

---

# APPLIED aspects of COMPUTATIONAL GEOMETRY

A Gentle Introduction to 

---

Eric Berberich  
School of Computer Science  
Tel Aviv University

---

# Convex Hull

- Input: set of points  $P$  (or objects)
- Output: the convex hull, i.e., smallest convex set  $S$  with  $P$  being subset of  $S$
- Now: Demo

# Convex Hull in CGAL-C++

```
#include<CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/convex_hull_2.h>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;

int main() {
    std::vector< Point_2 > in, out;
    in.push_back(Point_2(0, 0)); in.push_back(Point_2(2, 4));
    in.push_back(Point_2(1, 3)); in.push_back(Point_2(-3, 10));
    in.push_back(Point_2(-10, -23)); in.push_back(Point_2(5, -2));

    CGAL::convex_hull_2(in.begin(), in.end(), std::back_inserter(out));
    return 0;
}
```

# Lesson overview

- |                             |             |
|-----------------------------|-------------|
| ■ Example 😊                 | Schedule:   |
| ■ CGAL                      | 16:10-17:00 |
| □ Overview                  | 17:10-18:00 |
| □ Generic Programming       | 18:10-19:00 |
| □ More simple examples      |             |
| ■ Three showcases:          |             |
| □ Convex Hull - reloaded    |             |
| □ (Delaunay) Triangulation  |             |
| □ Arrangement               |             |
| ■ CGAL-Setup + Installation |             |

---

# CGAL – Goals & Ingredients

- robust geometric computing
  - Robust (correctness, degeneracies)
  - Efficient (nevertheless: reasonable fast)
  - Ease of use (for users)
  - Homogeneity
- Implementations of geometric
  - **Objects + Predicates + Constructions**, **Kernels**
  - **Algorithms + Data structures**

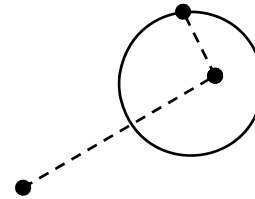
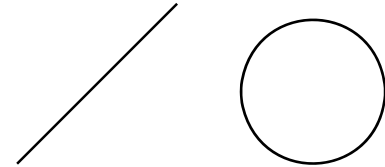
---

# History + Facts

- Started in 1995. CGAL 1.0 in 1997
  - Following the generic programming paradigm
  - Consortium of research institutes (TAU, MPI, Inria, ETH,...) + Geometry Factory
  - ~20-30 active developers
  - Release every 6 months: Newest v3.4
  - Licenses: Open source + commercial (if code should be hidden)
  - Editorial Board reviews new software
-

# CGAL 1-2-3

- Geometric **Objects**, e.g.,
  - Points, Lines, Segments, Circles
- Geometric **Predicates + Constructions**, e.g.,
  - Orientation of three points
  - Point in circle
  - Intersections of segments + circle



---

# CGAL 1-2-3

- **Objects + Predicates** = (**Kernel**) Link
  - 2D, 3D, dD
  - Exact, Filtered
  - Cartesian or homogeneous coordinates
  - Reference counting (actual rep of objects stored only once, access by light-weight handles)



# CGAL 1-2-3 (Alg + DS)

- Combinatorial algorithm & data structures [Link](#)
  - Convex Hull, Triangulations, Arrangement, Voronoi, Meshing, Optimization, Kinetic Data structures
  - **Execution path/status** based on **evaluation** of **geometric predicates and constructions on geometric objects**
  - Algorithm/structure expects a certain set of types, operations: it defines **concept**  
(more in part on generic programming)

---

# CGAL 1-2-3 (Models)

- Instantiation with a **model** defines behavior:
  - Arrangement of **segments, circles, function graphs**
  - We usually refer to a **traits class** for such a model
  - Often: **Kernel** can already serve as parameter

# CGAL ... and 4

- Support library
  - STL extensions, Circulators, Generators
  - Adapters, e.g., Boost graph
  - Sorting + Linear/quadratic programming
- “Math” - for predicates (and constructions)
  - Algebraic foundations
  - **Number types** + Arithmetic
  - Polynomials
- IO + Visualization support

# Number types

- Build-in: int, double, ...
  - Fast, but inexact
- CGAL:
  - “Exact”: Quotient, MP\_Float, Root\_of\_2
  - Lazy\_exact\_nt<NT> (tries an approximation, first)
- Boost:
  - interval
- GMP:
  - Gmpz, Gmpq
- LEDA & Core:
  - Integer, Rational, “Reals”
- Possible to provide own number types

---

# Rationale: Correctness

- Design to deal with all cases
- Robustness issues
  - Exact evaluation (and maybe construction)
    - Sign of expression (complicated if close to 0)
    - Rounding problems (esp. for real numbers, as sqrt)
  - Handling of all combinatorial degeneracies
    - Three points on a line
    - Several curves running through the same point

---

# Rationale: Flexibility

- Rely on other libraries
- Modular: Separation between
  - Geometry
  - Topology / Combinatorics
- Possibility to provide own (geometric) types and operations on them
- Data structures and algorithms are extendible
  - own sweep line based algorithm on set of curves

---

# Rationale: Ease of use

- Manuals
  - Examples
  - Demos
  - Standard-Design: C++, STL, Boost
  - Smooth learning curve
- 
- Nemo would say: Templates are your friends

---

# Rationale: Efficiency

- Implements state-of-the art algorithms taken from within academia
- Efficient geometric objects and operations
- Filtering
  - Compute first approximate version
  - If not sufficient: Exact version
- Polymorphism resolved at compile-time
  - no virtual function table overhead
- Select best option (due to flexibility)



---

# Generic Programming

- Generic implementations consists of 2 parts:
  - Instructions that determine control-flow or updates
  - Set of requirements that determine the properties the algorithm's arguments/objects must satisfy
    - We call such a set a **concept**
  - It is abstract, i.e., not working without being instantiated by a **model** that fulfills the concept

# Generic Programming



- Example: Car with empty engine-bay
  - Supposed to drive, **if one mounts an engine**
  - Different models available
    - Diesel
    - Gas engine
    - Electrical engine
    - Your own engine ... as long as it “fits”:
  - Interface:
    - drive-axis
    - Mount-points
    - ... and some more



# A C++ example

- Swap:

```
template <class T>
void swap(T& a, T& b) {
    T tmp = a; a = b; b = tmp;
}
```

- Argument: type **T** which must be
  - default constructible
  - assignable
- `int a = 2, b = 4; std::swap(a, b);`

# Two other C++ examples

## ■ Vector + Sort

```
std::vector< int > v = {3, 4, 2, 1, 5};  
std::sort(v.begin(), v.end());  
int i = v[2]; // = 2
```

```
double w[4] = {8.4, 2.1, 4.2, 4.5, 1.1};  
std::sort(w, w+4);  
double d = w[3]; // 4.2
```

## ■ `std::vector<T>` is a container to store objects of type T

- is a model of Container concept
  - Provides random access iterator (`.begin()`, `w+4`)
  - Provides operator `[]`

## ■ Sort expects arguments

- to be random access iterator
- The iterator's value-type is `LessThanComparable`

# Sorting again

- Sort with another “Less”

```
template< class NT >
class MyLess {
    bool operator()(NT &a, NT &b) {
        return a > b;
    }
}
```

```
std::vector< int > v = {3, 4, 2, 1, 5};
std::sort(v.begin(), v.end(), MyLess<int>());
int i = v[2]; // = 4
```

- Simpler:

```
std::sort(v.begin(), v.end(), std::greater<int>());
```

# Generic Programming

- GP is widespread:
  - STL, Boost, STXXL, CGAL
- Terms to remember:
  - Model + Concept, Refinement
  - Class + Function Template + Template parameter
  - Traits (I'll explained it below)
- STL-Examples of generic algorithms & data structures:
  - Iterators, Adapters (insert)
  - copy, search, reverse, unique, random\_shuffle, ...
  - list, set, queue, ...
  - see <http://www.sgi.com/tech/stl/>

---

# Geometric Programming

- Generic Programming
- Exact Geometric Computing Paradigm (by Yap)
  - All predicates asked by a combinatorial algorithm compute the correct answer
  
