Universite de Bourgogne

Software Engineering

Semester Project Report

Implementation of Harris Operator for Interest Point Detection on 3D Meshes

Computation, 3D Rendering and GUI

Author:

Zohaib Salahuddin
Abdullah Thabit
Ahmed Gouda

Supervisor:
Yohan Fougerolle

Contents

1 Introduction

1.1 The program and its functionalities

Detecting interest points in 3D objects has become a growing field of research in the past few years, and that comes due to the increasing capabilities of capturing devices and the needs to have few descriptive points for representing these objects, such as in objects' detection and registration. The program developed here detects interest points on 3D meshes based on the article *"Harris 3D: a robust extension of the Harris operator for interest point detection on 3D meshes"* by I. Sipiren and B. Bustos [1]. However, this program differs than their implementation in a way that it doesn't relay on the CGAL library in representing the 3D meshes, moreover, it adds a Graphical User Interface (GUI) with various functionalities for easier usage and better visualization.

In a nutshell, the program reads an OFF file that represents a 3D object (only OFF files of 3 vertices per face), and displays the 3D object on the screen. Then, based on the parameters provided by the user, it detects the interest points and overlay them on the 3D object. The parameters inserted by the user control the way of calculating the neighborhood around the object's vertices and how interest points are detected. The program also allows for visualizing the neighborhood used in detecting each interest point.

On the GUI window, as shown in **Figure 1**, interest-points' parameters are found on the top-left corner, the display window is on the top-right corner, display parameters are below and to the left of the display window, and the list of detected interest points are found on the bottom-left corner.

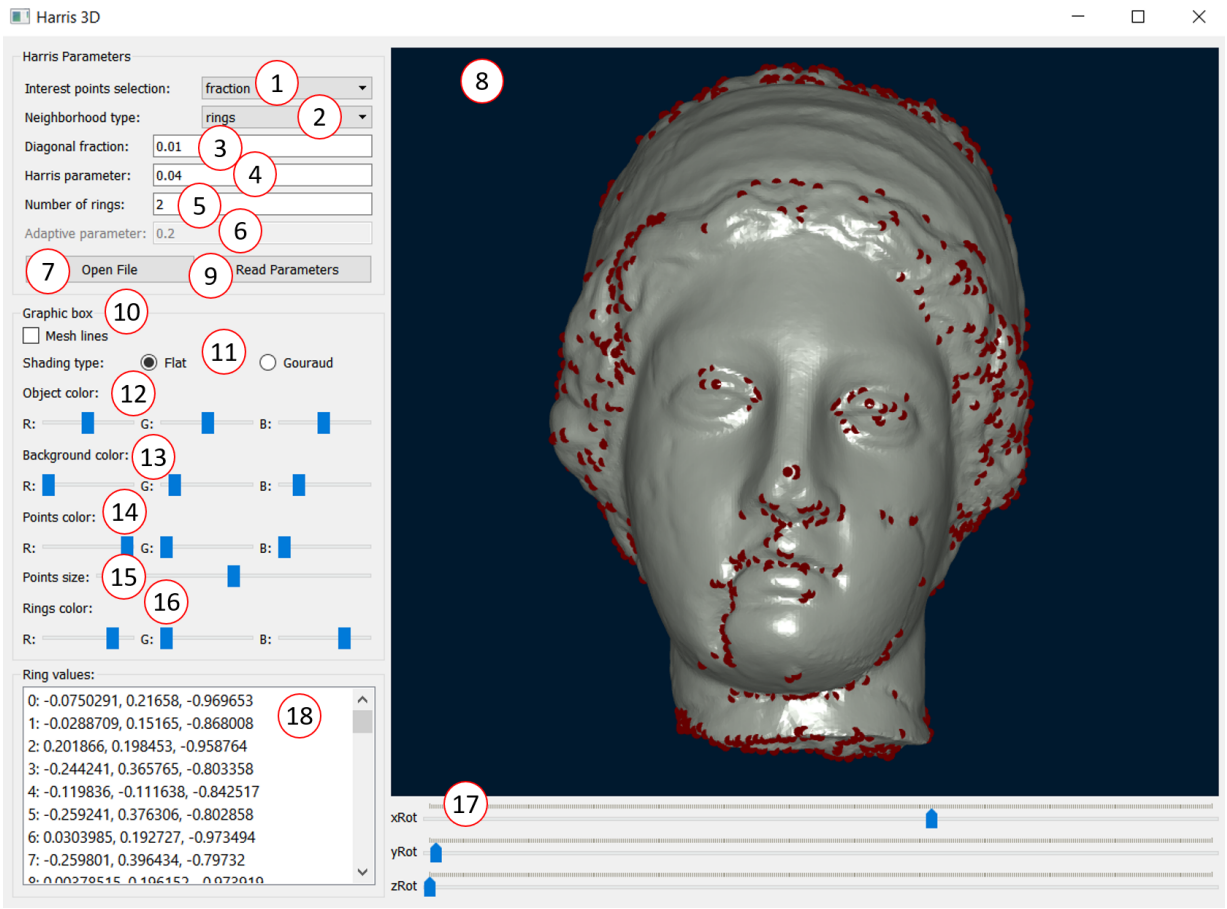


Figure 1: Program Graphical User Interface

For the interest-points parameters,

- ① is a drop list for choosing the type of interest-points calculation method, which can be either **Fractional** or **Clustering**. Fractional, choose interest-points based on multiplying the Diagonal Fraction parameter by the total number of interest points detected. Whereas, clustering, choose the interest-points based on multiplying the same parameter by the diagonal of the object's bounding box.

- ② is a drop list for choosing the method of calculating the neighborhood around the vertices, and it can be either based on **Rings** or **Adaptive** neighborhood. Rings method uses the Number of rings parameter to

determine the maximum number of rings to be included in calculating the neighborhood. Adaptive method multiplies the Adaptive parameter by the diagonal of the bounding box to determine the maximum distance for a vertex to be included in the neighborhood.

- ③ is the **Diagonal Fraction** parameter (a double between 0 and 1) that is used in the interest-points calculation method as stated above.
- ④ is **Harris parameter** (a double) that is used in calculating the interest points. These parameters will be discussed more in details in a later section.
- ⑤ is the **Number of rings** parameter (integer higher than 1) that is used in the calculations in the Rings neighborhood method.
- ⑥ is the **Adaptive parameter** (double between 0 and 1) that is used in the calculations of the Adaptive neighborhood method. It gets activated when the **Adaptive Neighborhood** method is activated.
- ⑦ is the **Open File** push button, which opens up a new window to insert the OFF file, and when an OFF file is chosen, it renders the 3D object and displays it on the display window.
- ⑧ is the **Display window**; Two functionalities worth mentioning here regarding the 3D object rendering, one is the lighting correction, and two, object scaling in case of a large object.
- ⑨ is the **Read parameters** push button. When it is clicked, it passes the provided parameters on to the detection algorithm and calculates the interest-points.

For displaying parameters,

- ⑩ is the **Mesh lines** tick box, which shows and hides the edges of the object faces.
- ⑪ is the **Shading type**, which determines the shading effects on the object's surface. it could give either a **flat** (smooth) or **Gouraud** (rough) look on the surface.
- ⑫ is the **Object color RGB** sliders that controls the color of the displayed 3D object.

- ⑬ is the **Background color RGB** sliders, which also controls the color of the displaying background.

- ⑭ and ⑮ are the **Points color RGB** and **Points size** sliders. They manage the color and the size of the detected interest points respectively.

- ⑯ is the **Ring color RGB** slider that control the color of the interest point neighborhood if an interest point is clicked from the interest-points list box.

- ⑰ are the **xRot**, **yRot** and **zRot** sliders, which are found on the bottom, below the displaying window. These three sliders can be used to rotate the 3D object. It is also worth mentioning that object rotation can be managed by clicking and dragging the mouse right-click as well. Whereas, clicking and dragging the mouse left-click allows for moving the 3D object freely. The zooming functionality, however, is achieved through using the mouse wheel to zoom in and zoom out.

Finally, for the detected interest points,

- ⑱ is the list box in which the (x, y, z) coordinates of all the detected interest points are being shown after detection. And, when an interest point is clicked from the list, the neighborhood used in calculating that interest-point is displayed on the 3D object as shown in **Figure 2**.

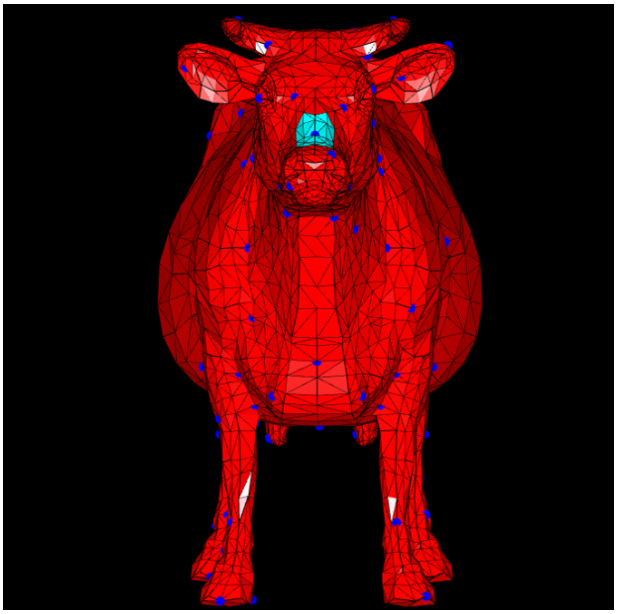


Figure 2: Displaying Interest-point neighborhood

1.2 Project Management

The project was done in a modular structure, meaning it was divided into three main parts: GUI creation, OpenGL 3D rendering, and interest-points calculation, which were managed and tested separately by Abdullah, Ahmad and Zohaib group members respectively, and then after each part was done, they were integrated together to make up the final version of the program.

The team members divided the work and started the implementation end of October. Since then, work milestones have been tackled along the way. Some of the main achieved milestones are presented below:

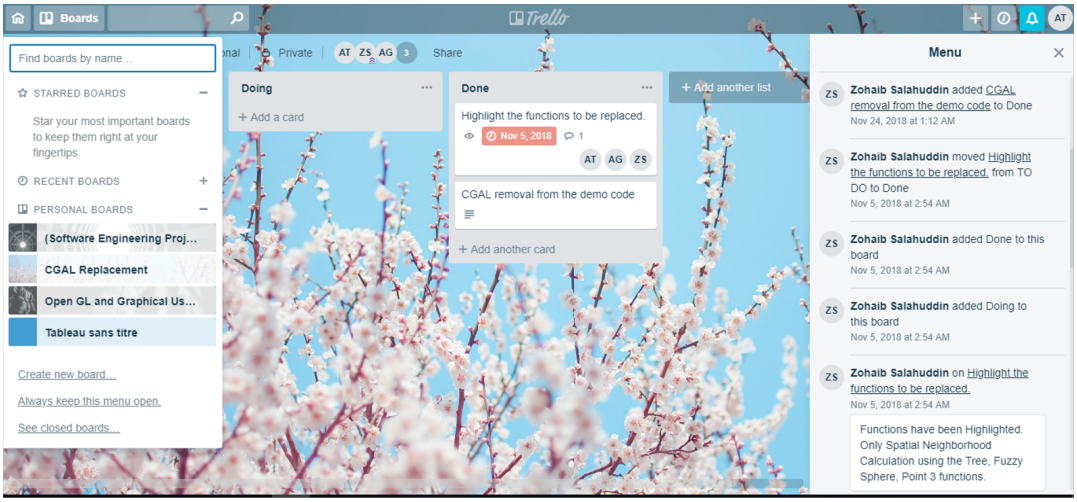
- 04/11/2018: Testing the original code of the article authors
- 24/11/2018: Spatial neighborhood calculations
- 25/11/2018: Creation of the GUI
- 29/11/2018: Initial OpenGL rendering
- 05/12/2018: OpenGL light correction and scaling

- 08/12/2018: GUI and OpenGL integration
- 08/12/2018: Initial results of interest-points detection
- 23/12/2018: Interest points detection and OpenGL GUI integration

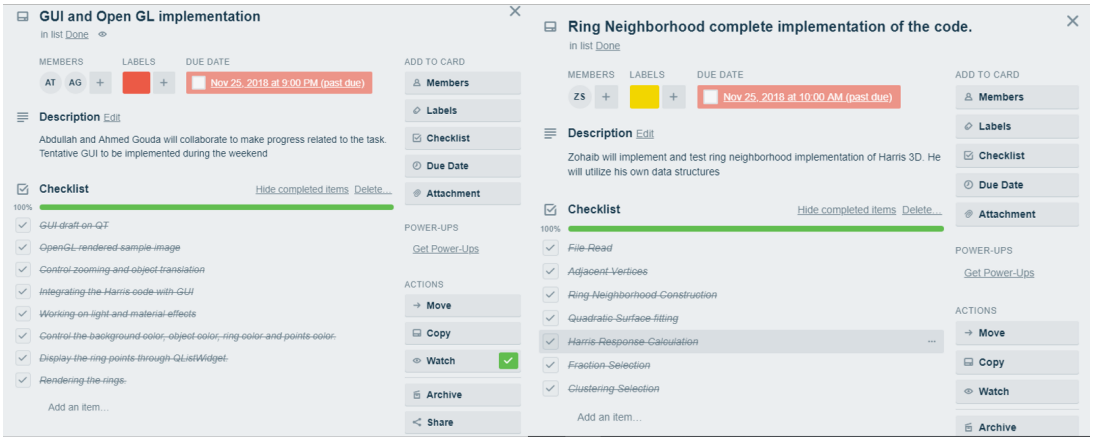
From the end of December, the program was fully functional and the team only worked on adding more functionalities, and also debugging, cleaning and properly commenting the codes.

