Algorithms for 3D Printing and Other Manufacturing Methodologies

Efi Fogel

Tel Aviv University

2D Arrangements
Apr. 24th, 2017

Outline

1. 2D Arrangements
   - Definitions & Complexity
   - Representation
   - Queries
     - Vertical Decomposition
     - Point Location Queries
   - The Zone Computation Algorithmic Framework
   - The Plane Sweep Algorithmic Framework
   - Arrangement of Unbounded Curves
   - Literature

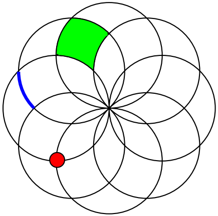
Outline

1. 2D Arrangements
   - Definitions & Complexity
   - Representation
   - Queries
     - Vertical Decomposition
     - Point Location Queries
   - The Zone Computation Algorithmic Framework
   - The Plane Sweep Algorithmic Framework
   - Arrangement of Unbounded Curves
   - Literature

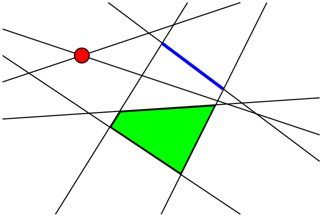
Two Dimensional Arrangements

Definition (Arrangement)

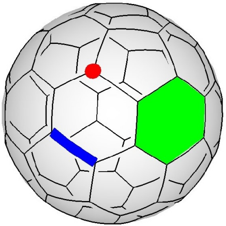
Given a collection $\mathscr{C}$ of curves on a surface, the arrangement $\mathscr{A}(\mathscr{C})$ is the partition of the surface into vertices, edges and faces induced by the curves of $\mathscr{C}$.



An arrangement of circles in the plane.



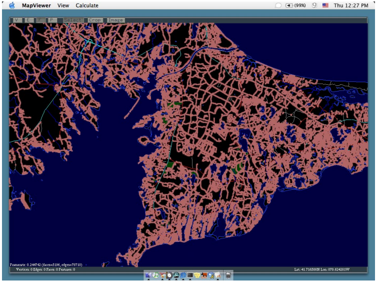
An arrangement of lines in the plane.

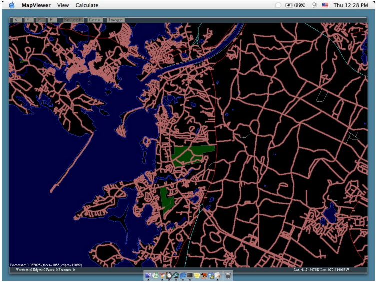


An arrangement of great-circle arcs on a sphere.

Arrangement Background

- Arrangements have numerous applications
  - robot motion planning, computer vision, GIS, optimization, computational molecular biology





A planar map of the Boston area showing the top of the arm of cape cod.

Raw data comes from the US Census 2000 TIGER/line data files

Arrangement 2D Complexity

Definition (Well Behaved Curves)

Curves in a set $\mathscr{C}$ are well behaved, if each pair of curves in $\mathscr{C}$ intersect at most some constant number of times.

Theorem (Arrangement in $\mathbb{R}^2$)

The maximum combinatorial complexity of an arrangement of n well-behaved curves in the plane is $\Theta(n^2)$.

The complexity of arrangements induced by n non-parallel lines is $\Omega(n^2)$.

Arrangement dD Complexity

Definition (Hyperplane)

A hyperplane is the set of solutions to a single equation $AX = c$, where A and X are vectors and c is some constant.

A hyperplane is any codimension-1 vector subspace of a vector space.

Definition (Hypersurface)

A hypersurface is the set of solutions to a single equation $f(x_1, x_2, \ldots, x_n) = 0$.

Theorem (Arrangement in $\mathbb{R}^d$)

The maximum combinatorial complexity of an arrangement of n well-behaved (hyper)surfaces in $\mathbb{R}^d$ is $\Theta(n^d)$.

The complexity of arrangements induced by n non-parallel hyperplanes is $\Omega(n^d)$.

Planar Maps

Definition (Planar Graph)

A planar graph is a graph that can be embedded in the plane.

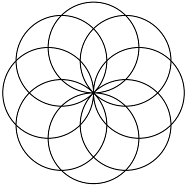
Definition (Planar Map)

A planar map is the embedding of a planar graph in the plane. It is a subdivision of the plane into vertices, (bounded) edges, and faces.

Theorem (Euler Formula)

Let v, e, and f be the number of vertices, edges, and faces (including the unbounded face) of a planar map, then $v - e + f = 2$.

8 circles



vertices — 25

edges — 56

faces — 33 (including the unbounded face)

Surface Maps

Planar maps generalize to surfaces!

Definition (genus)

A topologically invariant property of a surface defined as the largest number of nonintersecting simple closed curves that can be drawn on the surface without separating it.

Theorem (Euler Formula)

Let v, e, and f be the number of vertices, edges, and faces of a map embedded on a surface with genus g, then $v - e + f = 2 - 2g$.

If each face is incident to at least 3 edges $\Longrightarrow 3f \leq 2e$

$$3v - 3e + 3f = 6 - 6g \leq 3v - 3e + 2e$$

$$e \leq 3v - 6 + 6g$$

In a planar triangulation $e = 3v - 6,\ f = 2v - 4$

Outline

1. 2D Arrangements
   - Definitions & Complexity
   - Representation
   - Queries
     - Vertical Decomposition
     - Point Location Queries
   - The Zone Computation Algorithmic Framework
   - The Plane Sweep Algorithmic Framework
   - Arrangement of Unbounded Curves
   - Literature

The CGAL Arrangement_on_surface_2 Package

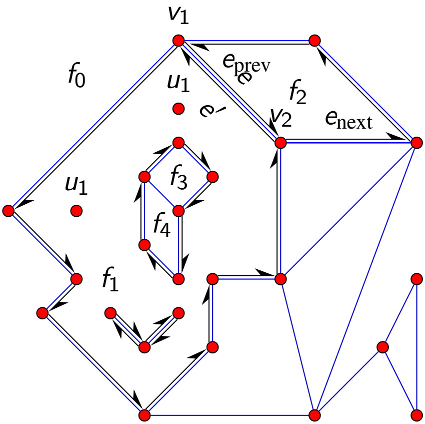
- Constructs, maintains, modifies, traverses, queries, and presents arrangements on two-dimensional parametric surfaces.
- Complete and Robust
  - All inputs are handled correctly (including degenerate input).
  - Exact number types are used to achieve robustness.
- Generic – easy to interface, extend, and adapt
- Modular – geometric and topological aspects are separated
- Supports among the others:
  - various point location strategies
  - zone-construction paradigm
  - sweep-line paradigm
    - ⋆ vertical decomposition
    - ⋆ overlay computation
    - ⋆ batched point location
- Part of the CGAL basic library

Arrangement_2<Traits ,Dcel>

- Is the main component in the *2D Arrangements* package.
- An instance of this class template represents 2D arrangements.
- The representation of the arrangements and the various geometric algorithms that operate on them are separated.
- The topological and geometric aspects are separated.
  - The Traits template-parameter must be substituted by a model of a geometry-traits concept, e.g., *ArrangementBasicTraits_2*.
    - ⋆ Defines the type X_monotone_curve_2 that represents x-monotone curves.
    - ⋆ Defines the type Point_2 that represents two-dimensional points.
    - ⋆ Supports basic geometric predicates on these types.
  - The Dcel template-parameter must be substituted by a model of the *ArrangementDcel* concept, e.g., Arr_default_dcel<Traits>.

The Doubly-Connected Edge List

- One of a family of combinatorial data-structures called the *halfedge data-structures*.
- Represents each edge using a pair of directed *halfedges*.
- Maintains incidence relations among cells of 0 (vertex), 1 (edge), and 2 (face) dimensions.



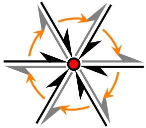
- The target vertex of a halfedge and the halefedge are incident to each other.
- The source and target vertices of a halfedge are adjacent.

The Doubly-Connected Edge List Components

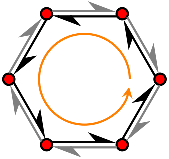
- Vertex
  - An incident halfedge pointing at the vertex.
- Halfedge
  - The opposite halfedge.
  - The previous halfedge in the component boundary.
  - The next halfedge in the component boundary.
  - The target vertex of the halfedge.
  - The incident face.
- Face
  - An incident halfedge on the outer CCB.
  - An incident halfedge on each inner CCB.
- Connected component of the boundary (CCB)
  - The circular chains of halfedges around faces.

Arrangement Representation

- The halfedges incident to a vertex form a circular list.
- The halfedges are clockwise oriented around the vertex.

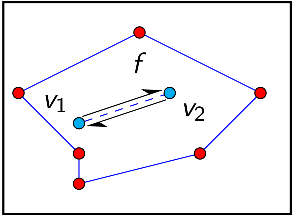


- The halfedges around faces form circular chains.
- All halfedges of a chain are incident to the same face.
- The halfedges are counterclockwise oriented along the boundary.

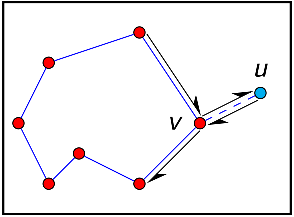


- Geometric interpretation is added by classes built on top of the halfedge data-structure.

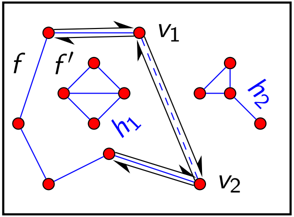
Modifying the Arrangement



Inserting a curve that induces a new hole inside the face f, `arr.insert_in_face_interior(c,f)`.



