Triangle Meshes

In this chapter, we define a subclass of triangle meshes, which we call **smeshes**. We introduce simple data structures for representing smeshes, low-level operators for building and accessing these representations, and algorithms for constructing and traversing smeshes. These primitive tools will provide us with a common framework for the presentation of several compression approaches. They also lead to extremely simple implementations of algorithms for the compression, simplification, and analysis of triangle meshes. Hence, we discuss these tools and concepts in detail. In subsequent chapters, we explain how these simple concepts can be extended to more general triangle and polyhedral meshes.

1 - An initial focus on smeshes

Even though a variety of representations can be used to model 3D shapes, we have chosen to focus most of the chapters of this manuscript on triangle meshes for several reasons. Triangle meshes are popular in several application domains such as graphics, video-games, medical and scientific visualization. They are well supported by software and hardware graphics systems. Finally, they may be easily derived from most other representations by exact or approximating conversion algorithms.

Furthermore, we have chosen to initially restrict our attention to a particular class of triangle meshes, which we will call **smeshes**, where the "s" stands for "simple" or more precisely for "**homeomorphic** to a sphere". We will first study compression techniques for smeshes, because properties of smeshes can be exploited to simplify the compression algorithms and the theoretical analysis of the compressed file size. This simplicity will help the reader to internalize the fundamental concepts, terminology, and theoretical results before moving to more complex approaches that work on a broader class of models. The simplicity of smeshes will also lead to a trivial implementation of state-of-the art lossless compression and decompression algorithms.

Later, we will develop the terminology for discussing broader classes of meshes and present simple extensions of these compression and decompression technologies to meshes with handles, holes, non-manifold singularities, and even to simplicial and selective complexes. Many of these extensions will use pre- and post-processing steps to temporarily convert more general triangle meshes into structures that can be modeled as an smesh, and hence compressed using techniques developed for smeshes.

2 - Geometry and connectivity of a mesh

A **triangle mesh** M (for simplicity we will use the term **mesh**) is usually defined by specifying the set G of its **vertices** and a set T of its **triangles**. The vertices are sometimes called **samples** and the triangles are sometimes called **faces**. G is often referred to as the **geometry** of the mesh. T (or more precisely the information defining T from G) is sometimes referred to as the **topology** of the mesh, although we prefer to use the term **connectivity** here and reserve the term topology for referring to the global topological properties of the surface or graph represented by the mesh (such as the number of connected components or the genus of a component).

Let |G| (respectively |T|) denote the cardinality of G (respectively T), i.e., the number of vertices (respectively triangles) of M. |G| and |T| are often used to express the complexity of the mesh and also the size of a particular representation or compressed file. As we will see, there is a simple linear relation between |G| and |T| for a large class of meshes, including smeshes.

Each **vertex** v∈G is typically represented by its three **coordinates** (v.x,v.y,v.z) in some Cartesian coordinate system. Without loss of generality, the vertices may be ordered in some arbitrary fashion and each assigned a different integer ID, v, in [0 … |G|–1]. Hence, one usually stores the vertex coordinates in an **array** G[|G|][3]. We will use the notation v.x as a short for G[v][0], v.y as a short for G[v][1], and v.s as a short for G[v][2]. Hence, you may think of v as an object representing a particular vertex of M or, if you prefer, as an integer representing the ID of that vertex in the table representation of G.

3 - Incidence graph and its storage requirement

A **triangle** is defined by the IDs of the three vertices that it interpolates. We say that the triangle is **incident** upon these three vertices. Without loss of generality, the triangles of a mesh may be ordered and assigned integer IDs in [0…|T|–1]. For now, we will assume that the order is unimportant, although we will revisit this issue when we discuss **streaming** strategies. Consequently, the connectivity (i.e., these three indices per triangle) may be stored in a **table** V[|T|][3]. We will use the notation t.0 as a short for V[t][0], t.1 as a short for V[t][1], and t.2 as a short for V[t][2]. Therefore, you may think of t as an object representing a particular triangle of M or as an integer representing the ID of that triangle in the table V.

We will use the term **incidence graph** when referring to a representation of the connectivity. Note that our representation of the incidence graph (namely the G and V arrays) is a special case of the **Indexed-Face Set** representation popular in computer graphics, which defines each face (triangle, quad, or simple pentagon) of a polygonal mesh by the number of its bounding vertices and by the list (ordered set) of the indices of these vertices.

