

## CMSC 425: Lecture 8

### Geometric Data Structures for Games: Meshes and Manifolds

Tuesday, Feb 19, 2013

**Reading:** This material is based on Chapt 10 of *Game Engine Architecture* by J. Gregory and Chapt 12 of *Fundamentals of Computer Graphics* (3rd edition) by P. Shirley and S. Marschner. The material on DCELs is covered in *Computational Geometry: Algorithms and Applications* (3rd Edition) by M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars.

**Geometric Data Structures:** In modern games, it is often desirable to give the player the feeling of being emmersed in an interesting environment. Managing large geometric models can be computationally very challenging. The objects being modeled can be of various forms ranging from rigid structures like buildings, articulated entities like people and animals, and amorphous objects like water and smoke. There are many ways that a game may need to interact with such models:

**Rendering:** The objects visible to the camera need to be rendered. This involves lighting and texturing, hidden-surface removal, efficiently computing which objects are at least partially visible, computation of shadows, determining the effect of atmospheric effects such as smoke.

**Navigation and motion:** Non-player entities often need to plan an obstacle-free path through a complex and dynamically changing environment. Often groups of objects need to plan coordinated motion, like a group of soldiers, pedestrians walking on the street, or a moving herd of animals. Auxiliary data structures may need to be maintained to facilitate computing these paths.

**Collision detection:** It is necessary to determining collisions to forbid the player (or other moving entities) from violating basic physical laws. Some collisions, such as weapons hits, are significant to the internal state of the game. Some games involve interactions of movable physical structures (such sling-shooting a bird into a towers of blocks), it may be necessary to simulate how they topple and collapse.

In this lecture, we will discuss fundamental geometric data structures from a fairly general perspective. In future lectures, we will see how these basic data structure can be adapted and applied to tasks like those mentioned above. Examples include:

**Triangle Meshes:** Are used for the storage and manipulation of geometric models. Such meshes can also be structured hierarchically in order to provide level-of-detail approximations.

**Geometric graphs and subdivisions:** Geometric graphs provide an invisible structure which can guide the navigation of non-player characters (NPCs). Geometric subdivisions are often used for interpolation and providing a structure for computing the motion of fluids and gasses.

**Scene graphs:** These are used for representing hierarchical assemblies of objects and their relationships.

**Grids:** These are among the simplest structures for storing geometric objects and are often used in collision detection and fluid dynamics.

**Hierarchical Spatial Subdivisions:** Data structures like quadtrees, kd-trees, and BSP-trees are also used in collision detection, but they are much more flexible and adaptable than grids. They are also used in visibility culling, and ray-shooting.

**Triangle Meshes:** While there are a number of different methods for representing 3-dimensional solid objects, the most common method used in computer games is based on representing the surface of the object as a mesh of triangles. These are also known as *triangular meshes*, *triangulated meshes*, and *triangular irregular networks*<sup>1</sup> (TINs). (An example of a triangle mesh is shown in Fig. 1(a), and an example of a quadrilateral mesh is shown in Fig. 1(b).)

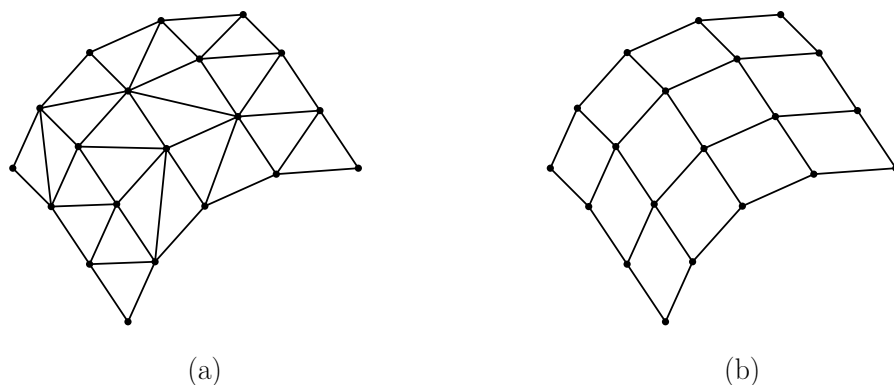


Fig. 1: A triangle mesh (a) and a quadrilateral mesh (b).

