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

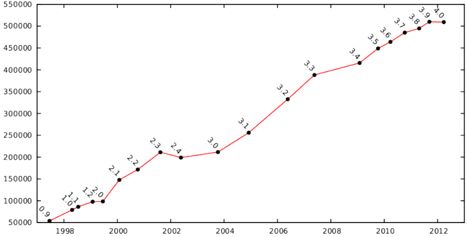
Menelaos I. Karavelas

http://www.cgal.org/

Geometric/Topological Software Minisymposium

CG-Week, Chapel Hill, NC, June 19th, 2012

The CGAL project



- Open source project
- Aims at providing *"easy access to efficient and reliable geometric algorithms in the form of a C++ library"*
- Development started in 1995 (two ESPRIT LTR European projects)
- Open source as of November 2003 (v3.0)
- LGPL/GPL v3+ as of March 2012 (v4.0)
- More than 500K lines of C++ code

The (current) world of CGAL in a glance

- 12 Institutes/Universities/Companies have participated in the development of CGAL
  - Europe, Israel, U.S.A.
  - 4 Institutes
  - 6 Universities
  - 2 Companies
- GeometryFactory (created in 2003): sells commercial licenses, provides support, develops customized solutions
- Open Source Project run by the *Editorial Board*
  - Currently 13 editors
  - Responsible for guiding the development of the library, developers, and the user community.

The project's structure

- Human resources categories
  - Editorial Board
  - Developers
  - Users
- Support for several platforms
  (g++ on Linux/MacOS/Windows, VC++ on Windows)
- About 20 active developers
- 3,500 pages manual
- 6-month release cycle

The project's structure (contd.)

- Contributors maintain their identity
- Editorial Board manages reviews of submissions
- Candidate packages are included in daily test suites
- svn is used as version control system
- Developer support:
  - manual for developers
  - dedicated mailing list
  - wiki
  - meetings (1-week long) once or twice per year

The design of the library

- Major goals
  1. Robust construction of geometric entities
  2. Efficiency
  3. Genericity

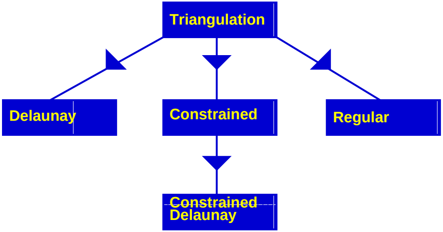
The design of the library

- Major goals
  1. Robust construction of geometric entities
  2. Efficiency
  3. Genericity
- Major design ideas:
  - Separation between algorithms/data structures and predicates
  - Predicates/Constructions are encapsulated in *kernels* and *traits classes*
  - Predicate evaluation: Exact Geometric Computation (EGC) Paradigm $\rightsquigarrow$ Robustness
  - Arithmetic/geometric filtering techniques (interval arithmetic) $\rightsquigarrow$ Efficiency
  - Generic programming via templates & concept/model development paradigm $\rightsquigarrow$ Genericity; at least one model per concept in the library

Parts of the library

1. Arithmetic & algebra layer: framework for utilizing number types, polynomials, support for kernels (esp. for non-linear objects)
2. Kernel concepts: 2D, 3D, dD kernels
3. Support library: STL extensions, interface with BGL, geometric generators
4. Packages (bulk of the library):
   - arrangements, convex hulls, triangulations, Voronoi diagrams, meshes
   - geometric optimization, geometry processing, spatial searching
   - support for Kinetic Data Structures, operations on cell complexes, operations on polyhedra

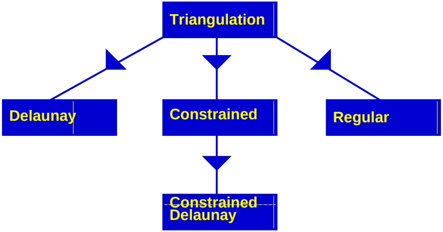
2D Triangulations overview



Support for 2D triangulations in CGAL:

- Basic triangulations
- Delaunay triangulations
- Regular triangulations
- Constrained triangulations
- Constrained Delaunay triangulations

2D Triangulations overview



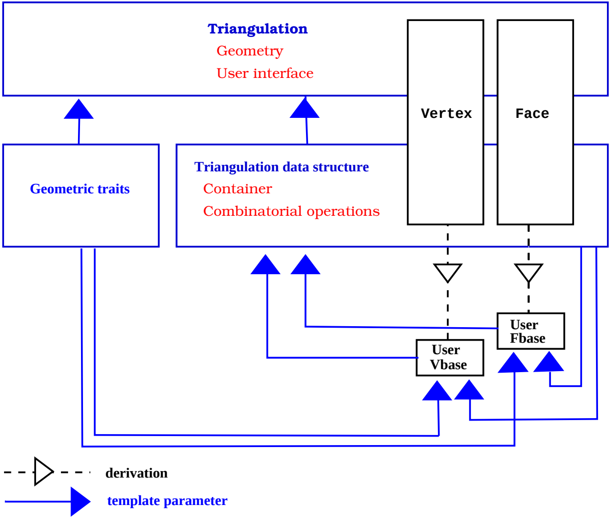
Support for 2D triangulations in CGAL:

- Basic triangulations
- Delaunay triangulations
- Regular triangulations
- Constrained triangulations
- Constrained Delaunay triangulations

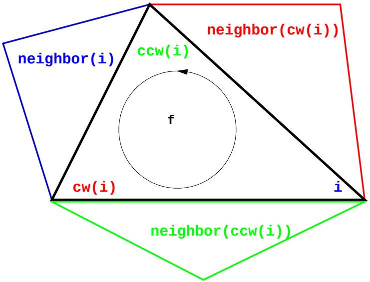
Built on top of 2D triangulations:

- Conforming triangulations & meshes
- Alpha shapes
- *Apollonius graphs*
- Segment Delaunay graphs

The software design of 2D triangulations



The 2D triangulation data structure



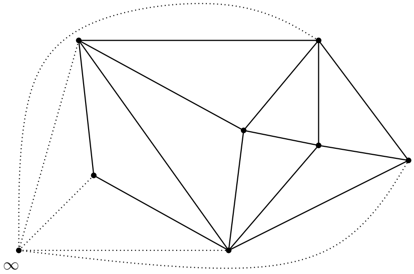
- Can represent any orientable triangulated surface
- Has containers for faces and vertices
- 3 pointers to defining vertices and 3 pointers to neighboring faces per face
- 1 pointer to incident face per vertex
- Faces and vertices are accessed via *handles*
- Edges are represented as pair of a face and an index