- Example:  
CGAL: `convex_hull_2`(i n.begin(), i n.end(), std::back\_inserter(out));
  
- More examples in this lecture – and now

# Example: **Kernels**<NumberType>

## ■ Cartesian< FieldNumberType >

- `typedef CGAL::Cartesian< gmpq > K;`
- `typedef CGAL::Simple_cartesian< double > K;`  
`// no reference-counting, inexact instantiation`

## ■ Homogeneous< RingNumberType >

- `typedef CGAL::Homogeneous< Core::BigInt > K;`

## ■ d-dimensional Cartesian\_d and Homogeneous\_d

## ■ Types + Operations

- `K::Point_2, K::Segment_3`
- `K::Less_xy_2, K::Construct_bisector_3`



# Predefined **Kernels**

- 3 pre-defined Cartesian Kernels
  - construction of points from double Cartesian coordinates.
  - exact geometric predicates.
  - They handle geometric constructions differently:
    - *Exact\_predicates\_exact\_constructions\_kernel*
    - *Exact\_predicates\_exact\_constructions\_kernel\_with\_sqrt*  
its number type supports the square root operation exactly
    - *Exact\_predicates\_inexact\_constructions\_kernel*  
geometric constructions may be inexact due to round-off errors.  
It is however enough for most CGAL algorithms, and faster

---

# Special **Kernels**

- Filtered kernels
- Circular\_kernel\_2
- Circular\_kernel\_3
  
- Refer to CGAL's manual for more details

# Example: Orientation of points

- ```
#include <CGAL/MP_Float.h>
#include <CGAL/Homogeneous.h>
typedef CGAL::Homogeneous<CGAL::MP_Float> Kernel;
typedef Kernel::Point_2 Point_2;
typedef Kernel::Orientation_2 Orientation_2;

int main() {
    Kernel kernel;

    // option 1:
    Orientation_2 orientation =
        kernel.orientation_2_object();
    Point_2 p(1, 1), q(10, 3), r(12, 19);
    if (orientation(q, p, r) == CGAL::LEFT_TURN) {

        // option 2:
        if (CGAL::orientation(p, r, Point(0, 0)) return 1;
    }
    return 0;
}
```
- Similar for other (kernel) predicates

# Example: Intersection of lines

- Given two lines, compute intersections

- `typedef Kernel::Line_2 Line_2;`

`using CGAL; // to simplify examples, but I encourage not to use`

```
int main() {
    Kernel kernel;

    Point_2 p(1, 1), q(2, 3), r(-12, 19);
    Line_2 l1(p, q), l2(r, p);

    if (do_intersect(l1, l2))
        CGAL::Object obj = intersection(l1, l2);
        if (const Point_2 *point = CGAL::object_cast<Point_2>(&obj)) {
            /* do something with *point */
        } else if (const Segment_2 *segment = object_cast<Segment_2>(&obj)) {
            /* do something with *segment */
        }
    }
    return 0;
}
```

---

# Break 1

- Lecture continues at 17:10 ... then:
  - Convex hull reloaded
  - Triangulation

# Convex Hull

- Demo: CGAL::convex\_hull\_2
- But several other algorithms exists:
- ```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/ch_graham_andrew.h>
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;
int main() {
    CGAL::set_ascii_mode (std::cin);
    CGAL::set_ascii_mode (std::cout);
    std::istream_iterator< Point_2 > in_start( std::cin );
    std::istream_iterator< Point_2 > in_end;
    std::ostream_iterator< Point_2 > out( std::cout, "\n" );

    // nice way to read and write to std::io ☺
    CGAL::ch_graham_andrew( in_start, in_end, out );
    return 0;
}
```

# Beyond CGAL::convex\_hull\_2

- Given  $n$  points and  $h$  extreme points ([Link](#))
  - `CGAL::ch_aki_toussaint`  $O(n \log n)$
  - `CGAL::ch_bykat`  $O(nh)$
  - `CGAL::ch_eddy`  $O(nh)$
  - `CGAL::ch_graham_andrew`  $O(n \log n)$
  - `CGAL::ch_jarvis`  $O(nh)$
  - `CGAL::ch_melkman`  $O(n)$  (simple polygon)
- All define the same concept:  
*ConvexHullTraits\_2*

# ConvexHullTraits\_2

- ```
template <class InputIterator, class OutputIterator> OutputIterator  
convex_hull_2(InputIterator first, InputIterator beyond,  
              OutputIterator result,  
              Traits ch_traits = Default_traits)
```
- Default\_traits is the kernel in which the type *InputIterator::value\_type* is defined
- Type: **Point\_2**
- Operations on n points as functors
  - n = 2: **Equal\_2, Less\_xy\_2, Less\_yx\_2**
  - n = 3: **Left\_turn\_2,**  
**Less\_signed\_distance\_to\_line\_2,**  
**Less\_rotate\_ccw\_2** (see manual for later two)
- Misc:
  - CopyConstructor for traits class
  - `traits.equal_2_object(), ...`



# Models for ConvexHullTraits\_2

- Kernel\_2 ☺
- Convex\_hull\_traits\_2<R>
- Convex\_hull\_constructive\_traits\_2<R>
  - avoids repeated constructions (e.g., determinants)
- Convex\_hull\_projective\_xy\_traits\_2<Point\_3>
  - used to compute the convex hull of a set of 3D points projected onto the  $xy$ -plane (*i.e.*, by ignoring the  $z$  coordinate).
  - similar for  $xz$  and  $yz$

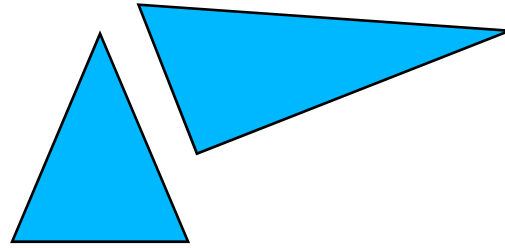
# CH-Substructures

- `CGAL::lower_hull_2`, `CGAL::upper_hull_2`
  - Computation of extreme points of proper hull in CCW order.
  - Andrew's variant of Graham's scan algorithm,  $O(n \log n)$
- `CGAL::ch_jarvis_march`
  - sorted sequence of extreme points on the convex hull between start and stop point
- `CGAL::ch_graham_andrew_scan`
  - sorted sequence of extreme points not left to a given line

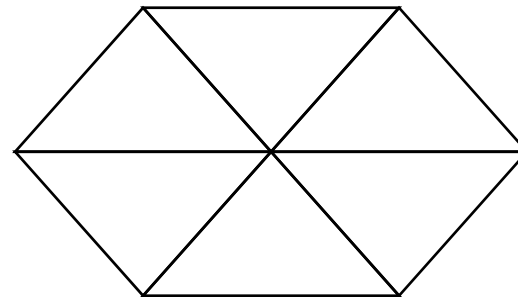
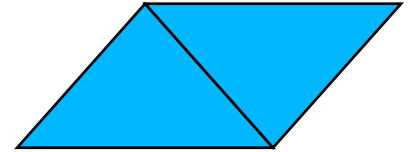
# CH Misc - predicate/algorithm?

