CHAPTER 4

DATA STRUCTURES IMPLEMENTED

The efficient handling of geometric objects in triangulations requires — just like in any other field of computer science — the design of suitable data structures. From a computational point of view it is important to find a suitable representation of a triangulation and hence an adequate data structure.

For each vertex, three coordinates, and other connectivity information about the triangulation are required. Hence the data structure should be efficient in terms of storage. Rapid access to adjacent vertices, edges and triangles is another important aspect. Hence the data structure should preserve enough information to support the efficient implementation of these operations and maintain edge and vertex adjacency information for triangulations over the surface.

For each specific problem in geometry processing a characteristic set of operations can be identified by which the computation is dominated and capable of handling queries on attributes or most important on the object’s
position within the domain. One should also be able to access and maintain edge and vertex adjacency information for triangulations over the surface. Rapid access to adjacent vertices, edges and triangles is an important aspect. Hence we have to choose an appropriate data structure, which preserves enough information to support the efficient implementation of these operators. Unlike regular matrices, in which each node is connected to either four or eight adjacent neighbors, the number of neighbors connected to a given node in a Delaunay triangulation may in theory be arbitrarily large. Ideally, a data structure should represent this variable connectivity in a way that (1) provides rapid access to adjacent mesh elements without demanding excessive storage space and (2) is flexible enough to handle dynamic changes in the mesh itself. For dynamic modeling applications, an additional requirement is the need to maximize computational speed.

Novel data structures and algorithms for the construction, manipulation, and traversal of triangulations suitable for surface representation are presented in this chapter. Five data structures have been implemented and their performance studied. They are:

1) Quadedge
2) Triedge
3) Triplet
4) Biedge
5) TIN

4.1 POINT STRUCTURE

Each sample point has the X and Y coordinates (also known as "Easting" and "Northing" values respectively) indicating the planimetric location
and a surface or z-value indicating height / elevation. These points are to be connected by edges to form a set of non-overlapping triangles used to represent the surface. Hence, in all the data structures, a data point is represented by a Point2d data structure consisting of the x and y coordinates, a z-value and a fourth parameter, a pointer to any additional data that might be required to be stored and processed.

4.2 QUATEDGE STRUCTURE

The popular data structure for Delaunay Triangulation is quad-edge-based data structure, which was initially proposed by Leonidas Guibas and Jorge Stolfi (1985). It is popular because it's elegant and simultaneously represents a graph and its dual (such as Delaunay triangulation and Voronoi diagram).

![Diagram of quadedge structure](image)

**FIGURE 4.1** The quadedge data structure
Here, the original quad-edge structure has been augmented to hold the constrained edge information. Each quad-edge record groups together four directed edges corresponding to a single undirected edge in the subdivision and to its dual edge (Figure 4.1a). Each directed edge has two pointers: a next pointer to the next counterclockwise edge around its origin and a data pointer to geometrical and other nontopological information (such as the coordinates of its origin) and the constrained edge index. Figures 4.1b and 4.1c illustrate how three edges incident on the same vertex are represented using the quad-edge data structure: the vertex itself corresponds to the inner cycle of pointers in Figure 4.1c. The remaining three cycles correspond to the three faces meeting at the vertex.

FIGURE 4.2 “Make-Edge” and “Splice” operations
Construction and modification of the mesh are done by two basic operators:

- **Make-Edge** creates an individual edge with four connected “Quad” objects and every Quad has three pointers.
- **Splice** - splits a face into two pieces or merges two faces into one.

Figure 4.2 shows these two simple operations on the quadedge structure. The edge algebra operators suffice to maintain the surface triangulations.

The quadedge data structure provides all necessary information for a given edge. It is simple to implement and allows an edge-to-edge navigation through the mesh by means of its algebraic operations. The main disadvantage of the method is that storage costs are relatively high.

### 4.3 TRIEDGE STRUCTURE

The mesh consists of three structures: vertices, triangles and triedges.

A triangle $t$ is a pair of triples: the first triple consists of the vertices $v_0$, $v_1$ and $v_2$ and the second triple is a triedge containing edges $e_0$, $e_1$ and $e_2$.