This steady state of progress was managed through Trello project management boards and through Github platform for code updating and sharing among the members.

In Trello, three boards were created for the project; initially one board for sharing information about the project and understanding the mathematics behind it, and then another two boards were added to keep track of the project progress during coding and implementation.[2] **Figure 3** below shows the boards and some examples of how the project milestones were tracked and achieved.



(a)



(b) (c)

Figure 3: Project management boards in Trello, where (a) shows the three boards on the left and (b-c) shows samples of the checklists in the boards

On the other hand, Github were used mainly for updating and exchanging the codes between the group members. A master branch was added to hold the original and final version of the code, and also another 3 branches, one for each member were created. The codes on the branches were frequently being updated, such as when changes are done to the codes or more functionalities are added. **Figure 4** below shows the time line of updates in one of the

branches.[3]

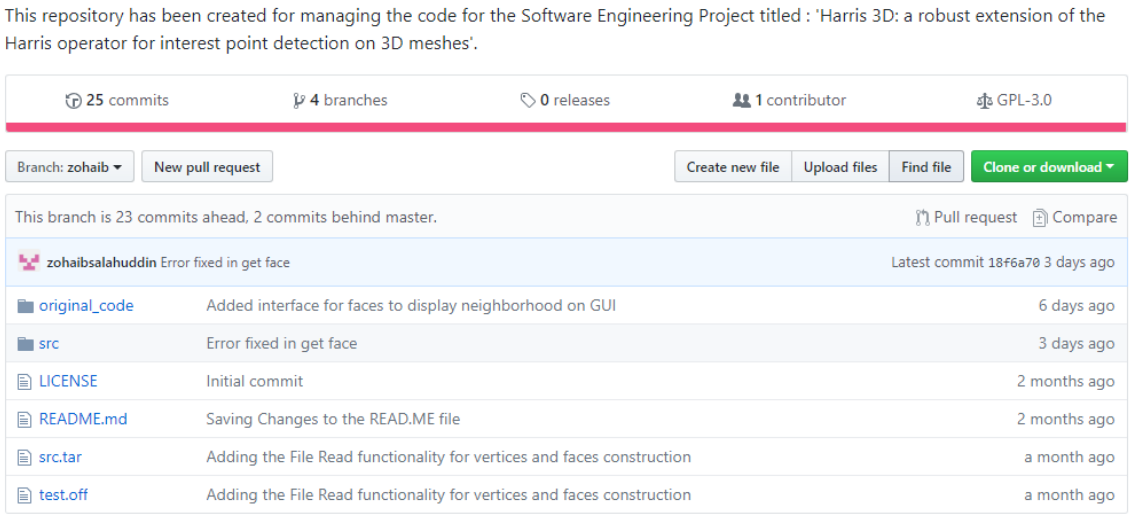


Figure 4: Github time-line updates for the code of interest-points detection

2 Overview of the Steps for Interest Points Calculation

The whole Project can be divided into two parts mainly.

- The first part is concerned with the interest point calculation according to the various parameters that it gets from the GUI.
- The second part is concerned with the display of the interest points calculated overlayed on the 3D meshes.

The first part will describe in detail the implementation of steps for interest points calculation. The following is the summary for interest points calculation:

2.1 Steps for Interest Points

2.1.1 Populating the 3D Mesh

The first step in the interest points calculation is the population of the Mesh. the following three things are populated while reading from the OFF file.

- The Vertices are read from the OFF file. They are x,y,z co-ordinates are populated along with their index.All the faces that include that vertex are also stored in the Vertices class. Information about each vertex's very adjacent vertices index is also stored.
- The faces are read from the off file. They are populated along with their index and the vertices that constitute those faces.
- The information about each edge is stored. Two Vertices that are the part of the edge are stored along with the edge index

Steps 2-5 are repeated for all the Vertices

2.1.2 Neighborhood Calculation

Two types of Neighborhoods are Calculated depending on the type of parameters that have been provided:

- **Ring Neighborhood** is calculated depending on the number of ring that has been provided as the input. If Ring Number is provided as 1, the neighborhood includes only the adjacent vertices. If the Ring Number is provided as 2, it also includes adjacent vertices of the adjacent vertices and so on.

- **Adaptive Neighborhood** is calculated based on a specific radius. If the distance of any of the vertex to be added in the neighborhood increases the specified threshold, that vertex is added and further calculation is halted.

2.1.3 Principle Component Analysis

After we get the neighborhood, we perform Principle Component Analysis to reduce the dimension as such that the z axis points normal to the surface of the plane formed the other two axis. This dimensionality reduction is achieved with the help of Eigen Library so that a quadratic surface can be fit through a set of points.This process consists of the following steps:

- Shift all the neighborhood points as such that the center of the neighborhood is at zero.
- Calculate the Covariance Matrix and in turn eigen vectors in descending order.
- Multiply the Data with the Eigen Vectors so that the Z axis Points towards the normal.

2.1.4 Quadratic Fitting

After we transform our data as such that the z axis is in the normal direction. All we need to do is solve quadratic equation solution with six variables using *Linear Least Squares*. We have to solve **AX=B**, where A is an **N x 6** matrix containing all the vertices with their X and Y co-ordinates only, X is the matrix of the Coefficients that we want to calculate for the Quadratic Fitting and B is the normal vector which is z co-ordinates. This

is essentially so because each vertex can be represented by it's normal only.

2.1.5 Harris Response

After the Coefficients of the Quadratic Surface Fit have been calculated, the harris response calculation simply reduced to the plugging in of the values and finding the Harris Response.

2.1.6 Local Maximum Points Detection

After the interest points have been calculated, we checked for all the points that had harris response greater than their adjacent vertices. All these points were separated.

2.1.7 Selection of Interest Points

Then the interest points can be chosen from these points based on the following two methods:

- **Selection By Fraction** Sort the Interests Points in the Descending order of their Harris Response. Select the first (fraction * total) points. These are the interest points
- **Selection By Clustering** Add the first point in the interest point list. All the subsequent Points from the list of local maxima should be chosen only if they are a certain threshold distance apart from the interest points already in the list. After Iterating through all the local maxima points, the points in the list with a certain distance apart are the interst points.

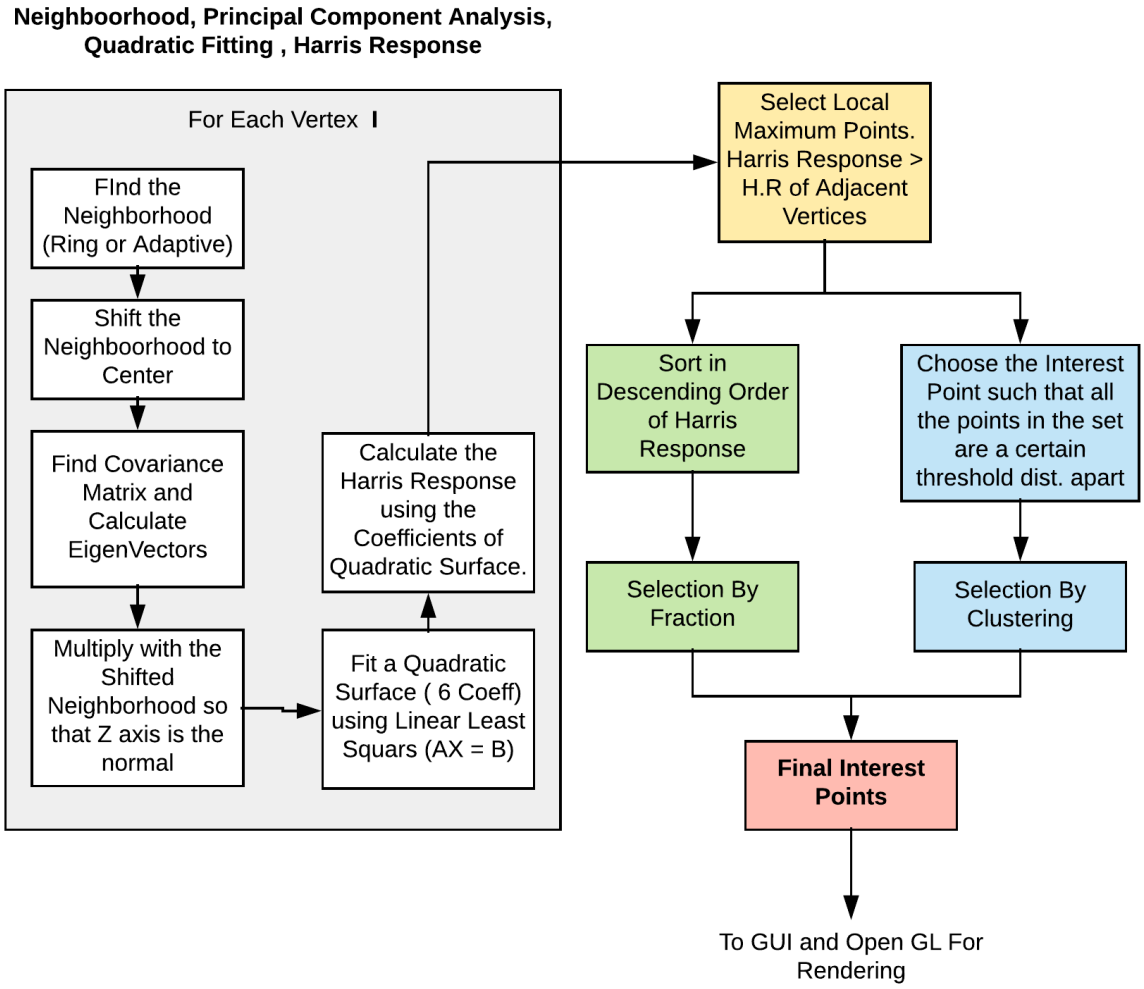


Figure 5: Overview of the Interest Point Selection Methodology

3 Structure of the Code

3.1 The Vertices Class

The Vertices class stores all information related to the vertices. The following diagram summarizes the information about the Vertices class and its constituents.

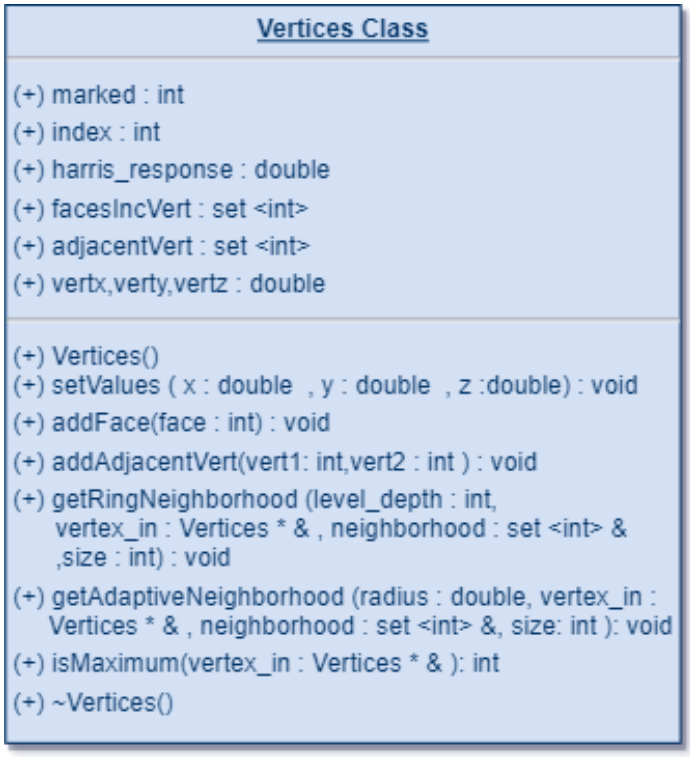


Figure 6: UML Diagram : Vertices Class

The objects of this class are used to represent the vertices of the 3D mesh.

Instances of this class equal to the number of vertices are the part of the ***Mesh*** *class.*

This Class in addition to the vertex co-ordinate contains information about the immediate neighbors which is populated when the file is being read. It also contains information about the faces its the part of. Its harris response and the functions related to calculate the neighborhood of a particular class are also the part of this class.

3.2 The Face Class

This class stores the information related to the faces that construct the 3D mesh. Instances of this class equal to the number of Faces are the part of the **Mesh** class.

