Algorithm Library Design

at the Example of two Geometric Algorithms Libraries

Computational Geometry
Algorithms Library

Efficient and Exact Algorithms
for Curves and Surfaces

Lutz Kettner
MPI für Informatik, Saarbrücken

April 8, 2005

Contents

- **Why is Geometric Computing Hard?**
- History and Overview of CGAL
- Overview of EXACUS
- Design of CGAL (and EXACUS)

Why is Geometric Computing Hard?

- The algorithms of computational geometry are designed for a machine model with exact real arithmetic.

  Substituting floating point arithmetic for the assumed real arithmetic may cause implementations to fail.

- Degeneracy handling

- Inherently complex geometric algorithms

Example

- Arrangements of circular arcs

Seen in Antibes, France, during SGP'04

Example

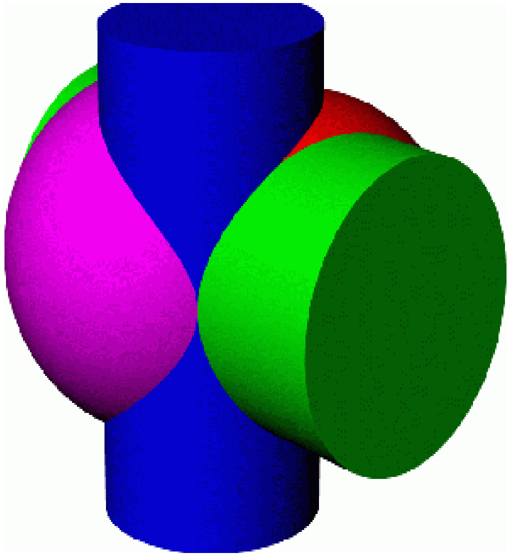
- Arrangements of circular arcs, with **double** arithmetic

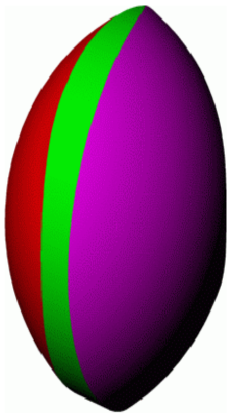
Example

- Arrangements of circular arcs, with **float** arithmetic

- Let’s be more serious …

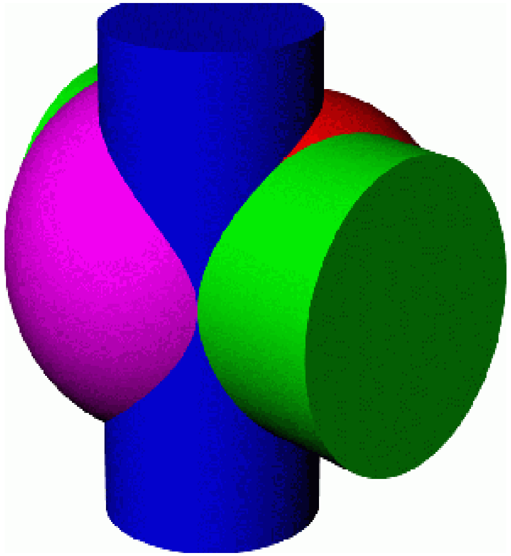
Intersection of Four Simple Solids

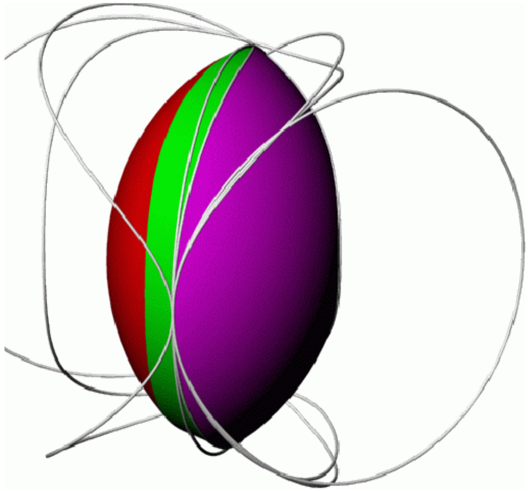




- Rhino3D:
  - $(((s_1 \cap s_2) \cap c_2) \cap c_1) \rightarrow$ **successful**
  - $(((c_1 \cap c_2) \cap s_1) \cap s_2) \rightarrow$ **``Boolean operation failed"**

Intersection of Four Simple Solids





- output is a combinatorial object plus coordinates (not a point set)

- Rhino3D:
  - $(((s_1 \cap s_2) \cap c_2) \cap c_1) \rightarrow$ **successful**
  - $(((c_1 \cap c_2) \cap s_1) \cap s_2) \rightarrow$ ``**Boolean operation failed**"

- Let’s be more serious …
- …, but that example is hard to study and explain

- Let’s do a really simple 2D convex hull algorithm.

Global Consequences

- A point outside sees a non-contig

p_1=(24.00000000000005, 24.000000000000053)

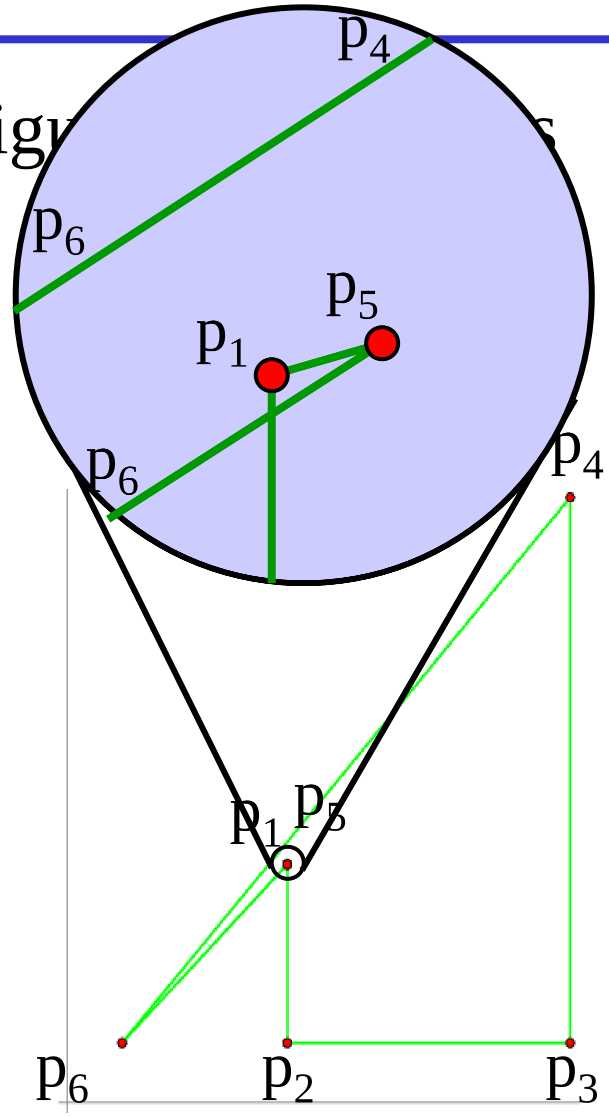
p_2=(24.0, **6.0**)

p_3=(54.85, 6.0)

p_4= (54.850000000000357, 61.000000000000121)

p_5=(24.000000000000068, 24.000000000000071)

p_6=(6, 6)



2D-Orientation of Three Points

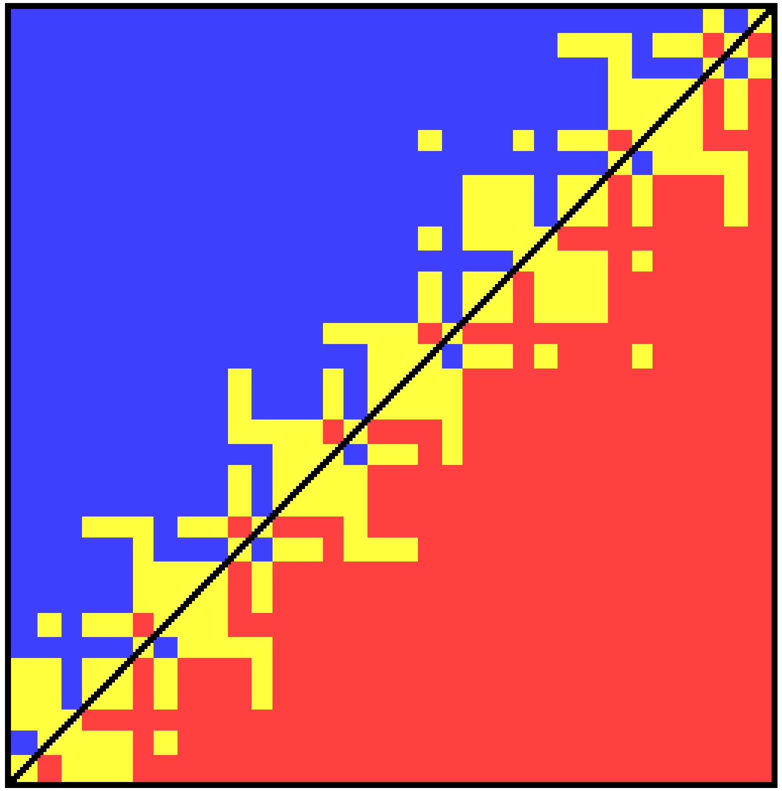
Orientation(p, q, r) = sign($(q_x\text{-}p_x)(r_y\text{-}p_y) - (q_y\text{-}p_y)(r_x\text{-}p_x)$)

$$p : \begin{pmatrix} 0.5 + x \cdot u \\ 0.5 + y \cdot u \end{pmatrix}$$

$$q : \begin{pmatrix} 12 \\ 12 \end{pmatrix}$$

$$r : \begin{pmatrix} 24 \\ 24 \end{pmatrix}$$

$$0 \le x, y < 256, u = 2^{-53}$$



256x256 pixel image

red=pos., yellow=0, blue=neg.

orientation evaluated with `double`

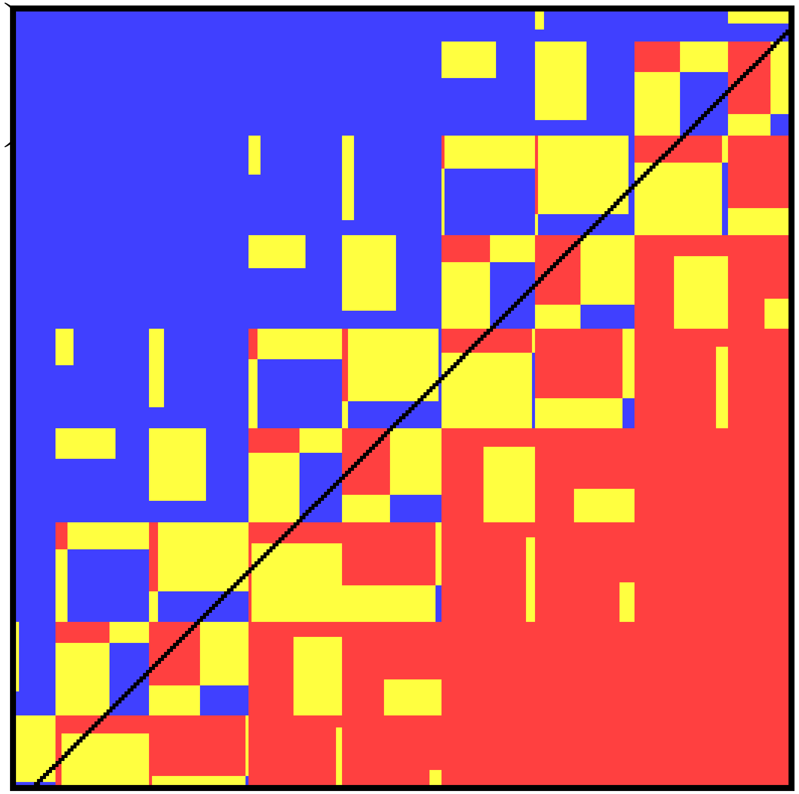
2D-Orientation of Three Points

Orientation(p, q, r) = sign((q_x-p_x)(r_y-p_y) – (q_y-p_y)(r_x-p_x))

$$p: \begin{pmatrix} 0.5000000000\,0002531 + x \cdot u \\ 0.5000000000\,0001710 + y \cdot u \end{pmatrix}$$

$$q: \begin{pmatrix} 17.3000000000\,00001 \\ 17.3000000000\,00001 \end{pmatrix}$$

$$r: \begin{pmatrix} 24.0000000000\,000500000 \\ 24.0000000000\,000517765 \end{pmatrix}$$



orientation evaluated with

`ext double`

What can be done?

