Lecture 5 (CGAL)

Panos Giannopoulos, Dror Atariah
AG TI

WS 2012/13
- Project chosen?
- Exercise 4? (See also the Voronoi package and its adaptor)
Outline

Note: The CGAL-Qt ‘interface’

Arrangements (2d)

Exercise 5
CGAL-Qt ‘interface’

- Qt-like graphic items for points, polygons, etc. (I.e., Q_Objects sending/receiving signals)

Example
Two Dimensional Arrangements

Definition (Arrangement)

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

- 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 $C$ are well behaved, if each pair of curves in $C$ 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 \( \Theta(n^d) \).
Constructs, maintains, modifies, traverses, queries, and presents arrangements on two-dimensional parametric surfaces.

Robust and exact
- All inputs are handled correctly (including degenerate input).
- Exact number types are used to achieve exact results.

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
- overlay computation

Part of the CGAL basic library
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 halfedge 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 boundary.

- Connected component of the boundary ($C_{CB}$)
  - The circular chains of halfedges around faces.
Arrangement Representation

- The halfedges incident to a vertex form a circular list.
- The halfedges are sorted in clockwise order around the vertex.
- The halfedges around faces form circular chains.
- All halfedges of a chain are incident to the same face.
- The halfedges are sorted in counterclockwise order along the boundary.
- Geometric interpretation is added by classes built on top of the halfedge data-structure.
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>.
Traversing an Arrangement Vertex

Print all the halfedges incident to a vertex.

```cpp
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 an Arrangement (Half)edge

Print all $x$-monotone curves along a given CCB

```cpp
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;
}
```

- $he\rightarrow\text{curve}()$ is equivalent to $he\rightarrow\text{twin()}\rightarrow\text{curve}()$,
- $he\rightarrow\text{source}()$ is equivalent to $he\rightarrow\text{twin()}\rightarrow\text{target}()$, and
- $he\rightarrow\text{target}()$ is equivalent to $he\rightarrow\text{twin()}\rightarrow\text{source}()$. 
Traversing an Arrangement Face

Print the outer and inner boundaries of a face.

```cpp
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.
    int 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.
    typedef 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.
    typedef 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.
    typedef 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);
}
Modifying the Arrangement

Inserting a curve that induces a new hole inside the face $f$, $\text{arr.insert_in_face_interior}(c, f)$.

Inserting a curve from an existing vertex $u$ that corresponds to one of its endpoints, $\text{insert_from_left_vertex}(c, v)$, $\text{insert_from_right_vertex}(c, v)$.

Inserting an $x$-monotone curve, the endpoints of which correspond to existing vertices $v_1$ and $v_2$, $\text{insert_at_vertices}(c, v_1, v_2)$.

- 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.
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 closes.
- Walk along line
  - Walk along a vertical line from infinity.
- Trapezoidal map Randomized Incremental-Construction (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.
Landmark Point Location

- Given an arrangement $\mathcal{A}$
**Landmark Point Location**

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

![Diagram showing an arrangement with landmarks](image)
Landmark Point Location

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

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

- Given an arrangement \( A \)
- Preprocess
  - Choose the landmarks and locate them in \( A \).
  - Store the landmarks in a nearest neighbor search-structure.
- Answer query
  - Given a query point \( q \)
  - Find the landmark \( \ell \) closest to \( q \) using the search structure.
  - The landmarks are on a grid \( \implies \) Nearest grid point found in \( O(1) \) time.
Landmark Point Location

- Given an arrangement \( \mathcal{A} \)
- Preprocess
  - Choose the landmarks and locate them in \( \mathcal{A} \).
  - Store the landmarks in a nearest neighbor search-structure.
- Answer query
  - Given a query point \( q \)
  - Find the landmark \( \ell \) closest to \( q \) using the search structure.
    - The landmarks are on a grid \( \Rightarrow \) Nearest grid point found in \( O(1) \) time.
  - “Walk along a line” from \( \ell \) to \( q \).
Trapezoidal Map
Randomized Incremental-Construction

\[ \mathcal{A} \] — an arrangement.
Trapezoidal Map
Randomized Incremental-Construction

\[ \mathcal{A} \] — an arrangement.

