Chapter 6

A Software Implementation: Detri2

Introduction

In this chapter, we describe the implementation of triangular mesh generation algorithms. This includes mesh data structures, local mesh operations (vertex insertion, vertex deletion, and edge flips), robust and efficient exact geometric predicates, and spatial point sorting.

We will describe a software implementation of a two-dimensional triangular mesh generator – `Detri2`. It is a C++ program to generate (weighted) Delaunay triangulation for a set of (weighted) points, to generate quality triangular meshes for a 2d polygonal domain. It implements various algorithms for generating (weighted Delaunay) triangulations, constrained (weighted Delaunay) triangulations, constrained Delaunay refinement as well as algorithms for (anisotropic) mesh adaptation. The source code of `Detri2` is freely available from the author's website.

6.1 A Triangle-based Data Structure

A triangulation (mesh) data structure should support iterating through the entities of a triangulation, performing queries on incidence (like element-vertex), adjacency (element-element), and classification (in particular, boundary classification) of entities, and modifying the topology of the triangulation. A basic requirement for it is that it can perform every incidence, adjacent, and classification query in a time independent of mesh size.

There are several data structures have been proposed in literatures, Floriani and Hui [10] gave a nice review of data structures for simplicity complexes, including manifold and non-manifold cases. In this section, we describe a triangle-based data structure which is particularly suitable for triangular mesh generation.

We first describe a core piece of the data structure which is a representation of the symmetry group of the standard triangle. Its main function will be to keep track of direction and orientation when we navigate the triangulation. We then describe the representation and storage of this data structure.

6.1.1 Ordered triangles

Let $t_{\mathbf{abc}}$ be a triangle with vertices $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$. There are three even permutations of the three vertices of a triangle, which are $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, $\{\mathbf{b}, \mathbf{c}, \mathbf{a}\}$, $\{\mathbf{c}, \mathbf{a}, \mathbf{b}\}$, respectively. We call each permutation an *ordered triangle*. Another way to think of the three ordered triangles is that they are the three directed edges which travel the boundary of this triangle in counterclockwise direction, see Figure 6.1.1.

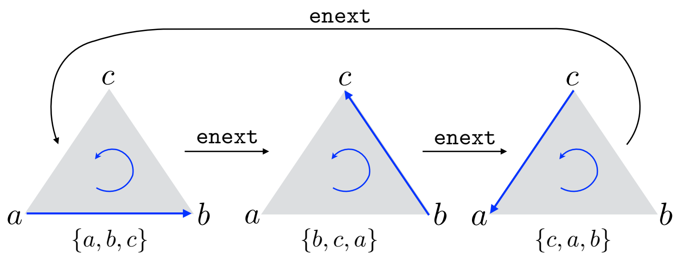


Figure 6.1: The three ordered triangles and the `enext` operation on them.

We use cyclic shift to move between them. As illustrated in Figure 6.1.1, the cyclic shift from $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ to $\{\mathbf{b}, \mathbf{c}, \mathbf{a}\}$ corresponds to advancing the leading edge to next position, from $\{\mathbf{a}, \mathbf{b}\}$ to $\{\mathbf{b}, \mathbf{c}\}$. We define the `enext` function to move between different ordered versions of the same triangle. Denote `E` as an ordered triangle $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, let `F` be $\{\mathbf{b}, \mathbf{c}, \mathbf{a}\}$, and `G` be $\{\mathbf{c}, \mathbf{a}, \mathbf{b}\}$. Then

- `E.enext` = `F`, and
- `E.enext`^{2} = `G`,

where `E.enext`^{2} is a short notation for `(E.enext).enext`.

We define the `org` function to get the origin vertex of the leading edge of an ordered triangle:

- $\mathbf{a} = \texttt{org}(t_{\mathbf{a},\mathbf{b},\mathbf{c}})$.
- $\mathbf{b} = \texttt{org}(\texttt{enext}(t_{\mathbf{a},\mathbf{b},\mathbf{c}}))$.
- $\mathbf{c} = \texttt{org}(\texttt{enext}(\texttt{enext}(t_{\mathbf{a},\mathbf{b},\mathbf{c}})))$.

The `enext` and `org` functions operate on the edges and vertices of a triangle. We next define the `esym` function which operates on an ordered triangle, $t_{\mathbf{a},\mathbf{b},\mathbf{c}}$, and returns its adjacent ordered triangle, $t_{\mathbf{b},\mathbf{a},\mathbf{d}}$. They belong to two triangles which share at the save edge in a triangulation.

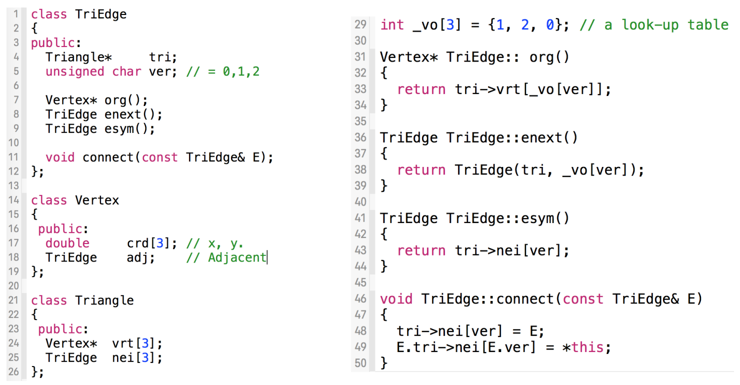
- $t_{\mathbf{b},\mathbf{a},\mathbf{d}} = \texttt{esym}(t_{\mathbf{a},\mathbf{b},\mathbf{c}})$.