- Redesign algorithms for FP arithmetic
  - works for some problems, no general theory
- Exact arithmetic
  - long integers, rationals, *k*-th roots, algebraic numbers
- FP filter for efficiency (interval arithmetic)
  - error bounds: static, semi-static, dynamic

- Type of arithmetic and filter depends on the application
  ➔Flexibility with generic programming and C++ templates

Exact Geometric Computing Paradigm

- Algorithms are expressed in terms of **geometric objects** and **primitive operations**.
- The correctness of the algorithm is derived from the correctness of the primitive operations.
- Primitive operations are **predicates** or **constructions**.
- FP filters are easier to realize for predicates.
- Filtered constructions are possible (lazy on demand).

Contents

- Why is Geometric Computing Hard?
- **History and Overview of CGAL**
- Overview of EXACUS
- Design of CGAL (and EXACUS)

Motivation

Computational Geometry Impact Task Force Report 1996
"*Application Challenges to Computational Geometry*" had four key recommendations:

1. Production and distr. of usable (and useful) geometric codes
2. Interdisciplinary forums
3. Experimentation
4. Reward structure for implementations in academia

Computational Geometry Algorithms Library

www.cgal.org

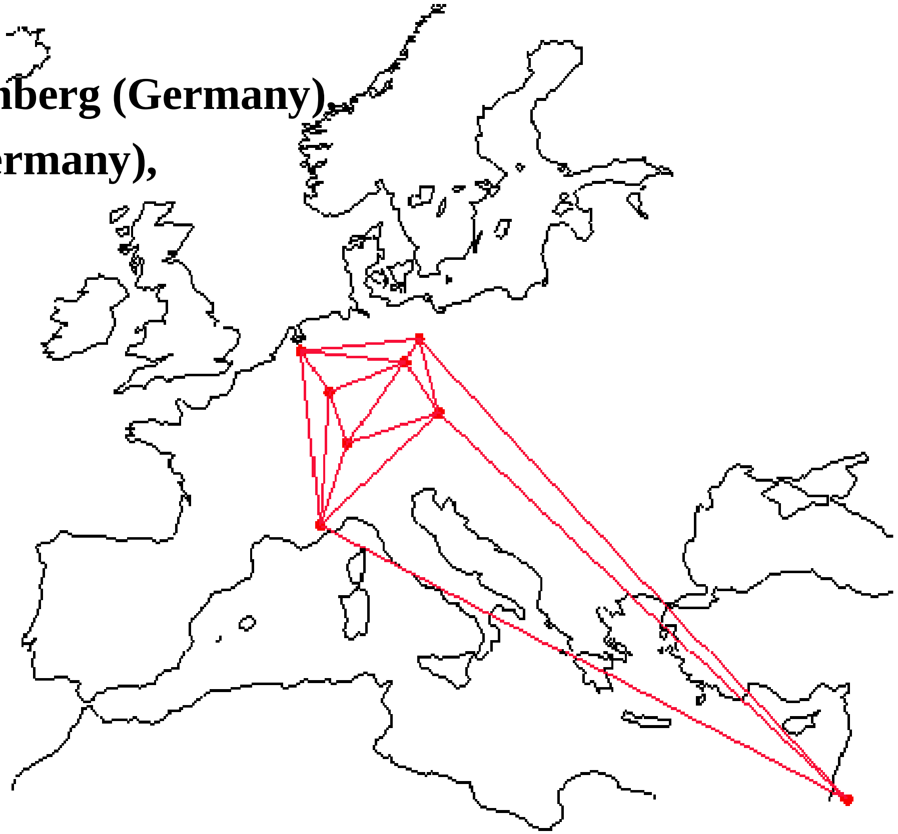
- Project Goal

  “make the large body of geometric algorithms developed in the field of computational geometry available for industrial applications.”

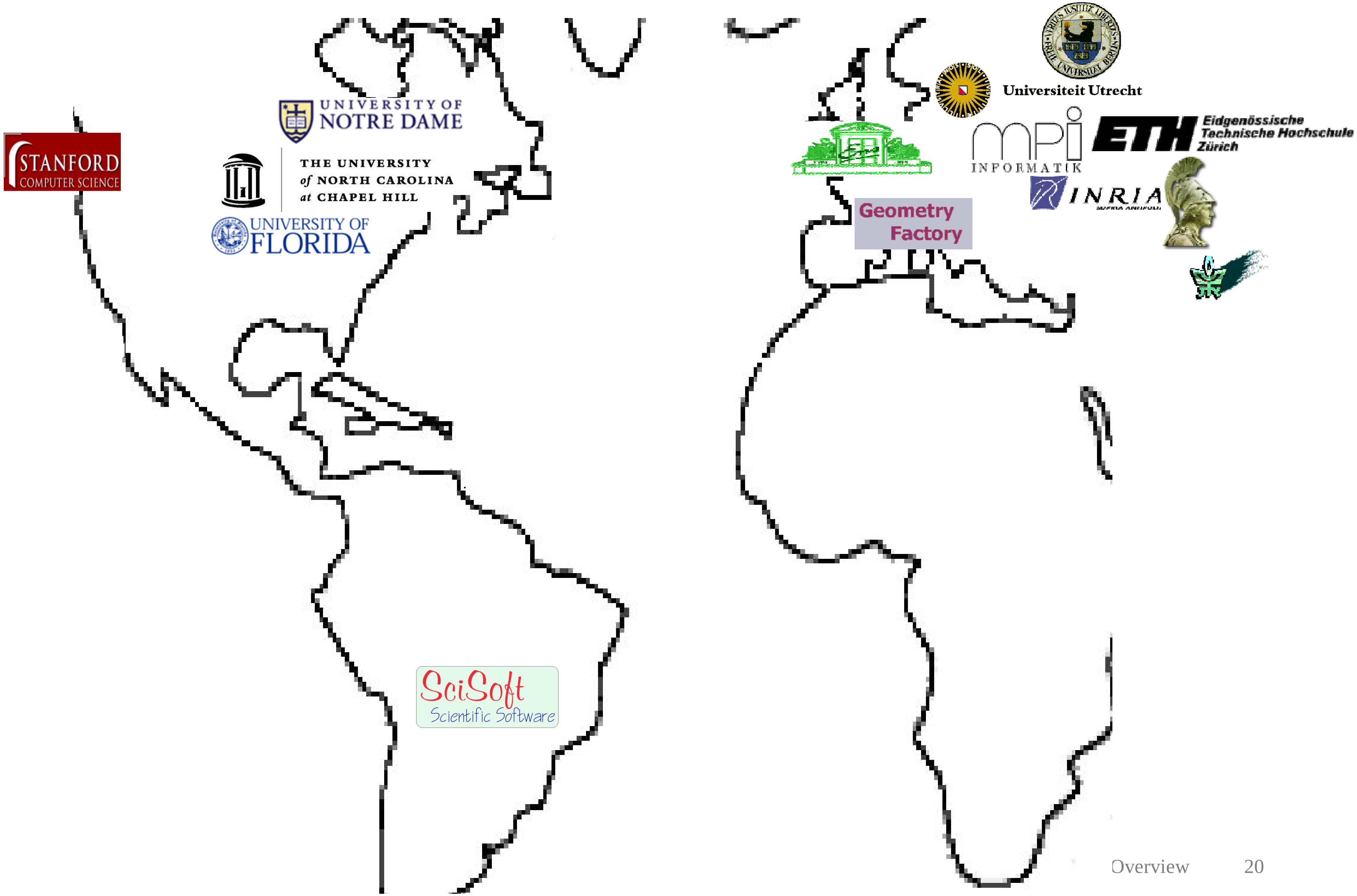
- C++ Library: Release 3.1 (Dec. 2004)

Development Started 1995

- ETH Zurich (Switzerland),
- Freie Universität Berlin (Germany),
- INRIA Sophia-Antipolis (France),
- Martin-Luther-Universität Halle-Wittenberg (Germany),
- Max-Planck-Institut für Informatik (Germany),
- RISC Linz (Austria),
- Tel Aviv University (Israel),
- Utrecht University (The Netherlands).



CGAL Developers 2004



CGAL in Numbers

- 1200 C++ classes, 300 KLOC, 2000 page manual
- ~50 developer years
- supported platforms
  - Linux, Irix, Solaris, Windows (OS X)
  - g++, SGI CC, SunPro CC, VC7, Intel
- Release cycle of ~12 months
- 6000 downloads per year
- 800 Users registered on user list
- 50 Developers registered on developer list

Development Process

- **Editorial board** reviews submissions
- Developer manual
- Own manual tools; LaTeX source ➔PS, PDF, HTML
- 1-2 developer meetings per year, 1 week long
- CVS server
- Three internal releases per week
- Automatic test suites for different compilers/platforms

Open Source License since 3.0 (2003)

- A guarantee that CGAL remains free
- Promote CGAL as a standard for users
- Opens CGAL for new contributions

- Different licenses for different parts
  - **LGPL** for Kernel and Support Library
  - **QPL** for Basic Library

  - **Commercial Licenses** from CGAL Start-up
    GeometryFactory, founded by Andreas Fabri in 2003

Commercial Customers

TOSHIBA

HyperMedia

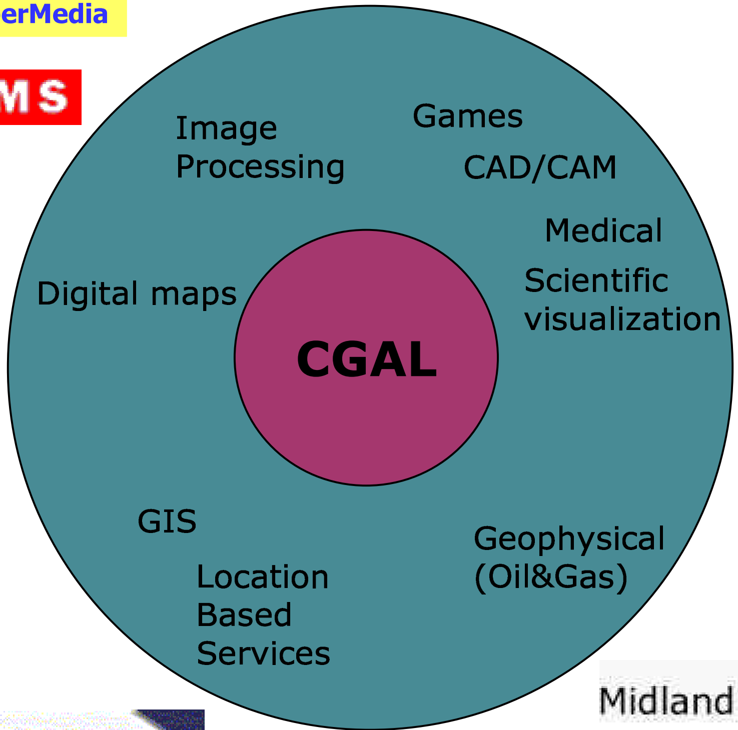
BAE SYSTEMS

QinetiQ

Leica Geosystems

TruePosition

france telecom



Agip

TGS

Midland Valley

IFP

Riesen&Stettler

Structure of CGAL

Basic Library

Algorithms and data structures

Geometric Kernel

Geom. primitives, predicates, operations

Core Library

Configuration, assertions, ...

Support Library

Visualization,
File I/O,
Numbertypes,
Generators,
...