- Special extreme points (?)
  - `CGAL::ch_nswe_point` (4 at once)
  - `CGAL::ch_ns_point`, `CGAL::ch_we_point` (2 at once)
  - `CGAL::ch_n_point`, `CGAL::ch_s_point`,  
`CGAL::ch_w_point`, `CGAL::ch_e_point` (single)
- Convexity
  - `CGAL::is_ccw_strongly_convex_2` and  
`CGAL::is_cc_strongly_convex_2`  
check whether a given sequence of 2D points forms a  
(counter)clockwise strongly convex polygon (postcondition)

# Triangulation

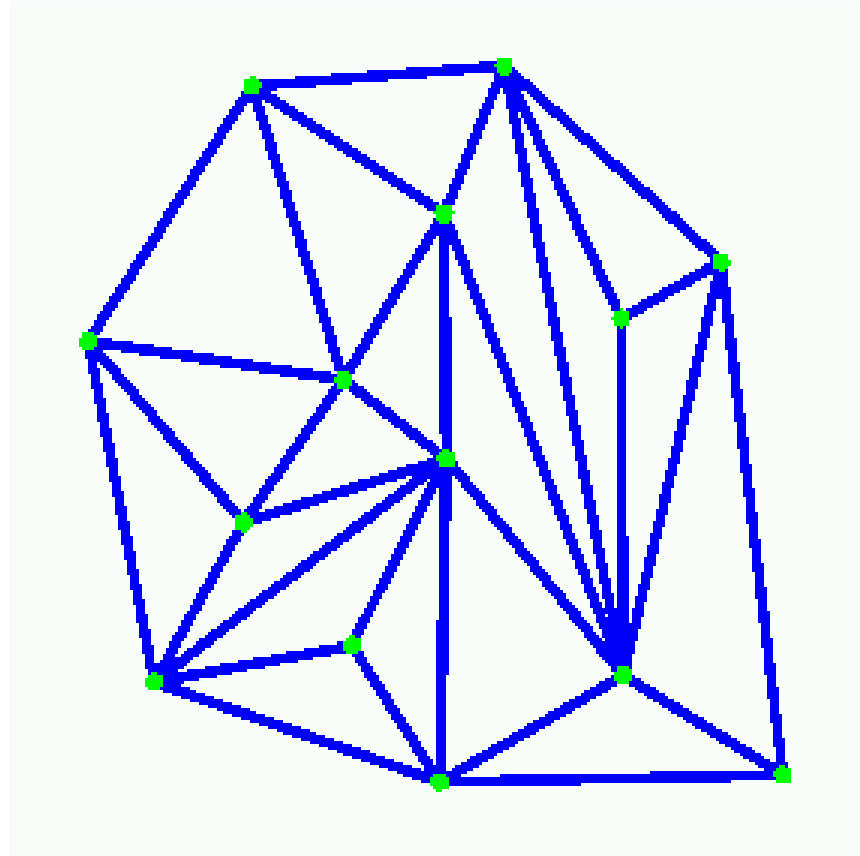


- Given set of points  $P$  in the plane
- Compute a set  $T$  of triangles
  - Interior disjoint: two only shares an edge or a vertex
  - Adjacent: two triangle share an edge and the induced graph is connected
  - Union of triangles has no singularity (surrounding environment is neither a topological ball or disc)
- => Simplicial complex



- Now: Demo

# Triangulation: Example



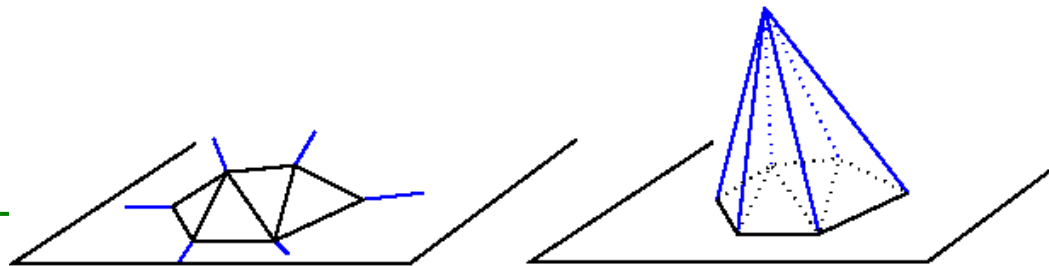
---

# Triangulation: Properties

- Each triangle can have an orientation
  - Induces orientation on edges
  - Orientation of two adjacent triangles is **consistent**, if the shared edge has different orientation in each
- Triangulation is **orientable**, if orientation of each triangle can be chosen, such that all pairs of adjacent triangles are consistent.

# Triangulation in CGAL

- Supports any orientable triangulations
  - without boundaries
  - possible to embed triangulation geometrically
  - Complete, i.e., domain is convex hull ☺ of all vertices
- Thus,  $T$  is a planar partition of the CH
  - Complement of CH is not triangular:
  - Infinite vertex, to which all vertices of CH are connected
    - $\Rightarrow$  only triangles: finite & “infinite”



# Triangulation: First example code

```
■ #include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_2.h>
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Triangulation_2<K> Triangulation;
typedef Triangulation::Vertex_circulator Vertex_circulator;
typedef Triangulation::Point Point;

int main() {
    std::vector< Point > pts =
        { Point(0,0), Point(1,2), Point(3,2), Point(2,2), Point(4,7) };
    Triangulation t;
    t.insert(pts.begin(), pts.end());

    Vertex_circulator vc = t.incident_vertices(t.infinite_vertex()), done(vc);
    if (vc != 0) {
        do {
            std::cout << vc->point() << std::endl;
        } while(++vc != done);
    }
    return 0;
}
```



---

# Software Design

- `Triangulation_2<Traits, Tds>`
  - `Two` parameters (more on next slides)
    - `Geometry Traits (Traits)`
    - `Triangulation Data Structure (Tds)`
  - Access through iterators and circulators
    - See operations below
  - Tests for infinity-ness
  - Point location
  - Modification: Insert, delete, flipping

---

# Triangulation: Geometry Traits

- Three types: `Point_2`, `Segment_2`, `Triangle_2`
- Operations:
  - Comparison of points' x- and y-coordinates
  - Orientation test for three points
- Examples:
  - `Triangulation_euclidean_traits_2<K>`
  - `Triangulation_euclidean_traits_xy_3<K>`
    - Ignores z-coordinates
    - Useful for terrain, e.g.,  
in Geographic Information Systems

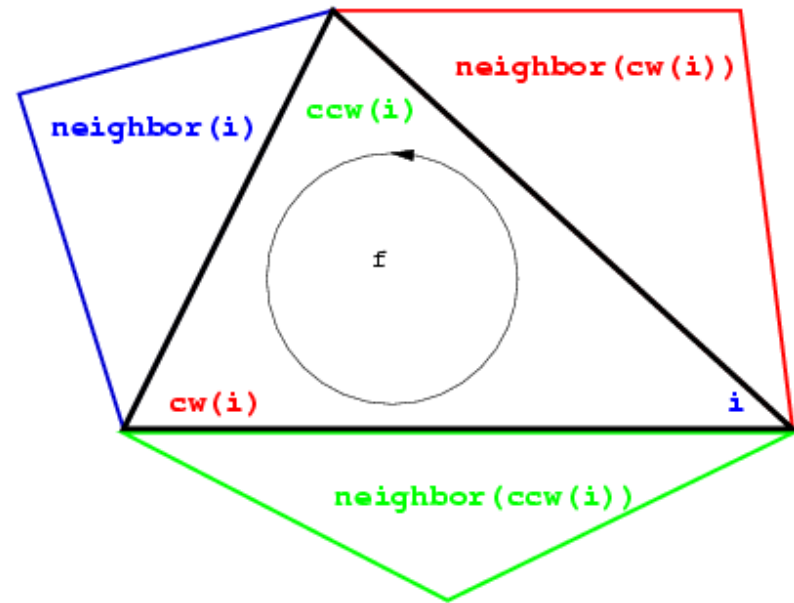
---

# Triangulation data structure

- Container class for vertices and faces
  - themselves,
  - and their incidences and adjacencies
- Responsible for the **combinatorial integrity** of T
  - Operations are purely topological
    - Insert a vertex in a face/edge
    - Flip two edges (one of next slides)
  - I.e., do not depend on the geometric embedding
- More details [online](#)

# Triangulation: Representation

- Based on vertices and faces, not edges
  - Saves storage
  - Results in faster algorithms
- Access of triangle
  - Three incident vertices, indexed 0,1,2 in CCW
  - $\text{neighbor}(i)$  is opposite to  $\text{vertex}(i)$



# Operations: Access

- `int t.dimension()`
  - Returns the dimension of the convex hull.
- `size_type t.number_of_vertices()`  
`size_type t.number_of_faces()`
  - Returns the number of finite vertices/ finite faces
- `Face_handle t.infinite_face()`
  - a face incident to the infinite vertex
- `Vertex_handle t.infinite_vertex()`
  - the infinite vertex
- `Vertex_handle t.finite_vertex()`
  - a vertex distinct from the infinite vertex