Inserting a curve from an existing vertex u that corresponds to one of its endpoints, `insert_from_left_vertex(c,v)`, `insert_from_right_vertex(c,v)`.



Inserting an x-monotone curve, the endpoints of which correspond to existing vertices v_1 and v_2, `insert_at_vertices(c,v1,v2)`.

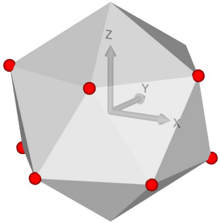
- The new pair of halfedges close a new face f'.
- The hole h_1, which belonged to f before the insertion, becomes a hole in this new face.

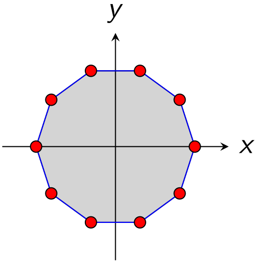
Application: Obtaining Silhouettes of Polytopes

Application

Given a convex polytope P obtain the outline of the shadow of P cast on the xy-plane, where the scene is illuminated by a light source at infinity directed along the negative z-axis.

- The silhouette is represented as an arrangement with two faces:
  - an unbounded face and
  - a single hole inside the unbounded face.





An icosahedron and its silhouette.

Application: Obtaining Silhouettes of Polytopes: Insertion

- Insert an edge into the arrangement only once to avoid overlaps.
  - Maintain a set of handles to polytope edges the projection of which have already been inserted into the arrangement.
  - Implemented with the `std::set` data-structure.
    - ⋆ Requires the provision of a model of the *StrictWeakOrdering*.
    - ⋆ A functor that compares handles:

```
struct Less_than_handle {
  template <typename Type>
  bool operator()(Type s1, Type s2) const { return (&(*s1) < &(*s2)); }
};
```

```
std::set<Polyhedron_halfedge_const_handle, Less_than_handle>}
```

- Determine the appropriate insertion routines.
  - Maintain a map that maps polyhedron vertices to corresponding arrangement vertices.
  - Implemented with the `std::map` data-structure.

```
std::map<typename Polyhedron::Vertex_const_handle,
         typename Arrangement::Vertex_handle, Less_than_handle>
```

Application: Obtaining Silhouettes of Polytopes: Construction

Obtain the arrangement $\mathscr{A}$ that represents the silhouette of a Convex Polytope P
1. Construct the input convex polytope P.
2. Compute the normals to all facets of P.
3. **for each** facet f of P
4.   **if** f is facing upwards (has a positive z component)
5.   **for each** edge e on the boundary of f
6.   **if** the projection of e hasn't been inserted yet into $\mathscr{A}$
7.   Insert the projection of e into $\mathscr{A}$.

Computes the normal to a facet.

```
struct Normal_equation {
  template <typename Facet> typename Facet::Plane_3 operator()(Facet& f) {
    typename Facet::Halfedge_handle h = f.halfedge();
    return CGAL::cross_product(h->next()->vertex()->point() -
                               h->vertex()->point(),
                               h->next()->next()->vertex()->point() -
                               h->next()->vertex()->point());
  }
};
```

Traversing the Halfedges Incident to an Arrangement Vertex

Print all the halfedges incident to a vertex.

```
template <typename Arrangement>
void print_incident_halfedges(typename Arrangement::Vertex_const_handle v)
{
  if (v->is_isolated()) {
    std::cout << "The␣vertex␣(" << v->point() << ")␣is␣isolated" << std::endl;
    return;
  }
  std::cout << "The␣neighbors␣of␣the␣vertex␣(" << v->point() << ")␣are:";
  typename Arrangement::Halfedge_around_vertex_const_circulator  first, curr;
  first = curr = v->incident_halfedges();
  do std::cout << "␣(" << curr->source()->point() << ")";
  while (++curr != first);
  std::cout << std::endl;
}
```

Traversing the Halfedges of an Arrangement CCB

Print all x-monotone curves along a given CCB

```
template <typename Arrangement>
void print_ccb(typename Arrangement::Ccb_halfedge_const_circulator circ)
{
  std::cout << "(" << circ->source()->point() << ")";
  typename Arrangement::Ccb_halfedge_const_circulator  curr = circ;
  do {
    typename Arrangement::Halfedge_const_handle he = curr;
    std::cout << "   [" << he->curve() << "]   "
              << "(" << he->target()->point() << ")";
  } while (++curr != circ);
  std::cout << std::endl;
}
```

- `he->curve()` is equivalent to `he->twin()->curve()`,
- `he->source()` is equivalent to `he->twin()->target()`, and
- `he->target()` is equivalent to `he->twin()->source()`.

Traversing the CCBs of an Arrangement Face

Print the outer and inner boundaries of a face.

```
template <typename Arrangement>
void print_face(typename Arrangement::Face_const_handle f)
{
  // Print the outer boundary.
  if (f->is_unbounded()) std::cout << "Unbounded face. " << std::endl;
  else {
    std::cout << "Outer boundary: ";
    print_ccb<Arrangement>(f->outer_ccb());
  }

  // Print the boundary of each of the holes.
  size_t index = 1;
  typename Arrangement::Hole_const_iterator  hole;
  for (hole = f->holes_begin(); hole != f->holes_end(); ++hole, ++index) {
    std::cout << "    Hole #" << index << ": ";
    print_ccb<Arrangement>(*hole);
  }

  // Print the isolated vertices.
  typename Arrangement::Isolated_vertex_const_iterator iv;
  for (iv = f->isolated_vertices_begin(), index = 1;
       iv != f->isolated_vertices_end(); ++iv, ++index)
    std::cout << "    Isolated vertex #" << index << ": "
              << "(" << iv->point() << ")" << std::endl;
}
```

Traversing an Arrangement

Print all the cells of an arrangement.

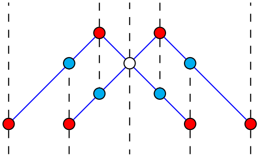
```
template <typename Arrangement>
void print_arrangement(const Arrangement& arr)
{
  CGAL_precondition(arr.is_valid());

  // Print the arrangement vertices.
  typename Arrangement::Vertex_const_iterator vit;
  std::cout << arr.number_of_vertices() << "␣vertices:" << std::endl;
  for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
    std::cout << "(" << vit->point() << ")";
    if (vit->is_isolated()) std::cout << "␣-␣Isolated." << std::endl;
    else std::cout << "␣-␣degree␣" << vit->degree() << std::endl;
  }

  // Print the arrangement edges.
  typename Arrangement::Edge_const_iterator eit;
  std::cout << arr.number_of_edges() << "␣edges:" << std::endl;
  for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
    std::cout << "[" << eit->curve() << "]" << std::endl;

  // Print the arrangement faces.
  typename Arrangement::Face_const_iterator fit;
  std::cout << arr.number_of_faces() << "␣faces:" << std::endl;
  for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
    print_face<Arrangement>(fit);
}
```

Outline

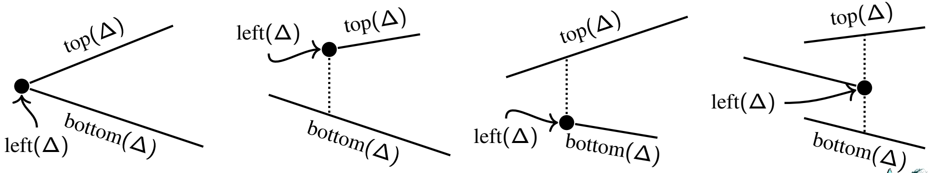
1. 2D Arrangements
   - Definitions & Complexity
   - Representation
   - Queries
     - Vertical Decomposition
     - Point Location Queries
   - The Zone Computation Algorithmic Framework
   - The Plane Sweep Algorithmic Framework
   - Arrangement of Unbounded Curves
   - Literature

Vertical Decomposition

- Is a refinement of the original subdivision $\mathscr{A}$ of n edges.
- In the plane
  - Contains $O(n)$ pseudo trapezoids (triangles and trapezoids).
  - A pseudo trapezoid is determined by
    - ★ 2 vertices $\mathrm{left}(\Delta)$ and $\mathrm{right}(\Delta)$, and
    - ★ 2 segments $\mathrm{top}(\Delta)$ and $\mathrm{bottom}(\Delta)$.



- Generalizes to higher dimensions and arrangements induces by well behaved objects.



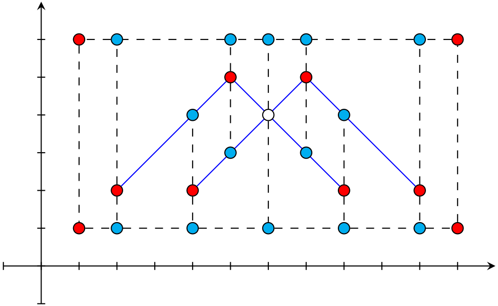
Vertical Decomposition Complexity

- R—a bounding rectangle
- S—a set of n interior disjoint line segments
- $\mathscr{T}(S)$—the trapezoidal map of S
- $\mathscr{T}(S)$ is a planar map with v vertices, e edges, and f faces
- A vertex of $\mathscr{T}(S)$ is either
  - a vertex of R,
  - an endpoint of a segment in S, or
  - the point where the vertical extension hits
- $v \leq 4 + 2n + 2(2n) = 6n + 4$
- $f \leq 3n + 1$
  - The lower left corner of R is $\mathrm{left}(\Delta)$ of one trapezoid
  - The right endpoint of a segment can be $\mathrm{left}(\Delta)$ of one trapezoid
  - The left endpoint of a segment can be $\mathrm{left}(\Delta)$ of two trapezoid

Application: Decomposing an Arrangement of Line Segments

Application

Constructs the vertical decomposition of a given arrangement.

