CMSC 425: Lecture 14
Solid Modeling

Reading: The material on CSG comes from standard texts on solid modeling. The material on DCELs comes from Chapter 2 in *Computational Geometry: Algorithms and Applications* by de Berg, Cheong, van Kreveld, and Overmars. It is also covered in Chapter 2 of *Foundations of Multidimensional Data Structures* by Samet.

Solid Object Representations: A major issue in geometric modeling for game assets is how to represent solid geometric objects. There are two common ways in which to represent solids. The first is based on representing an object's boundary surface, called a *boundary representation* or *B-rep* for short. The second is based on representing the object as boolean operations applied to solid volumes, called *CSG* for *constructive solid geometry*. Both have their advantages and disadvantages.

Volume Based Representations: One of the most popular volume-based representations is *constructive solid geometry*, or *CSG* for short. It is widely used in manufacturing applications. One of the most intuitive ways to describe complex objects, especially those arising in manufacturing applications, is as set of *boolean operations* (that is, set union, intersection, difference) applied to a basic set of primitive objects. Manufacturing is an important application of computer graphics, and manufactured parts made by various milling and drilling operations can be described most naturally in this way. For example, consider the object shown in Fig. 1. It can be described as a rectangular block, minus the central rectangular notch, plus (union) two cylindrical holes.

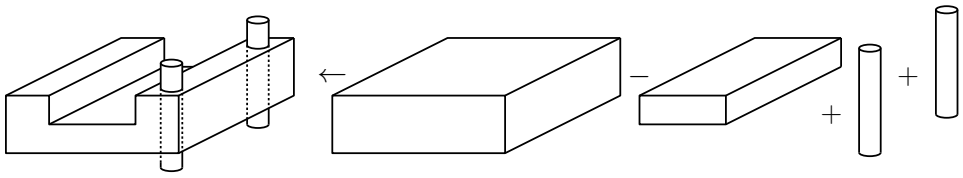

Fig. 1: Constructive Solid Geometry.

This idea naturally leads to a tree representation of the object, where the leaves of the tree are certain *primitive object types* (rectangular blocks, cylinders, cones, spheres, etc.) and the internal nodes of the tree are *boolean operations*, union ($X \cup Y$), intersection ($X \cap Y$), difference ($X - Y$), etc. For example, the object above might be described with a tree of the following sort. (In Fig. 1 we have used "+" for union.)

The primitive objects stored in the leaf nodes are represented in terms of a primitive *object type* (block, cylinder, sphere, etc.) and a set of defining *parameters* (location, orientation, lengths, radii, etc.) to define the location and shape of the primitive. The nodes of the tree are also labeled by transformation matrices, indicating the transformation to be applied to the object prior to applying the operation. By storing both the transformation and its inverse, as we traverse the tree we can convert coordinates from the world coordinates (at the root of the tree) to the appropriate local coordinate systems in each of the subtrees.

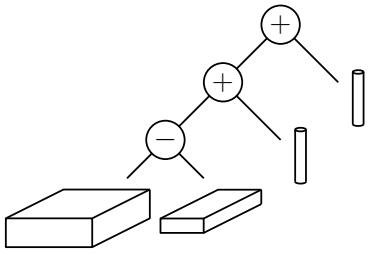

Fig. 2: CSG Tree.

This method is called constructive solid geometry (CSG) and the tree representation is called a CSG tree. One nice aspect to CSG and this hierarchical representation is that once a complex part has been designed it can be reused by replicating the tree representing that object. (Or if we share subtrees we get a representation as a directed acyclic graph or DAG.)

Point membership: CSG trees are examples of *unevaluated models*. For example, unlike a B-rep representation in which each individual element of the representation describes a feature that we know is a part of the object, it is not generally possible to infer the existence of any feature locally (without looking at the entire tree).

