# COL781 Assignment 2: Mesh Processing

Garvit Dhawan - 2020CS50425
Harshit Goyal - 2020MT10806

## Mesh Structure

The mesh is defined as a vector of vertices and a vector of triangles, where each vertex stores its position, its normal, and a vector containing indices of all the triangles incident on that vector. The triangle stores the indices of the vertices making that triangle.

```
// Define a structure for vertex
struct Vertex
{
  glm::vec3 position;
  glm::vec3 normal;
  std::vector<int> adjacentTriangles;
};

// Define a structure for triangle
struct Triangle
{
  int vertices[3];
};

// Define a mesh class
class Mesh
{
private:
  std::vector<Vertex> vertices;
  std::vector<Triangle> triangles;
```

The mesh supports the following functions:

- AddVertex(position, normal) - adds a vertex with given position, normal to the mesh, returns the index of the added vertex
- AddTriangle(vertexIndex1, vertexIndex2, vertexIndex3) - adds a triangle with given vertex indices to the mesh
- getNeighboringVertices(vertexIndex) - returns a vector of all the vertices neighboring the given vertex in the mesh
- getNeighboringTriangles(vertexIndex) - returns a vector of all triangles incident on that vertex in the mesh
- render() - renders and views the mesh using the Viewer.
- smoothMesh(lambda, iterations) - Smooths the mesh using the umbrella operator for the given $\lambda$, number of iterations

- taubinSmoothMesh(lambda, nu, iterations) - Smooths the mesh using Taubin smoothing for the given λ, μ, number of iterations
- edgeFlip(vertexIndex1, vertexIndex2) - performs the edge flip operation
- edgeSplit(vertexIndex1, vertexIndex2) - performs the edge split operation
- edgeCollapse(vertexIndex1, vertexIndex2) - performs the edge collapse operation
- edgeExists(vertexIndex1, vertexIndex2) - checks if an edge exists between the given vertices
- isValid()- checks if the mesh is valid or not

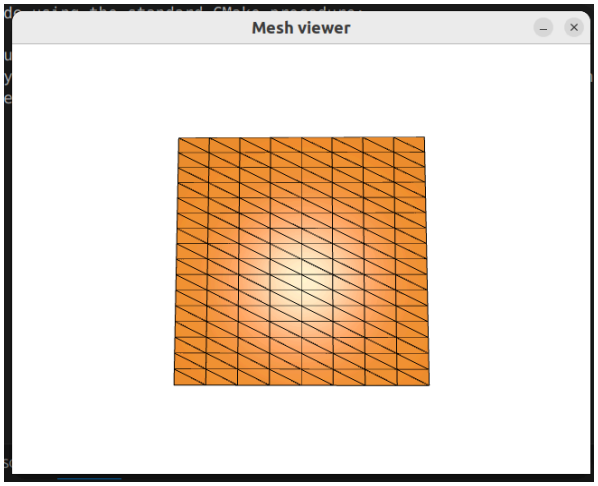
# Simple Meshes

The mesh_example1.cpp file contains the functionality to create the meshes for a Unit Square Grid mesh, and a Sphere mesh.
To run the example, build the project. Then from the project root directory, run the following command:

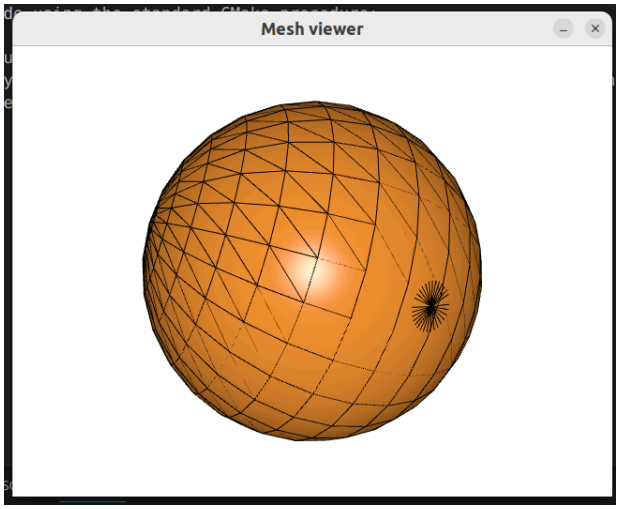
build/mesh_example1 <t> <m> <n>
For eg: build/mesh_example1 1 16 16

