# Computational Geometry Algorithm Library

**Efi Fogel**

Tel Aviv University

CGAL Crash Course
FUB, July 23rd-24th, 2012, (09:00-12:30)

## Outline





## Geometric Computing: The Goal

(Re)design and implement geometric algorithms and data structures that are at once certified and efficient in practice.



## Geometric Computing: The Assumptions

- Input data is in general position
  - Degenerate input, e.g., three curves intersecting at a common point, is precluded.
- Computational model: the real RAM
  - Operations on real numbers yield accurate results.
- Each basic operation on a small (constant-size) set of simple objects takes unit time.



## Geometric Computing: The Problems

These assumptions often do not hold in practice

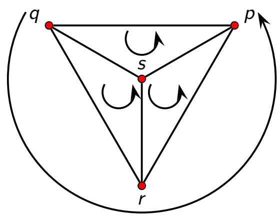
- Degenerate input is commonplace in practical applications.
- Numerical errors are inevitable while using standard computer arithmetic.
- Naive use of floating-point arithmetic causes geometric programs to:
  - Crash after invariant violation
  - Enter an infinite loop
  - Produce wrong output
- There is a gap between Geometry in theory and Geometry with floating-point arithmetic.
  - Standard cs-theory asymptotic performance measures many times poor predictors of practical performance.



## Geometry in Theory

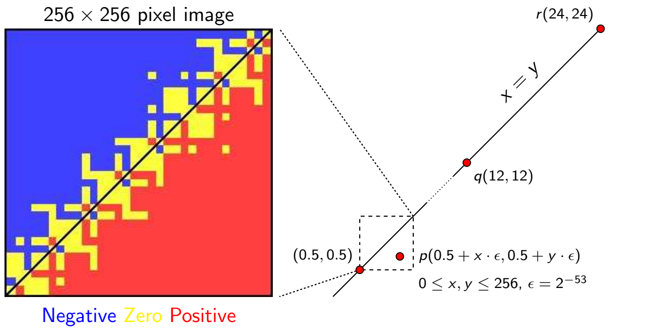
$$\begin{aligned}\mathrm{orientation}(p,q,r) &= \mathrm{sign}\left(\det\begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1\end{bmatrix}\right) \\ &= \mathrm{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x))\end{aligned}$$

$$\mathrm{ccw}(s,q,r) \cap \mathrm{ccw}(s,r,p) \cap \mathrm{ccw}(s,p,q) \Rightarrow \mathrm{ccw}(p,q,r)$$



## Geometry in Practice: Trouble with Double

$$\text{orientation}(p, q, r) = \text{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x))$$




[KMP+08]



## The Naive Solution: Exact Multi-Precision Arithmetic

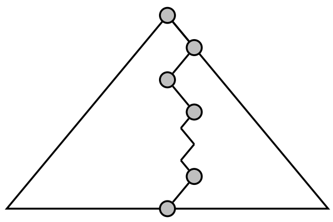
- Implemented for several number types:
  - Integers, rational (e.g., GMP, CORE, and LEDA)
  - Even algebraic numbers (e.g., CORE and LEDA)
  - No solution for transcendental numbers!
- Exact up to memory limit.
- Slow running time.

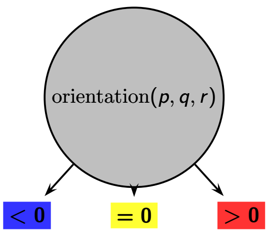


## The Efficient Solution: Exact Geometric Computation

Ensure that the control flow in the implementation corresponds to the control flow with exact arithmetic. [Yap04]

- Evaluate predicate instantiated with limited precision.
- If uncertain $\Longrightarrow$ evaluate predicate instantiated with multiple precision.








## Arithmetic Filters

- The Concept:
  - For expression $E$ compute approximation $\tilde{E}$ and bound $B$, such that $|E - \tilde{E}| \leq B$ or equivalently:

$$E \in I = [\tilde{E} - B, \tilde{E} + B]$$

  - If $0 \in I$ report failure, else return $sign(\tilde{E})$.
- Require only constant time for easy instances.
- Amortize cost for hard cases that use exact arithmetic.



## Floating-Point Arithmetic

- A double float $f$ uses 64 bits
  - 1 bit for the sign $s$.
  - 52 bits for the mantissa $m = m_1 \ldots m_{52}$.
  - 11 bits for the exponent $e = e_1 \ldots e_{52}$.
- $f = -1^s \cdot (1 + \Sigma_{1 \leq i \leq 52} m_i 2^{-i}) \cdot 2^{e-2013}$, if $0 < e < 2^{11} - 1$
- ...
- Notation
  - For $a \in \mathbb{R}$, let $\mathrm{fl}(a)$ denote the closest float to $a$.
  - For $a \in \mathbb{Z}$, $|a - \mathrm{fl}(a)| \leq \epsilon |\mathrm{fl}(a)|$, where $\epsilon = 2^{-53}$.
  - For $o \in \{+, -, \times\}$, $|f_1 \tilde{o} f_2| \leq \epsilon |f_1 \tilde{o} f_2|$.
- Floating-point arithmetic is monotone.
  - e.g., $b \leq c \Rightarrow a \oplus b \leq a \oplus c$.



## Computing the Error Bound

For expression $E$ define $d_E$ and $mes_E$ recursively:

| $E$ | $\tilde{E}$ | $mes_E$ | $d_E$ |
|---|---|---|---|
| $a$, float | $\mathrm{fl}(a)$ | $\lvert\mathrm{fl}(a)\rvert$ | 0 |
| $a \in \mathbb{Z}$ | $\mathrm{fl}(a)$ | $\lvert\mathrm{fl}(a)\rvert$ | 1 |
| $X + Y$ | $\tilde{X} \oplus \tilde{Y}$ | $\lvert\tilde{X}\rvert \oplus \lvert\tilde{Y}\rvert$ | $1 + \max(d_X, d_Y)$ |
| $X - Y$ | $\tilde{X} \ominus \tilde{Y}$ | $\lvert\tilde{X}\rvert \ominus \lvert\tilde{Y}\rvert$ | $1 + \max(d_X, d_Y)$ |
| $X \times Y$ | $\tilde{X} \otimes \tilde{Y}$ | $\lvert\tilde{X}\rvert \otimes \lvert\tilde{Y}\rvert$ | $1 + d_X + d_Y$ |

Then $B$ is defined as follows:

$$|E - \tilde{E}| \leq B = ((1 + \epsilon)^{d_E} - 1) \cdot mes_E$$

[MN00]

## Geometric-Computing Bibliography


Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee K. Yap.
Classroom Examples of Robustness Problems in Geometric Computations.
*Computational Geometry: Theory and Applications*, 40(1):61–78, 2008.

Chee K. Yap.
Robust geomtric computation.
In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, Boca Raton, FL, $2^{n}d$ edition, 2004.

Kurt Mehlhorn and Stefan Näher.
LEDA: *A Platform for Combinatorial and Geometric Computing.*
Cambridge University Press, Cambridge, UK, 2000.




## Generic Programming Paradigm

**Definition (Generic Programming)**

A discipline that consists of the gradual lifting of concrete algorithms abstracting over details, while retaining the algorithm semantics and efficiency.

[MS88]

Translation:

- You do not want to write the same algorithm again and again !
- ⇒ You even want to make it independent from the used types.

See also: http://en.wikipedia.org/wiki/Generic_programming



## Terms and Definitions

**Class Template** A specification for generating (instantiating) classes based on parameters.

**Function Template** A specification for generating (instantiating) functions based on parameters.

**Template Parameter**

**Specialization** A particular instantiation from a template for for a given set of template parameters.



## Generic Programming Dictionary

**Concept** A set of requirements that a class must fulfill.

**Model** A class that fulfills the requirements of a concept.

**Traits** Models that describe behaviors.

**Refinement** An extension of the requirements of another concept.

**Generalization** A reduction of the requirements of another concept.



## Some Generic Programming Libraries

**STL** The C++ Standard Template Library.

**BOOST** A large set of portable and high quality C++ libraries that work well with, and are in the same spirit as, the C++ STL.

**LEDA** The Library of efficient data types and algorithms.

**CGAL** The computational geometry algorithms and data structures library.



## STL Components

**Container** A class template, an instance of which stores collection of objects.

**Iterator** Generalization of pointers; an object that points to another object.

**Algorithm**

**Function Object (Functor)** A computer programming construct invoked as though it were an ordinary function.

**Adaptor** A type that transforms the interface of other types.

**Allocator** An objects for allocating space.

## Generic Algorithms

- A generic algorithm has 2 parts:
  - The actual instructions that describe the steps of the algorithm.
  - A set of requirements that specify which properties its argument types must satisfy.



## A Trivial Example: swap()

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

- When a function call is compiled the function template is instantiated.
- The template parameter T is substituted with a data type.
- The data type must have
  1. a copy constructor, and
  2. an assignment operator.

  In formal words:
  - T is a model of the concept *CopyConstructible*.
  - T is a model of the concept *Assignable*.

  The `int` data type is a model of the 2 concepts.

```
int a = 2, b = 4; std::swap(a,b);
```



## Concept

A concept is a set of requirements divided into four categories:

**Associated Types** — auxiliary types, for example
- `Point_2` — a type that represents a two-dimensional point.

**Valid Expressions** — C++ expressions that must compile successfully, for example
- `p = q`, where `p` and `q` are objects of type `Point_2`.

**Runtime Characteristics** — characteristics of the variables involved in the valid expressions that apply during the variables' lifespans,
- pre/post-conditions.

**Complexity Guarantees** — maximum limits on the computing resources consumed by the valid expressions.



## Generic Programming & the STL Bibliography


- Andrei Alexandrescu.
  *Modern C++ Design: Generic Programming And Design Patterns Applied.*
  Addison-Wesley, Boston, MA, USA, 2001.
- Matthew H. Austern.
  *Generic Programming and the STL.*
  Addison-Wesley, Boston, MA, USA, 1999.
- Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides.
  *Design Patterns — Elements of Reusable Object-Oriented Software.*
  Addison-Wesley, Boston, MA, USA, 1995.
- David R. Musser, Gillmer J. Derge, and Atul Saini.
  *STL tutorial and reference guide: C++ programming with the standard template library.*
  Addison-Wesley, Boston, MA, USA, 2nd edition, Professional Computing Series, 2001.
- David Vandevoorde and Nicolai M. Josuttis.
  *C++ Templates: The Complete Guide,*
  Addison-Wesley, Boston, MA, USA, 2002.
- Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine.
  *The Boost Graph Library,*
  Addison-Wesley, Boston, MA, USA, 2002.
- David A. Musser and Alexander A. Stepanov.
  Generic programming.
  In *Proceedings of International Conference on Symbolic and Algebraic Computation*, volume 358 of LNCS, pages 13–25. Springer, 1988.
- The planned new standard for the C++ programming language.
  `http://en.wikipedia.org/wiki/C++0x#References` .
- The SGI STL.
  `http://www.sgi.com/tech/stl/` .



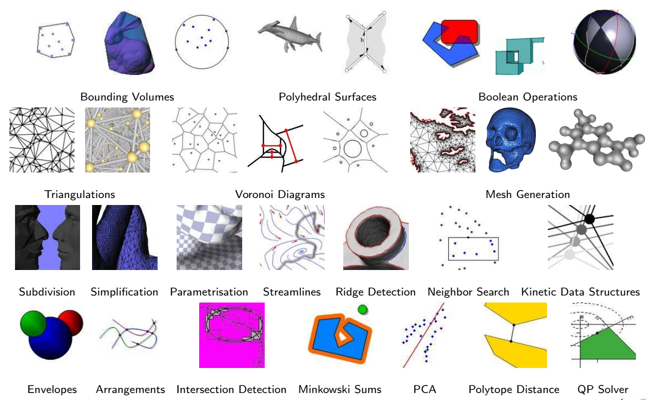

## CGAL: Mission

"Make the large body of geometric algorithms developed in the field of computational geometry available for industrial applications"

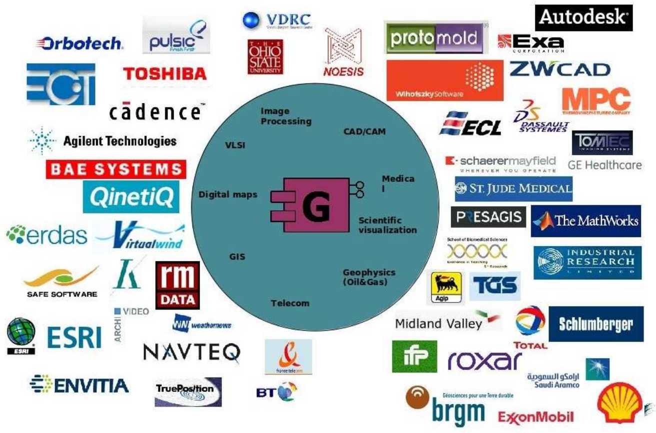
CGAL Project Proposal, 1996



## Some of CGAL Content



## Some CGAL Commercial Users






## CGAL Facts

- Written in C++
- Follows the *generic programming* paradigm
- Development started in 1995
- Active European sites:

1. INRIA Sophia Antipolis
2. MPII Saarbrücken
3. Tel Aviv University
4. ETH Zürich (Plageo)
5. University of Crete and FO.R.T.H.
6. INRIA Nancy
7. Université Claude Bernard de Lyon
8. ENS Paris
9. University of Eindhoven
10. University of California, San Francisco
11. University of Athens



## CGAL History

| Year | Version Released | Other Milestones |
|---|---|---|
| 1996 | | CGAL founded |
| 1998 | July 1.1 | |
| 1999 | | Work continued after end of European support |
| 2001 | Aug 2.3 | Editorial Board established |
| 2002 | May 2.4 | |
| 2003 | Nov 3.0 | GEOMETRY FACTORY founded |
| 2004 | Dec 3.1 | |
| 2006 | May 3.2 | |
| 2007 | Jun 3.3 | |
| 2009 | Jan 3.4, Oct 3.5 | |
| 2010 | Mar 3.6, Oct 3.7 | CGAL participated in Google Summer of Code 2010 |
| 2011 | Apr 3.8, Aug 3.9 | CGAL participated in GSoC 2011 |
| 2012 | Mar 4.0 | CGAL is participating in GSoC 2012 |



