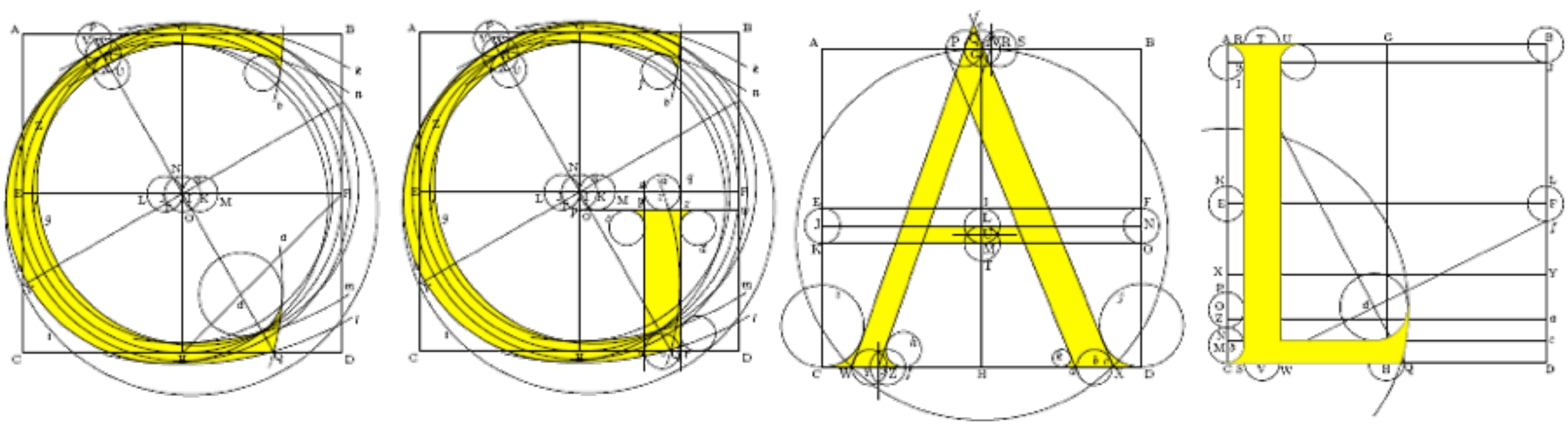
PROXIMITY STRUCTURES IN



The Computational Geometry Algorithms Library

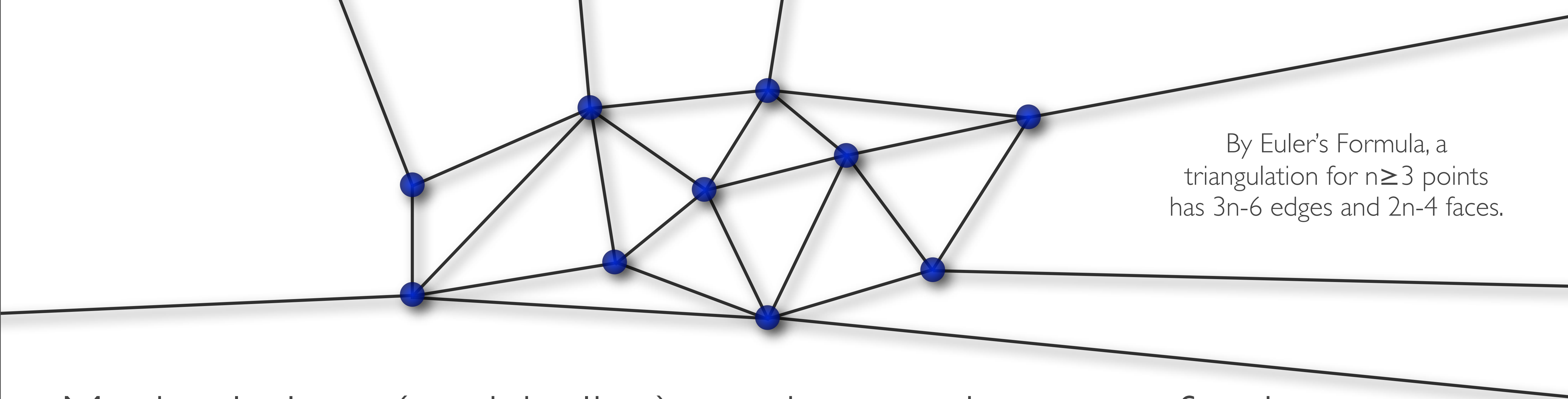
Michael Hoffmann <hoffmann@inf.ethz.ch>

(Based on work by Pierre Alliez, Andreas Fabri, Efi Fogel, Lutz Kettner, Sylvain Pion, Monique Teillaud, Mariette Yvinec, and probably many others.)

PART V:

Proximity Structures

TRIANGULATIONS

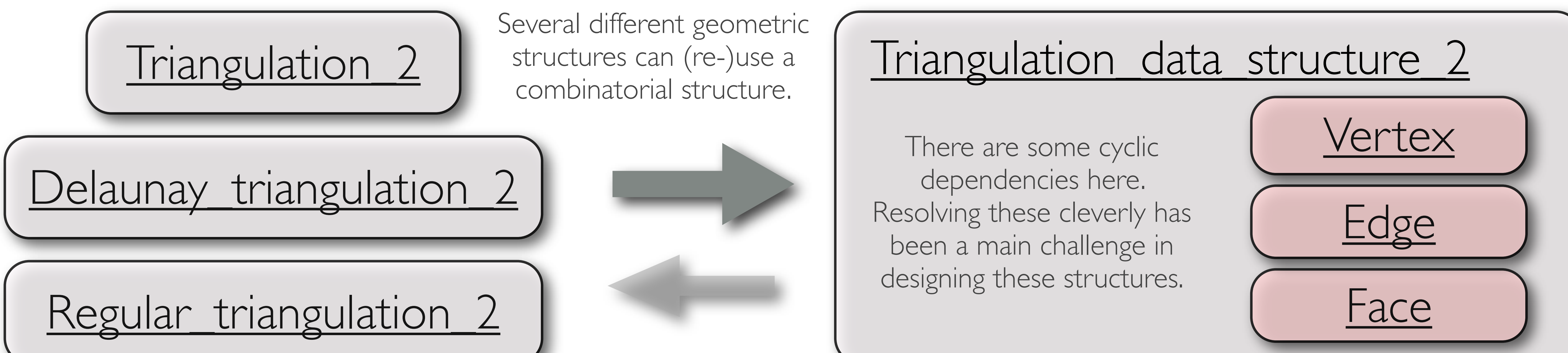


By Euler's Formula, a triangulation for $n \geq 3$ points has $3n-6$ edges and $2n-4$ faces.