The rebind mechanism

- The user can plug-in own vertex and face classes
- The TDS recovers their types via the *rebind* mechanism:

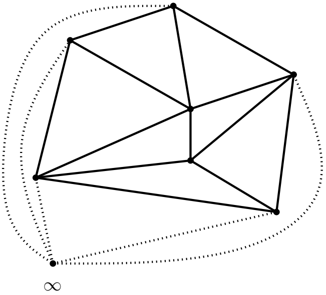
```
template<class Vb = Triangulation_ds_vertex_base_2<> >
class MyVertex : public Vb
{
  template <typename TDS2>
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other   Vb2;
    typedef MyVertex<Vb2>                                   Other;

  };
};


template < class Vb = Triangulation_ds_vertex_base_2<>,
           class Fb = Triangulation_ds_face_base_2<> >
class Triangulation_data_structure_2
{
  typedef Triangulation_data_structure_2<Vb,Fb>  Tds;

  typedef typename Vb::template Rebind_TDS<Tds>::Other  Vertex;
  typedef typename Fb::template Rebind_TDS<Tds>::Other  Face;
};
```

From the TDS to a triangulation



- TDS is of entirely combinatorial nature
- Geometry is added at a higher level
  - The geometric traits/kernel provides the geometrical information
  - A fictitious site is added at infinity

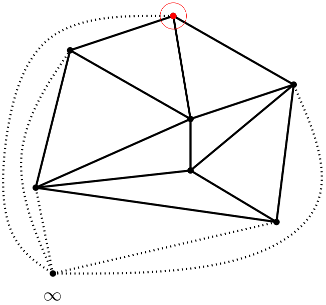
Access to features - All vertices iterator



- Iterator to all vertices

```
Tr::All_vertices_iterator it;
for (it = tr.all_vertices_begin();
     it != tr.all_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

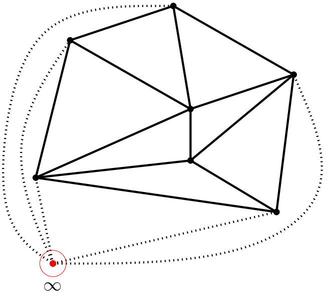
Access to features - All vertices iterator



- Iterator to all vertices

```
Tr::All_vertices_iterator it;
for (it = tr.all_vertices_begin();
     it != tr.all_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

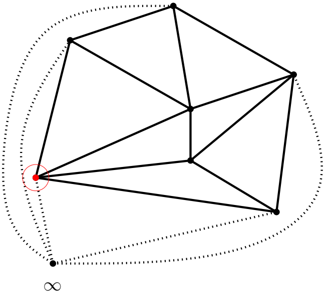
Access to features - All vertices iterator



- Iterator to all vertices

```
Tr::All_vertices_iterator it;
for (it = tr.all_vertices_begin();
     it != tr.all_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

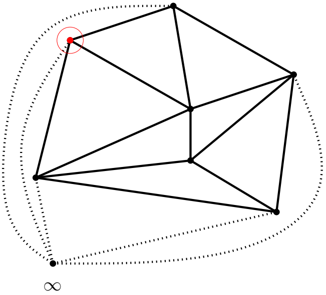
Access to features - All vertices iterator



- Iterator to all vertices

```
Tr::All_vertices_iterator it;
for (it = tr.all_vertices_begin();
     it != tr.all_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

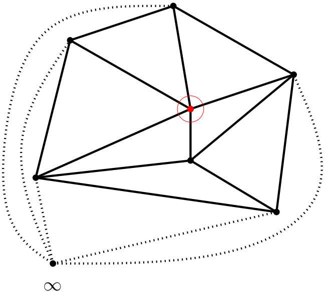
Access to features - All vertices iterator



- Iterator to all vertices

```
Tr::All_vertices_iterator it;
for (it = tr.all_vertices_begin();
     it != tr.all_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

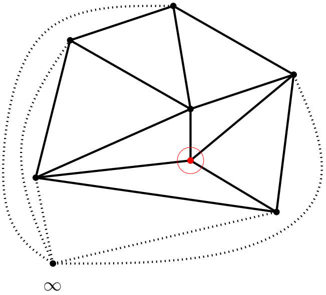
Access to features - All vertices iterator



- Iterator to all vertices

```
Tr::All_vertices_iterator it;
for (it = tr.all_vertices_begin();
     it != tr.all_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

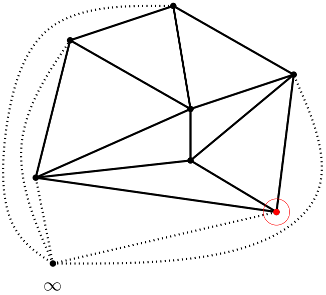
Access to features - All vertices iterator



- Iterator to all vertices

```
Tr::All_vertices_iterator it;
for (it = tr.all_vertices_begin();
     it != tr.all_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

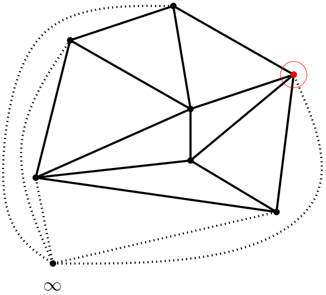
Access to features - All vertices iterator



- Iterator to all vertices

```
Tr::All_vertices_iterator it;
for (it = tr.all_vertices_begin();
     it != tr.all_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

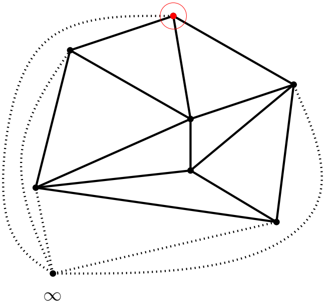
Access to features - All vertices iterator



- Iterator to all vertices

```
Tr::All_vertices_iterator it;
for (it = tr.all_vertices_begin();
     it != tr.all_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

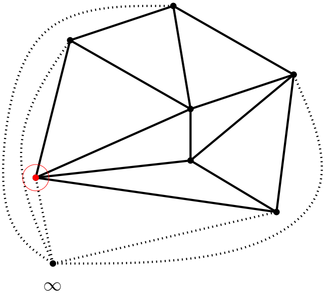
Access to features - Finite vertices iterator



- Iterator to finite vertices

```
Tr::Finite_vertices_iterator it;
for (it = tr.finite_vertices_begin();
     it != tr.finite_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

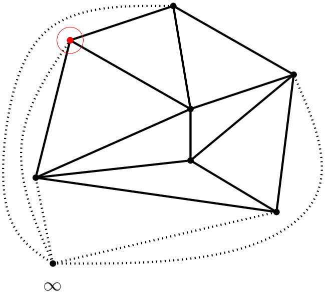
Access to features - Finite vertices iterator



