Solving problems with CGAL: an example using the 2D Apollonius package

Menelaos I. Karavelas*,†

1 Introduction

The CGAL project [3] is an open source project that aims at providing *"easy access to efficient and reliable geometric algorithms in the form of a C++ library"*, as it is stated on the project's web site. The development of the library started in 1995 and was initially funded by two ESPRIT LTR European projects. As of November 2003 it has been an open source project, while as of its latest public release 4.0, CGAL is distributed under the LGPL/GPL v3+ licenses. CGAL currently consists of more that 500K lines of C++ code, and supports several development platforms.

One of the major goals of CGAL is to provide robust construction of geometric entities, while at the same time strive for efficiency and genericity. These goals have influenced the library's design since the beginning of its existence. Genericity has been achieved by employing *generic programming techniques* and by following the concept/model development paradigm. Algorithms in CGAL depend on *concepts*, and at least one *model* per concept is provided by the library. Components are interchangeable as long as they satisfy the proper concept requirements, a feature that makes CGAL a sound basis for experimentation, as well as for the development of prototype or mature codes for academic and industrial uses.

The development of CGAL is characterized by the clear separation between algorithms and data structures (i.e., the combinatorial parts of a geometric algorithm) and the geometric predicates and constructions (this is the numerical part of the geometric algorithms). Predicates and constructions are encapsulated in *kernels* or *traits classes*, and they guarantee robustness by means of the Exact Geometric Computation (EGC) paradigm [10] that the library follows. Efficiency on the other hand is achieved by employing filtering techniques (for example, cf. [2]), either at the arithmetic or the geometric level.

The library is divided in four major parts: the arithmetic and algebra layer, the geometry kernels, the various packages and the support library. The arithmetic and algebra layer offers the framework for utilizing different number types, for providing polynomials, and, in more general terms, for supporting the various kernels and packages that apply to (non-)linear geometric objects. There are currently three distinct linear kernel concepts in CGAL, and at least one model for each such concept: the 2D kernel, 3D kernel and the dD kernel[1]. The support library offers, among others, STL extensions of the library, interfaces between CGAL and BGL [1] and geometric objects' generators. The bulk of the library lies, however, in its packages: there are packages for computing: arrangements, convex hulls, triangulations, Voronoi diagrams, and meshes, for performing: geometric optimization, geometry processing, and spatial searching, as well as support for: kinetic data structures and operations on cell complexes and polyhedra. The interested reader or prospective user of CGAL may consult the User and Reference Manual of the library [9].

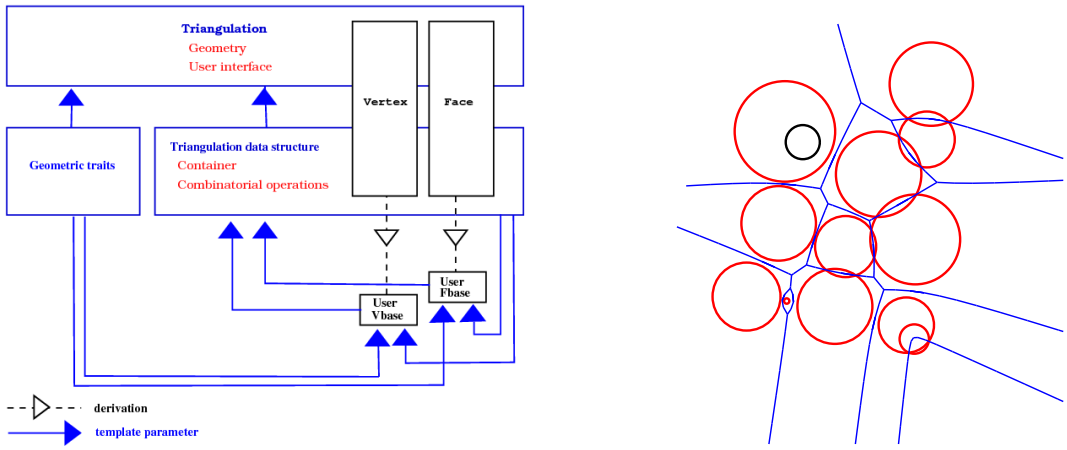
2 2D Triangulations and Apollonius graphs in CGAL