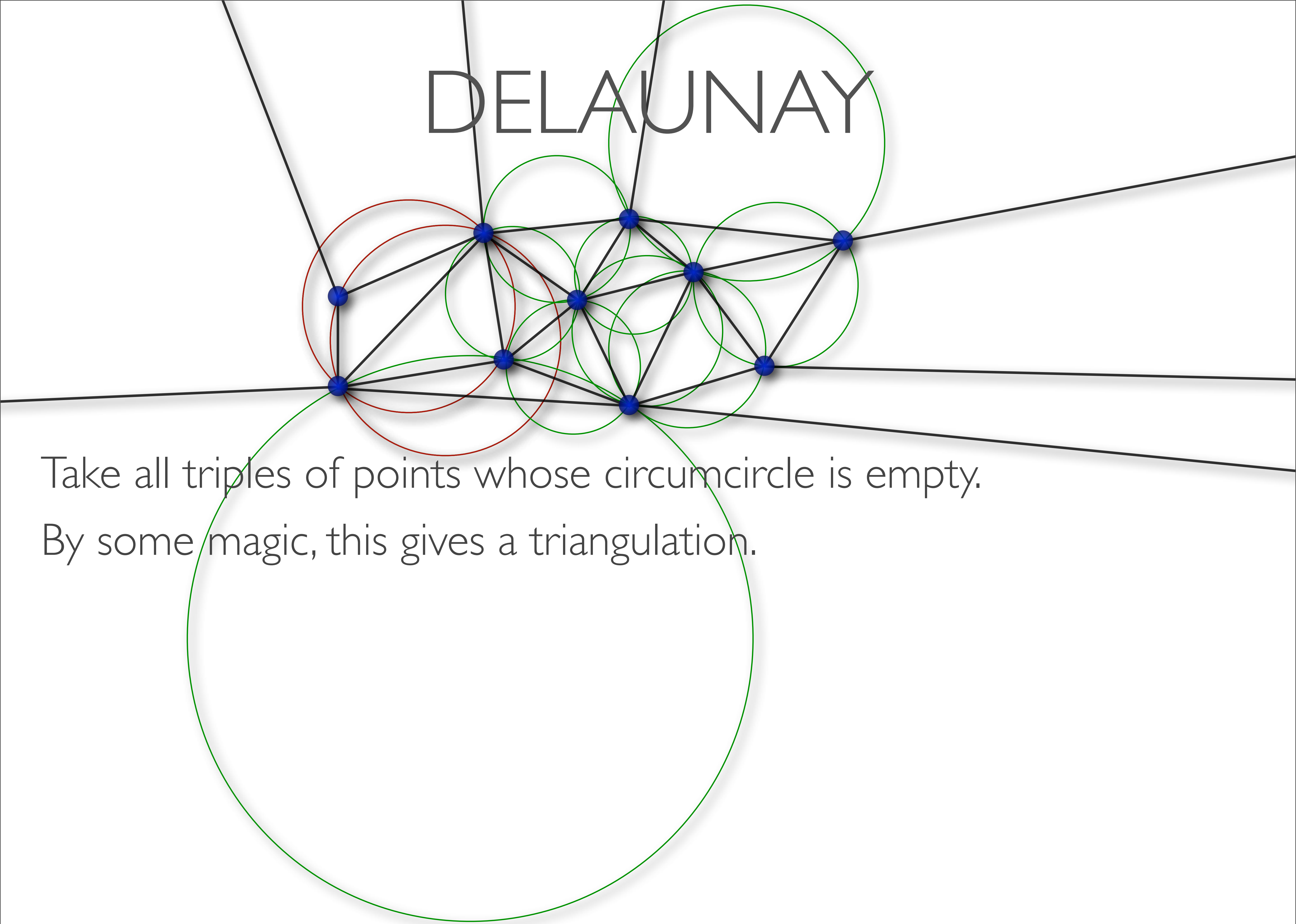
Maximal plane (straight line) graph on a given set of points.

An "infinite vertex" triangulates the exterior of the convex hull.

The combinatorial graph structure is separated from the geometry.



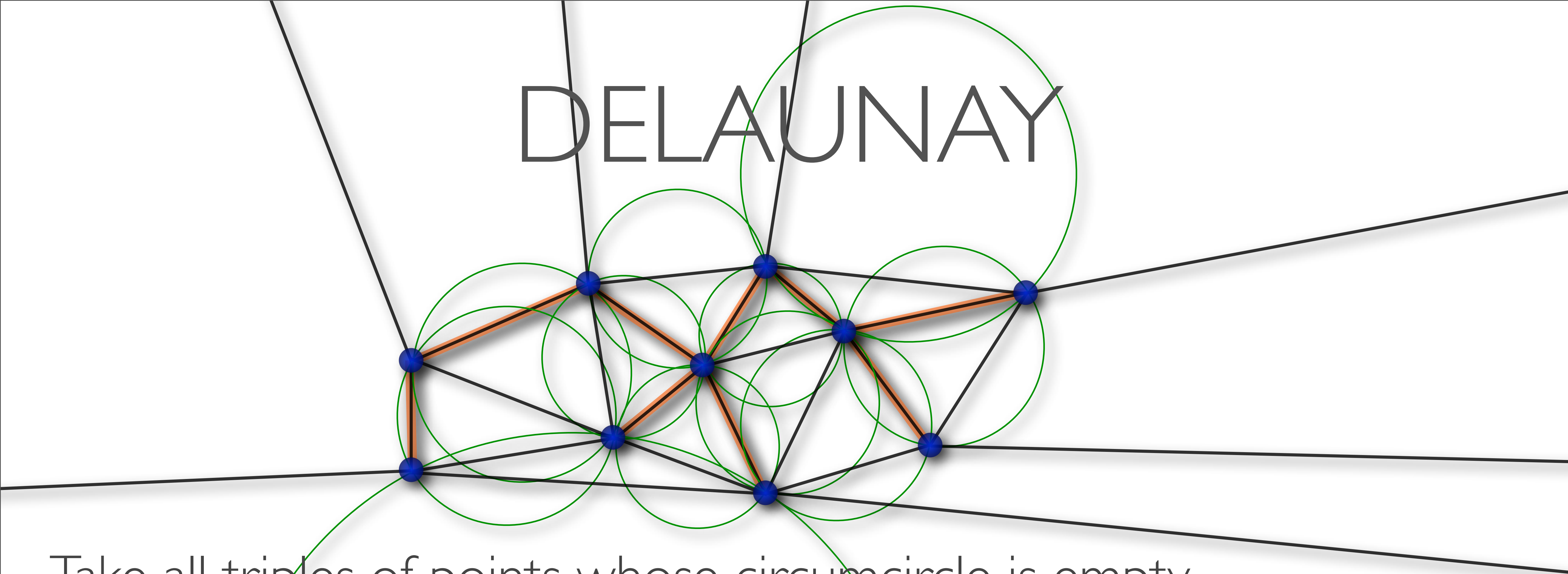
DELAUNAY



Take all triples of points whose circumcircle is empty.