- Iterator to finite vertices

```
Tr::Finite_vertices_iterator it;
for (it = tr.finite_vertices_begin();
     it != tr.finite_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

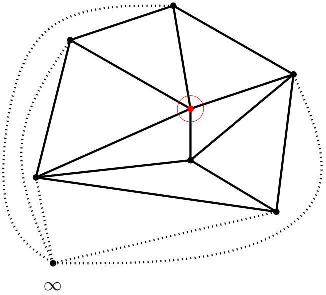
Access to features - Finite vertices iterator



- Iterator to finite vertices

```
Tr::Finite_vertices_iterator it;
for (it = tr.finite_vertices_begin();
     it != tr.finite_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

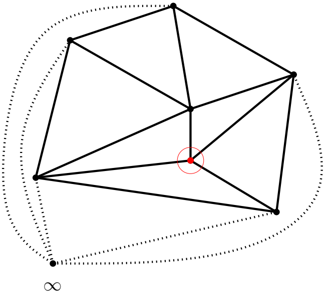
Access to features - Finite vertices iterator



- Iterator to finite vertices

```
Tr::Finite_vertices_iterator it;
for (it = tr.finite_vertices_begin();
     it != tr.finite_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

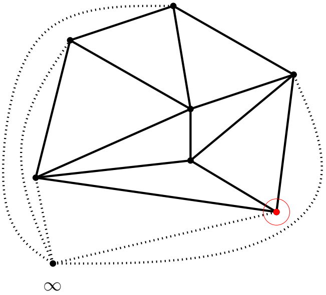
Access to features - Finite vertices iterator



- Iterator to finite vertices

```
Tr::Finite_vertices_iterator it;
for (it = tr.finite_vertices_begin();
     it != tr.finite_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

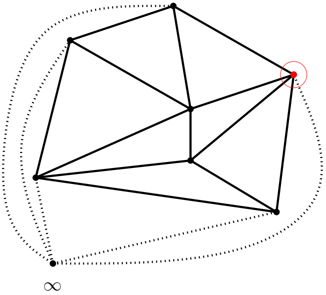
Access to features - Finite vertices iterator



- Iterator to finite vertices

```
Tr::Finite_vertices_iterator it;
for (it = tr.finite_vertices_begin();
     it != tr.finite_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

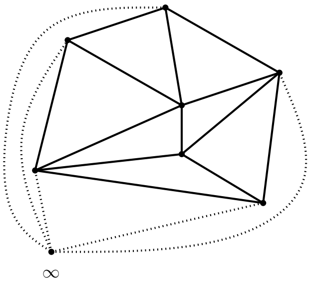
Access to features - Finite vertices iterator



- Iterator to finite vertices

```
Tr::Finite_vertices_iterator it;
for (it = tr.finite_vertices_begin();
     it != tr.finite_vertices_end(); ++it)
{
  Tr::Vertex_handle v(it);
  //...do what needs to be done with v
}
```

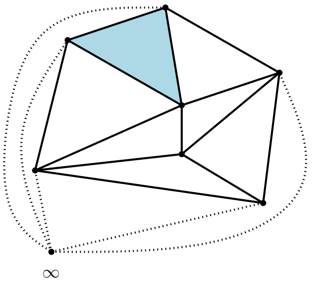
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

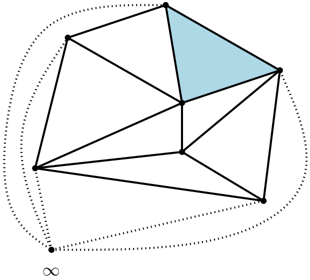
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

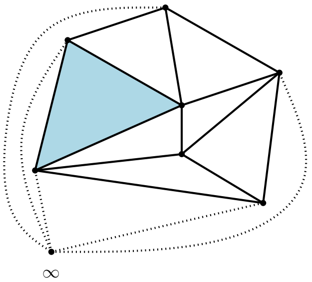
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

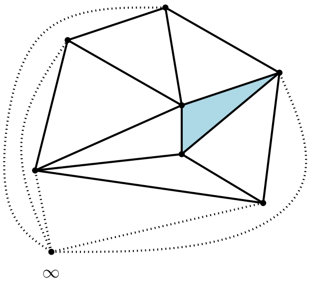
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

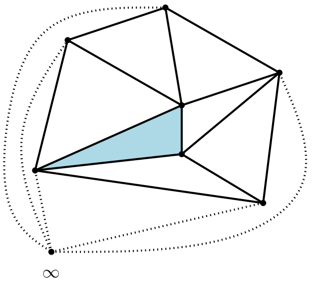
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

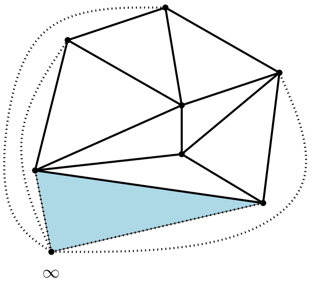
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

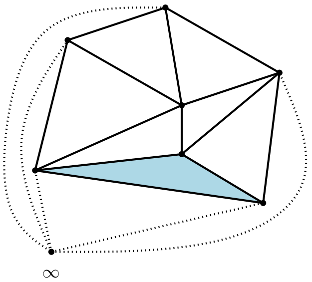
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

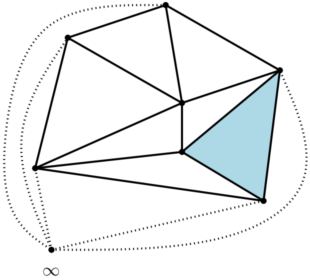
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

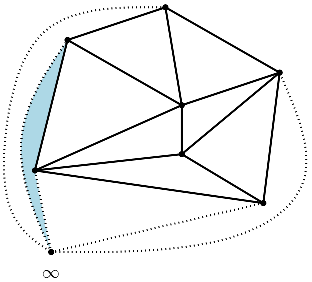
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

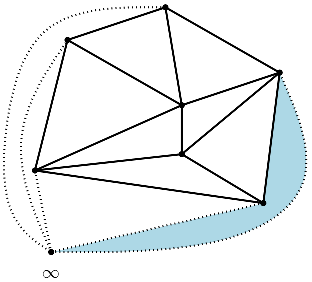
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

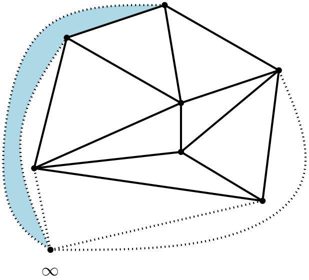
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

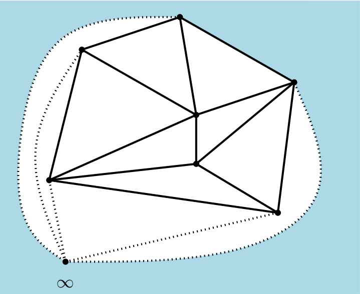
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

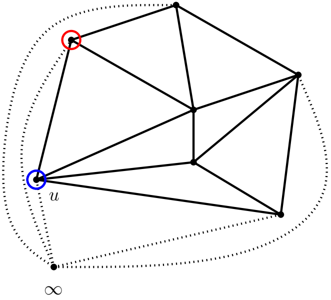
Access to features - All faces iterator



- Iterator to all faces

