

A Generic Geometry Library

Barend Gehrels
Geodan Holding B.V.
Amsterdam
the Netherlands
barend.gehrels@geodan.nl

Bruno Lalande
Paris
France
bruno.lalande@gmail.com

ABSTRACT

The Generic Geometry Library (GGL) is a library for geometry and geography. It is based on templates and static polymorphism. It is agnostic in coordinate space, coordinate system, and dimensions. The library is non intrusive and uses concepts for geometries for its algorithms. Developers can either use the provided geometries or define or use their own geometry types and specializations. Essential algorithms such as distance calculations, area calculations, point in polygon calculations are available. A geometry library like described would be a valuable addition to the Boost Library Collection.

Keywords

Geometry, geography, GIS, Boost, library, projections, maps, points, lines, polygons.

1. INTRODUCTION

The Boost Library Collection currently still does not have any geometry library. In the past years several discussions and previews have been published on the Boost Mailing Lists. There is certainly interest for a library handling geometry within the Boost community; it would be valuable if geometry algorithms are available for the Boost users. Such algorithms are for instance: the distance between two points, the area of a polygon, or determining if a point is in a polygon.

Geometry is used in many disciplines and there are many approaches possible. Depending on their interest programmers feel other needs. Developers creating games will need other geometries, other algorithms, other dimensions and coordinate representations than developers creating CAD, geographic applications, astronomic or control software for robotics. Besides that, programmers have their personal flavors.

It is a major challenge to develop a consistent library containing every aspect of geometry, on which everyone agrees.

This article describes one attempt of a geometry library. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

BoostCon'09, May 3-8, 2009, Aspen, Colorado, US.
Copyright 2004 ACM 1-58113-000-0/00/0004...\$5.00.

Generic Geometry Library (GGL) developed and restyled in 2008 has been presented, at the time of writing, as a preview three times. With the feedback from the Boost community the library is enhanced and extended, it has been changed a lot and the library is better than it was before. It is still in preview phase and not yet reviewed nor accepted. There are other candidate libraries for geometry as well. This paper describes the characteristics and basic principles of the Generic Geometry Library.

The Generic Geometry Library has been initiated by Geodan. Geodan is a company for Geographic Information Systems. Bruno Lalande, who has contributed to Boost libraries before, joined the team of developers.

2. GEOMETRY CONCEPTS

A geometry library should define geometry concepts and should implement algorithms working with those concepts. The library can then handle any geometry type modeled according to the concept.

The task of selecting geometries and giving them proper names can already be a challenge and a subject for much discussion. The Wikipedia list of geometric shapes contains more than 30 different polygons, furthermore a dozen of triangles, rectangles, squares, kites, trapezoids, circles, ellipses, three dimensional shapes as polyhedrons, spheres. The list is probably never ending.

The Generic Geometry Library is not attempting to create concepts for all these shapes. Instead, it models four concepts of basic geometries:

- a **point**
- a sequence of points (a **linestring**)
- a point sequence which is closed (a (linear) **ring**)
- a **polygon** (an outer ring which can optionally contain inner rings, also called donuts or holes)

With those four concepts, all two dimensional shapes composed of straight line segments (triangles, rectangular and polygonal, etc.) can be built. How to handle curved geometries in two dimensions is not yet worked out. An equivalent system might be used where segments between three points are marked as *curved*.

The Generic Geometry Library defines concepts and provides default implementations of these concepts. Besides these predefined types, other types can be handled as well. Any point type that defines the required properties, at compile-time, and thus satisfies the point concept requirements can be used.

The linear ring and the linestring concepts follow Boost Range Library concepts. Ranges are used to iterate through sequences of points. Therefore algorithms as *length* or *area* also work on a `std::vector` of points. The names of these concepts and of the provided geometries are consistent with the naming system of the OGC, described below. The names are also used for tags related to the tag dispatching mechanism, which is used internally to distinguish geometry types.