Why use triangles? They have a number of nice properties. First, they are the simplest polygon, unlike  $k$ -sided polygons for  $k \geq 4$ , triangles are always convex and (in 3-dimensional space) they are always planar. Finally, because they are so ubiquitous, graphics hardware has been optimized to efficiently render triangle meshes. The most common alternative to the triangle is the quadrilateral. One reason that quadrilaterals are popular is that if they are sufficiently regular, they can be stored implicitly in a 2-dimensional array of vertices.

Why connect the triangles into a mesh, as opposed to just considering a set of triangles? By putting triangles in a mesh, it is possible to perform global optimizations that would not be easy given an arbitrary set of triangles. For example, to compute the shadow cast by a mesh, it suffices to visit just the border edges of the mesh, which is typically many fewer than the total number. Also, because many triangles may share the same vertex, we can perform optimizations such as computing the illumination for each vertex once, and then reuse this information for each triangle that shares this vertex.

**Graphics Representation:** What is the best way to represent a 3-dimensional triangle mesh for the sake of rendering? Perhaps the simplest method would be to create a class called *Triangle*, which would consist of an array of three vertices, where each vertex would be represented by a vector of three floats or doubles. This means that every triangle would involve nine floating point numbers. (By the way, this is just what is needed to represent the geometry. We should also store information for lighting such as surface normal vectors at each of the vertices and coordinates for texture mapping.) Clearly, this naive solution is not very space efficient.

A better approach would be to generate two arrays. The first array holds the *vertices*. Each entry

<sup>1</sup>You might wonder, what is “irregular” about triangular meshes. In the early days of meshing, most meshes were based on a regular 2-dimensional array of vertices, which were linked together in a grid of quadrilaterals, like a large fish net. Except along the boundary, each vertex in such a grid has exactly four neighbors, and so the grid is *regular*. Triangle meshes are not so constrained, and hence are *irregular*.

consists of three vertex objects, where a vertex object is a vector, consisting say of three floating point coordinates (see Fig. 2(b)). An alternative approach (which is used by OpenGL) is to store all the coordinates in a single 1-dimensional array, where each three consecutive entries ( $[0, 1, 2]$ ,  $[3, 4, 5]$ , and so on) are the coordinates of a single vertex.

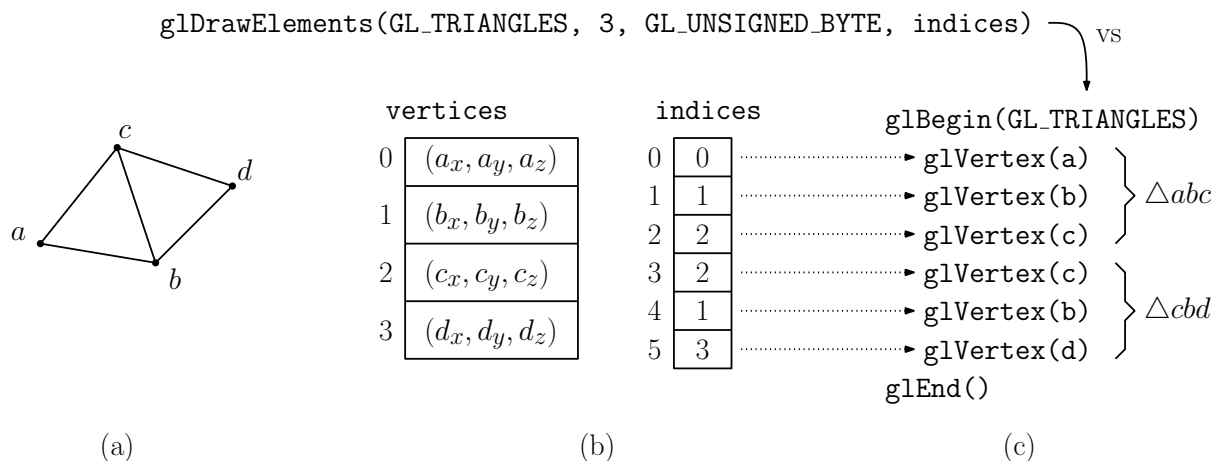


Fig. 2: Drawing a triangle mesh using an OpenGL index array.