```
Tr::All_faces_iterator it;
for (it = tr.all_faces_begin();
     it != tr.all_faces_end(); ++it)
{
  Tr::Face_handle f(it);
  //...do what needs to be done with f
}
```

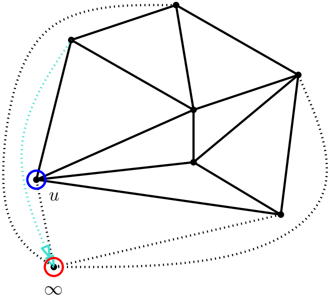
Access to features - Vertex circulator



- Circulator for vertices neighboring a vertex

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

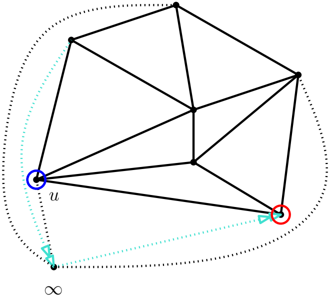
Access to features - Vertex circulator



- Circulator for vertices neighboring a vertex

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

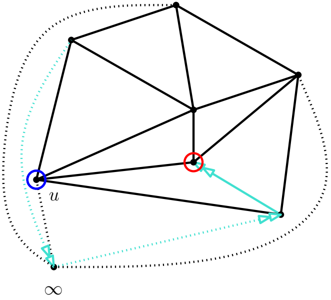
Access to features - Vertex circulator



- Circulator for vertices neighboring a vertex

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

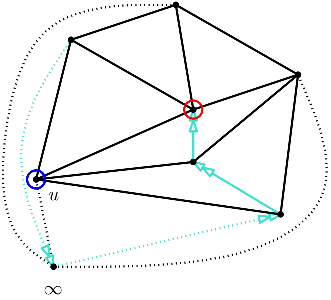
Access to features - Vertex circulator



- Circulator for vertices neighboring a vertex

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

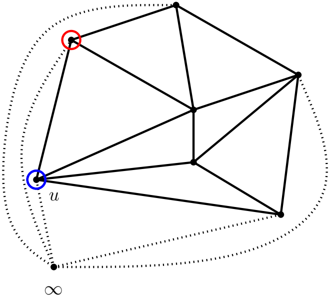
Access to features - Vertex circulator



- Circulator for vertices neighboring a vertex

```
Tr::Vertex_circulator vc_start =
                tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

Access to features - Vertex circulator



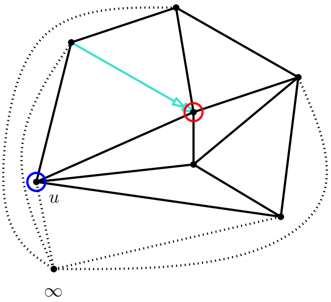
- Circulator for vertices neighboring a vertex

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

- Can also circulate clockwise:

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  --vc;
} while (vc != vc_start);
```

Access to features - Vertex circulator



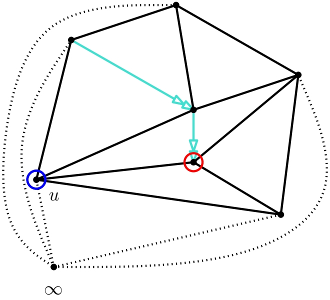
- Circulator for vertices neighboring a vertex

```
Tr::Vertex_circulator vc_start =
                  tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

- Can also circulate clockwise:

```
Tr::Vertex_circulator vc_start =
                  tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  --vc;
} while (vc != vc_start);
```

Access to features - Vertex circulator



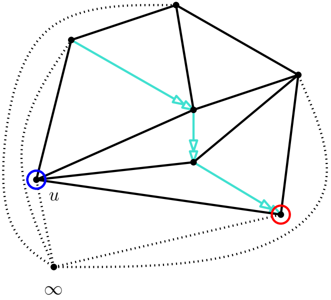
- Circulator for vertices neighboring a vertex

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

- Can also circulate clockwise:

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  --vc;
} while (vc != vc_start);
```

Access to features - Vertex circulator



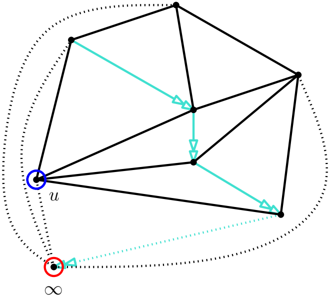
- Circulator for vertices neighboring a vertex

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

- Can also circulate clockwise:

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  --vc;
} while (vc != vc_start);
```

Access to features - Vertex circulator



- Circulator for vertices neighboring a vertex

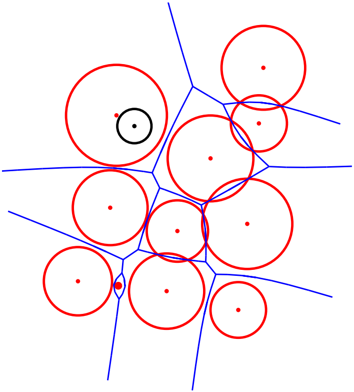
```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  ++vc;
} while (vc != vc_start);
```

- Can also circulate clockwise:

```
Tr::Vertex_circulator vc_start =
              tr.incident_vertices(u);
Tr::Vertex_circulator vc = vc_start;
do {
  Tr::Vertex_handle v(vc);
  //...do what needs to be done with v
  --vc;
} while (vc != vc_start);
```

The 2D Apollonius diagram

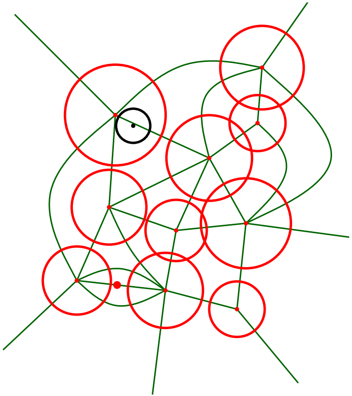
(aka additively-weighted Voronoi diagram)



