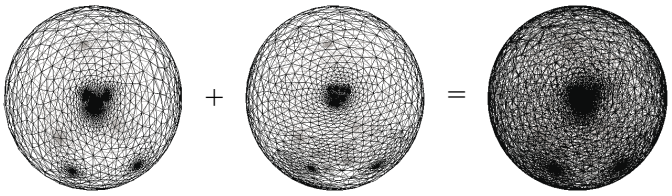
Chapter 3

Constructing representations



Given two embeddings W_0, W_1 of meshes $(V_0, K_0), (V_1, K_1)$ on a common domain D we aim at generating one mesh connectivity K with vertex positions $V(0)$,$V(1)$ so that the original shapes are reproduced, i.e.

$$\phi_V(0)(|K|) = \phi_{V_0}(|K_0|), \phi_V(1)(|K|) = \phi_{V_1}(|K_1|). \tag{3.1}$$

Note that the vertex positions $V(0), V(1)$ are already available using the barycentric coordinates of each vertex w.r.t. the base domain. These barycentric coordinates allow to map each vertex from one mesh to the other. However, the exact mapping of vertices onto the piecewise linear surface might lead to bad results. The next subsection discusses better alternatives for the absolute position of vertices.

The main point of this chapter is to establish the common connectivity K. The typical approach found in the morphing literature is to generate a supergraph of the connectivities K_0, K_1, i.e. one that contains the simplices of both plus additional vertices if edges cross. This graph is found by *map overlay*. Here, we distinguish two cases:

1. Bounded meshes embedded in a disk.

2. Unbounded meshes, assuming the geometries of several meshes are sufficiently close.

These cases stem from the parameterization methods presented in the previous chapter.

Looking at multiresolution techniques for meshes is an alternative way of generating a common connectivity is remeshing. In particular, the parameterization is exploited to map planar coordinates of refinement operators to coordinates on the surface of the shapes. Provided the base domain accurately represents sharp features of the meshes this approach has the advantage that it is much easier to scale. The size can be easily adapted to the desired precision. For the same reason this approach is easier to extend to more than two meshes.

3.1 Mapping parameter values to the surface

After the meshes have been parameterized it is easy to find the position of a particular vertex on the surface of a mesh. Assume we want to find the position of vertex $\mathbf{v_{1_i}}$ of the first mesh on the second mesh. We determine the vertices $\{\mathbf{w_{2_j}}\}$ comprising the face in the parameterization in which the parameter domain position $\mathbf{w_{1_i}}$ lies. Then, $\mathbf{w_{1_i}}$ is represented in barycentric coordinates with respect to $\{\mathbf{w_{2_j}}\}$:

$$\mathbf{w_{1_i}} = \sum b_k \mathbf{w_{2_{j(k)}}} \tag{3.2}$$

The position of $\mathbf{v_{1_i}}$ in the other mesh is found as

$$\mathbf{w_{1_i}}' = \sum b_k \mathbf{v_{2_{j(k)}}} \tag{3.3}$$

This is the exact position on the piecewise linear shape and the way used in most of the morphing literature.

However, this does not take into account the idea that piecewise linear shapes are (in most cases) just approximations of smooth shapes. Particular practical problems occur when normals have to be rebuild from these new geometric positions: Vertices inside a face get the face's normal. If standard rendering methods are used (vertex normals and Gouraud shading) this results in degenerate shading.

It would be advantageous to find positions which result in a smooth surface. More specifically, we would like to use the barycentric coordinates to find positions *over* a triangular face and not necessarily on the face. This calls for methods defining a smooth surface from a coarse mesh. An obvious choice for such a method would be subdivision (e.g., Loop subdivision [Loop & DeRose 1990] or Kobbelts $\sqrt{3}$-scheme [Kobbelt 2000]).