When the coordinates of each vertex are represented as floating points, the **storage required** for G is 3×32×|G| bits. If the vertex IDs are represented as 32-bit integers, the storage required for V is 3×32×|T| bits. As discussed below, there are roughly twice as many triangles as vertices (|T|≈2|G|). Hence, the storage size is roughly 9×32×|G|. To better compare storage requirements and compression results, we often report them as the average number of **bits per triangle** (abbreviated **bpt**) or as the average number of **bits per vertex** (abbreviated **bpv**). The representation discussed above requires about 144 bpt (i.e., 288 bpv). The compression solutions discussed in this book reduce this storage to less than 10 bpt in most cases encountered in practice.

4 - Benefit of connectivity over point-based models

A sufficiently dense **cloud** of **surface samples** (points on the surface of an object) can be used to represent a 3D solid S. However, in general, such a **point-cloud** representation is ambiguous, meaning that, without further information, one may not be able to decide whether a point p lies in S or not. Nevertheless, point-clouds provide a reasonably precise visual impression of the object. The **Point-Based Rendering** (**PBR**) community is focused on developing techniques for rendering 3D shapes using only such surface samples and possibly associated **normals** (unit vectors orthogonal to the surface of S at the sample). This approach has several advantages over rendering triangle meshes. Firstly, there is no need to store, transmit, or process any connectivity information. Secondly, the progressive transmission of point clouds is simpler to program than the progressive transmission of triangle meshes because it may be accomplished by selecting a regularly distributed sub-sampling of the points, transmitting them first, and then, if needed, transmit the rest of the samples, either all at once or in small batches, to increase the accuracy as the viewer is approaching the object.

However, to ensure that the rendered image is correct, i.e. does not have holes or other sampling artifacts, the density of samples must be nearly uniform and sufficiently high relatively to the projected area on the screen, even in flat portions of the mesh. Specifically, one must ensure that the space between sample projections on the screen is completely filled when each sample is rendered as a small disk (**splat**). In order to eliminate the cost of transmitting and rendering dense sets of samples of flat or nearly flat regions that could be represented by a few large triangles rather than by a large number of samples, one may use a triangulation, which specifies how distant samples should be interpolated by a continuous piecewise flat surface. Hence, meshes have two advantages over point clouds: (1) they provide an unambiguous model of the desired surface and (2) they make it possible to sub-sample the surface in nearly flat regions, resulting in fewer samples (i.e., vertices) to transmit and render, which translates to faster transmission and increased graphics performance.

Because algorithms that automatically construct a triangle mesh interpolation of a point-cloud exist, one may consider storing and transmitting the only point-cloud and invoking these algorithms to restore the triangle information when needed. However, point-cloud triangulation is computationally expensive. Furthermore, as we shall see, the connectivity of a triangle mesh may be compressed to about 1 bit per triangle—a small overhead above the cost of transmitting the geometry. Once decoded, connectivity information may be used by the decompressor to predict the location of the next vertex and hence to improve geometry compression. So, although the results may depend on the model and on the compression schemes used, transmitting connectivity (rather than recovering it from a point-cloud) will in general reduce the overall file size.

5 - The corners of a triangle

Note that the incidence graph may be thought of as a boundary graph having triangle (face) and vertex nodes and having **links** (oriented edges) from triangle nodes to vertex nodes. We say that an oriented link leaves a triangle and reaches a vertex. The **valence** of a vertex v is the number of links that reach v. Similarly, the valence of a triangle t is the number of links that leave t. In an smesh, each triangle has valence 3. In most meshes encountered in practice, the vertex valence is larger than 2 and rarely exceeds 9. Its average is close to 6.

Each link represents the association of a triangle t with a vertex v. We will use the term **corner** to refer to such an association. You may think of a corner c as the corner of t at which t touches v. Alternatively, you may think of a corner as an entry in the V array. Indeed, there are |T|×3 entries in the V array and each one is associated with a triangle t and defines one of the vertices bounding t. Hence, a particular corner will be identified by two integers (t,k), where t identifies the triangle and where k is 0, 1, or 2 and identifies one of the vertices of t.