- Input: set of n weighted sites $S_i = (c_i, r_i)$ (circles with center c_i and radius r_i)
- Distance: $\delta(x, S_i) = \|x - c_i\|_2 - r_i$
- Output: Voronoi diagram (defined the usual way)
- Three sites can have up to two Voronoi circles
- Bisectors are [branches of] hyperbolas
- A site can have *empty* Voronoi region; such a site is called *hidden*
- The 1-skeleton may have multiple connected components (that are connected at infinity)

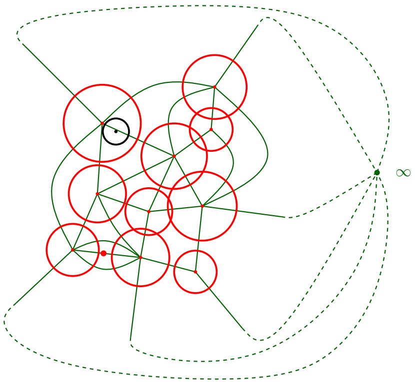
The 2D Apollonius diagram

(aka additively-weighted Voronoi diagram)



- Input: set of n weighted sites $S_i = (c_i, r_i)$ (circles with center c_i and radius r_i)
- Distance: $\delta(x, S_i) = \|x - c_i\|_2 - r_i$
- Output: Voronoi diagram (defined the usual way)
- Three sites can have up to two Voronoi circles
- Bisectors are [branches of] hyperbolas
- A site can have *empty* Voronoi region; such a site is called *hidden*
- The 1-skeleton may have multiple connected components (that are connected at infinity)

The Apollonius_graph_2 package



- The algorithm is dynamic
- Dual of the Voronoi diagram (a.k.a. Apollonius graph) is computed and stored; actually the compactified version
- The Apollonius graph (up to degeneracies) is planar and has triangular faces
- Two triangles can have two edges in common
- Two sites can be connected with multiple edges
- A site can appear multiple times on the convex hull

The dynamic algorithm

Insertion: to insert the new site $S = (c, r)$

- We perform point-location of c in the existing Voronoi diagram
- We determine whether S is hidden or not
- If S is not hidden, find the portion of the Voronoi diagram to be destroyed (conflict region)
- Destroy the conflict region and create the Voronoi region of S.

The dynamic algorithm

Insertion: to insert the new site $S = (c, r)$

- We perform point-location of c in the existing Voronoi diagram
- We determine whether S is hidden or not
- If S is not hidden, find the portion of the Voronoi diagram to be destroyed (conflict region)
- Destroy the conflict region and create the Voronoi region of S.

Deletion: to delete an existing site $S = (c, r)$

- Construct the "small" Voronoi diagram of the neighbors of S
- Destroy the star of S in the "big" Voronoi diagram
- Use the "small" diagram to fill-in the hole just created
- Finally, insert in the new diagram the sites than were hidden by S

The functionality of the package

- Basically the same with triangulations (+ some differences):
  - ✔ Provides iterators for all/finite vertices/edges/faces
  - ✔ Provides circulators for neighboring vertices
    - neighboring vertices may be reported multiple times
  - ✔ Provides circulators for edges/faces incident to a vertex
  - ✔ Provides access to hidden/visible sites (via iterators)
  - ✔ Supports nearest-neighbor queries for points (these are point-location queries in the Apollonius diagram)

The functionality of the package

- Basically the same with triangulations (+ some differences):
  - ✔ Provides iterators for all/finite vertices/edges/faces
  - ✔ Provides circulators for neighboring vertices
    - neighboring vertices may be reported multiple times
  - ✔ Provides circulators for edges/faces incident to a vertex
  - ✔ Provides access to hidden/visible sites (via iterators)
  - ✔ Supports nearest-neighbor queries for points (these are point-location queries in the Apollonius diagram)
  - ✗ Does not support point-location queries on the Apollonius graph
    - this is possible in basic, Delaunay and regular triangulations
  - ✗ Degeneracies are handled via an implicit perturbation scheme that depends on order of insertion
    - ✔ but we are working on a canonical perturbation scheme

The functionality of the package

- Basically the same with triangulations (+ some differences):
  - ✔ Provides iterators for all/finite vertices/edges/faces
  - ✔ Provides circulators for neighboring vertices
    - neighboring vertices may be reported multiple times
  - ✔ Provides circulators for edges/faces incident to a vertex
  - ✔ Provides access to hidden/visible sites (via iterators)
  - ✔ Supports nearest-neighbor queries for points (these are point-location queries in the Apollonius diagram)
  - ✗ Does not support point-location queries on the Apollonius graph
    - this is possible in basic, Delaunay and regular triangulations
  - ✗ Degeneracies are handled via an implicit perturbation scheme that depends on order of insertion
    - ✔ but we are working on a canonical perturbation scheme
  - ✔ In the incremental-only scenario, it is possible to save storage by not keeping track of the hidden sites
    - done at the level of the vertex base class

The design of the package

Follows the same design with triangulations (+ some differences again):

- `Apollonius_graph_2` class is templated by the traits and the data structure, which much be models of corresponding concepts

The design of the package

Follows the same design with triangulations (+ some differences again):

- `Apollonius_graph_2` class is templated by the traits and the data structure, which much be models of corresponding concepts
- The data structure concept is the same as for triangulations
    - however, we need to use a vertex base that is different from that for triangulations

The design of the package

Follows the same design with triangulations (+ some differences again):

- `Apollonius_graph_2` class is templated by the traits and the data structure, which much be models of corresponding concepts
- The data structure concept is the same as for triangulations
    - however, we need to use a vertex base that is different from that for triangulations
- The traits concept lists requirements for predicates and constructions
    - unlike the case of triangulations, the CGAL 2D kernels are not models: more predicates and constructions are needed

The design of the package

Follows the same design with triangulations (+ some differences again):

- `Apollonius_graph_2` class is templated by the traits and the data structure, which much be models of corresponding concepts
- The data structure concept is the same as for triangulations
  - however, we need to use a vertex base that is different from that for triangulations
- The traits concept lists requirements for predicates and constructions
  - unlike the case of triangulations, the CGAL 2D kernels are not models: more predicates and constructions are needed
- There is a hierarchical version of the `Apollonius_graph_2` class (analogous to the Delaunay hierarchy), which can speed up the computation of the diagram for large enough data sets.

