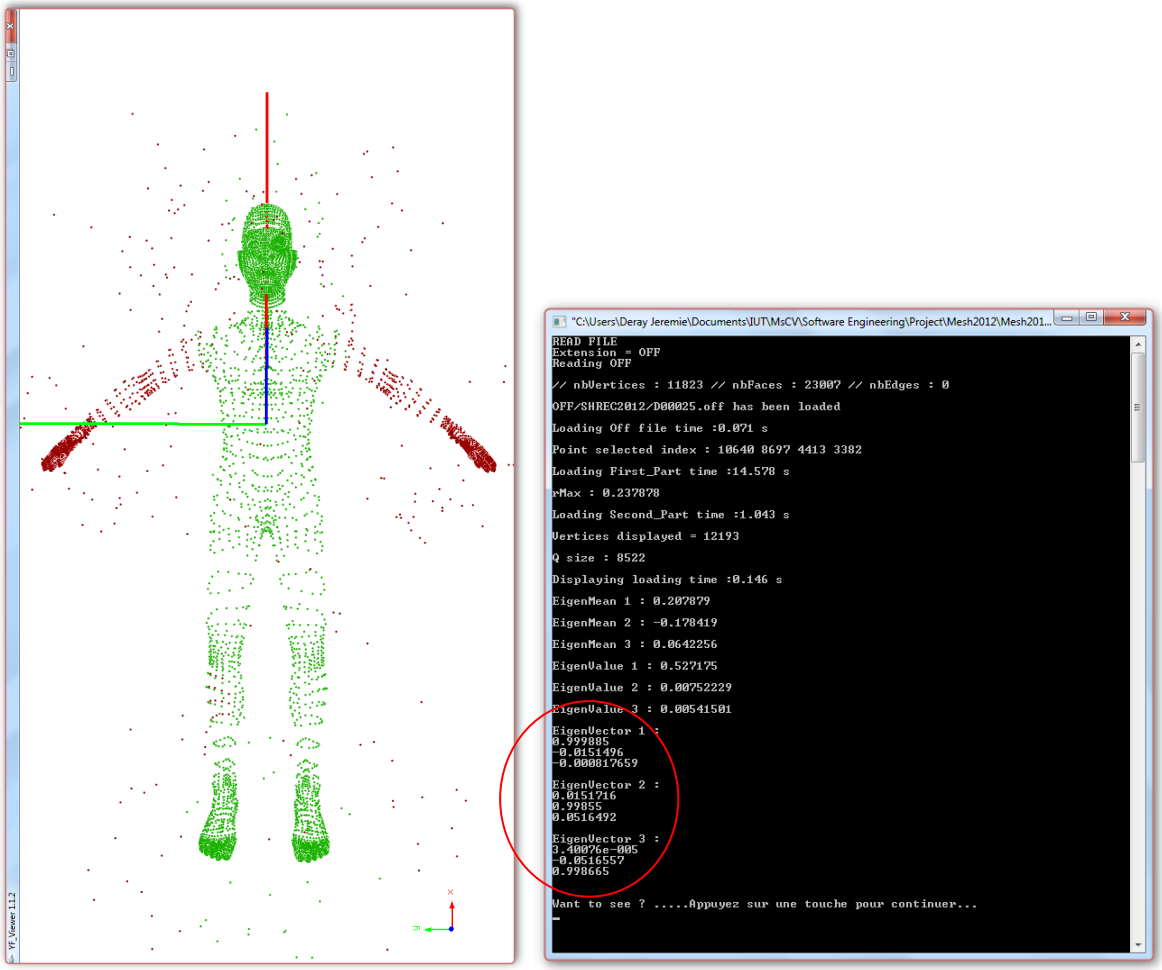
Here is presented the mesh used during this project. It is composed of 11823 vertices.

Highlighted here is the three principal axes found through our implementation with alpha value = 3.