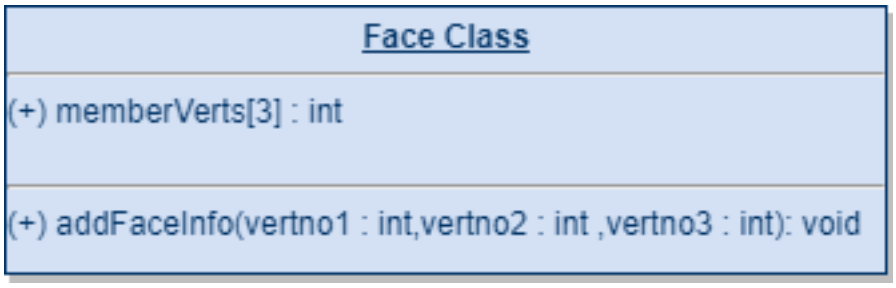


Figure 7: UML Diagram : Face Class

3.3 The Edges Class

This class stores the information related to the edges that construct the 3D mesh. Instances of this class equal to the number of Edges are the part of the **Mesh** class.

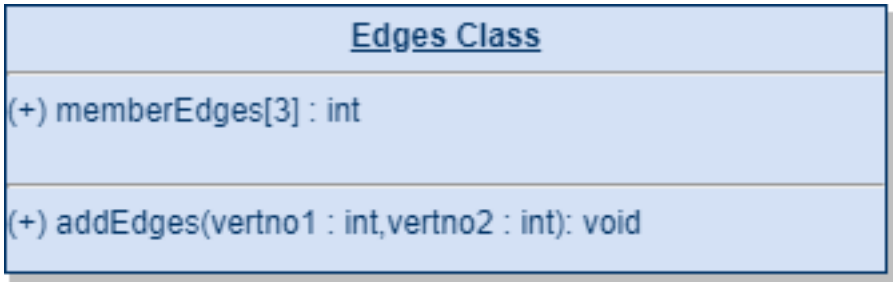


Figure 8: UML Diagram: Edge Class

3.4 The Mesh Class

The Mesh Class uses the above three Classes (Face, Edges and Vertices) combined with other information to construct the 3D Mesh of the object. It includes all the functions to read the given OFF file and populate the member Vertices , Faces and Edges that constitute this Mesh Object. Also, it includes information regards total number of Vertices, Faces and Edge objects that it includes. The following diagram suitably describes the Mesh Class:

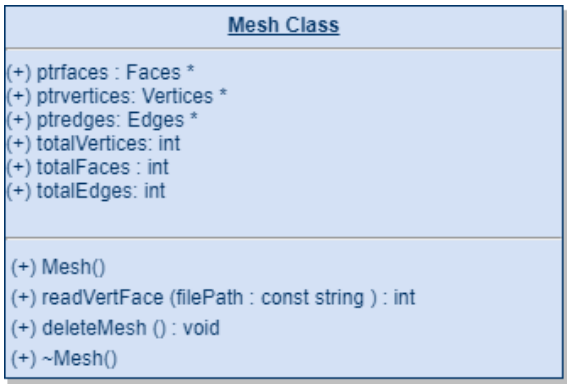


Figure 9: UML Diagram : Mesh Class

3.5 PCA.cpp - PCA, Quadrattic Fitting and Harris Response

Once the Mesh has been populated in the Mesh class, getNeighborhood Function is called for each Vertex in the mesh using the constituent functions in the Vertices class. Once we have the neighborhood we proceed with performing the following functions with **PCA.cpp**:

- Principle Component Analysis
- Quadratic Fitting
- Harris Response Calculation
- Selection through Clustering or Fraction

All the above mentioned objectives are achieved through the functions in *PCA.cpp* file as mentioned below:

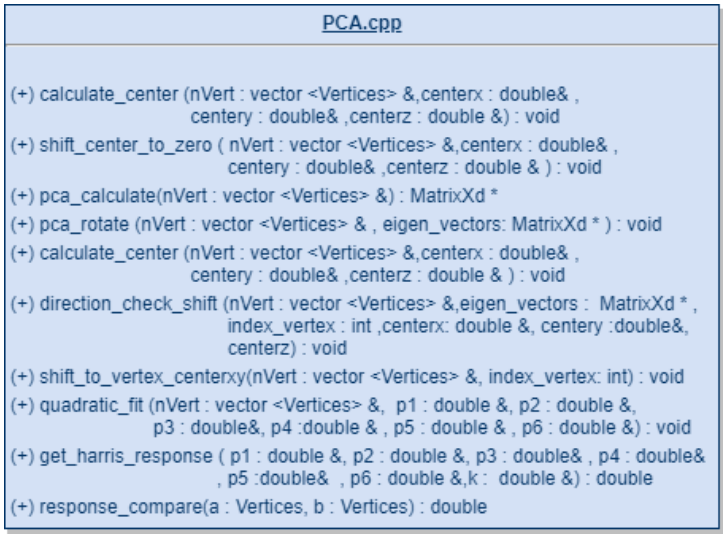


Figure 10: Functions in the PCA.cpp file

Please Note that the Eigen Library has been used to make calculations related to PCA and Quadratic Fitting.

3.6 Harris.cpp - The Interface to GUI and OpenGL

These functions provide the interface between the GUI , OpenGL and the Interest points calculation. Since the two parts of the whole code were developed in a modular fashion, this part receive all the parameters necessary for calculating the harris response for all the vertices. These functions then return the interest points back to the GUI to be displayed.

It contains two main functions.

- The function **cal interest points** returns the interest points to be displayed.
- The function **get faces** returns the specified faces in the neighborhood for a particular point.

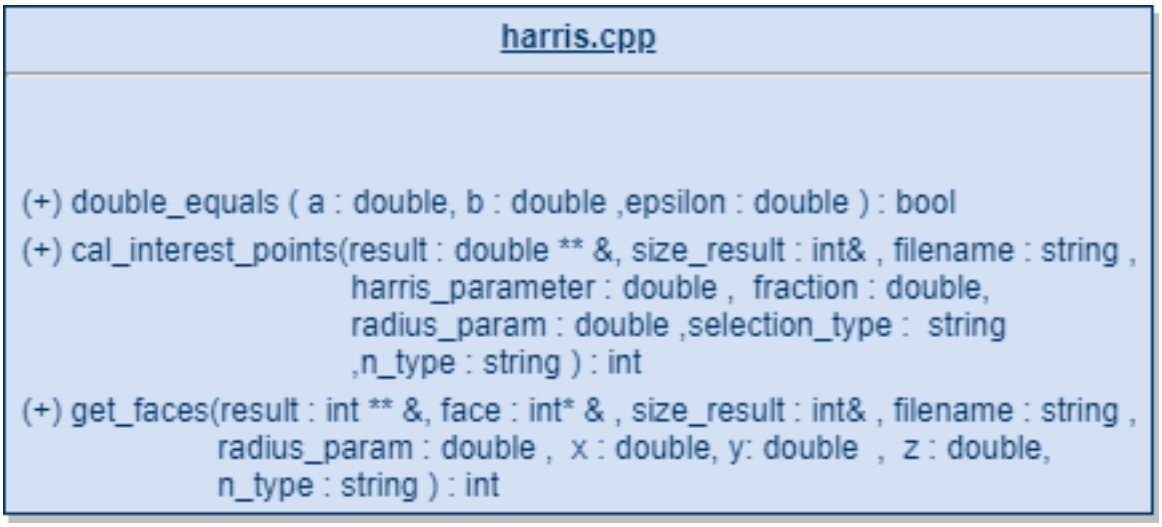


Figure 11: Functions in the harris.cpp file

4 Neighborhood Calculation

The Neighbor hoods are Calculated Based on the Neighborhood type that has been provided as the parameter. The following two neighborhood types have been implemented:

4.1 Ring Neighborhood

The Neighborhood for each Vertex is calculated with the help of the **getRingNeighborhood** function provided for each vertex in the Vertices class. The following algorithm has been used to implement the Ring Neighborhood :

Algorithm Used for Ring Neighborhood Detection

set <int> A - Containing Index of Vertices in the neighborhood Calculated.
int k - The index of vertex for which the neighborhood is required
int Rings - The size of the ring neighborhood

```
set <int> tempA.insert(k)

Begin:
      for  i =1 : Rings
           temp.insert(A.begin(),A.end())
           for j=1 : sizeof(A)
                temp.insert(indices of adjacent Vertices of Vertex j)
           end
           A.insert(temp.begin(),temp.end())
           temp.clear()
      end
end

A - contains the "Rings" size neighborhood for the vertex index k
```

Figure 12: Algorithm for Ring Neighborhood Calculation

The Ring Neighborhood Calculated for each of the points is displayed can also be displayed on GUI. For each of the Interest points that have been calculated, they are displayed as a list on the left corner in the GUI. Clicking on the GUI will display the Neighborhood used for that point for the Harris Response Computation. This also helped us to visualize that we were using the correct neighborhood and the algorithm was working perfection. In the following image, we can see a 3 Ring Neighborhood that can be seen around one of the points on the image:

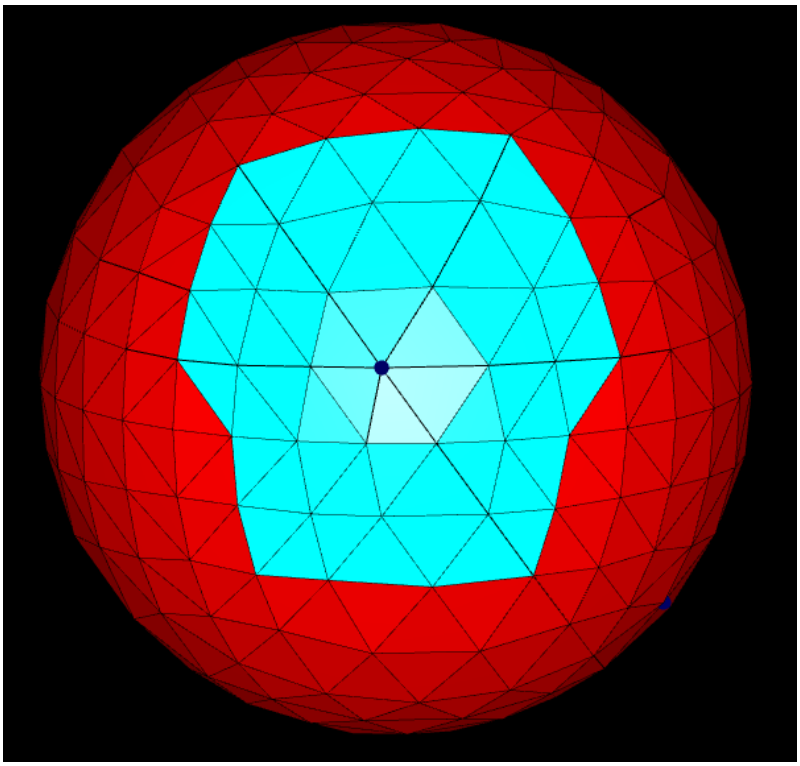


Figure 13: 3 Ring Neighbor Displayed on GUI for a point by Clicking

4.2 Adaptive NeighborHood

The adaptive neighborhood has been implemented so that the neighborhood doesn't unexpected grow in one direction rather a uniform neighborhood from all the direction is assumed. One of the input parameters provided for calculating this neighborhood is the percentage of the total diagonal of the box bounding all the vertices in the three axis.The following algorithm has been used to implement the Ring Neighborhood :

Algorithm Used for Adaptive Neighborhood Detection

set <int> A - Containing Index of Vertices in the neighborhood Calculated.
int k - The index of vertex for which the neighborhood is required
int threshold - The threshold for the adaptive neighborhood

```
set <int> temp.insert(k);
set <int> temp2; flag = 0;
Begin:
        while(!flag)
             temp.insert(A.begin(),A.end())
            for j=1 : sizeof(A)
                temp.insert(indices of adjacent Vertices of Vertex j)
                temp2.insert(indices of adjacent Vertices of Vertex j)
                for l = 1:sizeof(temp2)
                      if(dist_vert(j,k) < threshold)
                        flag =1
                end
                temp2.clear()
            end
           A.insert(temp.begin(),temp.end())
       end
end
A - the neighborhood for the vertex index k ring expansion limited by distance.
```

Figure 14: Algorithm for Adaptive Ring Calculation

This has been implemented in such a way that if the distance between the center vertex and any of the vertices to be added in

the neighborhood is higher than the threshold mentioned, then these vertices are added to the neighborhood and further addition stops to the neighborhood and the process is terminated. However, if the distance is less the above specified condition is not met, it keeps on adding more and more vertices to the neighborhood until the above mentioned threshold is exceeded. The following image shows adaptive neighborhood implementation on one of the images:

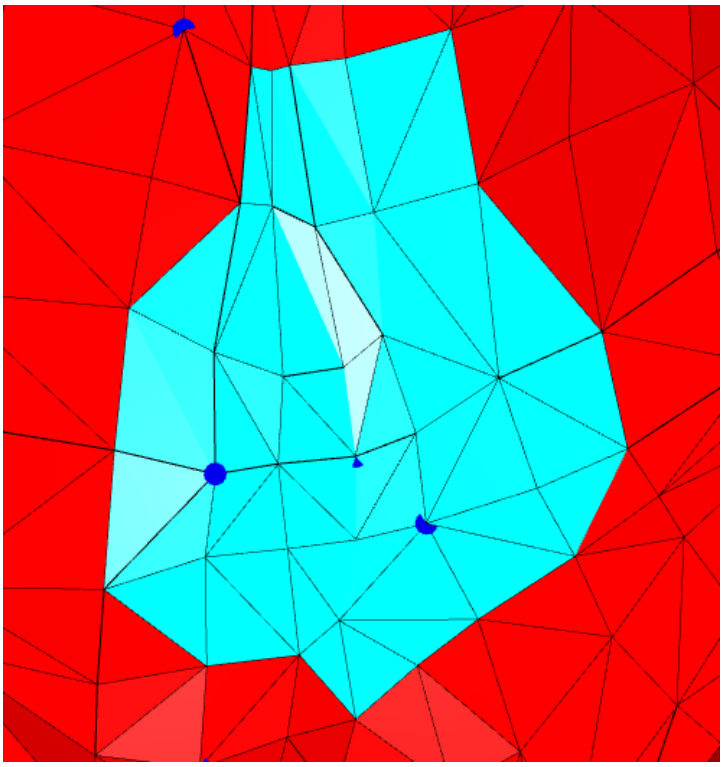


Figure 15: Adaptive Neighborhood for an Image of Rabbit zoomed In.

