Design of the CGAL Spherical Kernel

Pedro M. M. de Castro, Frédéric Cazals, Sébastien Loriot, Monique Teillaud

WRSO 2007/09/26 @ INRIA Sophia

Work partially supported by the EU STREP Project

IST-006413
The Computational Geometry Algorithms Library
Open Source project
www.cgal.org

> 400,000 lines of C++ code
> 3,000 pages manual
~ 12,000 downloads per year
~ 850 users on public mailing list
~ 50 developers
licenses LGPL or QPL
start-up GeometryFactory
interfaces: Python, Scilab

Robustness and efficiency
Quality:

- Editorial board
  (3 members in Geometrica / 12 members)
- Test-suites each night
...
A kernel consists of constant-size non-modifiable geometric primitive objects and operations on these objects.

Predicates are basic units of geometric algorithms $\iff$ decisions; Value returned belong to an enum

Constructions generate objects that are neither of type bool nor enum types

CGAL Kernels

- Primitives: elementary geometric objects (points, segments, lines, ...)
- Predicates and constructions: Elementary operations on them (intersection tests, intersection computations, ...)

For example CGAL::Polyhedron is not in a kernel
A kernel concept defines requirements for a kernel in order to be able to construct generic geometric algorithms based only on requirements (usable with any kernel model of the concept).

→ Each kernel in CGAL is a model of a kernel Concept

Kernel concept design guidelines

– **Code reuse**: ability to reuse the CGAL kernel for points, circles, number types,…
– **Flexibility**: possibility to use other implementations for points, circles, number types,… possibility to use several algebraic implementations

Up to release 3.1 (Dec’04):

essentially linear objects

Release 3.2 (May ’06):

2D circular kernel [Pion-Teillaud]
2D circular kernel concept
[Emiris-Kakargias-Pion-Teillaud-Tsigaridas socg’04]

template < LinearKernel, AlgebraicKernel >
class CircularKernel

Types:

- Must be defined by LinearKernel
  basic number types, points, lines,

- Must be defined by AlgebraicKernel
  algebraic numbers, polynomials

- Defined by CircularKernel
  Circular_arc_2, Line_arc_2,
  Circular_arc_point_2
2D circular kernel concept

[Emiris-Kakargias-Pion-Teillaud-Tsigaridas socg’04]

template < LinearKernel, AlgebraicKernel >
class CircularKernel

Types:

- Must be defined by LinearKernel
  basic number types, points, lines, ...
- Must be defined by AlgebraicKernel
  algebraic numbers, polynomials
- Defined by CircularKernel
  Circular_arc_2, Line_arc_2,
  Circular_arc_point_2

Predicates: intersection tests, comparisons of intersection points, ...
Constructions: computation of intersection points
3D Spherical Kernel : Concept

Following the same design, we define a new geometric kernel: the 3D spherical kernel.

template < LinearKernel, AlgebraicKernel >
class SphericalKernel

which must define the following types:

Circle_3
Circular_arc_3
Line_arc_3
Circular_arc_point_3
3D Spherical Kernel : Concept

Access functions: Define the interface with kernel objects

- **Circle_3**
  - center()
  - squared_radius()
  - supporting_plane()
  - diametrial_sphere()

- **CircularArcPoint_3**
  - x(), y(), z()

- **LineArc_3**
  - source(), target()
  - supporting_line()

- **CircularArc_3**
  - source(), target()
  - supporting_circle()
Coordinate system chosen: **Cartesian Coordinates**.

- **Circle_3**
  represented by a plane and a sphere.

- **Circular_arc_3**
  represented by a circle and two endpoints
  (Circular_arc_point_3)

- **Line_arc_3**
  represented by a kernel line with two endpoints
  (Circular_arc_point_3)

- **Circular_arc_point_3**
  represented by an algebraic number per each cartesian coordinates
User frontend to 3D Spherical Kernel: Geometric functions

Predicates:
- Has_on_3, Do_overlap_3
- Compare_x_3, Compare_y_3, Compare_z_3
  Compare cartesian coordinates of Circular_arc_point_3
- Side_of_3
  position of a Circular_arc_point_3 wrt a plane or a sphere

Constructions:
- Intersect_3
  (from 2 or 3 objects among planes, circle arcs, line and spheres)
Requirements to Algebraic Kernel

**Type:**
- FT
- Polynomial_1_3
- Polynomial_for_spheres_2_3
- Polynomial_for_lines_3
- Polynomial_for_circles_3
- Root_of_2
- Root_for_spheres_2_3

**Constructions and predicates:**
- Constructors for algebraic types from geometric objects
- Solve
- Sign_at
Example: Using algebra for geometric constructions

SK::Intersect_3

SK::Circle_3
diametral_sphere() supporting_plane()

SK::Sphere_3 SK::Plane_3

SK::Get_equation

AK::Polynomial_for_spheres_2_3 AK::Polynomial_1_3 AK::Polynomial_1_3

AK::Solve

SK::Root_for_spheres_2_3

SK::Circular_arc_point_3
Application: specialization

**Spherical Bentley-Ottmann**

[Cazals, Loriot06].