For convenience three other concepts are defined:

- a **box**: to be able to select things, for spatial indexing, and to keep track of boundaries of geometries
- an **n-sphere**: circle or sphere, to be able to select things
- a **segment**: a part of line bounded by two points, to calculate e.g. distances and intersections

Besides defining custom types, library users can make specializations of algorithms for their geometry types and in such way implement support for, for example, triangles. This will be worked out below.

The Boost Concept Check Library (BCCL) is used internally to check if the developer's input follows the concepts expected by the library.

3. DIMENSIONS

The Generic Geometry Library is basically agnostic in its dimensions. A point is templated by the number of dimensions, by coordinate type (integer, float, double, GMP, ...), and by the coordinate system.

Examples:

- `point<double, 2, cartesian>`: a point in two dimensions
- `point<double, 3, cartesian>`: a point in three dimensions
- `polygon<point<double, 2, spherical<radian>>>`: a spherical polygon, measured in radians

Although points of any dimensions can be created, most algorithms are currently not prepared to handle all possible dimensions. Most algorithms that are provided handle points in Cartesian, Geographic or Spherical coordinate systems.

A template parameter defining how to handle calculations in algorithms using big number types might be added in the future.

4. COORDINATE SYSTEMS

A generic library handling geometry should be neutral in its coordinate space. In geography, points can be specified using X and Y (and Z), but also using Latitude and Longitude (and Height). In other disciplines there might be need for other names, for example: red, green and blue (defining colors in a color cube), or right ascension and declination (in astronomy).

The concepts of the Generic Geometry Library use therefore neutral names (get and set) for accessing and modifying coordinates. Example, setting a point to (3,2):

```
geometry::point<double, 2, cartesian> a;
geometry::set<0>(a, 3);
geometry::set<1>(a, 2);
```

Or, alternatively:

```
geometry::assign(a, 3, 2);
```

These calls are generic. Library users might also choose to use or define more context targeted methods as **a.x()** and **a.y()** for setting and getting coordinates, or non default constructors. This as long as they also provide the specializations needed to enable the library to play with points the way shown above.

The Generic Geometry Library provides a generic **transform** algorithm which can transform geometries from one coordinate system to another.

5. TYPES AND STRATEGIES

Algorithms depend on coordinate system. Let's take, for example, the distance calculation between two points. For points in a Cartesian coordinate space the simple and well known Pythagorean theorem applies. For points in the geographic coordinate system (latitude longitude, also known as latlong, lola or ll) the distance can be calculated using the haversine formula (it is possible to select other formulas as well, which is described below).

The Generic Geometry Library provides a tag dispatching system which selects strategies based on coordinate system. So distances and areas for spherical coordinates are internally calculated differently then distances and areas for Cartesian coordinates. This is, by default, not visible for the library user, it is an automatic tag dispatching system.

The tag dispatching system is also used to distinguish geometry types internally. So the generic *distance* algorithm can get two points (in any coordinate system) as input, but also a point and a linestring, a point and a polygon, et cetera.

Example showing different strategies for different coordinate systems:

```
geometry::point_xy<double> a(1,1), b(3,4);
std::cout << geometry::distance(a, b);

typedef geometry::point_ll<> LL;
LL A, B;
// initialize lat-long coordinates
// e.g., Amsterdam and Barcelona
// ...

std::cout << geometry::distance(A, B);
```

This example uses two different point types. In the first part the Pythagorean theorem is used to calculate the distance between point a and point b. In the second part the distance along the

sphere of earth is used (as the crow flies). Note that the non default constructor shown in the first line is not part of the concept; it is part of the implementation (so part of the example above).

As described, distance is calculated using strategies for different coordinate systems. The distance algorithm selects automatically right specialization belonging to the geometry type, and then it selects the strategy belonging to the coordinate system. This strategy is then used for calculation.