In this abstract we focus on 2D triangulations and 2D Delaunay graphs in CGAL, and, in particular, 2D Apollonius graphs. *Voronoi diagrams* in CGAL are computed implicitly via their dual compactified *Delaunay graphs*; there is support for point and segment Euclidean Voronoi diagrams [11, 5], power diagrams [11], as well as Apollonius diagrams (a.k.a., additively-weighted Voronoi diagrams) [6]. Their dual graphs are represented via the 2D Triangulation Data Structure (TDS) [8], which can actually handle any orientable triangulated surface without boundary. The data structure has containers for triangular faces and vertices, while edges are represented implicitly. Each face has pointers to its three neighbors and vertices, and each vertex points to one of its incident faces. The user has the ability to plug-in his/her own custom vertex and face class, which are then recovered by

*Department of Applied Mathematics, University of Crete, Greece, mkaravel@tem.uoc.gr
†Foundation for Research and Technology - Hellas, Greece

[1] In fact, the models of the 2D kernel are so far also models for the 3D kernel, so at the model level, the 2D and 3D kernels are indistinguishable.

This is an abstract related to a tutorial or demo given at GTS 2012. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear in a conference with formal proceedings and/or in a journal.



(a) (b)

Figure 1: (a) Schematic of the design of 2D Triangulations in CGAL. (b) An instance of the 2D Apollonius diagram.

the TDS via a *rebind* mechanism. The TDS is of purely combinatorial nature; geometry is added at a higher level, where also the traits classes computing the various types of Delaunay graphs are introduced; see also Fig. 1(a) for a schematic of the design of the CGAL 2D Triangulation package. From the user's perspective, vertices and faces are seen abstractly as *handles*. The user has the ability to *traverse* the triangulation in many different ways via iterators and circulators: there are iterators for iterating over the vertices, edges or faces of the triangulation, and circulators for going around the vertices, edges and faces incident to a specific vertex. Last, but not least, point location in the Voronoi diagram, or equivalently nearest neighbor queries, is offered by all Delaunay graph classes in CGAL, while all algorithms offer incremental construction of the Delaunay graph, and, in most cases, allow for dynamic site deletion.

In the Apollonius diagram the distance of a point p from a weighted point (or site) $S = (c, w)$ is defined as $d(p, S) = \|p - c\|_2 - w$. A site can be interpreted geometrically as a disk centered at c with radius w. The Apollonius diagram has several major differences with respect to its point counterpart: sites may have empty Voronoi cells, the 1-skeleton may be disconnected (although its compactified version is connected via the site at infinity), two sites may be connected by more than one Delaunay edge, and two faces (i.e., combinatorial triangles in the Delaunay graph) may have up to two edges in common. All these characteristics are apparent in the Apollonius diagram in Fig. 1(b). The 2D Apollonius graphs in CGAL follow the same design as 2D triangulations. The user can plug-in his/her own vertex and face, has access to vertices, edges and faces via circulators and iterators, can ask for the number of the connected components of the 1-skeleton of the Apollonius diagram, and can perform nearest neighbor location queries.

3 An example application

Suppose we are given a set $\mathcal{D}$ of n disks $D_1, D_2, \ldots, D_n$ on the plane, and we want to build a data structure that supports the following type of queries:

Given two disks D_i and D_j in $\mathcal{D}$, do they belong to the same connected component of the union $\cup_{k=1}^{n} D_k$?

Another way to pose the same problem is the following: Let $\mathcal{I}_\mathcal{D}$ be the intersection graph of $\mathcal{D}$. Given two disks in $\mathcal{D}$, do they belong to the same connected component of $\mathcal{I}_\mathcal{D}$? For simplicity, we will assume below that there is no disk in $\mathcal{D}$ that is contained fully inside another one: the general scenario where we allow disks to be contained in other disks can also be handled, in a slightly more complicated way.

Our practical solution to this problem will be to use the 2D Apollonius graph package of CGAL [6]. Under the assumption mentioned above that there is no disk in $\mathcal{D}$ that is contained inside another disk, the disks in $\mathcal{D}$ correspond to vertices of the 2D Apollonius graph $AG(\mathcal{D})$ of $\mathcal{D}$, and, more importantly, there exists a subgraph G of $AG(\mathcal{D})$ having the same connected components as the intersection graph of $\mathcal{D}$ (in fact, we will compute G to be a spanning forest of $\mathcal{I}_\mathcal{D}$). We will exploit this fact, and show how to build a mini-application on top of the 2D Apollonius graph, that also supports same-connected-component queries in $O(1)$ time. Our solution is static, in the sense that it assumes that all disks in $\mathcal{D}$ are known in advance.