Note that the function `esym` will only work correctly if the mesh data structure is correct, i.e., it connects the right adjacent triangle at each edge. It then can be used to test the correctness of the mesh data structure.

6.1.2 Implementation

We now describe a C++ implementation of the triangle-based data structure. Figure 6.2 Left shows the basic data structures realised in C++. The class `TriEdge` represents an ordered triangle. It is a pair, a reference `tri` to a `Triangle`, and a two-bit integer `ver`, ranged from 0 to 2, identifying the ordered version of the triangle.

The class `Triangle` stores the information of a triangle. It has an array `vrt[]` of three references (C++ pointers) to its vertices (`Vertex`) and an array `nei[]` of three ordered triangles (`TriEdge`) of its adjacent triangles,



```
class TriEdge
{
public:
  Triangle*     tri;
  unsigned char ver; // = 0,1,2

  Vertex* org();
  TriEdge enext();
  TriEdge esym();

  void connect(const TriEdge& E);
};

class Vertex
{
 public:
  double     crd[3]; // x, y.
  TriEdge    adj;    // Adjacent
};

class Triangle
{
 public:
  Vertex*  vrt[3];
  TriEdge  nei[3];
};
```

```
int _vo[3] = {1, 2, 0}; // a look-up table

Vertex* TriEdge:: org()
{
  return tri->vrt[_vo[ver]];
}

TriEdge TriEdge::enext()
{
  return TriEdge(tri, _vo[ver]);
}

TriEdge TriEdge::esym()
{
  return tri->nei[ver];
}

void TriEdge::connect(const TriEdge& E)
{
  tri->nei[ver] = E;
  E.tri->nei[E.ver] = *this;
}
```

Figure 6.2: The Triangle-based data structure.

such that `nei[i]` stores the adjacent ordered triangle at the edge opposite to `ver[i]`, $i = 0, 1, 2$, see Figure 6.3.

The class `Vertex` contains an array `crd[]` of its geometric coordinates, where `crd[0]` and `crd[1]` are its x- and y-coordinates, respectively. It also contains an ordered triangle `adj`, which points to one of its incident triangles (from which any other incident triangle can then be accessed).

Figure 6.2 Right shows an implementation of the primitive functions defined in the class `TriEdge`. The functions `org`, `enext` can be implemented by increment the `ver` value with a modulo 3 operation. Hence it can be efficiently done by a fast look-up table. The function `connect` builds the connection between two triangles. It assumes that the input `TriEdge E` must share the same edge of the current one.

6.1.3 The infinite vertex and infinite triangles

For various reasons, it is convenient to think that the complement of the convex hull of a triangulation in the plane is also a part of this triangulation, so that this triangulation covers the whole plane and it becomes a sphere (which has no boundary). Such a “complete” triangulation has many advantages which make it useful in algorithm implementation. For example, every edge in this triangulation will have two adjacent triangles. The star or link of any boundary vertex is the same as interior vertices.

To realise this property, one adds to any triangulation a fictitious vertex,

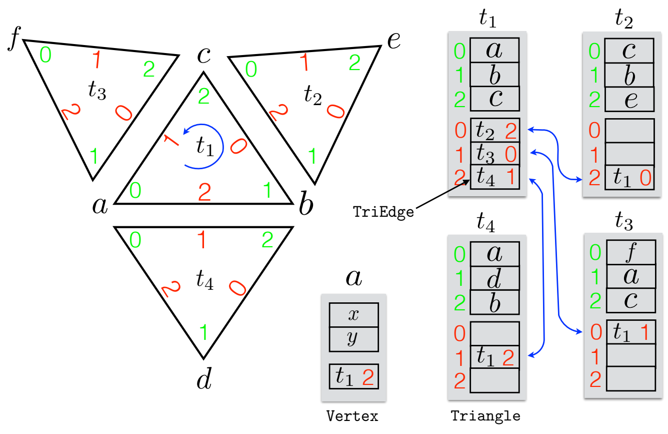


Figure 6.3: The Triangle-based data structure.

called *infinite vertex*, with the convention that every edge on the convex hull forms an *infinite triangle* with this vertex. The augmented triangulation is combinatorially equivalent to a triangulated 2-dimensional sphere which has no boundary. The infinite vertex has no geometric coordinates, it does not participate in the geometric computations with other vertices of this triangulation. In general, an algorithm can treat an infinite triangle as a real triangle and operate on it.

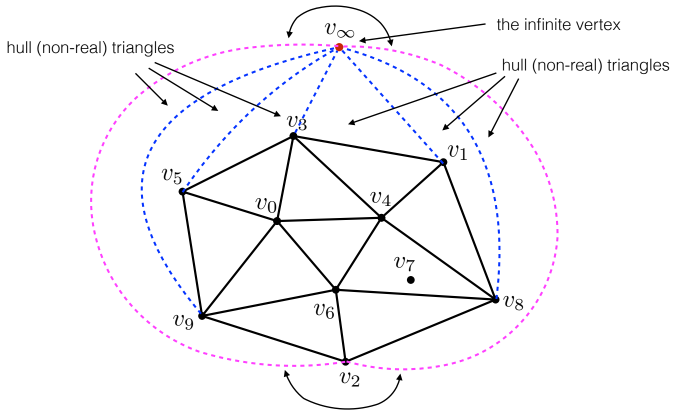


Figure 6.4: The infinite vertex and infinite triangles