## CGAL in Numbers

| | |
|---|---|
| 900,000 | lines of C++ code |
| 10,000 | downloads per year not including Linux distributions |
| 3,500 | manual pages |
| 3,000 | subscribers to cgal-announce list |
| 1,000 | subscribers to cgal-discuss list |
| 120 | packages |
| 60 | commercial users |
| 25 | active developers |
| 6 | months release cycle |
| 7 | Google's page rank for cgal.org.com |
| 2 | licenses: Open Source and commercial |



## CGAL Administration

- INRIAGforge—Collaborative Development Environment
  - Source control, svn ⟹ git
  - Bug tracking
  - Web-based administration, e.g., accounts
- Build system, CMake (cross-platform)
- Nightly testsuite, proprietary ⟹ CTest
- Documentation, proprietary ⟹ Doxygen
- Mailing lists, i.e., discuss, developer, announce
- Developer meetings, 2 annual 1-week
- Websites
  - CGAL, manual ⟹ Content Managment System
  - GEOMETRY FACTORY
  - CGL at TAU, Plone
  - Wiki pages (internal)



## CGAL Properties

- Reliability
  - Explicitly handles degeneracies
  - Follows the Exact Geometric Computation (EGC) paradigm
- Flexibility
  - Is an open library
  - Depends on other libraries (e.g., BOOST, GMP, MPFR, QT, & CORE)
  - Has a modular structure, e.g., geometry and topology are separated
  - Is adaptable to user code
  - Is extensible, e.g., data structures can be extended
- Ease of Use
  - Has didactic and exhaustive Manuals
  - Follows standard concepts (e.g., C++ and STL)
  - Characterizes with a smooth learning-curve
- Efficiency
  - Adheres to the generic-programming paradigm
    - Polymorphism is resolved at compile time

## CGAL Structure

**Basic Library**

Algorithms and Data Structures
e.g., Triangulations, Surfaces, and Arrangements

**Kernel**

Elementary geometric objects
Elementary geometric computations on them

**Support Library**

Configurations, Assertions,...

Visualization
Files
I/O
Number Types
Generators
...



## CGAL Kernel Concept

- Geometric objects of constant size.
- Geometric operations on object of constant size.

| Primitives 2D, 3D, dD | | Operations: Predicates | Operations: Constructions |
|---|---|---|---|
| point | • | comparison | intersection |
| vector | → | orientation | squared distance |
| triangle | △ | containment | ... |
| iso rectangle | ▭ | ... | |
| circle | ○ | | |
| ... | | | |



## CGAL Kernel Affine Geometry

point - origin → vector
point - point → vector
point + vector → point

point + point ← Illegal
midpoint$(a, b) = a + 1/2 \times (b - a)$



## CGAL Kernel Classification

- Dimension: 2, 3, arbitrary
- Number types:
  - Ring: $+, -, \times$
  - Euclidean ring (adds integer division and gcd) (e.g., CGAL::Gmpz).
  - Field: $+, -, \times, /$ (e.g., CGAL::Quotient(CGAL::Gmpz)).
  - Exact sign evaluation for expressions with roots (Field_with_sqr).
- Coordinate representation
  - Cartesian — requires a *field* number type or *Euclidean ring* if no constructions are performed.
  - Homegeneous — requires *Euclidean ring*.
- Reference counting
- Exact, Filtered



## CGAL Kernels and Number Types

| Cartesian representation | Homogeneous representation |
|---|---|
| point $\left\vert \begin{array}{l} x = \frac{hx}{hw} \\ y = \frac{hy}{hw} \end{array}\right.$ | point $\left\vert \begin{array}{l} hx \\ hy \\ hw \end{array}\right.$ |

Intersection of two lines

| | |
|---|---|
| $\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases}$ | $\begin{cases} a_1hx + b_1hy + c_1hw = 0 \\ a_2hx + b_2hy + c_2hw = 0 \end{cases}$ |
| $(x, y) = \left( \frac{\begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}, -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \right)$ | $(hx, hy, hw) = \left( \begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}, -\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}, \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \right)$ |
| Field operations | Ring operations |



## Example: Kernels<NumberType>

- Cartesian<FieldNumberType>
  - typedef CGAL::Cartesian<Gmpq> Kernel;
  - typedef CGAL::Simple_cartesian<double> Kernel;
    - ⋆ No reference-counting, inexact instantiation
- Homogeneous<RingNumberType>
  - typdef CGAL::Homogeneous<Core::BigInt> Kernel;
- d-dimensional Cartesian_d and Homogeneous_d
- Types + Operations
  - Kernel::Point_2, Kernel::Segment_3
  - Kernel::Less_xy_2, Kernel::Construct_bisector_3

## CGAL Numerical Issues

```
typedef CGAL::Cartesian<NT> Kernel;
NT sqrt2 = sqrt(NT(2));

Kernel::Point_2 p(0,0), q(sqrt2,sqrt2);
Kernel::Circle_2 C(p,4);

assert(C.has_on_boundary(q));
```

- OK if NT supports exact sqrt.
- Assertion violation otherwise.



## CGAL Pre-defined Cartesian Kernels

- Support construction of points from `double` Cartesian coordinates.
- Support exact geometric predicates.
- Handle geometric constructions differently:
  - `CGAL::Exact_predicates_inexact_constructions_kernel`
    - ⋆ Geometric constructions may be inexact due to round-off errors.
    - ⋆ It is however more efficient and sufficient for most CGAL algorithms.
  - `CGAL::Exact_predicates_exact_constructions_kernel`
  - `CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt`
    - ⋆ Its number type supports the exact square-root operation.



## CGAL Special Kernels

- Filtered kernels
- 2D circular kernel
- 3D spherical kernel
- Refer to CGAL's manual for more details.



## Computing the Orientation

- imperative style

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel   Kernel;
typedef Kernel::Point_2                                        Point_2;

int main()
{
  Point_2 p(0,0), q(10,3), r(12,19);
  return (CGAL::orientation(q,p,r) == CGAL::LEFT_TURN) ? 0 : 1;
}
```

- precative style

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel   Kernel;
typedef Kernel::Point_2                                        Point_2;
typedef Kernel::Orientation_2                                  Orientation_2;

int main()
{
  Kernel kernel;
  Orientation_2 orientation = kernel.orientation_2_object();

  Point_2 p(0,0), q(10,3), r(12,19);
  return (orientation(q,p,r) == CGAL::LEFT_TURN) ? 0 : 1;
}
```



## Computing the Intersection

```
typedef Kernel::Line_2 Line_2;

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 = CGAL::intersection(l1, l2);
    if (const Point_2* point = 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;
}
```



## CGAL Basic Library

- Generic data structures are parameterized with Traits
  - Separates algorithms and data structures from the geometric kernel.
- Generic algorithms are parameterized with iterator ranges
  - Decouples the algorithm from the data structure.

# CGAL Bibliography


A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr.
On the design of CGAL a computational geometry algorithms library.
*Software — Practice and Experience*, 30(11):1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.

A. Fabri and S. Pion.
A generic lazy evaluation scheme for exact geometric computations.
In $2^{nd}$ *Library-Centric Software Design Workshop*, 2006.

M. H. Overmars.
Designing the computational geometry algorithms library CGAL.
In *Proceedings of ACM Workshop on Applied Computational Geometry, Towards Geometric Engineering*, volume 1148, pages 53–58, London, UK, 1996. Springer.

The CGAL Project.
CGAL *User and Reference Manual.*
CGAL Editorial Board, 3.9 edition, 2010. http://www.cgal.org/Manual/3.9/doc_html/cgal_manual/contents.html .

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

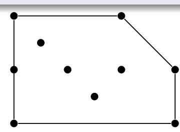



# Convex Hull Terms and Definitions

**Definition (convex hull)**

The convex hull of a set of points $P \subseteq \mathbb{R}^d$, denoted as conv($P$), is the smallest (inclusionwise) convex set containing $P$.

When an elastic band stretched open to encompass the input points is released, it assumes the shape of the convex hull.



$n$ — the number of input points.
$h$ — the number of points in the hull.

- Time complexities of convex hull computation:
  - Optimal, output sensitive: $O(n \log h)$. [Chan96]
  - QuickHull (expected): $O(n \log n)$. [BDH96]



# Convex Hull Properties

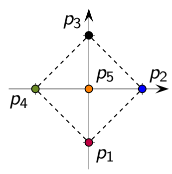
- A subset $S \subseteq \mathbb{R}^d$ is convex $\Leftrightarrow$ the line segment $\overline{pq} \in S$ for any two points $p, q \in S$.
- The convex hull of a set $S$ is the smallest convex set containing $S$.
- The convex hull of a set of points $P$ is a convex polygon with vertices in $P$.

- Input: set of points $P$ (or objects).
- Output: the convex hull $S \subseteq P$ of $P$.



# CGAL Convex Hull

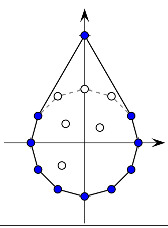
```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/convex_hull_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel  Kernel;
typedef Kernel::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");
  CGAL::convex_hull_2(in_start, in_end, out);
  return 0;
}
```






# Incremental Convex Hull

- The edge $\overline{pq}$ is visible from $r$
  $\Leftrightarrow \mathrm{orientation}(p, q, r) < 0$
- The edge $\overline{pq}$ is weakly visible from $r$
  $\Leftrightarrow \mathrm{orientation}(p, q, r) \leq 0$



Maintain the current convex hull $S$ of a set of points seen so far

1. Initialize $S$ to the counter-clockwise sequence $\{a, b, c\} \subset P$
2. Remove $a$, $b$, and $c$ from $P$
3. **for all** $r \in P$ **do**
4. **if** there is an edge $e$ visible from $r$ **then**
5. Compute the sequence of edges, $\{\overline{v_i v_{i+1}}, \ldots, \overline{v_{j-1} v_j}\}$, weakly visible from $r$
6. Replace the sequence $\{v_{i+1}, \ldots, v_{j-1}\}$ by $r$

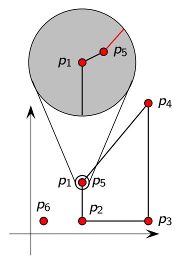
- The sequence of edges weakly visible from $r$, $\{\overline{v_i v_{i+1}}, \ldots, \overline{v_{j-1} v_j}\}$, is a consecutive chain



# Wrong Incremental Convex Hull

$p_1 = (24.00000000000005, 24.000000000000053)$
$p_2 = (24.0, 6.0)$
$p_3 = (54.85, 6.0)$
$p_4 = (54.850000000000357, 61.000000000000121)$
$p_5 = (24.000000000000068, 24.000000000000071)$
$p_6 = (6, 6)$

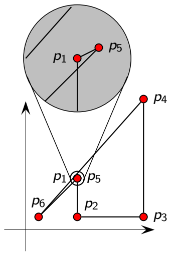
- $(p_1, p_2, p_3, p_4)$ form a convex quadrilateral.
- $p_5$ is truly inside this quadrilateral.
- $\mathrm{orientation}{*}(p_4, p_1, p_5) < 0$.
- $\mathrm{orientation}{*}(p_4, p_5, p_6) < 0$.
- $\mathrm{orientation}{*}(p_5, p_1, p_6) > 0$.
- $\mathrm{orientation}{*}(p_1, p_2, p_6) < 0$.




[KMP+08]

## Wrong Incremental Convex Hull

$p_1 =$ (24.00000000000005, 24.000000000000053)
$p_2 =$ (24.0, 6.0)
$p_3 =$ (54.85, 6.0)
$p_4 =$ (54.850000000000357, 61.000000000000121)
$p_5 =$ (24.000000000000068, 24.000000000000071)
$p_6 =$ (6, 6)




- $(p_1, p_2, p_3, p_4)$ form a convex quadrilateral.
- $p_5$ is truly inside this quadrilateral.
- $\text{orientation}*(p_4, p_1, p_5) < 0$.
- $\text{orientation}*(p_4, p_5, p_6) < 0$.
- $\text{orientation}*(p_5, p_1, p_6) > 0$.
- $\text{orientation}*(p_1, p_2, p_6) < 0$.