CGAL Geometric Kernel

Primitives

2D, 3D, dD

- Point, Vector
- Line, Ray, Segment
- Triangle
- Iso_rectangle
- Bbox
- Circle

- Affine transformation

Predicates

- Order predicates
- Orientation test
- Incircle test

Constructions

- Center point
- Intersection
- Squared distance

CGAL Geometric Kernels

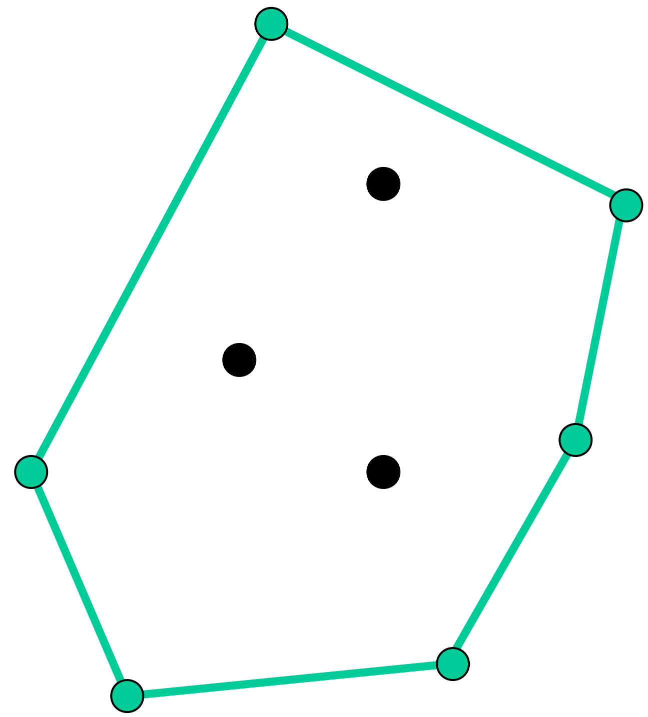
- Primitives, Predicates, Constructions
  - `Kernel::Point_2`
  - `Kernel::Left_turn_2`

- Convenient typedefs for common kernel
  - `Exact_predicates_inexact_constructions_kernel`
  - `Exact_predicates_exact_constructions_kernel`
  - `Exact_predicates_exact_constructions_kernel_with_sqrt`

- In principle choice of:
  - Cartesian and homogeneous representation
  - Reference counting and no reference counting
  - Floating-point filters as kernel adaptors or number types
  - `CGAL::Simple_cartesian<double> // non-robust!!!`

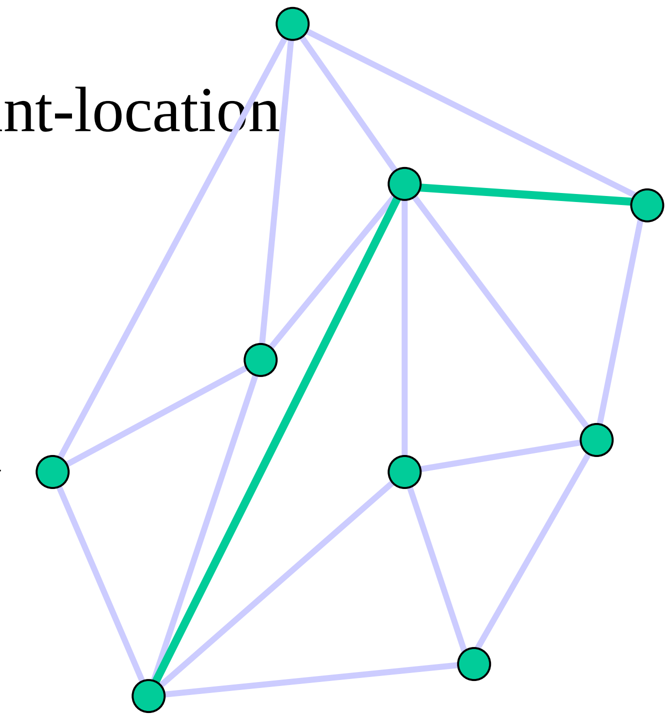
Basic Library: Convex Hull

- 5 different algorithms for 2D
- 3 different algorithms for 3D
  - Static (quickhull)
  - Randomized incremental
  - Dynamic (tetrahedrization)



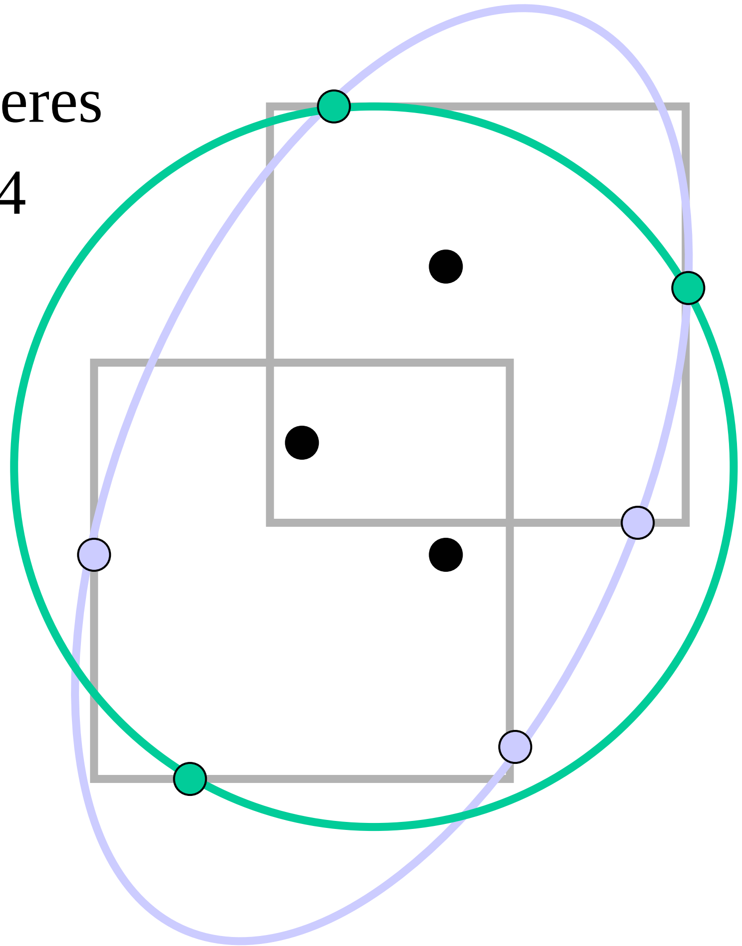
Basic Library: Triangulation

- Triangle-based data-structure
  - compact, fast, walk for point location
- Delaunay/Voronoi
  - Delaunay hierarchy for fast point-location
- Constrained Delaunay
  - => terrain triangulations
- Regular triangulations
  - Weighted points, bio-geometry
- Tetrahedrization in 3D



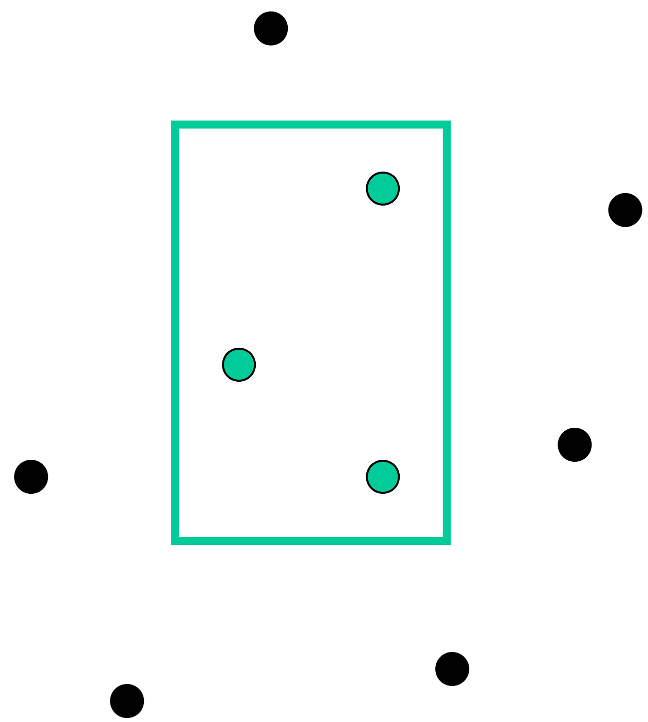
Basic Library: Geometric Optimization

- Smallest enclosing circle and ellipse in 2D
- Smallest enclosing sphere in dD
- Smallest enclosing sphere of spheres
- Rectangular p center, 2 <= p <= 4
- Width in 2D and 3D



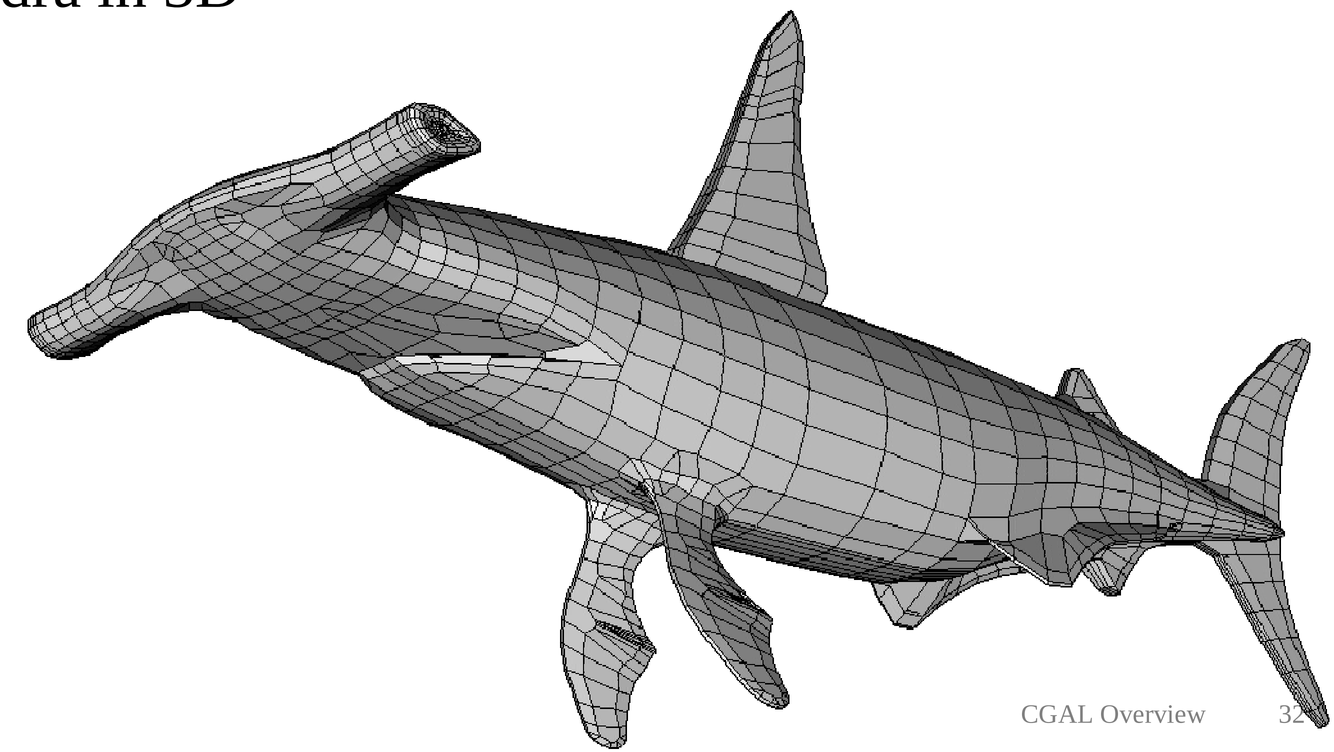
Basic Library: Search Structures

- Range-, segment-, KD-tree
- Arbitrary dimension
- Mixed segment-range-trees
- Static
- Window query, enclosing query
- Nearest neighbors
- Approximate nearest neighbors