Consider the simple membership question: Given a point P, does P lie inside or outside an object described by a CSG tree? (For now, let us ignore the degenerate case where it lies on the boundary.) How would you write an algorithm to solve this problem? The idea is to work recursively, solving the problem on the subtrees first, and then combining results from the subtrees to determine the result at the parent. We will write a procedure isMember(Point p, CSGnode u) where p is the point, and u is pointer to a node in the CSG tree. This procedure returns True if the object defined by the subtree rooted at u contains p and False otherwise. If u is an internal node, let u.left and u.right denote the children of u. The algorithm breaks down into the following cases.

Membership Test for CSG Tree

```
bool isMember(Point p, CSGnode u) {
    if (u.isLeaf)
        return u.primitiveMemberTest(p);
    else if (u.isUnion)
        return isMember(p, u.left) ||  isMember(p, u.right);
    else if (u.isIntersect)
        return isMember(p, u.left) &&  isMember(p, u.right);
    else if (u.isDifference)
        return isMember(p, u.left) && !isMember(p, u.right);
}
```

Note that the semantics of operations "||" and "&&" avoid making recursive calls when they are not needed. For example, in the case of union, if p lies in the right subtree, then the left subtree need not be searched.

Regularized boolean operations (Optional): There is a tricky issue in dealing with boolean operations. This goes back to a the same tricky issue that arose in polygon filling, what to

do about object boundaries. Consider two bodies A and B in Fig. 3(a), where B just touches one of the vertical walls of A. The intersection $A \cap B$ shown in Fig. 3(b) contains a "dangling piece" that has no width. That is, it is locally two-dimensional.

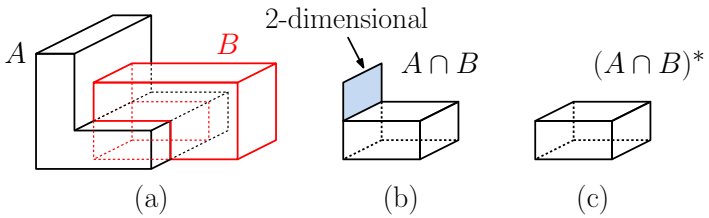

Fig. 3: (a) A and B, (b) $A \cap B$, (c) $A \cap^* B$.

These lower-dimensional parts can result from boolean operations, and are usually unwanted. For this reason, it is common to modify the notion of a boolean operation to perform a *regularization* step. Given a 3-dimensional set A, the regularization of A, denoted A^*, is the set with all components of dimension less than 3 removed.

In order to define this formally, we must introduce some terms from topology. We can think of every (reasonable) shape as consisting of three disjoint parts, its *interior* of a shape A, denoted $\text{int}(A)$, its *exterior*, denoted $\text{ext}(A)$, and its *boundary*, denoted $\text{bnd}(A)$ (or often ∂A). Define the *closure* of any set to be the union of itself and its boundary, that is, $\text{closure}(A) = A \cup \text{bnd}(A)$.

Topologically, the regularization A^* is defined to be the closure of the interior of A

$$A^* = \text{closure}(\text{int}(A)).$$

Note that $\text{int}(A)$ does not contain the dangling element, and then its closure adds back the boundary.

When performing operations in CSG trees, we assume that the operations are all *regularized*, meaning that the resulting objects are regularized after the operation is performed.

$$A \text{ op}^* B = \text{closure}(\text{int}(A \text{ op } B)).$$

where "op" is either $\cap$, $\cup$, or $-$. Eliminating these dangling elements tends to complicate CSG algorithms, because it requires a bit more care in how geometric intersections are represented.

Meshes and Some Terminology: As mentioned above, the other common method used for representing solid objects is based on boundary representations. This is the more common method, since it fits well with modern graphics systems, which are highly optimized for rendering mesh-based models. An important question is how to efficiently store such models. We will describe such a data structure called the *doubly-connected edge list*, or *DCEL* (often pronouced "dee cell") 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. 4(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. 4(b) for examples of violations.) We 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.