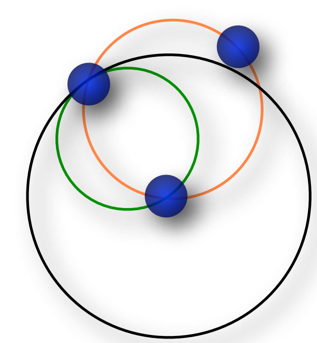
By some magic, this gives a triangulation.

DELAUNAY



Take all triples of points whose circumcircle is empty.

By some magic, this gives a triangulation.



This Delaunay Triangulation has several nice properties:

- It maximizes the smallest angle. Among all triangulations of these points.
- It contains the Euclidean minimum spanning tree and the nearest neighbor graph. Each point has an edge to all closest other points.
- It is unique for points in general position. No four points cocircular.
- It can be constructed efficiently. O(n log n) in 2D, O(n^2) in 3D.

DELAUNAY TRIANGULATION

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Finite_faces_iterator  Face_iterator;

int main()
{
  // read number of points
  std::size_t n;
  std::cin >> n;
  // construct triangulation
  Triangulation t;
  for (std::size_t i = 0; i < n; ++i) {
       Triangulation::Point p;
       std::cin >> p;
       t.insert(p);
  }
  // output all triangles
  for (Face_iterator f = t.finite_faces_begin(); f != t.finite_faces_end(); ++f)
       std::cout << t.triangle(f) << "\n";
}
```

No exact constructions needed, output points == input points.

We do not want to output the infinite faces outside the convex hull. Otw, use `All_faces_iterator`...

To get edges instead, replace `Face` by `Edge` and `faces` by `edges` everywhere, and use `t.segment(...)` instead of `t.triangle(...)`.

Not `*f`! The triangulation interface is based on so-called *handles*. These are an abstraction of pointers. Think of them as something that can be dereferenced to yield (in this case) a `Triangulation::Face`. In particular, iterators (like f here) convert to the corresponding handles.

The corresponding type is called `Triangulation::Face_handle`.

DELAUNAY TRIANGULATION

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Finite_faces_iterator  Face_iterator;

int main()
{
  // read number of points
  std::size_t n;
  std::cin >> n;
  // construct triangulation
  Triangulation t;
  for (std::size_t i = 0; i < n; ++i) {
       Triangulation::Point p;
       std::cin >> p;
       t.insert(p);
  }
  // output all triangles
  for (Face_iterator f = t.finite_faces_begin(); f != t.finite_faces_end(); ++f)
       std::cout << t.triangle(f) << "\n";
}
```

This works, but inserting the points one by one is dangerous in terms of efficiency, as the performance of the triangulation depends on the insertion order. A (uniformly) random order yields an expected runtime of $O(n \log n)$, but there are point sets that have bad orders for which the runtime becomes quadratic…

https://elabs.inf.ethz.ch/file.php/29/CGALWeek2/Sample_Programs/delaunay-basic.cpp

DELAUNAY TRIANGULATION

```
...

int main()
{
  ...

  // read points
  std::vector<K::Point_2> pts;
  pts.reserve(n);
  for (std::size_t i = 0; i < n; ++i) {
    K::Point_2 p;
    std::cin >> p;
    pts.push_back(p);
  }
  // construct triangulation
  Triangulation t;
  t.insert(pts.begin(), pts.end());

  ...
}
```

A safe strategy is to let the triangulation choose a suitable insertion order: Instead of inserting points one by one using `t.insert(p)`, insert a whole (iterator) range [b,e) of points using `t.insert(b,e)`.

Here the input points are first read into a vector and then inserted as a whole into the triangulation.

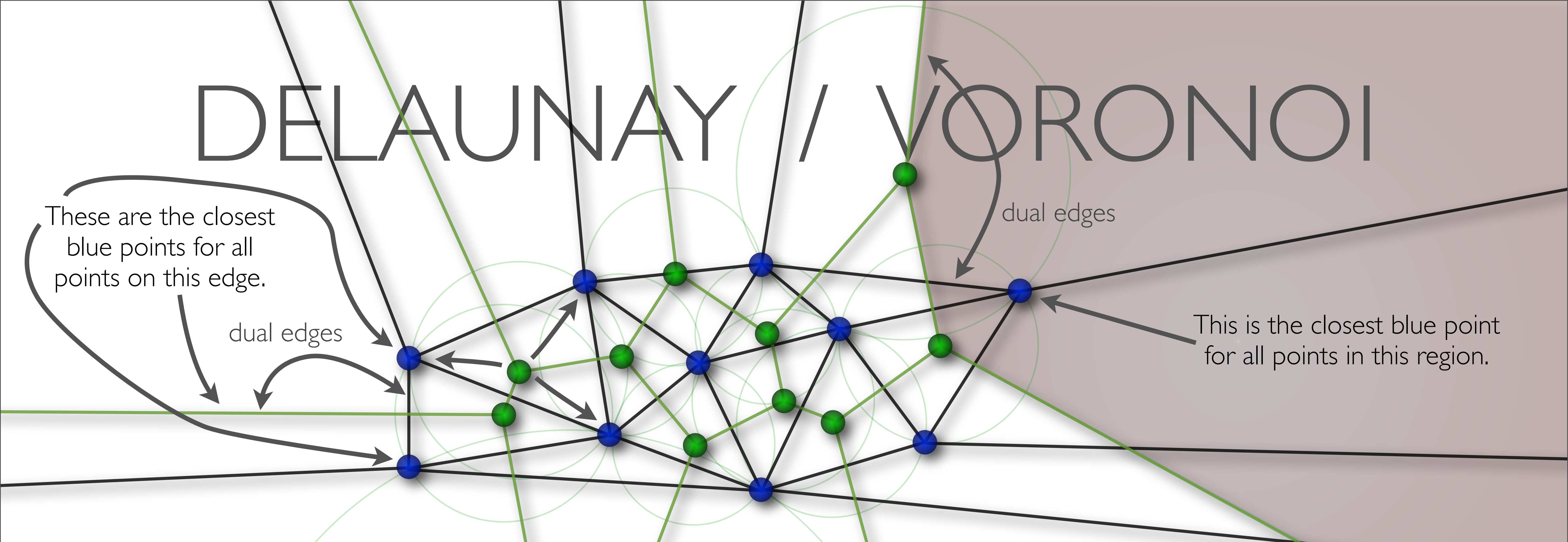
Internally, the range insertion uses `CGAL::spatial_sort()` to determine a good insertion order.