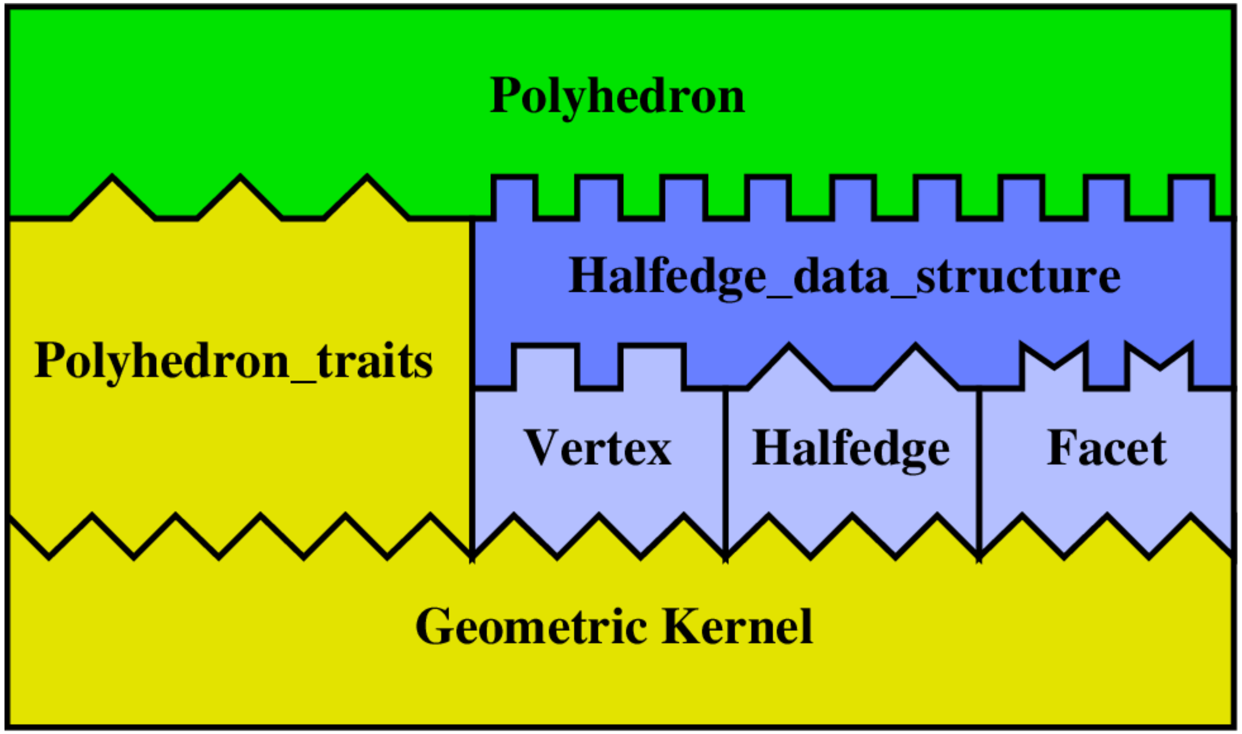
Basic Library: Halfedge Data-Structure

- Polyhedral surface: orientable 2-manifolds with boundary
- Planar map and arrangements
- Nef polygons; closed under Boolean operations
- Nef polygons embedded on the sphere
- Nef polyhedra in 3D



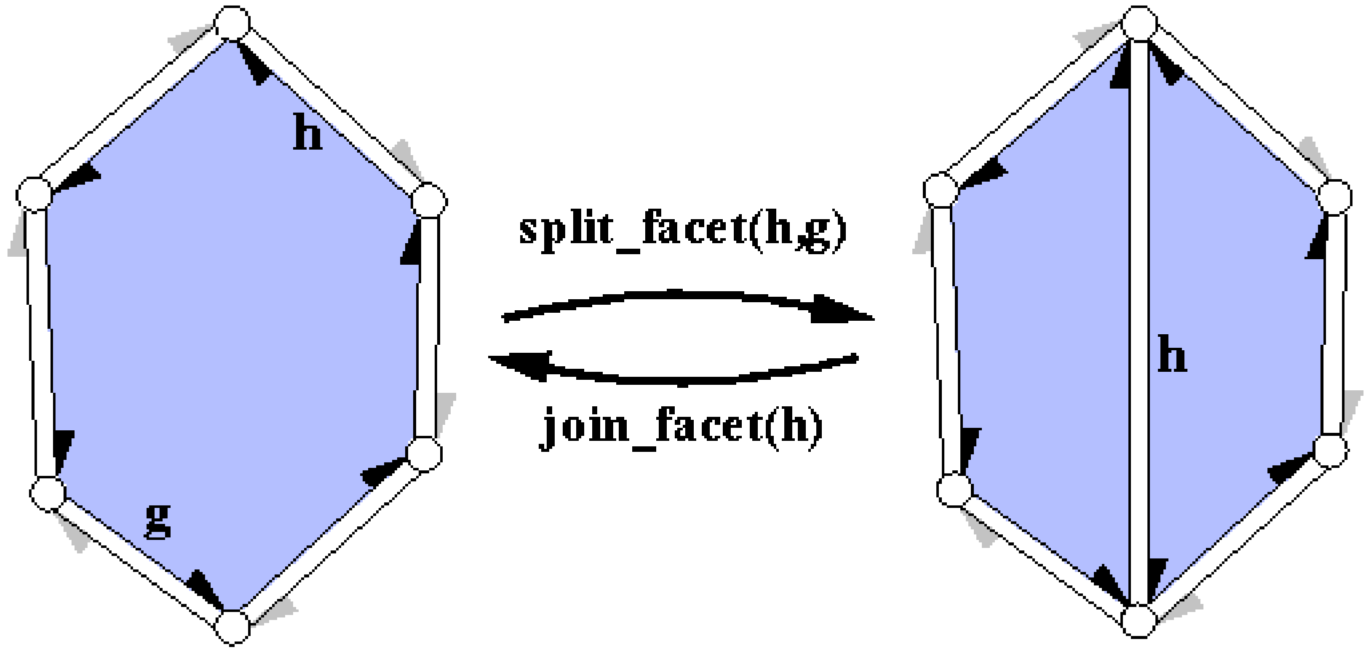
Polyhedral Surfaces

Building blocks assembled with C++ templates

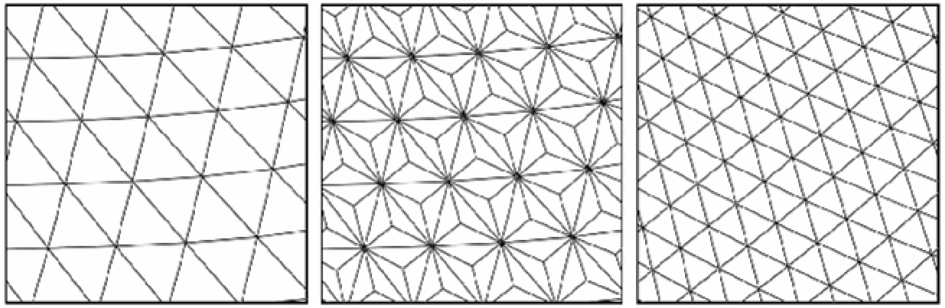


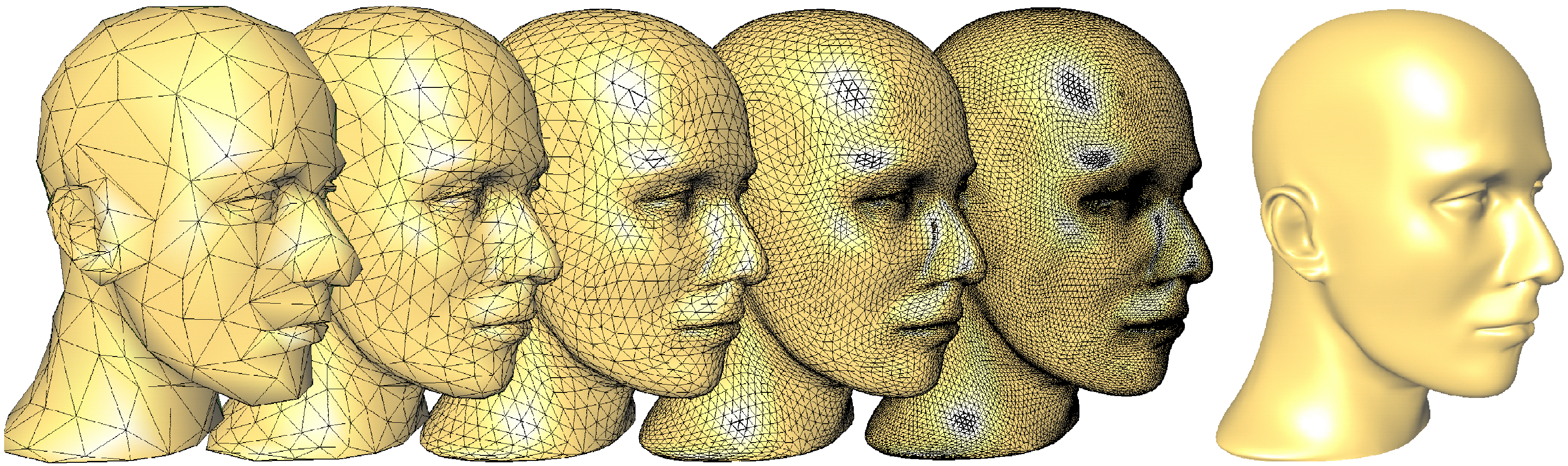
Euler Operators

- Preserve the Euler-Poincaré equation
- Abstract from direct pointer manipulations



$\sqrt{3}$ -Subdivision [Kobbelt'00]





$\sqrt{3}$ -Subdivision [Kobbelt'00]

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/HalfedgeDS_vector.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <iostream>
#include <algorithm>
#include <vector>

using std::cerr; using std::endl; using std::cout; using std::cin;
using std::exit;

class Polyhedron_min_items_3 {
public:
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef typename Traits::Point_3 Point;
        typedef CGAL::HalfedgeDS_vertex_base< Refs, CGAL::Tag_true, Point>
                                                                  Vertex;
    };
    template < class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef CGAL::HalfedgeDS_halfedge_base< Refs, CGAL::Tag_true>
                                                                Halfedge;
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef CGAL::HalfedgeDS_face_base< Refs, CGAL::Tag_true>    Face;
    };
};

typedef CGAL::Simple_cartesian<double>                      Kernel;
typedef Kernel::Vector_3                                    Vector;
typedef Kernel::Point_3                                     Point;
typedef CGAL::Polyhedron_3<Kernel, Polyhedron_min_items_3,
                           CGAL::HalfedgeDS_vector>         Polyhedron;

typedef Polyhedron::Vertex                                  Vertex;
typedef Polyhedron::Vertex_iterator                Vertex_iterator;
typedef Polyhedron::Halfedge_handle                Halfedge_handle;
typedef Polyhedron::Edge_iterator                    Edge_iterator;
typedef Polyhedron::Facet_iterator                  Facet_iterator;
typedef Polyhedron::Halfedge_around_vertex_const_circulator HV_circulator;
typedef Polyhedron::Halfedge_around_facet_circulator        HF_circulator;

void create_centroid( Polyhedron& P, Facet_iterator f) {
    Halfedge_handle h = f->halfedge();
    Vector vec = h->vertex()->point() - CGAL::ORIGIN;
    vec = vec + (h->next()->vertex()->point() - CGAL::ORIGIN);
    vec = vec + (h->next()->next()->vertex()->point() - CGAL::ORIGIN);
    Halfedge_handle new_center = P.create_center_vertex( h);
    new_center->vertex()->point() = CGAL::ORIGIN + (vec / 3.0);
}
```

```
struct Smooth_old_vertex {
    Point operator()( const Vertex& v) const {
        std::size_t degree = CGAL::circulator_size( v.vertex_begin()) / 2
        double alpha = ( 4.0 - 2.0 * cos( 2.0 * CGAL_PI / degree)) / 9.0;
        Vector vec = (v.point() - CGAL::ORIGIN) * ( 1.0 - alpha);
        HV_circulator h = v.vertex_begin();
        do {
            vec = vec + ( h->opposite()->vertex()->point() - CGAL::ORIGIN
                          * alpha / degree;
            ++ h; ++ h;
        } while ( h != v.vertex_begin());
        return (CGAL::ORIGIN + vec);
    }
};
void subdiv( Polyhedron& P) {
    if ( P.size_of_facets() == 0)
        return;
    // We use that new vertices/halfedges/facets are appended at the end.
    std::size_t nv = P.size_of_vertices();
    Vertex_iterator last_v = P.vertices_end();
    -- last_v;  // the last of the old vertices
    Edge_iterator last_e = P.edges_end();
    -- last_e;  // the last of the old edges
    Facet_iterator last_f = P.facets_end();
    -- last_f;  // the last of the old facets