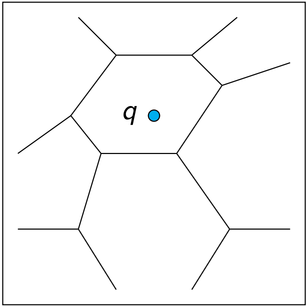


Decomposing an Arrangement of Line Segments: Code

```
template <typename Arrangement, typename Kernel>
void vertical_decomposition(Arrangement& arr, Kernel& ker)
{
  typedef std::pair<typename Arrangement::Vertex_const_handle,
                    std::pair<CGAL::Object, CGAL::Object> > Vd_entry;

  // For each vertex in the arrangment, locate the feature that lies
  // directly below it and the feature that lies directly above it.
  std::list<Vd_entry> vd_list;
  CGAL::decompose(arr, std::back_inserter(vd_list));

  // Go over the vertices (given in ascending lexicographical xy-order),
  // and add segements to the feautres below and above it.
  const typename Kernel::Equal_2 equal = ker.equal_2_object();
  typename std::list<Vd_entry>::iterator  it, prev = vd_list.end();
  for (it = vd_list.begin(); it != vd_list.end(); ++it) {
    // If the feature above the previous vertex is not the current vertex,
    // Add a vertical segment to the feature below the vertex.
    typename Arrangement::Vertex_const_handle v;
    if ((prev == vd_list.end()) ||
        !CGAL::assign(v, prev->second.second) ||
        !equal(v->point(), it->first->point()))
      add_vertical_segment(arr, arr.non_const_handle(it->first), it->second.first, ker);
    // Add a vertical segment to the feature above the vertex.
    add_vertical_segment(arr, arr.non_const_handle(it->first), it->second.second, ker);
    prev = it;
  }
}
```

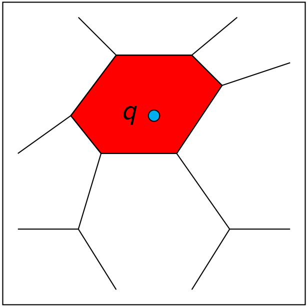
Arrangement Point Location

Given a subdivision A of the space into cells and a query point q, find the cell of A containing q.



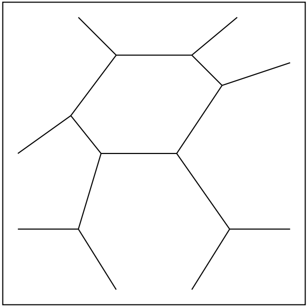
Arrangement Point Location

Given a subdivision A of the space into cells and a query point q, find the cell of A containing q.



Arrangement Point Location

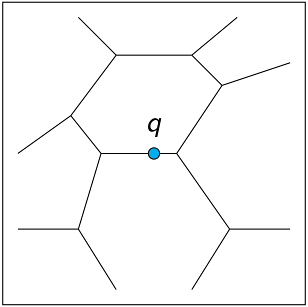
Given a subdivision A of the space into cells and a query point q, find the cell of A containing q.



- In degenerate situations the query point can

Arrangement Point Location

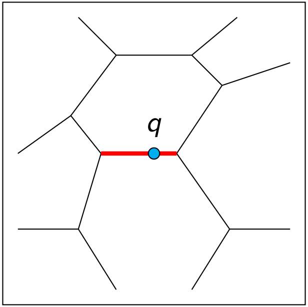
Given a subdivision A of the space into cells and a query point q, find the cell of A containing q.



- In degenerate situations the query point can
  - lie on an edge, or

Arrangement Point Location

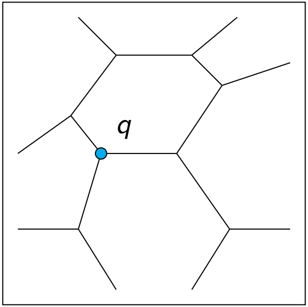
Given a subdivision A of the space into cells and a query point q, find the cell of A containing q.



- In degenerate situations the query point can
  - lie on an edge, or

Arrangement Point Location

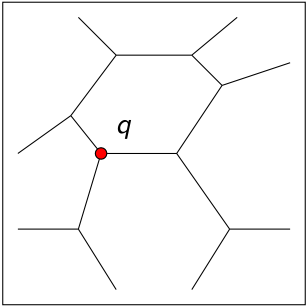
Given a subdivision A of the space into cells and a query point q, find the cell of A containing q.



- In degenerate situations the query point can
  - lie on an edge, or
  - coincide with a vertex.

Arrangement Point Location

Given a subdivision A of the space into cells and a query point q, find the cell of A containing q.



- In degenerate situations the query point can
  - lie on an edge, or
  - coincide with a vertex.

Point Location Algorithms

- Traditional Point Location Strategies
  - Hierarchical data structure [Kir83]
  - Persistent search trees [ST86]
  - Random Incremental Construction [Mul91, Sei91]
- Point-location in Triangulations
  - Walk along a line [DPT02]
  - The Delaunay Hierarchy [Dev02]
  - Jump & Walk [DMZ98, DLM99]
- Other algorithms
  - Entropy based algorithms [Ary01]
  - Point location using Grid [EKA84]

CGAL Point Location Strategies

- Naive
  - Traverse all edges of the arrangement to find the closest.
- Walk along line
  - Walk along a vertical line from infinity.
- Trapezoidal map **R**andomized **I**ncremental-**C**onstruction (RIC)
- Landmark

Walk Along a Line

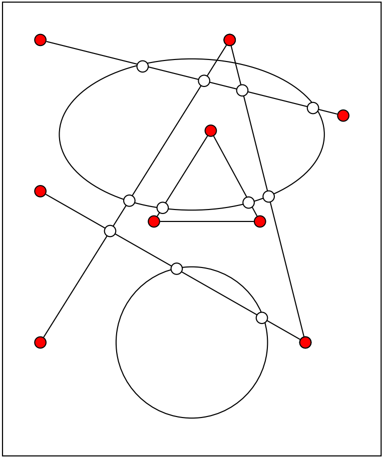
- Start from a known place in the arrangement and walk from there towards the query point through a straight line.
  - No preprocessing performed.
  - No storage space consumed.
- The implementation in CGAL:
  - Start from the unbounded face.
  - Walk down to the point through a vertical line.
  - Asymptotically $O(n)$ time.
  - In practice: quite good, and easy to maintain.

Triangulation Point Location

- Preprocessing:
  - Triangulate the planar map.
    - ⋆ Triangles are much simpler than the arbitrary shapes of faces.
    - ⋆ $O(n \log n)$ time and $O(n)$ space.
    - ⋆ Retain relations between planar map vertices and triangulation.
- Query:
  - Find the triangle P containing the query point q.
    - ⋆ Walk from an arbitrary vertex.
    - ⋆ $O(n)$ time in the worst case, but $O(\sqrt{n})$ time on average, if the vertices are distributed uniformly at random.
  - Find the face in the arrangement that contains the triangle P.

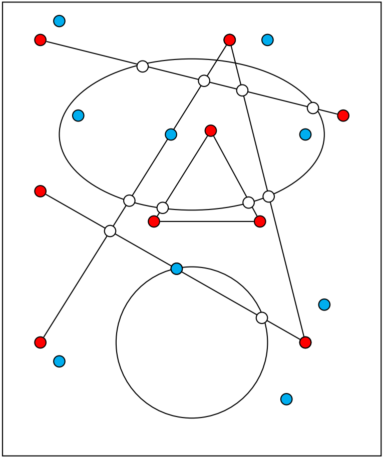
Landmark Point Location

- Given an arrangement $\mathscr{A}$



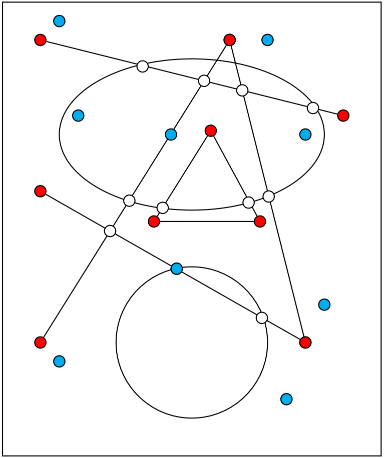
Landmark Point Location

- Given an arrangement $\mathscr{A}$
- Preprocess
  - Choose the landmarks and locate them in $\mathscr{A}$.



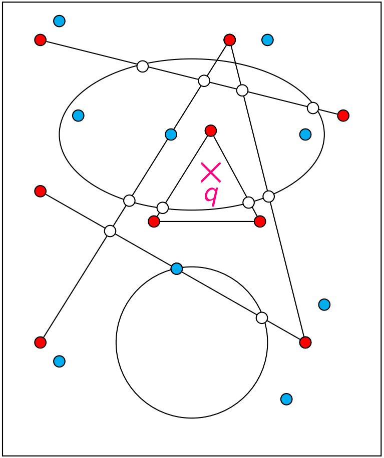
Landmark Point Location

- Given an arrangement $\mathscr{A}$
- Preprocess
  - Choose the landmarks and locate them in $\mathscr{A}$.
  - Store the landmarks in a nearest neighbor search-structure.