This function is generally useful to speedup batch processing, for instance, when localizing many points in a triangulation...

NB: Watch out in case of duplicate input points: These are inserted once only. (The points of a triangulation form a set, not a multiset.)

https://elabs.inf.ethz.ch/file.php/29/CGALWeek2/Sample_Programs/delaunay.cpp

DELAUNAY / VORONOI



The Delaunay Triangulation has several nice properties:

- It is the straight-line dual of the *Voronoi-Diagram*.

Delaunay vertex ≅ Voronoi face,
Delaunay triangle ≅ Voronoi vertex.

The Voronoi-Diagram for a set P of points partitions the plane into regions for which the closest point from P is the same.

For points ...

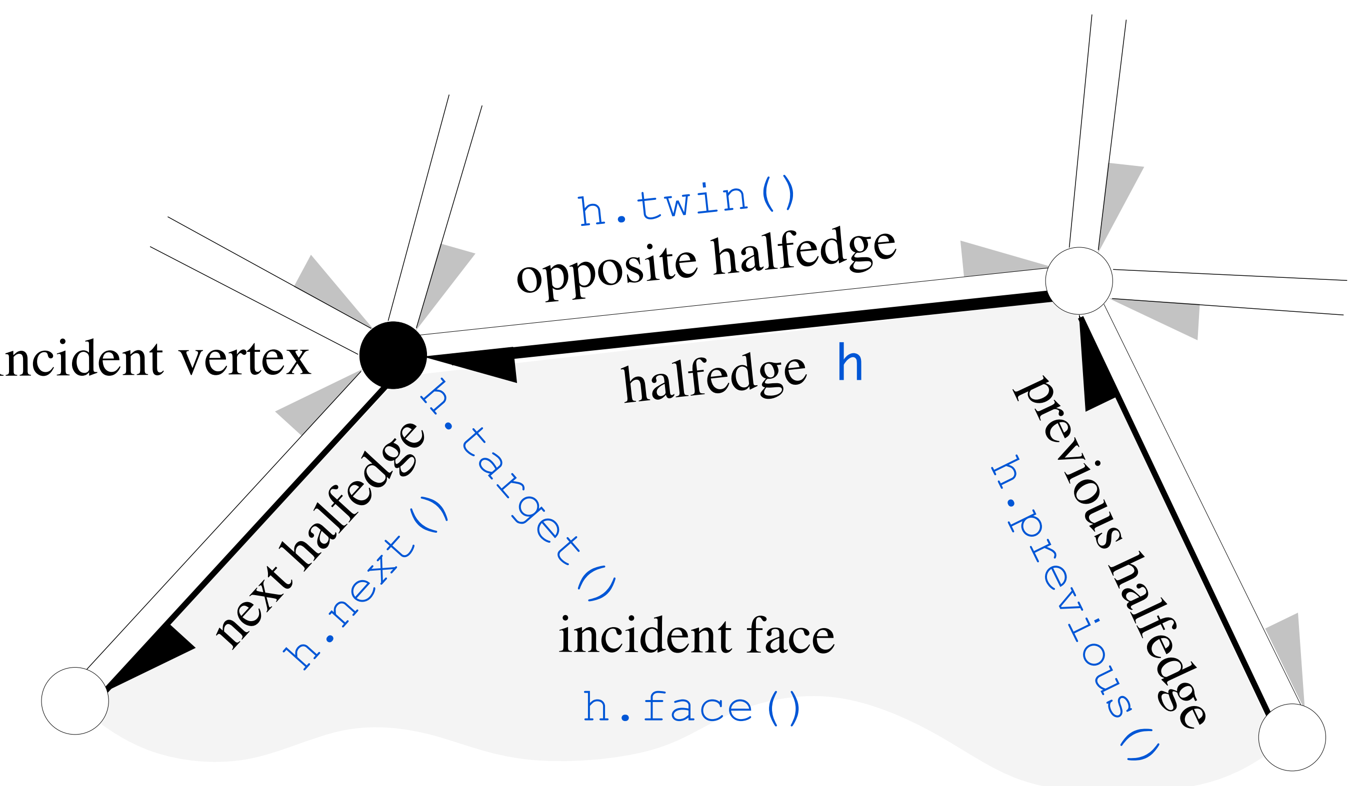
- in the interior of a Voronoi region, there is one closest point from P;
- in the relative interior of a Voronoi edge, there are two closest points from P;
- on a Voronoi vertex, there are three (or more) closest points from P.