# Triangulation: Traversal

- Via circulators/iterators
  - All\_face\_iterator, All\_edges\_iterator, All\_vertices\_iterator
    - Similar for finite counterparts only
  - Point\_iterator
  - Vertex\_circulator, Edge\_circulator, Face\_circulator
    - Circulate features around a given vertex
- ... and handles (allow \* and ->)
  - Vertex\_handle, Edge\_handle, Face\_handle

# Operations: Predicates

- `bool is_infinite(Vertex_handle v)`
  - True iff `v` is infinite
- `bool is_edge(Vertex_handle va, Vertex_handle vb)`
  - True iff there is an edge between `va` and `vb` as vertices
- `bool is_face(Vertex_handle va, Vertex_handle vb, Vertex_handle vc)`
  - True iff there is a face having `va`, `vb` and `vc` as vertices
- and more ... read manual

---

# Locate

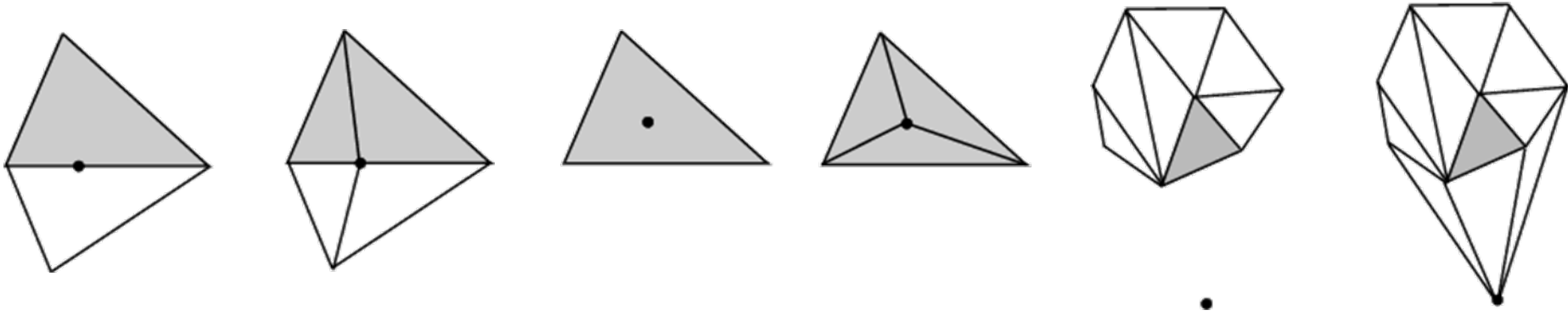
- `Face_handle t.locate(Point q, ...)`
  - Returns a face (triangle) that contains `q` in its interior or its boundary
  - Special result if `q` lies outside `T`, see [manual](#)
- Similar version that also returns
  - enum: VERTEX, EDGE, FACE, OUTSIDE\_CONVEX\_HULL, OUTSIDE\_AFFINE\_HULL
  - if VERTEX or EDGE: index `i`



# Triangulation: Modifiers I

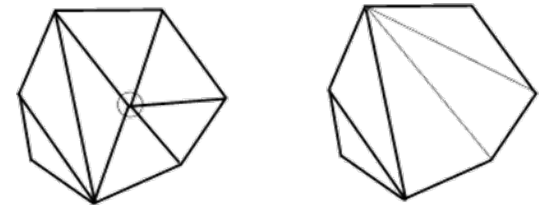
## ■ Insert

- `Vertex_handle t.insert(Point p, ...)`
  - Similar version with previous enum + index
- `template< class InputIter >`  
`int t.insert(InputIter begin, InputIter end)`



## ■ Remove

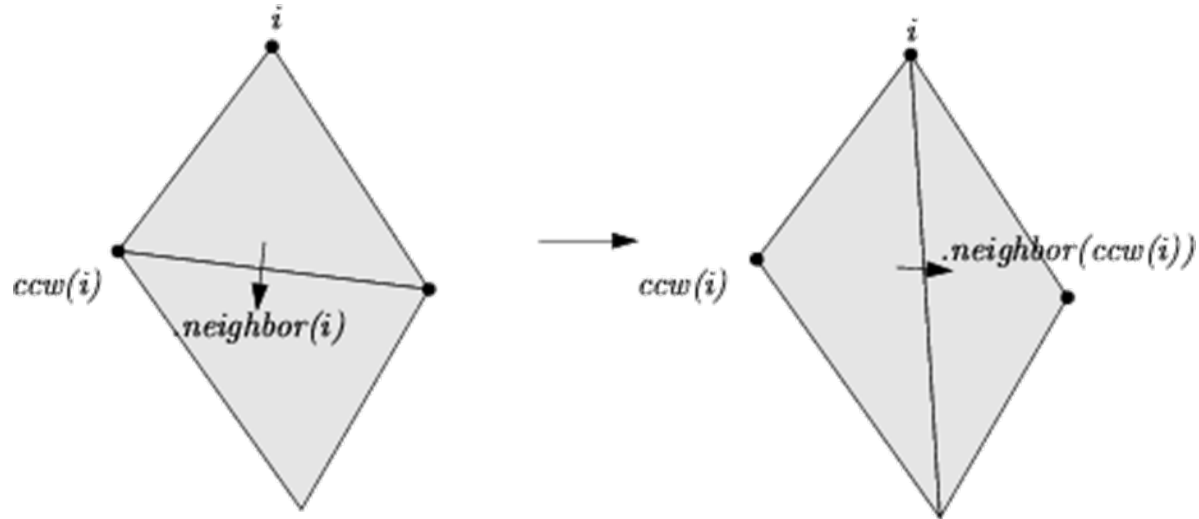
- `void t.remove(Vertex_handle v)`



# Triangulation: Modifiers II

## ■ Flip

- void t.flip(Face\_handle f, int i)
  - Exchanges the edge incident to  $f$  and  $f \rightarrow neighbor(i)$  with the other diagonal of the quadrilateral formed by  $f$  and  $f \rightarrow neighbor(i)$ .