![Triangle $t$ consisting of a set of vertices $v_i$, and a tri-edge consisting of edges $e_i$, $i = 0,1,2$. Edge $e = \{t, l\}$ is also shown.](image)

![Datastructure Consistency Diagram](image)

FIGURE 4.3 Triedge Data Structure
An edge $e$ is represented by a triple
\[ e = \{t, i, c\} \]
where $t$ is a triangle,
- $i$ is 0, 1 or 2 and
- $c$ is the constrained edge no.
The edge $e$ specifies the triangle $t$ to which it belongs and $i$ is the position of the edge within the triangle. Hence, the position index in this context is always taken modulo 3.

FIGURE 4.4 The connectivity of the mesh created by an edge $e$ from vertex $v_0$ to $v_1$

At the core of this structure is a recasting of the quad-edge data structure and accompanying edge algebra introduced by Guibas and Stolfi (1985) to the special case of a triangular mesh. Their notion of an edge reference and its associated operators remains largely intact, leading to a simple and precise formalism for specifying algorithms.

The following three basic operators manipulate the mesh:
- **MakeEdge** - constructs a mesh
- **Splice** – manipulates the connectivity of a mesh
- **Swap** – mesh connectivity operator
MakeEdge

MakeEdge takes a pair of vertices as input, say \( v_0 \) and \( v_1 \). It constructs a simple mesh consisting of two triangles (\( a \) and \( b \)) and returns an edge \( e \) such that

\[
\text{org}(e) = v_0 \quad \text{and} \quad \text{dest}(e) = v_1.
\]

Vertex \( v_\infty \) is a special boundary vertex. A triangle containing \( v_\infty \) is a boundary triangle. A triangle not containing \( v_\infty \) is an interior triangle.

![Diagram](image1)

FIGURE 4.5 The effect of MakeEdge on two vertices \( v_0 \) and \( v_1 \)

Splice

Splice takes a pair of edges as input and rearranges the edge links within the associated triangles. It manipulates the connectivity of a mesh.

![Diagram](image2)

FIGURE 4.6 The effect of Splice on two edges \( a \) and \( b \)
Swap

Swap sets edge $e$ to the opposite diagonal of a quadrilateral. Its effect is illustrated by Figure 4.7.

The following set of operators is defined for traversing edges and accessing elements of a mesh.

- $\text{rot}(e) = \{e_i, e_i + 1\}$,
- $\text{invrot}(e) = \text{rot}(\text{rot}(e))$,
- $\text{onext}(e) = \text{edge } e_i \text{ of triangle } e_t$,
- $\text{oprev}(e) = \text{rot}(\text{onext}(\text{rot}(e)))$,
- $\text{sym}(e) = \text{rot}(\text{onext}(e))$,
- $\text{org}(e) = \text{vertex } v_i \text{ of triangle } e_t$,
- $\text{dest}(e) = \text{org}(\text{invrot}(e))$,
- $\text{right}(e) = \text{org}(\text{rot}(e))$,
- $\text{left}(e) = \text{dest}(\text{onext}(e))$.
Figure 4.9 illustrates the construction of a simple triangular mesh with the three vertices $a$, $b$, $c$. Four triangles: $t_0$, $t_1$, $t_2$, $t_3$ are created. One of these i.e. $t_0$ is an interior triangle and the three remaining triangles i.e. $t_1$, $t_2$ and $t_3$ are boundary triangles.

![Diagram of construction of a simple triangular mesh](image)

**FIGURE 4.9** Illustration of construction of a simple triangular mesh

### 4.4 TRIPLET STRUCTURE

Peucker (1977), Lawson (1977), Gold (2000) suggested preserving for each triangle, pointers to the three data points forming the vertices as well as to the three neighbouring triangles.

In the triplet structure, the basic mesh data structure comprises of the following linked by pointers:

- vertices,
- triplets and
- cedges.
A triplet consists of a list of three vertices, a list of three adjoining triangles, a list of three cedges (when constrained edges exist). For a triangle on a boundary of the mesh, some or of the neighboring triangles may not be present. Such triangles are represented by a triplet with null vertices.