    Facet_iterator f = P.facets_begin();    // create new center vertices
    do {
        create_centroid( P, f);
    } while ( f++ != last_f);

    std::vector<Point> pts;                    // smooth the old vertices
    pts.reserve( nv);  // get intermediate space for the new points
    ++ last_v; // make it the past-the-end position again
    std::transform( P.vertices_begin(), last_v, std::back_inserter( pts),
                    Smooth_old_vertex());
    std::copy( pts.begin(), pts.end(), P.points_begin());

    ++ last_e; // make it the past-the-end position again
    for ( Edge_iterator e = P.edges_begin(); e != last_e; ++e)
        P.flip_edge(e);    // flip the old edges
}
int main() {
    Polyhedron P;
    cin >> P;
    P.reserve( P.size_of_vertices() + P.size_of_facets(),
               P.size_of_halfedges() + 6 * P.size_of_facets(),
               3 * P.size_of_facets());
    subdiv( P);
    cout << P;
    return 0;
}
```

$\sqrt{3}$ -Subdivision [Kobbelt'00]

- Comparison with OpenMesh 1.0.0-beta4
- Lion vase: 400k triangles
- 2 subdivision steps

	CGAL		OpenMesh
$\sqrt{3}$-subdivision	float	double	float
Lion vase: step 1	0.87	1.33	1.33
Lion vase: step 2	3.03	4.68	4.83

Geometric Algorithms

Self Intersection Test

- Based on fast box intersections [Zomorodian&Edelsbrunner'02]
- Needs exact predicates

Smallest Enclosing Sphere (of Spheres)

- Linear time algorithm (randomized) [Fischer&Gärtner'03]
- Needs exact constructions, but robust with `double`'s

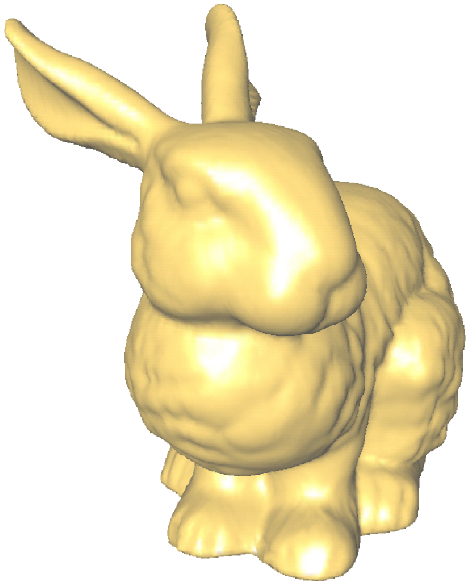
Convex Hull and Width

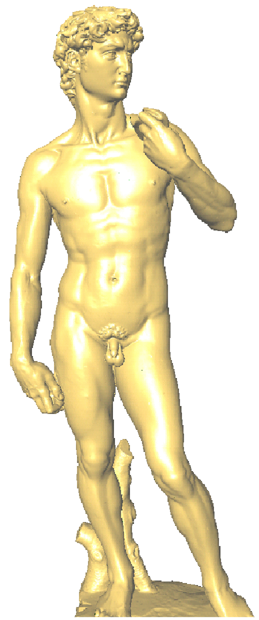
- Quickhull [Barber et al.'96], width can be quadratic
- Convex hull needs exact predicates, width needs exact constr.

Geometric Algorithms

time in seconds	min. sphere double	min. sphere gmpq	convex hull	min. width	self inter-section
Bunny	0.02	14	3.5	111	2.3
Lion vase	0.19	396	13.1	276	15.5
David	0.12	215	20.3	112	31.5