In order to represent the triangles, we generate an array of *indices*, where the first three entries specify the vertices of the first triangle, the next three entries specify the vertices of the second triangle, and so on (see Fig. 2(b) for the mesh given in Fig. 2(a)). Note that each vertex is stored only *once*, no matter how many triangles contain it. Once the vertex array has been set up, we only need one index to reference an individual vertex. The straightforward method of doing this would involve transmitting 18 floating point values and making 8 OpenGL calls (see Fig. 2(c)).

We have seen how to draw a sequence of triangles in OpenGL using `glBegin(GL_TRIANGLES)`. This involves transmitting nine coordinates for each triangle that is drawn. OpenGL supports a more efficient mechanism, which involves setting an array of vertices and array of indices. Through the use of the OpenGL function `glDrawElements` it is possible to render the entire mesh by passing in the arrays vertices and the indices (see the code block below). (I will not explain the various arguments, but instead refer you to the OpenGL documentation. There are, by the way, more sophisticated ways of maintaining multiple buffers of vertices and other information, such as colors.)

**Mesh Toplogy:** A mesh is characterized by two important features, (1) where in space are the vertices that make up its triangles and (2) how are these triangles connected together? The answer to question (1) defines the *geometry* of the mesh. The answer to (2) defines the *topology* of the mesh.

When defining how triangles can be joined to make a mesh, there are usually certain requirements that are laid down. The first requirement is that the mesh be a *cell complex*. Saying that a mesh is a cell complex means that the triangular elements of the mesh are “properly joined” to each other. What does this mean? For example, when two triangles intersect, they either share an entire edge in common or they share just a vertex in common. There are many illegal ways that triangles might intersect (see Fig. 3), but these cannot occur in a cell complex.

Cell complexes (that represent surfaces) are composed of three types of elements: 0-dimensional

---

 Rendering a Triangle Mesh using an Index Array

```

GLfloat vertices[] = { ax, ay, az,          // define vertices
                      bx, by, bz,
                      cx, cy, cz,
                      dx, dy, dz };
GLubyte indices[] = { 0, 1, 2, 2, 1, 3 };    // define indices

glEnableClientState(GL_VERTEX_ARRAY);        // enable vertex array
glEnableClientState(GL_INDEX_ARRAY);         // enable index array

glIndexPointer(GL_UNSIGNED_BYTE, 0, indices); // location of indices
glVertexPointer(3, GL_FLOAT, 0, vertices);    // location of vertices
                                              // draw the mesh
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, indices);
  
```

---

elements called *vertices*, 1-dimensional elements called *edges*, and 2-dimensional elements called *faces*.

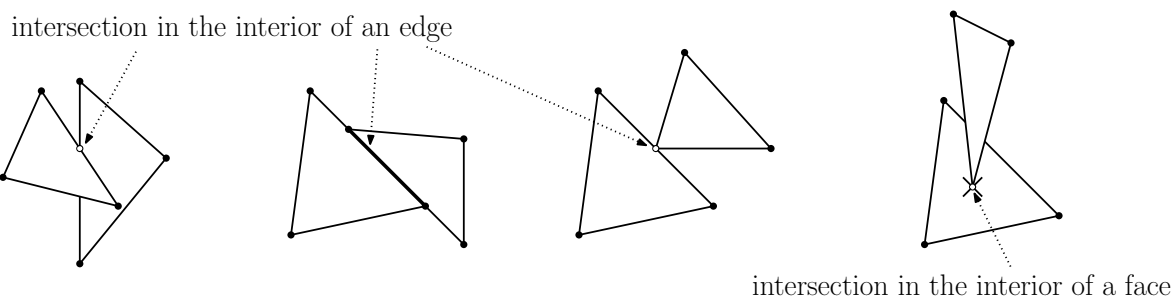


Fig. 3: Examples of triangle intersections that cannot occur in a cell complex.