Preprocess
\[ \text{For each segment in random order.} \]
Trapezoidal Map
Randomized Incremental-Construction

- $\mathcal{A}$ — an arrangement.
- Preprocess
  - For each segment in random order.
    - Update the trapezoidal map.
Trapezoidal Map
Randomized Incremental-Construction

- $\mathcal{A}$ — 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.
Arrangements ([E. Fogel])

Trapezoidal Map
Randomized Incremental-Construction

- $\mathcal{A}$ — 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 $q$
Arrangements

Trapezoidal Map
Randomized Incremental-Construction

- $\mathcal{A}$ — 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.
Arrangements

Trapezoidal Map
Randomized Incremental-Construction

- \( \mathcal{A} \) — 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.

<table>
<thead>
<tr>
<th></th>
<th>Naive</th>
<th>Walk</th>
<th>RIC</th>
<th>Landmarks</th>
<th>Triangulat</th>
<th>PST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocess time</td>
<td>none</td>
<td>none</td>
<td>$O(n \log n)$</td>
<td>$O(k \log k)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Memory space</td>
<td>none</td>
<td>none</td>
<td>$O(n)$</td>
<td>$O(k)$</td>
<td>$O(n)$</td>
<td>$O(n \log n)^{*}$</td>
</tr>
<tr>
<td>Query time</td>
<td>bad</td>
<td>reasonable</td>
<td>good</td>
<td>good</td>
<td>quite good</td>
<td>good</td>
</tr>
<tr>
<td>Code</td>
<td>simple</td>
<td>quite simple</td>
<td>complicated</td>
<td>quite simple</td>
<td>modular</td>
<td>complicated</td>
</tr>
</tbody>
</table>

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.

```cpp
template <typename Arrangement>
void print_point_location(const typename Arrangement::Point_2& q,
const CGAL::Object& obj)
{
    typename Arrangement::Vertex_const_handle v;
    typename Arrangement::Halfedge_const_handle e;
    typename Arrangement::Face_const_handle f;

    std::cout << "The point (" << q << ") is located \";
    if (CGAL::assign(f, obj)) {
        // q is located inside a face
        if (f->is_unbounded())
            std::cout << "inside the unbounded face." << std::endl;
        else std::cout << "inside a bounded face." << std::endl;
    }
    else if (CGAL::assign(e, obj)) {
        // q is located on an edge
        std::cout << "on an edge: \" " << e->curve() << std::endl;
    }
    else if (CGAL::assign(v, obj)) {
        // q is located on a vertex
        if (v->is_isolated())
            std::cout << "on an isolated vertex: \" " << v->point() << std::endl;
        else std::cout << "on a vertex: \" " << v->point() << std::endl;
    }
    else CGAL_error_msg("Invalid object!"); // this should never happen
}
```
Point Location: Locate

```cpp
template <typename Point_location>
void locate_point(const Point_location& pl,
                   const typename Point_location::Arrangement_2::Point_2& q)
{
    CGAL::Object obj = pl.locate(q);  // perform the point-location query
    // Print the result.
    print_point_location<typename Point_location::Arrangement_2>(q, obj);
}
```
Point Location: Example

```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;

    // Perform some point-location queries using the naive strategy.
    Naive_pl naive_pl(arr);
    construct_segments_arr(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;
}
```
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);
  insert_non_intersecting_curve(arr, Segment_2(Point_2(1, 0), Point_2(2, 4)), pl);
  insert_non_intersecting_curve(arr, Segment_2(Point_2(5, 0), Point_2(5, 5)));
  insert(arr, Segment_2(Point_2(1, 0), Point_2(5, 3)), pl);
  insert(arr, Segment_2(Point_2(0, 2), Point_2(6, 0)));
  insert(arr, Segment_2(Point_2(3, 0), Point_2(5, 5)), pl);
  print_arrangement_size(arr);
  return 0;
}
```
Exercise 5

1. Read Ch. 20.1 - 20.4 of CGAL’s manual.
2. Read Ch. 2.1 - 2-3 of de Berg et al.
3. Construct an arrangement of segments, print the number of vertices, halfedges, and faces.
4. Draw the arrangement in Qt. Do that efficiently, i.e., no points, edges should be drawn twice.
5. Write a function template that accepts a face (not necessarily convex) of an arrangement induced by line segments, and returns a point located inside the face. Use two template parameters, one for the arrangement, one for the kernel, with the following preconditions: (i) The traits class used supports line segments, (ii) the types Arrangement::Traits_2::Point_2 and Kernel::Point_2 are convertible to one another, and (iii) the Kernel has a nested functor called Construct_midpoint_2, the functor of which accepts two points p, q and returns the midpoint of the segment pq.
6. Perform point location queries on an arrangement using the different point location methods.