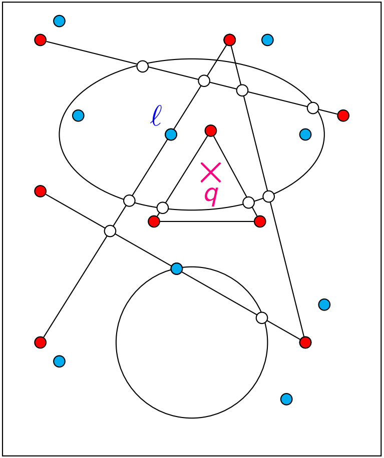
Landmark Point Location

- Given an arrangement $\mathscr{A}$
- Preprocess
  - Choose the landmarks and locate them in $\mathscr{A}$.
  - Store the landmarks in a nearest neighbor search-structure.
- Answer query
  - Given a query point q



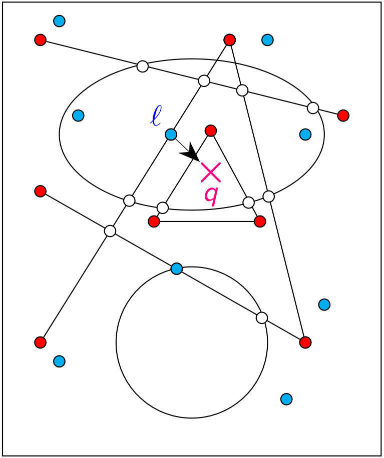
Landmark Point Location

- Given an arrangement $\mathscr{A}$
- Preprocess
  - Choose the landmarks and locate them in $\mathscr{A}$.
  - Store the landmarks in a nearest neighbor search-structure.
- Answer query
  - Given a query point q
  - Find the landmark ℓ closest to q using the search structure.
    - ★ The landmarks are on a grid $\Longrightarrow$ Nearest grid point found in $O(1)$ time.



Landmark Point Location

- Given an arrangement $\mathscr{A}$
- Preprocess
  - Choose the landmarks and locate them in $\mathscr{A}$.
  - Store the landmarks in a nearest neighbor search-structure.
- Answer query
  - Given a query point q
  - Find the landmark ℓ closest to q using the search structure.
    - ★ The landmarks are on a grid $\Longrightarrow$ Nearest grid point found in $O(1)$ time.
  - "Walk along a line" from ℓ to q.



Trapezoidal Map
Randomized Incremental-Construction

- $\mathscr{A}$ — an arrangement.

Trapezoidal Map
Randomized Incremental-Construction

- $\mathscr{A}$ — an arrangement.
- Preprocess
  - For each segment in random order.

Trapezoidal Map
Randomized Incremental-Construction

- $\mathscr{A}$ — an arrangement.
- Preprocess
  - For each segment in random order.
    - ⋆ Update the trapezoidal map.

Trapezoidal Map
Randomized Incremental-Construction

- $\mathscr{A}$ — an arrangement.
- Preprocess
  - For each segment in random order.
    - ⋆ Update the trapezoidal map.
    - ⋆ Insert the new trapezoid into a search structure.
  - $O(n \log n)$ time, $O(n)$ space.

Trapezoidal Map
Randomized Incremental-Construction

- $\mathscr{A}$ — an arrangement.
- Preprocess
  - For each segment in random order.
    - ⋆ Update the trapezoidal map.
    - ⋆ Insert the new trapezoid into a search structure.
  - $O(n \log n)$ time, $O(n)$ space.
- Answer query
  - Given a query point q

Trapezoidal Map
Randomized Incremental-Construction

- $\mathscr{A}$ — an arrangement.
- Preprocess
  - For each segment in random order.
    - ⋆ Update the trapezoidal map.
    - ⋆ Insert the new trapezoid into a search structure.
  - $O(n \log n)$ time, $O(n)$ space.
- Answer query
  - Given a query point q
  - Search the trapezoid in the search structure.

Trapezoidal Map
Randomized Incremental-Construction

- $\mathscr{A}$ — an arrangement.
- Preprocess
  - For each segment in random order.
    - ⋆ Update the trapezoidal map.
    - ⋆ Insert the new trapezoid into a search structure.
  - $O(n \log n)$ time, $O(n)$ space.
- Answer query
  - Given a query point q
  - Search the trapezoid in the search structure.
  - Obtain the cell containing the trapezoid.
  - $O(\log n)$ expected time (if the segments were processed in random order).

Point Location Complexity

Requirements:

- Fast query processing.
- Reasonably fast preprocessing.
- Small space data structure.

	Naive	Walk	RIC	Landmarks	Triangulat	PST
Preprocess time	none	none	$O(n \log n)$	$O(k \log k)$	$O(n \log n)$	$O(n \log n)$
Memory space	none	none	$O(n)$	$O(k)$	$O(n)$	$O(n \log n)^{(*)}$
Query time	bad	reasonable	good	good	quite good	good
Code	simple	quite simple	complicated	quite simple	modular	complicated

Walk — Walk along a line **RIC** — Random Incremental Construction based on trapezoidal decomposition
Triangulat — Triangulation **PST** — Persistent Search Tree
k — number of landmarks
(*) Can be reduced to $O(n)$

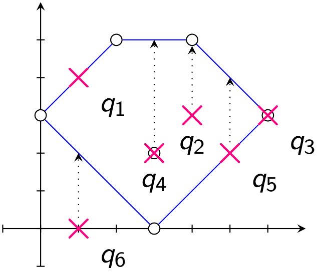
Point Location: Print

Print a polymorphic object.

```
template <typename Arrangement_>
void print_point_location(const typename Arrangement_::Point_2& q,
                          CGAL::Arr_point_location_result<Arrangement_>::Type& obj)
{
  typedef Arrangement_                                  Arrangement;
  typedef typename Arrangement::Vertex_const_handle     Vertex_const_handle;
  typedef typename Arrangement::Halfedge_const_handle   Halfedge_const_handle;
  typedef typename Arrangement::Face_const_handle       Face_const_handle;

  const Vertex_const_handle*   v;
  const Halfedge_const_handle* e;
  const Face_const_handle*     f;

  std::cout << "The point (" << q << ") is located ";
  if ((f = boost::get<Face_const_handle>(&obj)))         // located inside a face
    std::cout << "inside "
              << (((*f)->is_unbounded()) ? "the unbounded" : "a bounded")
              << " face." << std::endl;
  else if ((e = boost::get<Halfedge_const_handle>(&obj))) // located on an edge
    std::cout << "on an edge: " << (*e)->curve() << std::endl;
  else if ((v = boost::get<Vertex_const_handle>(&obj)))   // located on a vertex
    std::cout << "on " << (((*v)->is_isolated()) ? "an isolated" : "a")
              << " vertex: " << (*v)->point() << std::endl;
  else CGAL_error_msg("Invalid object.");                 // this should never happen
}
```

Point Location: Locate

```
template <typename PointLocation>
void locate_point(const PointLocation& pl,
                  const typename Point_location::Arrangement_2::Point_2& q)
{
  typedef PointLocation                                  Point_location;
  typedef typename Point_location::Arrangement_2        Arrangement_2;
  typename CGAL::Arr_point_location_result<Arrangement_2>::Type obj = pl.locate(q);

  // Print the result.
  print_point_location<Arrangement_2>(q, obj);
}
```



Point Location: Example

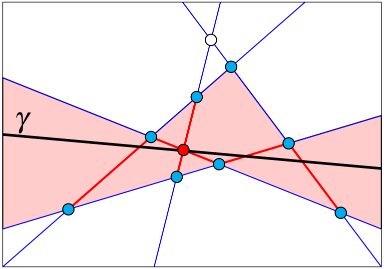
```cpp
// File: ex_point_location.cpp

#include <CGAL/basic.h>
#include <CGAL/Arr_naive_point_location.h>
#include <CGAL/Arr_landmarks_point_location.h>

#include "arr_inexact_construction_segments.h"
#include "point_location_utils.h"

typedef CGAL::Arr_naive_point_location<Arrangement_2>      Naive_pl;
typedef CGAL::Arr_landmarks_point_location<Arrangement_2>  Landmarks_pl;

int main()
{
  // Construct the arrangement.
  Arrangement_2   arr;
  construct_segments_arr(arr);

  // Perform some point-location queries using the naive strategy.
  Naive_pl        naive_pl(arr);
  locate_point(naive_pl, Point_2(1, 4));          // q1

  // Attach the landmarks object to the arrangement and perform queries.
  Landmarks_pl landmarks_pl;
  landmarks_pl.attach(arr);
  locate_point(landmarks_pl, Point_2(3, 2));      // q4

  return 0;
}
```

Outline

The Zone of Curves in Arrangements

Definition (Zone)

Given an arrangement of curves $\mathscr{A} = \mathscr{A}(\mathscr{C})$ in the plane, the zone of an additional curve $\gamma \notin \mathscr{C}$ in $\mathscr{A}$ is the union of the features of $\mathscr{A}$, whose closure is intersected by γ.



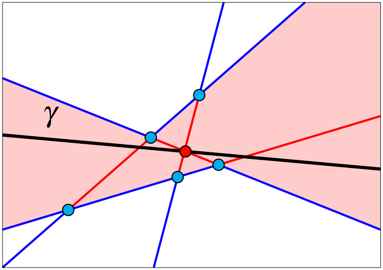
The zone of a line γ in an arrangement of lines.

The Zone of lines in an arrangement of Lines

The complexity of a zone is the total complexity of all features the zone consists of.