The idea is to build a spanning forest $\mathcal{F}_\mathcal{D}$ of $\mathcal{I}_\mathcal{D}$ on top of $AG(\mathcal{D})$. We will do so as follows:

1. We will modify the Apollonius graph vertex base class by adding fields for supporting a tree-like structure. This will give us an in-place implementation of the spanning forest $\mathcal{F}_\mathcal{D}$ of $\mathcal{I}_\mathcal{D}$.
2. We will create a new traits class that enriches the 2D Apollonius graph traits class with the additional predicates that are required for the computation of $\mathcal{F}_\mathcal{D}$.
3. We will implement the high-level class, called `Disk_intersection_subgraph_2`, that:
   - will be responsible for computing $\mathcal{F}_\mathcal{D}$,
   - will provide access to the connected components of $\mathcal{F}_\mathcal{D}$ via iterators,
   - will support the same-connected-component queries.

3.1 The vertex class

In the new vertex base class for the Apollonius graph, we are going to add three new fields: two that will be used for representing the in-place forest, understood a collection of rooted trees, and one for storing a vertex of $\mathcal{F}_\mathcal{D}$ that will be unique for each tree in $\mathcal{F}_\mathcal{D}$, and thus will act a *representative* for each tree in $\mathcal{F}_\mathcal{D}$ (in fact this representative node will be the root of each tree in $\mathcal{F}_\mathcal{D}$). The first two fields will be: (1) a vertex to the parent node in the tree of $\mathcal{F}_\mathcal{D}$ that our vertex belongs to, and (2) the set of children of the current vertex in the tree it belongs to. The set of children will be implemented as a `std::set`, so as to be able to efficiently determine if a vertex is a child of another vertex in some rooted tree of $\mathcal{F}_\mathcal{D}$. The representative vertex, which will be the third added field, will eventually be used for answering the same-connected-component query of two vertices in $\mathcal{F}_\mathcal{D}$, by simply comparing their representative vertices.

For implementing the vertex class of the `Disk_intersection_subgraph_2` class, named `Disk_intersection_subgraph_vertex_base_2`, we will derive from the `Apollonius_graph_vertex_base_2` class. Since the `Disk_intersection_subgraph_2` requires as traits class a superset of the traits class for computing the 2D Apollonius graph, we will pass this superset traits class as a template parameter. Last, but not least, due to the circular dependency between vertices and faces of the triangulation data structure, will also need to implement the *rebind* mechanism in order for the triangulation data structure to know the correct type of the vertex base class used.

In the code fragment below we see most of the implementation of the `Disk_intersection_subgraph_vertex_base_2` class. The implementation of some member methods is obvious and is omitted.

```
template<class Gt, bool StoreHidden = false,
         class Vb = Apollonius_graph_vertex_base_2<Gt,StoreHidden> >
class Disk_intersection_subgraph_vertex_base_2 : public Vb {
private:
  typedef Vb Base;

public:
  // public types (required by the ApolloniusGraphVertexBase_2 concept)
  typedef typename Base::Geom_traits                          Geom_traits;
  typedef typename Base::Site_2                               Site_2;
  typedef typename Base::Apollonius_graph_data_structure_2    Apollonius_graph_data_structure_2;
  typedef typename Base::Face_handle                          Face_handle;
  typedef typename Base::Vertex_handle                        Vertex_handle;

  static const bool Store_hidden = StoreHidden;

  // the rebind mechanism
  template < typename AGDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<AGDS2>::Other   Vb2;
    typedef Disk_intersection_subgraph_vertex_base_2<Gt,Store_hidden,Vb2>  Other;
  };

private:
  // the comparator functor that will be used in the std::set; it uses the Compare_site_2 predicate
  // which is a new predicate (it is not provided by the model of the ApolloniusGraphTraits_2 concept)
  struct Vertex_less { ... };

  // type for the set of children nodes
  typedef std::set<Vertex_handle,Vertex_less> Children_set;
```

```
  // the representative vertex, the parent vertex and the set of children
  Vertex_handle rep_vertex, v_parent;
  Children_set  children;
public:
  // type for the iterator on the childred
  typedef typename Children_set::const_iterator Children_iterator;

  // constructors
  Disk_intersection_subgraph_vertex_base_2() : Base(), rep_vertex(), v_parent() {}
  Disk_intersection_subgraph_vertex_base_2(const Site_2& p) : Base(p), rep_vertex(), v_parent() {}
  Disk_intersection_subgraph_vertex_base_2(const Site_2& p, Face_handle f)
    : Base(p, f), rep_vertex(), v_parent() {}

  // set/get the representative vertex
  inline void          representative(Vertex_handle rep)       { rep_vertex = rep; }
  inline Vertex_handle representative()                  const { return rep_vertex; }

  // set/get the parent vertex
  inline void          parent(Vertex_handle vp)       { v_parent = vp; }
  inline Vertex_handle parent()                 const { return v_parent; }

  // add a new child
  inline void add_child(Vertex_handle n) { children.insert(n); }

  // test if v is a child of *this  vertex
  inline bool has_child(Vertex_handle v) const { return children.find(v) != children.end(); }

  // iterators for children
  inline Children_iterator children_begin() const { return children.begin(); }
  inline Children_iterator children_end()   const { return children.end(); }

  // the number of children
  inline typename Children_set::size_type number_of_children() const { return children.size(); }

  // clear the container of the child nodes
  inline void clear_children_container() { children.clear(); }
};
```