[KMP+08]



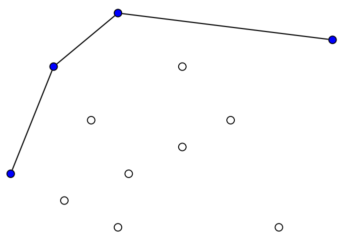
## Graham's Scan Convex Hull

Andrew's variant of Graham's scan algorithm

Compute the convex hull $S$ of a set of points $P$
Sort the points in lexicographic order, resulting in the sequence $p_1, \ldots, p_n$
$S_{\text{upper}} \leftarrow \{p_1, p_2\}$
**for** $i \leftarrow 3$ **to** $n$
  **while** $|S_{\text{upper}}| > 2$ & !rightturn$(q, r, p_i)$, $q$ and $r$ are the last points of $S_{\text{upper}}$
    $S_{\text{upper}} \leftarrow S_{\text{upper}} \setminus \{r\}$
  $S_{\text{upper}} \leftarrow S_{\text{upper}} \cup \{p_i\}$
$S_{\text{lower}} \leftarrow \{p_n, p_{n-1}\}$
**for** $i \leftarrow n-2$ **downto** 1
  **while** $|S_{\text{lower}}| > 2$ & !rightturn$(q, r, p_i)$, $q$ and $r$ are the last points of $S_{\text{lower}}$
    $S_{\text{lower}} \leftarrow S_{\text{lower}} \setminus \{r\}$
  $S_{\text{lower}} \leftarrow S_{\text{lower}} \cup \{p_i\}$
Remove the last points from $S_{\text{lower}}$ and $S_{\text{upper}}$
$S \leftarrow S_{\text{upper}} \cup S_{\text{lower}}$



## Graham's Scan Convex Hull



**Theorem (Convex Hull)**

*The convex hull of a set of n points in the plane can be computed in $O(n \log n)$ time.*



## CGAL Convex-Hull Implementations

- Given $n$ points and $h$ extreme points

| | | |
|---|---|---|
| CGAL::ch_akl_toussaint() | $O(n \log n)$ | |
| CGAL::ch_bykat() | $O(nh)$ | |
| CGAL::ch_eddy() | $O(nh)$ | |
| CGAL::ch_graham_andrew() | $O(nlogn)$ | |
| CGAL::ch_jarvis() | $O(nh)$ | |
| CGAL::ch_melkman() | $O(n)$ | (simple polygon) |



## Upper Convex Hull

```
#include <iostream>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
typedef CGAL::Exact_predicates_inexact_constructions_kernel   Kernel;
int main()
{
  Kernel kernel;
  std::vector<Kernel::Point_2> in;
  std::vector<Kernel::Point_2> out;
  upper_hull(in.begin(), in.end(), std::back_inserter(out), kernel);
  return 0;
}
```

- Using Random Access Iterator. [upper_convex_hull_1.cpp]
  - Provides both increment and decrement.
  - Constant-time methods for moving forward and backward in arbitrary-sized steps ('[]' operator).
- Maintaining the current point separately. [upper_convex_hull_2.cpp]
- Maintaining an iterator that points to the current point separately. [upper_convex_hull_3.cpp]
- More generic — can be used for computing the lower hull. [upper_convex_hull_4.cpp]



## Convex-Hull Bibliography


C. Bradford Barber, David P. Dobkin, and Hannu T. Huhdanpaa
The Quickhull algorithm for convex hulls.
*ACM Transactions on Mathematical Software*, 22(4):469-483, 1996.

Timothy M. Chan
Optimal output-sensitive convex hull algorithms in two and three dimensions.
*Discrete & Computational Geometry*, 16:361–368, 1996.

Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee K. Yap.
Classroom Examples of Robustness Problems in Geometric Computations.
*Computational Geometry: Theory and Applications*, 40(1):61–78, 2008.

## Outline

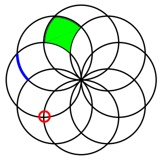




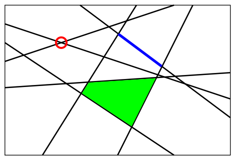
## Two Dimensional Arrangements

**Definition (Arrangement)**

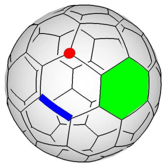
Given a collection $\mathcal{C}$ of curves on a surface, the arrangement $\mathcal{A}(\mathcal{C})$ is the partition of the surface into vertices, edges and faces induced by the curves of $\mathcal{C}$.



An arrangement of circles in the plane.



An arrangement of lines in the plane.

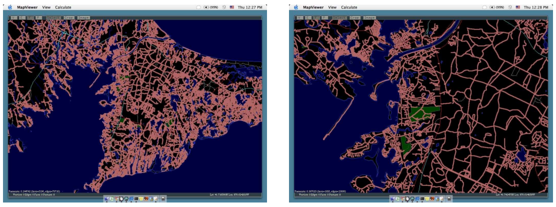


An arrangement of great-circle arcs on a sphere.



## 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 $\mathcal{C}$ are well behaved, if each pair of curves in $\mathcal{C}$ 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 $\Omega(n^d)$.

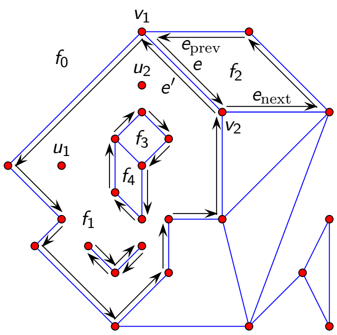


## The CGAL `Arrangement_on_surface_2` Package

- 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.



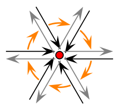
## The Doubly-Connected Edge List Components

- 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 (CCB)
  - The circular chains of halfedges around faces.

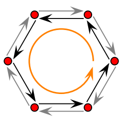


## Arrangement Representation

- 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.

```
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

```
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 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;
}
```



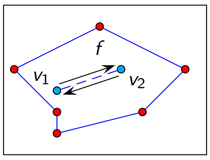
## Traversing an Arrangement

Print all the cells of an arrangement.

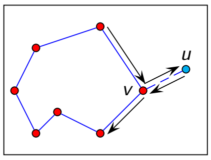
```cpp
template <typename Arrangement>
void print_arrangement(const Arrangement& arr)
{
  CGAL_precondition(arr.is_valid());

  // 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);
}
```



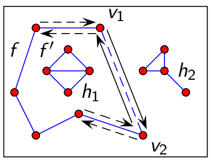
## Modifying the Arrangement




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)`.



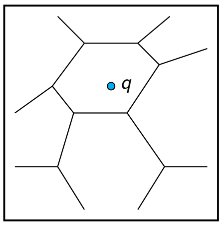

Inserting an $x$-monotone curve, the endpoints of which correspond to existing vertices $v_1$ and $v_2$, `insert_at_vertices(c,v1,v2)`.

- 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.



## Arrangement Point Location

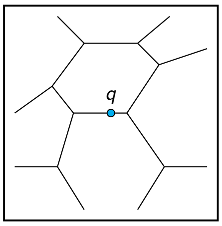
Given a subdivision $A$ of the space into cells and a query point $q$, find the cell of $A$ containing $q$.






## Arrangement Point Location

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

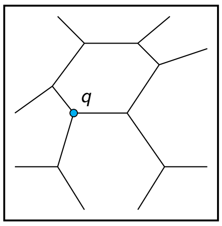



- In degenerate situations the query point can
  - lie on an edge, or



## 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.

## 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 **R**andomized **I**ncremental-**C**onstruction (RIC)
- Landmark



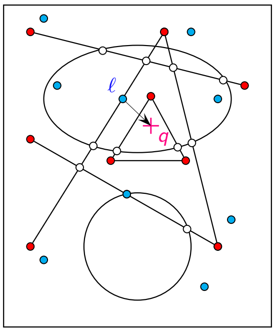
## Walk Along a Line

- 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.



## 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 $\Longrightarrow$ 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.
- 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

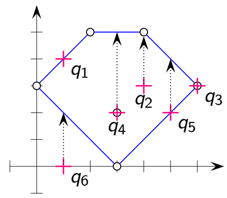
Print a polymorphic object.

```
template <typename Arrangement>
void print_point_location(const typename Arrangement::Point_2& q,
                          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

```
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

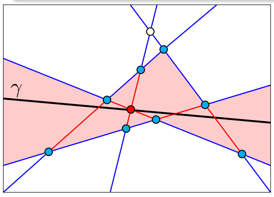
```
// File: ex_point_location.cpp

#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;

  // 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;
}
```



## The Zone of Curves in Arrangements

**Definition (Zone)**

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




The zone of a line $\gamma$ in an arrangement of lines.



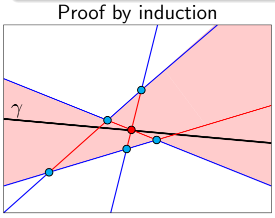
## The Zone of lines in Arrangements of Lines

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.*

Proof by induction




| | Vertices | Edges | Faces | Total |
|---|---|---|---|---|
| Number | 1 | 6 | 6 | 13 |
| Complexity | 1 | 11 | 16 | 28 |



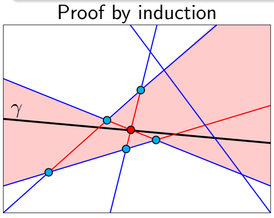
## The Zone of lines in Arrangements of Lines

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.*

Proof by induction




| | Vertices | Edges | Faces | Total |
|---|---|---|---|---|
| Number | 1 | 6 | 6 | 13 |
| Complexity | 1 | 11 | 16 | 28 |

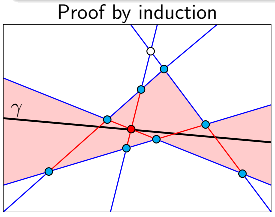
## The Zone of lines in Arrangements of Lines

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.*

Proof by induction




| | Vertices | Edges | Faces | Total |
|---|---|---|---|---|
| Number | 1 | 7 | 7 | 15 |
| Complexity | 1 | 14 | 22 | 37 |



## Zone Application: Incremental Insertion

**Definition (Incremental Insertion)**

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

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



## Zone Complexity

- $\mathcal{C}$ — a collection of $n$ well-behaved curves,
- $s$ — the maximum number of intersections between curves in $\mathcal{C}$.
- $\gamma$ — another well-behaved curve intersecting each curve of $\mathcal{C}$ in at most some constant number of points.
- $\lambda_s(n)$ — the maximum length of an $(n, s)$-Davenport-Schinzel sequence.

**Theorem (Zone Complexity)**

*The complexity of the zone of $\gamma$ in the arrangement $\mathcal{A}(\mathcal{C})$ is $O(\lambda_s(n))$ if the curves in $\mathcal{C}$ are all unbounded, or $O(\lambda_{s+2}(n))$ in case they are bounded. The zone can be computed in time close to the complexity of the computed zone.*

[SA95]



## Davenport-Schinzel Sequences

**Definition (Davenport-Schinzel Sequence)**

a Davenport-Schinzel sequence is a sequence of symbols in which the number of times any two symbols may appear in alternation is limited.

- $n$, $s$ — positive integers.
- $U = u_1, u_2, \ldots, u_m$ — a sequence of integers.
- $U$ is called an $(n, s)$-DS sequence if
  - $\forall i, 1 \le i \le m, 1 \le u_i \le n$ (the alphabet).
  - $\forall i, 1 \le i < m, u_i \ne u_{i+1}$ (distinct consecutive values).
  - There do not exist $s + 2$ indices $i_1 < i_2 < \ldots i_{s+2}$, such that $u_{i_1} = u_{i_3} = \ldots = u_{i_{s+1}} = j$ and $u_{i_2} = u_{i_4} = \ldots = u_{i_{s+2}} = k$ for two distinct numbers $1 \le j, k \le n$.
- $\lambda_s(n) = \max\{|U| \mid U$ is an $(n, s)$-DS sequence$\}$.



## Davenport-Schinzel Sequence Example

- $x$ and $y$ are two distinct values occurring in the sequence.
- The sequence does not contain a subsequence $\ldots, x, \ldots, y, \ldots, x, \ldots, y, \ldots$ consisting of $s + 2$ values alternating between $x$ and $y$.
- The sequence $1, 2, 1, 3, 1, 3, 2, 4, 5, 4, 5, 2, 3$ is a $(5, 3)$-DS sequence.
- It contains alternating subsequences of length four, such as $\ldots, 1, \ldots, 2, \ldots, 1, \ldots, 2, \ldots$
  - Which appears in four different ways as a subsequence of the whole sequence.
- It does not contain any alternating subsequences of length five.



## Davenport-Schinzel Sequence Length Bounds

- $A$ — the rapidly growing Ackerman function.
- The best bounds known on $\lambda_s$ involve the inverse Ackermann function:
$$\alpha(n) = \min\{m \mid A(m, m) \ge n\}$$
- $\alpha(n)$ grows very slowly.
- For problems of any practical size $\alpha(n) \le 4$.

$$\lambda_1(n) = n$$
$$\lambda_2(n) = 2n - 1$$
$$\lambda_3(n) \le 2n\alpha(n) + O(n\sqrt{\alpha(n)})$$

## The complexity of a single face

- The appearances of the curves along a CCB of a face constitute a DS sequence.
- $\mathcal{C}$ — a collection of $n$ well-behaved curves,
- $s$ — the maximum number of intersections between curves in $\mathcal{C}$.
- $\mathcal{A}$ — the arrangement induced by $\mathcal{C}$.
- the complexity of a face in $\mathcal{A}$ is:
  - $\lambda_s(n)$ — if the curves in $\mathcal{C}$ are unbounded.
  - $\lambda_{s+2}(n)$ — if the curves in $\mathcal{C}$ are bounded.
  - $\lambda_{s+1}(n)$ — if the curves in $\mathcal{C}$ are bounded on one side.



## Constructing a Single Face

- Deterministic algorithm
  - $O(\lambda_s(n) \log^2 n)$ time — if the curves in $\mathcal{C}$ are unbounded.
  - $O(\lambda_{s+2}(n) \log^2 n)$ time — if the curves in $\mathcal{C}$ are bounded.
- Randomized algorithm
  - Expected $O(\lambda_s(n) \log n)$ time — if the curves in $\mathcal{C}$ are unbounded.
  - Expected $O(\lambda_{s+2}(n) \log n)$ time — if the curves in $\mathcal{C}$ are bounded.
- For bounded curves the complexity and construction-time complexity of the zone follows.



## The Zone Computation Algorithmic Framework

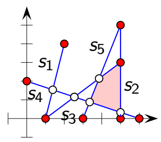
Arrangement_zone_2 class template

- 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);
  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;
}
```

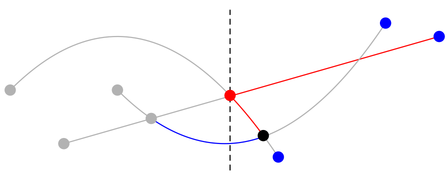





## The Plane Sweep Algorithmic Framework

[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



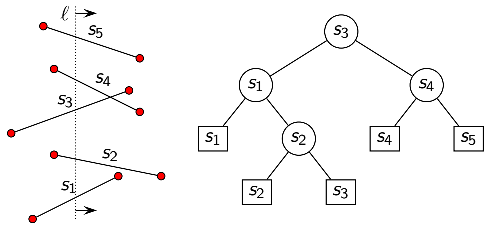


## Plane Sweep: Event Queue

- 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 — $(O(logm)$ 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**

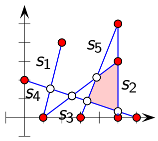
*All points of intersection between the curves in $\mathcal{C}$ can be reported in $O((n+k)\log n)$ time and $O(n)$ space.*

- $\mathcal{C}$ — a set of $n$ $x$-monotone curves in the plane.
- $k$ is— 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
    - $p$ does not have right curves $\implies$ 1 event might be generated.
    - $p$ has right curves $\implies$ 2 events might be generated.
- 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.



