CS133 Lab 8–DCEL (Part I)

Objective

- In this lab, you will implement the basic Doubly-Connected Edge List (DCEL) structure to use in future labs to implement Voronoi Diagram and/or Delaunay Triangulation algorithms.

Detailed Requirements

Create a DCEL data structure that contains a list of vertices, half edges, and surfaces as detailed below.

Structures

- Struct Vertex
  A Vertex object contains a single DCEL pointer, named “leaving”, to a HalfEdge object. This pointer points to a single HalfEdge that has this Vertex object as its origin. If multiple HalfEdges have this Vertex object as their origin, the leaving pointer can point to any one of them arbitrarily.
- Struct HalfEdge
  The HalfEdge object contains a pointer to a Vertex, named “origin”, a pointer to a Face named "face", and two pointers to HalfEdges, one named "twin" and one named “next”. The origin is the vertex from which the HalfEdge starts. The face is the face to the “left” side of the HalfEdge, while the twin pointer points to the HalfEdge on the “right” side of the HalfEdge that completes its edge. The “next” pointer points to the HalfEdge that starts from h->twin->origin and ends at the next vertex in h->face, traveling counterclockwise around the boundary. This pointer allows us to traverse a polygon, by following next pointers until we arrive back at the HalfEdge we began at.
- Struct Face
  A Face object contains a single DCEL pointer, named “edge”, to a HalfEdge object. This pointer points to a single HalfEdge that has this Face object as its face. This HalfEdge can be any one of the Face object's boundary HalfEdges.

Operations

Below, is a list of primitive operations in DCEL that you will implement in this lab. In these functions, PVertex, PEdge, and PFace, are pointers to a Vertex, a HalfEdge, or a Face, respectively.

1. `PEdge Vertex::nextLeaving(PEdge)`
   Given an edge leaving a vertex v, this function returns the next edge leaving the vertex v. Note that in the vertex, we only store one leaving edge. This function should be used to iterate over *all* leaving edges of a vertex.

2. `PEdge::destination() : PVertex`
   Returns the destination vertex of a HalfEdge. Note that we only store the source vertex of a HalfEdge. This function is used to return the destination vertex as well.
3. `PVertex DCEL:createVertex(double x, double y)`
   Creates and returns a new vertex at the given point location. Initially, this vertex is not connected to any half edges.
4. `std::vector<PFace> DCEL::findFaces(PVertex)`
   Returns all faces that are adjacent to a vertex v.
5. `PFace DCEL::findCommonFace(PVertex, PVertex)`
   Finds a common face between two vertices. A common face has the two given vertices on its boundary. If the two vertices have more than one common face, any bounded face is returned. That is, if they have two common faces and one of them is the unbounded face, the other one is returned. If more than one bounded face exists, any of them is returned. If no common faces exist, a NULL pointer is returned.
6. `bool DCEL::isConnected(PVertex, PVertex)`
   Returns true if and only if the two given vertices have a common edge between them.
7. `PEdge DCEL::findIncidentEdge(PVertex v, PFace f)`
   If exists, returns the HalfEdge that has v as its source vertex and f as its face. If such an edge does not exist, a NULL pointer is returned.
8. `PFace DCEL::getUnboundedFace()`
   Returns the single unbounded face in this DCEL. Keep in mind that the unbounded face is a special face that never gets deleted.
9. `PEdge DCEL::createEdge(PFace f, PVertex v1, PVertex v2)`
   Connects the two vertices v1 and v2 that belong to the same face (f) without splitting that face. Returns any of the newly created half edges.
10. `PEdge DCEL::splitFace(PFace f, PVertex v1, PVertex v2)`
    Creates a new half edge between the vertices v1 and v2 which splits the face f into two new faces. Returns any of the newly created half edges.
11. `DCEL mergeAndDestroy(DCEL& d1, DCEL& d2)`
    Given two DCEL structures, this function combines them together into a new DCEL. The returned DCEL combines all the vertices, edges, and faces in both DCELs. Except for the unbounded face, it is assumed that all vertices, edges, and faces in both DCELs are different. For efficiency purposes, the two input DCELs are destroyed as a result of calling this function.

Feel free to add more supporting functions as you need.

Example

Below is a simple example that you can use to test your implementation. It starts with a simple empty DCEL and manipulates it using the functions given above. You should add more sophisticated test cases to ensure the correctness of your implementation.

```
DCEL dcel;
PVertex v1 = dcel.createVertex(1,0);
```

```
PVertex v2 = dcel.createVertex(2,1);
PVertex v3 = dcel.createVertex(0,1);
PVertex v4 = dcel.createVertex(0,0);
// v1 has one face (the unbounded face)
assert(dcel.findFaces(v1).size() == 1);

PEdge e1 = dcel.createEdge(dcel.getUnboundedFace(), v1, v2);
// v1 still has one face which is not NULL
assert(dcel.findFaces(v1).size() == 1);
assert(dcel.findFaces(v1).front() != NULL);
// v2 also has one unbounded face
assert(dcel.findFaces(v2).size() == 1);
// All vertices have one common face which is the unbounded face
assert(dcel.findCommonFace(v1, v2) != NULL);
assert(dcel.findCommonFace(v1, v3) != NULL);
// Find the newly created edge using the findIncidentEdge function
assert(dcel.findIncidentEdge(e1->origin, e1->face) == e1);
dcel.createEdge(dcel.getUnboundedFace(), v2, v3);
// v1 and v2 are still connected
assert(dcel.isConnected(v1, v2));
// v1 and v3 are not connected (i.e., not adjacent)
assert(!dcel.isConnected(v1, v3));
// Create two new edges to create the first face
dcel.createEdge(dcel.getUnboundedFace(), v3, v4);
dcel.splitFace(dcel.getUnboundedFace(), v4, v1);
// Now there are two faces, the newly created face and the unbounded face
assert(dcel.getFaceCount() == 2);
// All the four vertices are adjacent to the two faces
assert(dcel.findFaces(v1).size() == 2);
assert(dcel.findFaces(v2).size() == 2);
assert(dcel.findFaces(v3).size() == 2);
assert(dcel.findFaces(v4).size() == 2);
// v1 and v2 have two common faces, but the bounded face should be returned
assert(dcel.findCommonFace(v1, v2) != dcel.getUnboundedFace());
assert(dcel.findCommonFace(v1, v2)->getBoundary().size() == 4);
// Create a new edge that will result in a new face
dcel.splitFace(dcel.findCommonFace(v1, v2), v4, v2);
assert(dcel.getFaceCount() == 3);
assert(dcel.findFaces(v4).size() == 3);
```