To make things more precise, let us first define several **corner operators** which will help us to move from one corner to a neighboring corner on a mesh and to access the corresponding triangle or vertex. Then, we will use these operators to express mesh properties and algorithms for traversing the mesh, for renumbering its triangles and vertices, for compressing its connectivity, and for predicting and compressing its geometry. Finally, we will propose a simple data-structure for the mesh connectivity, which we call the Corner Table.

We will use the notation c.t for the **triangle of corner** c and c.v for its **vertex**. Assume a cyclic order of the three corners of a triangle. The notation c.n defines the **next corner** coming after c in the cyclic order of the corners of triangle c.t. Similarly, c.p denotes the **previous corner**. Hence, c.p may be computed as c.n.n. Note that we are cascading the object-oriented semantic, so that the notation c.p.q stands for (c.p).q. Also note that for any triangle, there are exactly two cyclic orders. For example, consider a triangle t that interpolates the vertices i, j, and k. Consider corners a, b, and c, such that a.v=i, b.v=j, and c.v=k. If we decide that a.n=b, then we also have b.n=c and c.n=a. Otherwise, we can decide that a.n=c, in which case we also have c.n=b and b.n=a. This choice is called the **orientation** of the triangle. It is usually encoded by the order in which the vertex IDs are stored in the table V. We will discuss below a technique for ensuring that this order is consistent for all triangles with a global orientation of the mesh, but for now, assume that it is arbitrary.

6 - Computing vertex normals

Many algorithms discussed in this book are expressed in terms of corners. Several rendering and geometry processing algorithms access one triangle or one vertex at a time in random order. Hence, we express such algorithms using constructs such as “`for each corner c do`”. The data structure (G and V tables) and operators (c.t, c.v, c.n, c.p) defined above suffice to support simple and efficient implementations of these algorithms.

For instance, to compute the vertex normals, we may use v.n (stored as an entry N[v] in table N) to store the normal vector associated with vertex v. We initialize all v.n to the null vector. Then, we execute “`for each corner c do {c.v.n+=normal(c.v,c.n.v,c.p.v);};`”, where the procedure `normal(i,j,k)` returns the cross product (G[V[j]]–G[V[i]])×(G[V[k]]–G[V[i]]), a vector orthogonal to the triangle with a magnitude proportional to the area of the triangle. Note that we assume that all triangles have a consistent orientation. Finally, we normalize each v.n. Note that we did not need to access the neighbors of a vertex or triangle.

7 - The opposite corner

Compression algorithms, and algorithms that identify the connected components of a triangle mesh, visit the mesh by walking from one triangle to an adjacent one, over their common edge. To implement them, we use corners and corner-operators rather than triangles or vertices. Indeed, a triangle has three **neighbors**, so we would have to somehow specify which neighbor we want to visit next. Similarly, a vertex has several neighbors. A corner unambiguously defines both a vertex and one of its incident triangles. Hence, if a corner c of a triangle t has been identified, we can easily say go to an adjacent triangle u by crossing the edge e of t that is not incident upon the vertex of c. The two vertices of e are common to both u and t. In fact, we may say, continue walking on u until you reach the opposite corner, defined as the corner of u whose vertex is not bounding the crossed edge. To make this operation inexpensive, we will pre-compute and cache with each corner c the reference c.o to its opposite corner.

If only the incidence graph is stored (i.e. the V table), one must identify the desired neighbor u of t by searching the V table for a triangle u with the desired vertex IDs. To avoid this search cost, we cache the result of such a search. Since we base our algorithms on corners, we cache the id of the opposite corner. Specifically, we say that corners c and o are **opposite corners** if ((c.n==o.p)&&(c.p==o.n))||((c.n==o.n)&&(c.p==o.p)). In other words, the unordered sets of vertices {c.n.v,c.p.v} and {o.n.v,o.p.v} are identical. In general, a corner may have several opposites. However, in smeshes, and in fact for the broader class of **manifold** and **pseudo-manifold** meshes (discussed later), each corner c will have a unique **opposite corner**, which we will then denote c.o. We will describe below a simple data structure (the array O) for caching c.o.

8 - Connected manifolds

Now, we define more precisely the class of smeshes by specifying their topological properties.

A mesh M is **edge-manifold** when (i.e. if and only if) each corner has a unique opposite. From now on, we will assume that M is edge-manifold and hence that each corner c will have a unique opposite corner denoted c.o.