## Aggregate Insertion

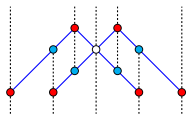
```cpp
// File: ex_aggregated_insertion.cpp

#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;
  insert(arr, segments, segments + sizeof(segments)/sizeof(Segment_2));
  print_arrangement_size(arr);
  return 0;
}
```






## Vertical Decomposition

- Is a refinement of the original subdivision.
- In the plane
  - Contains pseudo trapezoids (triangles and trapezoids).
  - A pseudo trapezoid is determined by 2 vertices and 2 segments.
    - ★ $\mathcal{A}$ — an arrangement.
    - ★ $n$ — # of edges in $\mathcal{A}$.
    - ★ $O(n)$ — # of trapezoids ($\leq 3n+1$).

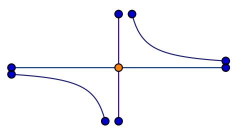


- Generalizes to higher dimensions and arrangements induces by well behaved objects.



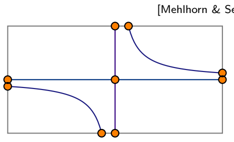
## Handling Enpoints at Infinity

Clipping the unbounded curves



Using an infimaximal box

[Mehlhorn & Seel, 2003]

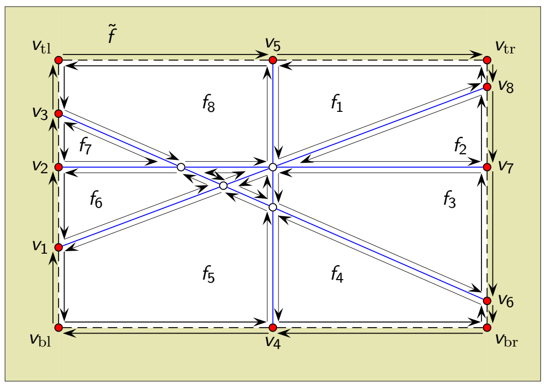


- Simple, the sweep algorithm is unchanged
- Not online
- The resulting arrangement has a single unbounded face

- 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 Curves



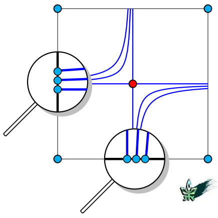
## Unbounded Arrangement Vertices

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:

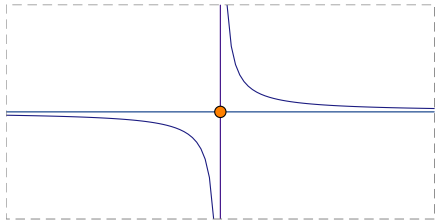
- 1 edge associated with an $x$-monotone curve, and
- 2 fictitious edges connecting the vertex to its adjacent vertices at infinity or the corners of the bounding rectangle.





## Sweeping Unbounded Curves

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

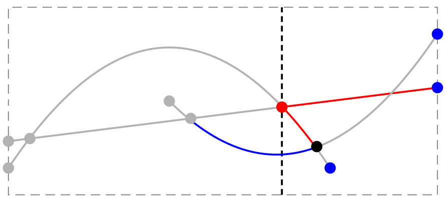


$xy = 1$, $x = 0$, and $y = 0$



## 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!





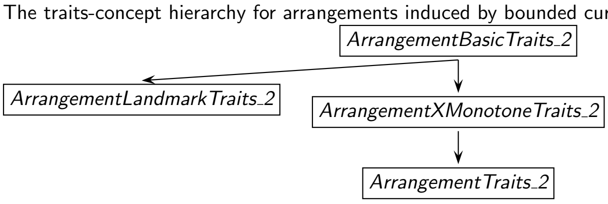
## Arrangement Geometry Traits

- Separates geometric aspects from topological aspects
  - `Arrangement_2<Traits, Dcel>` — main component.
- Is a parameter of the data structures and algorithms.
  - Defines the family of curves that induce the arrangement.
  - A parameterized data structure or algorithm can be used with any family of curves for which a traits class is supplied.
- Aggregates
  - Geometric types (point, curve).
  - Operations over types (accessors, predicates, constructors).
- Each input curve is subdivided into x-monotone subcurves.
  - Most operations involve points and x-monotone curves.



## Arrangement Traits Hierarchy

The traits-concept hierarchy for arrangements induced by bounded curves.



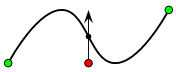



## *ArrangementBasicTraits_2* Concept

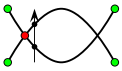
- Types:
  - `Point_2`
  - `X_monotone_curve_2`
- Methods:
  1. `Compare_x_2` — Compares the $x$-coordinates of 2 points.
  2. `Compare_xy_2` — Lexicographically compares the $x$-coordinates of 2 points.
  3. `Equal_2`
     - Are two points represent the same geometric entity?
     - Are two $x$-monotone curves represent the same geometric entity?
  4. `Construct_min_vertex` — Returns the lexicographically smallest endpoint of an $x$-monotone curve.
  5. `Construct_max_vertex` — Returns the lexicographically lasgest endpoint of an $x$-monotone curve.
  6. `Is_vertical` — Determines whether an $x$-monotone curve is vertical.

## *ArrangementBasicTraits_2* Concept (Cont.)

- Methods:
  7. Compare_y_at_x_2 — Determines the relative position of an $x$-monotone curve and a point.



  8. Compare_y_at_x_right_2 — Determines the relative position of 2 $x$-monotone curves to the right of a point.



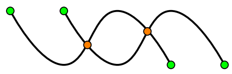
  9. Compare_y_at_x_left_2 — Determines the relative position of 2 $x$-monotone curves to the left of a point (optional).
- Categories:
  - Has_left_category — Determines whether the predicate Compare_y_at_x_left_2 is supported.
  - Determines whether $x$-monotone curves may reach the corresponding boundary.
    - ⋆ Arr_left_side_category
    - ⋆ Arr_right_side_category
    - ⋆ Arr_bottom_side_category
    - ⋆ Arr_top_side_category



## *ArrangementXMonotoneTraits_2* Concept

Supporting intersecting bounded curves.

- Methods:
  1. Split_2 — Splits an $x$-monotone curve at a point into two interior disjoint subcurves.
  2. Are_mergeable_2 — Determines whether two curves can be merged into a single curve.
  3. Merge_2 — Merge two mergeable curves into a single curve.
  4. Intersection_2 — Find all intersections of 2 $x$-monotone curves.

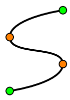




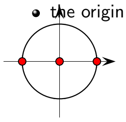
## *ArrangementTraits_2* Concept

Supporting arbitrary bounded curves.

- Types: Curve_2
- Methods:
  1. Make_x_monotone_2 — Subdivides a curve into $x$-monotone curves and isolated points.



- $(x^2+y^2)(x^2+y^2-1)=0$ — the defining polynomial of a curve $c$.
- $c$ comprizes of
  - the unit circle (the locus of all points for which $x^2+y^2=1$) and
  - the origin (the singular point $(0,0)$).

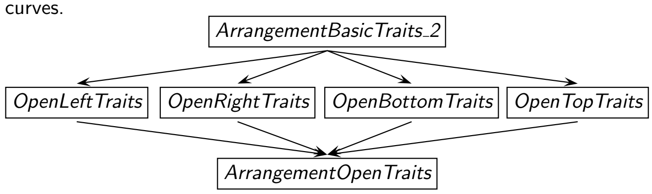


- $c$ is subdivided into two circular arcs and an isolated point.



## Arrangement Traits Hierarchy

The traits-concept hierarchy for arrangements induced by unbounded curves.

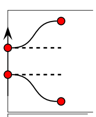




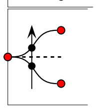

## Arrangement *ArrangementOpenTraits* Concept

Supporting unbounded curves.

- Methods:
  1. Parameter_space_in_x_2 — Determines the location of the curve end along the $x$-dimension.
  2. Compare_y_limit_on_boundary_2 — Compare the $y$-coordinate of 2 curve ends at their limits.



  3. Compare_y_near_boundary_2 — Compare the $y$-coordinate of 2 curve ends near their limits.
     - ⋆ Precondition: the $y$-coordinate of the curves at their limit is equal.

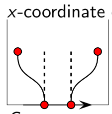




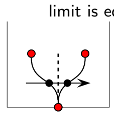
## *ArrangementOpenTraits* Concept (cont.)

Supporting unbounded curves.

- Methods:
  1. Parameter_space_in_y_2 — Determines the location of the curve end along the $y$-dimension.
  2. Compare_x_limit_on_boundary_2 — Compare the $x$-coordinate of 2 curve ends at their limits.



  3. Compare_x_near_boundary_2 — Compare the $x$-coordinate of 2 curve ends near their limits.
     - ⋆ Precondition: the $x$-coordinate of the curves at their limit is equal.



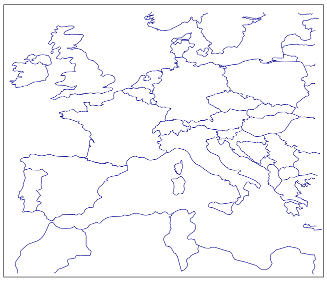
## Arrangement Traits Models

- Line segments:
  1. Uses the kernel point and segment types.
  2. Caches the underlying line.
- Linear curves, i.e., line segments, rays, and lines.
- Circular arcs and line segments.
- Conic curves
- Arcs of rational functions.
- Bézier curves.
- Algebraic curves of arbitrary degrees.



## Traits Model: Non Caching Line Segments

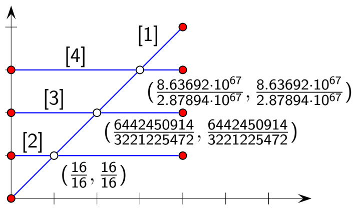
- `Arr_non_caching_segment_traits_2<Kernel>`
- Nested point type is `Kernel::Point_2`.
- Nested curve type is `Kernel::Segment_2`.
- Most of the defined operations are delegations of the corresponding operations of the `Kernel` type.





## The Effect of Cascading of Intersections

- An arrangement induced by 4 line segments.
- The segments are inserted in the order indicated in brackets.
- The insertion creates a cascading effect of segment intersection.
- The bit-lengths of the intersection-point coords grow exponentially.
- Indiscriminate normalization considerably slows down the arrangement construction.





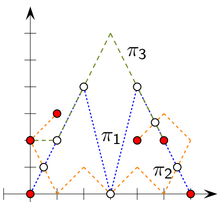

## Traits Model: Caching Line Segments

- `Arr_segment_traits_2<Kernel>`
- Nested point type is `Kernel::Point_2` (like `Arr_segment_non_caching_traits_2`)
- A segment is represented by:
  - its two endpoints,
  - its supporting line,
  - a flag indicating whether the segment is vertical, and
  - a flag indicating whether the segment target-point is lexicographically larger than its source.
- Superior when the number of intersections is large.



## Traits Model: Polylines

- `Arr_polyline_traits_2<SegmentTraits>`
- A polyline is a continuous piecewise linear curves.
- Polylines
  - can be used to approximate more complex curves, and
  - are easier to handle than higher-degree algebraic curves.
    - Rational arithmetic is sufficient.
- You are free to choose the underlying line-segment traits.

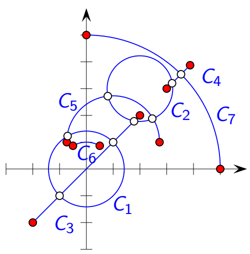



`ex_polylines.cpp`



## Traits Model: Circular Arcs & Line Segments

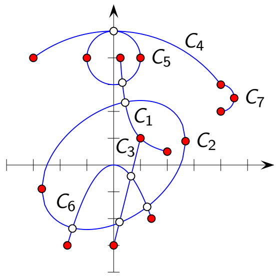
- `Arr_circle_segment_traits_2<Kernel>`
- Supports line segments, circular arcs, and circles.
- Line coefficients, circle-center coordinates, and radius squares are rational numbers.
- Intersections between two circles are numbers of degree 2.
  - $\alpha + \beta\sqrt{\gamma}$
- These numbers are represented by the dedicated square-root extension type.




`ex_circular_arcs.cpp`

## Traits Model: Conic Curves

- Arr_conic_traits_2 <RatKernel , AlgKernel , NtTraits >
- A conic curve is an algebraic curves of degree 2.
- Supports bounded conics
  - arcs of circles, ellipses, parabolas, and hyperbolas, and
  - whole circles and ellipses.
- RatKernel — A rational kernel.
- AlgKernel — An algebraic kernel.
- NtTraits — Provides numerical operations.
  - conversion between number types,
  - solving quadratic equations, and
  - extracting the real roots of a polynomial.




ex_conic_arcs.cpp



## Traits Model: Arcs of Rational Functions

- Arr_rational_arc_traits_2 <AlgKernel , NtTraits >

**Definition (polynomial)**
A polynomial is an expression of finite length constructed from variables and constants, using only the operations of addition, subtraction, multiplication, and non-negative integer exponents.

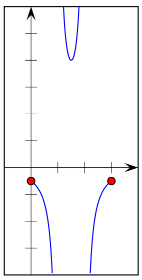
For example: $x^2 - 4x + 7$

- $P(x)$, $Q(x)$ — Univariate polynomials of arbitrary degrees.
- $y = \frac{P(x)}{Q(x)}$ — A rational function.
- The coefficient are rational numbers.
- $[x_{\min}, x_{\max}]$ is an interval over which an arc is defined $\Longrightarrow$ $x_{\min}$ and $x_{\max}$ can be arbitrary algebraic numbers.
- Suports arcs of rational functions.