5 Principle Component Analysis

Once the Neighborhood has been implemented, the Principle Component Analysis is Performed. *To perform PCA,* ***Eigen Library*** *has been utilized.[4]* Following are the steps performed for PCA on each neighborhood of the vertex:

5.1 Shifting the Center of the Neighborhood to Zero

The Center of the Neighborhood Vertices is calculated and it is subtracted from each value of the vertex to shift the neighborhood to the center. It is done with the help of the following two functions:

```
void calculate_center (vector <Vertices> & nVert,double&
    centerx,double& centery,double &centerz);
void shift_center_to_zero ( vector <Vertices> & nVert,double&
    centerx,double& centery,double &centerz );
```

5.2 Calculating Eigen Vectors and Eigen Values

Once the Centroid has been shifted to the center, the Eigen Values and Eigen Vectors are Calculated using the Eigen Library. First, the data is populated in the **MatrixXd data** of size **3 x N**.Then the Covariance matrix is obtained by Matrix Multiplication of data and data transpose. Then Eigen Library is used to solve for Eigen Values and Eigen Vectors. Following Snippet of code demonstrates this:

```
    MatrixXd data = MatrixXd::Zero(3,nVert.size());

    for(int i =0; i < nVert.size(); i++)
    {
```

```
        data(0,i) = nVert[i].vertx;
        data(1,i) = nVert[i].verty;
        data(2,i) = nVert[i].vertz;
    }

    MatrixXd covariance = MatrixXd::Zero(3, 3);
    covariance = (1 / (double) (nVert.size()-1)) * data *
        data.transpose();

    SelfAdjointEigenSolver<MatrixXd> solver(covariance);
    VectorXd eigen_values = VectorXd::Zero(3);
    eigen_values = solver.eigenvalues().real();

    MatrixXd * eigen_vectors = new
        MatrixXd(MatrixXd::Zero(nVert.size(), 3));
    *eigen_vectors = solver.eigenvectors().real();
    double temp_eigen_val = 0;
    VectorXd temp= VectorXd::Zero(3);
```

The Eigen Values and Eigen Vectors Obtained are in ascending order, we sorted them again in ascending order.

5.3 Direction Check of Eigen Vectors

Since the Eigen Vectors are not unique, it may so happen that the third eigen vector with the lowest Eigen value, points in a direction that is opposite to the required direction. It ordered to counter that we check for the product of the vertex who response is being calculated with the third eigen vector. If it is greater than zero, we exchange the first two eigen vectors and change the sign of the third eigen vector in accordance with the right hand rule.The following function helps us in achieving this.

```
void direction_check_shift (vector <Vertices> & nVert, MatrixXd *
    eigen_vectors, int index_vertex,double& centerx,double&
    centery,double &centerz);
```

5.4 PCA Rotation

Finally once we have the correct Eigen Vectors, we multiply the data matrix with the eigen vectors so that the X and Y co-ordinates form the plane and the Z axis points in a direction normal to the plane formed by the first two. The following function is used for the PCA rotation.

```
void pca_rotate (vector <Vertices> & nVert, MatrixXd *
    eigen_vectors);
```

After the PCA rotation is done, we have z co-ordinates in a direction that is normal to the plane formed by x and y co-ordinates.

6 Quadratic Fitting,Harris Response and Local Maximum Points

6.1 Quadratic Surface Fitting

Since a Quadratic Surface can be represented by the normal, all we need to do now is solve the following system by **Linear Least Squares**:

$$AX = B \tag{1}$$

Here B is equal to the Z cordinate value for each vertex in the neighborhood. A will be a **N x 6** matrix and X will be a **6 x 1** matrix as specified by the following equation:

$$z = 0.5 * p_1x^2 + p_2xy + 0.5p_3y^2 + p_4x + p_5y + p_6 \tag{2}$$

We wrote the following piece of code to solve this using eigen library:

```
MatrixXd data = MatrixXd::Zero(nVert.size(),6);
   VectorXd B= VectorXd::Zero(nVert.size());
   VectorXd X= VectorXd::Zero(6);
   for(int i =0; i < nVert.size(); i++)
   {
       data(i,0) = (nVert[i].vertx* nVert[i].vertx)/2;
       data(i,1) = nVert[i].vertx * nVert[i].verty;
       data(i,2) = (nVert[i].verty * nVert[i].verty)/2;
       data(i,3) = nVert[i].vertx;
       data(i,4) = nVert[i].verty;
       data(i,5) = 1;
       B(i) = nVert[i].vertz;
   }

   X =  data.colPivHouseholderQr().solve(B);
```

Finally we have the values of the coefficients for the fitted quadratic surface.

6.2 Harris Response

Once we have the Coefficients values, the final step reduces to simply putting the coefficients in the equations to get the value for harris response. We did it in the following way:

```
A = (p4*p4) + (2*p1*p1) + (2*p2*p2);
B = (p4*p4) + (2*p2*p2) + (2*p3*p3);
C = (p4*p5) + (2*p1*p2) + (2*p2*p3);


det = (A*B) - (C*C);
tr = A+B;
response = det - (k*tr*tr);
```

6.3 Local Maximum Points

Once we have response available for all the points, we check for all the points who have their harris response greater than their adjacent neighbors. These points with their response greater than their adjacent vertices could be the possible interest points depending on the method of selection we use.

7 Interest Points Selection

Once the local maximum points have been obtained, you can proceed to select the final interest points in the following two ways:

7.1 Fraction

- Sort all the local maximum points in the decreasing order of their harris response.
- Depending on the fraction input parameter, Calculate **M = fraction * total no of Vertices**. Round to the nearest integer.
- Select **first M** number of interest points from the list to get the final interest points.

The following images represents the Interest Points selected from objects using fraction selection:

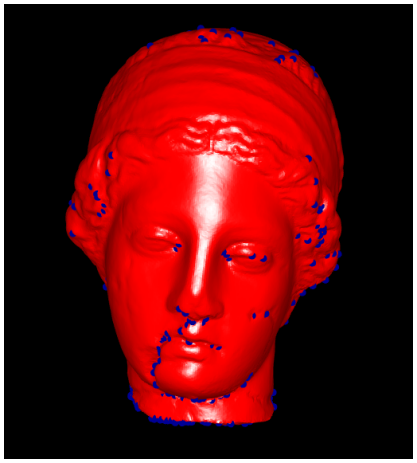


Figure 16: Interest Point Selection Using Fraction, Sample Image 1

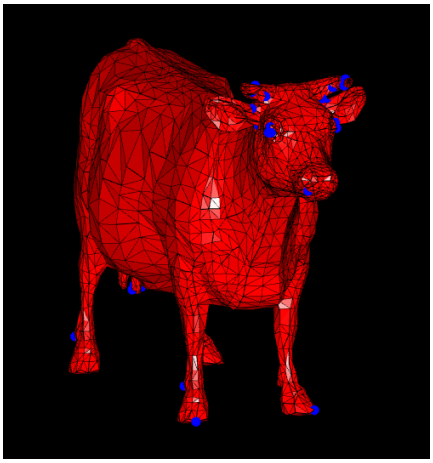


Figure 17: Interest Point Selection Using Fraction, Sample Image 2

7.2 Clustering

The choice of interest points from the local maxima points can be done by following the below mentioned steps:

- Multiply **Clustering Selection Fraction** by the bounding box diagonal containing all the vertices. This gives the *Distance Threshold*
- Once the Distance threshold is obtain, choose one point from the list of local maximum points and add it to the list of interest points.
- For all the other points, *if next point in the local maximum points list has a distance greater than the threshold from all the points all ready present in the interest points list, only then it should be added.* Otherwise the point is discarded.

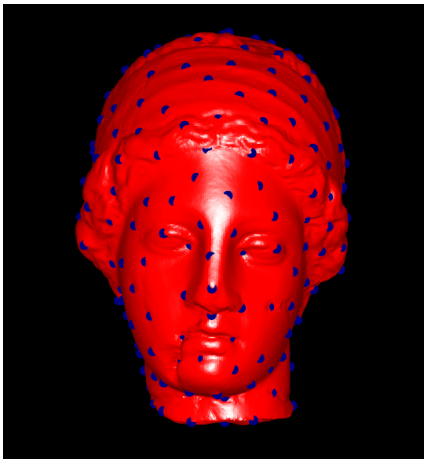


Figure 18: Interest Point Selection Using Clustering, Sample Image 1

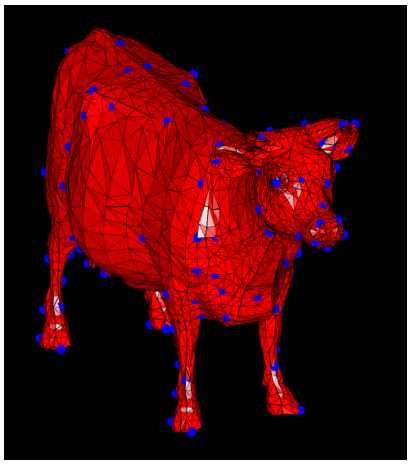


Figure 19: Interest Point Selection Using Clustering, Sample Image 2

8 Interface for GUI and OpenGL

The Interest Points Selection is linked with the GUI using the two functions that are mentioned below:

```
int cal_interest_points(double ** & result, int & size_result,
    string filename, double harris_parameter, double fraction,
    double radius_param, string selection_type,string n_type);
int get_faces(int ** & result, int * & face, int & size_result,
    string filename,double radius_param, double x, double y ,
    double z,string n_type);
```

8.1 cal interest points(...)

This function receives the input parameters from the GUI and returns the interest points in the form of an Array of size **N x 3**. The size N is also returned with this.This function includes all the implementation for the interest points calculation. You just need to provide it with the input parameters. This function is linked with the GUI with the *Read Parameters* button

8.2 get faces (...)

When you call the above function and the interest points have been calculated and displayed both on the mesh and the left window on the lower left corner containing the interest points. If you click on one of the interest points in the left corner, then this function is called for that point. It returns the neighborhood faces for that point that have been used to calculate the harris response for that point. These faces are then displayed overlayed on the mesh in a different color.

9 Open Graphics Library (OpenGL)

OpenGL is defined as a cross-platform application programming interface (API) for rendering 2D and 3D models by using vertices of geometry shapes. These geometry shapes are called primitives such as triangle, point, line and quad. Figure 20 shows the main stages for processing graphical rendering:

- The application loads a set of varices and primitives for the 3D model. Then it connects the vertices points of each primitives.
- The next stage is Rasterization. In this stage each connected primitive are filled with a the pixel grid which are called fragments.
- In the Fragment Processing stage each Fragment each pixel value is changed according to color, normal vector position and texture.
- In the Output Merging stage the fragments are combined together to from 3D or 2D model for the display.