Let us assume from here on that our meshes are cell complexes. The second condition that one would like to have satisfied by a mesh is that it defines 2-dimensional surface. In topology terms, this is called a *2-manifold*. Formally, this means that, if you consider a small neighborhood around any point (which might be a vertex, in the interior of an edge, or in the interior of a triangle face) the region around this point forms a 2-dimensional topological disk. Why might this fail to happen? Consider the neighborhoods shown in Fig. 4(a), (b), and (c). In all three cases, the neighborhood of the point is topologically equivalent to a 2-dimensional disk. However, in Fig. 4(d) and (e), the neighborhood of the point is definitely not a disk.

An equivalent characterization of a 2-manifold for cell complexes is that each edge should be incident to exactly two triangles and each vertex should have a single loop of triangles about it.

Unfortunately, pure 2-manifolds do not allow for models that have a boundary, since the neighborhood surrounding a boundary point is essentially a topological half-disk. We say that a surface is a *2-manifold with boundary* if every interior point of the mesh satisfies the above definition for 2-manifolds, and each boundary point has a single semi-disk as its neighborhood.

Although there are a few applications of non-manifold surfaces, it is common to assume that all the triangular meshes that we will deal with are 2-manifold cell complexes (possibly with a boundary).

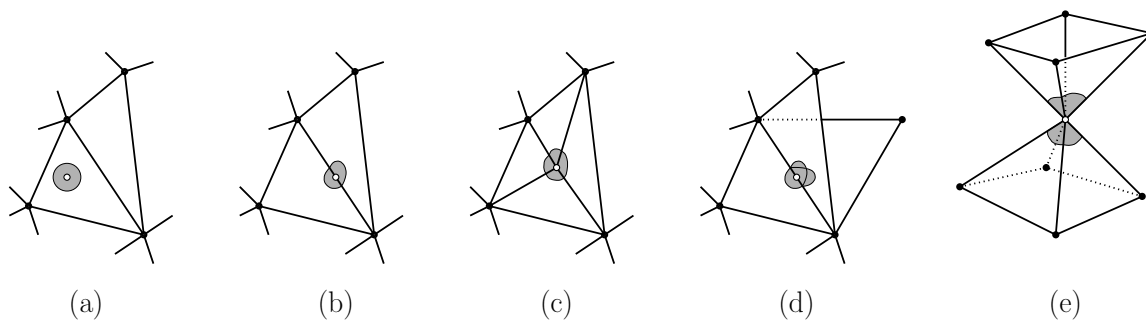


Fig. 4: Examples of cell complexes that are 2-manifolds (a)–(c), and those that are *not* 2-manifolds (d)–(e).

**Doubly-connected Edge List:** What sort of data structure can be used for storing triangle meshes? As far as OpenGL is concerned, a simple index array is sufficient. But your game program may require more structure. For example, suppose that you are using a 2-manifold to represent a terrain, and bug is walking across this terrain. As the bug walks leaves one triangle, we would like to be able to determine efficiently which new triangle it is entering. One way to do this would be “walk” around the edges of triangles of the mesh that the bug visits (see Fig. 5). In order to do this, we need to know which edges are adjacent to each triangle, and for each edge, we need to know what triangle lies on the other side of this edge.

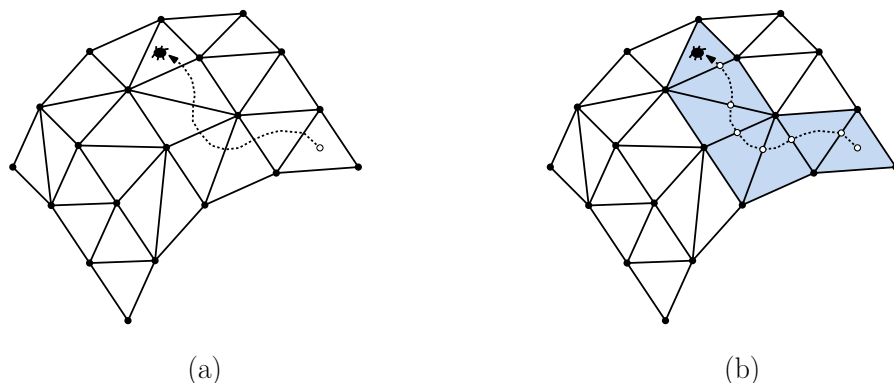


Fig. 5: Walking a bug on a mesh.