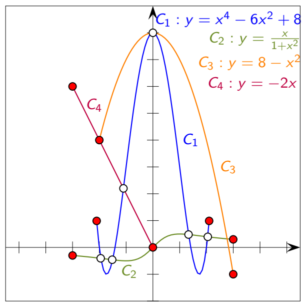
## Traits Model: Arcs of Rational Functions (Cont.)

- A rational arc is always $x$-monotone.
- A rational arc is not necessarily continuous.
- $y = \frac{1}{(x-1)(2-x)}$ defined over the interval $[0, 3]$.
  - Has two singularities at $x = 1$ and at $x = 2$.
  - Is subdivided by Make_x_monotone_2 into 3 continuous portions defined over the intervals $[0, 1)$, $(1, 2)$, and $(2, 3]$, respectively.

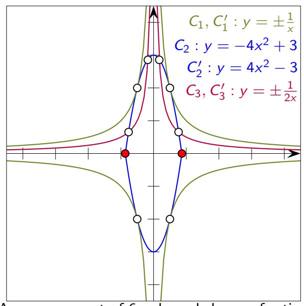




## Traits Model: Arcs of Rational Functions (Cont.)




An arrangement of 4 bounded rational arcs (ex_rational_functions.cpp).



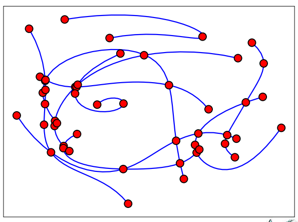

An arrangement of 6 unbounded arcs of rational functions (ex_unbounded_rational_functions.cpp



## Traits Model: Bézier Curves

**Definition (Bézier curve)**
Given a set of $n + 1$ control points $P_0, P_1, \ldots, P_n$, the corresponding Bézier curve is given by $C(t) = (X(t), Y(t)) = \Sigma_{i=0}^{n} P_i \binom{n}{i} t^i (1 - t)^{n-i}$, where $t \in [0, 1]$.

- $X(t)$, $Y(t)$ — Univariate polynomials of degree $n$.
- Supports self-intersecting Bézier curves.
- Control-point coords are rational number.
- Intersection-points coords are algebraic numbers.
  - Access to the approximate coordinates is possible.
  - You cannot obtain the point coords as algebraic numbers.



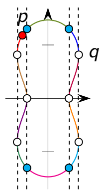


## Traits Model: Algebraic Curves

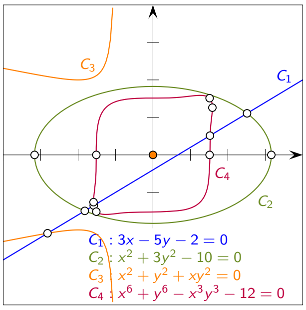
- Arr_algebraic_segment_traits_2 <Coefficient >

**Definition (Algebraic curve)**
An algebraic curve is the (real) zero set of a bivariate polynomial $f(x, y)$.

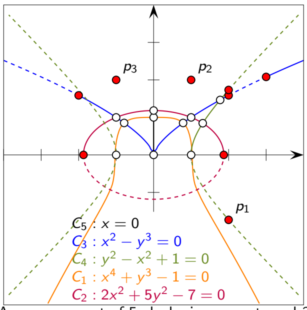
- Supports
  - algebraic curves and
  - continuous $x$-monotone segments of algebraic curves, which are not necessarily maximal.
  - Non $x$-monotone segments are not supported.
  - $x$-monotone segments are not necessarily maximal.
- An Oval of Cassini, $(y^2 + x^2 + 1)^2 - 4y^2 = 4/3$.
  - The induced arrangement consists of 2 faces, 10 edges, and 10 vertices.



## Traits Model: Algebraic Curves




An arrangement of 4 algebraic curves (ex_algebraic_curves.cpp).




An arrangement of 5 algebraic segments and 3 isolated points (algebraic_segments.cpp). The supporting curves are drawn as dashed lines.



## Arrangement Geometry Traits Models

① — *ArrangementLandmarkTraits_2* ② — *ArrangementTraits_2*
③ — *ArrangementDirectionalXMonotoneTraits_2* ④ — *ArrangementOpenTraits_2*.

| Model Name | Curve Family | Degree | Concepts | |
|---|---|---|---|---|
| Arr_non_caching_segment_basic_traits_2 | line segments | 1 | ① | |
| Arr_non_caching_segment_traits_2 | line segments | 1 | ①,②,③ | |
| Arr_segment_traits_2 | line segments | 1 | ①,②,③ | |
| Arr_linear_traits_2 | line segments, rays, and lines | 1 | ①,②,③,④ | |
| Arr_circle_segment_traits_2 | line segments and circular arcs | $\leq 2$ | ②,③ | |
| Arr_circular_line_arc_traits_2 | line segments and circular arcs | $\leq 2$ | ② | |
| Arr_conic_traits_2 | circles, ellipses, and conic arcs, | $\leq 2$ | ①,②,③ | |
| Arr_rational_function_traits_2 | curves of rational functions | $\leq 2$ | ①,②,③,④ | |
| Arr_Bezier_curve_traits_2 | Bézier curves | $\leq n$ | ②,③ | |
| Arr_algebraic_segment_traits_2 | algebraic curves | $\leq n$ | ②,③,④ | |
| Arr_polyline_traits_2 | polylines | $\infty$ | ①,②,③ | |



## The Notification Mechanism

**Definition (Observer)**

An observer defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.

- The *2D Arrangements* package offers a mechanism that uses *observers*.
- The observed type is derived from an instance of Arr_observer<Arrangement>.
- The observed object does not know anything about the observers.
- Each arrangement object stores a list of pointers to Arr_observer objects.
- The trapezoidal-RIC and the landmark point-location strategies use observers to keep their auxiliary data-structures up-to-date.



## Observer Notification Functions

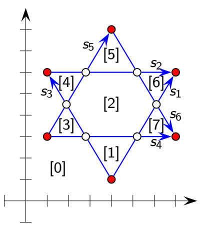
The set of functions can be divided into 3 categories:

1. Notifiers on changes that affect the topological structure of the arrangement. There are 2 pairs (*before* and *after*) that notify when
   - the arrangement is cleared or
   - the arrangement is assigned with the contents of another one.
2. Pairs of notifiers before and after of a local change that occurs in the topological structure.
   - A new vertex is constructed or deleted.
   - An new edge is constructed or deleted.
   - 1 edge is split into 2 edges, or 2 are merged into 1.
   - 1 face is split into 2 faces, or 2 are merged into 1.
   - 1 hole is created in the interior of a face or removed from it.
   - 2 holes are merged into 1, or 1 is split into 2.
   - A hole is moved from one face to another.
3. Notifiers on a structural change caused by a free function. A single pair before_global_change() and after_global_change().



## Extending the DCEL Faces

- An instance of Arr_face_extended_dcel<Traits,FaceData> is a DCEL that extends the face record with the FaceData type.
- Data-fields must be maintained by the user application.
  - You can construct an arrangement, go over the faces, and store data in the appropriate face data-fields.
  - You can use an observer that receives updates whenever a face is modified and sets its data fields accordingly.




- ex_face_extension.cpp
- Assigns indeces to all faces in the order of creation.



## Extending the DCEL Faces (Cont.)

```
#include <CGAL/basic.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arr_observer.h>

#include "arr_exact_construction_segments.h"

typedef CGAL::Arr_face_extended_dcel<Traits_2, unsigned int>  Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>                   Ex_arrangement_2;

// An arrangement observer used to receive notifications of face splits and
// to update the indices of the newly created faces.
class Face_index_observer : public CGAL::Arr_observer<Ex_arrangement_2> {
private:
  unsigned int  n_faces;                // the current number of faces

public:
  Face_index_observer(Ex_arrangement_2& arr) :
    CGAL::Arr_observer<Ex_arrangement_2>(arr), n_faces(0)
  {
    CGAL_precondition(arr.is_empty());
    arr.unbounded_face()->set_data(0);
  }

  virtual void after_split_face(Face_handle old_face, Face_handle new_face, bool)
  {
    new_face->set_data(++n_faces);        // assign index to the new face
  }
};
```

## Extending all the DCEL Records

- An instance of

  `Arr_extended_dcel<Traits, VertexData, HalfedgeData, FaceData>`

  is a DCEL that extends the vertex, halfedge, and face records with the corresponding types.

```
enum Color {BLUE, RED, WHITE};

typedef CGAL::Arr_extended_dcel<Traits_2, Color, bool, unsigned int>     Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>                      Ex_arrangement_2;
```

```
  Ex_arrangement_2::Vertex_iterator          vit;
  for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
    unsigned int degree = vit->degree();
    vit->set_data((degree == 0) ? BLUE : ((degree <= 2) ? RED : WHITE));
  }
```

```
  std::cout << "The arrangement vertices:" << std::endl;
  for (vit = arr2.vertices_begin(); vit != arr2.vertices_end(); ++vit) {
    std::cout << '(' << vit->point() << ") - ";
    switch (vit->data()) {
      case BLUE  : std::cout << "BLUE."  << std::endl; break;
      case RED   : std::cout << "RED."   << std::endl; break;
      case WHITE : std::cout << "WHITE." << std::endl; break;
    }
  }
```



## Extending all the DCEL Records (Cont.)

```
#include <string>

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

/*! The map extended dcel vertex */
template <typename Point_2>
class Arr_map_vertex : public CGAL::Arr_vertex_base<Point_2> {
public:
  std::string name, type;
};

/*! The map extended dcel halfedge */
template <typename X_monotone_curve_2>
class Arr_map_halfedge : public CGAL::Arr_halfedge_base<X_monotone_curve_2> {
public:
  std::string name, type;
};

/*! The map extended dcel face */
class Arr_map_face : public CGAL::Arr_face_base {
public:
  std::string name, type;
};

/*! The map extended dcel */
template <typename Traits>
class Arr_map_dcel : public
  CGAL::Arr_dcel_base<Arr_map_vertex<typename Traits::Point_2>,
                      Arr_map_halfedge<typename Traits::X_monotone_curve_2>,
                      Arr_map_face>
{};
```

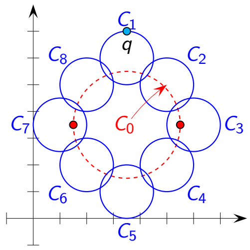


## Storing the Curve History

- $\mathcal{C}$ — a set of arbitrary planar curves.
- $\mathcal{C}'$ — the set of $x$-monotone subcurves.
- $\mathcal{C}''$ — the set of $x$-monotone subcurves that are pairwise disjoint in their interior.
  - They are associated with the (half)edges of an arrangement (object of type `Arrangement_2`).
- The connection between the subcurves in $\mathcal{C}$ and in $\mathcal{C}''$ is lost during the arrangement construction.
- `Arrangement_with_history_2<Traits,Dcel>` — a class template that extends `Arrangement_2` with:
  - a container that stores $\mathcal{C}$ and
  - a cross mapping between the curves in $\mathcal{C}$ and the halfedges they induce.
- The `Traits_2` template parameter must be substituted with a model of the *ArrangementTraits_2* concept.



## Edge Manipulation




- An arrangement of 9 circles.
- $C_0$ induces 18 edges.

`ex_edge_manipulation_curve_history.cpp`



## Edge Manipulation (Cont.)

```
#include <CGAL/basic.h>
#include <CGAL/Arrangement_with_history_2.h>

#include "arr_circular.h"
#include "arr_print.h"

typedef CGAL::Arrangement_with_history_2<Traits_2>              Arr_with_hist_2;
typedef Arr_with_hist_2::Curve_handle                           Curve_handle;

int main()
{
  // Construct an arrangement containing nine circles: C[0] of radius 2, the rest of radius
  const Number_type  _7_halves = Number_type(7) / Number_type(2);
  Curve_2            C[9];
  C[0] = Circle_2(Kernel::Point_2(_7_halves, _7_halves), 4, CGAL::CLOCKWISE);
  C[1] = Circle_2(Kernel::Point_2(_7_halves, 6), 1, CGAL::CLOCKWISE);
  C[2] = Circle_2(Kernel::Point_2(5, 5), 1, CGAL::CLOCKWISE);
  C[3] = Circle_2(Kernel::Point_2(6, _7_halves), 1, CGAL::CLOCKWISE);
  C[4] = Circle_2(Kernel::Point_2(5, 2), 1, CGAL::CLOCKWISE);
  C[5] = Circle_2(Kernel::Point_2(_7_halves, 1), 1, CGAL::CLOCKWISE);
  C[6] = Circle_2(Kernel::Point_2(2, 2), 1, CGAL::CLOCKWISE);
  C[7] = Circle_2(Kernel::Point_2(1, _7_halves), 1, CGAL::CLOCKWISE);
  C[8] = Circle_2(Kernel::Point_2(2, 5), 1, CGAL::CLOCKWISE);

  Arr_with_hist_2    arr;
  Curve_handle       handles[9];
  for (int k = 0; k < 9; k++) handles[k] = insert(arr, C[k]);

  // Remove the large circle C[0].
  std::cout << remove_curve(arr, handles[0]) << " edges removed." << std::endl;
  return 0;
}
```



## Map Overlay

**Definition (map overlay)**

The map overlay of two planar subdivisions $\mathcal{S}_1$ and $\mathcal{S}_2$, denoted as overlay$(\mathcal{S}_1, \mathcal{S}_2)$, is a planar subdivision $\mathcal{S}$, such that there is a face $f$ in $\mathcal{S}$ if and only if there are faces $f_1$ and $f_2$ in $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, such that $f$ is a maximal connected subset of $f_1 \cap f_2$.

The overlay of two subdivisions embedded on a surface in $\mathbb{R}^3$ is defined similarly.

$n_1$, $n_2$, $n$ — number of vertices in $\mathcal{S}_1$, $\mathcal{S}_2$, overlay$(\mathcal{S}_1, \mathcal{S}_2)$.

- Time complexities of the computation of the overlay of 2 subdivisions embedded on surfaces in $\mathbb{R}^3$:
  - Using sweep-line: $O((n)\log(n_1 + n_2))$. [BO79]
  - Using trapezoidal decomposition: $O(n)$. [FH95]
    - ★ Precondition: $\mathcal{S}_1$ and $\mathcal{S}_2$ are simply connected.

