1.

Implement a simple mesh data structure capable of representing an all-triangle mesh and its constituent vertices. This data structure should include functions to do the following:

//Geometry

struct Point {

double x, y, z;

};

//Topology

class Vertex {

public:

// d. Return the coordinates of a given vertex.

Point point;

// b. Given a vertex , return the adjacent faces / vertices.

std::vector<int> adjVertices;

std::vector<int> adjFaces;

};

class Face {

public:

// b. Given a face, return the adjacent faces / vertices.

int vertices[3];

std::vector<int> adjFaces;

bool is\_deleted = false;

};

a.

Read/write basic .obj files; sample files are included with this specification. Code can assume mesh is all triangles. Code should be capable of reading files with smoothing group ('s'), object names ('o'), polygon groups ('g'), material libraries ('mtllib'), and use materials ('usemtl'), but does not need to provide query access to those data, nor write those data.

b. Given a vertex/face, return the adjacent faces/vertices.

Vertices: std::vector<int> adjVertices; std::vector<int> adjFaces;

Faces : std::vector<int> adjFaces;

c. Return all the vertices or faces.

class Mesh {

// c. Return all the vertices or faces.

public:

Vertices vertices;

Faces faces;

};

d. Return the coordinates of a given vertex.

class Vertex {

public:

// d. Return the coordinates of a given vertex.

Point point;

}

e. Delete a vertex or face, with optional flag to delete all connected faces (if a vertex).

// e. Delete a face

void remove() {

is\_deleted = true;

}

void remove(Vertex v, Faces& faces, bool flag = true) {

v.is\_deleted = true;

if (flag)

for (auto f : v.adjFaces)

faces[f].remove();

}

f. Construct a new face from vertices, and a new vertex from coordinates.

// f. Construct a new face from vertices, and a new vertex from coordinates

Face(int v0, int v1, int v2) {

vertices[0] = v0;

vertices[1] = v1;

vertices[2] = v2;

is\_deleted = false;

}

g. Flip the sense of a face.

// g. Flip the sense of a face.

void flip\_sense() {

std::swap(vertices[1], vertices[2]);

}

2. Write a function that returns whether all faces are consistently oriented.

// 2. Write a function that returns whether all faces are consistently oriented.

bool checkOriention(Edges& edges, int i, int j) {

Edge edge(i, j);

if (edges.count(edge) != 0)

return false;

edges.emplace(i,j);

return true;

}

bool checkOriention(const Faces& faces) {

std::unordered\_set<Edge, Hash\_of\_Edge> edges;

edges.reserve(faces.size() \* 3);

for (const auto& face : faces) {

if (!checkOriention(edges, face.vertices[0], face.vertices[1])) return false;

if (!checkOriention(edges, face.vertices[1], face.vertices[2])) return false;

if (!checkOriention(edges, face.vertices[2], face.vertices[0])) return false;

}

}

3. Write a function that returns the number of loops bounding a surface mesh.

// 3. Write a function that returns the number of loops bounding a surface mesh

void addEdges(Edges& edges, int i, int j,int& num\_edges) {

Edge edge(j, i);

if (edges.count(edge) != 0)

return;

edges.emplace(i, j);

num\_edges++;

}

typedef std::unordered\_multimap<int, Edge> VertexToEdge;

int num\_loops(const Faces& faces, int num\_vertices, int num\_faces) {

std::unordered\_set<Edge, Hash\_of\_Edge> edges;

int num\_edges = 0;

edges.reserve(faces.size() \* 3);

for (const auto& face : faces) {

addEdges(edges, face.vertices[0], face.vertices[1], num\_edges);

addEdges(edges, face.vertices[1], face.vertices[2], num\_edges);

addEdges(edges, face.vertices[2], face.vertices[0], num\_edges);

}

int euler\_characteristic = num\_vertices - num\_edges + num\_faces;

return 2 - euler\_characteristic;

//VertexToEdge vertexToEdgemap;

//vertexToEdgemap.reserve(edges.size() \* 2);

//for (const auto& edge : edges) {

// vertexToEdgemap.emplace(edge.i, edge);

// vertexToEdgemap.emplace(edge.j, edge);

//}

//auto i = edges.begin()->i;

//auto j = edges.begin()->j;

}

4. Write a function that returns all faces with minimum angle below a specified angle in degrees.

// 4. Write a function that returns all faces with minimum angle below a specified angle in degrees.

bool check\_below\_threshold\_angle(const Vertices& v, const double& cos2\_angle) const {

const Vertex& i = v[vertices[0]];

const Vertex& j = v[vertices[1]];

const Vertex& k = v[vertices[2]];

const double a = dist(i, j);

const double b = dist(j, k);

const double c = dist(k, i);

if (c + b > a) { double cos2\_a = (c + b - a) \* (c + b - a) / (4 \* c \* b); if (cos2\_angle > cos2\_a) return true; }

if (a + c > b) { double cos2\_b = (a + c - b) \* (a + c - b) / (4 \* a \* c); if (cos2\_angle > cos2\_b) return true; }

if (a + b > c) { double cos2\_c = (a + b - c) \* (a + b - c) / (4 \* a \* b); if (cos2\_angle > cos2\_c) return true; }

return false;

}

5. Write a function that collapses all edges with length below a specified threshold.

// 5. Write a function that collapses all edges with length below a specified threshold.

void Edge\_Colapse(Faces faces, Vertices v, int i,int j){

for (auto f : v[j].adjFaces)

for(int ii = 0; ii<3;ii++)

if (faces[f].vertices[ii] == j)

faces[f].vertices[ii] = i;

for (const auto& adjV : v[j].adjVertices)

for (auto& adjV2 : v[adjV].adjVertices)

if (adjV2 == j)

adjV2 = i;

// add adjFaces to i

}

6. (stretch) Write a function that performs a diagonal edge swap for triangles having an obtuse angle and an angle below a specified threshold in degrees.

// 6. (stretch) Write a function that performs a diagonal edge swap for triangles having an obtuse angle and an angle below a specified threshold in degrees.

void get\_opposite\_vertices(int i, int& j, int& k) {

if (i == 0) { j == 1; k == 2; return; }

if (i == 1) { j == 2; k == 0; return; }

if (i == 2) { j == 0; k == 1; return; }

}

void diagonal\_edge\_swap(Face& f1, Face& f2) {

bool found = false;

int j1, k1;

int j2, k2;

for (size\_t i1 = 0; i1 < 3; i1++)

{

get\_opposite\_vertices(i1, j1, k1);

for (size\_t i2 = 0; i2 < 3; i2++)

{

get\_opposite\_vertices(i2, j2, k2);

if (f1.vertices[j1] == f2.vertices[k2] && f1.vertices[k1] == f2.vertices[j2])

{

f1.vertices[k1] = f2.vertices[i2];

f2.vertices[k2] = f1.vertices[i1];

return;

}

}

}

}

Test files