Middle East Technical University | Department of Computer Engineering

CENG 213

Data Structures

Fall 2025-2026

Programming Assignment 3

1 Objectives

In this programming assignment, you will implement an interface for parsing 3D meshes and compute important properties of the mesh as well as use heaps to orginize those properties. While doing so, you will learn about meshes, which are specialized graphs, as well as mesh traversal and navigation.

Keywords: C++, Data Structures, Graph, Heap, Mesh

2 Mesh

Mesh is a subcategory of undirected graph, that represents a 3D object in space. Main differences between a graph and a mesh are

- Each mesh vertex has a 3D coordinate associated with it, and placed at that exact point in 3D space.
- In addition to edges between vertices, the polygons (triangles for our and most cases) created by the edges are called surfaces and are also of great importance. They are so important that most of the time, we describe meshes by listing the surfaces, not edges.
- For 2-manifold non-intersecting watertight[1] meshes (all meshes for our case), surfaces do not intersect each other and an edge is shared by exactly 2 surfaces.

With these in mind, we will still be working with graphs, with vertices and edges.

2.1 OFF File Format

We will read meshes from a very simple and widely used file format called OFF (Object File Format). Each OFF file is made out of three things, the header, the vertices, and the surfaces.

Header of the file starts with line denoting the type of the off file. We will use only the most basic form of OFF specification, meaning the first line will always say "OFF". This line is followed by 3 integers, informing us about the number of vertices, surfaces and edges in the mesh, in that order. Because the mesh is later described in terms of surfaces and not edges, the number of edges in the header is considered non-essential information which can be derived after reading the mesh, and it is commonly set to 0.

[1] You do not need to know about these terms, they are specified here to prevent misinformation by generalization

Following the header, we arrive at the vertices, which contains a number of vertices equal to the amount declared in the header, each vertex residing at separate lines. Each line consists of three floating point values, describing the xyz coordinate of the vertex. Also the index of each vertex is determined by their order in the file, starting from 0.

In the end of the file, surfaces are stored. Just like vertices, their number equals to what is declared in the header, and each surface resides on a different line. First integer of each surface line describes how many vertices constructs that surface (always 3 for our case), followed the number of integers equal to the first number, each denoting the index of a different vertex that makes up that surface. Same as the vertices, the index of each surface is determined by their order in the file, starting from 0.

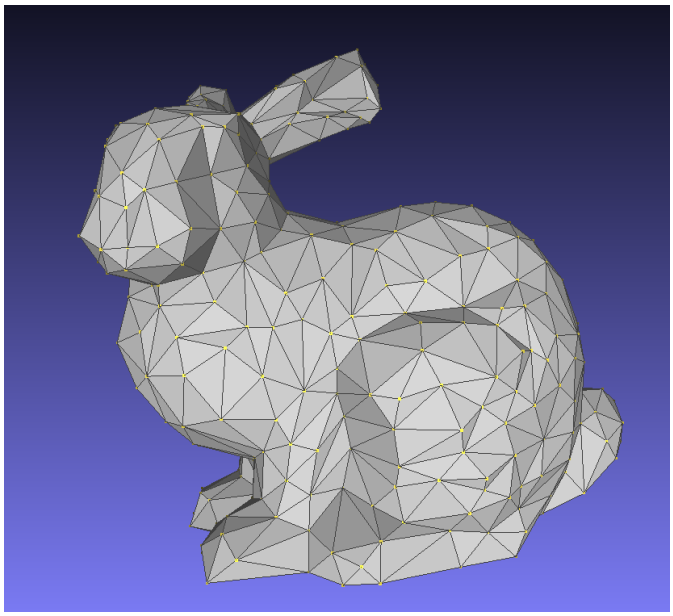

Figure 1: Stanford Bunny captured using MeshLab

A snippet taken from an off file looks like this:

```
OFF
502 1000 0
-1.196301 1.429399 -0.1727576
-0.4162832 -0.2702349 -0.5697511
-0.966646 -0.2865776 0.1722701
-1.376502 2.03476 -0.9334655
0.9088099 0.5777116 0.01478398
0.6233937 -0.2057004 0.5461224
...
3 342 442 489
3 467 481 144
3 261 260 421
3 115 373 40
3 31 335 72
3 384 286 220
3 44 133 32
3 48 12 350
3 300 470 289
3 207 310 451
```

...

To visualize the 3D shape that describes an OFF file, a recommend free software such as MeshLab and Blender, both being available in Windows and Linux as well as inek computers in the labs. For example, visualization of the mesh where the above OFF file snippet is taken from can be seen in Figure 1.

3 Implementation

3.1 Basic Structures

Basic data structures are defined in `DataStructures.h`.

3.1.1 `Vertex`

Already Implemented.
A data structure that holds vertex data. Consists of:

- `double coords[3]`: Coordinates of the vertex.
- `size_t idx`: Index of the vertex.
- `vector<size_t> vertList`: Index of the vertices that are connected to this vertex.
- `vector<size_t> triList`: Index of the triangles (surfaces) that are connected to this vertex.
- `vector<size_t> edgeList`: Index of the edges that are connected to this vertex.