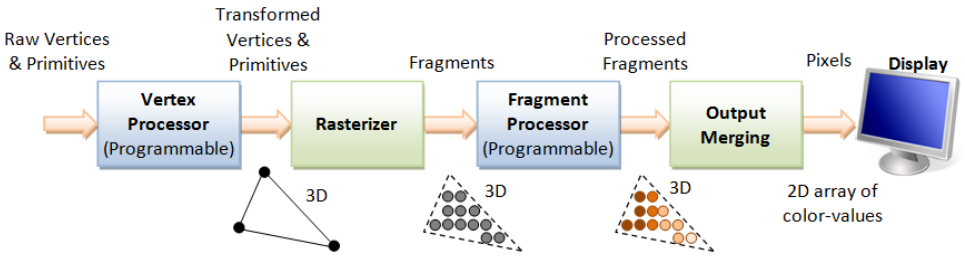


Figure 20: Graphic Rendering Pipeline

9.1 Loading OFF file

OFF stands for (Object File Format). It is a file format that contains a set of vertices and of geometry polygons(faces) which are considered basic elements for rendering 3D objects. The file is divided into 4 sections:

- The first section is file format description. The first line in the file is "OFF" word.
- The second section contains three numbers which are number of vertices, number of faces, number of edges (edges in some OFF files is usually ignored).
- The third section is list of vertices in x, y and z coordinates.
- The fourth section is list of faces. It starts with the number of vertices, followed by the numbers of vertices index (starts from zero).

9.1.1 Extracting Vertices and Faces

To load off file in GUI, the 'Open file' button is clicked then a 3D model OFF file is selected. Then, new object is created with 'MyOffFile' class. This class is developed to extract the verices and faces parameters for 3D model with triangular faces only. It then can return address locations of give dynamic arrays as a getters which are:

- Vertices_Buffer: Dynamic array with type float with three columns. It contains all vertices values.
- Faces_Buffer: Dynamic array with type int with three columns. It contains all vertices values.

- normalized_Vertices_Buffer: Dynamic array with type int with three columns. It contains all normalized vertices values.
- Faces_Normal_Vectors_Buffer: Dynamic array with type float with three columns. It contains the normal vectors for each face. It is used in flat shading effect.
- Vertices_Normal_Vectors_Buffer: Dynamic array with type float with three columns. It contains the normal vectors for each vertices. It is used in light for Gouraud shading effect.

'MyOffFile' can also return the values of total number of varices, total number of faces, global maximum, global minimum values for vertices, and global maximum value and global minimum values for normalized verices.

The following diagram in Figure 21 is a flow chart for MyOffFile' class to extract the OFF file parameters.

9.1.2 Normalize Vertices

Since each 3D model differs in size, the vertices are normalized to fit the orthographic model. The 'normalize_vertices( )' function normalizes the each vertex by detecting the global minimum and maximum for all vertices coordinates. Then, it divides each vertex coordinates by (global maximum global minimum).

9.1.3 Flat shading normal vectors

The normalization vectors for each face are calculated by 'calculate_faces_normal_vectors()' function. It is calculated by calculating the cross product for each face vectors to get the normalize

vector which represents normalize vector for the whole face as

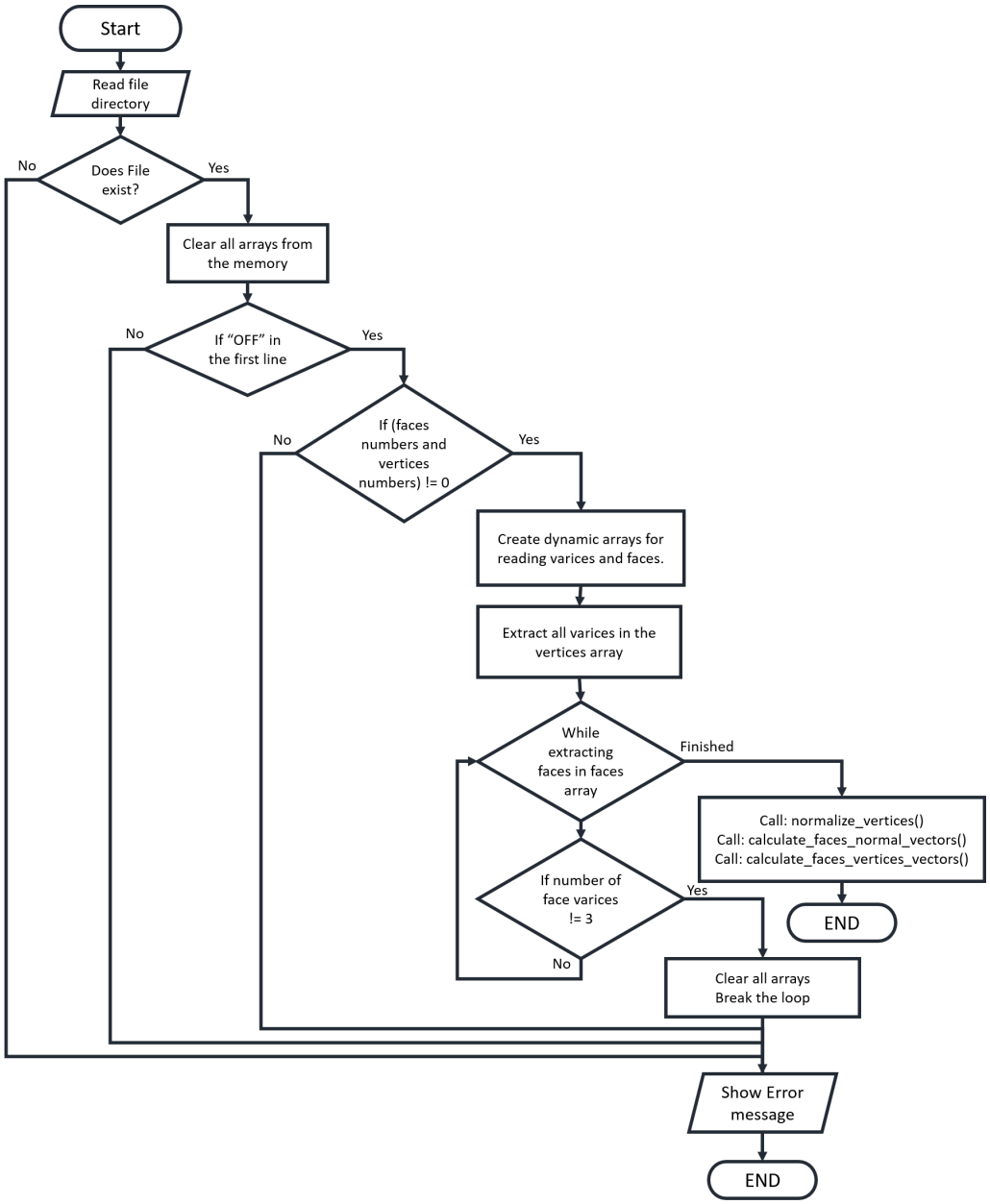


Figure 21: Flow Chart for Reading .OFF file

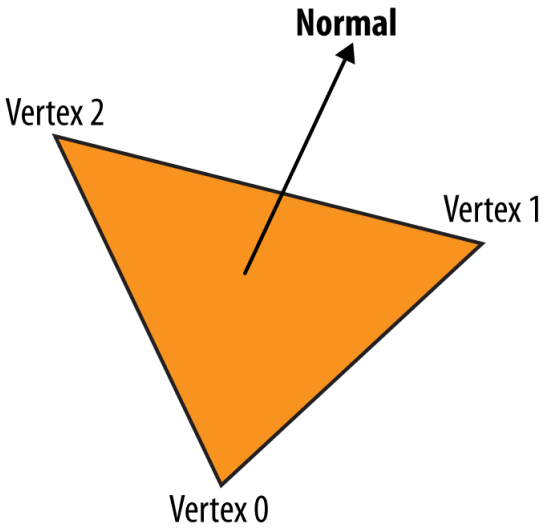


Figure 22: Normal Vector Direction in Flat Shading

shown in Figure Figure 22.

Then the output vertices can be normalized mathematically or by using 'glEnable(GL_NORMALIZE)' command during initialization.

$$V_{norm} = \frac{V}{|V|}$$

9.1.4 Gouraud shading normal vectors

The 'calculate_vertices_normal_vectors()' function is used in calculating the normalization vectors for each vertex are calculated by finding the faces which share each vertices. Then adding and normalizing the normalize vector for these face which are previously calculated, and then normalize them to get the normal vector for each vertex. This new normal vector represents all

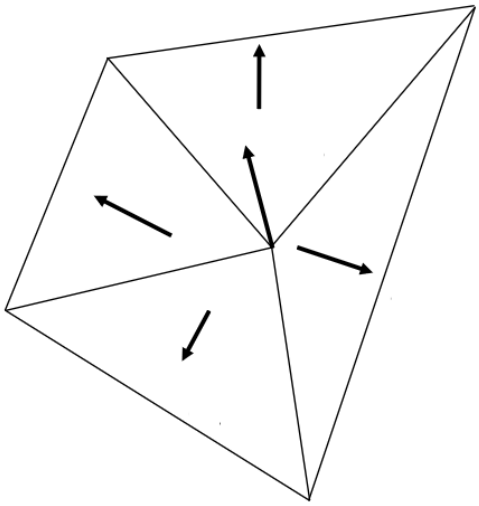


Figure 23: Normal Vector Direction in Gouraud Shading

the normal vectors of the neighbouring faces as shown in Figure 23.This method provides more smooth surface effect for the object.

In this condition the normalized normal vector for each vertex is calculated as shown in the following equation where n is total number of faces.

$$V_{norm.} = \frac{V_1 + V_2 + ...V_n}{|V_1 + V_2 + ...V_n|}$$

9.2 OpenGL parameters

In this section, the initialization parameters and OpenGL functions that are applied in the project. Theses functions are included 'MyGLWidget' class and they are explained in this section.

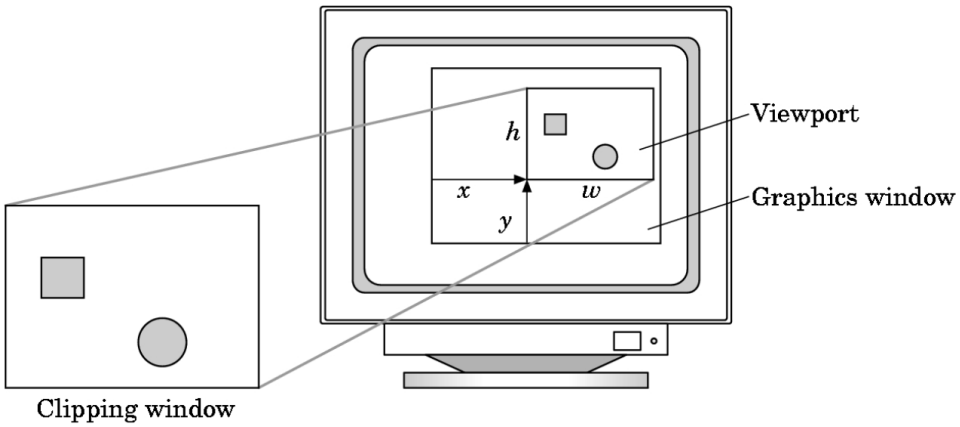


Figure 24: OpenGL Viewport and Graphic Window Adjustment

9.2.1 View adjustment and projection Matrix

The 'resizeGL(int width, int height)' is a member function of 'MyGLWidget' class which is used for initializing the Viewport and orthographic volume to show the rendered shape.

Firstly, glViewport(x,y,w,h) is initialized. This function is used for fitting the OpenGL canvas boundaries inside the program window. The x and y parameters define the lower left corner within the OpenGL canvas, while w and h define the width and height of the GLwidget window as shown in Figure 24.

The 'glViewport' function parameters are calculated according to the dimensions of GUI window as shown in the following code:

```
int side = qMin(width, height);
glViewport((width - side) / 2, (height - side) / 2, side, side);
```

The 'glMatrixMode(GL_PROJECTIONmatrix)' function is used in enable projection matrix which is used to define the frustum. This frustum is used to define the clipping boundaries. In addi-

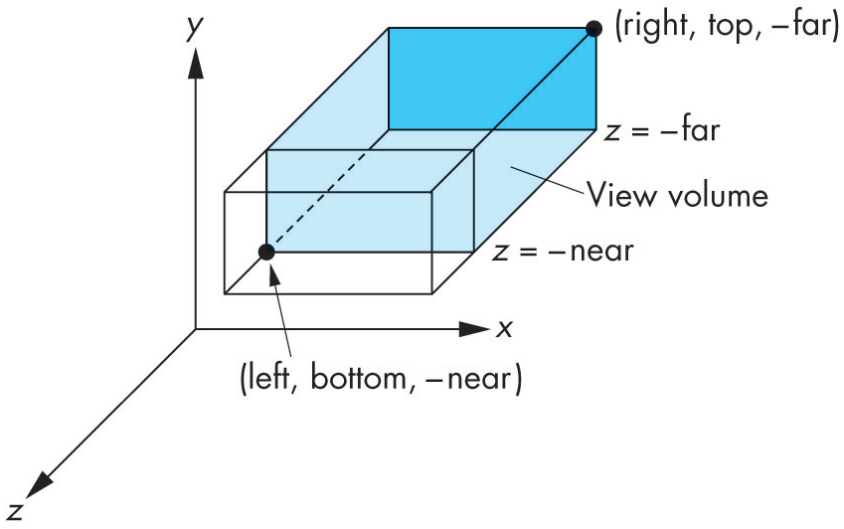


Figure 25: Orthographic Projection Parameters Adjustment

tion , it is used to define the projection of 3D object in screen. The 'glLoadIdentity()' function then is called for setting initialize ModelView matrix to the identity matrix. It is used for resetting the location of object with respect to the light source when loading a new 3D model. The 'glOrtho(left,right,top,bottom,near,far)' function is used for producing a orthographic (parallel) projection of the 3D model within specific volume. This volume is defined by 6 parameters to specify 6 clipping volume planes. These planes are left, right, bottom, top, near and far plane as shown in Figure 25.