The vertex base class – Part 1

```
template <class Gt, bool StoreHidden = true, class Vb = Triangulation_ds_vertex_base_2<> >
class Apollonius_graph_vertex_base_2
  : public Vb
{
private:
  typedef typename Vb::Triangulation_data_structure   AGDS;

public:
  // TYPES
  //------
  typedef Gt                              Geom_traits;
  typedef Vb                              Base;
  typedef typename Gt::Site_2             Site_2;
  typedef AGDS                            Apollonius_graph_data_structure_2;
  typedef typename AGDS::Face_handle      Face_handle;
  typedef typename AGDS::Vertex_handle    Vertex_handle;

  enum {Store_hidden = StoreHidden};

  template < typename AGDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<AGDS2>::Other      Vb2;
    typedef Apollonius_graph_vertex_base_2<Gt,StoreHidden,Vb2>  Other;
  };

private:
  // local types
  typedef std::list<Site_2>         Container;
```

The vertex base class – Part 2

```
public:
  // TYPES (continued)
  //------------------
  typedef typename Container::iterator     Hidden_sites_iterator;

public:
  // CREATION
  //---------
  Apollonius_graph_vertex_base_2() : Vb() {}
  Apollonius_graph_vertex_base_2(const Site_2& p) : Vb(), _p(p) {}
  Apollonius_graph_vertex_base_2(const Site_2& p, Face_handle f) : Vb(f), _p(p) {}

  ~Apollonius_graph_vertex_base_2() { clear_hidden_sites_container(); }

  // ACCESS METHODS
  //---------------
  const Site_2& site() const { return _p; }
  Site_2& site() { return _p; }

  Face_handle face() const { return Vb::face(); }

  std::size_t number_of_hidden_sites() const { return hidden_site_list.size(); }

  Hidden_sites_iterator hidden_sites_begin() { return hidden_site_list.begin(); }

  Hidden_sites_iterator hidden_sites_end() { return hidden_site_list.end(); }
```

The vertex base class – Part 3

```
public:
  // SETTING AND UNSETTING
  //-----------------------
  void set_site(const Site_2& p) { _p = p; }

  void add_hidden_site(const Site_2& p)
  {
    if ( StoreHidden ) {
      hidden_site_list.push_back(p);
    }
  }

  void clear_hidden_sites_container()
  {
    hidden_site_list.clear();
  }

public:
  // VALIDITY CHECK
  bool is_valid(bool verbose = false, int level = 0) const {
    return Vb::is_valid(verbose, level);
  }

private:
  // class variables
  Container hidden_site_list;
  Site_2 _p;
};
```

Our *"toy"* problem

Suppose we are given a set $\mathcal{D}$ of n disks $D_1, \ldots, D_n$, we want to build a data structure that supports (efficiently) the following query:

Query

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

Our *"toy"* problem

Suppose we are given a set $\mathcal{D}$ of n disks $D_1, \ldots, D_n$, we want to build a data structure that supports (efficiently) the following query:

Let $\mathcal{I}_{\mathcal{D}}$ be the intersection graph of $\mathcal{D}$.

Query

Given two disks D_i and D_j in $\mathcal{D}$, do they belong to the same connected component of $\mathcal{I}_{\mathcal{D}}$?

Our *"toy"* problem

Suppose we are given a set $\mathcal{D}$ of n disks $D_1, \ldots, D_n$, we want to build a data structure that supports (efficiently) the following query:

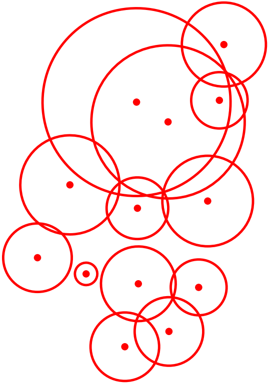
Let $\mathcal{I}_{\mathcal{D}}$ be the intersection graph of $\mathcal{D}$.

Query

Given two disks D_i and D_j in $\mathcal{D}$, do they belong to the same connected component of $\mathcal{I}_{\mathcal{D}}$?

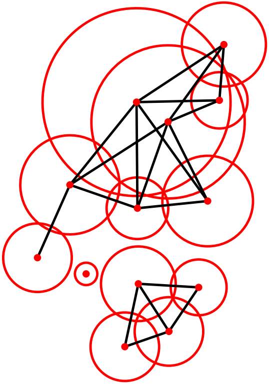
- The solution that will be presented today is based on the `Apollonius_graph_2` CGAL package.
- We will assume that there are no hidden sites
- We will describe a static solution (*i.e.*, all sites are known in advance)
- The query time will be $O(1)$.

Our *"toy"* solution



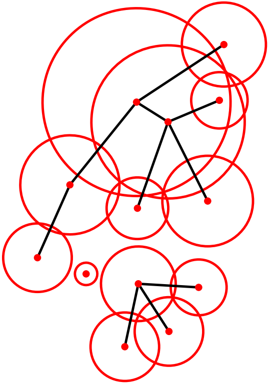
- Let $AG(\mathcal{D})$ denote the Apollonius graph of $\mathcal{D}$.
- There exists a subgraph G of $AG(\mathcal{D})$ having the same connected components as $\mathcal{I}_{\mathcal{D}}$.
  - in fact, we will compute G to be a *spanning forest* $\mathcal{F}_{\mathcal{D}}$ of G.

Our *"toy"* solution



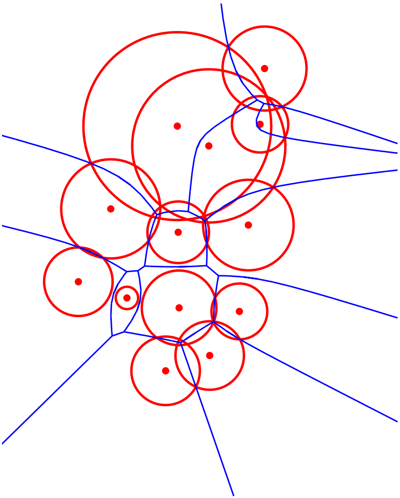
- Let $AG(\mathcal{D})$ denote the Apollonius graph of $\mathcal{D}$.
- There exists a subgraph G of $AG(\mathcal{D})$ having the same connected components as $\mathcal{I}_{\mathcal{D}}$.
  - in fact, we will compute G to be a *spanning forest* $\mathcal{F}_{\mathcal{D}}$ of G.

