the Delaunay condition. Those elements form the Delaunay *cavity.* The faces that bound the cavity are called the *horizon.* The mesh is modified by removing the elements in the cavity and connecting the new vertex to the horizon.

The cavity is connected, thus can be found by a local search on the current mesh. When a point p is inserted, the cavity is determined starting from any element that will get removed by the insertion of p. To achieve optimal runtime bounds we use the idea of Guibas et al.[43] and maintain an association of every point p not yet inserted into the mesh with the element t_p that contains p. The search for the cavity of p will start at t_p. Their algorithm keeps the history of the mesh and uses that history to locate the t_p for each p as it is inserted. In contrast we do not keep the mesh history but maintain the association of noninserted points p to containing elements t_p on the current mesh.

At each incremental step all points on cavity elements have to be reassociated with new elements using lineside tests in 2D and planeside tests in 3D, which accounts for the dominant cost of the algorithm. We have carefully implemented the *bulldozing* idea described in Ref. [40] and extended it to three dimensions.

Our implementation does not require extra memory for the lists of points since at any time a point is either a vertex in the mesh or in one such list. The memory that will be used to store the vertex in the mesh can first be used as a list node.

The algorithm maintains a work queue of elements whose interiors contain points. When no elements contain points (*i.e.*, all have been added to the mesh), the algorithm terminates.

In this scenario all points are known at the beginning. We generate labels for the input points using cuts along coordinate directions as described at the end of Section 5. The runtimes reported in the next section include this preprocessing step.

6.5. *Delaunay Refinement*

To test our implementation's performance for the case when new points are dynamically generated at runtime, we implemented a 2D Delaunay refinement code in the style of Ruppert.[44] We augment a Delaunay triangulation by adding circumcenters of badly shaped triangles while maintaining the Delaunay property. When the initial triangulation is built we walk through the mesh once and check the quality of each element, queuing the ones not satisfying a preset minimum angle bound. The same work queue used in the triangulation phase of the algorithm is used to store the list of triangles to be split.

Whenever a new point p is generated the algorithm assigns a new label by considering the *horizon* vertices H of the cavity created by p and calculating the value v that minimizes the sum of the log norms to H. It then finds the closest label to v that is not yet used.

In the pure triangulation code, all vertices are known at the beginning, so we can store the point coordinates and the first level vertex arrays densely. In the refinement code we can only fill these arrays up to about 85% before the open