Theorem (Zone Complexity)

The complexity of the zone *of a line in an arrangement of n lines in the plane is $O(n)$. It can be computed in $O(n)$ time.*



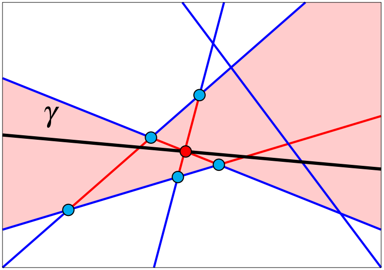
	Vertices	Edges	Faces	Total
Number	1	6	6	13
Complexity	1	17	41	53

The Zone of lines in an arrangement of Lines

The complexity of a zone is the total complexity of all features the zone consists of.

Theorem (Zone Complexity)

The complexity of the zone of a line in an arrangement of n lines in the plane is $O(n)$. It can be computed in $O(n)$ time.



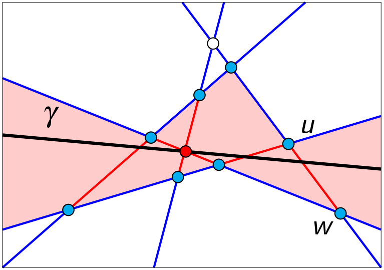
	Vertices	Edges	Faces	Total
Number	1	6	6	13
Complexity	1	17	41	53

The Zone of lines in an arrangement of Lines

The complexity of a zone is the total complexity of all features the zone consists of.

Theorem (Zone Complexity)

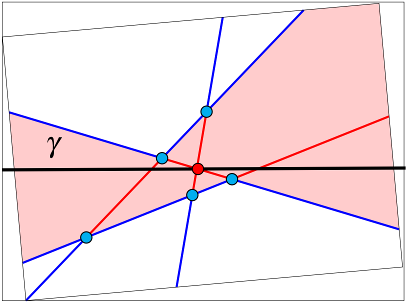
The complexity of the zone of a line in an arrangement of n lines in the plane is $O(n)$. It can be computed in $O(n)$ time.



	Vertices	Edges	Faces	Total
Number	1	7	7	15
Complexity	1	21	53	68

The Zone of lines in arrangement of Lines Complexity

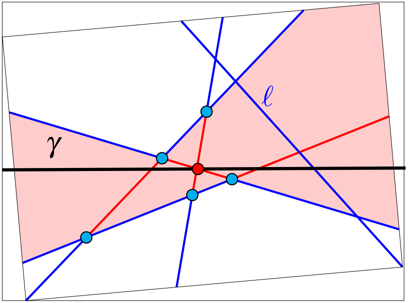
- The number of left bounding edges of the faces in the zone of γ is $\leq 3n$
- By symmetry, the number of right bounding edges is $\leq 3n$ as well
- Proof by induction on n
- ℓ is the line that has the rightmost intersection with γ
- uw is a new left bounding edge—this adds 1
- ℓ splits a left bounding edge at u and w—this adds ≤ 2



- The proof assumes general position
    - It can be extended to handle degeneracies.

The Zone of lines in arrangement of Lines Complexity

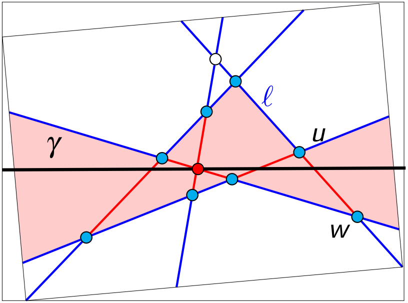
- The number of left bounding edges of the faces in the zone of γ is $\leq 3n$
- By symmetry, the number of right bounding edges is $\leq 3n$ as well
- Proof by induction on n
- ℓ is the line that has the rightmost intersection with γ
- uw is a new left bounding edge—this adds 1
- ℓ splits a left bounding edge at u and w—this adds ≤ 2



- The proof assumes general position
  - It can be extended to handle degeneracies.

The Zone of lines in arrangement of Lines Complexity

- The number of left bounding edges of the faces in the zone of γ is $\leq 3n$
- By symmetry, the number of right bounding edges is $\leq 3n$ as well
- Proof by induction on n
- ℓ is the line that has the rightmost intersection with γ
- uw is a new left bounding edge—this adds 1
- ℓ splits a left bounding edge at u and w—this adds ≤ 2



- The proof assumes general position
  - It can be extended to handle degeneracies.

Zone Application: Incremental Insertion

Definition (Incremental Insertion)

Given an x-monotone curve γ and an arrangement $\mathscr{A}$ induced by a set of curves $\mathscr{C}$, where all curves in $\{\gamma\} \cup \mathscr{C}$ are well behaved, insert γ into $\mathscr{A}$.

- Find the location of one endpoint of the curve γ in $\mathscr{A}$.
- Traverse the zone of the curve γ.
  - Each time γ crosses an existing vertex v split γ at v into subcurves.
  - Each time γ crosses an existing edge e split γ and e into subcurves, respectively.

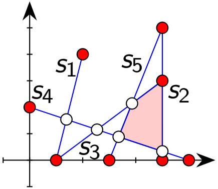
The Zone Computation Algorithmic Framework

`Arrangement_zone_2` class template

- Computes the zone of an arrangement.
- Is part of *2D Arrangements* package.
- Is parameterized with a zone visitor
  - Models the concept `ZoneVisitor_2`
- Serves as the foundation of a family of concrete operations
  - Inserting a single curve into an arrangement
    - ⋆ The visitor modifies the arrangement operand as the computation progresses.
  - Determining whether a query curve intersects with the curves of an arrangement.
  - Determining whether a query curve passes through an existing arrangement vertex.
    - ⋆ If the answer is positive, the process can terminate as soon as the vertex is located.

Incremental Insertion

```
// File: ex_incremental_insertion.cpp

#include <CGAL/basic.h>
#include <CGAL/Arr_naive_point_location.h>

#include "arr_exact_construction_segments.h"
#include "arr_print.h"

int main()
{
  // Construct the arrangement of five line segments.
  Arrangement_2 arr;
  Naive_pl pl(arr);
  CGAL::insert_non_intersecting_curve(arr, Segment_2(Point_2(1, 0), Point_2(2, 4)), pl);
  CGAL::insert_non_intersecting_curve(arr, Segment_2(Point_2(5, 0), Point_2(5, 5)));
  CGAL::insert(arr, Segment_2(Point_2(1, 0), Point_2(5, 3)), pl);
  CGAL::insert(arr, Segment_2(Point_2(0, 2), Point_2(6, 0)));
  CGAL::insert(arr, Segment_2(Point_2(3, 0), Point_2(5, 5)), pl);
  print_arrangement_size(arr);
  return 0;
}
```

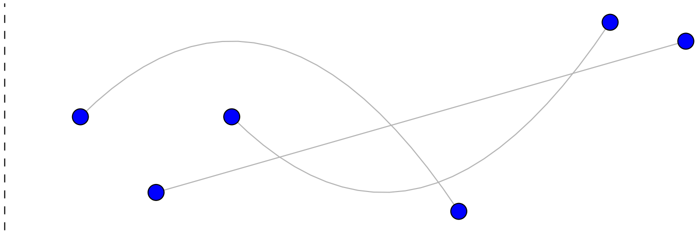


Outline

The Plane Sweep Algorithmic Framework

[BO79]

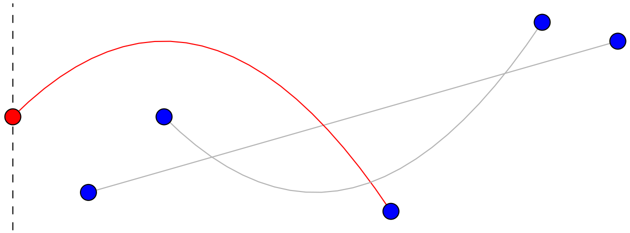
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

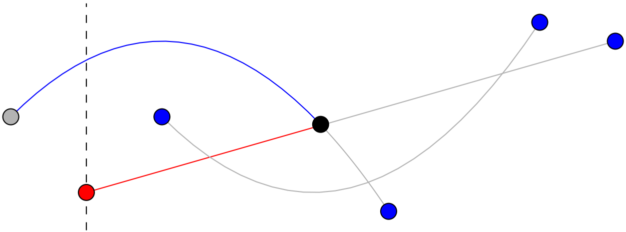
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

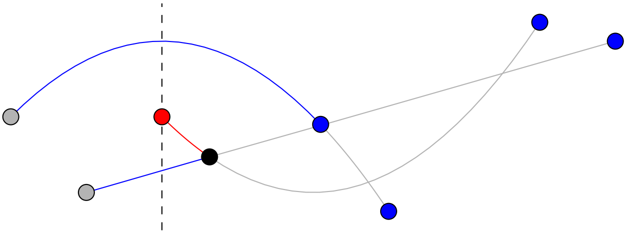
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

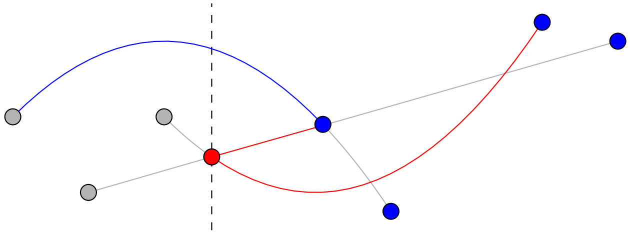
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

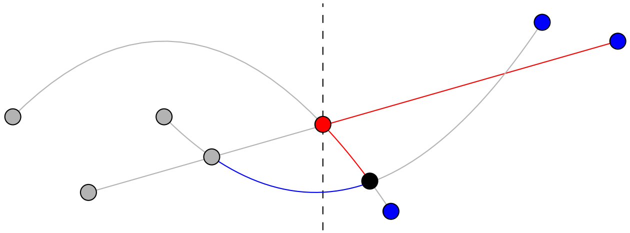
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