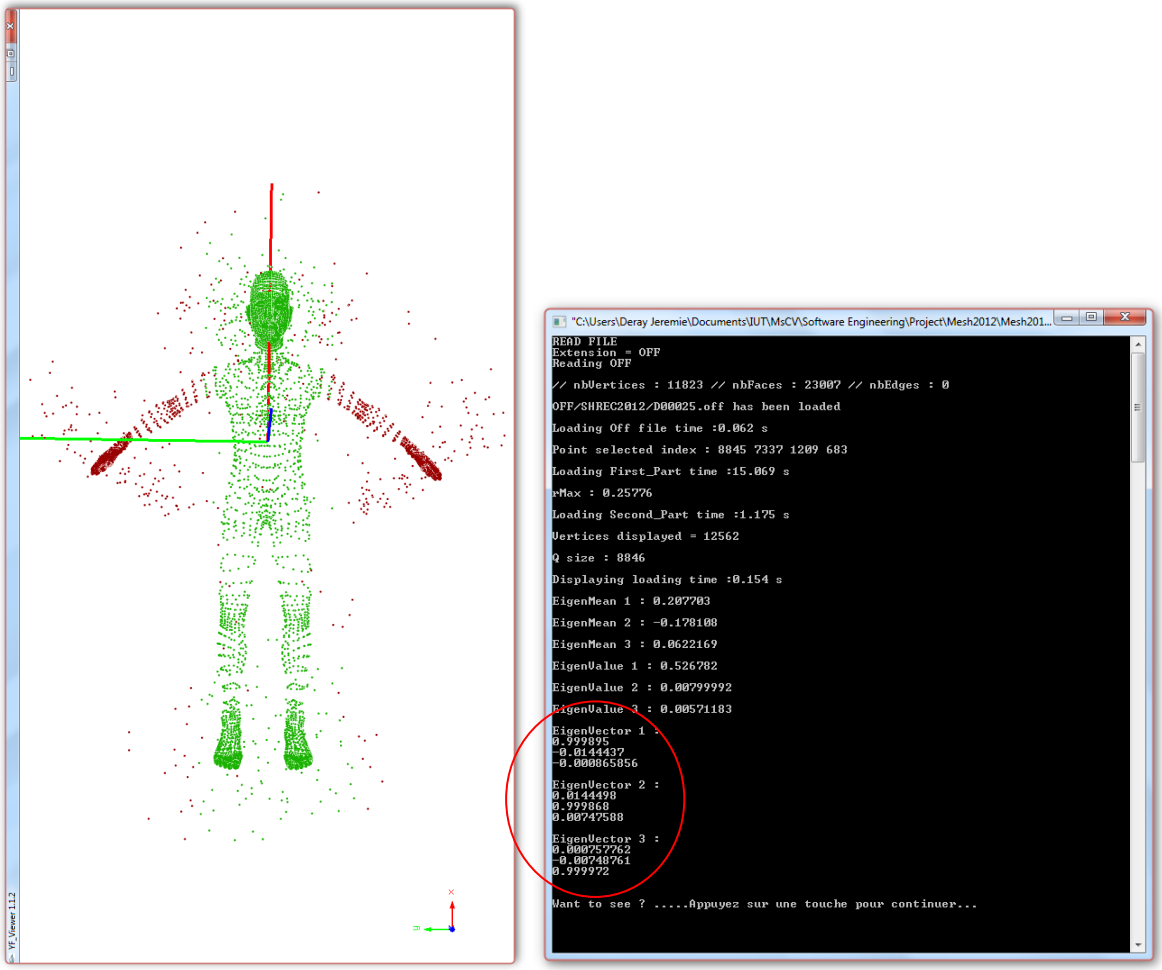
The following pictures are the same 3D shape presented here with different amount of noise. Their principal axes are also highlighted; they have been computed with a different alpha value which is 1.3.