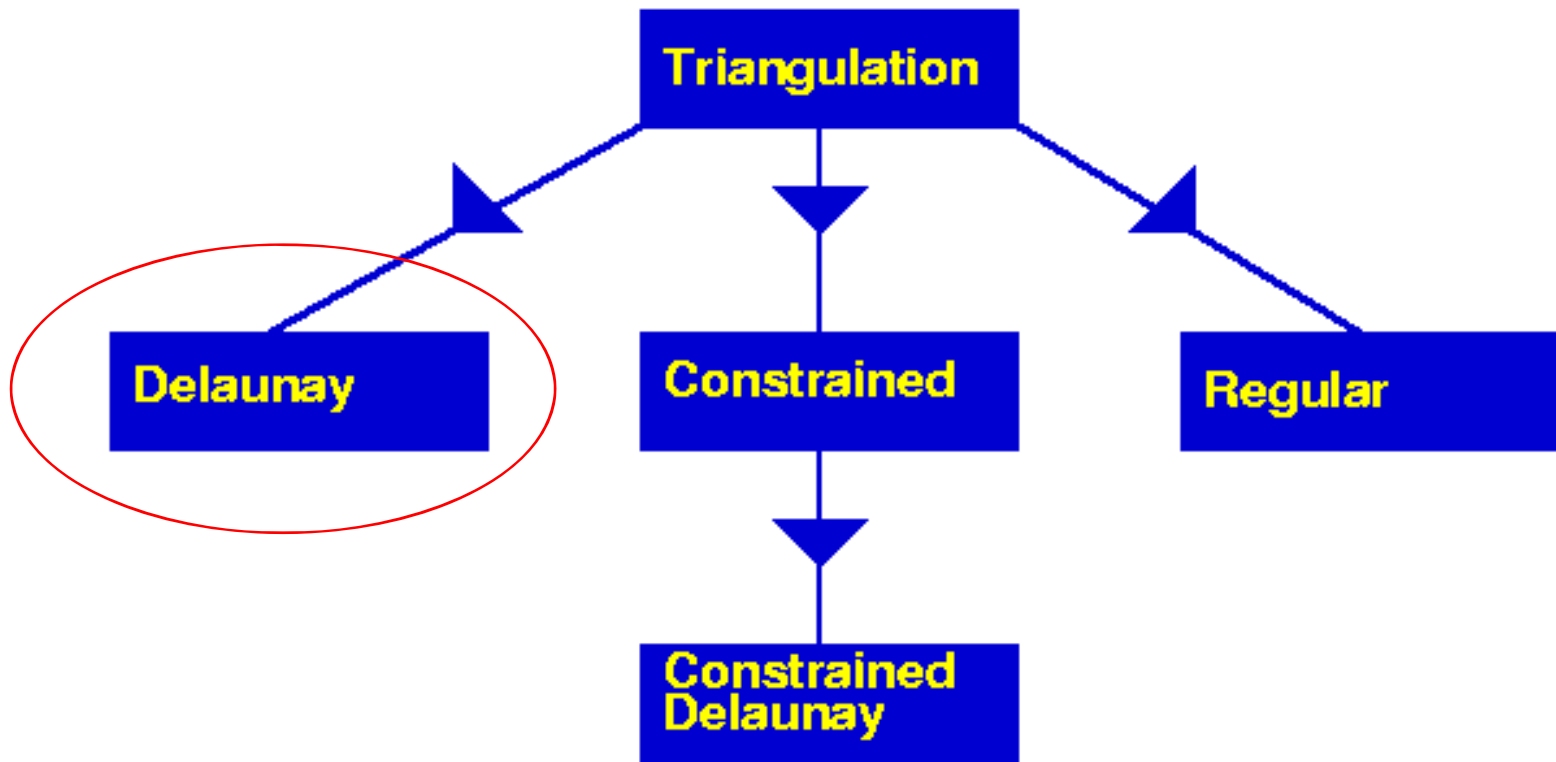


---

# Triangulation: More operations

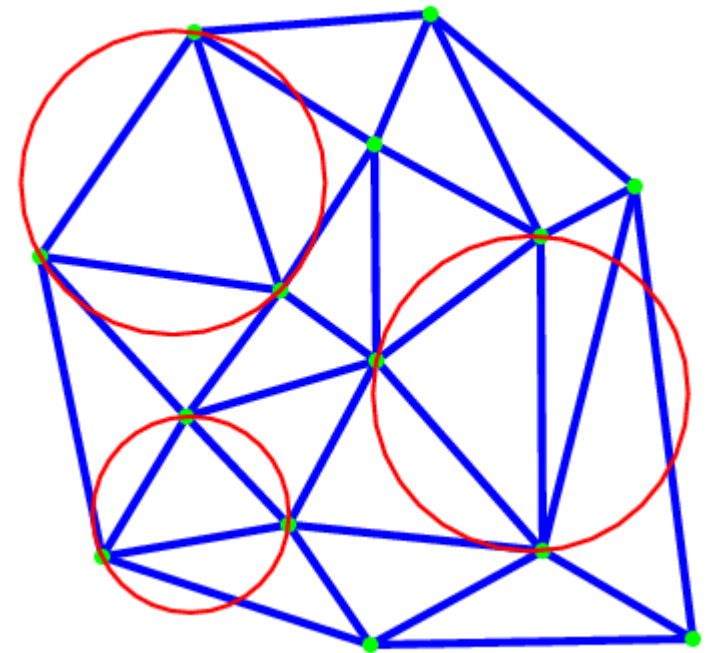
- Line walk
- Convex hull traversal
- Circumcenter
- IO
- ...

# More Triangulations



# Delaunay Triangulation

- Fullfilling the **empty-circle property**:
  - **circumscribing circle of any triangle contains no other data point in its interior**
  - **Unique**, if point-set contains not subset of four co-circular points
  - Its dual corresponds to P's **Voronoi** diagram



---

# Traits concepts for Delaunay

- Geometry traits:
  - Add test for side of oriented circle ([more](#)).
- Delaunay triangulation data structure
  - Is based on known one for triangulations
  - Overwrites insertion / removal, respecting now delaunay property
  - New member to access nearest neighbor
  - Provides access to Voronoi diagram

# Example: Delaunay for a terrain

```
■ #include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_euclidean_traits_xy_3.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <fstream>
```

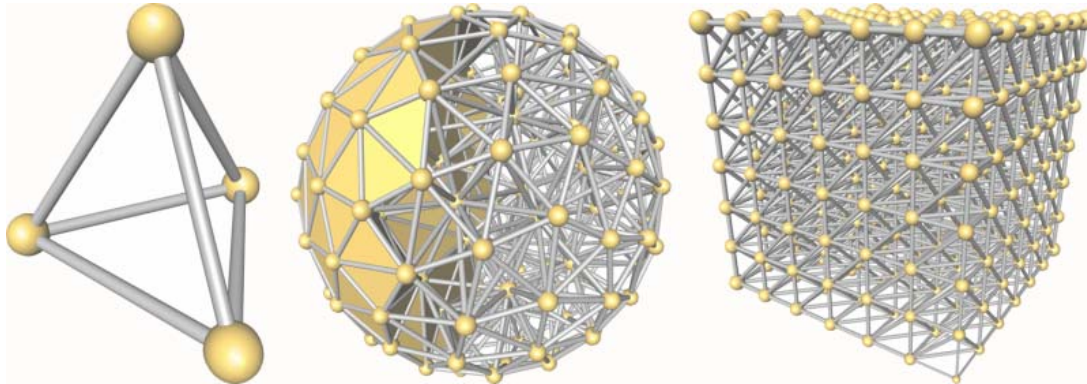
```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Triangulation_euclidean_traits_xy_3<K> Gt;
typedef CGAL::Delaunay_triangulation_2<Gt> Delaunay;
typedef K::Point_3 Point;
```

```
int main() {
    std::ifstream in("data/terrain.cin");
    std::istream_iterator<Point> begin(in);
    std::istream_iterator<Point> end;
```

```
    Delaunay dt; // this and the following line is new (plus includes)
    dt.insert(begin, end);
    std::cout << dt.number_of_vertices() << std::endl;
    return 0;
}
```

# Beyond 2D

- Triangulations in 3D



- Periodic Triangulations (upcoming)
- Meshing
- and much more ...




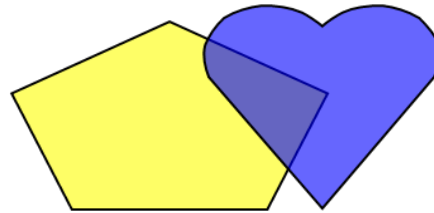
---

# Break 2

- Lectures continues at 18:10 ... then:
  - All about arrangements
  - CGAL-Installation
  - Help for upcoming exercise

# Example: Boolean Set Operations

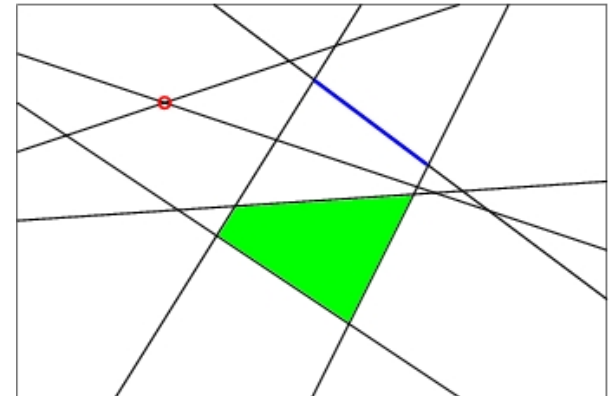
- Given polygons  $P$ ,  $Q$
- Compute Boolean operations on them
  - Intersection 
  - Union
  - (Symmetric) Difference