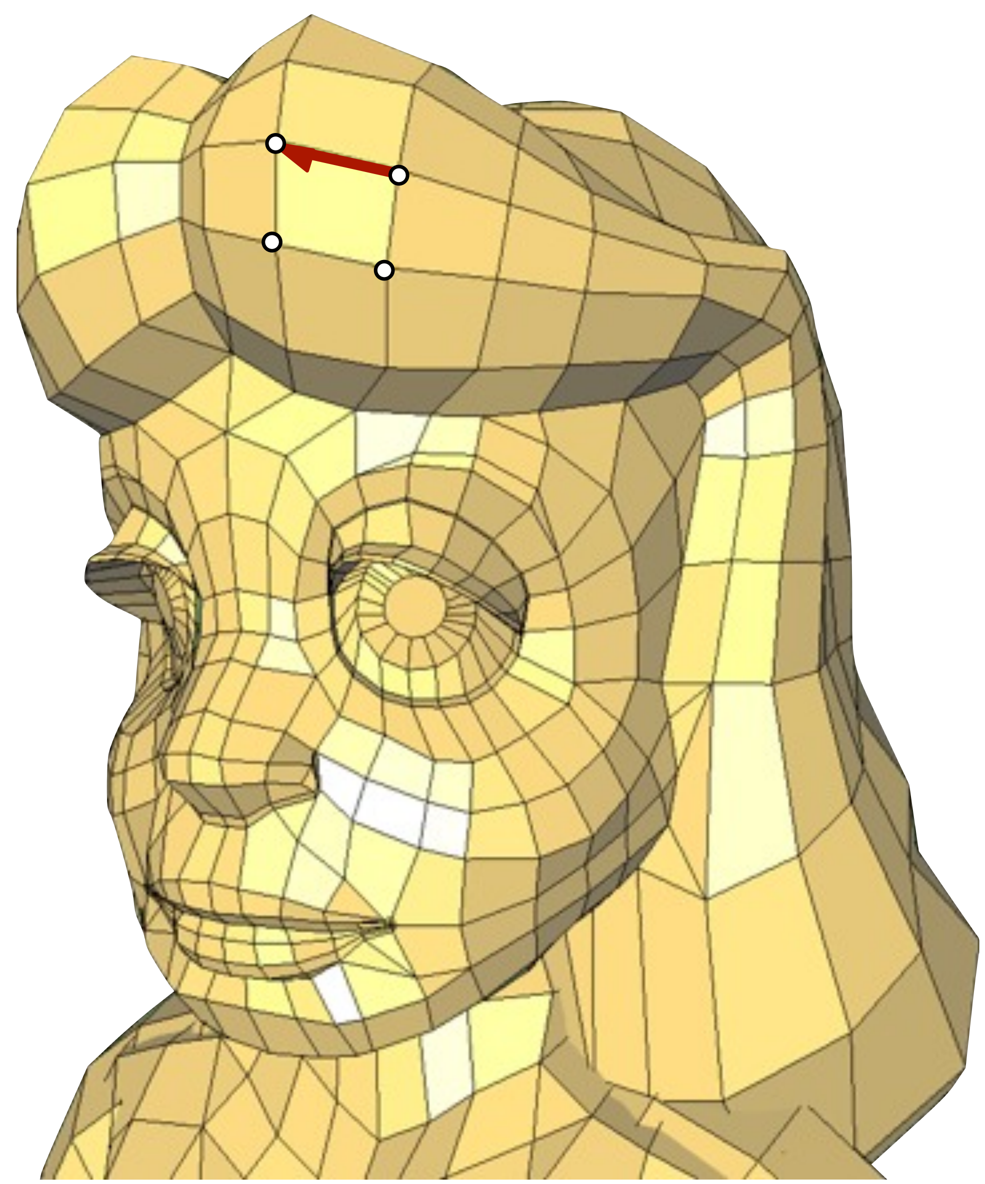
A Delaunay edge is a convex hull edge <=> its dual Voronoi edge is a ray.

HALFEDGE DATA STRUCTURE

a.k.a. Doubly Connected Edge List (DCEL)

Standard representation for orientable 2-manifolds.

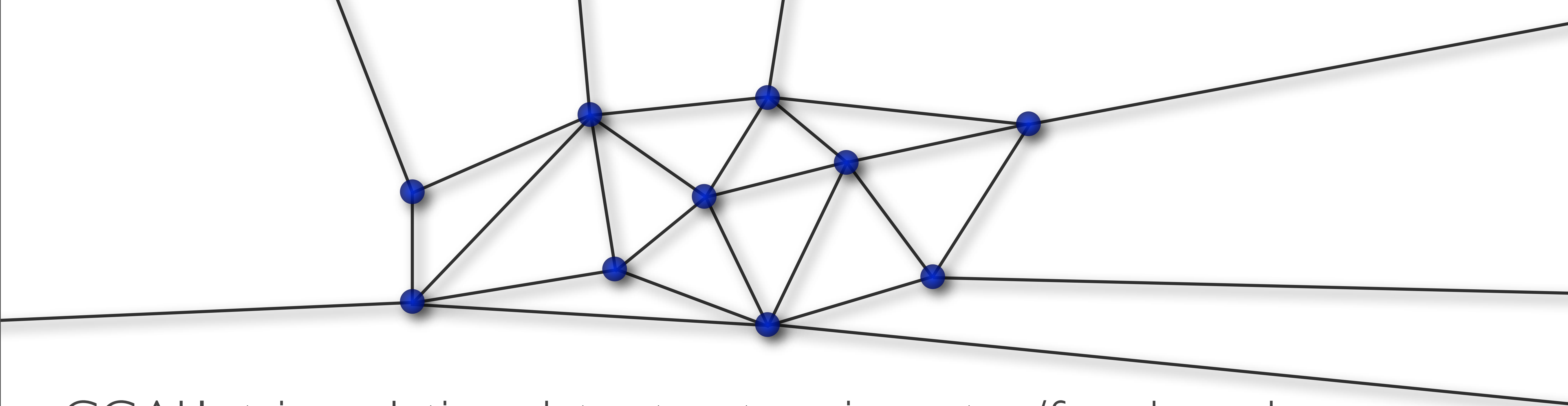




~~Representation for Voronoi diagram.~~

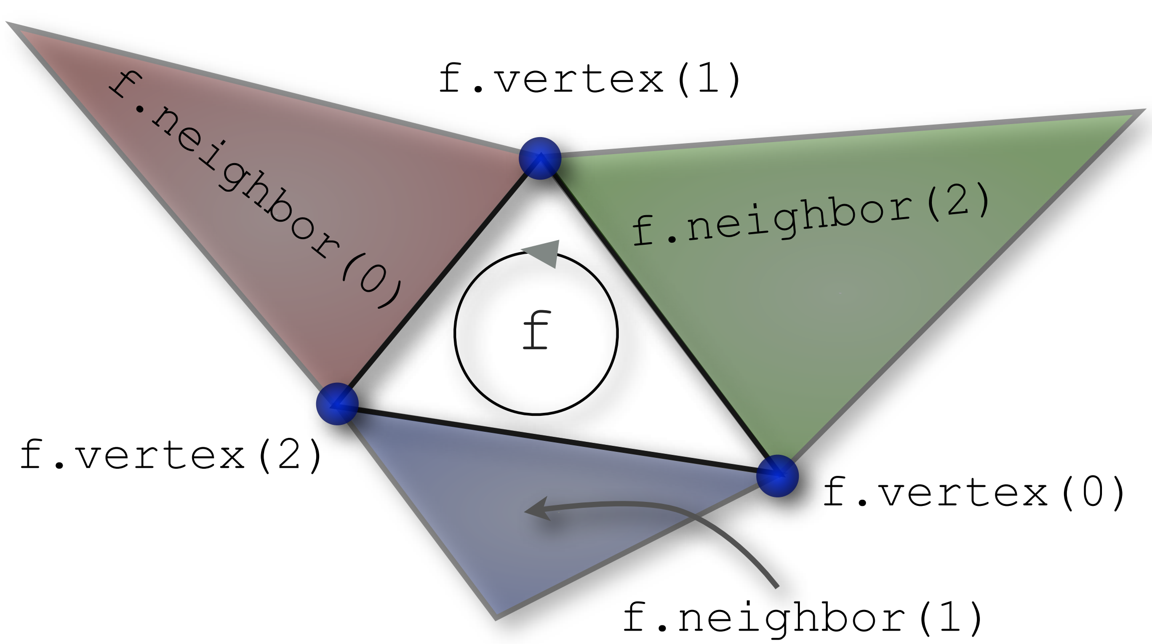
Not for Delaunay...