---

**Input**
- A central sphere
- Set of spheres intersecting the central one (or set of planes)

**Output**
- HDS containing faces of the arrangement of intersection circles
- For each face, a list of sphere which ball covering it
Specialization on a given sphere

Natural extension to handle objects on a common sphere using cylindrical coordinates

Primitives:
- Circle_on_reference_sphere_3
- Circular_arc_point_on_reference_sphere_3
- Circular_arc_on_reference_sphere_3
- Theta_rep

Predicates:
- Compare_theta_3
- Compare_z_at_theta_3
- Compare_z_to_left_3

Constructions:
- Intersect_3
- Make_theta_monotonic_3
- Theta_extremal_point_3
Example: `Compare_theta_3`

Bentley-Ottmann on a sphere $\mapsto$ sort event points (critical and intersection points) according to $(\theta, z)$ value.

Our implementation:

- Need `Theta_extremal_point_3` for critical points
- Need new `Intersect_3` for intersection points

- Each event point is in a hquadrant
- We obtain $\tan \theta$ and $\cot \theta$ as AN of degree 2

$\mapsto$ comparison of $\theta$ coordinates:
- compare hquadrant indices
- compare AN of degree 2
Needs of the Spherical BO

- Predicates
  - Initialize vertical ordering
    \( (\text{Compare}_z\text{to}_left\_3,\text{Compare}_z\text{at}_\theta\_3) \)
  - Sort event points using cylindrical coordinates
    \( (\text{Compare}_\theta\_z\_3) \)
  - Detect/create intersection points
    \( (\text{Do}_\text{intersect}_3,\text{Intersect}_3) \)
  - Insert circle starting \( (\text{Compare}_z\text{at}_\theta\_3) \)

- DS: Face, Vertex and halfedge types for HDS encoding arrangement of circles

- Gauss-Bonnet formula to compute approximate area of a spherical face.
Illustration

Video
Improvements

Several level of filtering

- Arithmetic Filtering
  - Computation on intervals: failure implies exact computation
- Filtered constructions
  - `CGAL::Lazy_kernel`: a construction create one node in the dag (vs set of operations) [Fabri, Pion06]
- Geometric Filtering
  - Using predicates on Bbox
- Static Filters [Melquiond, Pion05]
  - Design bounds for arithmetic operations in order to guarantee double computations (specific to each predicates).

We take advantage of these strategies to design filtered version of kernels.
Conclusion and Future work

The SphericalKernel provides

- a generic framework for algorithms involving spheres
- Robust and efficient primitives and predicates
- extension for manipulating circle arcs on a common sphere

Future work

- Traits class for Arrangement_2 for arrangement of circle on a sphere using the Spherical Kernel.
- Efficient DS to encode arrangement of spheres.
typedef CGAL::Quotient< CGAL::MP_Float> NT;
typedef CGAL::Cartesian<NT> Linear_k;
typedef CGAL::Algebraic_kernel_for_spheres_2_3<NT> Algebraic_k;
typedef CGAL::Spherical_kernel_3<Linear_k,Algebraic_k> SK;

int main(){
    //construction of 3 spheres from their centers and squared radii
    SK::Sphere_3 s1(SK::Point_3(0,0,0),2);
    SK::Sphere_3 s2(SK::Point_3(0,1,0),1);
    SK::Sphere_3 s3(SK::Point_3(1,0,0),3);

    SK::Intersect_3 inter;
    SK::Compare_xyz_3 cmp;
    std::vector< CGAL::Object > intersections;
    inter(s1,s2,s3,std::back_inserter(intersections));

    std::pair<SK::Circular_arc_point_3,unsigned> p1,p2;
    //unsigned integer indicates multiplicity of intersection point
    if (intersections.size() >1){
        //as intersection can return several types (points with multiplicity, circle,...),
        //CGAL::Object and CGAL::assign are used to recover the expected type
        if (CGAL::assign(p1,intersections[0]) && CGAL::assign(p2,intersections[1]))
            std::cout << "Two different intersection points" << std::endl;
        else
            std::cout << "Error" << std::endl;
    }
    //intersection points are sorted lexicographically
    CGAL_assertion(cmp(p1.first,p2.first)==CGAL::SMALLER);
    return 0;
}