## Map Overlay of CGAL

```
template <typename TraitsRed, typename TraitsBlue, typename TraitsRes,
          typename DcelRed, typename DcelBlue, typename DcelRes,
          typename OverlayTraits>
void overlay (const Arrangement_2<TraitsRed, DcelRed> & arr1,
              const Arrangement_2<TraitsBlue, DcelBlue> & arr2,
              Arrangement_2<TraitsRes, DcelRes> & arr_res, OverlayTraits & ovl_tr)
```

The concept *OverlayTraits* requires the provision of ten functions that handle all possible cases as follows:

1. A new vertex $v$ is induced by coinciding vertices $v_r$ and $v_b$.
2. A new vertex $v$ is induced by a vertex $v_r$ that lies on an edge $e_b$.
3. An analogous case of a vertex $v_b$ that lies on an edge $e_r$.
4. A new vertex $v$ is induced by a vertex $v_r$ that is contained in a face $f_b$.
5. An analogous case of a vertex $v_b$ contained in a face $f_r$.
6. A new vertex $v$ is induced by the intersection of two edges $e_r$ and $e_b$.
7. A new edge $e$ is induced by the overlap of two edges $e_r$ and $e_b$.
8. A new edge $e$ is induced by the an edge $e_r$ that is contained in a face $f_b$.
9. An analogous case of an edge $e_b$ contained in a face $f_r$.
10. A new face $f$ is induced by the overlap of two faces $f_r$ and $f_b$.



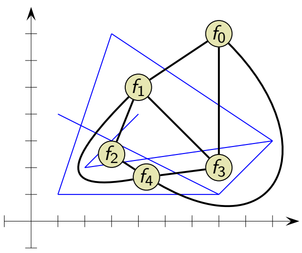
## The Primal Arrangement Representation

- BOOST provides a collection of free peer-reviewed portable C++ source libraries that work well with, and are in the same spirit as, the STL.
- The BOOST Graph Library (BGL) offers an extensive set of generic graph-algorithms, e.g., Dijkstra's shortest-path. [SLL02]
- Arrangement objects (the type of which are instances of `Arrangement_2`) are adapted as BOOST graphs by specializing the `boost::graph_traits<Graph>` class-template.
- The primal adaptation
  - Arrangement vertices and edges are adapted as BOOST graph vertices and edges, respectively.
  - `Graph` is substituted by `Arrangement_2`.
  - `boost::graph_traits` operations are implemented based on arrangement operations.



## Dual Arrangement Representations

- The Dual adaptation
  - Arrangement faces and edges are adapted as BOOST graph vertices and edges, respectively.
  - Two graph vertices are adjacent iff the corresponding arrangement faces share a common edge.




- The incidence-graph adaptation
  - Arrangement cells (vertices, edges, and faces) are adapted as BOOST graph vertices.
  - Every incidence relation between two cells of the arrangement is adapted as an edge of the BOOST graph.



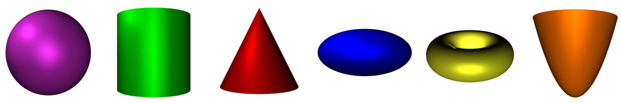
## Parametric Surfaces in $\mathbb{R}^3$

**Definition (Parametric surface)**

A *parametric surface* $S$ of two parameters is a surface defined by parametric equations involving two parameters $u$ and $v$:

$$f_S(u, v) = (x(u, v),\ y(u, v),\ z(u, v))$$

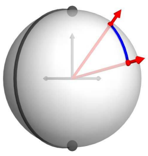
Thus, $f_S : \mathbb{P} \longrightarrow \mathbb{R}^3$ and $S = f_S(\mathbb{P})$, where $\mathbb{P}$ is a continuous and simply connected two-dimensional parameter space



- We deal with orientable parametric surfaces



## Geometry Traits of Geodesic Arcs on $\mathbb{S}^2$



- The point type is represented by
  - a direction in $\mathbb{R}^3$
  - an enumeration that indicates whether the point coincides with a contraction point or lie on an identification arc
- Each curve or $u$-monotone curve type is represented by
  - the source and target endpoints of type point
  - the normal of the plane that contains the 2 endpoint directions and the origin
    - ★ The plane orientation and the 2 endpoints determine which one of the two great arcs is considered
  - Boolean flags that cache geometric information
- This representation enables an exact yet efficient implementation of all geometric operations using exact rational arithmetic
  - Normalizing directions and plane normals is completely avoided



## Arrangement Bibliography I


- 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 11th 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*. 3.7 edition, 2010. `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


Jon Louis Bentley and Thomas Ottmann.
Algorithms for Reporting and Counting Geometric Intersections.
*IEEE Transactions on Computers.* 28(9):643–647, 1979.

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.




## Arrangement Bibliography IV


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.

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




## Outline



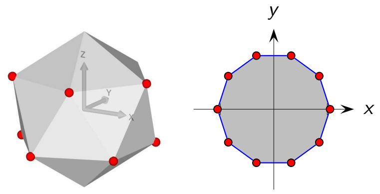


## **Application**: Obtaining Silhouettes of Polytopes

**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 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 $\mathcal{A}$ 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 $\mathcal{A}$
7.                 Insert the projection of $e$ into $\mathcal{A}$.

Computes the normal to a facet.

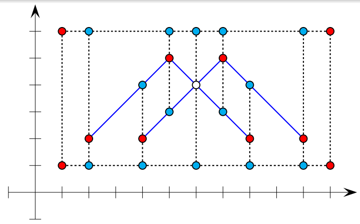
```
struct Normal_equation {
  template <typename Facet> typename Facet::Plane_3 operator()(Facet & f) {
    typename Facet::Halfedge_handle h = f.halfedge();
    return CGAL::cross_product(h->next()->vertex()->point() -
                               h->vertex()->point(),
                          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



## **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;
  }
}
```



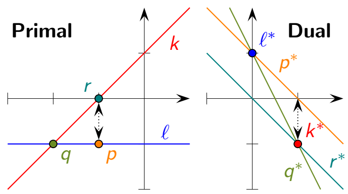
## Point-Line Duality Transform

- Points and lines are transformed into lines and points, respectively.

| **Primal Plane** | **Dual Plane** |
|---|---|
| the point $p : (a, b)$ | the line $p^* : y = ax - b$ |
| the line $l : y = cx + d$ | the point $l^* : (c, -d)$ |

- This duality transform does not handle vertical lines!

- The transform is incidence preserving.
- The transform preserves the above/below relation.
- The transform preserves the vertical distance between a point and a line.






## Point-Line Duality Code

```
template <typename Traits> typename Traits::Point_2
primal_point(const typename Traits::X_monotone_curve_2& cv)
{
  // If the supporting dual line of the linear curve is a*x + b*y + c = 0,
  // the primal point is (-a/b, c/b).
  const typename Traits::Line_2&  line = cv.supporting_line();
  CGAL_assertion(CGAL::sign(line.b()) != CGAL::ZERO);
  return typename Traits::Point_2(-line.a() / line.b(), line.c() / line.b());
}
```

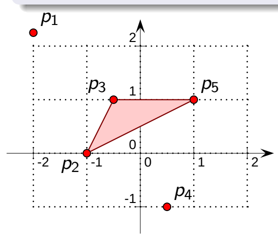
```
template <typename Traits> typename Traits::X_monotone_curve_2
dual_line(const typename Traits::Point_2& p)
{
  // The line dual to the point (p_x,p_y) is y = p_x*x-p_y (or p_x*x-y-p_y = 0).
  typename Traits::Line_2 line(p.x(), -1, -p.y());
  return typename Traits::X_monotone_curve_2(line);
}
```



## **Application**: Minimum-Area Triangle

**Application (Minimum-Area Triangle)**

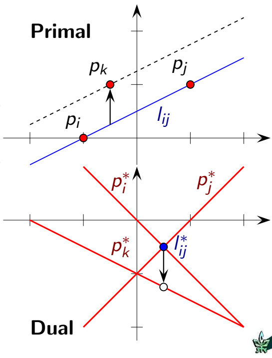
*Given a set $P = \{p_1, p_2, \ldots, p_n\}$ of $n$ points in the plane, find three distinct points $p_i, p_j, p_k \in P$ such that the area of the triangle $\triangle p_i p_j p_k$ is minimal among all other triangles defined by three distinct points in the set.*




- A naive algorithm requires $O(n^3)$ time.
- It is possible to compute in $O(n^2)$ time.
  - The analysis of the algorithm time-complexity uses the zone complexity theorem.

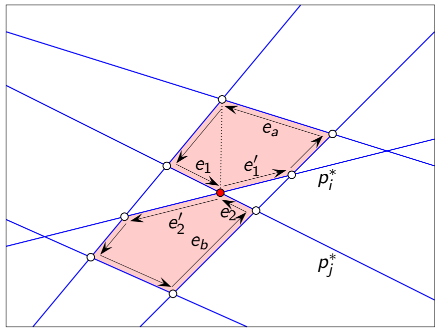
## Minimum Area Triangle: Duality

- $P^*$ — the set of lines dual to the input points.
  - $P^*$ does not contain vertical lines.
- $p_i^*, p_j^* \in P^*$
- $\ell_{ij}^*$ — the point of intersection between $p_i^*$ and $p_j^*$.
- $\ell_{ij}$ — the line that contains $p_i$ and $p_j$.
- $p_k$ — the point that defines the minimum-area triangle with $p_i$ and $p_j$.
- $p_k$ is the closest point to $\ell_{ij} \Longrightarrow$ $p_k$ is the closest point to $\ell_{ij}$ in the vertical distance.
- $p_k^*$ — the line immedialy above or below the point $\ell_{ij}^*$.






## Minimum Area Triangle: Details






## Minimum Area Triangle: Complexity

- Computing the bounding box
  - Naively $O(n^2)$.
  - Can be done in $O(n \log n)$.
- Finding where to insert line $i$
  - Simple, $O(i)$.
- Inserting line $i$.
  - $O(i)$ — zone construction.
- Searching for the minimum triangle
  - $O(i)$ — zone construction.
- Overall $O(n^2)$ time.



## Minimum Area Triangle: Notes

**Definition (Simplex)**

An *n*-simplex is the generalization of a tetrahedral region of space to $n$ dimensions.

The boundary of a $k$-simplex has $k+1$ 0-faces (polytope vertices), $k(k+1)/2$ 1-faces (polytope edges), and $\binom{k+1}{i+1}$ $i$-faces

- The solution to the minimum-area-triangle problem [CGL85]
- The solution for any fixed dimension (minimum volume simplex) [Ede87]
- The efficiency of the solution in any dimension relies on a hyperplane zone theorem [ESS93]
- No better solution is known to the problem; related to the so-called 3-sum hard problems.
- See also Incremental construction of arrangements of lines [BKO+08, Chapter 8]

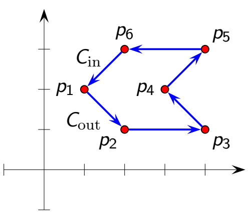


## **Application**: Polygon Orientation

**Application (Polygon Orientation)**

*Given a sequence of $x$-monotone segments that compose a closed curve, which represents the boundary of a point set, determine whether the orientation of the boundary of a point set is clockwise or counterclockwise.*

- $P$ — an input polygon.
- Search for the smallest point $p$ on the boundary of $P$.
- $p$ is incident to 2 curve segments $C_{\text{in}}$ and $C_{\text{out}}$, which lie to the right of $p$.
- Compare the $y$-coordinate of $C_{\text{in}}$ and $C_{\text{out}}$ immediately to the right of $p$.




- $p$ is not the smallest $\Longrightarrow$ wrong conclusion, even if the curve segments both lie to its right; see $p_4$.

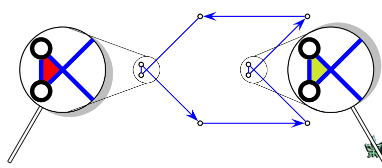


## **Application**: Polygon Repairing

**Application (Polygon Repairing)**

*Given a sequence of $x$-monotone segments that compose a closed curve, which represents the boundary of a point set, subdivide the point set into as few as possible simple point-sets, each bounded by a counterclockwise-oriented boundary comprising $x$-monotone segments that are pairwise disjoint in their interior.*

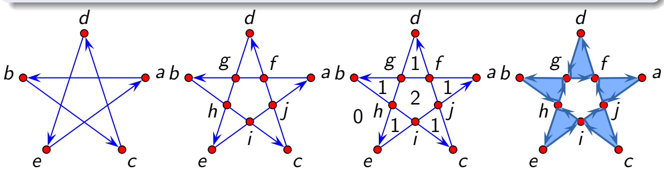
- We need to convert invalid point-sets to point sets the interiors and exteriors of which are not well defined.
- Including the green small triangle and excluding the red small triangle is a good guess.



## Winding Numbers

### Definition (Winding number)

The winding number of a point is the number of counterclockwise cycles the oriented boundary makes around the point.




1. A self-crossing polygon given by $\{a, b, c, d, e\}$.
2. The arrangement data-structure constructed from the polygon edges.
3. The arrangement data-structure with updated face winding-numbers.
4. The resulting polygons considering only faces with odd winding numbers.