There are a number of different data structures for doing this. These include the *winged-edge data structure*, the *half-edge data structure*, and *doubly-connected edge list*, and the *quad-edge data structure*. All of these structures are equivalent, in the sense that given one, it is an easy matter to convert it into any of the others. We will discuss the *doubly-connected edge list* (or DCEL).

In the DCEL of a mesh, there are three sets of records one for each element in the cell complex: *vertex records*, *edge records*, and *face records*. For the purposes of unambiguously defining left and right, each undirected edge is represented by two directed *half-edges*.

**Vertex:** Each vertex stores its coordinates, along with a reference to any incident directed edge that has this vertex as its origin, `v.inc_edge`.

**Edge:** Each undirected edge is represented as two directed edges. Each edge has a reference to the oppositely directed edge, called its *twin*. Each directed edge is implicitly associated with two vertices, its *origin* and *destination*. Each directed edge is also implicitly associated with two faces, the one to its left and the one to its right.

Each edge stores a reference to the origin vertex `e.org`. (Note that we do not need to store the destination vertex, since it may be computed as `e.twin.org`.) Each edge also stores a reference to the face to the left of the edge `e.left`. (Again, we do not need to store the face to the right, since it can be computed as `e.twin.left`.) We also store the next and previous directed edges in counterclockwise order about the incident face, called `e.next` and `e.prev`, respectively.

**Face:** Each face  $f$  stores a reference to an arbitrary directed edge such that this face lies to the left of the edge, called `f.inc_edge`.

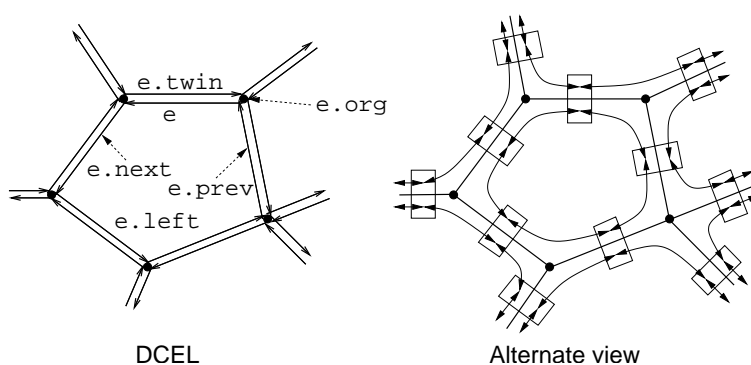


Fig. 6: Doubly-connected edge list.

Fig. 6 shows two ways of visualizing the DCEL. One is in terms of a collection of doubled-up directed edges. An alternate 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 reference provide links around each face of the polygon. The next references are directed counterclockwise around each face and the prev references 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. As an example of how to use the DCEL, suppose that we wanted to enumerate the vertices that lie on some face  $f$  in counterclockwise order. The code block below shows how to do this.

---

Vertex enumeration about a face using DCEL

```

verticesOnFaceCCW(Face f) {
    Edge e = f.inc_edge;           // any edge such that f is to the left
    do {
        output e.org;              // output e's origin vertex
        e = e.next;                // next edge about f in CCW order
    } while (e != f.inc_edge);    // done when we return to start
}

```

---

Let's try a slightly trickier one. Suppose that you are at a vertex  $v$ , and you wish to list all the vertices that are adjacent to  $v$  in counterclockwise order. First, we would access `v.inc_edge` to find any edge  $e$  that has  $v$  as its origin. The vertex on the other side of this edge is given as `e.twin.org`. Next, how would we determine the next adjacent vertex in counterclockwise order? First, observe that `e.prev` is the reverse of the next edge in counterclockwise order emanating from  $v$ . We then reverse this edge to obtain the desired next edge, `e.prev.twin`. Here is the code:

---

Adjacent vertex enumeration about a vertex using DCEL

```
adjacentVerticesCCW(Vertex v) {  
    Edge e = v.inc_edge;           // find starting edge  
    do {  
        output e.twin.org;         // output vertex on opposite end of e  
        e = e.prev.twin;           // jump to the next edge in CCW order  
    } while (e != v.inc_edge);     // repeat until looping back  
}
```

---

(As an exercise to see whether you understand this, you might try repeating these two enumerations, but this time do it in clockwise order.)