3.2 Map overlay data structure

We need a data structure to store the meshes, which allows to add and remove edges, gives quick access to topological information (e.g., the ordering of edges around a vertex), and is not too heavy in terms of storage. We choose the doubly connected edge list [Muller & Preparata 1978] (sometimes called twin-edge data structure). The basic data type of this data structure is the edge. Edges are stored as two directed half edges. More specifically, the following information is stored:

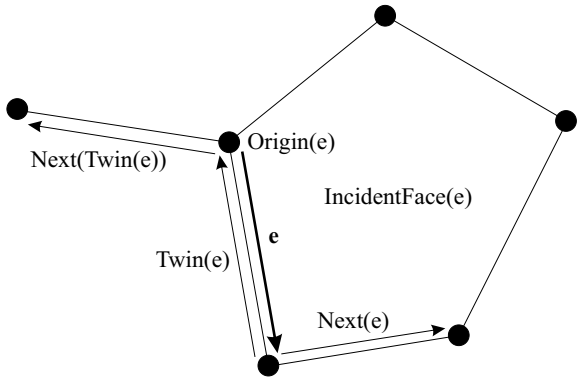


Figure 3.1: The doubly connected edge list.

Face The face record contains a pointer to an arbitrary half edge on its boundary.

Edge Each edge record contains pointers to

- its originating vertex,
- the face it bounds,
- the half edge connecting the same vertices but in the opposite direction (its *twin*),
- the next half edge along the boundary of the bounded face.

Vertex The vertex record contains a pointer to an arbitrary half edge originating from this vertex as well as location in space and other attributes (e.g., normal, color, texture coordinate).

Figure3.1 illustrates the data structure. Note that it is particular easy to iterate along the boundaries of faces (next pointers) or through all edges incident upon a vertex in their circular order (twin $\rightarrow$ next). A good description of the doubly connected edge list can be found in de Berg et al. [1997].

3.3 Open meshes embedded in a disk

Several algorithms were proposed for the problem of overlaying planar graphs - see a textbook [de Berg et al. 1997]. In general, the planar map overlay has the complexity $O(n \log n + k)$, where n is the number of edges and k is the number of intersections. If the two subdivisions are connected (as in our case) the planar overlay can be computed in $O(n + k)$ [Finke & Hinrichs 1995].

The general paradigm for planar overlay is *plane sweep*. Sweep algorithms process the input with a virtual line moving along its normal direction. Whenever a vertex intersects the sweep line the corresponding edge is added (the vertex is the

starting point of this edge) or removed (the vertex is the endpoint) from the list of active edges. The list of active edges is tested for intersection with added edges. To further reduce the number of necessary intersection tests the active edges are stored in their order along the sweep line. This is done by inserting edges in the correct position. In addition, the order has to be updated at intersection points. Using the ordering, only neighboring edges have to be tested for intersection. This processing leads to an algorithm with complexity $O(n \log n + k)$. By exploiting that two connected graphs are intersected the complexity can be reduced to $O(n + k)$.

In the case that meshes are embedded on the disk special care has to be taken for the boundaries of the meshes. While we assume that the embedding is surjective (i.e. fills the disk), the boundary in fact is a polygon leaving small empty regions between the disk and the polygon. However, it is clear that the boundaries of the meshes to overlay should be mapped onto each other. So in order to avoid that the boundary polygons intersect with inner edges of the other mesh the boundaries have to be merged first. This is done by simply connecting the vertices of all meshes on the disk along the linear order given by the disks boundary. After this boundary polygon has been established the planar mesh overlay procedure can be computed.

3.4 Closed meshes in arbitrary position

There seem to be only a few publications about the overlay of meshes in general position (i.e. the triangulated surfaces are close to each other but not e.g. planar). Note that plane sweep solutions are not applicable in this case. Few publications deal with overlaying two subdivision of the sphere. Kent et al. [1992] give an algorithm for the sphere overlay problem, which needs $O(n + k \log k)$ time. We have presented a solution to this particular problem, which *reports* the intersection of two spherical subdivisions in the optimum time of $O(n + k)$ [Alexa 2000]. Also, both algorithms exploit the topological properties of both subdivisions, which are used to guarantee the correct order of intersections. Here, we generalize these algorithms to work on two arbitrary shaped meshes, which are assumed to be sufficiently close to each other. We also alleviate the problem that the published version [Alexa 2000] had a worst case complexity of $O(n + k \log n)$ for the *construction* of the merged mesh using the already reported intersections.

The algorithm consists of two main parts: First, finding all intersections, and second, constructing a representation for the merged model.