Let us assume that M is edge-connected. A subset M' of a mesh M is **edge-connected** when for each pair of corners b and c of M' we can express b as a cascade of operators in the set {n, p, o} applied to c. For example, we could have b=c.n.o.n.o.p.o.n. Note that the path (i.e. sequence of operators) is not unique. We are merely interested in its existence. Intuitively, a mesh is edge-connected if one could walk from any triangle to any other triangle in the mesh by only crossing edges between adjacent triangles. From now on, for simplicity, we will assume that the mesh is edge-connected. Note that if an edge is not edge-connected, we can identify its edge-connected components and process them one at a time, paying attention to the fact that two component may share one or more vertices.

The **star** of a vertex v, denoted v.star, is the set of triangles incident upon v. An edge-manifold mesh M is said to be **manifold** when, for each vertex v of M, v.star is edge-connected. Hence, in a manifold mesh, the triangles incident upon any given vertex form a simple edge-connected cycle (also called triangle **fan**). From now on, unless explicitly specified, we will assume that the mesh is edge-connected and manifold. Note that if we are given a manifold mesh, we can decompose it into its edge-connected components knowing that each vertex will belong to exactly one components, since it is manifold.

9 - Swirls

To orient and compress the mesh, we will use a procedure that starts at a seed triangle and visits all triangles by recursively walking from one triangle to its not-yet visited neighbors. We use the term **swirl** to describe this process, because, when the mesh is oriented (as explained below), it tends to visit the triangles in a spiraling order making swirling spiral trajectories. In this section, we do not assume any mesh orientation, so the traversal may look more chaotic. In order to keep track of which triangle we come from and to distinguish the other two that we may walk to, swirl uses corners. To keep track of which triangles have been already visited, it marks them, by setting t.m, which is a Boolean stored as entry M[t] in the table M[|T|]. Hence, to traverse the mesh, we reset M and call swirl(s) for some arbitrary seed corner s. A recursive implementation of swirl may be formulated as follows:

```
    void swirl (corner c) {if (!c.t.m) {c.t.m=true; swirl(c.l); swirl(c.r);}; }
```

Note that for simplicity, we have used the shortcuts c.l for c.p.o and c.r for c.n.o. We will propose later an implementation that avoids most of the recursion calls. An example of the initial steps of a swirl is shown below assuming a consistent orientation.

10 - Orientation

A manifold mesh is **oriented** when c.n==c.o.p for all corners c.

A manifold mesh is **orientable**, if there exists an assignment of triangle orientations that makes the mesh oriented. Note that some manifold meshes are not orientable. For example, the Klein bottle is not. Such meshes do not have a valid imbedding in the three dimensional space. From now on, we will assume that the mesh is orientable.

Some file formats do not enforce a consistent orientation and simply list the IDs of the three vertices of each triangle in an order that does not ensure the c.n==c.o.p condition. Hence, we must be able to orient the mesh. To do so, we may have to **flip** the orientations of some triangles, by swapping two of the IDs in the list of their 3 vertex IDs. Which triangles should we flip? We can accomplish this by a modified version of swirl. We pick a **seed** triangle, mark it as visited, and call `orient(s)`, shown below, for one of its corners s.

```
void orient (corner c) {
 if (!c.t.m) {c.t.m=true;
           if (c.n!=c.o.p) {flip(c);}; orient(c.l); orient(c.r);}; }
```

The procedure `flip(c)` swaps the c.n and c.p entries and updates their opposites as follows:

```
void Flip(c) {corner b; b=c.n; c.n=c.p; c.p=b; c.n.o.o=c.p; c.p.o.o=c.n;};
```

11 - Imbedding and pseudo-manifold representations

The operators and properties of the mesh discussed so far have been formulated independently of the imbedding (location of the vertices). In fact, the representations and many of the connectivity compression techniques discussed in this book are also independent of the imbedding. Hence, they may be used even if the imbedding is invalid (i.e., when the incidence graph does not capture the topology of the actual surface defined as the union of the triangles of the mesh). For instance, in an invalid imbedding, two vertices with different IDs may have the same location. Hence, a triangle incident upon them will have zero area or the corresponding point will be non-manifold. A vertex may lie on a triangle that, according to the incidence graph, is not incident upon it. Two triangles may intersect. And so on.