## Winding Numbers (Cont.)

```
#include <utility>

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

template <typename Arrangement> class Winding_number {
private:
  Arrangement& _arr;
  typename Arrangement::Traits_2::Compare_endpoints_xy_2 _cmp_endpoints;

  // The Boolean flag indicates whether the face has been discovered already
  // during the traversal. The integral field stores the winding number.
  typedef std::pair<bool, int>                              Data;

public:
  Winding_number(Arrangement& arr) : _arr(arr)
  {
    // Initialize the winding numbers of all faces.
    typename Arrangement::Face_iterator fi;
    for (fi = _arr.faces_begin(); fi != _arr.faces_end(); ++fi)
      fi->set_data(Data(false, 0));
    _cmp_endpoints = _arr.traits()->compare_endpoints_xy_2_object();
    propagate_face(_arr.unbounded_face(), 0);   // compute the winding numbers
  }
```



## Winding Numbers (Cont.)

```
private:
  // Count the net change to the winding number when crossing a halfedge.
  int count(typename Arrangement::Halfedge_handle he)
  {
    bool l2r = he->direction() == CGAL::ARR_LEFT_TO_RIGHT;
    typename Arrangement::Originating_curve_iterator ocit;
    int num = 0;
    for (ocit = _arr.originating_curves_begin(he);
         ocit != _arr.originating_curves_end(he); ++ocit)
      (l2r == (_cmp_endpoints(*ocit) == CGAL::SMALLER)) ? ++num : --num;
    return num;
  }
```



## Winding Numbers (Cont.)

```
private:
  // Traverse all faces neighboring the given face and compute their
  // winding numbers of all faces while traversing the arrangement.
  void propagate_face(typename Arrangement::Face_handle fh, int num)
  {
    if (fh->data().first) return;
    fh->set_data(Data(true, num));

    // Traverse the inner boundary (holes).
    typename Arrangement::Hole_iterator hit;
    for (hit = fh->holes_begin(); hit != fh->holes_end(); ++hit) {
      typename Arrangement::Ccb_halfedge_circulator cch = *hit;
      do {
        typename Arrangement::Face_handle inner_face = cch->twin()->face();
        if (inner_face == cch->face()) continue;          // discard antenas
        propagate_face(inner_face, num + count(cch->twin()));
      } while (++cch != *hit);
    }

    // Traverse the outer boundary.
    if (fh->is_unbounded()) return;
    typename Arrangement::Ccb_halfedge_circulator cco = fh->outer_ccb();
    do {
      typename Arrangement::Face_handle outer_face = cco->twin()->face();
      propagate_face(outer_face, num + count(cco->twin()));
    } while (++cco != fh->outer_ccb());
  }
};
```

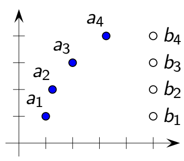


## **Application**: Largest Common Point-Set Under $\epsilon$-Congruence

### Application (Largest Common Point-Set Under $\epsilon$-Congruence)

*Given two point sets $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_n\}$ in the plane, and a real parameter $\epsilon > 0$, find a translation $T$ and two maximum subsets $\{i_1, \ldots, i_M\} \subseteq \{1, \ldots, m\}$ and $\{j_1, \ldots, j_M\} \subseteq \{1, \ldots, n\}$, such that $\|T(a_{i_k}) - b_{j_k}\| < \epsilon$ for each $1 \le k \le M$.*

- $\epsilon = 0.1 \Longrightarrow \{\langle 1,1\rangle\}$
- $\epsilon = 0.25 \Longrightarrow \{\langle 1,1\rangle, \langle 2,2\rangle\}$
- $\epsilon = 1 \Longrightarrow \{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle\}$
- $\epsilon = 2 \Longrightarrow \{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle, \langle 4,4\rangle\}$



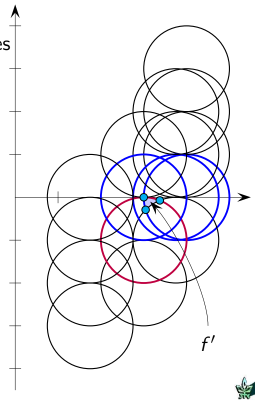



## Maximum (Cardinality) Bipartite Matching

- The BGL function `boost::edmonds_maximum_cardinality_matching()` computes the maximum-cardinality matching for general graphs.
- The implementation closely follows Tarjan's description of Edmonds' algorithm. [Edm65][Tar83, Chapter 9]
- $m$ and $n$ — The number of edges and vertices in the input graph.
- It runs in $O(mn\alpha(m, n))$ time.
- Edmonds' algorithm has been improved to run in $O(\sqrt{n}m)$ time, matching the time for maximum bipartite matching. [MV80]
- Another algorithm by Mucha and Sankowski runs in $O(V^{2.376})$ time. [MS04]

## LCP: The Arrangement of Circles

- The translation space for $\epsilon = 1$.
- The arrangement is induced by 16 circles $C_{ij} = \{p \in \mathbb{R}^2 \mid \|p - (b_j - a_i)\| = \epsilon\}$.
- $f'$ is covered by 3 discs among other, which are associated with the pairs of indices, respectively, that compose the solution $\{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle\}$.
- There are faces in the arrangement covered by as many as 5 discs.
- We are looking for a maximum bipartite matching.






## LCP: Code Samples

```
unsigned int bipartite_matching(unsigned int m, unsigned int n,
                                const Index_pair_set& index_pair_set,
                                Index_pair_set* match_set = NULL)
{
  typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::undirectedS> Graph;

  // Create a graph with (m + n) vertices.
  Graph G(m + n);

  Index_pair_set::const_iterator it;
  for (it = index_pair_set.begin(); it != index_pair_set.end(); ++it)
    boost::add_edge(it->first, it->second + m, G);
  std::vector<boost::graph_traits<Graph>::vertex_descriptor> mate(m + n);
  boost::edmonds_maximum_cardinality_matching(G, &mate[0]);
  unsigned int match_size = boost::matching_size(G, &mate[0]);
  .
  .
  .
  return match_size;
}
```

```
  // Perform breadth-first search from the unbounded face. Use the event
  // visitor to associate each arrangement face with its index-pair set.
  Ex_arrangement_2::Face_handle                 uf = arr.unbounded_face();
  boost::breadth_first_search(Dual_arrangement_2(arr), uf,
                              boost::vertex_index_map(index_map).
                              visitor(boost::make_bfs_visitor(Index_pair_set_visitor())));
```



## Minkowski Sum Definition

Hermann Minkowski
1864–1909

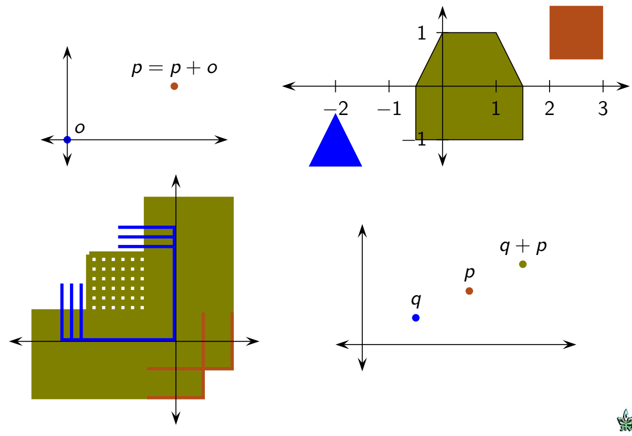
**Definition (Minkowski sum)**

Let $P$ and $Q$ be two point sets in $\mathbb{R}^d$. The Minkowski sum of $P$ and $Q$, denoted as $P \oplus Q$, is the point set $\{p + q \mid p \in P, q \in Q\}$.

- Applies to every dimension $d$.
- Applies to arbitrary point sets.

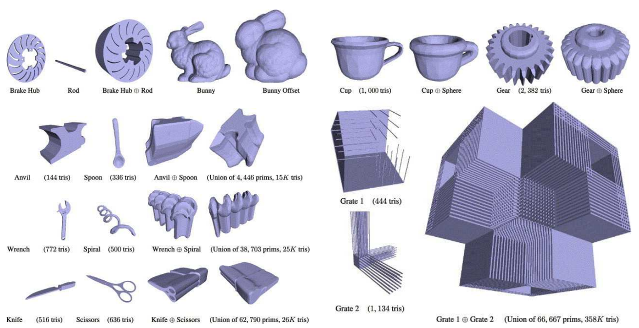


## Minkowski Sum Examples in $\mathbb{R}^2$






## Minkowski Sum Examples in $\mathbb{R}^3$



[VM06]



## Minkowski Sum Properties

- The Minkowski sum of two (non-parallel) line segments in $\mathbb{R}^2$ is a convex polygon.
- The Minkowski sum of two (non-parallel) polygons in $\mathbb{R}^3$ is a convex polyhedron.
- $P = P \oplus \{o\}$, where $o$ is the origin.
- If $P$ and $Q$ are convex, then $P \oplus Q$ is convex.
- $P \oplus Q = Q \oplus P$.
- $\lambda(P \oplus Q) = \lambda P \oplus \lambda Q$, where $\lambda P = \{\lambda p \mid p \in P\}$.
- $2P \subseteq P \oplus P$, $3P \subseteq P \oplus P \oplus P$, etc.
- $P \oplus (Q \cup R) = (P \oplus Q) \cup (P \oplus R)$.

## Minkowski Sum Construction Approaches

- Decomposition
  - Decompose $P$ and $Q$ into convex sub-polygons $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_\ell$
    - $\bigcup_{i=1}^{k} P_i = P$ and $\bigcup_{j=1}^{\ell} Q_j = Q$.
  - Calculate the pairwise sums $S_{ij} = P_i \oplus Q_j$ of the convex sub-polygons.
  - Compute the union $P \oplus Q = \bigcup_{ij} S_{ij}$.
- Convolution, denoted $P \otimes Q$. Description is deferred.



## Minkowski Sum Construction: Decomposition

1. Decompose $P$ and $Q$ into convex sub-polygons $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_\ell$
2. Calculate the pairwise sums $S_{ij} = P_i \oplus Q_j$ of the convex sub-polygons.
3. Compute the union $P \oplus Q = \bigcup_{ij} S_{ij}$.

- Issues
  - Which union strategy.
  - How to handle degeneracies.
  - Which decomposition algorithm.
- Addressing of the issues is based on emprical results.
- The oddity of computing the union.



## The Union of Many Polygons

**Definition (3SUM)**

3SUM is the following computational problem conjectured to require roughly quadratic time: Given a set $S$ of $n$ integers, are there elements $a, b, c \in S$ such that $a + b + c = 0$?

- A problem is called 3SUM-hard if solving it in subquadratic time implies a subquadratic-time algorithm for 3SUM.
- Computing the union of polygons is 3SUM-hard.

Union Algorithms:

- Aggregate.
- Incremental.
- Divide-and-conquer.



## Divide and Conquer Union Algorithm

**Definition (Divide and Conquer)**

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type.

- $\mathcal{P}$ — a set of $n$ polygons.
- Compute the union of each pair of polygons in $\mathcal{P}$ to yield $n/2$ polygons.
  - Use the arrangement-union algorithm.
- Repeat recursively $\log n$ times.



## Minkowski Sum Construction: Convex Decomposition

- No Steiner points
  - Minimize the number of convex sub-polygons, $O(n^4)$ time, $O(n^3)$ space. [Gre83]
  - Approximate the minimum number of convex sub-polygons, $O(n)$ after triangulation. [HM85]
  - Approximate the minimum number of convex sub-polygons, $O(n \log n)$ time, $O(n)$ space. [Gre83]
  - Small side angle-bisector decomposition. [AFH02]
- Triangulation
  - Naive triangulation.
  - Minimizing the maximum degree triangulation.
  - Minimizing $\Sigma d_i^2$ triangulation.
- Allowing Steiner points
  - Slab decomposition.
  - Angle-bisector decomposition. [DC85]
  - KD decomposition.

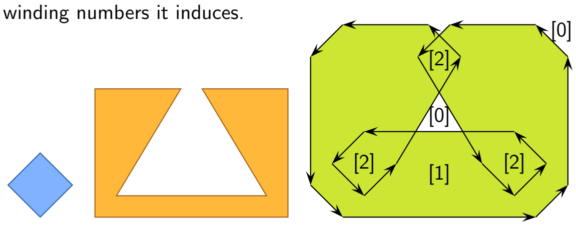


## Minkowski Sum Construction: Convolution

- $P$, $Q$ — polygons with vertices $(p_0, \ldots, p_{m-1})$ and $(q_0, \ldots, q_{n-1})$.
  - $P$ and $Q$ have positive orientations
- $P \otimes Q$ — the convolution of $P$ and $Q$ is a collection of line segments:
  - $[p_i + q_j, p_{i+1} + q_j]$, where $\overrightarrow{p_i p_{i+1}}$ lies between $\overrightarrow{q_{j-1} q_j}$ and $\overrightarrow{q_j q_{j+1}}$.
  - $[p_i + q_j, p_i + q_{j+1}]$, where $\overrightarrow{q_j q_{j+1}}$ lies between $\overrightarrow{p_{i-1} p_i}$ and $\overrightarrow{p_i p_{i+1}}$.
- The segments of the convolution form a number of closed polygonal curves called convolution cycles.
  - $P$ (or $Q$) is convex $\Longrightarrow$ 1 convolution cycle.
- The Minkowski sum $P \oplus Q$ is the set of points having a non-zero winding number in the arrangement of convolution cycles.

## Minkowski Sum Construction: Convolution Efficiency

The convolution cycle of a convex polygon and a non-convex polygon and the winding numbers it induces.




- The number of segments in the convolution is usually smaller than the number of segments of the sub-sums of the decomposition.
- Both approaches construct the arrangement of these segments and extract the sum from this arrangement.
- Computing Minkowski sums using the convolution approach usually generates a smaller intermediate arrangement.
- The convolution approach is faster and consumes less space.



## Polytope Definition

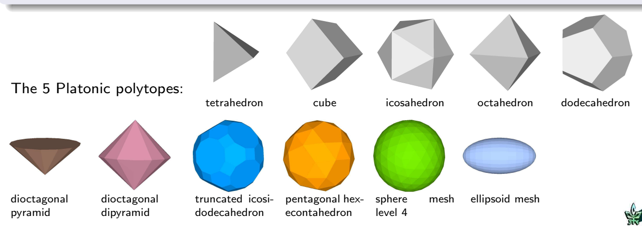
**Definition (convex polyhedron)**

A convex set $Q \subseteq \mathbb{R}^d$ given as an intersection of finite number of closed half-spaces $H = \{h \in \mathbb{R}^d \mid Ah \leq B\}$ is called convex polyhedron.

**Definition (polytope)**

A bounded polyhedron $P \subset \mathbb{R}^d$ is called polytope.