3.5 Finding the intersections

In the algorithm two geometric functions are needed: One to decide if and where two edges intersect on the sphere, and a second to decide whether a point lies inside a face. Both geometric properties can be checked in a projection to the tangent

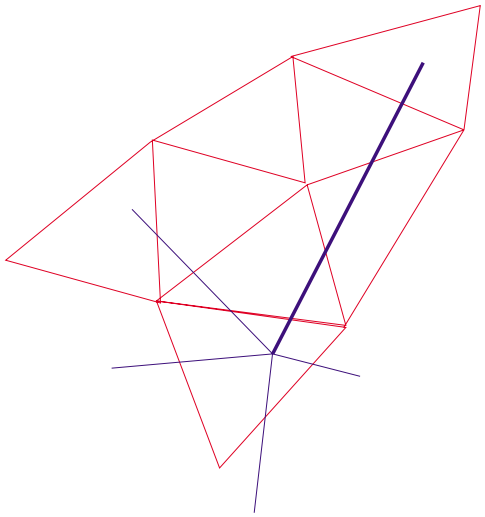


Figure 3.2: Edge-edge intersections are determined by following an edge (blue in this illustration) over the faces of the other triangulation (red). After finding an intersection the face-to-face coherence exploited and only the edges of the next face are tested.

plane of the surfaces. Since the meshes are supposed to be close in space their tangent planes should not differ to much. A suitable way of finding a common tangent plane is to take the cross product of two edges (i.e. the two edges to intersect, or two edges of the face to check).

The basic idea is to traverse the graphs breadth first, keeping information about the face that contains the current working edge and exploiting face-to-face neighbor information. Choose an arbitrary vertex $\{i\} \in K_0$ and search the 2-simplex $f = \{f_1, f_2, f_3\} \in K_1$ that contains it in under the bijective mapping. Start with an edge $e \in \mathcal{S}(i)$. Store e together with f on a stack. In general, the stack will always contain a directed edge together with the face in the other mesh containing the origin of this edge. The basic idea of the traversal is to walk over the faces following an edge (see Figure 3.2). Each edge $e = \{e_1, e_2\}$ is intersected first with the three edges $\{f_1, f_2\}, \{f_2, f_3\}, \{f_3, f_1\}$ bounding f, which contains $\phi_{W_0}(e_1)$. When an intersection is found the working edge e is emanating to the next face, i.e. the one that shares the intersected edge. This face is set to be f and is inspected in turn.

The same process is repeated with edges in K_1. This is necessary to find the topological orders of edges in K_0 cutting edges in K_1. Each edge is tested against three edges plus two additional intersection tests for each intersection being found. Thus, the algorithm has constant costs per edge and per intersection and the complexity is $O(n + k)$.

3.6 Generating the data structures

An appropriate data structure for storing the intersections is needed. Information about an intersection should be accessible from both intersecting edges at constant costs. We use a hashtable with edge indices as key values. When edge-edge intersections are found and stored in the intersection lists a pointer to the entry in the hashtable is stored. This means, both edges point to the same data structure containing information about the intersection (the intersecting edges in the beginning). The hashtable is only needed to access the entry when the intersections have already been computed by processing K_0 and need to be found when intersections from K_1 are generated. After reporting all intersections the hashtable is discarded.

The following two step algorithm constructs the merged mesh: First, edges in K_0 are cut. We iterate through the intersection list of an edge and cut the edge at each intersection point. Thus, a new edge (two half edges) are generated for each intersection. The new edge represents the part of the edge that has to be processed. At each intersection the data structure containing the respective information is updated to now contain the two parts of the edge incident upon the intersection point. At this point only the twin pointers of the half edges are updated. The next pointers are left empty.

Second, edges in K_1 are processed. As in the first step edges are cut into two pieces at each intersection point. However, this time also the next pointers are updated. This is done by using the information stored in the intersection data structure, which now contains both edges of the already cut edge in K_1.

After all intersections are processed in this way we have a valid vertex and edge lists of the embedding. It remains to compute the records for the faces. Note that faces created from intersecting triangles are convex polygons with 3 to 6 sides, which should be triangulated. This is another subtlety, which is more involved as it may seem: While the polygon resulting from the intersection is convex it is not clear what shape it has in other geometric configurations, e.g. those of the source meshes. In principle one should find a triangulation that is admissible in all source geometries. This might be difficult and could lead to the need for additional vertices. The problem is known as *compatible triangulation* and discussed in detail in another context in Section 4.4.1.