The left and bottom parameters are set to equal the global minimum of normalized vertices. while the right and top parameters are set to equal to the global maximum of normalized vertices. The near and far parameters along z direction are set to equal 1.0 and 15.0 respectively as shown in the following code.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

```
#ifdef QT_OPENGL_ES_1
    glOrthof(read_globalVerMinNorm, read_globalVerMaxNorm,
        read_globalVerMinNorm, read_globalVerMaxNorm, 1.0, 15.0);
#else
    glOrtho(read_globalVerMinNorm, read_globalVerMaxNorm,
        read_globalVerMinNorm, read_globalVerMaxNorm, 1.0, 15.0);
#endif
```

9.2.2 Light and Martial

In real world the eye detects the effect of the color as a result of the reflected light sources from the the colored material surfaces. When the incident light hits an object, the reflected color intensity changes according to three parameters. The first parameter is the location and the position of light source. The second parameter is the location of the viewer. In addition, The last parameter is reflection and scattering factors of the material surface. In real world the color that the viewer perceives is reflected from 3D object. In order to detect 3D effect in 2D screen, a small variation in colors can be applied in the model which is called shading.

Therefore, the light source in openGL is enabled from the following functions:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

The light effect can be modeled into three components which is called Phong lighting model. These three components are ambient, diffuse and specular lighting as shown in Figure 26:.

- Ambient lighting: It simulates detecting the object in the dark with small distant light.

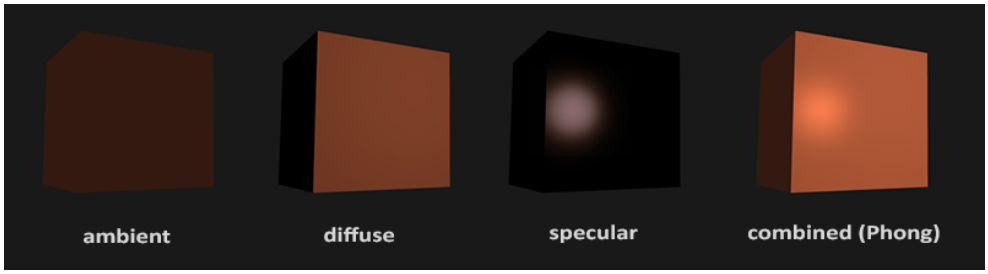


Figure 26: Phong Model for Light Effect

- Diffuse lighting: It represents the effect of light direction on the object. Which causes a darker or brighter color in different faces of the object.
- Specular lighting: It represent the bright light spot effect due to a close light source to shiny object.

In OpenGL the light for each component is applied by using 'glLightfv' function. This function has three parameters. The first parameter is enabled light source 'GL_LIGHT0', the second parameter is the light component 'GL_AMBIENT','GL_DIFFUSE' or 'GL_SPECULAR'. The last parameter is an array represents the color of light source in RGBA. The elements of this array is float values from 0 to 1. To get a good light effect the addition results of diffuse and ambient lights are set to equal one, while specular color components is set to 1 as shown in the following code:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,GL_TRUE);
GLfloat light_ambient[] = {0.2, 0.2, 0.2, 1.0};
GLfloat light_diffuse[] = {0.8, 0.8, 0.8, 1.0};
GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

'GL_POSITION' components defines the location of the light source in x,y and z space. The x and y are set to equal 0, while z is set to equal 2.

```
GLfloat lightPosition[4] = { 0, 0, 2, 1.0 };
 glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
```

The material type defined by the three components as in light which are ambient, diffuse and specular with additional component which is the shininess. The ambient material represents the color which object reflects in the case of ambient lighting. The diffuse material vector represents the object color under diffuse lighting. The specular material is the effect of color specular light on the object. Lastly, the shininess represents the scattering factor of the material in specular light. These 4 components can be used to define the type of material for objects in real-world. In OpenGL, glMaterialfv function controls the material as similar as the light function. In the following code, the diffuse and ambient material are set to black, while the specular material is set to white color. The shininess factor is a float value from 0 to 128 and it was set to equal to 64. In addition, 'glEnable(GL_COLOR_MATERIAL)' function must be called.

```
GLfloat qaBlack[] = {0.0, 0.0, 0.0, 1.0};
GLfloat qaWhite[] = {1.0, 1.0, 1.0, 1.0};
GLfloat low_sh[] = {64.0};
glMaterialfv(GL_FRONT, GL_AMBIENT, qaBlack);
glMaterialfv(GL_FRONT, GL_DIFFUSE, qaBlack);
glMaterialfv(GL_FRONT, GL_SPECULAR, qaWhite);
glMaterialfv(GL_FRONT, GL_SHININESS, low_sh);
glEnable(GL_COLOR_MATERIAL);
```

9.2.3 Model Transformation

1. Scaling: 'glScale(x,y,z)' function is used to apply a nonuniform scaling along the x, y, and z axis. These parameters are use to control zoom in and zoom out the shape. as shown in the following code.

```
glScalef(zoomValue, zoomValue, zoomValue);
```

   The zoomValue is controlled by using the scroll mouse wheel as shown in the following event:

```
void MyGLWidget::wheelEvent(QWheelEvent *event)
{

    QPoint numDegrees = event->angleDelta() / 8;
    if (!numDegrees.isNull()) {
        if(numDegrees.y()>0)
            zoomValue = zoomValue + 0.05;
        else
            zoomValue = zoomValue - 0.05;
        updateGL();
    }
}
```

2. Translation: The 'glTranslate(x,y,z)' function translates the shape in x, y, and z directions as shown in the following code.

```
glTranslatef(xPos, yPos, -10.0);
```

   xPos and yPos values are controlled by dragging the object by the left mouse button as shown in the following event:

```
void MyGLWidget::mouseMoveEvent(QMouseEvent *event){
    int dx = event->x() - lastPos.x();
    int dy = event->y() - lastPos.y();
```

```
        if (event->buttons() & Qt::LeftButton) {
            xPos = xPos + double(dx)/750;
            yPos = yPos - double(dy)/750;
            updateGL();

        } else if (event->buttons() & Qt::RightButton) {
            setXRotation(xRot + 8 * dy);
            setYRotation(yRot + 8 * dx);
        }
        lastPos = event->pos();
    }
```

3. Rotation: The 'glRotatef(angel,x,y,z)' function is used to rotate the object. It reads four parameters the first parameter is rotation angle, and the other three parameters are the rotation axis x, y and z as show in the following code.The rotation function angles xRot, yRot and zRot are controlled by dragging the object by the right mouse button as in mouseMoveEvent() function or by the rotation sliders x,y and z in the GUI.

```
glRotatef(xRot / 16.0, 1.0, 0.0, 0.0);
glRotatef(yRot / 16.0, 0.0, 1.0, 0.0);
glRotatef(zRot / 16.0, 0.0, 0.0, 1.0);
```

9.3 Rendering Model

OpenGL supports rendering for three different types of geometric primitives which are points, line segments, and closed polygons. Each type of primitives is specified through vertices. In addition, OpenGL provides additional attributes for each vertex such as the position, color, normal and texture. It can support 10 different types of geometric primitives as shown in Figure 27.

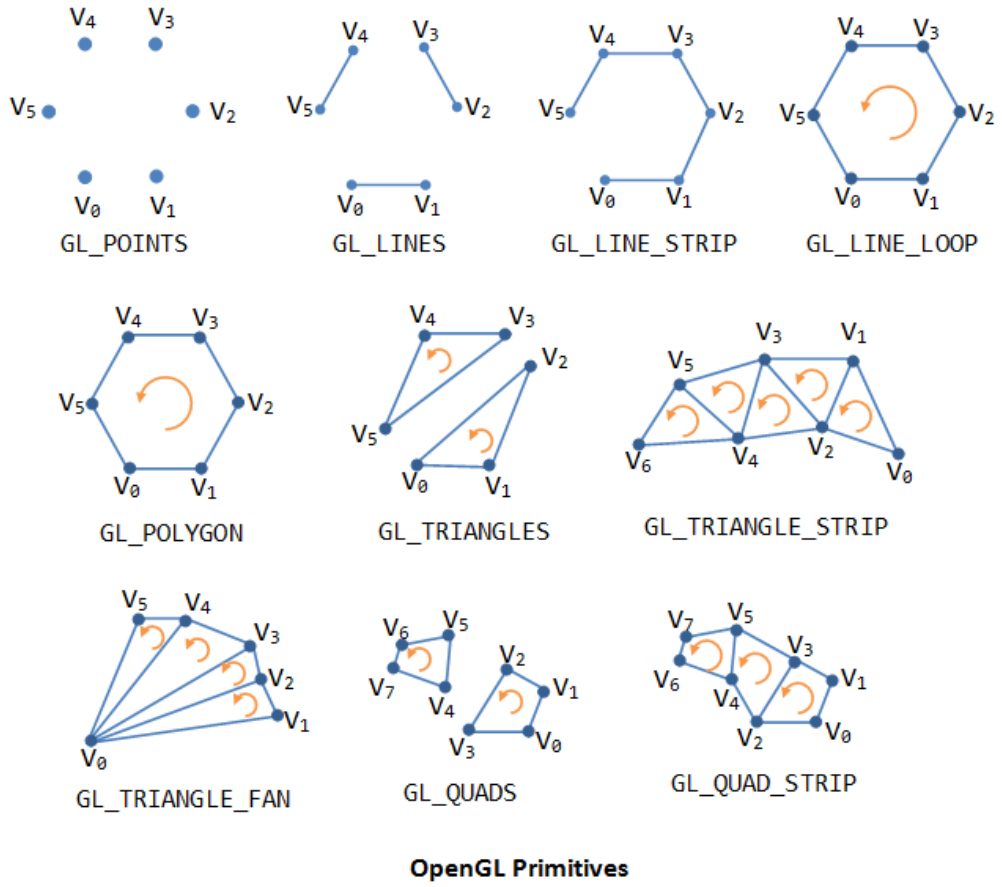


Figure 27: OpenGL Geometric Primitives Rendering Techniques

For rendering object in the case of Flat shading, the 'glPolygonMode' function is firstly set to 'GL_FILL' in order to fill each fragment. Then the required RGB color for the entire object is defined through 'glColor3f' function. Next the normal vector value is loaded the for each face using by 'glNormal3f' function. Finally, The mesh is rendered by using 'glBegin' function with 'GL_TRIANGLES' type for building the fragment from the three vertices for each face as shown in the following code.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glColor3f(RObjColor,GObjColor,BObjColor);
for(int i=0;i<read_number_of_faces;i++){
    glNormal3f(Fa_NV[i][0],Fa_NV[i][1],Fa_NV[i][2]);
    glBegin(GL_TRIANGLES);
        glVertex3f(Ve[Fa[i][0]][0],Ve[Fa[i][0]][1],Ve[Fa[i][0]][2]);
        glVertex3f(Ve[Fa[i][1]][0],Ve[Fa[i][1]][1],Ve[Fa[i][1]][2]);
        glVertex3f(Ve[Fa[i][2]][0],Ve[Fa[i][2]][1],Ve[Fa[i][2]][2]);
    glEnd();
}
```

The rendering technique for Gouraud shading differs from the Flat shading. In this technique the normal vector for each x, y and z vertex is called before each vertex by using using 'glNormal3f' function as shown in the following.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glColor3f(RObjColor,GObjColor,BObjColor);
for(int i=0;i<read_number_of_faces;i++){
    glBegin(GL_TRIANGLES);
        glNormal3f(Ve_NV[Fa[i][0]][0],Ve_NV[Fa[i][0]][1]
                ,Ve_NV[Fa[i][0]][2]);
        glVertex3f(Ve[Fa[i][0]][0],Ve[Fa[i][0]][1],Ve[Fa[i][0]][2]);
        glNormal3f(Ve_NV[Fa[i][1]][0],Ve_NV[Fa[i][1]][1]
                ,Ve_NV[Fa[i][1]][2]);
        glVertex3f(Ve[Fa[i][1]][0],Ve[Fa[i][1]][1],Ve[Fa[i][1]][2]);
        glNormal3f(Ve_NV[Fa[i][2]][0],Ve_NV[Fa[i][2]][1]
                ,Ve_NV[Fa[i][2]][2]);
        glVertex3f(Ve[Fa[i][2]][0],Ve[Fa[i][2]][1],Ve[Fa[i][2]][2]);
    glEnd();
}
```

In order to rendering the black mesh lines over the shape in the case of checking the mesh line box, the 'glPolygonMode' function is firstly set to 'GL_LINE'. This case does not require a normal vectors.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

```
glColor3f(RObjColor,GObjColor,BObjColor);
for(int i=0;i<read_number_of_faces;i++){
    glBegin(GL_TRIANGLES);
        glVertex3f(Ve[Fa[i][0]][0],Ve[Fa[i][0]][1],Ve[Fa[i][0]][2]);
        glVertex3f(Ve[Fa[i][1]][0],Ve[Fa[i][1]][1],Ve[Fa[i][1]][2]);
        glVertex3f(Ve[Fa[i][2]][0],Ve[Fa[i][2]][1],Ve[Fa[i][2]][2]);
    glEnd();
}
```

In this project, the detected Harris interest points are rendered. In the following code the points size colors are loaded by 'glPointSize' and 'glColor3f' function. Then the points are rendered by using 'GL_POINTS' option as shown in the following code.

```
    glPointSize(PointSize);
    glColor3f(RPointColor, GPointColor, BPointColor);
    for (int i = 0; i < points_size; i++)
    {
        glBegin (GL_POINTS);
            glVertex3f (P[i][0], P[i][1], P[i][2]);
        glEnd ();
    }
```

10 Signals and Slots

Signals and slots is considered a communication technique for interfacing between GUI objects. In this project, we added GUI objects for selecting Harris parameters, for controlling the transformation and color graphical OpenGL objects, and for displaying the interest point and neighbour rings. Figure 28 shows the signal directions between GLwidget and the GUI objects.