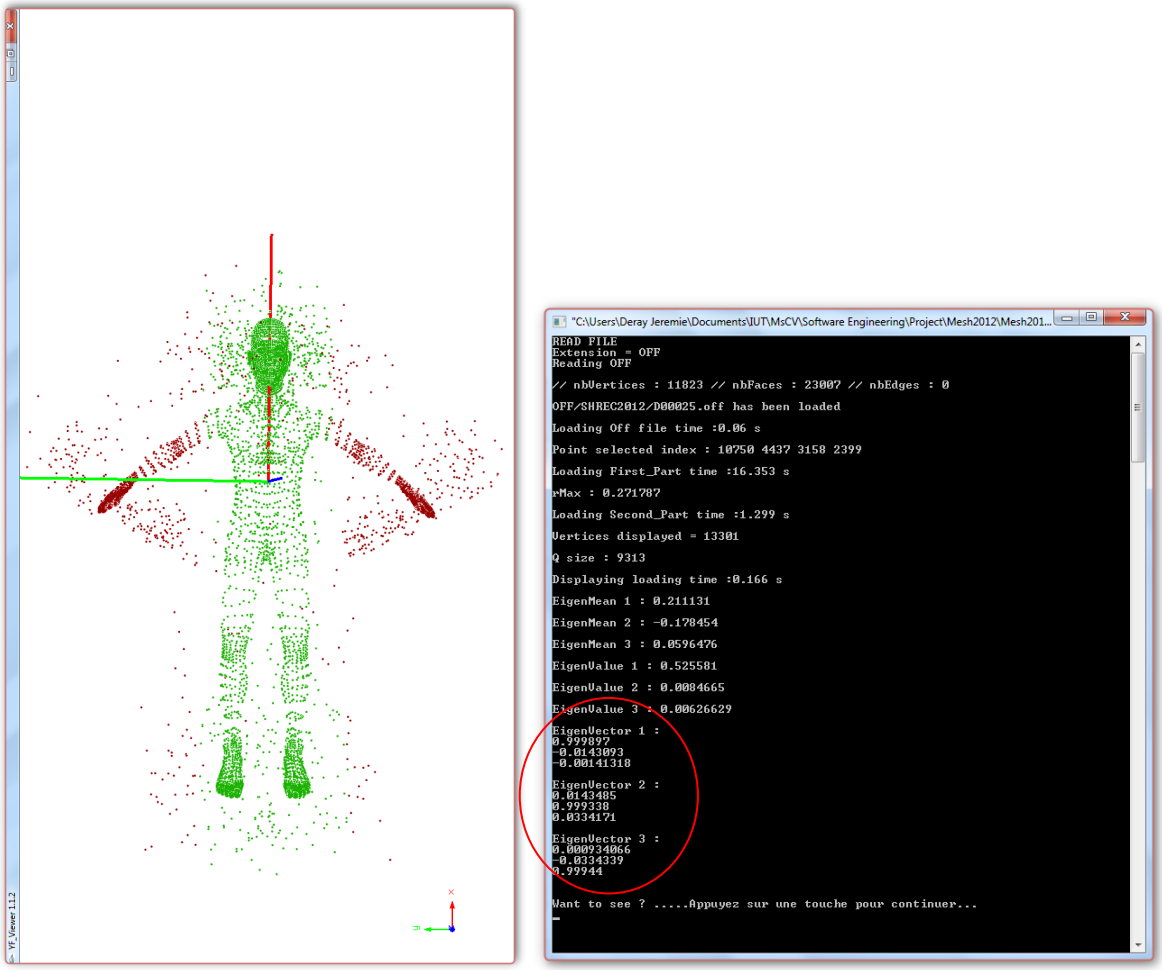
Here is the mesh used during this project with 3% of noise. Its principal axes have been computed with an alpha value = 3. As we can see some vertices which were previously considered as outliers are now within the major region. But we will see in the next pictures that by decreasing alpha from 3 to 1.3. The major region remains the same even if we increase the noise.



Here is the mesh used during this project with 6% of noise.



