CMSC 754: Lecture 10
The Doubly-Connected Edge List

Reading: Chapter 2 in the 4M's.

Doubly-connected Edge List: In our next lecture, we will discuss two important planar subdivisions, Voronoi diagrams and Delaunay triangulations. An important question is how these objects can be represented. The mathematical structures that constitute planar subdivisions go by various names, including *planar straight-line graph* (or PSLG) and *cell complex* (see Fig. 1). Such a structure represents a decomposition of the plane into vertices (0-dimensional cells), edges (1-dimensional cells), and faces (2-dimensional cells).

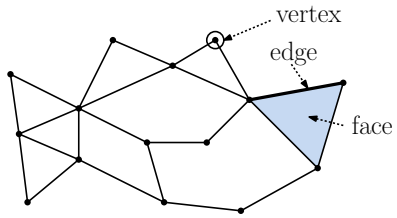

Fig. 1: Cell complex (planar straight-line graph).

In this lecture we consider the question of how to represent planar cell complexes, called a *doubly-connected edge list* (or DCEL). The DCEL is a common *edge-based representation*. Vertex and face information is also included for whatever geometric application is using the data structure. There are three sets of records one for each element in the cell complex: *vertex records*, a *edge records*, and *face records*. For the purposes of unambiguously defining left and right, each undirected edge is represented by two directed *half-edges*.

We will assume that the faces of complex do not have holes inside of them. (More formally, we say that the boundary of each face is *simply connected*.) This assumption can be always be satisfied by introducing some number of *dummy edges* joining each hole either to the outer boundary of the face, or to some other hole that has been connected to the outer boundary in this way. With this assumption, we may assume that the edges bounding each face form a single cyclic list.

Here are the basic elements of the DCEL:

Vertex: Each vertex stores information pertinent to the vertex, such as its coordinates and identity. Along with this, it stores a pointer to any incident directed edge that has this vertex as its origin, `v.inc_edge`.

Edge: Each undirected edge is represented as two oppositely-directed half-edges. Each edge has a pointer to the oppositely directed edge, called its *twin*. It also has an *origin* and *destination* vertex. Each directed edge is associate with two faces, one to its left and one to its right (with respect to an observer facing the edge's direction).

We store a pointer to the origin vertex `e.org`. (We do not need to define the destination, `e.dest`, since it may be defined to be `e.twin.org`.)

We store a pointer to the face to the left of the edge `e.left` (we can access the face to the right from the twin edge). This is called the *incident face*. We also store the next and previous directed edges in counterclockwise order about the incident face, `e.next` and `e.prev`, respectively.

Face: Each face f stores a pointer to a single edge for which this face is the incident face, `f.inc_edge`. (See the text for the more general case of dealing with holes.)

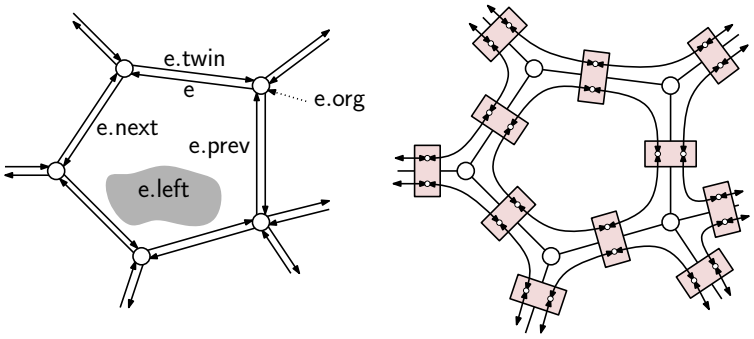

Fig. 2: Doubly-connected edge list.

The figure shows two ways of visualizing the DCEL. One is in terms of a collection of doubled-up directed edges. An alternative way of viewing the data structure that gives a better sense of the connectivity structure is based on covering each edge with a two element block, one for e and the other for its twin. The next and prev pointers provide links around each face of the polygon. The next pointers are directed counterclockwise around each face and the prev pointers are directed clockwise.

Of course, in addition the data structure may be enhanced with whatever application data is relevant. In some applications, it is not necessary to know either the face or vertex information (or both) at all, and if so these records may be deleted. See the book for a complete example.

For example, suppose that we wanted to enumerate the vertices that lie on some face f. Here is the code:

Vertex enumeration using DCEL

```
enumerate_vertices(Face f) {
    Edge start = f.inc_edge;     // some edge oriented CCW with respect to this face
    Edge e = start;
    do {
        output e.org;           // output the origin vertex of this edge
        e = e.next;             // advance to the next edge in CCW about the face
    } while (e != start);       // ... until we return to the start edge
}
```

Merging subdivisions: To illustrate the use of the DCEL data structure, consider the following application. We are given two planar subdivisions, A and B, each represented as a DCEL, and we want to compute their overlay. We will make the general-position assumption that no two vertices share the same location, and no two edges are collinear. Thus, the only

interactions between the two subdivisions occur when a pair of edges cross over one another. In particular, whenever two edges of these subdivision cross, we want to create a new vertex at the intersection point, split the two edges in two fragment, and connect these fragments together about this vertex (see Fig. 3).