- Now: Demo in CGAL
-

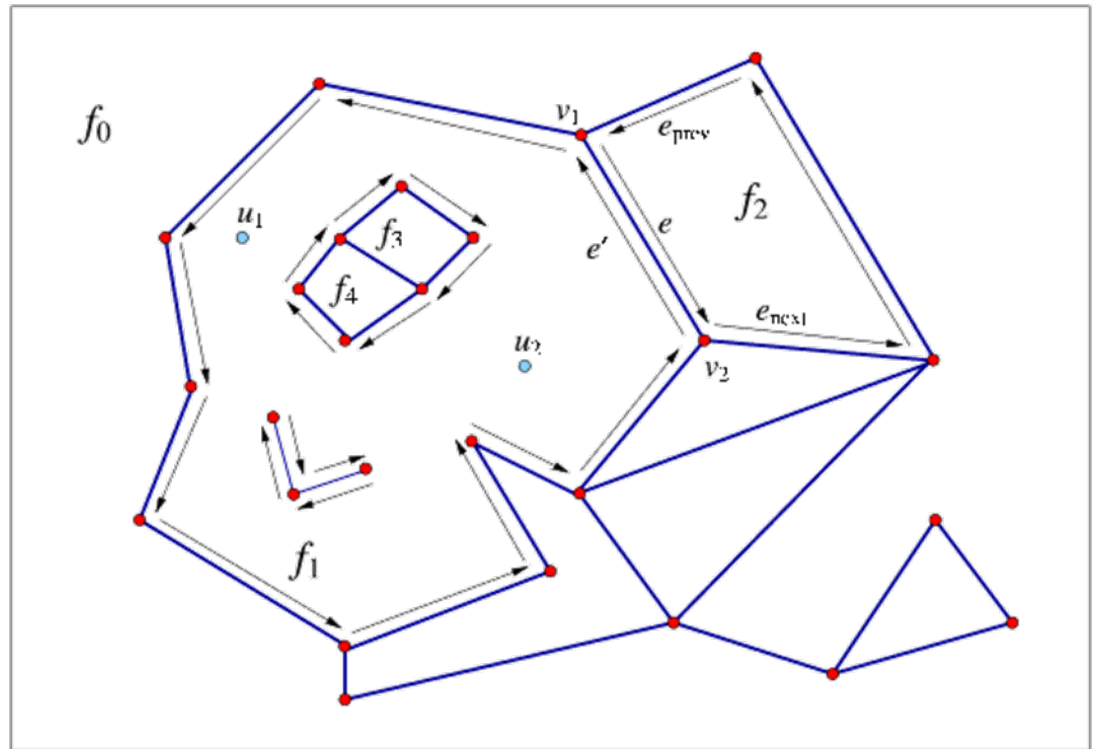
# Arrangements

- Given set of curves  $C$  + isolated points  $P$  (2D)
- Compute induced decomposition of plane into cells of dimension 2, 1, and 0
- Arrangement\_2 package
  - Data structure + Algorithm
  - General input curves, internal x-monotone
  - Extensions



# Arrangements: DCEL

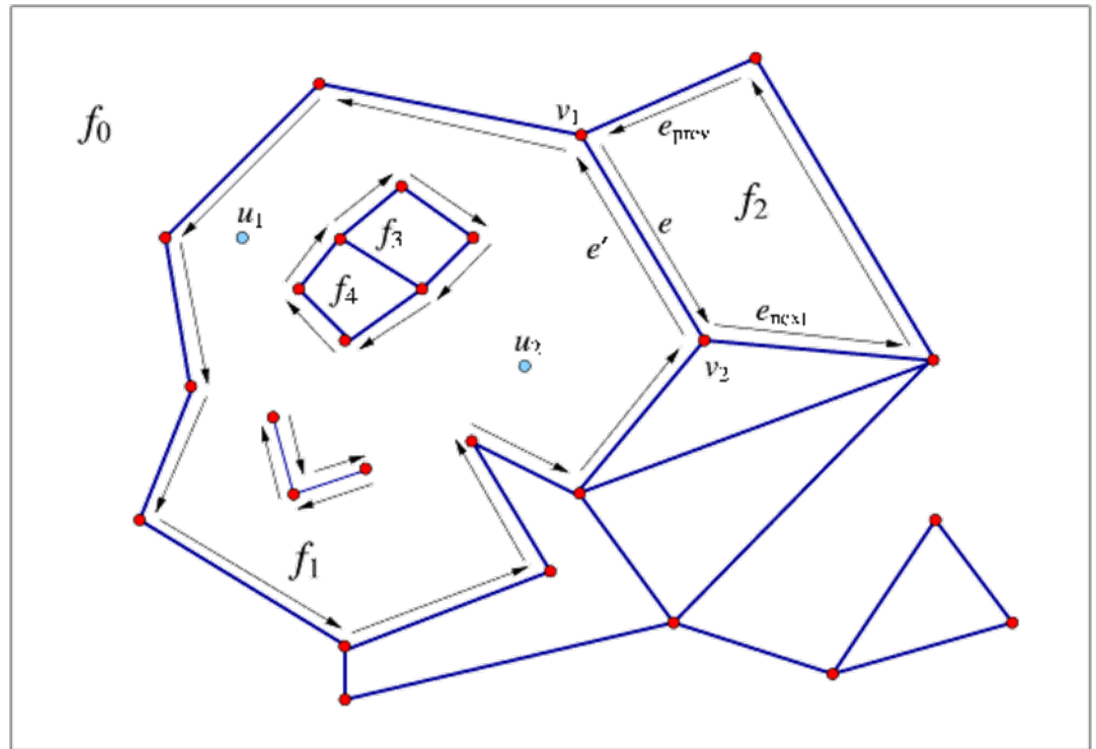
- Arrangement is stored as DCEL (doubly-connected-edge-list)
  - Vertices
  - (Half)edges
  - Faces
  - CCB of face
    - Cycles of halfedges
    - Outer (CCW)
    - Inner (CW)
  - Circulators
    - Edges around  $v$
    - Along CCB



# Arrangements: DCEL

- Associations:

- Edge: `X_monotone_curve_2`
- Vertex: `Point_2`
- Faces: implicit



# Arrangement: Define instance

- `Arrangement_2` < `GeometryTraits`, `Dcel` >
  - recently `Dcel` has been replaced (omit details here)
- `GeometryTraits` must be a model of *ArrangementTraits\_2* concept
  - Types:
    - `Curve_2`
    - `X_monotone_curve_2`
    - `Point_2`
  - Operations: later, when we introduced some algorithms
  - `ArrangementTraits_2` is leaf in a refinement tree
    - Compare to others: expects most number types + operations (for all algorithms/structures that we present today)
    - More details in CGAL manual

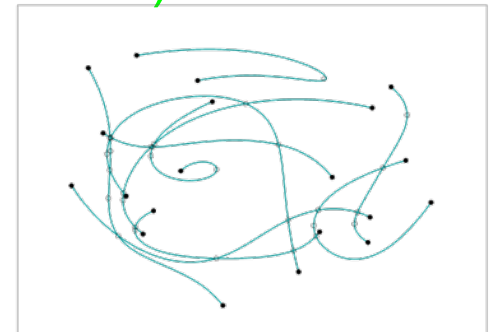
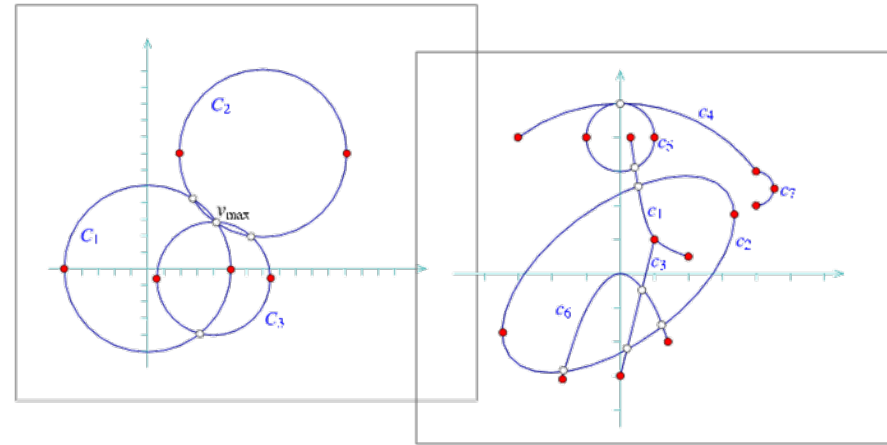
# Arrangement: Available Curves