TRIANGULATION DATA STRUCTURE



CGAL's triangulation data structure is vertex/face based.
Edges are represented implicitly only.

Similarly in 3D it is vertex/cell based.



Space consumption is ~12n instead of ~30n for DCEL

Geometric information is stored at vertices: each vertex has a `.point()` member function.

EDGE REPRESENTATION

Edges in CGAL::Triangulation_data_structure_2 are represented as a `std::pair<Face_handle,int>`.

A pair (`f,i`) represents the i-th edge along the boundary of `*f`.

$0 <= i < 3$

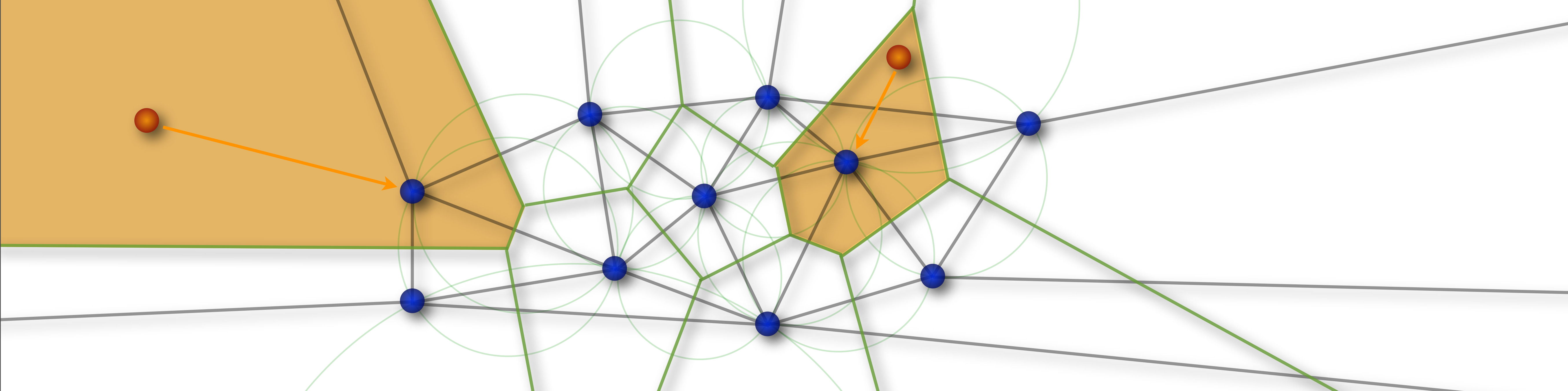
The edge connects the vertices `(i+1)%3` and `(i+2)%3` of `*f`.

Therefore, we can obtain the vertices of an edges as follows:

```
...
Triangulation::Edge e;
...
// get the vertices of e
Triangulation::Vertex_handle v1 = e.first->vertex((e.second + 1) % 3);
Triangulation::Vertex_handle v2 = e.first->vertex((e.second + 2) % 3);
std::cout << "e = " << v1->point() << " <-> " << v2->point() << std::endl;
...
```

If we wanted these points only, `t.segment(e)` would have done it. But if we need the vertices...

DELAUNAY / VORONOI



Post Office Problem:
Process a set P of n points, s.t. for any given query point q (not necessarily from P) the closest point from P can be found quickly.

→ Find Voronoi region that contain q.

The Delaunay triangulation offers `t.nearest_vertex()`, which often is much more efficient than computing the Voronoi diagram.

Why? Because it uses predicates only...

VORONOI DIAGRAMS

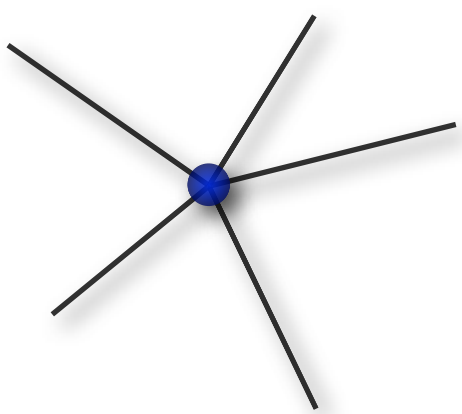
There is an explicit Voronoi adaptor in CGAL.
But for our purposes, we can extract all information needed from the Delaunay triangulation.

```
...
// process all Voronoi vertices
for (Face_iterator f = t.finite_faces_begin(); f != t.finite_faces_end(); ++f) {
  K::Point_2 p = t.dual(f);
  ...
}
// process all Voronoi edges
for (Edge_iterator e = t.finite_edges_begin(); e != t.finite_edges_end(); ++e) {
  CGAL::Object o = t.dual(e);
  // o can be a segment, a ray or a line ...
  ...
}
...
```

CIRCULATORS

… are like iterators, but for circular rather than linear structures.

For instance, the circular sequence of edges incident to a vertex in a triangulation.



For a circulator c, the range [c,c) denotes the full circular sequence.