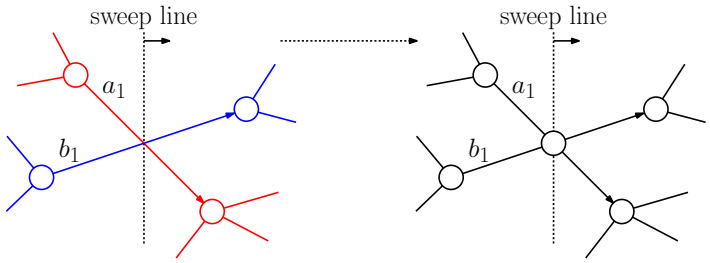

Fig. 3: Merging subdivisions by creating a vertex at an intersection point.

Our approach will be to modify the plane-sweep algorithm to generate the DCEL of the overlaid subdivision. The algorithm will destroy the original subdivisions, so it may be desirable to copy them before beginning this process. The first part of the process is straightforward, but perhaps a little tedious. This part consists of building the edge and vertex records for the new subdivision. The second part involves building the face records. It is more complicated because it is generally not possible to know the face structure at the moment that the sweep is advancing, without looking "into the future" of the sweep to see whether regions will merge. (You might try to convince yourself of this.) Our textbook explains how to update the face information. We will focus on updating just the edge information.

The critical step of the overlaying process occurs with we sweep an intersection event between two edges, one from each of the subdivisions. Let us denote these edges as $a_1 \in A$ and $b_1 \in B$. Recall that each edge of the subdivision is represented by two half edges. We will assume that a_1 and b_1 are selected so that they are directed from left to right across the sweep-line (see Fig. 3). The process will make use of two auxiliary procedures:

- `split(a1, a2)` splits an edge a_1 into two consecutive edges a_1 followed by a_2, and links a_2 into the structure (see Fig. 4(a)).
- `splice(a1, a2, b1, b2)` takes two such split edges, which are assumed to meet cyclically in counterclockwise order about a common intersection point in the order $\langle a_1, b_1, a_2, b_2 \rangle$, and links them all together about a common vertex (see Fig. 4(b)).

The splitting procedure creates the new edge and links it into place (see the code block below). The edge constructor is given the origin and destination of the new edge and creates a new edge and its twin. The procedure below initializes all the other fields. Also note that the destination of a_1, that is the origin of a_1's twin must be updated, which we have omitted.

The splice procedure interlinks four edges around a common vertex in the counterclockwise order a_1 (entering), b_1 (entering), a_2 (leaving), b_2 (leaving). (See the code block below.)

Given these two utilities, the function `merge(a1, b1)` given in the following code block splits the edges and links them to a common vertex.

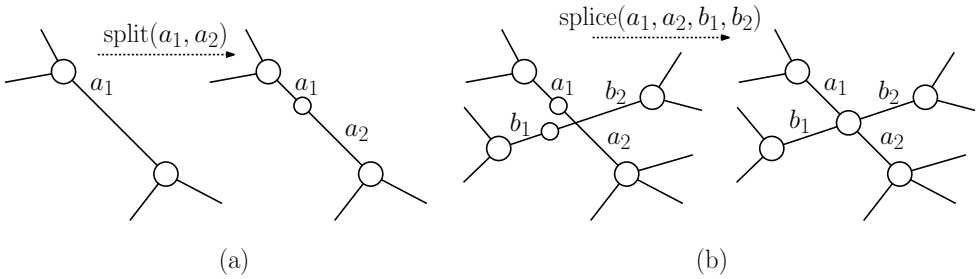

Fig. 4: The `split` and `splice` operations.

Split an edge into two edges

```
split(edge a1, edge a2) {                // a2 is returned
    a2 = new edge(v, a1.dest());         // create edge (v,a1.dest)
    a2.next  = a1.next;   a1.next.prev  = a2;
    a1.next  = a2;        a2.prev  = a1;
    a1t = a1.twin;        a2t = a2.twin; // the twins
    a2t.prev = a1t.prev;  a1t.prev.next = a2t;
    a1t.prev = a2t;       a2t.next  = a1t;
}
```

Splice four edges together

```
splice(edge a1, edge a2, edge b1, edge b2) {
    a1t = a1.twin;   a2t = a2.twin;      // get the twins
    b1t = b1.twin;   b2t = b2.twin;
    a1.next  = b2;   b2.prev  = a1;      // link the edges together
    b2t.next = a2;   a2.prev  = b2t;
    a2t.next = b1t;  b1t.prev = a2t;
    b1.next  = a1t;  a1t.prev = b1;
}
```

Splice four edges together

```
merge(edge a1, edge b1) {
    Create a new vertex v where a1 and b1 intersect
    a2 = split(a1); b2 = split(b1);      // split the two edges
    splice(a1, a2, b1, b2);              // splice them together about the vertex v
}
```