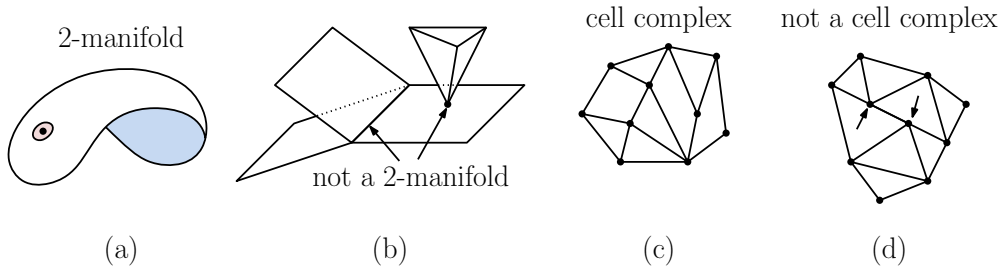

Fig. 4: 2-Manifolds and cell complexes.

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. 4(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. 4(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.

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 do 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, the edges of 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 single incident half-edge that has this vertex as its origin, $v.\mathsf{incident}$.

Face: Each face f stores a reference to a single half-edge for which this face is the incident face, f.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. 5):

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

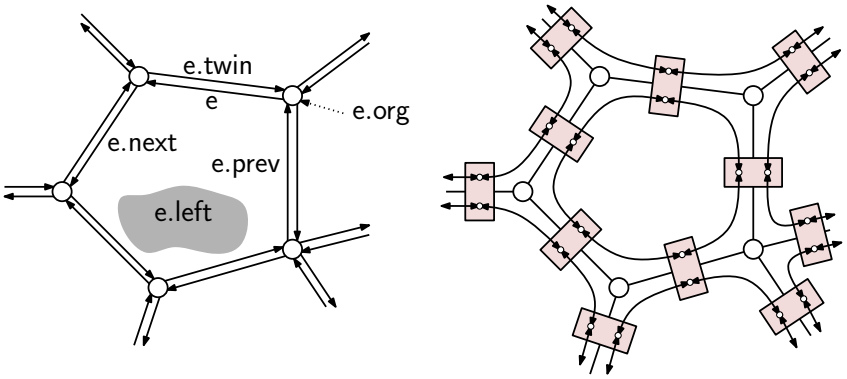

Fig. 5: Doubly-connected edge list.

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.dest: e's destination vertex (e.dest $\leftarrow$ e.twin.org)
- e.right: the face on e's right side (e.right $\leftarrow$ e.twin.left)
- e.onext: the next half-edge that shares e's origin that comes after e in counterclockwise order (e.onext $\leftarrow$ e.prev.twin)
- e.oprev: the previous half-edge that shares e's origin that comes before e in counterclockwise order (e.oprev $\leftarrow$ e.twin.next)

Fig. 5 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 references define a doubly-linked list around each of the faces of the mesh. The next references are directed counterclockwise around each face and the 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.org, and then go to the next edge in counterclockwise order about the face e.next. This is presented in the procedure faceVertices below.

Enumerate the vertices about a face

```
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.

Enumerate the neighbors of a vertex

```
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);
}
```

There are a number of data structures for storing cell complexes that are functionally equivalent to the DCEL (for example, the *winged-edge data structure*, the *half-edge data structure*, and the *quad-edge data structure*.) These data structures all share the useful property that it is possible to traverse the edges (and hence indirectly the vertices and faces) that are incident to any vertex or any face in time proportional to the number of such edges. Furthermore, this traversal may be done in either clockwise or counterclockwise order. Furthermore, it is possible to efficiently make changes to the structure (e.g., splitting a face by adding an edge or collapsing an edge into a vertex). These operations are often useful when dealing with dynamic meshes. Intuitively, these data structures provide a natural way to generalize the concept of a *doubly linked list* to 2-dimensional cell complexes.