- In Arrangement\_2 package

- Segments
- Poly\_segments
- Linear Objects
- Circular Arcs (+ segments)
- Arcs of conics (e.g., ellipses, hyperbola, parabola)
- Graphs of functions  $f(x) = p(x)/q(x)$
- Bezier curves

- In CGAL

- Circular Kernel
- Algebraic curves of any degree (only internal)
  - Now: Online Demo



# Arrangement of line segments

```
■ #include <CGAL/Simple_cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

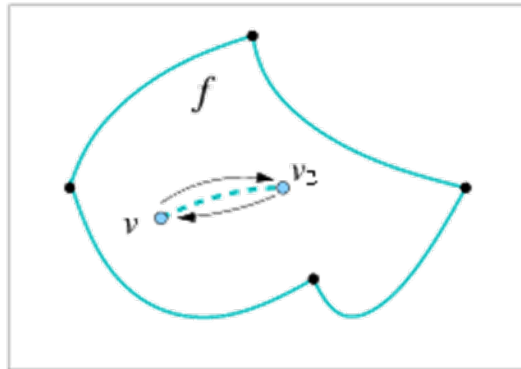
typedef int Number_type;
typedef CGAL::Simple_cartesian<Number_type> Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::Point_2 Point_2;
typedef Traits_2::X_monotone_curve_2 Segment_2;
typedef CGAL::Arrangement_2<Traits_2> Arrangement_2;

typedef Arrangement_2::Vertex_handle Vertex_handle;
typedef Arrangement_2::Halfedge_handle Halfedge_handle;

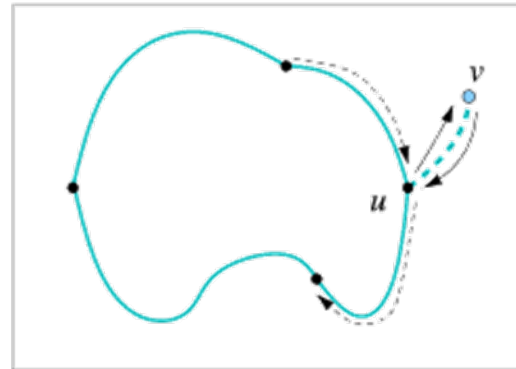
int main () {
    Arrangement_2 arr;
    /* more below ...*/
}
```



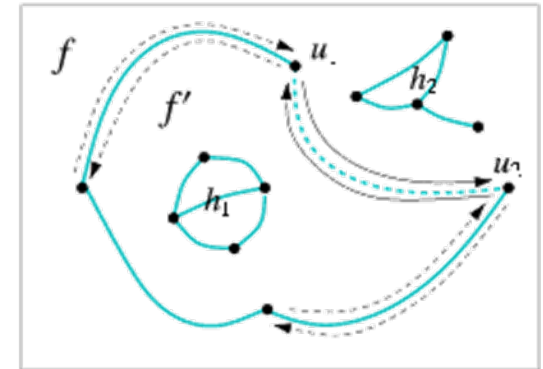
# Basic insertions into DCEL



(a)



(b)



(c)

- (d) Connecting two components
  - Merges CCB
- (e) Insert isolated point

# Arrangement: Insert curves/points

```
int main () {
  Arrangement_2 arr;
  Segment_2 s1 (Point_2 (1, 3), Point_2 (3, 5));
  Segment_2 s2 (Point_2 (3, 5), Point_2 (5, 3));
  Segment_2 s3 (Point_2 (5, 3), Point_2 (3, 1));
  Segment_2 s4 (Point_2 (3, 1), Point_2 (1, 3));
  Segment_2 s5 (Point_2 (1, 3), Point_2 (5, 3));

  Hal_fedge_handle e1 =
    arr.insert_in_face_interior (s1, arr.unbounded_face());

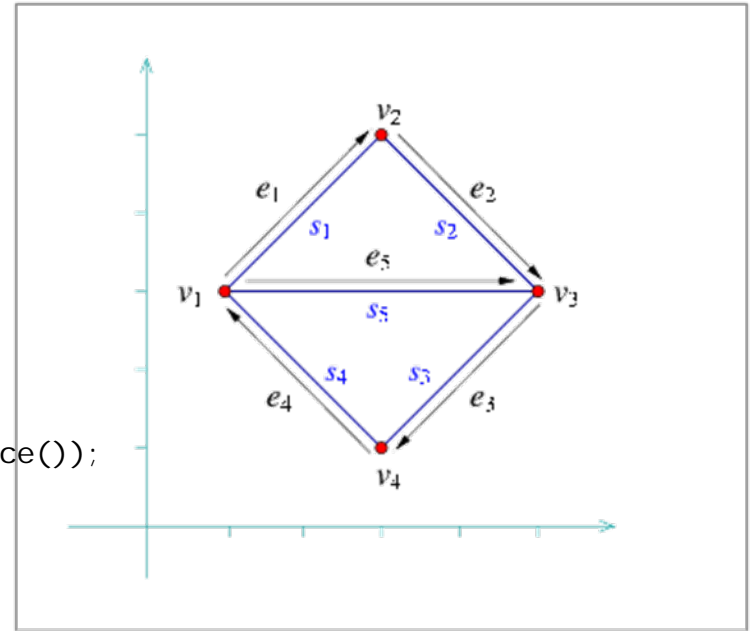
  Vertex_handle v1 = e1->source();
  Vertex_handle v2 = e1->target();
  Hal_fedge_handle e2 =
    arr.insert_from_left_vertex (s2, v2);

  Vertex_handle v3 = e2->target();
  Hal_fedge_handle e3 =
    arr.insert_from_right_vertex (s3, v3);

  Vertex_handle v4 = e3->target();
  Hal_fedge_handle e4 =
    arr.insert_at_vertices (s4, v4, v1);

  Hal_fedge_handle e5 =
    arr.insert_at_vertices (s5, v1, v3);

  return 0;
}
```



There is also `CGAL::insert_vertex(f)`

# Arrangement: Insert curve/point

- Basic insertions are **annoying** – for a user!

- Needs to split curves to be all interior disjoint
- ensure proper calls

```
Poi nt_2 pt (... );  
X_monotone_curve_2 xcv (... );  
Curve_2 cv (... );
```

- Free functions:

- `CGAL::insert(arr, pt);`

- Basic insert, or split-edge

- `CGAL::insert(arr, xcv);`

- Zone algorithm

- `CGAL::insert(arr, cv)`

- Split cv into x-monotone pieces + isolated vertices

- Insert each of them (see below)

# Arrangement: Insert curves/points

```
std::vector< Point_2 > pts;  
std::vector< X_monotone_curve_2 > xcvs;  
std::vector< Curve_2 > cvs;
```

- `CGAL::insert(arr, xcvs.begin(), xcvs.end())`
  - similar function for x-monotone curves & points
  - use the **sweep-line paradigm**
- `CGAL::insert(arr, cvs.begin(), cvs.end())`
  - splits curves into x-monotone subcurves and isolated points, before calling previous function

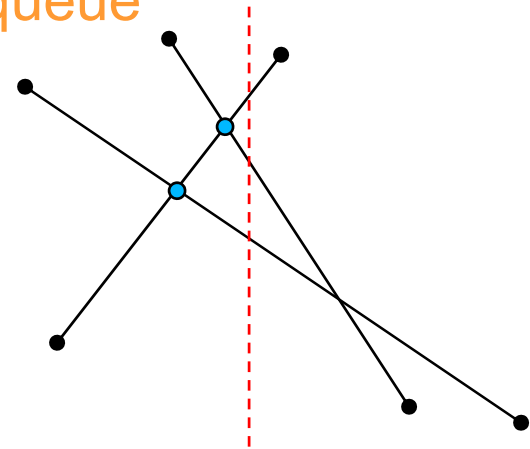
---

# Arrangement: Zone

- Zone:  
Cell of an arrangement intersected by a curve
- Locate minimal end of curve
  - vertex, edge, face
- Traverse curve to maximal end
- During traversal:  
Insert found subcurves with basic insertions
- Example on *blackboard*

