Meshes and Some Terminology: In the previous lecture, we discussed navigation meshes. This structure provided a way to represent the “walkable surface” of a domain as a triangulated mesh. In order to walk from one mesh element to the next, we need a representation that allows us to easily identify neighboring mesh elements. In this lecture, we will describe such a data structure, which is called the doubly-connected edge list, or DCEL for short.

When representing meshes, it is common to distinguish between two facets of the representations, geometry and topology. The geometric information involves the locations of objects, such as the coordinates of vertices or the equations of its faces. The topological information involves how the elements of the mesh are connected together. This also involves issues such as whether there are holes or cavities within the model.

In the field of topology, a surface patch is called 2-manifold (see Fig. 1(a)). A defining property of 2-manifolds is that in any sufficiently small local neighborhood surrounding any interior point of the surface looks (up to stretching) like a small circular disk. (See Fig. 1(b) for examples of violations.) Our manifolds to have boundaries, and they may generally contain holes. Intuitively, you can think of a 2-manifold (with boundary) to be a very thin rubber sheet, to which someone may have cut out holes.

In order to represent surfaces, it is common to break them up into small polygonal elements, which are typically triangles. (Triangles are nice, because are always convex and always planar. In general, a polygonal in 3-dimensional space that is built using four or more vertices might fail either of these properties.) When two triangles of the mesh are joined together, they are joined edge-to-edge (see Fig. 1(c)). This implies, in particular, that a vertex of one triangle will not appear in the interior of an edge or a face of another triangle (as in Fig. 1(d)). Such a decomposition is called a cell complex or (when triangles only are involved) a simplicial complex. The DCEL data structure is used for representing cell complexes on 2-manifold surfaces.

![Fig. 1: 2-Manifolds and cell complexes.](image-url)
The DCEL: A cell complex subdivides a mesh into three types of elements, vertices (0-dimensional), edges (1-dimensional), and faces (2-dimensional). Thus, we can encode the topological information of a mesh as an undirected graph. For the purposes of unambiguously distinguishing left from right, it will be convenient to represent each undirected edge by two oppositely directed edges, called half-edges. An edge directed from \( u \) to \( v \) is said to have \( u \) as its origin and \( v \) as its destination.

For now, let us make the simplifying assumption that the faces of the mesh not have holes inside of them. (This assumption can always be satisfied by introducing some number of bridging edges that join the outer boundary of the face to each of the holes. With this assumption, it may be assumed that the edges edges each face form a single cyclic list.

The DCEL consists of three principal elements, vertices, edges, and faces. For each, we store the following information:

**Vertex:** Each vertex stores its spatial coordinates, along with a reference to any incident half-edge that has this vertex as its origin, \( v.\text{incident} \).

**Face:** Each face \( f \) stores a reference to a single half-edge for which this face is the incident face, \( f.\text{incident} \). (Such a half-edge will be directed counterclockwise about the face.)

**Edge:** Each half-edge \((u, v)\) is naturally associated with two vertices, its origin \( u \) and its destination \( v \), its twin half-edge \((v, u)\), and its two incident faces, one to the left and one to the right. To distinguish left from right, let us assume that our mesh has an outward facing side and an inward side. (This works fine for most meshes that arise in solid modeling, since they enclose solid bodies. There are exceptions, however, such as a Mobius strip.) Consider the half-edge \((u, v)\) and imagine that you are standing in the middle of the edge on the outer side of the mesh with \( u \) behind you and \( v \) in front. The face to your left is the half-edge’s left face, and the other is the right face.

Each half-edge \( e \) the DCEL stores the following references (see Fig. 2):

- \( e.\text{org} \): \( e \)'s origin
- \( e.\text{twin} \): \( e \)'s oppositely directed twin half-edge
- \( e.\text{left} \): the face on \( e \)'s left side
- \( e.\text{next} \): the next half-edge after \( e \) in counterclockwise order about \( e \)'s left face
- \( e.\text{prev} \): the previous half-edge to \( e \) in counterclockwise order about \( e \)'s left face (that is, the next edge in clockwise order).

You might observe that there are a number of potentially useful references that we did not store. This is done to save space, because they can all be computed easily from the above:

- \( e.\text{dest} \): \( e \)'s destination vertex \((e.\text{dest} ← e.\text{twin.org})\)
- \( e.\text{right} \): the face on \( e \)'s right side \((e.\text{right} ← e.\text{twin.left})\)
- \( e.\text{onext} \): the next half-edge that shares \( e \)'s origin that comes after \( e \) in counterclockwise order \((e.\text{onext} ← e.\text{prev.twin})\)
- \( e.\text{oprev} \): the previous half-edge that shares \( e \)'s origin that comes before \( e \) in counterclockwise order \((e.\text{oprev} ← e.\text{twin.next})\)
Fig. 2: Doubly-connected edge list.

Fig. 2 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 \texttt{next} and \texttt{prev} references define a doubly-linked list around each of the faces of the mesh. The \texttt{next} references are directed counterclockwise around each face and the \texttt{prev} references are directed clockwise.

**Mesh traversal with DCELs:** Suppose that we have a mesh stored and a DCEL, and we want to enumerate the vertices that lie on some face \( f \) in counterclockwise order about the face. We could start at any incident edge \( e \), output its origin \( e.\text{org} \), and then go to the next edge in counterclockwise order about the face \( e.\text{next} \). This is presented in the procedure \texttt{faceVertices} below.

```plaintext
faceVerticesCCW(Face f) {
    Edge start = f.incident;
    Edge e = start;
    do {
        output e.org;
        e = e.next;
    } while (e != start);
}
```

As another example, suppose that we want to enumerate all the vertices that are neighbors of a given vertex \( v \) in clockwise order about this vertex. We could start at any incident edge \( e \) (which by definition has \( e \) as its origin), output its destination vertex, and then visit the next vertex about the origin in clockwise order.
vertexNeighborsCW(Vertex v) {
    Edge start = v.incident;
    Edge e = start;
    do {
        output e.dest; // formally: output e.twin.org
        e = e.oprev; // formally: e = e.twin.next
    } while (e != start);
}