A cedge is a special data structure used to represent a constrained edge of the mesh. It consists of a list of two vertices, a list of adjoining constrained edges and a list of two adjoining triangles.

The pointers to adjoining vertices, triplets and cedges are ordered in the counter-clockwise direction, thus paving a way to indicate their geometric relation to each other. Each of these pointers also contain position information; this position index can take values 0, 1 or 2 for a triangle and 0 or 1 for a cedge. Each pointer to an adjoining triangle indicates which face of that triangle is adjacent. The position index in the triangle context is always taken modulo 3. Similarly, each pointer to an adjoining cedge indicates which side of that cedge is contacted, and how the cedge is oriented relative to the triangle.

FIGURE 4.10 A triangulation and its equivalent triplet data structure
The triplet abc represents a triangle abc i.e. a triangle whose origin is a, destination is b and apex is c. These vertices occur in counterclockwise order about the triangle. Therefore, the triplet abc simultaneously denotes vertex a, edge ab, and triangle abc.

The following set of operators is defined for a triplet structure to traverse and access the elements of a mesh:

\[
\begin{align*}
org(abc) &= a, \\
dest(abc) &= b, \\
apex(abc) &= c, \\
sym(abc) &= ba^* & (* \text{ - apex of the respective triangle}), \\
lnext(abc) &= bca & \text{(next counterclockwise edge of triangle abc)}, \\
lprev(abc) &= cab & \text{(previous counterclockwise edge of triangle abc)}. \\
\end{align*}
\]

The following set of operators is defined for a cedge structure:

\[
\begin{align*}
sym(ab) &= ba, \\
eorg(ab) &= a, \\
edest(ab) &= b. \\
\end{align*}
\]

4.5 BIEDGE STRUCTURE

This is similar to the quadedge structure except that the dual information is removed. An edge is therefore represented by just two edges. Hence, the constrained edge information is augmented as an integer field.
4.6 TIN STRUCTURE

The simplest representation for triangle meshes would just store a set of individual triangles with references to the side and point data. This is the implementation in the TIN structure.

The TIN structure comprises of the following data structures: TINPoints, Triangle and Sides.

**TINPoints** is a structure that denotes the x, y, z coordinates of a point, the number of edges emanating from this point, and a pointer to this list of the edges.

**Triangle** is a structure that points to the three edges of a triangle.

An edge may be a boundary edge or an interior edge. In the case of a boundary edge, it denotes the side of only one triangle; whereas for an interior edge, it denotes the sides of two triangles. This is the information represented by a Sides structure. Thus, **Sides** is a structure that represents the two endpoints, the two triangles whose side it is, whether it is a boundary edge and whether it has been fully connected to the mesh.

A maximum triangle side length parameter is set to prevent the formation of large inaccurate triangles.

The algorithm starts with the creation of an initial triangle, called a seeding triangle. This triangle is obtained by checking from the first point to the
last, whether it is connectable or not. Three points are connectable if no other triangle intersects this triangle.

The three edges of the triangle are made and added to the Sides list, the triangle is made and added to the Triangle list.

Then for each side, the point that can be connected to form a triangle fulfilling the Delaunay property i.e. “empty circumcircle” property is found and the corresponding edges and triangle are added, under the condition that the length of each side does not exceed a specified maximum length.

If a triangle cannot be formed under the condition that the length of each side does not exceed the specified maximum length, a new seeding triangle is created.

![FIGURE 4.11 Triangulation using TIN structure](image)

This process is repeated till all the vertices are exhausted or no more triangles can be formed.
Determining triangle and vertex adjacency is crucial to many algorithms that deal with surfaces, from refinement and smoothing to simplification and level of detail management. As processing speed increases and memory cost decreases, dynamic re-triangulation of surfaces to improve visual appearance and optimize the performance of rendering hardware is becoming a realistic goal. These systems will require simple, uniform and efficient access to and maintenance of triangle and vertex adjacency. The structures and algorithms presented here meet these goals and provide a means of specifying mesh based algorithms that is precise and free of ambiguities.

Despite its simplicity, it is immediately clear that the TIN structure is not sufficient for most requirements: connectivity information cannot be accessed explicitly.