Algorithms in the Generic Geometry Library also have an overload where the strategy can be specified by the library user. This is useful for, for example, calculating distance along the earth (or along any sphere). There are several methods for that calculation, using internally either a fast or a precise algorithm. The library user can select such a strategy or can implement his or her own strategy. For example, to use the exact Vincenty [4] geodetic calculation (provided by the library):

```
std::cout << distance(A, B,
    strategy::distance::vincenty<LL>())
    << std::endl;
```

Algorithms as simplification (removing non important points from point sequences such that shape is preserved) use, internally, distance calculations. The same mechanism is used there. Therefore simplification works for Cartesian polygons the same as for polygons with geographical coordinates. It is exactly the same implementation.

6. SPECIALIZATIONS

Besides strategies, library users have also other options for customization. For example: the Generic Geometry Library itself does not support triangles. Instead it supports polygons (which might contain holes) and linear rings. However, if the library user wants to use triangles directly, he can implement his own type. Let's work this example out. A triangle can be represented by three coordinate pairs. The library user can make a triangle of his own points (in this case: `custom_point`), using e.g. a `boost::array`.

```
struct triangle : public
    boost::array<custom_point, 3>
{};
```

The user then has to indicate that this geometry is handled as a linear ring. He therefore has to provide a specialization of the tag metafunction, defining a type for the tag dispatching mechanism. This should reside in the namespace `geometry`:

```
namespace geometry
{
    template <>
    struct tag<triangle>
    {
        typedef ring_tag type;
    };
}
```

As soon as this is done, the library will automatically select the ring specializations for all triangle algorithms. Because the Generic Geometry Library handles a ring as being a range, the user does not have to provide anything more. The Boost Range library concepts are used internally for all iterations.

So the area of the user's custom-points-triangle will be calculated correctly, using internally an iterator which walks through the point coordinates.

But wait, for triangle that is not as efficient as possible! No problem, the developer can, if desired, implement a specialization for the area function:

```
template<>
double area<triangle>(const triangle& t)
{
    return 0.5 * (
        (t[1].x() - t[0].x())
        * (t[2].y() - t[0].y())
        - (t[2].x() - t[0].x())
        * (t[1].y() - t[0].y())
    );
}
```

Now this specialization will be used to calculate the area of the developer's triangles, while for other algorithms (centroid, within, intersection, et cetera) the generic ring versions still being used.

The Generic Geometry Library also provides a second way to specialize. Within the dispatch namespace, structs are available for all operations. They can be partially specialized such that, for example, a triangle templated by point type will be handled correctly by the library.

7. NAMES AND CONVENTIONS

All geometry types and algorithms are named following the specifications of the Open Geospatial Consortium (OGC) [3]. The naming system of the OGC is carefully thought-out and has been agreed upon by a broad community. Additional algorithms and geometry types are named as much as possible using the same conventions and systematics.

OGC conventions, geometry types and spatial operations are widely used, a.o. in spatial databases, such as PostGreSQL, MySQL, Oracle, and SQL Server.

The OGC defines the following algorithms:

- Basic algorithms: dimension, envelope, as_text, as_binary, is_simple, is_empty
- Query algorithms: equals, disjoint, intersects, touches, crosses, within, contains, overlaps, relate, locate_along, locate_between
- Analysis algorithms: distance, buffer, convex_hull, intersection, union, difference, sym_difference
- Geometry type dependant algorithms: area, length, centroid, point_on_surface

All algorithms are planned to be implemented. Besides this there are some other algorithms provided or planned:

- Simplify, densify, disperse, spline, triangulate
- Coordinate system conversions: transform

The last one, transform, transforms geometries from any coordinate system to any other coordinate system. For example from radian to degree, but also from spherical to a 3D X,Y,Z coordinate system, or calling a map projection.

Textual representations of geometries (Well-Known Text or WKT) are also provided by the library, as well as the parsing of them.