Our *"toy"* solution



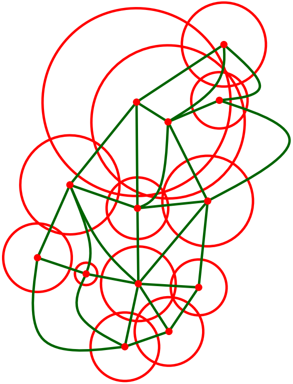
- Let $AG(\mathcal{D})$ denote the Apollonius graph of $\mathcal{D}$.
- There exists a subgraph G of $AG(\mathcal{D})$ having the same connected components as $\mathcal{I}_{\mathcal{D}}$.
  - in fact, we will compute G to be a *spanning forest* $\mathcal{F}_{\mathcal{D}}$ of G.

Our *"toy"* solution



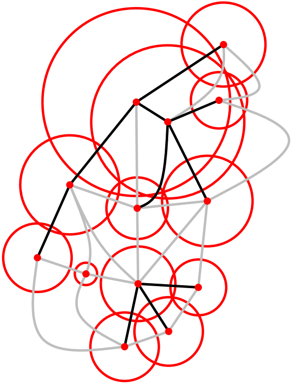
- Let $AG(\mathcal{D})$ denote the Apollonius graph of $\mathcal{D}$.
- There exists a subgraph G of $AG(\mathcal{D})$ having the same connected components as $\mathcal{I}_{\mathcal{D}}$.
  - in fact, we will compute G to be a *spanning forest* $\mathcal{F}_{\mathcal{D}}$ of G.

Our *"toy"* solution



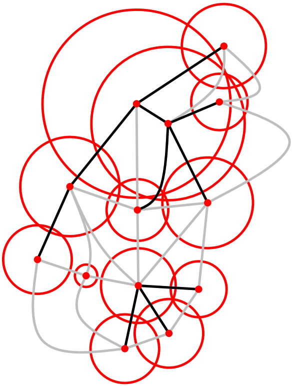
- Let $AG(\mathcal{D})$ denote the Apollonius graph of $\mathcal{D}$.
- There exists a subgraph G of $AG(\mathcal{D})$ having the same connected components as $\mathcal{I}_{\mathcal{D}}$.
  - in fact, we will compute G to be a *spanning forest* $\mathcal{F}_{\mathcal{D}}$ of G.

Our *"toy"* solution



- Let $AG(\mathcal{D})$ denote the Apollonius graph of $\mathcal{D}$.
- There exists a subgraph G of $AG(\mathcal{D})$ having the same connected components as $\mathcal{I}_{\mathcal{D}}$.
    - in fact, we will compute G to be a *spanning forest* $\mathcal{F}_{\mathcal{D}}$ of G.

Our *"toy"* solution



- Let $AG(\mathcal{D})$ denote the Apollonius graph of $\mathcal{D}$.
- There exists a subgraph G of $AG(\mathcal{D})$ having the same connected components as $\mathcal{I}_{\mathcal{D}}$.
  - in fact, we will compute G to be a *spanning forest* $\mathcal{F}_{\mathcal{D}}$ of G.
- We will compute $\mathcal{F}_{\mathcal{D}}$ by performing a DFS-like search on $AG(\mathcal{D})$:
  - for each non-visited disk v, we will find, among v's neighbors in $AG(\mathcal{D})$, all disks with which v intersects; call this set $\mathcal{I}_v$
  - we will mark v as visited
  - we will proceed recursively with all disks in $\mathcal{I}_v$

Implementing our solution

- We will implement the forest $\mathcal{F}_{\mathcal{D}}$ in-place. To do this we will:

Implementing our solution

- We will implement the forest $\mathcal{F}_{\mathcal{D}}$ in-place. To do this we will:
  1. Modify the vertex base class of $AG(\mathcal{D})$ by adding fields for storing
     1. the in-place forest (as a set of rooted trees)
     2. the root of the tree that the vertex belongs to (rep. vertex)

Implementing our solution

We will implement the forest $\mathcal{F}_{\mathcal{D}}$ in-place. To do this we will:

1. Modify the vertex base class of $AG(\mathcal{D})$ by adding fields for storing
   1. the in-place forest (as a set of rooted trees)
   2. the root of the tree that the vertex belongs to (rep. vertex)
2. Create a new traits class with the additional predicates needed for computing $\mathcal{F}_{\mathcal{D}}$

Implementing our solution

We will implement the forest $\mathcal{F}_{\mathcal{D}}$ in-place. To do this we will:

1. Modify the vertex base class of $AG(\mathcal{D})$ by adding fields for storing
   1. the in-place forest (as a set of rooted trees)
   2. the root of the tree that the vertex belongs to (rep. vertex)
2. Create a new traits class with the additional predicates needed for computing $\mathcal{F}_{\mathcal{D}}$
3. Implement the `Disk_intersection_subgraph_2` class that will
   1. compute $\mathcal{F}_{\mathcal{D}}$
   2. support the same-connected-component queries
   2. provide access to the connected components of $\mathcal{F}_{\mathcal{D}}$ via iterators

The new vertex base class

- Must be a model of the `ApolloniusGraphVertexBase_2` concept
- Additional fields:
  - `rep_vertex` (the representative vertex)
  - `parent` (the parent vertex in the tree)
  - `children` (the children in the tree)
- The children will be implemented as `std::set<Vertex_handle,Vertex_less>`
  - `Vertex_less` is the comparator functor used in the `std::set`

The Disk_intersection_subgraph_vertex_base_2 class – Part 1

```
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;
  };
```

The Disk_intersection_subgraph_vertex_base_2 class – Part 2

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

    bool operator()(const Vertex_handle& v1,
    const Vertex_handle& v2) const
    {
      return Compare_site_2()(v1->site(), v2->site()) == SMALLER;
    }
  };

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

  // the representative vertex
  Vertex_handle rep_vertex;
  // the parent vertex
  Vertex_handle v_parent;
  // the children
  Children_set  children;

public:
  // type for the iterator on the children
  typedef typename Children_set::const_iterator Children_iterator;
```

The Disk_intersection_subgraph_vertex_base_2 class – Part 3

```
public:
  // 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(); }
};
```

The additional predicates

Two additional predicates required:

1. A functor that compares two disks (returns a `Comparison_result`); must produce total order of $\mathcal{D}$
   - this predicate is somehow optional since it depends on our choice of data structure for the `Children_set` in the vertex base class
2. A functor that returns `true` if two disks intersect and `false` otherwise
   - given two disks $D_i = ((x_i, y_i), r_i)$, $i = 1, 2$, this predicate amounts to computing the sign of quantity:

$$(x_1 - x_2)^2 - (y_1 - y_2)^2 - (r_1 - r_2)^2$$

The disk comparator functor

Really simple, and based on existing predicates

- Gt stands for the disk intersection subgraph traits class