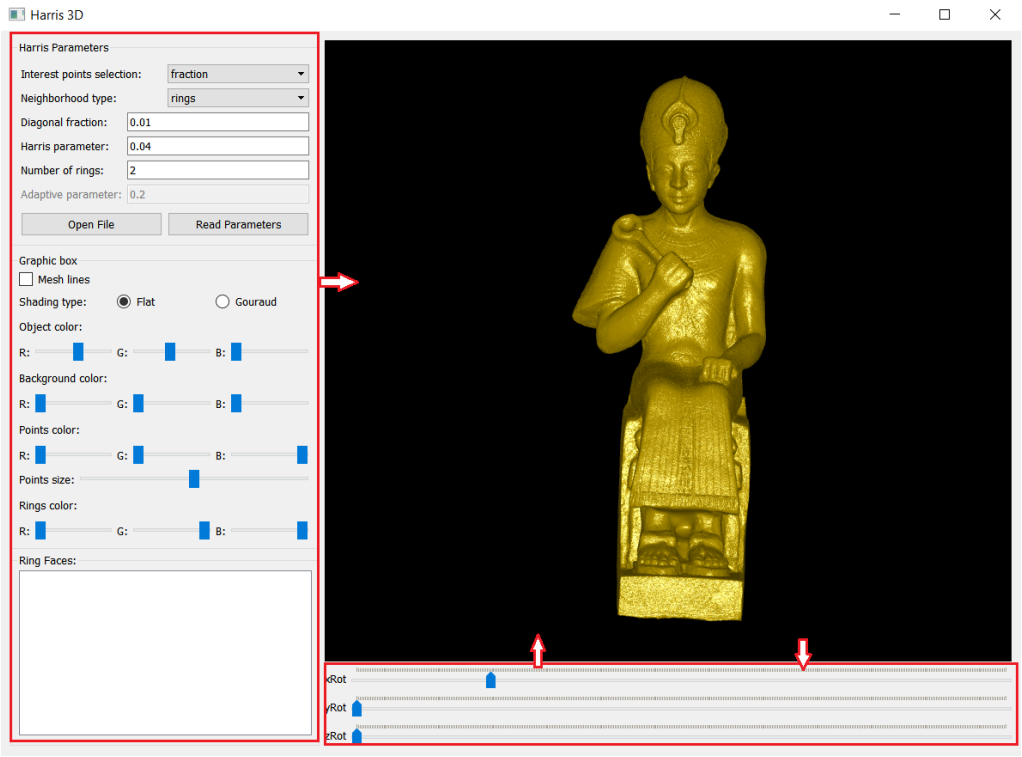


Figure 28: Signal Directions Between GUI Objects

10.1 Harris Parameters

Changing the parameters of the Combo Box and Line Edit boxes emit a signals to 'MyGLWidget' class to modify the input parameters of Harris interest points.

Signal (Combo Box)	Slot (OpenGL Widget)
currentIndexChanged(int)	getIPScomboBoxValue(int IPS_val)
currentIndexChanged(int)	getNTcomboBoxValue(int NT_val)

Signal (Line Edit)	Slot (OpenGL Widget)
textEdited(QString)	getDFlineEditValue(QString DF_val)
textEdited(QString)	getRNlineEditValue(QString RN_val)
textEdited(QString)	getHPlineEditValue(QString HP_val)
textEdited(QString)	getAPlineEditValue(QString AP_val)

In the case of pressing 'Open File' button, a signal emitted to from the button and received by a 'loadOFFFile()' slot in 'MyGLWidget' class. This function calls another functions for loading OFF file, extracting 3D model parameters and rendering objects. However, pressing 'Read Parameters' button emits a signal to 'loadOFFFile() ' slot in 'MyGLWidget'. This slot checks if 3D model is loaded. Then it checks if the Harris input parameters are correctly inserted. Next, it calls 'cal_interest_points' function with Harris input parameters.Finally it loads the Harris interest points in 'Ring Faces' list box, and it calls a function for rendering the Harris interest points.

Signal (Push Button)	Slot (OpenGL Widget)
clicked()	loadOFFFile()
clicked()	calculateHarrisPoints()

10.2 Graphic Box

If the check box toggled (checked or unchecked), it emits a signal to MLchecked(bool ML_chk) 'MyGLWidget' class to enable or disable black mesh line rendering as shown in Figure 29.

Signal (Check Box)	Slot (OpenGL Widget)
toggled(bool)	MLchecked(bool ML_chk)

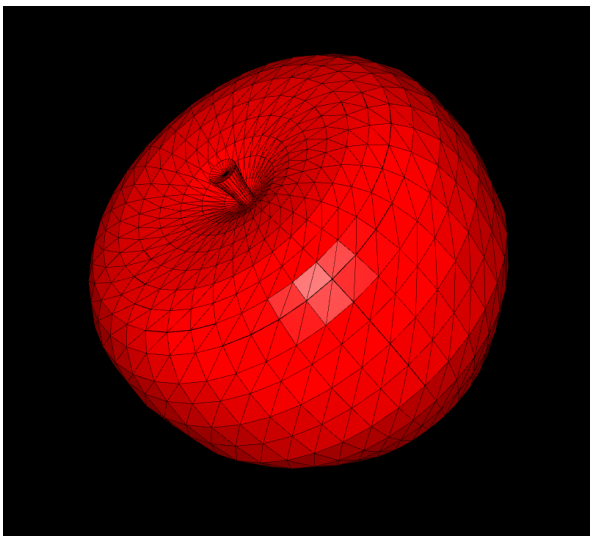


Figure 29: Checking Mesh Lines

Selecting between Flat and Gouraud radio buttons emit signals to SFchecked(), and SGchecked() slots respectively. These slots are used for changing the shade rendering rendering type between Flat and Gouraud as shown in Figures 30 and 31 .

Signal (Radio Button)	Slot (OpenGL Widget)
clicked()	SFchecked()
clicked()	SGchecked()

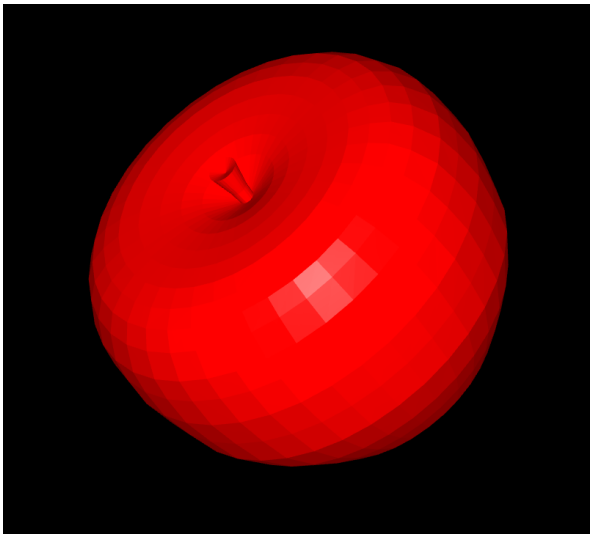


Figure 30: Flat Shading

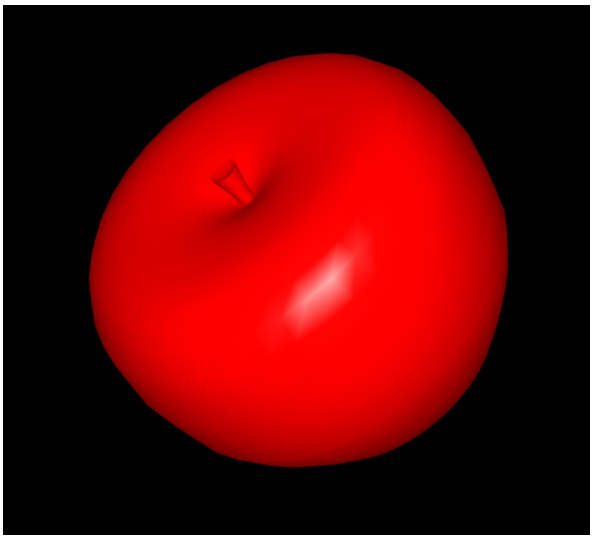


Figure 31: Gouraud Shading

Changing the color sliders in Graphic Box emit signals. These signal are received by different slots in 'MyGLWidget', and is changes 3D object, Background and Harris interest points color.

However, changing the point size slide emits a signal to change the size of Harris interest points. All these slots call a function to update GL by rendering the shapes agin with new colors.

Signal (Horizontal Slider)	Slot (OpenGL Widget)
valueChanged(int)	changeObjRColor(int R_obj)
valueChanged(int)	changeObjGColor(int G_obj)
valueChanged(int)	changeObjBColor(int B_obj)
valueChanged(int)	changeBackRColor(int R_back)
valueChanged(int)	changeBackGColor(int G_back)
valueChanged(int)	changeBackBColor(int B_back)
valueChanged(int)	changePointRColor(int R_point)
valueChanged(int)	changePointGColor(int G_point)
valueChanged(int)	changePointBColor(int B_point)
valueChanged(int)	changePointSize(int S_point)
valueChanged(int)	changeRingRColor(int R_ring)
valueChanged(int)	changeRingGColor(int G_ring)
valueChanged(int)	changeRingBColor(int B_ring)

10.3 Rings Faces

By double clicking each interest point parameter in 'Ring Faces' box it emits a signal with this point index to 'getHVlistWidget-Value(QModelIndex HV_Val)' slot in to calls a function 'get_faces' which returns the faces index for neighbour Rings, and then it calls a function to render these face.

Signal	Slot (OpenGL Widget)
doubleClicked(QModelindex)	getHVlistWidgetValue(QModelIndex HV_Val)

10.4 Object Rotation

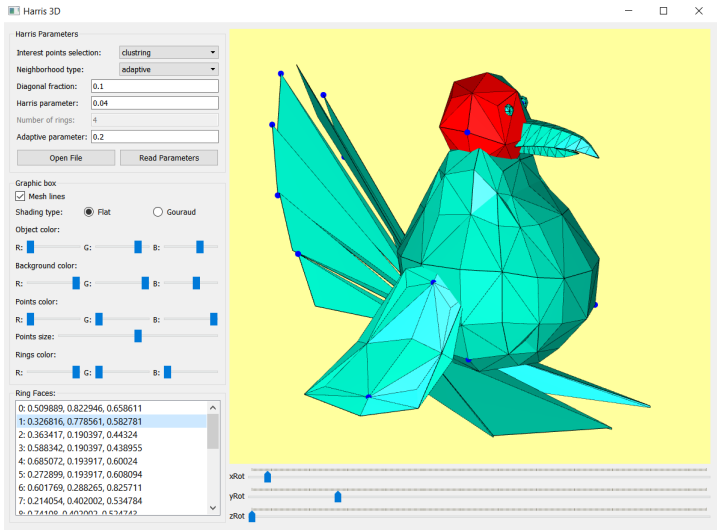
In the case of changing 'xRot', 'yRot' and 'zRot' sliders emit a signals to there corresponding slots in the followin table in 'MyGLWidget' class. Theses signals change rotates the 3D object around x, y and z axis.