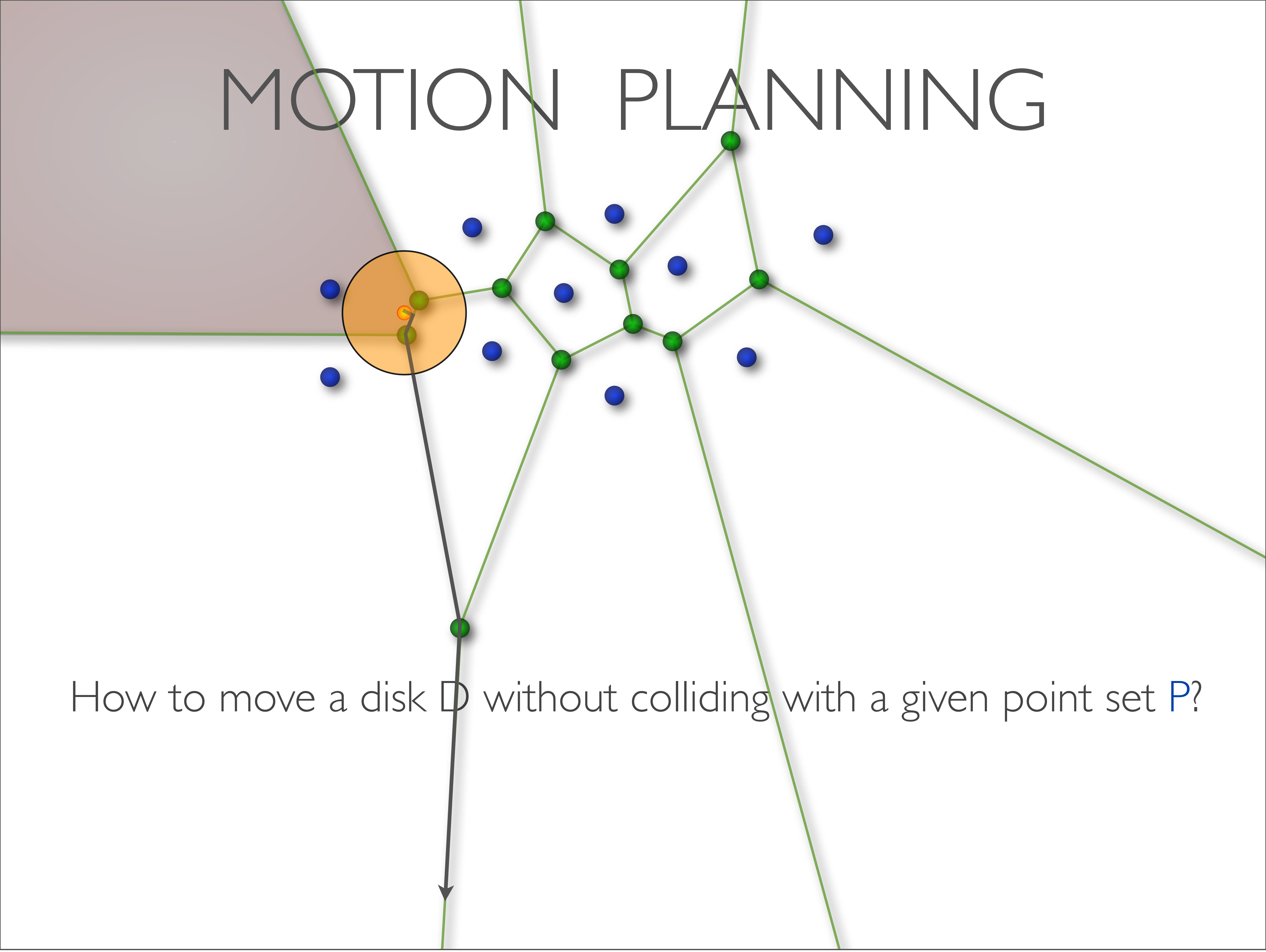
In contrast to iterators, where such a range is empty.

```
Triangulation t;
...
Triangulation::Vertex_handle v = ...;
// find all infinite edges incident to v
Triangulation::Edge_circulator c = t.incident_edges(v);
do {
  if (t.is_infinite(c)) { ... }
  ...
} while (++c != t.incident_edges(v));
```

The usual loop construct to circulate is `do ... while`. It ensures at least one iteration and the following increment and therefore works as desired for full circular ranges.

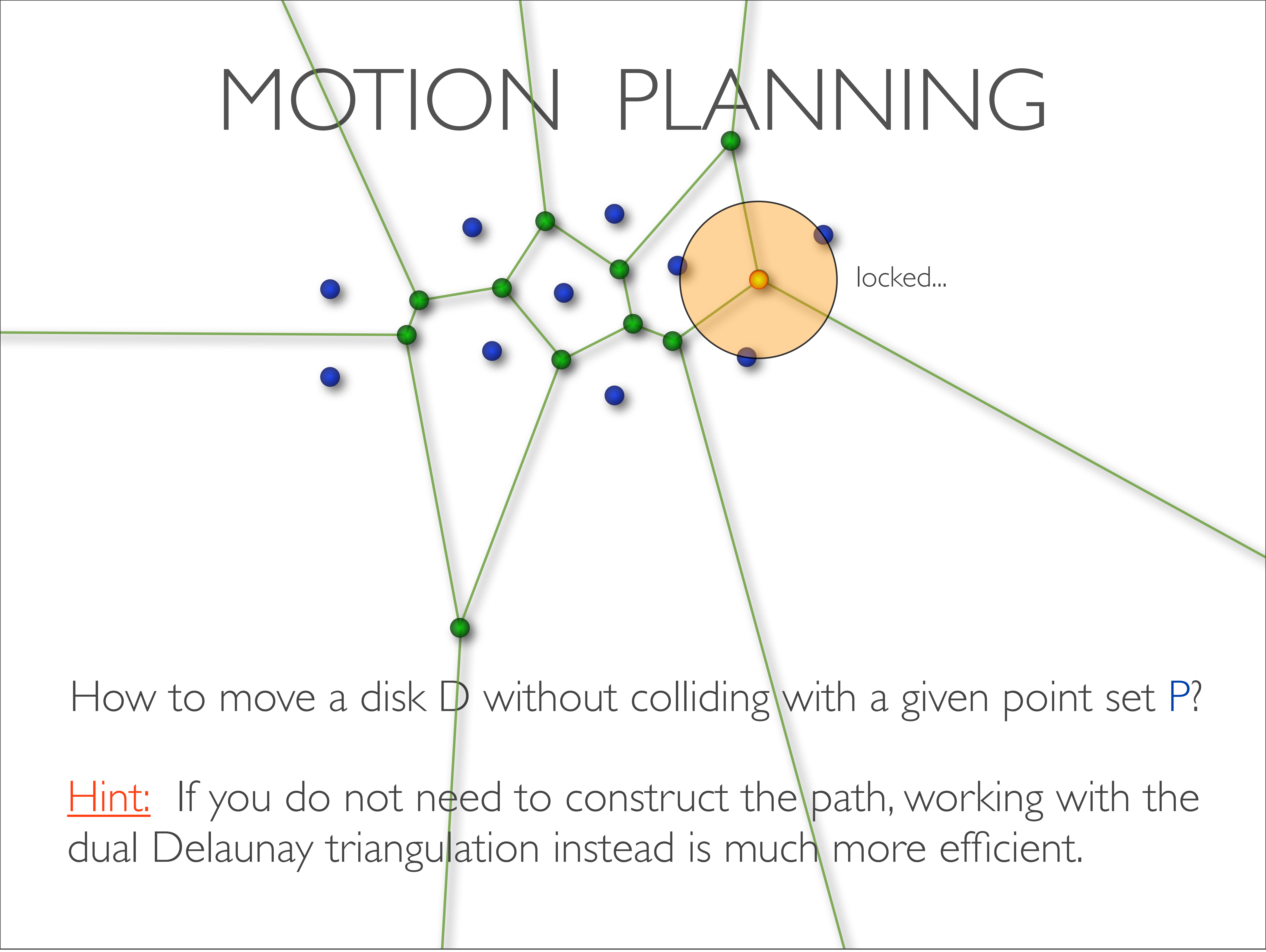
There are no isolated vertices in a triangulation. Otherwise, we would have to test `c != 0` first. (This is the way to describe an empty circular range.)