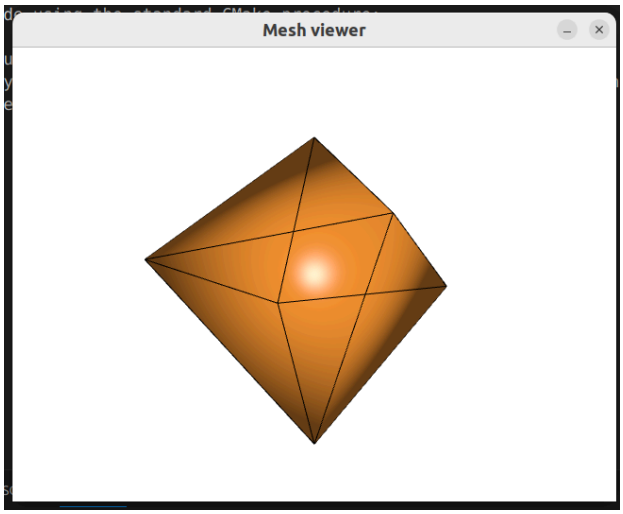
Here,
the parameter **t** should be set to **0** to create a Unit Square Mesh, and set to **1** to create a Sphere Mesh of radius 0.6;
The parameter **m** will be the **number of rows** for the square grid mesh , and the **number of slices** for the sphere mesh;
The parameter **n** will be the **number of columns** for the square grid mesh, and the **number of stacks** for the sphere mesh;

Here is what the unit square grid(tilted to show the mesh) looks like for m=16, n=8:




And here is what the sphere grid looks like for m=32, n=16 and m=4, n=2:




# OBJ Meshes

The parser takes the input from OBJ files and converts them to meshes. If the vertex normals are present, we keep them. Otherwise, we set the normals using the formula by Nelson Max:

$$\sum_{i=0}^{n-1} \frac{V_i \times V_{i+1}}{|V_i|^2 |V_{i+1}|^2} = cN.$$

Here,

n is the number of triangles incident on the given vertex v
For the ith triangle,
Vi is the vector from current vertex to the next vertex of the triangle (in anti-clockwise order)
Vi+1 is the vector from the current vertex to the next to next vertex of the triangle (in anti-clockwise order).
This order for the cross product ensures that the direction of the normal is correct.

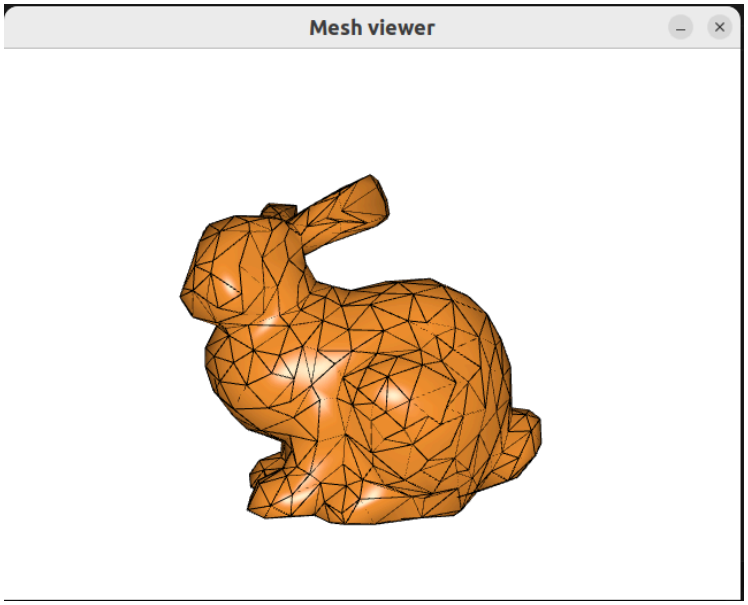
We can then use the view the mesh using mesh.render();

To parse any OBJ file, we can use the following command from the project root directory after building the project:
build/parser_example <filename>
For eg: build/parser_example meshes/teapot.obj