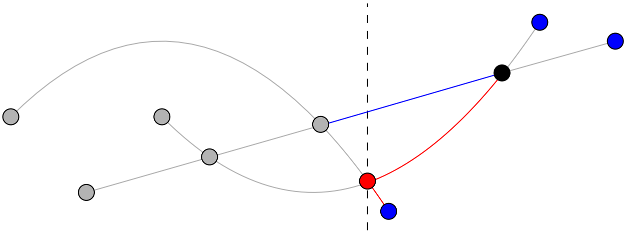
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

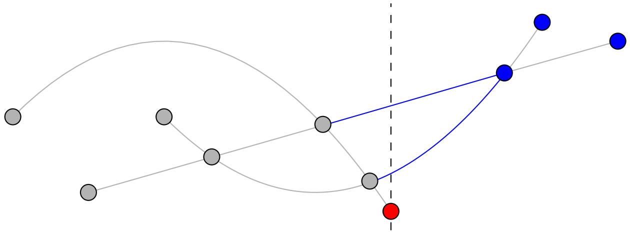
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

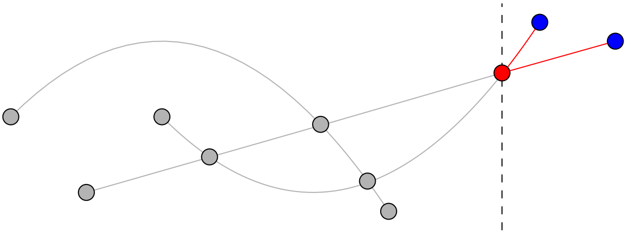
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

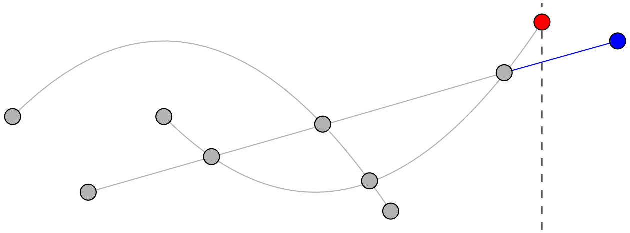
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

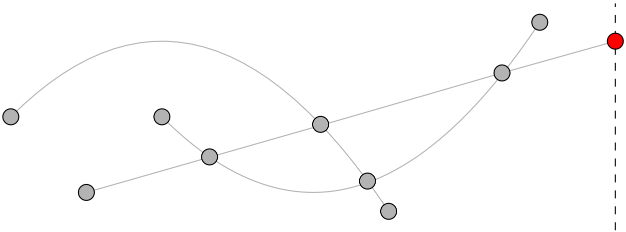
- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue



The Plane Sweep Algorithmic Framework

[BO79]

- Initialize an event queue with all endpoints sorted lexicographically
- While the queue is not empty, extract and process an event
  - Remove all x-monotone curves to the left of the current event point from a sorted container of curves
  - Insert all x-monotone curves to the right of the current event point into the curve container
  - Compute intersections between existing curves and newly inserted curves, and insert them into the event queue

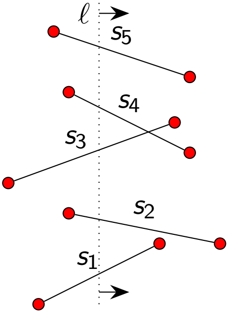


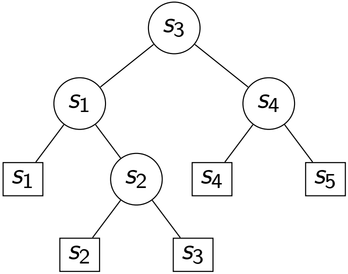
Plane Sweep: Event Queue

- Implemented as a balanced binary search tree (say red-black tree)
- Operations, m—number of events.
  - Fetching the next event—$O(\log m)$ amortized time.
  - Testing whether an event exists—($O(\log m)$ amortized time.
    - ⋆ Cannot use a heap!
  - Inserting an event—$O(\log m)$ amortized time.

Plane Sweep: Status Structure

- Is a dynamic one-dimensional arrangement along the sweep line.
- Implemented as a balanced binary search tree
  - Interior nodes —- guide the search, store the segment from the rightmost leaf in its left subtree.
  - Leaf nodes — segments.
- Operations—$O(\log n)$ amortized time.



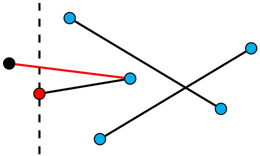


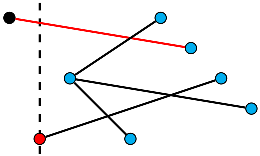
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



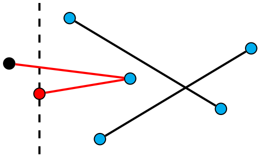


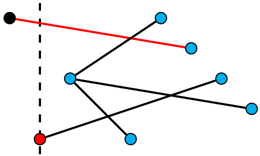
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



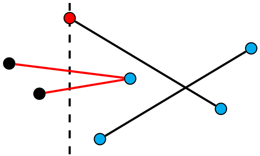


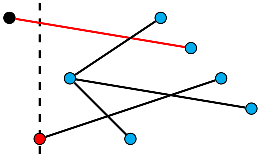
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



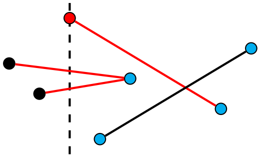


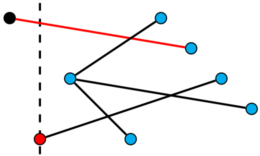
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



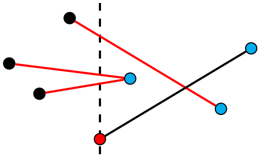


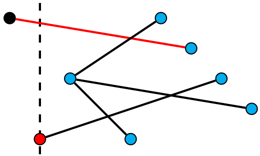
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



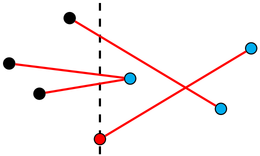


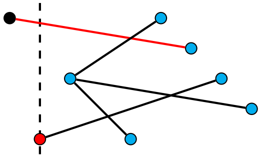
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



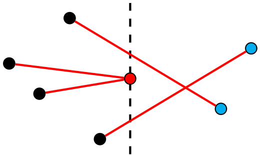


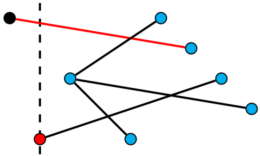
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



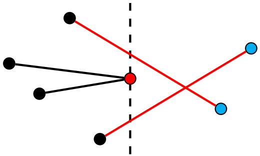


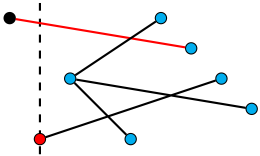
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



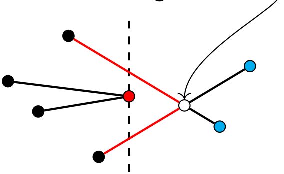


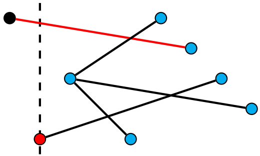
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



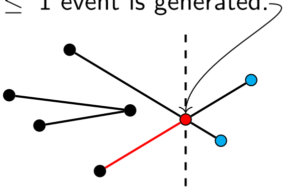


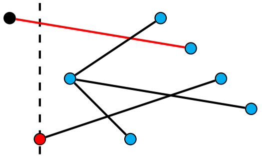
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



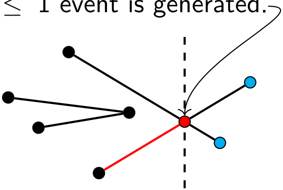


Plane Sweep Complexity

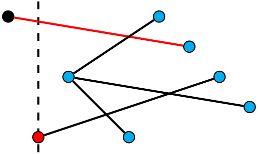
Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.



  - If p has right curves $\leq$ 2 events are generated.

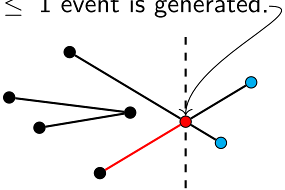


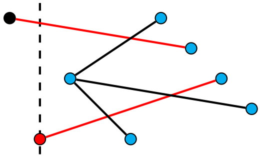
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



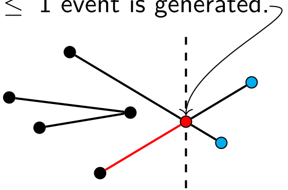


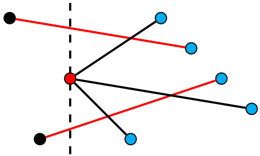
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



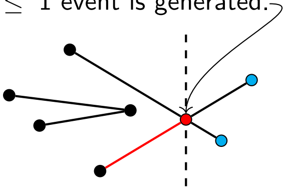


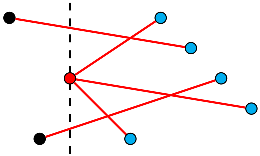
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



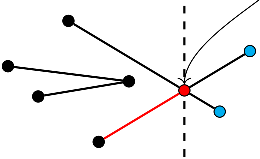


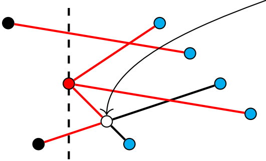
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



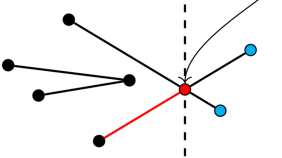


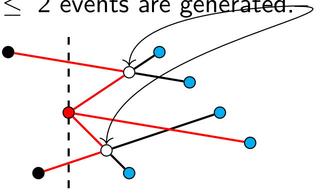
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



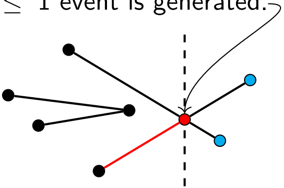


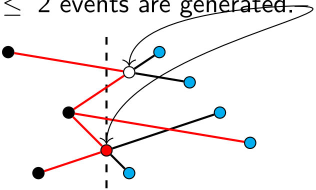
Plane Sweep Complexity

Theorem

All points of intersection between the curves in $\mathscr{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.