The 5 Platonic polytopes:






## Minkowski-Sum Construction: Convex Hull

**Observation**

*The Minkowski sum of two convex polytopes $P$ and $Q$ is the convex hull of the pairwise sums of vertices of $P$ and $Q$, respectively.*

```
typedef CGAL::Exact_predicates_exact_constructions_kernel   Kernel;
typedef Kernel::Point_3                                     Point;
typedef Kernel::Vector_3                                    Vector;
typedef CGAL::Polyhedron_3<Kernel>                          Polyhedron;

  std::vector<Point> in1, in2, points;
  Process input ...
  points.resize(in1.size() * in2.size());
  std::vector<Point>::const_iterator it1, it2;
  std::vector<Point>::iterator it3 = points.begin();
  for (it1 = in1.begin(); it1 != in1.end(); ++it1) \{
    Vector v(CGAL::ORIGIN, *it1);
    for (it2 = in2.begin(); it2 != in2.end(); ++it2) *it3++ = (*it2) + v;
  }
  Polyhedron polyhedron;
  CGAL::convex_hull_3(points.begin(), points.end(), polyhedron);
```

- `CGAL::convex_hull_3` implements QuickHull.
- Time complexities of Minkowski-sum constr. using convex hull:
  - Using `CGAL::convex_hull_3` (expected): $O(mn \log mn)$.
  - Optimal: $O(mn \log h)$.

code

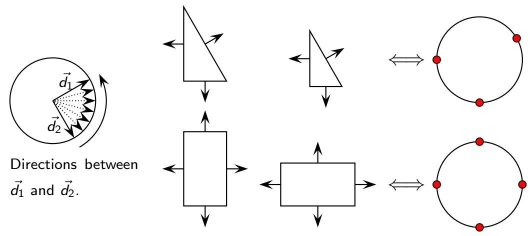


## Gasusian Map (Normal Diagram) in 2D

Carl Friedrich Gauss 1777–1855

**Definition (Gasusian map or normal diagram)**

The Gaussian map of a convex polygon $P$ is the decomposition of $\mathbb{S}$ into maximal connected arcs so that the extremal point of $P$ is the same for all directions within one region.




- Generalizes to higher dimensions.



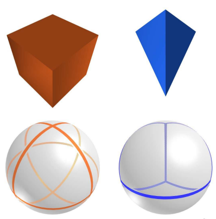
## Gasusian Map (normal diagram) in 3D

**Definition (Gasusian map (normal diagram) in 3D)**

The Gaussian map of a convex polytope in $\mathbb{R}^3$ $P$ is the decomposition of $\mathbb{S}^2$ into maximal connected regions so that the extremal point of $P$ is the same for all directions within one region.

$G$ is a set-valued function from $\partial P$ to $\mathbb{S}^2$. $G(p \in \partial P) =$ the set of outward unit normals to support planes to $P$ at $p$.

- $v, e, f$ — a vertex, an edge, a facet of $P$.
- $G(f) =$ outward unit normal to $f$.
- $G(e) =$ geodesic segment.
- $G(v) =$ spherical polygon.
- $G(P)$ is an arrangement on $\mathbb{S}^2$.
- $G(P)$ is unique $\Rightarrow G^{-1}(G(P)) = P$.



- Each face $G(v)$ of the arrangement is extended with $v$.



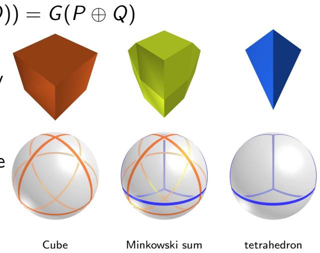
## Minkowski Sums Construction: Gaussian Map

**Observation**

*The overlay of the Gaussian maps of two convex polytopes $P$ and $Q$ is the Gaussian map of the Minkowski sum of $P$ and $Q$.*

$$\text{overlay}(G(P), G(Q)) = G(P \oplus Q)$$

- The overlay identifies all the pairs of features of $P$ and $Q$ respectively that have common supporting planes.
- These common features occupy the same space on $\mathbb{S}^2$.
- They identify the pairwise features that contribute to $\partial(P \oplus Q)$.



## Minkowski-Sums Construction: Gaussian Map

$m, n, k$ — number of facets in $P$, $Q$, $P \oplus Q$.

- Overlay of CGAL is based on sweep-line.
- $G(P)$ is a simply connected convex subdivision.
- Time complexities of Minkowski-sum constr. using Gaussian map:
  - Using `CGAL::overlay`: $O(k \log(m+n))$.
  - Optimal: $O(k)$ [FH95].



## The `Minkowski_sum_2` Package

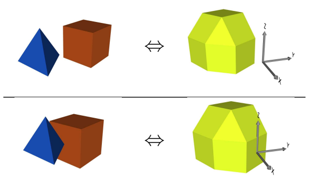
- Based on the `Arrangement_2`, `Polygon_2`, and `Partition_2` packages
- Works well with the `Boolean_set_operations_2` package
  - e.g., It is possible to compute the union of offset polygons
- Robust and efficient
- Supports Minkowski sums of two simple polygons
  - Implemented using either decomposition or convolution
  - Exact
- Supports Minkowski sums of a simple polygon and a disc (polygon offseting)
  - Offers either an exact computation or a conservative approximation scheme



## Minkowski Sum Application: Collision Detection

- $P$ and $Q$ are two polytopes in $\mathbb{R}^d$.
- $P$ translated by a vector $t$ is denoted by $P^t$.

$$P \cap Q \neq \emptyset \Leftrightarrow \text{Origin} \in M = P \oplus (-Q) \quad \text{collision detection}$$
$$\pi(P, Q) = \min\{\|t\| \mid P^t \cap Q \neq \emptyset, t \in \mathbb{R}^d\} \quad \text{separation distance}$$
$$\delta(P, Q) = \inf\{\|t\| \mid P^t \cap Q = \emptyset, t \in \mathbb{R}^d\} \quad \text{penetration depth}$$
$$\delta_v(P, Q) = \inf\{\alpha \mid P^{\alpha\vec{v}} \cap Q = \emptyset, \alpha \in \mathbb{R}\} \quad \text{directional penetration-depth}$$

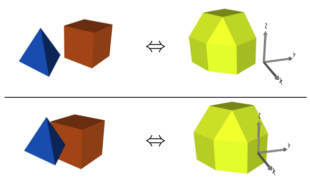




## Minkowski Sum Application: Collision Detection

- $P$ and $Q$ are two polytopes in $\mathbb{R}^d$.
- $P$ translated by a vector $t$ is denoted by $P^t$.

$$P^u \cap Q^w \neq \emptyset \Leftrightarrow w - u \in M = P \oplus (-Q) \quad \text{collision detection}$$
$$\pi(P, Q) = \min\{\|t\| \mid t \in M, t \in \mathbb{R}^d\} \quad \text{separation distance}$$
$$\delta(P, Q) = \inf\{\|t\| \mid t \notin M, t \in \mathbb{R}^d\} \quad \text{penetration depth}$$
$$\delta_v(P, Q) = \inf\{\alpha \mid \alpha\vec{v} \notin M, \alpha \in \mathbb{R}\} \quad \text{directional penetration-depth}$$

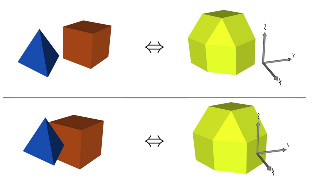




## Minkowski Sum Application: Collision Detection

- $P$ and $Q$ are two polytopes in $\mathbb{R}^d$.
- $P$ translated by a vector $t$ is denoted by $P^t$.

$$P^u \cap Q^w \neq \emptyset \Leftrightarrow w - u \in M = P \oplus (-Q) \quad \text{collision detection}$$
$$\pi(P^u, Q^w) = \min\{\|t\| \mid (w - u + t) \in M, t \in \mathbb{R}^d\} \quad \text{separation distance}$$
$$\delta(P^u, Q^w) = \inf\{\|t\| \mid (w - u + t) \notin M, t \in \mathbb{R}^d\} \quad \text{penetration depth}$$
$$\delta_v(P^u, Q^w) = \inf\{\alpha \mid (w - u + \alpha\vec{v}) \notin M, \alpha \in \mathbb{R}\} \quad \text{directional penetration-depth}$$





## Minkowski Sum Application: Width

**Definition (point-set width)**

The width of a set of points $P \subseteq \mathbb{R}^d$, denoted as width$(P)$, is the minimum distance between parallel hyperplanes supporting conv$(P)$.

**Definition (directional point-set width)**

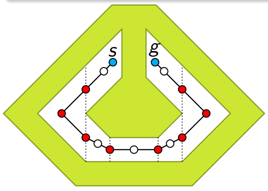
Given a normalized vector $v$, the directional width, denoted as width$_v(P)$ is the distance between parallel hyperplanes supporting conv$(P)$ and orthogonal to $v$.

- $width(P) = \delta(P, P) = \inf\{\|t\| \mid t \notin (P \oplus -P), t \in \mathbb{R}^d\}$
- Time complexities of width computation in $\mathbb{R}^3$:
  - Applied computation using CGAL Minkowski sum: $O(k \log n)$.
  - Optimal computation using Minkowski sum: $O(k)$.
  - CGAL::Width_3: $O(n^2)$. [FGHHS08]
  - Width optimal computation complexity: subquadratic.

## Motion Planning: A Translating Polygonal Robot

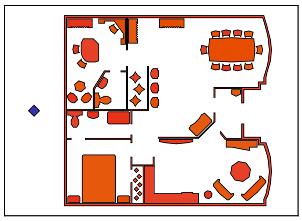
**Application (A Translating Polygonal Robot)**

*Given a simple-polygon robot $Q$, which can translate (but not rotate) in a room cluttered with pairwise interior-disjoint polygonal obstacles, devise a data structure that can efficiently answer queries of the following form: Given a start position $s$ of some reference point in $Q$ and a goal position $g$ of the same reference point, plan a collision-free path of the robot from $s$ to $g$.*



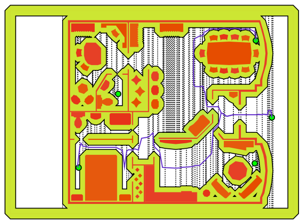



## Motion Planning: A Translating Polygonal Robot (cont.)



The workspace:

- a diamond-shaped robot translating in a house amidst polygonal obstacles



The configuration space:

- the forbidden-configuration space,
- the free-configuration space decomposed into trapezoidal faces
- the queries, and the resulting paths, if exist.



## Movie: Arrangements of Geodesic Arcs on the Sphere

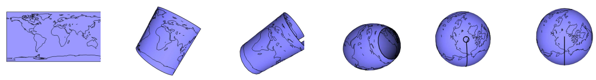


## Movies


Eyal Flato, Efi Fogel, Dan Halperin, and Eyal Leiserowitz.
Movie: Exact Minkowski Sums and Applications.
In *Proceedings of $18^{th}$ Annual ACM Symposium on Computational Geometry (SoCG)*, pages 273–274. Association for Computing Machinery (ACM) Press, 2005.

Efi Fogel and Dan Halperin.
Movie: Exact Minkowski sums of convex polyhedra.
In *Proceedings of $21^{st}$ Annual ACM Symposium on Computational Geometry (SoCG)*, pages 382–383. Association for Computing Machinery (ACM) Press, 2005.

Efi Fogel, Ophir Setter, and Dan Halperin.
Movie: Arrangements of Geodesic Arcs on the Sphere.
In *Proceedings of $24^{th}$ Annual ACM Symposium on Computational Geometry (SoCG)*, pages 218–219. Association for Computing Machinery (ACM) Press, 2008.



## Minkowski-Sum Bibliography I


Efi Fogel and Dan Halperin.
Exact and efficient construction of Minkowski sums of convex polyhedra with applications.
*Computer-Aided Design*, 39(11):929–940, 2007.

Efi Fogel, Dan Halperin, and Christophe Weibel.
On the exact maximum complexity of Minkowski sums of convex polyhedra.
*Discrete & Computational Geometry*, 42(4):654–669, 2009.

Komei Fukuda.
From the zonotope construction to the Minkowski addition of convex polytopes.
*Journal of Symbolic Computation*, 38(4):1261–1272, 2004.

Peter Hachenberger, Lutz Kettner, and Kurt Mehlhorn.
Boolean operations on 3D selective Nef complexes: Data structure, algorithms, optimized implementation and experiments.
*Computational Geometry: Theory and Applications*, 38(1-2):64–99, 2007.
Special issue on CGAL.

Gokul Varadhan and Dinesh Manocha.
Accurate Minkowski sum approximation of polyhedral models.
*Graphical Models and Image Processing*, 68(4):343–355, 2006.

Kaspar Fischer, Bernd Gärtner, Thomas Herrmann, Michael Hoffmann, and Sven Schönherr.
Optimal Distances.
In CGAL Editorial Board, editor, CGAL *User and Reference Manual*. 3.7 edition, 2008. `http://www.cgal.org/Manual/latest/doc_html/cgal_manual/packages.html#Pkg:Pkg:OptimalDistances`.

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.




## Minkowski-Sum Bibliography II


Bernard Chazelle.
Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm.
*SIAM Journal on Computing*, 13:488–507, 1984.

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.

Franco P. Preparata and Michael Ian Shamos
*Computational Geometry: An Introduction*.
Springer, New York, 1990.

Daniel H. Greene
The Decomposition of Polygons into Convex Parts.
*Computational Geometry*, Advances in Computing Research, 1:235–259, 1983.

Stefan Hertel and Kurt Mehlhorn
Fast triangulation of the plane with respect to simple polygons.
*Information and Control*, 64:52–76,1985.

Bernard Chazelle and David P. Dobkin.
Optimal convex decompositions.
*Computational Geometry*, North-Holland, pages 63–133, 1985

Pankaj Kumar Agarwal and Eyal Flato and Dan Halperin.
Polygon Decomposition for Efficient Construction of Minkowski Sums.
*Computational Geometry: Theory and Applications*, 21:39–61, 2002.