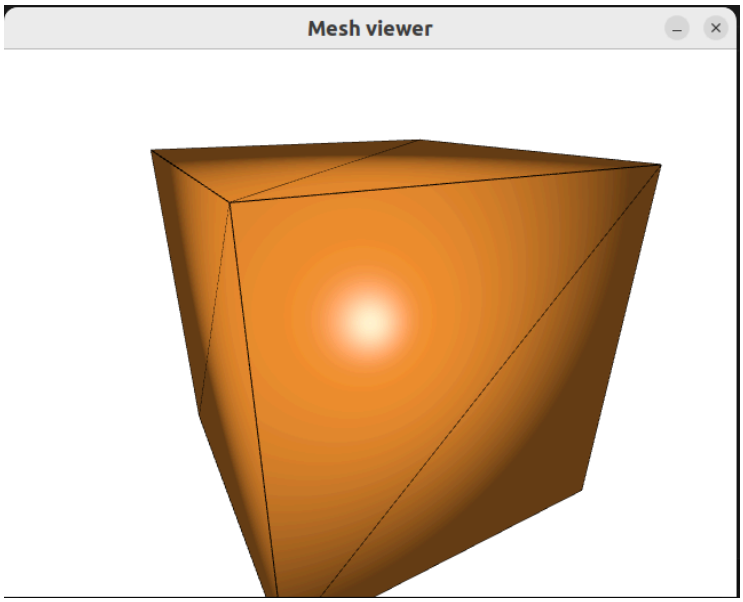
(If the filename is correct, an empty mesh will be shown with the error msg: “Cannot Open File <filename>. Please check filename”)

This is how the given meshes look like:

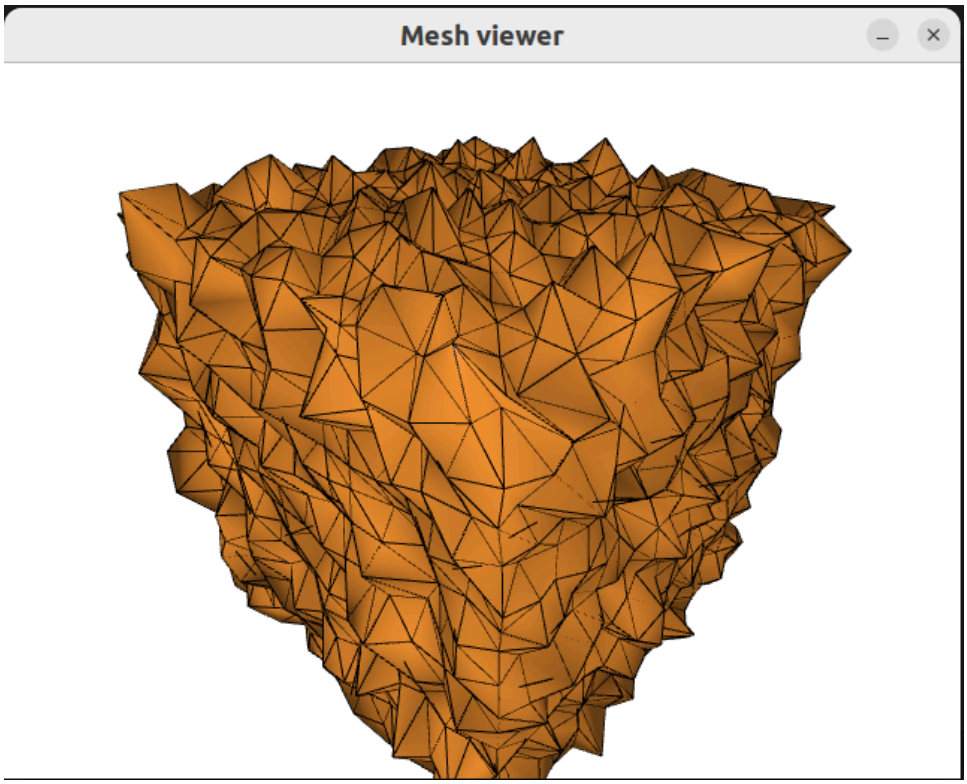
meshes/bunny-1k.obj



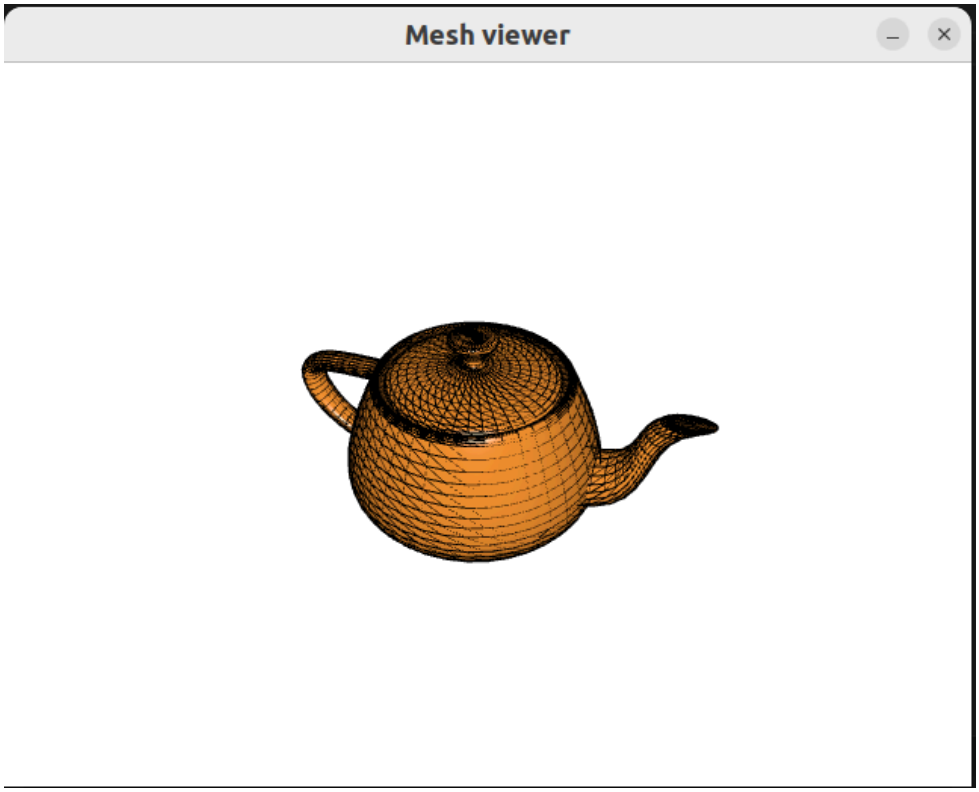

meshes/cube.obj (after some rotation)




meshes/noisycube.obj (after some rotation)




meshes/teapot.obj (after some rotation)




# Mesh Smoothing

The mesh smoothing function is defined in mesh.cpp (described above). The naive smoothing can be tested using mesh_smooth_example1.cpp, while Taubin smoothing can be tested by using mesh_smooth_example2.cpp. To run these, put the following command in the terminal after building the project from the project root directory:

build/mesh_smooth_example1 <filename> <lambda> <iterations>

Here,
filename is the name of the OBJ file to be parsed into a mesh
lambda is the factor λ
Iterations is the number of iterations

And,

build/mesh_smooth_example2 <filename> <lambda> <nu> <iterations>

Here,

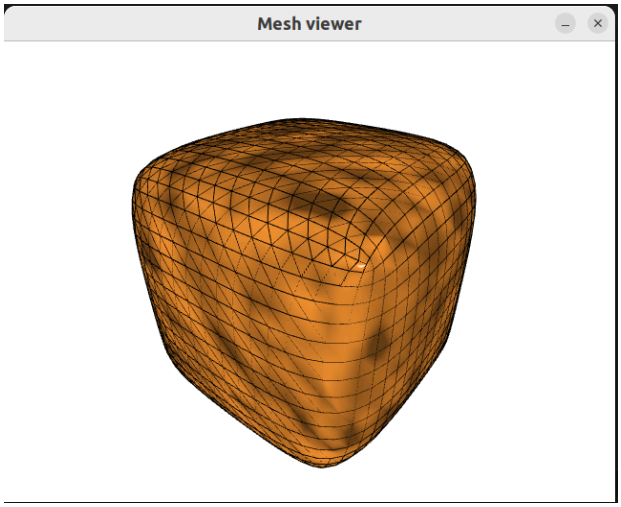
filename is the name of the OBJ file to be parsed into a mesh
lambda is the factor λ
nu is the negative factor μ
Iterations is the number of iterations

For eg:

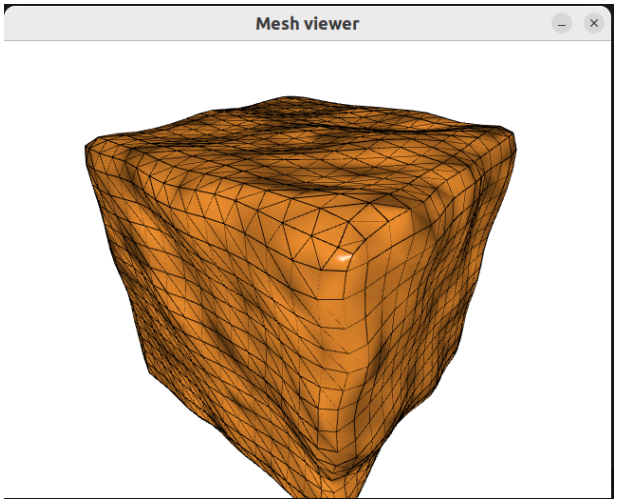
build/mesh_smooth_example1 meshes/noisycube.obj 0.33 50
build/mesh_smooth_example2 meshes/noisycube.obj 0.33 -0.34 50

