Chapter 6

Extending the Arrangement

Developing applications that use arrangements to solve problems that are a bit more complicated than the problems presented in previous chapters requires the ability to adapt the arrangement data structure to the application needs. One technique to do this is to extend the arrangement with auxiliary, usually non-geometric, data. In this chapter we describe several ways to extend an arrangement data structure.

6.1 The Notification Mechanism

In some cases it is essential to know exactly what happens inside a specific \texttt{Arrangement\_2} object. For example, when a new curve is inserted into an arrangement, it may be necessary to keep track of the faces that are split due to this insertion operation. Other important examples are the point-location strategies that require auxiliary data structures (see Section 3.1.1), which must be kept up-to-date when the arrangement changes. The 2D Arrangements package offers a mechanism that uses observers [83, Chapter 5]. The objective behind this mechanism is to define a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically. The observed object does not know anything about the observers. It merely “publishes” information about changes when they occur. In our case observers can be attached to an arrangement object. An attached observer receives notifications about the changes this arrangement undergoes.

An observer object, the type of which is an instance of the \texttt{Arr\_observer<Arrangement>} class template, stores a pointer to an arrangement object. When the \texttt{Arr\_observer<Arrangement>} class template is instantiated, the \texttt{Arrangement} parameter must be substituted with the type of the arrangement object. The observer receives notifications \textit{just before} a structural change occurs in the arrangement and \textit{immediately after} such a change takes place. \texttt{Arr\_observer} serves as a base class for other observer classes and defines a set of virtual notification functions, with default empty implementations. The set of functions can be divided into three categories, as follows:

1. Notifiers on changes that affect the entire topological structure of the arrangement. This category consists of two pairs (\textit{before} and \textit{after}) that notify the observer of the following changes:
   - The arrangement is cleared.
   - The arrangement is assigned with the contents of another arrangement.

2. Pairs of notifiers before and after a local change that occurs in the topological structure. Most notifier functions belong to this category. The relevant local changes include:
   - A new vertex is constructed and associated with a point.
   - An edge\footnote{The term “edge” refers here to a pair of twin halfedges.} is constructed and associated with an $x$-monotone curve.

E. Fogel et al., \textit{CGAL Arrangements and Their Applications}, Geometry and Computing 7, DOI 10.1007/978-3-642-17283-0_6, © Springer-Verlag Berlin Heidelberg 2012
• An edge is split into two edges.
• An existing face is split into two faces as a consequence of the insertion of a new edge.
• A hole is created in the interior of a face.
• Two holes are merged to form a single hole as a consequence of the insertion of a new edge.
• A hole is moved from one face to another as a consequence of a face split.
• Two edges are merged into one edge.
• Two faces are merged into one face as a consequence of the removal of an edge that used to separate them.
• One hole is split into two as a consequence of the deletion of an edge that used to connect the two components.
• A vertex is removed.
• An edge is removed.
• A hole is deleted from the interior of a face.

3. Notifiers on a structural change caused by a free function; see Sections 3.4 and 3.5 for a discussion on the free functions. This category consists of a single pair of notifiers, namely, `before_global_change()` and `after_global_change()`. Neither of these functions is invoked by methods of the `Arrangement_2` class template. Instead, they are called by the free functions themselves. It is implied that no point-location queries (or any other queries for that matter) are issued between the call to `before_global_change()` and the call to `after_global_change()`.

See the Reference Manual for a detailed specification of the `Arr_observer` class template and the prototypes of all notification functions.

Each arrangement object stores a list of pointers to `Arr_observer` objects. This list may be empty, in which case the arrangement does not have to notify any external class on the structural changes it undergoes. If, however, there are observers associated with the arrangement object, then whenever one of the structural changes listed in the first two categories above is about to take place, the arrangement object performs a forward traversal on this list and invokes the appropriate function of each observer. After the change takes place the observer list is traversed backward (from tail to head), and the appropriate notification function is invoked for each observer.

Concrete arrangement-observer classes should inherit from `Arr_observer`. When an observer object is constructed, it is attached to a valid arrangement supplied to the observer constructor, or alternatively the observer can be attached to the arrangement at a later time. When this happens, the observer object inserts itself into the observer list of the associated arrangement and starts receiving notifications whenever this arrangement changes thereafter. Subsequently, the observer object unregisters itself by removing itself from this list just before it is destroyed. Most concrete observer-classes do not need to use the full set of notifications. Thus, the bodies of all notification methods defined in the base class `Arr_observer` are empty. A concrete observer that inherits from `Arr_observer` needs to override only the relevant notification methods. The remaining methods are invoked when corresponding changes occur, but they do nothing.

The trapezoidal map RIC and the landmark point-location strategies both use observers to keep their auxiliary data structures up-to-date. In addition, you can define your own observer classes, inheriting from the base observer-class and overriding the relevant notification functions, as required by your application.

**Example:** The example below shows how to define and use an observer class. The observer in the example responds to changes in the arrangement faces. It prints a message whenever a face is split into two due to the insertion of an edge and whenever two faces merge into one due to the removal of an edge. The layout of the arrangement is depicted in the figure on the next page;