- $\mathscr{C}$—a set of n x-monotone curves in the plane.
- k—the number of intersection points.
- Constructing the event queue takes $O(n \log n)$ time.
- p—an event
  - p is fetched and removed from the event queue.
  - p is handled once.
  - If p does not have right curves $\leq$ 1 event is generated.
  - If p has right curves $\leq$ 2 events are generated.



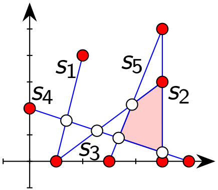


Plane Sweep Space Complexity

- The status-structure size is in $O(n)$
- The event-queue size is definitely at most $2n+k$
- It can be shown that the event-queue size is in $O(n \log^2 n)$
- The event-queue size can be kept linear.
  - Points of intersections between pairs of curves that are not adjacent on the sweep line are deleted from the event queue.
  - It increases the time complexity but only by a constant factor

Aggregate Insertion

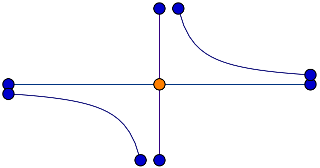
```
// File: ex_aggregated_insertion.cpp

#include "arr_exact_construction_segments.h"
#include "arr_print.h"

int main()
{
  // Aggregately construct the arrangement of five line segments.
  Segment_2 segments[] = {Segment_2(Point_2(1, 0), Point_2(2, 4)),
                          Segment_2(Point_2(5, 0), Point_2(5, 5)),
                          Segment_2(Point_2(1, 0), Point_2(5, 3)),
                          Segment_2(Point_2(0, 2), Point_2(6, 0)),
                          Segment_2(Point_2(3, 0), Point_2(5, 5))};
  Arrangement_2 arr;
  CGAL::insert(arr, segments, segments + sizeof(segments)/sizeof(Segment_2));
  print_arrangement_size(arr);
  return 0;
}
```



Outline

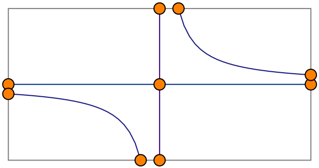
Handling Enpoints at Infinity

Clipping the unbounded curves



Using an infimaximal box

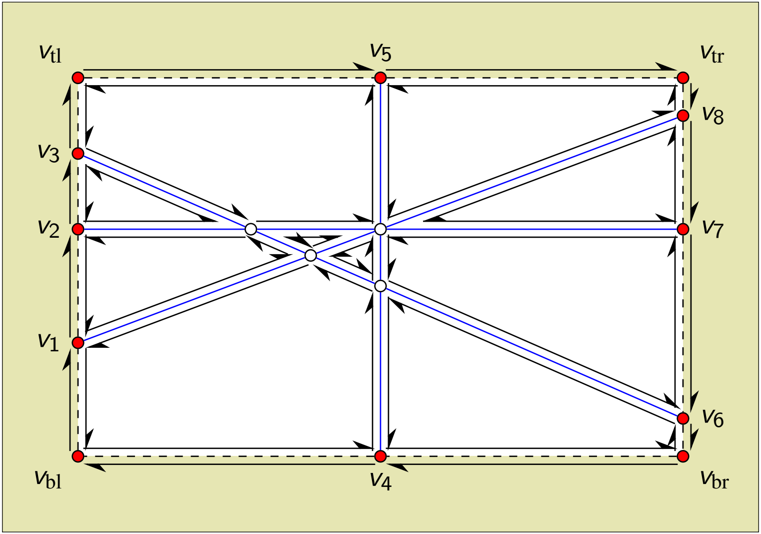
[Mehlhorn & Seel, 2003]



- Simple, the sweep algorithm is unchanged
- Not online
- The resulting arrangement has a single unbounded face

- Not simple
  - May require large bit-lengths
  - Designed for linear objects
- Online (no need for clipping)
- The resulting arrangement has multiple unbounded faces (and a single ficticious face)

Arrangement of (Unbounded) Lines



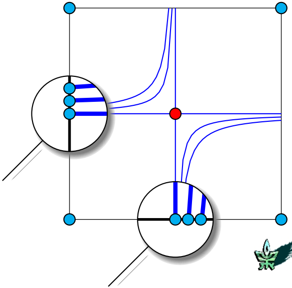
Vertices of Unbounded Arrangement

There are 4 types of unbounded-arrangement vertices

1. A "normal" vertex associated with a point in $\mathbb{R}^2$.
2. A vertex that represents an unbounded end of an x-monotone curve that approaches $x = -\infty$ or $x = \infty$.
3. A vertex that represents the unbounded end of a vertical line or ray or of a curve with a vertical asymptote (finite x-coordinate and an unbounded y-coordinate).
4. A fictitious vertices that represents one of 4 corners of the imaginary bounding rectangle.

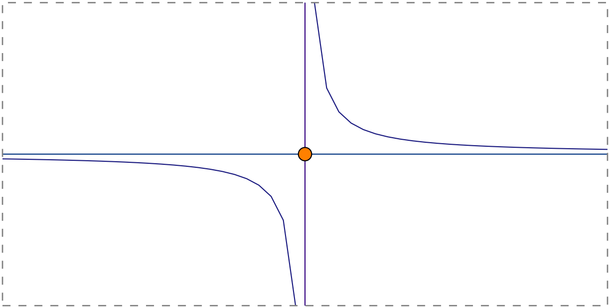
A vertex at infinity of Type 2 or Type 3 always has three incident edges:

- 1 edge associated with an x-monotone curve, and
- 2 fictitious edges connecting the vertex to its adjacent vertices at infinity or the corners of the bounding rectangle.



Sweeping Unbounded Curves

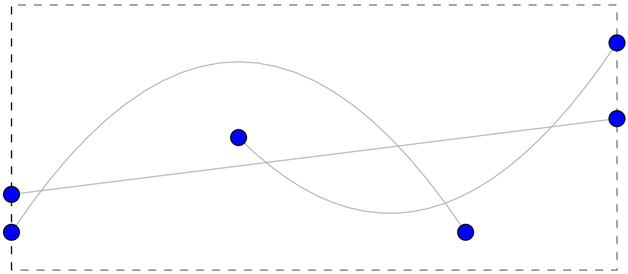
- Curves may not have finite endpoints
  - Initializing the event queue requires special treatment
- Intersection events are associated with finite points



$xy = 1$, $x = 0$, and $y = 0$

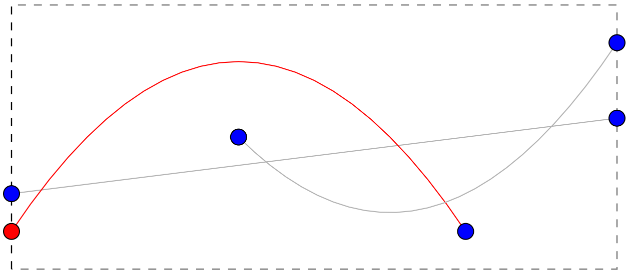
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



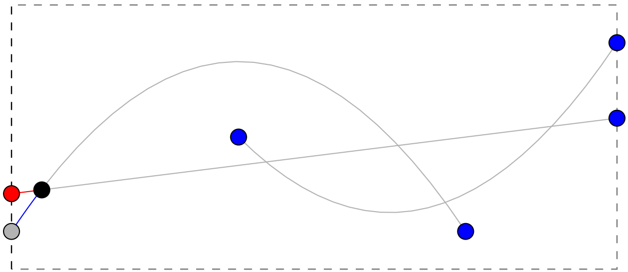
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



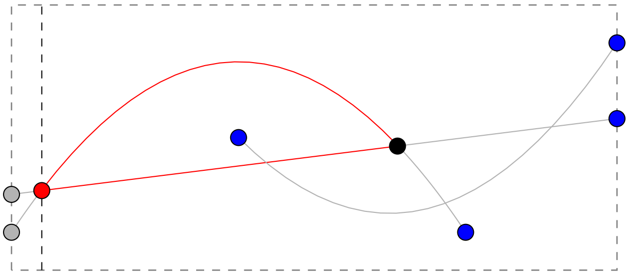
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



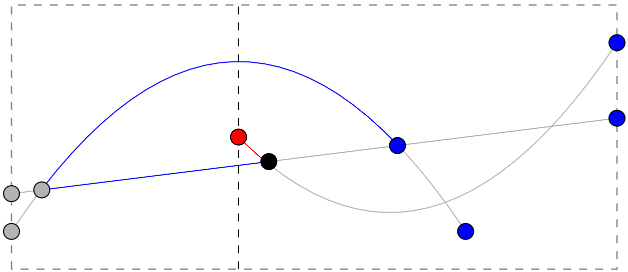
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