This flexibility of being able to use a manifold incidence graph to represent invalid imbeddings is exploited when the mesh is transformed or simplified in a manner that may produce self-intersections or when a manifold graph is used to represent a non-manifold mesh. In the later case, we say that a **pseudo-manifold** representation is used.

12 - Counterclockwise orientation

Many graphics systems assume **counterclockwise** orientation, which implies that if you are outside of the finite solid bounded by the mesh, the triangles you see will appear counterclockwise. More precisely, if you trace a vector on the screen from the center of a triangle c.t to a first vertex c.v.g and then rotate that vertex to c.n.v.g and then to c.p.v.g, the vector is rotated by 360 degrees in the counterclockwise direction.

We use the process below to check whether the mesh is counterclockwise. If not, we will flip all the triangles.

Let mp(U,V,W) denote the **mixed product** (U×V)•W. Note that mp(U,V,W) is the determinant of a matrix with these vectors as columns. Hence mp(U,V,W) = mp(W,U,V) = –mp(V,U,W). Consider an arbitrary point P. For numerical accuracy, we usually compute P as the average of the vertices of M.

For each triangle t_i of the mesh, we compute v_i=mp(PA,PB,PC), where A, B, and C are the locations of the three vertices of t_i, listed in the order that respects the orientation t_i (obtained by the `orient` procedure described above). Note that the absolute value of v_i is 6 times the **volume of the tetrahedron** (A,B,C,P) and that it is positive when the A, B, and C ordering appears clockwise from P.

Let v be the sum of all these |T| signed scalar quantities v_i. The mesh is oriented counterclockwise when v is positive and clockwise when v is negative. Unless the mesh is flat or self-intersecting, v cannot be zero. In fact, the absolute value of v is 6 times the **volume of the solid** (finite set) bounded by the mesh.

13 - Smeshes are planar triangle graphs

A **smesh** is an edge-connected manifold triangle mesh endowed with a counterclockwise orientation and the a linear relation between the number |T| of its triangles and the number |G| of its vertices, listed as Equation 1 below.

Equation 1: |T|=2|G|–4

For example, a tetraheron is an smesh, since its triangles form an edge-connected set, since it is manifold, since it is orientable, and since it satisfies Equation 1 (4=2×4–4).

Equation 1 implies that the mesh is a **planar triangle graph**. Recall that a simple graph is planar if its vertices and edges can be drawn on the plane so that no two edges intersect, except of course at their common vertex. Note that having a planar graph does not imply that the vertices of the mesh all line in some plane nor that one has to produce such a drawing. Furthermore, even though one of the faces (region delimited by the edges) in the drawing has infinite area, we still call it a triangle, since it is bounded by three edges and three vertices.

Equation 1 may be derived from the **Euler formula**, shown as Equation 2 below, which holds for more general (non-triangular) connected simple planar graphs.

Equation 2: |F|–|E|+|G|=2

Here, |F| denotes the number of faces and |E| denotes the number of edges. Equation 2 may be proven by induction, showing that any planar graph may be transformed into a single point by a series of transformations which either delete an edge or delete a pendent vertex and its incident edge and preserve both the connectivity of the graph and Equation 2. To obtain Equation 1, we assume that all faces are triangles and hence use |T| instead of |F|. We obtain Equation 1 by noting that each triangle uses 3 edges, hence there are 3|T| edge-uses, and that each edge is used twice (once by each one of its two incident triangles), hence there are 2|E| edge uses. From 3|T|=2|E|, we obtain |E|=3|T|/2, which by substitution in Equation 2, yields Equation 1.

Because an smesh is a planar graph, it has no **handles** (i.e., it has **genus** zero). This observation is important, since the presence of handles requires a modification of some of the compression algorithms designed for smeshes and leads to a storage overhead per handle.

14 - Corner Table representation and operators

So far, we assumed the existence of operators c.v, c.t, c.n, and c.o, which associate with each corner c its vertex c.v, its triangle c.t, the next vertex c.n after c in c.t according to the counterclockwise orientation of c.t, and the opposite corner c.o. From there, we have formulated other convenient shortcuts: c.p=c.n.n, c.l=c.p.o, and c.r=c.n.o. All are shown in (Figure 1).