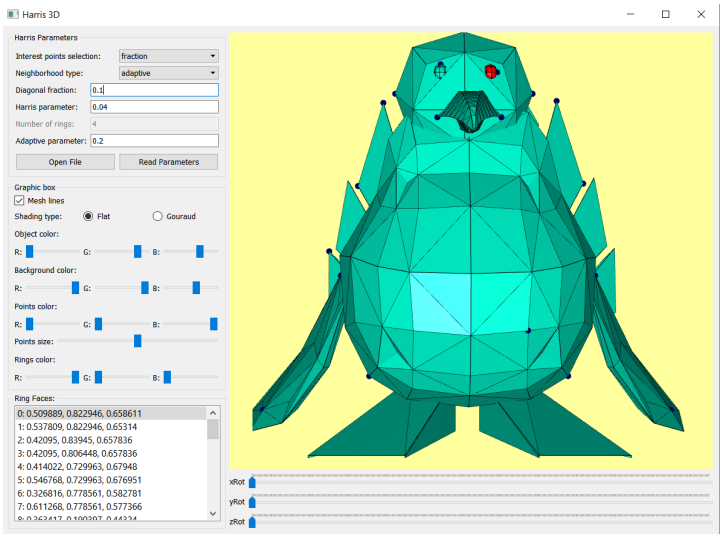
Signal (Horizontal Slider)	Slot (OpenGL Widget)
valueChanged(int)	setXRotation(int angle)
valueChanged(int)	setYRotation(int angle)
valueChanged(int)	setZRotation(int angle)

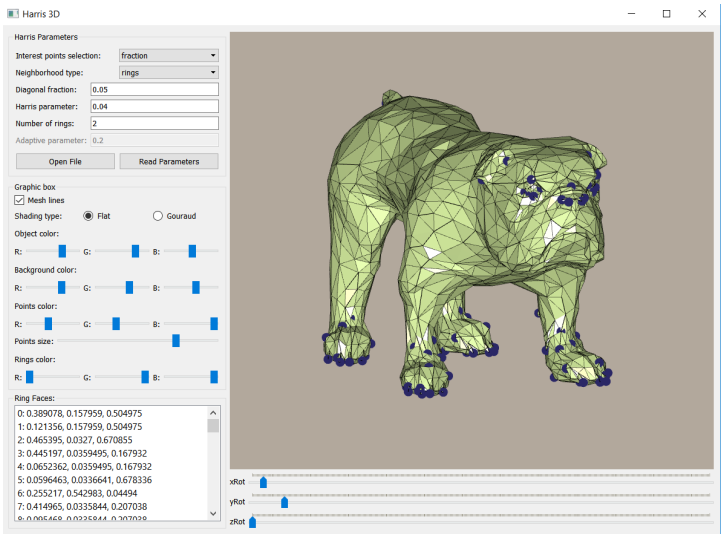
However, if the 3D object rotated by pressing the right button and dragging it around x, y and z axis, a group of signal are emitted back to change the slider pointer location.

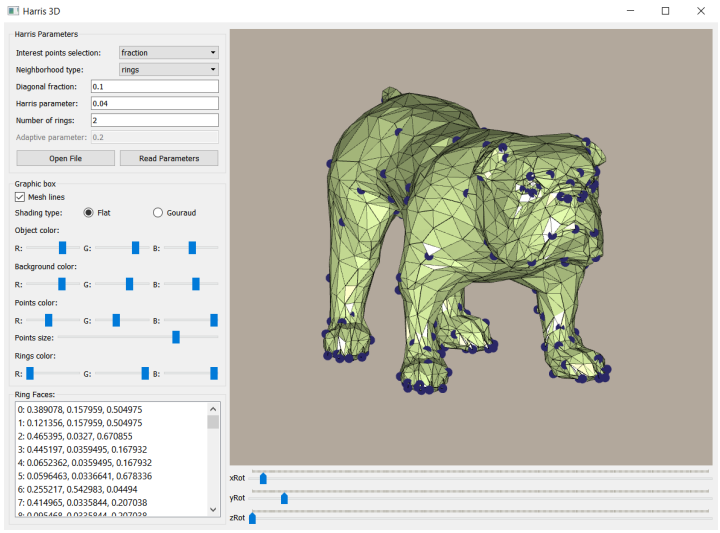
Signal (OpenGL Widget)	Slot (Horizontal Slider)
xRotationChanged(int angle);	setValue(int)
yRotationChanged(int angle);	setValue(int)
zRotationChanged(int angle);	setValue(int)

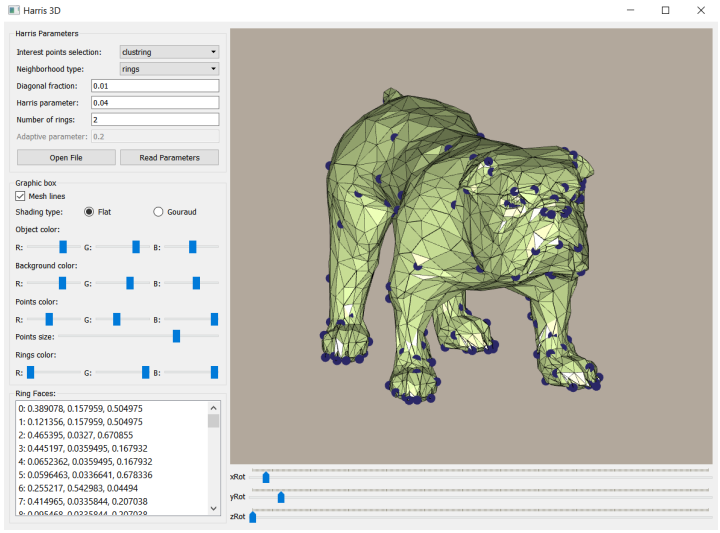
11 Some Result Images

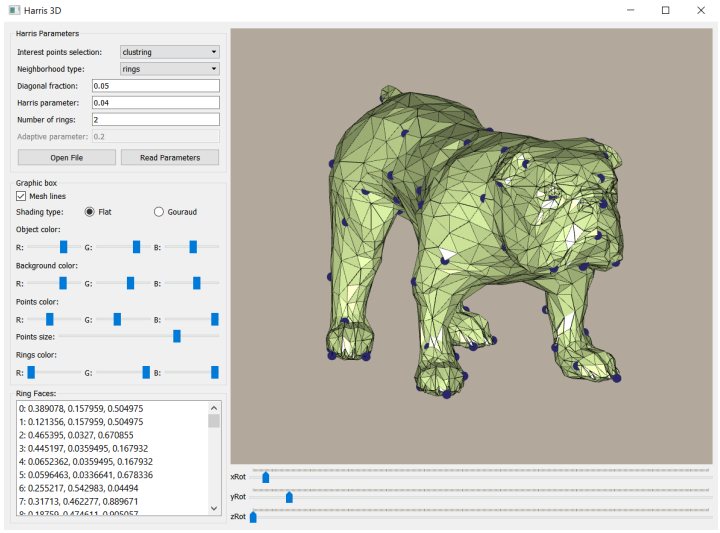


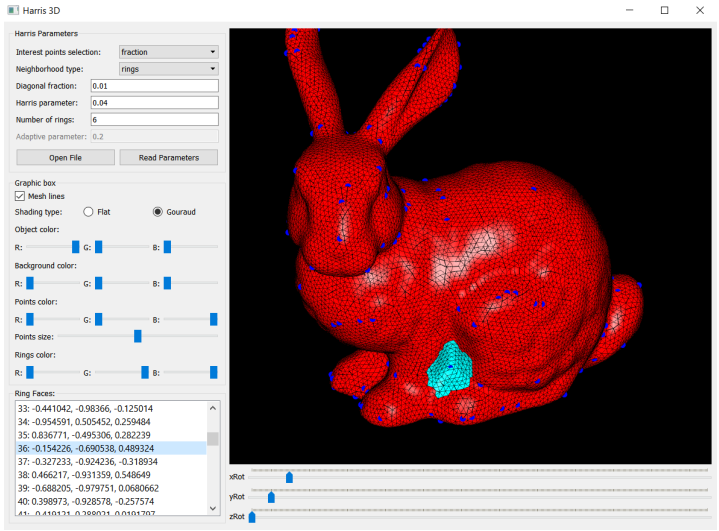



Harris 3D
Harris Parameters
Interest points selection:
fraction
Neighborhood type:
rings
Diagonal fraction:
0.05
Harris parameter:
0.04
Number of rings:
2
Adaptive parameter:
0.2
Open File
Read Parameters
Graphic box
Mesh lines
Shading type:
Flat
Gouraud
Object color:
Background color:
Points color:
Points size:
Rings color:
Ring Faces:
0: 0.389078, 0.157959, 0.504975
1: 0.121356, 0.157959, 0.504975
2: 0.465395, 0.0327, 0.670855
3: 0.445197, 0.0359495, 0.167932
4: 0.0652362, 0.0359495, 0.167932
5: 0.0596463, 0.0336641, 0.678336
6: 0.255217, 0.542983, 0.04494
7: 0.414965, 0.0335844, 0.207038
xRot
yRot
zRot


Harris 3D
Harris Parameters
Interest points selection:
fraction
Neighborhood type:
rings
Diagonal fraction:
0.1
Harris parameter:
0.04
Number of rings:
2
Adaptive parameter:
0.2
Open File
Read Parameters
Graphic box
Mesh lines
Shading type:
Flat
Gouraud
Object color:
Background color:
Points color:
Points size:
Rings color:
Ring Faces:
0: 0.389078, 0.157959, 0.504975
1: 0.121356, 0.157959, 0.504975
2: 0.465395, 0.0327, 0.670855
3: 0.445197, 0.0359495, 0.167932
4: 0.0652362, 0.0359495, 0.167932
5: 0.0596463, 0.0336641, 0.678336
6: 0.255217, 0.542983, 0.04494
7: 0.414965, 0.0335844, 0.207038
xRot
yRot
zRot


Harris 3D
Harris Parameters
Interest points selection:
clustring
Neighborhood type:
rings
Diagonal fraction:
0.01
Harris parameter:
0.04
Number of rings:
2
Adaptive parameter:
0.2
Open File
Read Parameters
Graphic box
Mesh lines
Shading type:
Flat
Gouraud
Object color:
Background color:
Points color:
Points size:
Rings color:
Ring Faces:
0: 0.389078, 0.157959, 0.504975
1: 0.121356, 0.157959, 0.504975
2: 0.465395, 0.0327, 0.670855
3: 0.445197, 0.0359495, 0.167932
4: 0.0652362, 0.0359495, 0.167932
5: 0.0596463, 0.0336641, 0.678336
6: 0.255217, 0.542983, 0.04494
7: 0.414965, 0.0335844, 0.207038
xRot
yRot
zRot


Harris 3D
Harris Parameters
Interest points selection:
clustring
Neighborhood type:
rings
Diagonal fraction:
0.05
Harris parameter:
0.04
Number of rings:
2
Adaptive parameter:
0.2
Open File
Read Parameters
Graphic box
Mesh lines
Shading type:
Flat
Gouraud
Object color:
Background color:
Points color:
Points size:
Rings color:
Ring Faces:
0: 0.389078, 0.157959, 0.504975
1: 0.121356, 0.157959, 0.504975
2: 0.465395, 0.0327, 0.670855
3: 0.445197, 0.0359495, 0.167932
4: 0.0652362, 0.0359495, 0.167932
5: 0.0596463, 0.0336641, 0.678336
6: 0.255217, 0.542983, 0.04494
7: 0.31713, 0.462277, 0.889671
xRot
yRot
zRot


Harris 3D
Harris Parameters
Interest points selection:
fraction
Neighborhood type:
rings
Diagonal fraction:
0.01
Harris parameter:
0.04
Number of rings:
6
Adaptive parameter:
0.2
Open File
Read Parameters
Graphic box
Mesh lines
Shading type:
Flat
Gouraud
Object color:
R:
G:
B:
Background color:
Points color:
Points size:
Rings color:
Ring Faces:
33: -0.441042, -0.98366, -0.125014
34: -0.954591, 0.505452, 0.259484
35: 0.836771, -0.495306, 0.282239
36: -0.154226, -0.690538, 0.489324
37: -0.327233, -0.924236, -0.318934
38: 0.466217, -0.931359, 0.548649
39: -0.688205, -0.979751, 0.0680662
40: 0.398973, -0.928578, -0.257574
xRot
yRot
zRot

12 References

References

[1] Ivan Sipiran: Harris 3D - Detecting interest points on 3D Meshes,
https://users.dcc.uchile.cl/ isipiran/harris3D.html

[2] Trello Boards For Project Management ,
https://trello.com/zohaibsalahuddin/boards

[3] Github for For Project Management ,
https://github.com/zohaibsalahuddin/harris3D/tree/zohaib

[4] Eigen Library ,
http://eigen.tuxfamily.org/index.php?title=Main_Page

[5] Open GL Lighting,
https://learnopengl.com/Lighting/Basic-Lighting

[6] OpenGL Basics,
http://www.ntu.edu.sg/home/ehchua/programming/opengl/cg_basicstheory.html#zz-5.2

[7] OpenGL Material,
https://www.khronos.org/registry/OpenGL-Refpages/es1.1/xhtml/glMaterial.xml

[8] OpenGL Transform,
http://www.songho.ca/opengl/gl_transform.html

[9] OpenGL Shading,
http://graphics.cs.cmu.edu/nsp/course/15-462/Spring04/slides/08-shading.pdf

[10] OpenGL Example,
https://www.bogotobogo.com/Qt/Qt5_OpenGL_QGLWidget.php