Here is the mesh used during this project with 12% of noise.



Here is the mesh used during this project with 24% of noise.

For an unknown reason the threshold “rt” as it has been defined (1) is too low to find a correct major region, moreover if we add noise without change in threshold. It seems coming from our selection of the first K-points. Once we are working with noise and alpha = 1.3 the increased noise do not change the results. It shows that the second part (algorithm 1) is working and strong regarding the outliers.

Chapter 5

5.1 Conclusion

In this paper, we presented an implementation of the method to compute the principal vectors of 3D shapes based on the work of yu lie and karthika ramani (1).

This work permitted us to practice what we learned during software engineering labs such as binary tree to do quad tree.

It also has been an introduction to the 3D world of point based shapes and its problematics such as the difficulty of well define 3D shapes with few features.

It has been very educational and showed us a new branch of the computer vision world.

5.2 Future work:

Investigate on the interest of extended algorithm for faces.

Investigate on methods to automatically define the algorithm parameters such as k and alpha.

Investigate on the interest of computing three major regions, one over each principal vectors whose shared vertices would be the final major region.

Annexure

1. Open CV

Installing OpenCV

This tutorial comes from the official website of the library [3] and explained how to install OpenCV under windows OS for Code:: Blocks using MinGW.

1. Install Code:: Blocks and C + + Compiler MinGW. Also install Cmake.
2. Once the software is installed, now recover the library, Simply select the latest version at the following address: http://sourceforge.net/projects/opencvlibrary/files/opencv-win/ then extract the zip file, for example at the root disk
    a. Zip file: OpenCV-2.3.0-win.zip
    b. extract file: C: \ OpenCV-2.3.0-build.
3. Run cmake (cmake-gui) and specify the source, here it is:
    a. Source File: C: \ OpenCV-2.3.0 \ OpenCV-2.3.0
    and the destination file:

    C: \ OpenCV-2.3.0-build.

Then click "Configure".
Cmake will create the new folder and then it must indicate the generator: MinGW Makefiles. Select "Specify native compilers" and click "next". Specify (for C + +):

C :/ Program Files / CodeBlocks / MinGW / bin / g + +. Exe.

Finally, click "Finish". In the configuration screen, type in "RELEASE" CMAKE_BUILD_TYPE Configuration and start again. Then click on "Generate". Finally close Cmake.

4. Start the console prompt, the destination file created by Cmake
    a. Enter: mingw32-makeAnd enter: mingw32-make install.
5. You can now open Code :: Blocks and create a new project.
    - In the menu: Project → Build Options → Linker settings → Link libraries, addLibraries, for example:C: \ OpenCV-2.3.0-build \ lib \ libopencv_calib3d220.dll.aAnd all other files ". dll.a" in this folder.
    - In the menu: Project → Build Options → Search → Compile directories, add the path:C: \ OpenCV-2.3.0-build \ include
    - In the menu: Project → Build Options → Linker → Search directories, add the path:C: \ OpenCV-2.3.0-build \ lib
6. Finally you need to tell windows the way these developments.
7. In Control Panel: System and Security → System Settings →Advanced System → Environment Variables → select PATH and click changed. Add to paths possibly already present:

C: \ OpenCV-2.3.0-build \ bin
The installation is complete, it remains only to write and run the program!
(This tutorial from the site http://opencv.willowgarage.com/wiki/MinGW)

REFERENCES

1. Robust Principal axes determination for point-based shapes using least median of squares by Yu-Shen Liu, Karthik Ramani.

2. Yohan Fougerolle project “Mesh2012”

3. http://opencv.willowgarage.com/wiki/MiniGW

4. http://videolectures.net/nipsworkshops2010_caramanis_rcf/

5. http://www.youtube.com/watch?v=oy937FcYJOk

6. http://docs.pointclouds.org/trunk/group__octree.html

7. http://www.Cplusplus.com

8. http://www.siteduzero.com