3.2 Augmenting the 2D Apollonius graph traits

For computing the forest $\mathcal{F}_{\mathcal{D}}$ we need two new predicates:

1. A functor that compares two disks and returns a `Comparison_result`. This comparison functor must produce a total ordering of the sites in $\mathcal{D}$. It is needed for constructing the set of children in each vertex of the forest (see previous subsection).
2. A functor that takes two disks and returns `true` if the two disks intersect and `false` otherwise.

One way to produce a total ordering for the disks is to first compare their centers lexicographically, and in case of equality, their radii; we are going to call the corresponding functor `Compare_site_2` (the code for this predicate is omitted). To implement the disk intersection predicate, we simply notice that, if $D_i = (c_i, r_i)$, $c_i = (x_i, y_i)$, $i = 1, 2$ are the two input disks of the predicate, they intersect if and only if the sign of the quality $(x_1 - x_2)^2 + (y_1 - y_2)^2 - (r_1 - r_2)^2$ is not positive. We are going to call this predicate `Do_intersect_2`, and its implementation may be found below. The template parameter `Gt` here stands for the `Disk_intersection_-subgraph_traits_2` class (defined below).

```
template<class Gt>
class Do_intersect_2 {
protected:
  typedef Gt                              Geom_traits;
  typedef typename Geom_traits::Site_2    Site_2;

  // functor, taken from the CGAL kernel, that computes the squared distance of two 2D points
  typedef typename Geom_traits::Kernel::Compute_squared_distance_2  D2;
```

```
public:
  // returns true if the (closures of the) disks s and t have non-empty intersection, false otherwise
  inline bool operator()(const Site_2& s, const Site_2& t) const {
    return CGAL::compare( CGAL::square(s.weight()+t.weight()), D2()(s.point(), t.point()) ) != SMALLER;
  }
};
```

Implementing the `Disk_intersection_subgraph_traits_2` class is now straightforward. We simply need to derive from the `Apollonius_graph_traits_2<K>` class, where `K` is a model of the CGAL 2D kernel, and add the two additional predicates.

```
template<class K>
class Disk_intersection_subgraph_traits_2 : public Apollonius_graph_traits_2<K> {
  typedef Disk_intersection_subgraph_traits_2<K>  Self;

  // some lines of code omitted...
public:
  // typedef for the template parameter that must be a model of the 2D CGAL kernel
  typedef K                                       Kernel;

  // types for the two new predicates
  typedef CGAL::Do_intersect_2<Self>              Do_intersect_2;
  typedef CGAL::Compare_site_2<Self>              Compare_site_2;
};
```

Implementing a traits class that supports arithmetic filtering is straightforward. However, this is a purely technical issue, that goes slightly beyond the scope of this abstract.