# Arrangement: Sweep

- Process a set of curves
  - **Status Structure**: sorted sequence of curves intersecting a vertical line
  - Line moves from left to right: sequence changes
    - at finite number of events: **Event queue**
      - start- and endpoints of curves
      - curves' intersections
    - Processing event:
      - Remove all curves that end
      - Reorder passing curves
      - Insert all curves that start
      - Check adjacent curves for future intersections



# Arrangement: Predicates

- Split curves into x-monotone curves & isolated points
- Compare x, then y of two points (**order of event queue**)
- Determine whether point lies below, above, or on an x-monotone subcurve (**position of curve in status structure**)
- Determine the vertical alignment of two curves to the right of an intersection (**position of curve in status structure: minimal end on existing curve**)
- Compute all intersections (**future intersection**)
- Others: Split and merge curves
  
- All expected by *ArrangementTraits\_2* concept

# Arrangement: Point location

- Given a point locate which face/edges/vertex contains it, e.g., at the beginning of zone
- `typedef CGAL::Arr_nai ve_poi nt_l ocati on<Arrangement_2> Nai ve_pl ;`

```
Arrangement_2 arr;  
/* ... insertions */
```

```
Nai ve_pl nai ve_pl (arr);
```

```
Poi nt_2 query (1, 4);  
CGAL::Object obj = nai ve_pl . l ocati on(query);
```

```
typename Arrangement_on_surface_2::Face_const_handle f;  
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;  
}  
/* ... and similar for edges and vertices */
```

- Other point location strategies:  
Walk along a line, landmarks, trapedoizal decomposition



# Extending the DCEL

- Possible to maintain auxiliary data attached to each vertex, edge, face

- ```
#include <CGAL/Arr_extended_dcel.h>
enum Color {BLUE, RED, WHITE};
```

```
typedef CGAL::Arr_segment_traits_2<Kernel> Traits_2;
typedef CGAL::Arr_extended_dcel<Traits_2, Color, bool, int> Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel> Arrangement_2;

for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
    if (vit->degree() == 0)
        vit->set_data (BLUE); // Isolated vertex.

    else if (vit->degree() <= 2)
        vit->set_data (RED); // Vertex represents an endpoint.

    else
        vit->set_data (WHITE); // Vertex represents an intersection
}
```

- ```
Color vertex_color = vit->data();
```

- Similar for edges + faces

# Arrangement: Overlay

- Given two arrangements, overlay them
  - Introduces new intersections
- ```
/* ... */  
#include <CGAL/Arr_overlay_2.h>  
#include <CGAL/Arr_default_overlay_traits.h>  
typedef CGAL::Arr_default_overlay_traits<Arrangement_2>  
    Overlay_traits;  
Arrangement_2 red, blue;  
/* ... insert curves ... */  
  
Arrangement_2 overlay;  
Overlay_traits overlay_traits;  
CGAL::overlay(arr1, arr2, overlay_arr, overlay_traits);
```
- Uses sweep line paradigm
- “Overlay traits” takes care (if needed) about data attached to red and blue faces, edges, vertices
  - To assign (new) data to “purple faces, edges, vertices”

# Boolean Set Operations

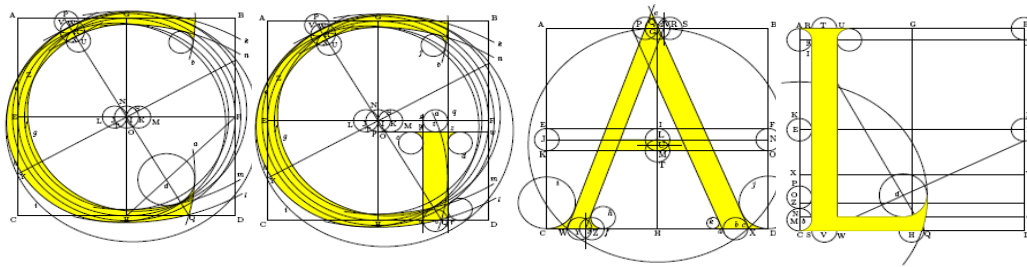
- Extend each vertex, edge, face with ‘bool’
  - true iff cell belongs to point set
- **Overlay + Overlay traits** implements Boolean operation
  - Union, Difference, ... or:
  - Intersection:
    - Red true face, blue true face => purple true face
    - Red false edge, blue false face => purple false edge
    - Red false edge, blue true vertex => purple \_\_\_\_\_ ?
- **Regularized Operations:**
  - Remove low-dimensional cells, as “antennas” and isolated points
  - Efficient Implementation available in CGAL (recall demo)

---

# and much more ...

- Removal of features
- Vertical ray shooting
- Vertical decomposition
- Notifications
- Curve history
- IO
- Adapting Arrangements to Boost Graphs
- (Arrangements on surfaces ...)

# It's your ...



---

# CGAL: Setup

- Various supported platforms:  
Windows, Linux, MacOS
  - Prerequisites:
    - Compiler (g++ > 4.1, MS Visual C++ 9.0)
    - cmake (> 2.4.8)
    - boost (> 1.33.1)
    - Number types (some are provided, like gmp)
    - Qt (for visualization, e.g., 4.5), libGLViewer
-

# CGAL: Installation

- Download CGAL from [www.cgal.org](http://www.cgal.org)
- Full installation details on [http://www.cgal.org/Manual/3.4/doc\\_html/installation\\_manual/contents.html](http://www.cgal.org/Manual/3.4/doc_html/installation_manual/contents.html) or <http://tinyurl.com/CGAL-install>
- More details/options for Boost, Qt, **CGAL** provided (next slide)

```
cd CGAL-3.4 # go to CGAL directory
cmake [options] . # configure CGAL
make # build the CGAL libraries
cd examples/Convex_hull_2 # go to an example directory
cmake -DCGAL_DIR=/pathto/CGAL-3.4 . # configure the examples
make # build the examples
```

- similar for demos and under linux (let's poll)

---

# CGAL-Installations

- Bring USB-Stick to grab Win32-downloads
  - @TAU: installation on the NetApp
    - Set CGAL\_DIR to /home/cgal/home/cgal/<CGAL>
      - Different installations
  - Debian-Packages
-



---

# Your own programs

Two options:

- Copy-and-adapt CGAL examples/demos
    - use cmake-mechanism to update build-environment
  - Build your own makefiles/project
  
  - Reads manuals and check for existing functionality
    - [STL](#)
    - [Boost](#)
    - [CGAL](#)
-

---

# Help for the exercises

- Timer ([Link](#))

```
#include <CGAL/Timer.h>
CGAL::Timer timer;
timer.start();
/* ... */
timer.stop();
std::cout << timer.time() << std::endl;
```

- Drawing with QtGraphicsScene ([Link](#))

- CGAL-3.4/demo/GraphicsView/min.cpp
  - see its “colored” version next slide
-

# Drawing-Example

```
■ #include <iostream>
#include <boost/format.hpp>
#include <QtGui>
#include <CGAL/Qt/GraphicsViewNavigation.h>
#include <QLineF>
#include <QRectF>

int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QGraphicsScene scene;

    scene.setSceneRect(0,0, 100, 100);
    scene.addRect(QRectF(0,0, 100, 100), QPen(QColor(255,0,0)));
    scene.addLine(QLineF(0,0, 100, 100));
    scene.addLine(QLineF(0,100, 100, 0));

    QGraphicsView* view = new QGraphicsView(&scene);
    CGAL::Qt::GraphicsViewNavigation navigation;

    view->installEventFilter(&navigation);
    view->viewport()->installEventFilter(&navigation);
    view->setRenderHint(QPainter::Antialiasing);
    view->show();

    return app.exec();
}
```

---

# Now - it's up to you!

- Have fun!
  - Discuss!
  - Ask questions! Helpdesk:
    - Mo, 15-16, R 018, Schreiber basement
  - Experiment! Implement variants!
  - Read the manual pages!  
<http://tinyurl.com/CGAL-manual>
  
  - Toda raba & layla tov!
-