```
template<class Gt>
class Compare_site_2
{
public:
  typedef typename Gt::Comparison_result  Comparison_result;
  typedef typename Gt::Site_2             Site_2;

protected:
  typedef typename Gt::Compare_x_2         Compare_x_2;
  typedef typename Gt::Compare_y_2         Compare_y_2;
  typedef typename Gt::Compare_weight_2    Compare_weight_2;

public:
  typedef Site_2             argument_type;
  typedef Comparison_result  result_type;

  Comparison_result  operator()(const Site_2& p, const Site_2& q) const
  {
    Comparison_result cr_w = Compare_weight_2()(p, q);
    if ( cr_w != EQUAL ) { return cr_w; }

    Comparison_result cr_x = Compare_x_2()(p, q);
    if ( cr_x != EQUAL ) { return cr_x; }

    return Compare_y_2()(p, q);
  }
};
```

The disk intersection predicate

Again simple; will use as much kernel functionality as possible

- again Gt stands for the disk intersection subgraph traits class

```
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  Distance_2;

public:
  typedef bool     result_type;
  typedef Site_2   argument_type;

  // 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()),
                          Distance_2()(s.point(), t.point())
                         ) != SMALLER;
  }
};
```

Putting the traits together

K is a model of the CGAL 2D kernel concept

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

protected:
  typedef Apollonius_graph_traits_2<K>            Base;

public:
  typedef K                                       Kernel;

  typedef typename Kernel::Comparison_result      Comparison_result;
  typedef typename Base::Site_2                   Site_2;

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

  // access to the two new predicates
  inline Compare_site_2
  compare_site_2_object() const { return Compare_site_2(); }

  inline Do_intersect_2
  do_intersect_2_object() const { return Do_intersect_2(); }
};
```

Implementing the Disk_intersection_subgraph_2 class

- Will derive from the Apollonius_graph_2 class in a protected manner
- Instantiate the TDS with our own vertex base class
- Use our augmented traits

```
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:
  typedef typename Base::Finite_vertices_iterator Vertex_iterator;
  typedef typename Base::Vertex_circulator        Vertex_circulator;
  typedef typename Base::Vertex_handle            Vertex_handle;
  typedef typename Base::Geom_traits              Geom_traits;
  typedef typename Base::size_type                size_type;
  typedef typename Base::Site_2                   Site_2;
  typedef typename Base::Point_2                  Point_2;

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

The main part of the class implementation

```
protected:
  void compute_intersection_subgraph();                                   // to be implemented
  void compute_intersection_subgraph(Queue& q, Vertex_handle v_rep);    // to be implemented

  size_type n_components; // the number of connected components
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();
  }

  bool is_valid(bool verbose = false, int level = 1) const
  {
    for (Vertex_iterator vit = vertices_begin(); vit != vertices_end(); ++vit) {
      if ( vit->representative() == Vertex_handle() ) { return false; }
      for (Children_iterator it = vit->children_begin(); it != vit->children_end(); ++it) {
        if ( (*it)->parent() != Vertex_handle(vit) ) { return false; }
        if ( !vit->has_child(*it) ) { return false; }
      }
    }
    return Base::is_valid(verbose, level);
  }
```

The various iterators

```
typedef typename Base::Triangulation_data_structure::Vertex::Children_iterator Children_iterator;

inline Vertex_iterator vertices_begin() const { return Base::finite_vertices_begin(); }
inline Vertex_iterator vertices_end()   const { return Base::finite_vertices_end(); }

typedef  Connected_comp_vertex_iterator<Vertex_iterator,Vertex_handle>
Connected_component_vertex_iterator;

typedef  Connected_comp_iterator<Vertex_iterator,Vertex_handle>
Connected_component_iterator;

typedef Connected_component_iterator Connected_component_handle;

inline Connected_component_iterator connected_components_begin() const {
  return Connected_component_iterator(vertices_end(), vertices_begin());
}

inline Connected_component_iterator connected_components_end() const {
  return Connected_component_iterator(vertices_end());
}

inline Connected_component_vertex_iterator vertices_begin(Connected_component_handle ch) const {
  return Connected_component_vertex_iterator(vertices_end(), ch->representative(), vertices_begin());
}

inline Connected_component_vertex_iterator vertices_end(Connected_component_handle ch) const
{
  return Connected_component_vertex_iterator(vertices_end(), ch->representative());
}
```

Counting vertices and connected components

```
  inline size_type number_of_connected_components() const { return n_components; }

  inline size_type number_of_connected_component_vertices(Connected_component_handle ch) const
  {
    size_type nv = number_of_vertices();
    if ( nv < 2 ) { return nv; }

    Queue q;
    q.push(ch->representative());

    size_type n(0);
    while ( !q.empty() ) {
      Vertex_handle v = q.front();
      q.pop();

      ++n;
      for (Children_iterator it = v->children_begin(); it != v->children_end(); ++it) {
        q.push(*it);
      }
    }

    return n;
  }

  inline size_type number_of_vertices() const { return Base::number_of_vertices(); }
};
```

Time to do the "dirty" job

- Files from the web site if you have not downloaded them yet
- CGAL is already setup in the VirtualBox image
- Can compile the files right away (`demo` and `examples` directories)
- What to do:
  - Open the file `Disk_intersection_subgraph_2.h` (`include/CGAL` directory) and fill-in the code for the two `compute_intersection_subgraph()` methods.
- Will be walking around to help

Going one step further

- ✗ The traits class presented assumes an exact predicates/exact constructions CGAL kernel (due to the computations in the `Do_intersect_2` predicate)
- ✔ A traits class that supports arithmetic filtering should also be implemented
  - easy and straightforward to do; it is a purely technical issue

Going one step further

- ✗ The traits class presented assumes an exact predicates/exact constructions CGAL kernel (due to the computations in the `Do_intersect_2` predicate)
- ✔ A traits class that supports arithmetic filtering should also be implemented
  - easy and straightforward to do; it is a purely technical issue
- ✔ The implementation could easily be made incremental: use the Union-Find data structure to compute the spanning forest
  - there is an implementation of Union-Find in the Support Library of CGAL

Going one step further

- ✗ The traits class presented assumes an exact predicates/exact constructions CGAL kernel (due to the computations in the `Do_intersect_2` predicate)
- ✔ A traits class that supports arithmetic filtering should also be implemented
  - easy and straightforward to do; it is a purely technical issue
- ✔ The implementation could easily be made incremental: use the Union-Find data structure to compute the spanning forest
  - there is an implementation of Union-Find in the Support Library of CGAL

This is it for today. Thank you