MOTION PLANNING



How to move a disk D without colliding with a given point set P?

MOTION PLANNING



How to move a disk D without colliding with a given point set P?

Hint: If you do not need to construct the path, working with the dual Delaunay triangulation instead is much more efficient.

ENHANCING FACES I

Add information (e.g., color) to a face using an external map.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <map>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
enum Color { Black = 0, White = 1, Red = 2 };
typedef std::map<Triangulation::Face_handle,Color> Colormap;

...
Triangulation t;
Colormap colors;
...
// color all finite faces white
for (Face_iterator f = t.finite_faces_begin(); f != t.finite_faces_end(); ++f)
  colors[f] = White;
...
```

Can be done in the same way for vertices and edges. (For edges, there are no handles, but the edge type can be used directly.)

ENHANCING FACES II

Add information to a face by storing it in the face directly.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>

enum Color { Black = 0, White = 1, Red = 2 };
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Triangulation_face_base_with_info_2<Color,K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds>  Triangulation;

...
Triangulation t;
...
// color all infinite faces black
Triangulation::Face_circulator f = t.incident_faces(t.infinite_vertex());
do {
  f->info() = Black;
} while (++f != t.incident_faces(t.infinite_vertex()));
...
```

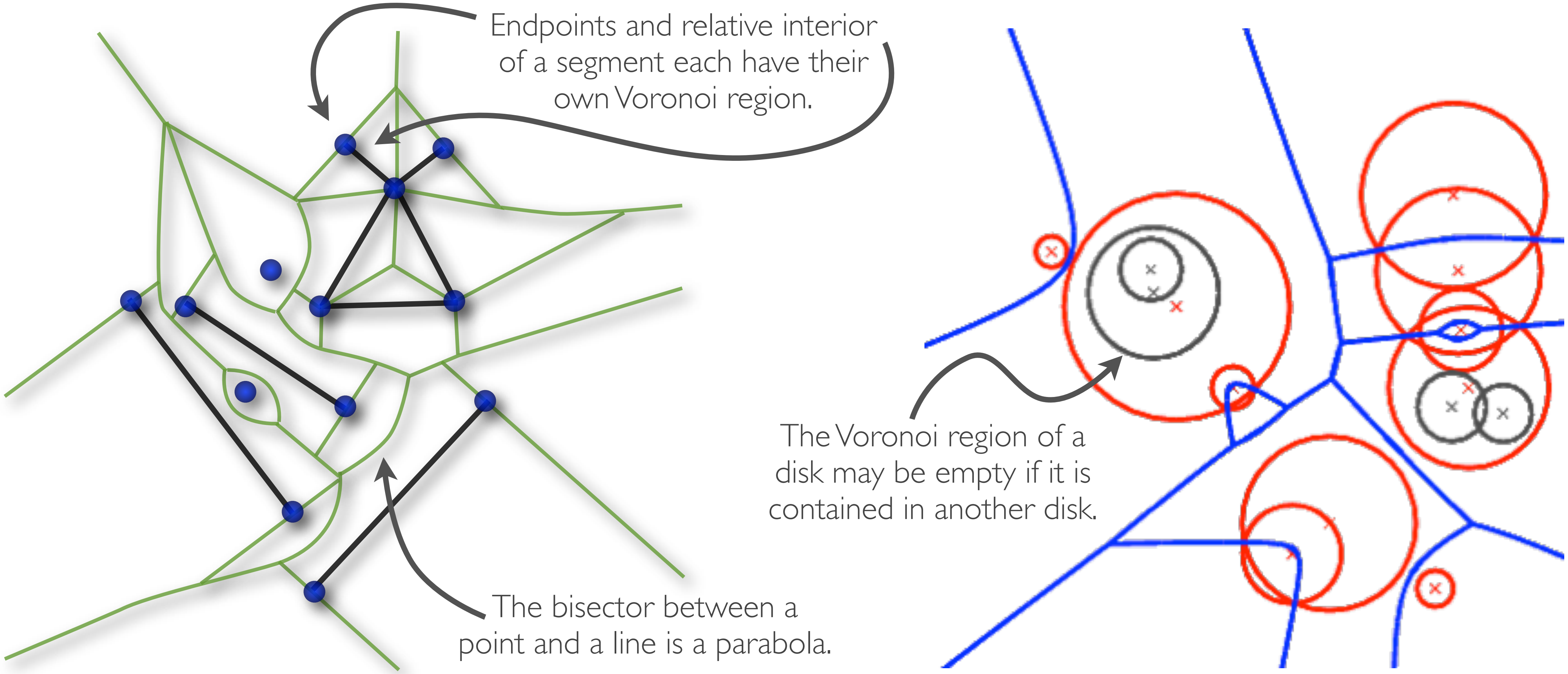
Info parameter. Here: each face gets a `Color`.

New face class, vertex class stays the same.

Change the underlying triangulation data structure (so far we've used the default).

Can be done in the same way for vertices. But for edges this does not work because they are represented implicitly only.

MORE VORONOI / DELAUNAY



Delaunay graphs and Voronoi diagrams can be defined analogously for objects other than points. CGAL has implementations for …

- Delaunay graphs of line segments (`Segment_Delaunay_graph_2`)
- Delaunay graphs of disks (`Apollonius_graph_2`)

Points can be regarded as (degenerate) line segments or disks.

SPECIFIC REFERENCES

- 2D/3D Kernel objects and operations:

http://www.cgal.org/Manual/3.8/doc_html/cgal_manual/Kernel_23_ref/Chapter_intro.html

- 2D Triangulations:

http://www.cgal.org/Manual/3.8/doc_html/cgal_manual/Triangulation_2/Chapter_main.html

- Voronoi:

http://www.cgal.org/Manual/3.8/doc_html/cgal_manual/Voronoi_diagram_2/Chapter_main.html

- Segment Delaunay:

http://www.cgal.org/Manual/3.8/doc_html/cgal_manual/Segment_Delaunay_graph_2/Chapter_main.html

In general, you'll have to follow a couple of links to find what you're after.

For instance, in order to find what `Triangulation_2::Edge` is about, go to the concept `TriangulationDataStructure_2`.

IO WITH EXACT FTS

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <iostream>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;

...
// this is nicer ...
K::Point_2 p;
std::cin >> p;
...
// this is much faster ...
double x, y;
std::cin >> x >> y;
K::Point_2 p(x, y);
```

assuming the input fits into a `double`...

But there is no (noticeable) difference for `CGAL::Exact_predicates_inexact_constructions_kernel`.