Raptor	0.35	589	45.5	123	78.2

#F 70k 400k 700k 2.000k






Contents

- Why is Geometric Computing Hard?
- History and Overview of CGAL
- **Overview of EXACUS**
- Design of CGAL (and EXACUS)

Efficient and Exact Algorithms for Curves and Surfaces

Max-Planck-Institut für Informatik

Group: Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Lutz Kettner, Kurt Mehlhorn, Joachim Reichel, Susanne Schmitt, Elmar Schömer, and Nicola Wolpert.

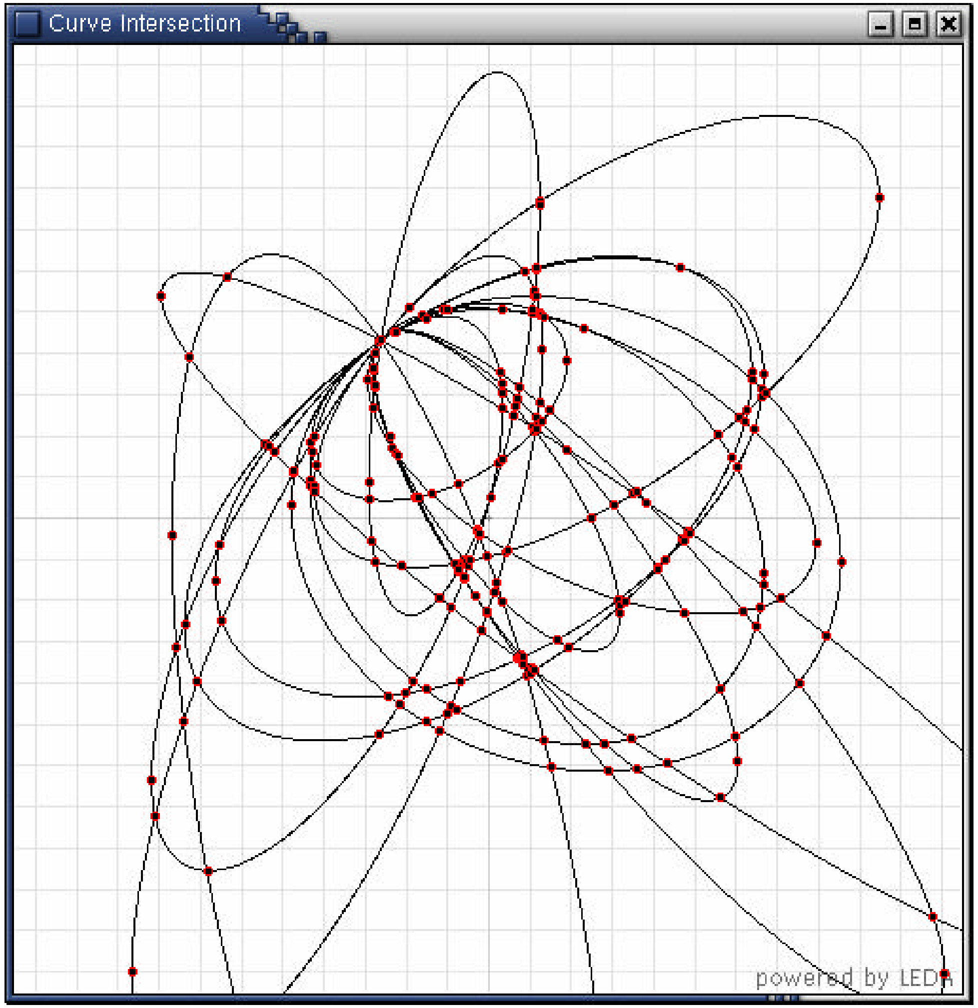
Overview

- **EXACUS** is a project at MPII since April 2002
- **EXACUS** is a collection of C++ libraries that extend computational geometry methods to curves and surfaces, starting with arrangement computations.
- **Motivation:** correct, complete, and efficient CAD geometry kernel

ConiX in EXACUS

Degenerate Arrangements of conics

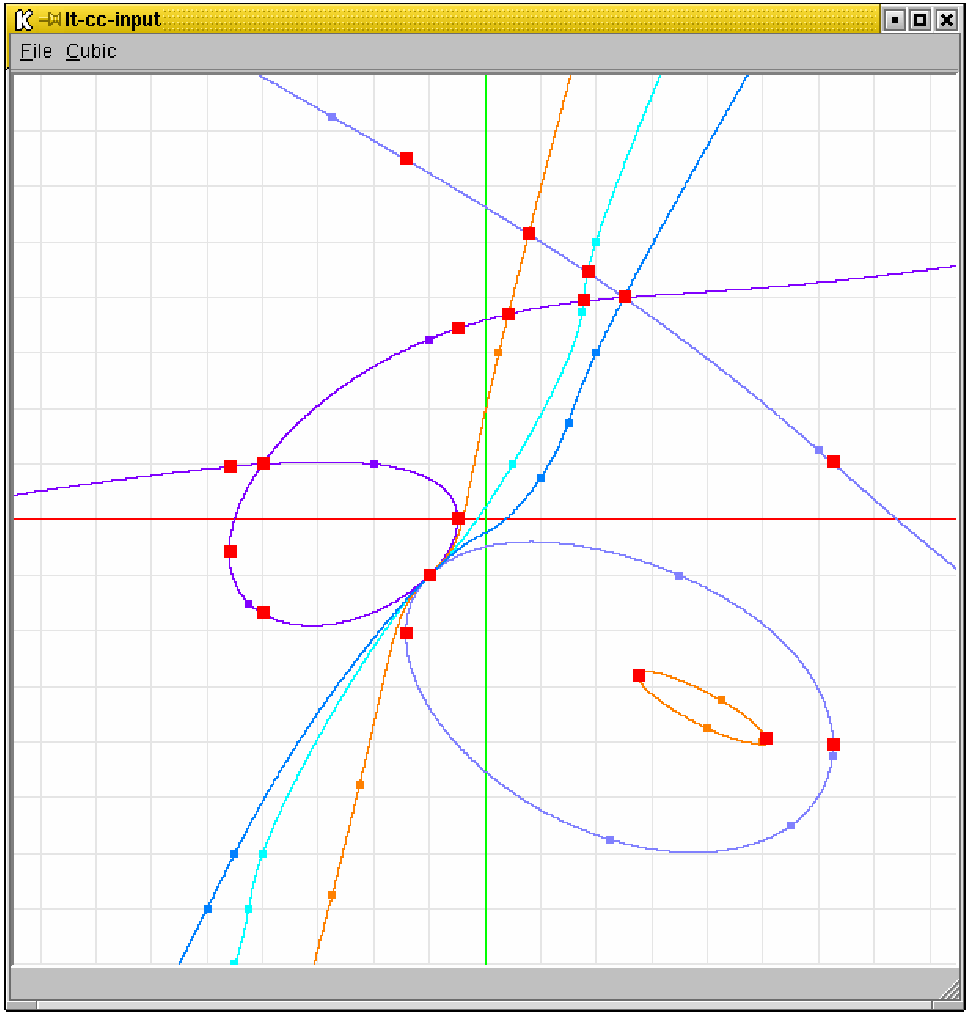
15 ellipses passing through a common point



CubiX in EXACUS

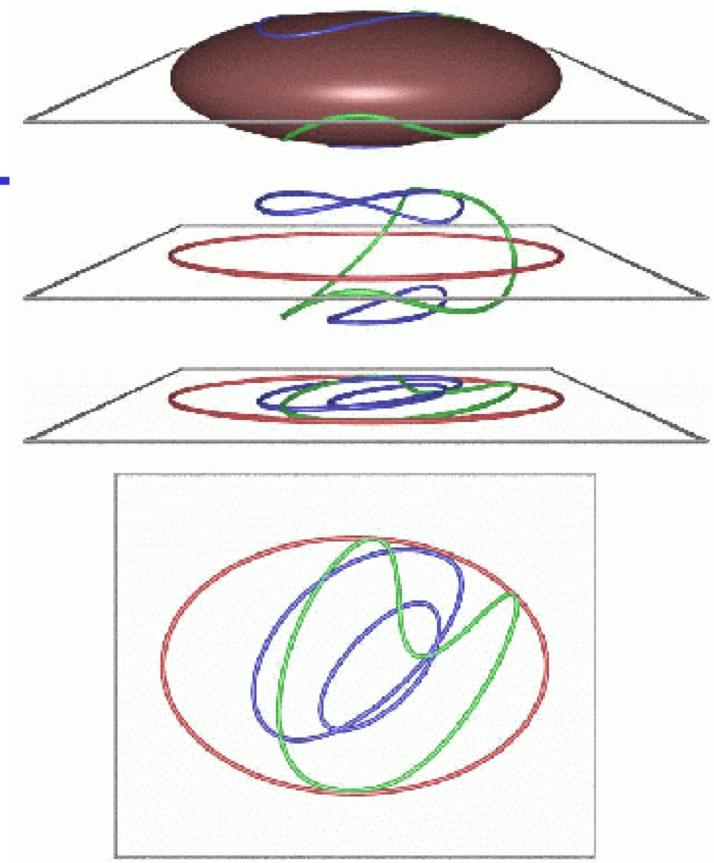
Degenerate Arrangements of cubic curves

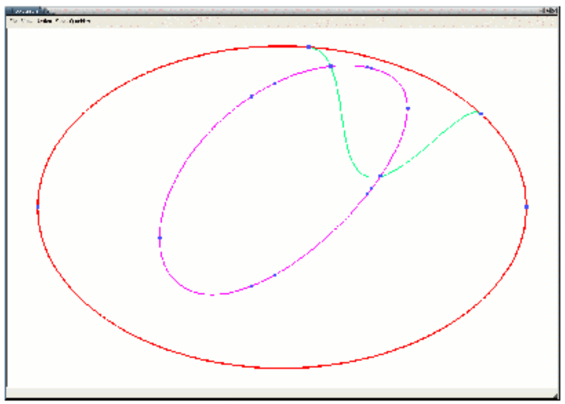
Triple intersection point

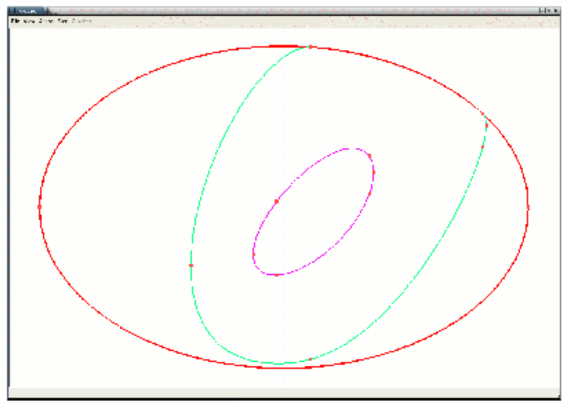


QuadriX in EXACUS

Arrangements of intersection and silhouette curves of arrangements of quadrics in space



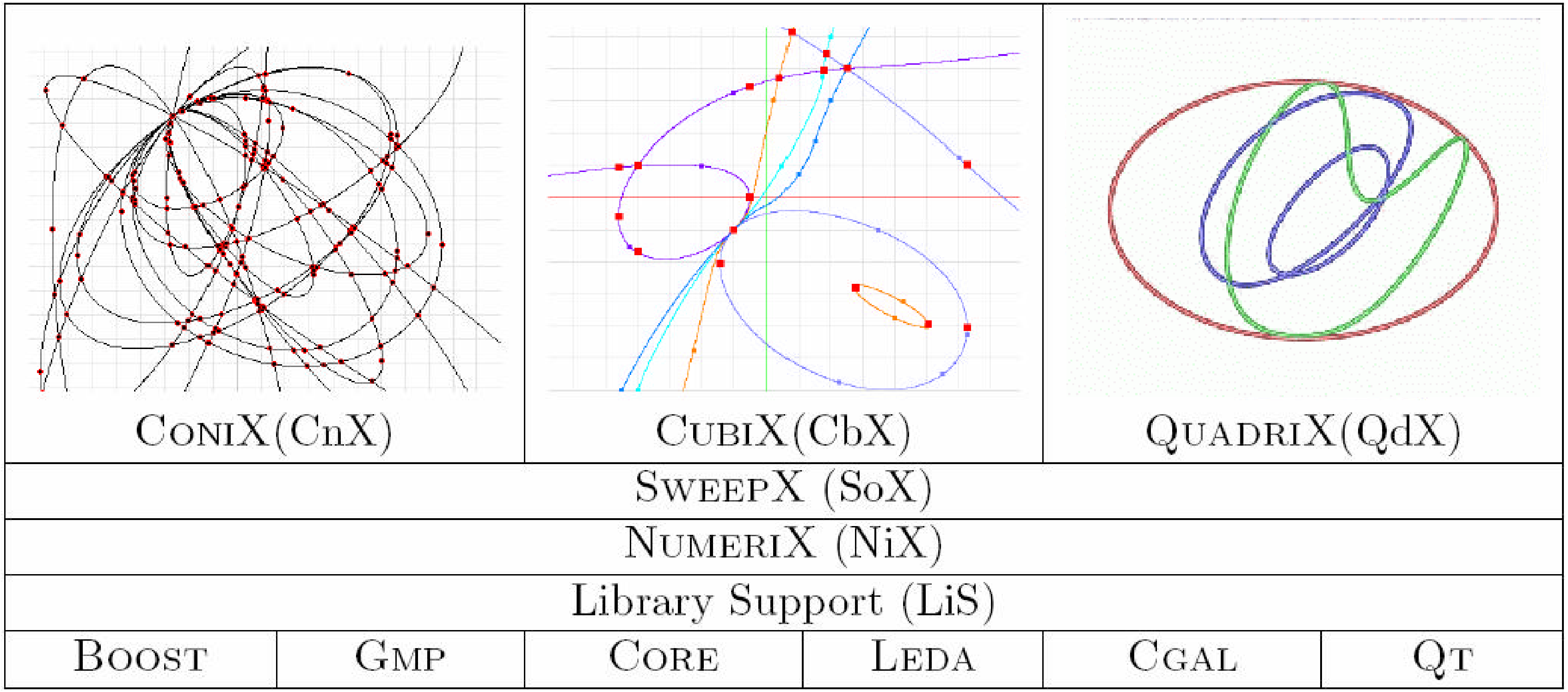




Some Facts

- 6 C++ library, according to sub-projects
- Release 0.9 2004 with the open source license QPL
- Follows the generic programming paradigm (C++ templates)
- 94000 lines of code and documentation (DOXYGEN)
- Supported are g++ on Linux and Solaris
- We use: CVS, autoconf, automake, libtool

Library Layers

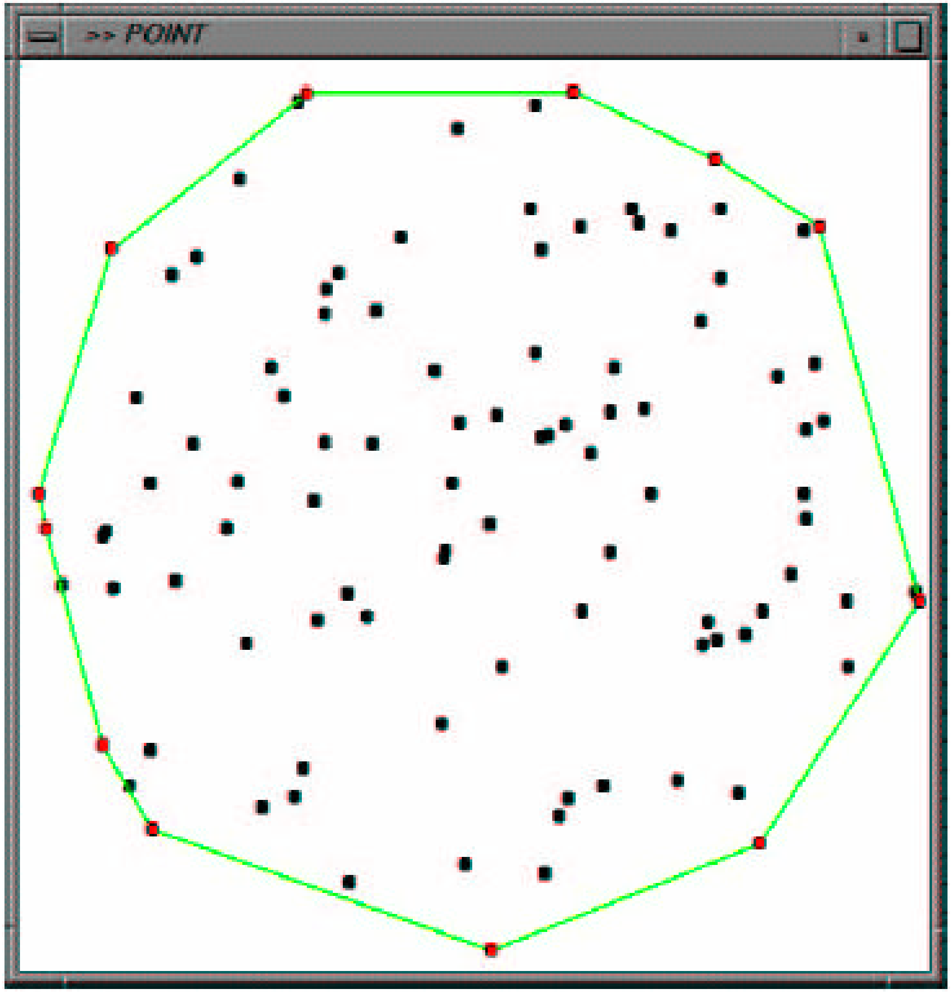


- Curved Kernel
- Algebraic Kernel

Contents

- Why is Geometric Computing Hard?
- History and Overview of CGAL
- Overview of EXACUS
- **Design of CGAL (and EXACUS)**
  - **Basic Library**
  - **Kernel**

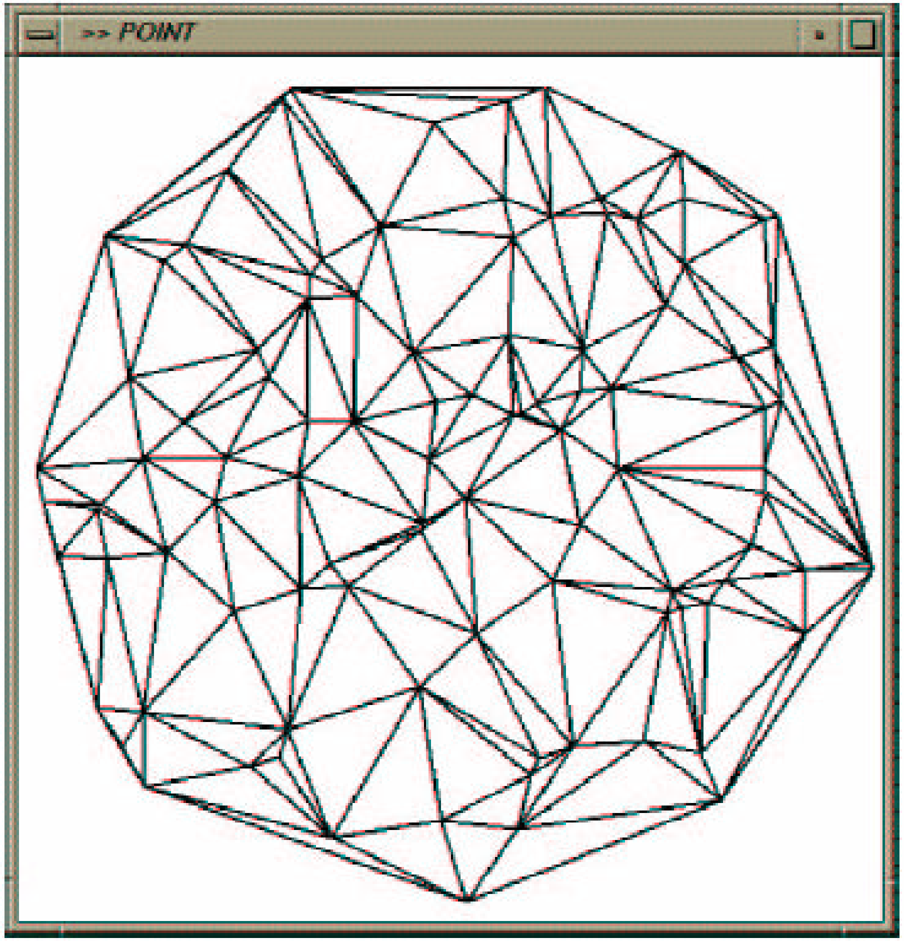
2D Convex Hull



2D Convex Hull