This is how the noisy cube mesh looks like after 50 iterations on lambda = 0.33 and nu = -0.34:

Naive smoothing




Taubin Smoothing




We can observe that the naive smoothing creates a more smooth mesh, but shrinks it (the shrinking is significant in higher number of iterations), while the taubin smoothing preserves size and rough shape of the original mesh.

# Mesh Editing Operations

We implemented the flip, split and collapse operations in mesh.cpp (described above).

For splitting, we add a new vertex to the mesh with position being the midpoint of the edge, and the normal for the new vertex is calculated using the Nelson Max formula (same as above).

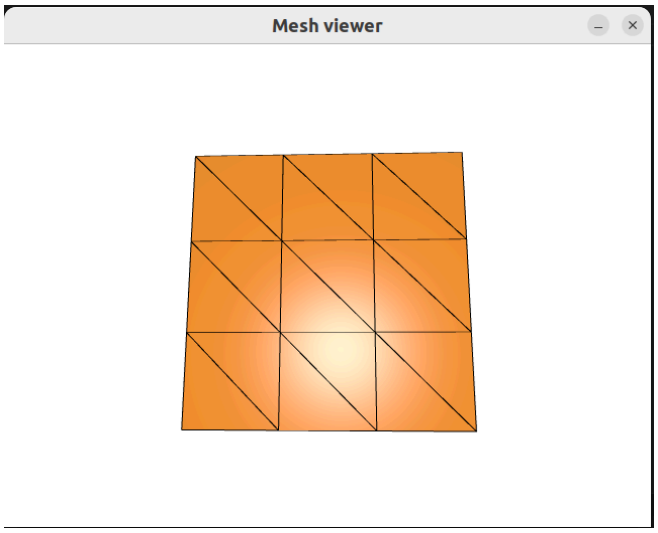
For collapsing, we delete one vertex of the edge from the mesh and shift the other vertex of the edge to the midpoint of the edge, and update the normal of the vertex to be the average of the normals of the endpoints of the edge.

To test these operations, we can run the following command from the project root directory after building the project:

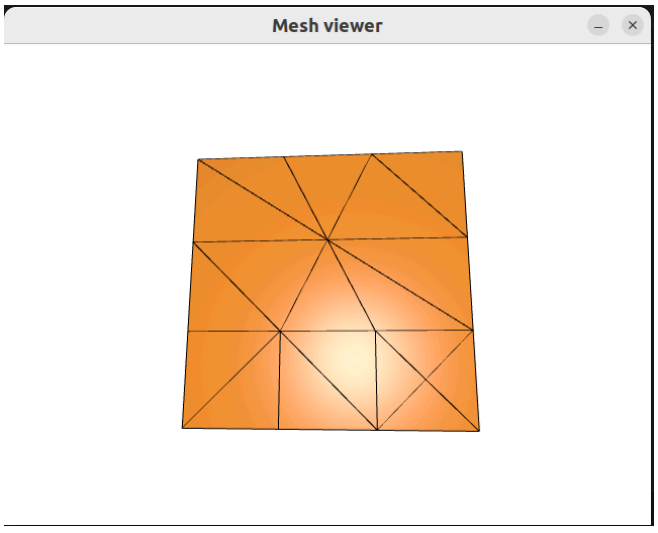
build/mesh_example2

This shows the result of all 3 of these operations on a 3 by 3 unit square grid mesh.

The original mesh:




The mesh after operations:



Here, the vertices are numbered from 0 to 15 in the original mesh starting from the bottom left corner looking like this:

12 13 14 15

8 9 10 11

4 5 6 7

0 1 2 3

We applied edge flip on the edge (1,4), edge split on the edge (3,6) and edge collapse on the edge (9,10). We can observe that all operations are successful and maintain the mesh validity.

# Mesh Subdivision

This part could not be done due to lack of time.