You can add any additional helper function you need, but these variables are required for grading and should stay intact.

3.1.2 `Edge`

Already implemented.
A data structure that holds edge data. Consists of:

- `size_t idx`: Index of the edge.
- `size_t v1i, v2i`: Indices of vertices the edge connects to.
- `double length`: Length of the edge.

You can add any additional helper function you need, but these variables are required for grading and should stay intact.

3.1.3 `Triangle`

Already implemented.
A data structure that holds triangle (surface) data. Consists of:

- `size_t idx`: Index of the triangle.
- `size_t v1i, v2i, v3i`: Indices of vertices the triangle connects to.

You can add any additional helper function you need, but these variables are required for grading and should stay intact.

3.2 Utility Functions

Already implemented.
Utility functions are located in `Arithmetic.h`. These are used for various 3D math operations. All of them are explained in detail in `Arithmetic.h` file. Your main concern is the `inline double dist(const double* p1, const double* p2);` function.

3.3 `Mesh Class`

Hold the rest of the variables required for the processing of 3D meshes. Its decelerations are located in `Mesh.h` and it should be implemented in `Mesh.cpp`. You can add any additional helper function you need.

3.3.1 Variables

This variables are already exist inside the class, but you are responsible with initializing and clearing them.

- `vector<Vertex> verts`: The vector that holds all vertices of a mesh where the index of a vertex is equal to its position in this vector.
- `vector<Edge> edges`: The vector that holds all edges of a mesh where the index of an edge is equal to its position in this vector.
- `vector<Triangle> tris`: The vector that holds all triangles of a mesh where the index of a triangle is equal to its position in this vector.
- `vector<size_t> minHeap`: The vector that holds the min heap holding edge indices sorted by their length. Should be constructed by the `void buildHeap()` function.

3.3.2 `Mesh();`

Already implemented.
Default constructor that creates an empty mesh.

3.3.3 `Mesh(const char* filename);`

Already implemented.
Reads and parses an OFF file. Requires other functions to work properly.

3.3.4 `~Mesh();`

Already implemented.
Destructor of the class, clears all vectors.

3.3.5 `size_t addVertex(double x, double y, double z);`

Adds a vertex with the coordinates `[x, y, z]` and returns its index.

3.3.6 `size_t addTriangle(size_t v1, size_t v2, size_t v3);`

Adds a triangle with the corners `[v1, v2, v3]` and returns its index.

3.4 `void buildHeap();`

Orders `vector<size_t> minHeap` using Floyd's heap building algorithm. You should not be adding edges one by one into a heap structure, but add all edges to the `minHeap` vector and call this function at the end of reading a mesh.

3.4.1 `inline Vertex& getVertex(size_t v);`

Already implemented.
Returns a reference to the vertex with the index `v`.

3.4.2 `inline Edge& getEdge(size_t e);`

Already implemented.
Returns a reference to the edge with the index `e`.

3.4.3 `Edge& getEdge(size_t v1, size_t v2);`

Returns a reference to the edge which lays between the vertices with indices `v1` and `v2`. If no such edge exists, return the edge with the index 0.

3.4.4 `inline Triangle& getTriangle(size_t t);`

Already implemented.
Returns a reference to the triangle with the index `t`.

3.4.5 `bool isVertsNeighbor(size_t v1, size_t v2);`

Return whether vertices with indices `v1` and `v2` are connected by an edge. Should return `false` if either vertices does not exist.

3.4.6 `void printVertex(size_t v);`

Prints the coordinates of the vertex with the index `v` to the standard output as `x y z` (With trailing space). If not such vertex exists, print nothing.

3.4.7 `void printTriangle(size_t t);`

Prints the corners of the triangle with the index `t` to the standard output as `3 v1 v2 v3` (With trailing space). If not such triangle exists, print nothing.

3.4.8 `size_t getDegree(size_t v);`

Returns the degree of the vertex with the index `v`. If not such vertex exists, return 0.

3.5 `int getJumpCount(size_t v1, size_t v2);`

Returns what is the minimum number of edges that needs to be traversed (jumped) to go from the vertex `v1` to `v2`, regardless of edge length. If there is no path connecting the two vertices, return -1. If `v1` and `v2` are equal, return 0. You are recommended to use BFS.

3.5.1 `double getGeodesicDistance(size_t v1, size_t v2);`

Computes the geodesic distance between to vertices. Geodesic distance is the distance between to vertices in the mesh found using the Dijkstra's Algorithm with edge lengths as the edge weights.

3.5.2 `void updateVertex(size_t v, double x, double y, double z);`

Updates the position of the vertex with the id `v` to [`x`, `y`, `z`]. This should update the length of the edges connected to it as well as update the `minHeap` according to new lengths of the edge.

3.5.3 `Edge& getKthShortestEdge(size_t k);`

Returns a reference to the edge which is the kth sortest edge. If k is larger then the number of edges in the mesh, return the edge with the index 0. Contents of your `minHeap` should remain the same.