```
int main () {
    Random rnd(1);
    Random_points_in_disc_2 rnd_pts( 1.0, rnd);
    list<Point_2> pts;
    copy_n( rnd_pts, 100, back_inserter( pts));
    Polygon_2 ch;
    CGAL::convex_hull_points_2( pts.begin(), pts.end(), back_inserter(ch));
    Window* window = demo_window();
    Window_iterator_point_2 wout( *window);
    copy( pts.begin(), pts.end(), wout);
    *window << CGAL::GREEN << ch << CGAL::RED;
    copy( ch.vertices_begin(), ch.vertices_end(), wout);
    Point_2 p; *window >> p; // wait for mouse click
    delete window; return 0;
}
```

2D Delaunay Triangulation

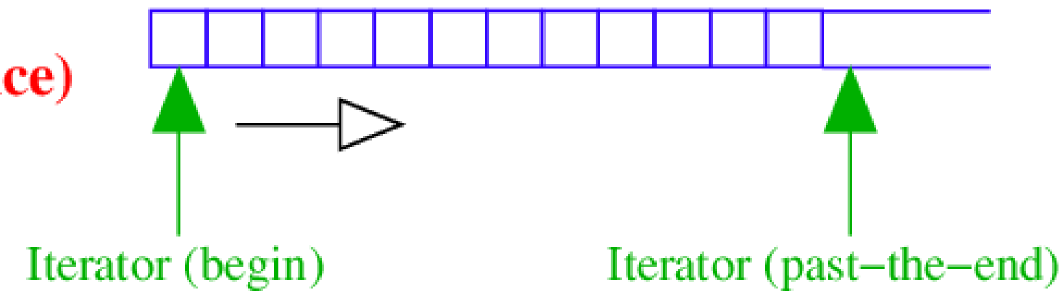


2D Delaunay Triangulation

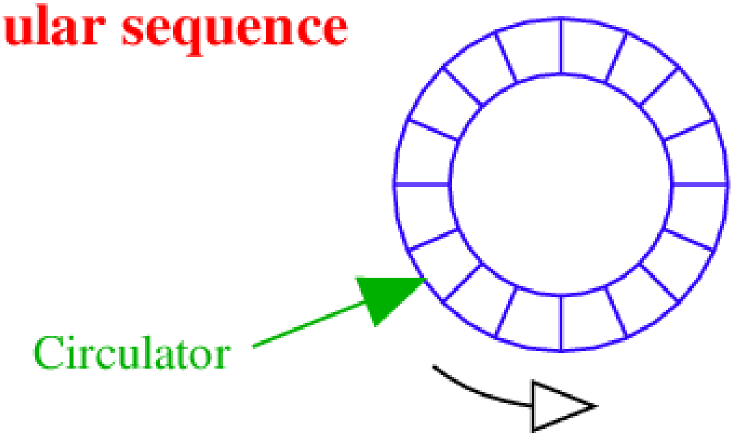
```
int main () {
    Random rnd(1);
    Random_points_in_disc_2 rnd_pts( 1.0, rnd);

    Delaunay_triangulation_2 dt;
    copy_n( rnd_pts, 100, back_inserter( dt));

    Window* window = demo_window();
    *window << dt;

     Point_2 p; *window >> p; // wait for mouse click
    delete window; return 0;
}
```

Iterators and Circulators

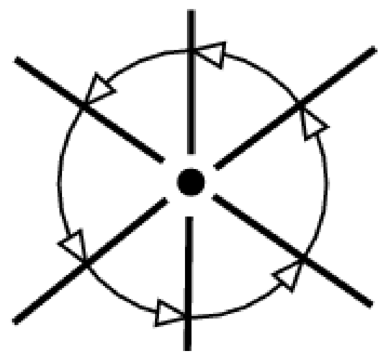
Container
(linear sequence)



Circular sequence



For example:
graph vertex



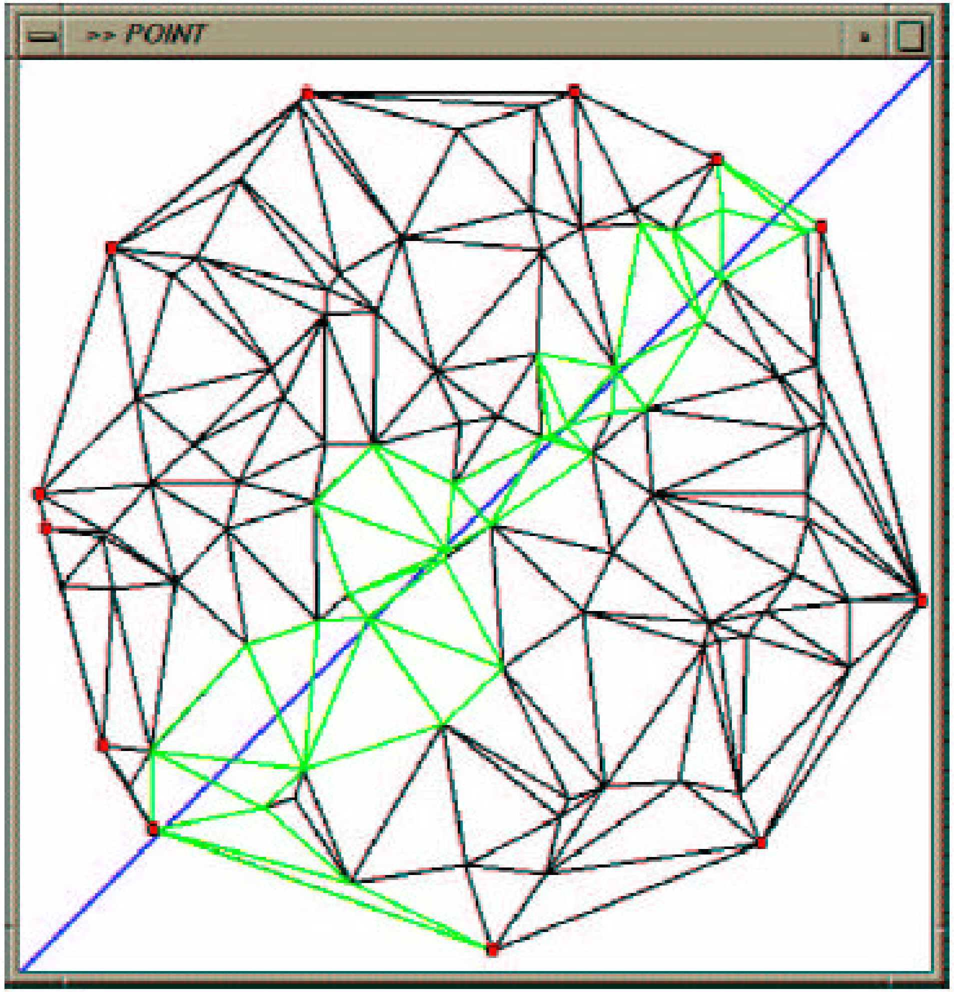
Circulators

Similar to iterators, but made for circular structures:

- Cyclic behavior of operator++
- do … while()-loop instead of while()-loop

```
template <class Circulator, class T>
bool contains( Circulator c, Circulator d, const T& value) {
   if ( c != NULL) {
      do {
         if ( *c == value)
            return true;
      } while ( ++c != d);
   }
   return false;
}
```

Circulators in Triangulations



Block-Structure is Misleading

Basic Library

Algorithms and data structures

Geometric Kernel

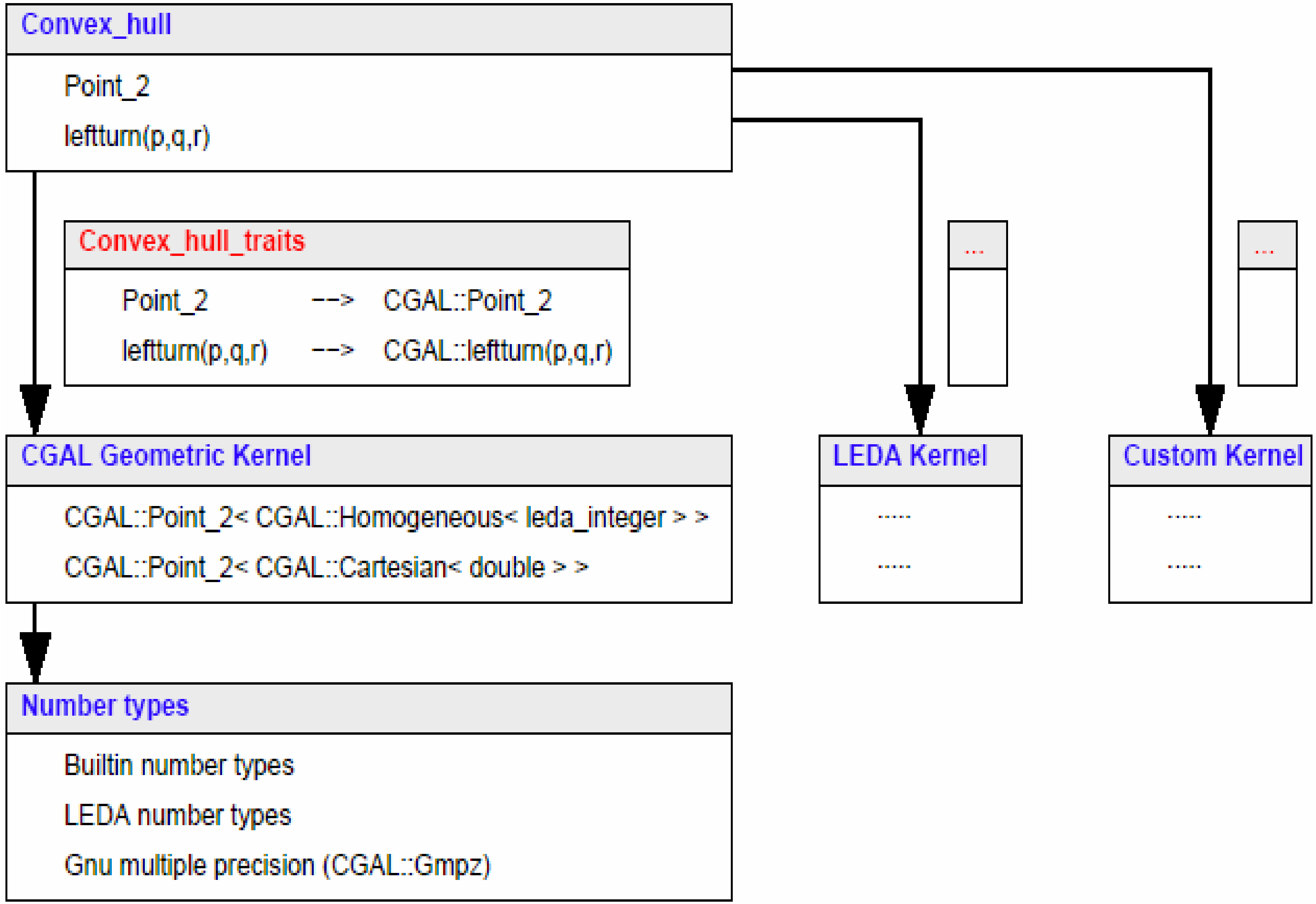
Geom. primitives, predicates, operations

The basic library does actually not depend on the geometric kernel.
It is decoupled with a template parameter, the geometric traits class.

Geometric Traits Class

- Collection of geometric primitives, predicates, and constructions in a traits class.
- Algorithms and data structures are parameterized with a geometric traits class
- i.e., for each algorithm and data structure exists a minimal list of requirements on its geometric traits class to make this algorithm and data structure work.
- Some traits class requirements are shared.
- Allows easy exchange of different geometry implementations.
- A CGAL Geometry Kernel is a valid model for many traits.

Geometric Traits Class



Triangulation of a 3D Terrain

```
typedef CGAL::Triangulation_euclidean_traits_xy_3<Kernel> Traits;
typedef CGAL::Delaunay_triangulation_2<Traits> Triangulation;

int main () {
    Triangulation dt;
    copy( istream_iterator<Point_3>(cin), istream_iterator<Point_3>(),
              back_inserter( dt));
    cout << dt;
    return 0;
}
```

Example of Geometric Traits

Let’s start with a simple point type:

```
template <class NT> struct Point_2 {
    typedef NT Number_type;
    NT x;
    NT y;
};
```

Example of Geometric Traits

And define a traits class with the orientation test:

```
template <class NT> struct Convex_hull_traits {
    typedef Point_2<NT> Point;
    struct Left_turn {
        bool operator()( const Point& p, const Point& q, const Point& r) {
            return (q.x-p.x) * (r.y-p.y) > (r.x-p.x) * (q.y-p.y);
        }
    };
    Left_turn left_turn_object() const { return Left_turn(); }
};
```

Example of Geometric Traits

A simple 2d convex hull algorithm with traits class interface:

```
template <class BidirectionalIterator, class OutputIterator, class Traits>
OutputIterator convex_hull( BidirectionalIterator first,
                            BidirectionalIterator beyond, // sorted range
                            OutputIterator result, const Traits& traits) {
  typedef typename Traits::Point Point;
  vector<Point> hull;
  hull.push_back( *first); // sentinel
  hull.push_back( *first); // lower convex hull (left to right)
  BidirectionalIterator i = first;
  for ( ++i; i != beyond; ++i) {
    while ( traits.left_turn_object()( hull.end()[-2], *i, hull.back()))
      hull.pop_back();
    hull.push_back( *i);
  }
```

Andrewas'variant of Graham's scan, one half only

Choose a Traits Default Impl.

With iterator traits we get the value type, which are our points, from which we get the number type used for coordinates and that we can use to instantiate our default geometric traits.

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator convex_hull( BidirectionalIterator first,
                            BidirectionalIterator beyond,
                            OutputIterator result) {
   typedef typename iterator_traits<BidirectionalIterator>::value_type P;
   typedef typename P::Number_type Number_type;
   typedef Convex_hull_traits<Number_type> Traits;
   return convex_hull( first, beyond, result, Traits());
}
```

Geometric Traits Classes

Use functors to implement predicates and constructions,

- which are type safe,
- efficient,
- and can have additional state, such as error bounds for filters
  - to pass the state safely around, the traits has to offer these access member functions, as in:

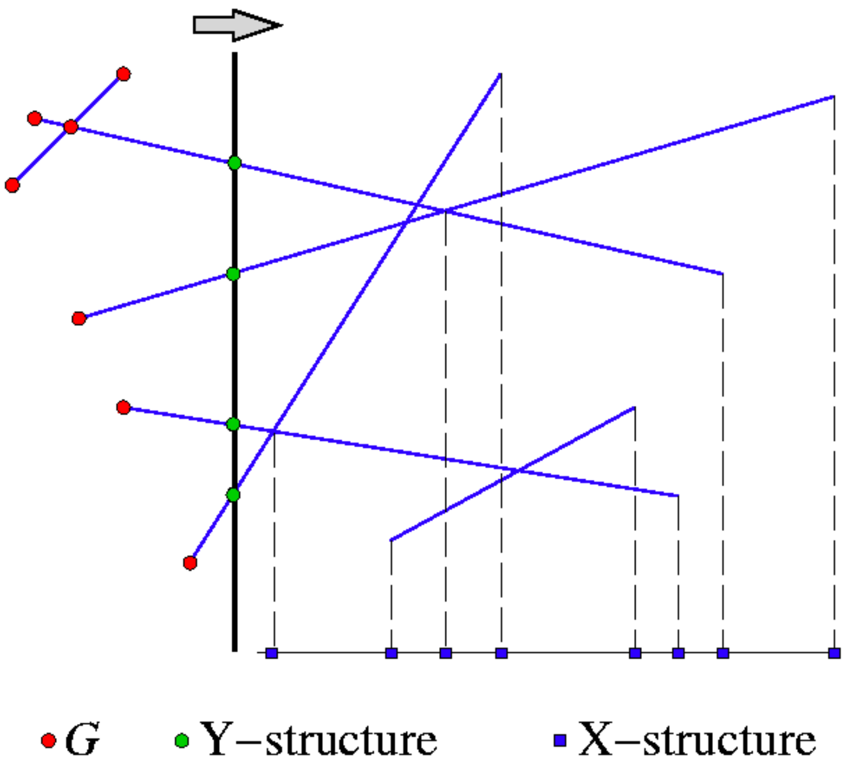
```
Left_turn left_turn_object() const { return Left_turn(); }
```

SoX:: SweepX

- Generic Bentley-Ottmann sweep-line algorithm
  - Variant handling all degeneracies
  - Output in LEDA graph
  - Gen_polygon_2 for regular boolean operations
  - Extensions necessary for curves:
    - sweeping through a high-degree vertex
    - geometric predicates
- Adaptor for CGAL arrangement traits

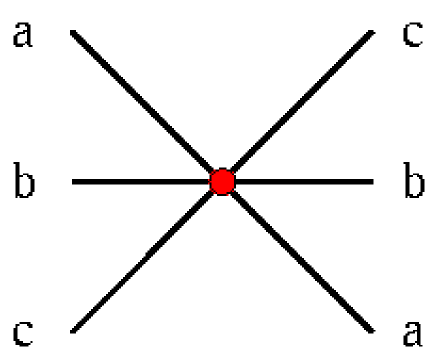
Sweep-Line Algorithm for Line Segments

- **Input:** a set of line segments
- **Output:** the planar map (graph) G defined by the segments; G has one vertex for each endpoint and each intersection

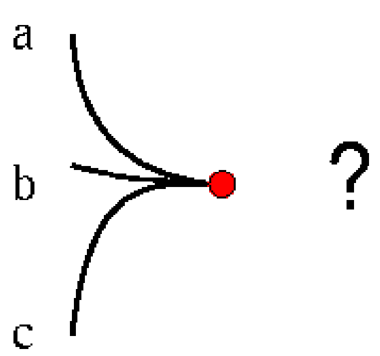


- sweep a vertical line L across the plane and maintain
- Y-structure = sorted sequence of interactions between L and segments
- X-structure = sorted known vertices ahead of sweep line
- update at event point
- G emerges to the left of L

Sweeping Through a High-Degree Vertex



- In the case of segments:
  - the y-order is reversed
  - update of Y-structure in linear time in degree of the vertex



- In the case of curves:
  - the y-order behaves more complicated
    - inters. of odd multiplicity change order
    - inters. of even multiplicity keep order
  - update of Y-structure in linear time in degree of the vertex times cost of determining multiplicities of neighboring curves

Predicates and Functions for the Sweep

- 1. Compute intersection points of two curves
- 2. Lex-order on intersection points and endpoints
  - for order in X-structure
  - and inserting new endpoints in Y-structure
- 3. Multiplicity of intersections
  - for sweeping through a vertex

`CurveSweepTraits_2` Concept

Types

- `Point_2` used for end-points and intersection points of segments.
- `Segment_2` used as curve segment in computing the arrangement.

We require for points and segments to be default-constructible and assignable. In addition, we need also a global `ID_Number` function overloaded for points and segments, which is necessary for the LEDA data structures used in the sweep-line algorithm.

`CurveSweepTraits_2` Concept

Predicates

- `Compare_xy_2` and `Less_xy_2` provide lexicographic comparison of two points, the former with a three-valued return type and the latter as a boolean predicate.
- `Is_degenerate_2` returns true if the segment consists of only one point.
- `Do_overlap_2` determines if two segments have infinitely many points in common (i.e. intersect in a non-degenerate segment).
- `Compare_y_at_x_2` determines the vertical placement of a point relative to a segment.
- `Equal_y_at_x_2` determines if a point lies on a segment (This functor can have a faster implementation than testing `Compare_y_at_x_2` for equality).
- `Multiplicity_of_intersection` computes the multiplicity of an intersection point between two segments. Used in the linear-time reordering of segments and hence used only for non-singular intersections.
- `Compare_y_right_of_point` determines the ordering of two segments just after they both pass through a common point. Used for the insertion of segments starting at an event point.

CurveSweepTraits_2 Concept

Accessors and Constructions

- `Source_2` returns the source point of a curve segment.
- `Target_2` returns the target point of a curve segment.
- `Construct_segment_2` constructs a degenerate curve segment from a point.
- `New_endpoints_2` and `New_endpoints_opposite_2` replace the endpoints of a curve segment with new representations and return this new curve segment. The latter functor also reverses the orientation of the segment. They are used in the initialization phase of the sweep-line algorithm where equal end-points are identified and where the segments are oriented canonically from left to right [24].
- `Intersect_2` constructs all intersection points between two segments in lexicographical order.
- `Intersect_right_of_point_2` constructs the first intersection point between two segments right of a given point. Used only for validity checking and when the intersection dictionary for caching of already computed intersections is not used.

Contents

- Why is Geometric Computing Hard?
- History and Overview of CGAL
- Overview of EXACUS
- **Design of CGAL (and EXACUS)**
  - **Basic Library**
  - **Kernel**

Points and Predicates

```
typedef Simple_cartesian< double> Kernel; // not robust!!
typedef Kernel::Point_2  Point;

int main() {
   Point p( 1.0, 0.0);
   Point q( 1.3, 1.7);
   Point r ( 2.2, 6.8);
   switch ( CGAL::orientation( p, q, r)) {
      case CGAL::LEFT_TURN:   cout << “Left turn”; break;
      case CGAL::RIGHT_TURN: cout << “Right turn”; break;
      case CGAL::COLLINEAR:   cout << “Collinear”; break;
   }
   return 0;
}
```

Exact Orientation Test

```
typedef Homogeneous< leda_integer> Kernel;
typedef Kernel::Point_2  Point;

int main() {
    Point p( 10,   0, 10);
    Point q( 13, 17, 10);
    Point r ( 22, 68, 10);
    switch ( CGAL::orientation( p, q, r)) {
        case CGAL::LEFT_TURN:   cout << “Left turn”; break;
        case CGAL::RIGHT_TURN: cout << “Right turn”; break;
        case CGAL::COLLINEAR:   cout << “Collinear”; break;
    }
    return 0;
}
```

Filtered Predicates Using Exceptions

- Interval arithmetic as C++ number type with operators
- Implement operator== etc. to throw an exception when unsafe
- Generic implementation of sign of 2x2 determinant

```
template <class FT>
Sign sign_of_det_2x2( const FT& a, const FT& b,
                      const FT& c, const FT& d) {
   return compare( a*d, c*b);
}
```

Filtered Predicates Using Exceptions

Implement filtered number type storing double's

- interval() member function to access interval number type
- exact() member function to access exact number type

```
Sign sign_of_det_2x2( Filtered a, Filtered b, Filtered c, Filtered d) {
    try {
        return sign_of_det_2x2( a.interval(), b.interval(),
                                c.interval(), d.interval());
    } catch ( Interval_nt_advanced::unsafe_comparison) {
        return sign_of_det_2x2( a.exact(), b.exact(),
                                c.exact(), d.exact());
    }
}
```

Can be done automatically at the cost of a dynamic expr. trees.

Polymorphic Return Types with CGAL::Object

```
int main() {
    Segment s( Point(1,1), Point(1,5));
    Segment t( Point(1,3), Point(1,8));
    if ( CGAL::do_intersect( s, t)) {
        CGAL::Object result = CGAL::intersection( s, t);
        Point pt;
        Segment seg;
        if ( CGAL::assign( pt, result))
            cout << “Point “ << pt;
        else if ( CGAL::assign( seg, result))
            cout << “Segment “ << seg;
    }
}
```

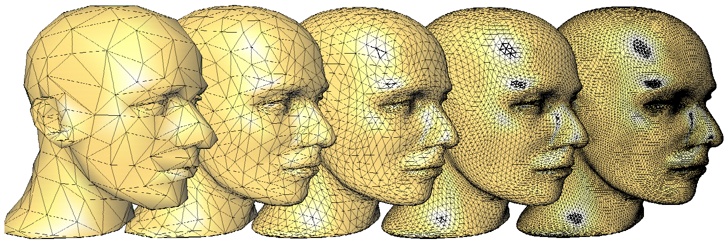
Polymorphic Return Types with CGAL::Object

- Looks like monolithic derivation hierarchy with one base object: CGAL::Object.
- Bit it isn’t. In fact, geometric types are not derived in CGAL and have no virtual member functions.
- Instead, a parallel set of objects is created on the fly to represent the necessary hierarchy with virtual member functions for the runtime polymorphism in CGAL::Object.

```
struct Object() { virtual ~Object(); … };
template class <T> struct Wrapper : public Object { … }
```

See also Gamma et.al., Design Patterns, 1995, Addison-Wesley.

Thank You!

- www.cgal.org
- 

  www.cgal.org/Tutorials/
- www.mpi-sb.mpg/EXACUS