Note that it has been communicated that this approach could be extended to bounded meshes, however, boundaries require special treatment beyond the scope of this work.

3.7 Remeshing

A mesh is typically just an approximation of a shape. We have already seen that the mesh overlay process together with using coordinates lying exactly on the mesh might introduce artifacts into the source meshes (see Section 4.1). Thus, even if the original mesh connectivities are available as subsets of K the reproduction of the

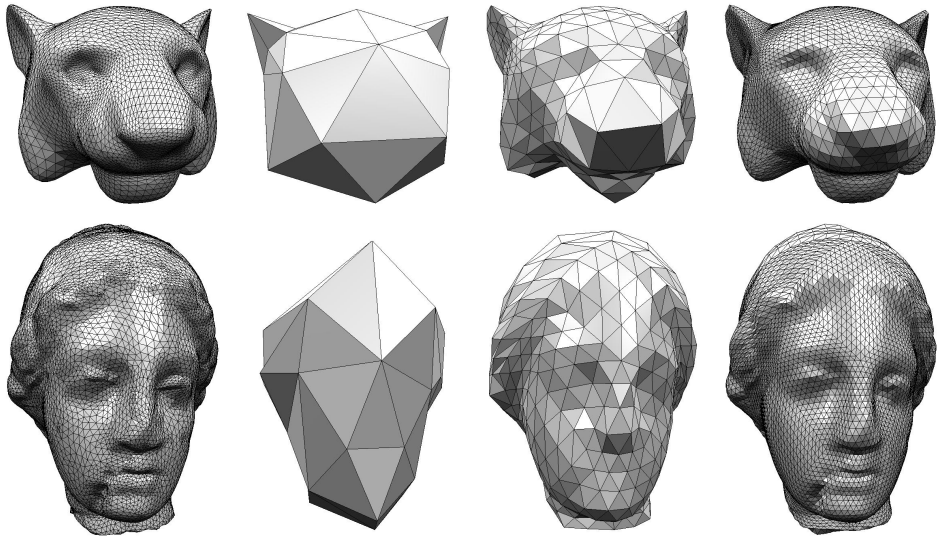


Figure 3.3: A multiresolution mesh representation build over the same base domain to represent two geometries. Reprinted from Michikawa et al. [2001].

original shapes though exact is not ideal. It seems that the perfect reconstruction of the source shapes is impossible and we could as well use any mesh connectivity to approximate both given shapes.

Remeshing techniques have been used to construct semi-regular meshes from irregular input [Lee et al. 1998]. The irregular mesh is reduced to an irregular base domain. The base domain is refined inserting only regular vertices. The idea is to use refinement operators as known from subdivision surfaces, however, without using the geometric rules attached to the refinement. Instead, geometric positions are found by exploiting the bijection between original surface geometry and the parameterization. For example, using the 1-4 split the parameter domain positions of inserted vertices are given as edge bisectors. This parameter leads to the coordinate on the surface of the mesh.

This idea has recently been used to construct morphable meshes by Michikawa et al. [2001] (see Figure 3.3). In this context, each parameter value leads to two coordinates. After several refinement steps a semi-regular mesh connectivity K is constructed together with coordinates $V(0), V(1)$ as desired. Because the refined connectivity is defined by the rules of the refinement used, only the base domain connectivity has to be stored explicitly.

To achieve a desired approximation accuracy, the number of refinement steps should be adapted to the geometric complexity of the meshes. Note that refinement could be done adaptively depending on the viewing conditions without necessarily computing and storing all coordinates of the refinement levels.

3.8 Comments

The remeshing approach is appealing because it allows to scale the size of the representation mesh. Its only limitation is the accurate representation of sharp features in the original shapes. In conventional multiresolution models this problem is alleviated by fitting the base domain to these features. In the context of morphing the base domain has to represent the features of several meshes, which do not necessarily coincide. This, again, incurs extra burden on the user, because a more complex base domain has to be induced on the input meshes. In addition, a more complex base domain limits the possibilities of automatic feature alignment methods. However, the flexible and lean representation mesh seems worth it.