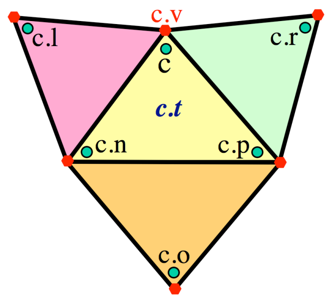

Figure 1. Using the V and O tables, given a corner, c, we can access: its triangle, c.t; its vertex, c.v; the previous and next corners, c.p and c.n, in c.t; the opposite corner, c.o; and neighboring corners c.l and c.r.

In this section, we describe a compact and convenient data structure for storing a triangle mesh and methods for supporting these corner operations efficiently.

Above, we have used two integers t and k to identify a corner, where t is the integer ID of a triangle and where k is an integer in {0,1,2} and identifies one of the three vertices of t. From now on, for simplicity, we will combine these two integers into one: c=3t+k. With this simple mapping, we can represent each corner by a single integer.

Given a corner c (represented by integer c), we can easily obtain its t and k values: t=int(c/3) is the integer part of the division of c by 3 and k=c%3 is the remainder of that division. Hence, we could use the two-dimensional array V as described above to store the vertex references and retrieve c.v as V[t][k]. For example, when c is 7, c.v is V[2][1]. However, to further simplify the discussion and the implementation, we will interpret the memory layout of V as a one-dimensional array of 3|T| entries. Note that these entries are stored in the order V[0][0], V[0][1], V[0][2], V[1][0], V[1][1], V[1][2], V[2][0], V[2][1], V[2][2], V[3][0], V[3][1], V[3][2]… in which they were stored in the original two-dimensional interpretation of V. Hence, the new format for V may be obtained either by mapping one data structure over the other or by copying the triangle-vertex incidence IDs from the old V table into a new one dimensional table V, respecting their order. Note that we no longer need to perform the integer divisions or the modulo for retrieving c.v. Instead, c.v stands for V[c]. The triangle c.t is `int(c/3)`. The next vertex, c.n, can be computed as `c.t+(c+1)%3`. Similarly, c.p is `c.t+(c+2)%3`.

To improve performance during repeated traversals of the mesh, we cache the c.o field for each corner c as the integer entry O[c] in the table O. We discuss in the next section how to fill the O table from the information contained in V. Note that V and O each have one entry per corner. Hence, we call this representation of the mesh connectivity the **Corner Table**.

For readability, we use the object-oriented notation, such as c.o.v. However, these expressions may be directly translated into a notation that uses arrays. For example, `c.o.v` stands for `V[O[c]]`. The same notation will be used when we wish to assign a new value to an entry of an array. For example, `c.l.o=c.r` translates into `O[c.l]=c.r`.

Finally, for book-keeping, we will also use one binary flag per vertex, stored in MV[|G|], and one binary flag per triangle, stored in MT[|T|]. We will use the same notation, .m, for both, since there is no ambiguity. Specifically, t.m will refer to MT[t] and v.m to MV[v]. For example, if we wish to mark the three vertices of triangle c.t as visited, we will write `{c.v.m=true; c.n.v.m=true; c.p.v.m=true;}`.

15 - Computing the O table

Given the V table, we wish to compute the entries in the O table. This may be achieved by the following procedure:

```
    for  (int c=0; c<3*|T|-1; c++) {
     for (int b=c+1; b<3*|T|; b++) {
           if ( (c.n.v == b.p.v ) && ( c.p.v == b.n.v) ) {c.o=b; b.o=c;};};}
```

Unfortunately, this simple procedure has quadratic complexity. A linear worst case complexity, and in practice a constant complexity, may be achieved by the following hashing and sorting process.

For sake of generality, we assume that we have the V table, but that the triangles may not have a consistent orientation. With each vertex v, we temporarily store a pointer, v.h, to an initially empty linked list of neighbors. We will refer to this list as L(v). Then, for each corner c, we compute v=min(c.n.v,c.p.v) and n=max(c.n.v,c.p.v). Then, we insert the pair (n,c) into L(v), while maintaining L(v) sorted. At the end of this process, for each vertex v, L(v) will contain an even number of entries. Furthermore, pairs of consecutive entries will have the same n element (same neighboring vertex ID). For example, L(v) may be {(n,a),(n,b),(m,c),(m,d)}. We simply make the corresponding corner opposite of each other. For example, `{a.o=b; b.o=a; c.o=d; d.o=c;}`

Once O is computed, we can use the `orient` algorithm proposed earlier to ensure a consistent orientation of all triangles.