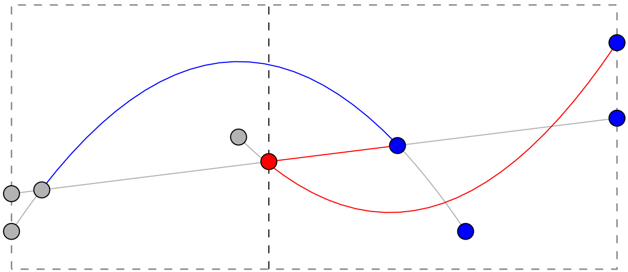
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



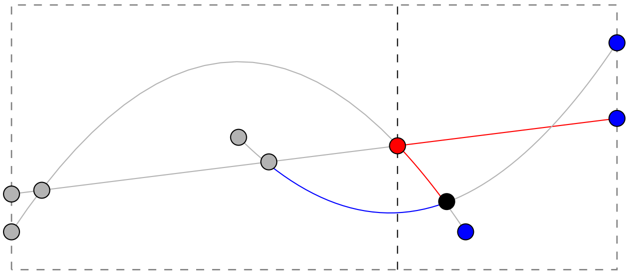
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



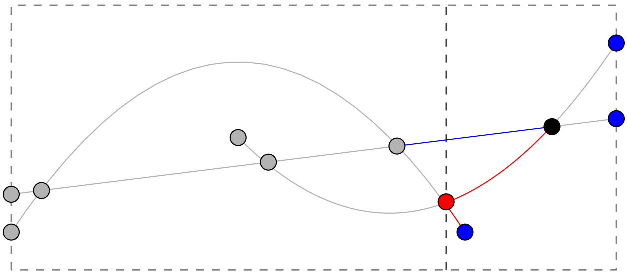
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



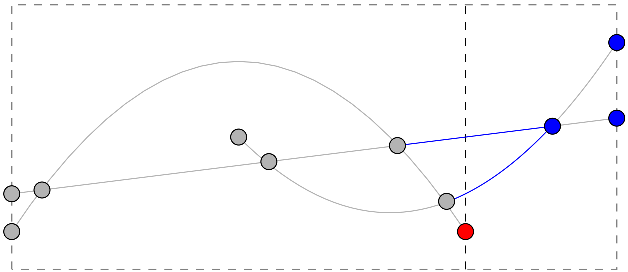
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



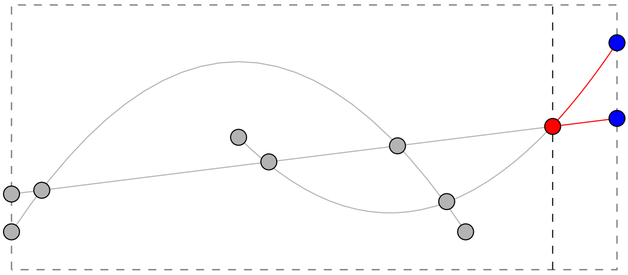
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



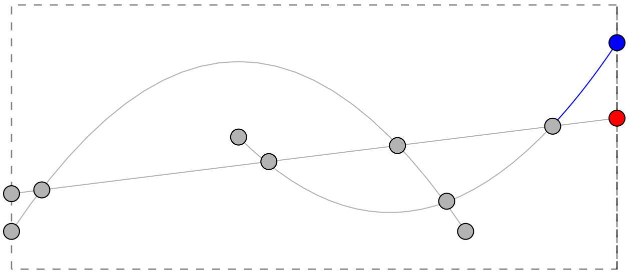
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



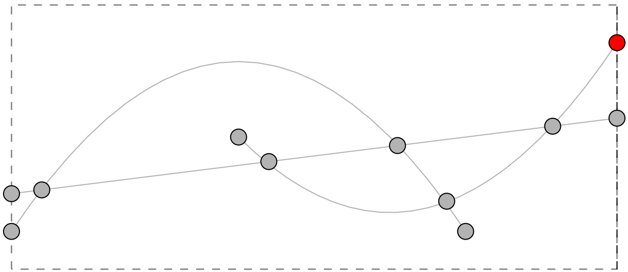
The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



The Augmented Sweep Line for Unbounded Curves

- Categorize all curve ends
- Initialize an event queue with all curve ends sorted lex.
  - Ends of unbounded curves do not coincide
  - Comparison between events are available through the traits
- While the queue is not empty proceed as usual
  - No need to look for unbounded events in the status line!



Outline

1. 2D Arrangements
   - Definitions & Complexity
   - Representation
   - Queries
     - Vertical Decomposition
     - Point Location Queries
   - The Zone Computation Algorithmic Framework
   - The Plane Sweep Algorithmic Framework
   - Arrangement of Unbounded Curves
   - Literature

Arrangement Bibliography I

- Boris Aronov and Dmitriy Drusvyatskiy
  Complexity of a Single Face in an Arrangement of s-Intersecting Curves
  arXiv:1108.4336, 2011
- Jon Louis Bentley and Thomas Ottmann.
  Algorithms for Reporting and Counting Geometric Intersections.
  IEEE Transactions on Computers, 28(9): 643–647, 1979.
- Eric Berberich, Efi Fogel, Dan Halperin, Michael Kerber, and Ophir Setter.
  Arrangements on parametric surfaces ii: Concretizations and applications, 2009.
  Mathematics in Computer Science, 4(1):67–91,2010.
- Ulrich Finke and Klaus H. Hinrichs.
  Overlaying simply connected planar subdivisions in linear time.
  In *Proceedings of 11^{th} Annual ACM Symposium on Computational Geometry (SoCG)*, pages 119–126. Association for Computing Machinery (ACM) Press, 1995.
- Ron Wein, Efi Fogel, Baruch Zukerman, Dan Halperin, and Eric Berberich.
  2D Arrangements.
  In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 4.4 edition, 2014.
  http://www.cgal.org/Manual/latest/doc_html/cgal_manual/packages.html#Pkg:Arrangement2.
- David G. Kirkpatrick.
  Optimal search in planar subdivisions.
  SIAM Journal on Computing. 12(1):28–35,1983.
- N. Sarnak and Robert E. Tarjan.
  Planar point location using persistent search trees.
  Communications of the ACM. 29(7):669–679, 1986.
- Kentan Mulmuley.
  A fast planar partition algorithm, I.
  Journal of Symbolic Computation. 10(3-4):253–280,1990.

Arrangement Bibliography II

Raimund Seidel
A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons.
Computational Geometry: Theory and Applications. 1(1):51–64, 1991.

Olivier Devillers, Sylvain Pion, and Monique Teillaud.
Walking in a triangulation.
International Journal of Foundations of Computer Science. 13:181–199,2002.

Luc Devroye Christophe, Christophe Lemaire, and Jean-Michel Moreau.
Fast Delaunay Point-Location with Search Structures.
In *Proceedings of* 11^{th} *Canadian Conference on Computational Geometry*. Pages 136–141, 1999.

Luc Devroye, Ernst Peter Mücke, and Binhai Zhu.
A Note on Point Location in Delaunay Triangulations of Random Points.
Algorithmica. 22:477–482, 1998.

Olivier Devillers.
The Delaunay hierarchy.
International Journal of Foundations of Computer Science. 13:163-180, 2002.

Sunil Arya
A Simple Entropy-Based Algorithm for Planar Point Location.
ACM Transactions on Graphics. 3(2), 2007

Masato Edahiro, Iwao Kokubo, And Takao Asano
A new Point-Location Algorithm and its Practical Efficiency: comparison with existing algorithms
ACM Transactions on Graphics. 3(2):86–109, 1984.

Micha Sharir and Pankaj Kumar Agarwal
Davenport-Schinzel Sequences and Their Geometric Applications.
Cambridge University Press, New York, NY, 1995.

Arrangement Bibliography III

Bernard Chazelle, Leonidas J. Guibas, and Der-Tsai Le.
The Power of Geometric Duality.
BIT, 25:76–90, 1985.

Herbert Edelsbrunner,
Algorithms in Combinatorial Geometry,
Springer, Heidelberg, 1987.

Mark de Berg, Mark van Kreveld, Mark H. Overmars, and Otfried Cheong.
Computational Geometry: Algorithms and Applications.
Springer, 3^{rd} edition, 2008.

Herbert Edelsbrunner, Raimund Seidel, and Micha Sharir.
On the Zone Theorem for Hyperplane Arrangements.
SIAM Journal on Computing. 22(2):418–429,1993.

Silvio Micali and Vijay V. Vazirani.
An $O(\sqrt{|V|}|E|)$ Algorithm for Finding Maximum Matching in General Graphs.
Proceedings of 21^{st} *Annual IEEE Symposium on the Foundations of Computer Science*, pages 17–27, 1980.

Jack Edmonds.
Paths, Trees, and Flowers.
Canadian Journal of Mathematics, 17:449–467,1965.

Robert Endre Tarjan.
Data structures and network algorithms, Society for Industrial and Applied Mathematics (SIAM), 1983.

Marcin Mucha and Piotr Sankowski.
Maximum Matchings via Gaussian Elimination
Proceedings of 45^{th} *Annual IEEE Symposium on the Foundations of Computer Science*, pages 248–255, 2004.

Arrangement Bibliography IV

Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine.
The BOOST *Graph Library*.
Addison-Wesley, 2002

Efi Fogel, Ron Wein, and Dan Halperin.
CGAL *Arrangements and Their Applications, A Step-by-Step Guide*.
Springer, 2012.