3.3 Implementing the high-level class

Having described how to modify the vertex base class, and how to augment the `Apollonius_graph_traits_2`, so as to provide the additional predicates needed for the computation of the disk intersection subgraph, we can now implement our high-level class that is responsible for the (static) computation of the disk intersection subgraph. At this point there are quite a few design choices that may be adopted. In the code extract below, we have chosen to derive, in a protected manner, from the `Apollonius_graph_2` class, instantiated with: (i) the `Triangulation_data_structure_2` class using our own vertex base, and (ii) our augmented traits class. We also show below the constructors provided for the `Disk_intersection_subgraph_2`, as well as the trivially implemented query for determining whether two disks belong to the same connected component of the union of the set of disks passed via the iterator range in the corresponding constructor.

The bulk of the work, however, is performed in the two `compute_intersection_subgraph` methods (cf. code extract below), which implement a DFS-like search on the computed Apollonius graph (the implementation is not shown here). In the first such method, we iterate over all vertices of the Apollonius graph; for each vertex v whose representative site has not been initialized, we initialize it (by setting its representative vertex to be the vertex itself), we insert it in an empty queue Q, and we start the DFS-like search by calling the second `compute_intersection_subgraph` method using as arguments the queue Q and the vertex v. In the second method, and while the queue is not empty, we pop the first element from the queue, say v, and look at its neighbors in the Apollonius graph. If a neighbor u of v has not been initialized and the disks corresponding to u and v intersect, we add u as a child of v in the same tree that v belongs to and we set the representative vertex of u to be the same as that of v. Finally, we push u at the end of the queue. Clearly, nothing has to be done if either u has already been assigned to a tree, or if u and v do not intersect.

```
template<class Gt>
class Disk_intersection_subgraph_2 : protected Apollonius_graph_2<Gt, Triangulation_data_structure_2<
                                                 Disk_intersection_subgraph_vertex_base_2<Gt,false>,
                                                 Triangulation_face_base_2<Gt> > >
{
  typedef Apollonius_graph_2<Gt, Triangulation_data_structure_2<
               Disk_intersection_subgraph_vertex_base_2<Gt,false>,
               Triangulation_face_base_2<Gt> > >                          Base;
public:
  // type definitions omitted ...
```

```
  typedef typename Base::Vertex_handle            Vertex_handle;
  typedef typename Base::Geom_traits              Geom_traits;

protected:
  typedef std::queue<Vertex_handle>  Queue;

  void compute_intersection_subgraph();
  void compute_intersection_subgraph(Queue& q, Vertex_handle v_rep);

public:
  // constructors
  Disk_intersection_subgraph_2(const Geom_traits& gt = Geom_traits()) : Base(gt) {}

  template<class Input_iterator>
  Disk_intersection_subgraph_2(Input_iterator first, Input_iterator beyond,
                               const Geom_traits& gt = Geom_traits()) : Base(first, beyond, gt)
    { compute_intersection_subgraph(); }

  inline bool in_same_connected_component(Vertex_handle v1, Vertex_handle v2) const
    { return v1->representative() == v2->representative(); }

  // code omitted ...
};
```

3.4 Going one step further

In our model implementation, we also provide methods for the number of connected components, as well as for the number of disks in each connected component. We have also implemented two types of iterators: one for iterating over the connected components of the intersection subgraph, and one for iterating over the vertices per connected components. In both cases we have used the `Filter_iterator` class provided in CGAL's STL extensions [4]. It should not be too complicated to adapt our approach above to a purely incremental one. Instead of building the spanning forest on top of the Apollonius graph, we should use the Union-Find data structure to incrementally compute the spanning subgraph. An implementation of Union-Find is already available within the Support Library of CGAL [7].

Acknowledgments. The work in this paper has been partially supported by the FP7-REGPOT-2009-1 project "Archimedes Center for Modeling, Analysis and Computation".

References

[1] The Boost Graph Library (BGL). http://www.boost.org/libs/graph/.

[2] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Appl. Math.*, 109:25–47, 2001.

[3] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[4] M. Hoffmann, L. Kettner, S. Pion, and R. Wein. Stl extensions for cgal. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:StlExtension.

[5] M. Karavelas. 2D segment Delaunay graphs. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:SegmentDelaunayGraph2.

[6] M. Karavelas and M. Yvinec. 2D Apollonius graphs (Delaunay graphs of disks). In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:ApolloniusGraph2.

[7] L. Kettner, S. Pion, and M. Seel. Profiling tools timers, hash map, union-find, modifiers. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:ProfilingTools.

[8] S. Pion and M. Yvinec. 2D triangulation data structure. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:TDS2.

[9] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html.

[10] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.

[11] M. Yvinec. 2D triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:Triangulation2.