The OGC also defines geometry types containing a collection of geometries, which are planned to be supported:

- multi_point
- multi_linestring
- multi_polygon

The Generic Geometry Library is not OGC compliant in the strict sense. It sometimes deviates from the specifications to enable genericity. It uses Boost Range concepts for the linestring geometry type. The library user can therefore use any standard container or Range compatible type as a linestring (a sequence of points). And the output of intersections is, for example, modeled as an output iterator instead of a multi_polygon or a multi_linestring.

8. SPATIAL INDEXES

In geometry or GIS, spatial indexes can be used to search geometries or to enhance or speed up determination of geometry relationships. In 2008, a student, within the framework of the Google Summer of Code, has worked out a first version of a spatial indexing library which uses the Generic Geometry Library. This will be worked out and extended in the future.

The Priority R-Tree, an R-Tree having optimal performance for any query window [1], will be available for fast searches within a set of geometries.

9. MAP PROJECTIONS

The Generic Geometry Library can be used as a generic library for Geographic Information Systems (GIS). It therefore contains map projections. Map projections are algorithms to map the spherical earth to a flat piece of paper. There are many ways to do this. The USGS has implemented over the years 92 projection algorithms, collected in an often used PROJ4 package, written in C. This package is recently fitted into the Generic Geometry Library by converting it to a set of C++ templates.

The projection collection contains both static as dynamic polymorphism. It is possible to instantiate a projection declaring the wished projection itself, for example:

```
typedef geometry::point_ll<double,
    geographic<> > LL;
typedef geometry::point_xy<double> XY;
projection::merc_ellipsoid<LL, XY> prj(
    "+ellps=WGS84 +lon_0=11.6E");

// Define Amsterdam / Barcelona
LL amsterdam, barcelona;
// Initialize A'dam / B'ona
...
XY ma, mb; // Declare points
// and project (from latlong to XY)
if (prj.forward(amsterdam, ma)
    && prj.forward(barcelona, mb))
{ ... }
```

This will use static polymorphism. Note that the initialization string is conform PROJ4, but it is also possible to initialize projections with e.g. a std::map.

Static polymorphism is also used if the already mentioned *transform* algorithm provided by the library is used, a generic algorithm for transforming or projecting geometries:

```
// declare type UTM zone 30, Cartesian
// using EPSG code 2040
typedef geometry::point_xy<double,
    epsg<2040> > UTM;
// Assign Amsterdam / Barcelona in UTM
...
// call generic transformation
geometry::transform (amsterdam, ma);
geometry::transform (barcelona, mb);
```

EPSG has defined the accepted standard coding for map projections [2], worldwide, and an EPSG code can be used as a template parameter in the provided EPSG Cartesian coordinate system.

Besides specific instantiation, any projection can also be instantiated using a factory:

```
projection::factory<LL, XY> fac;
boost::shared_ptr<projection
    ::projection<LL, XY> >
prj(fac.create_new(parameters));
```

The projection instance can now be any projection, created by the factory according to its parameters (which should contain the name of the projection). The instantiated projection-base-class derived class contains virtual methods for forward and inverse projection, but is still also modeled using template parameters for its point types.

10. CONCLUSIONS

The Generic Geometry Library as described and published in preview is an extensive generic template geometry library containing many algorithms. It can be used for simple algorithms such as distance calculation but it can also be used in larger contexts such as GIS or Web Mapping applications. It is an Open Source library following the Boost Software License. The Boost Community would profit from a geometry library as described in this article.

11. REFERENCES

- [1] Arge, L, Berg, M de, Haverkort, H.J. and Yi, K. (2004) The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree <http://www.daimi.au.dk/~large/Papers/prtreesigmod04.pdf>
- [2] EPSG (2008) EPSG Geodetic Parameter Dataset <http://www.epsg.org/CurrentDB.html>
- [3] OGC (2006) The OpenGIS® Simple Features Interface Standard (SFS) <http://www.opengeospatial.org/standards/sfa>
- [4] Vincenty, T. (1975) Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations Survey Review 22 (176): 88-93. http://www.ngs.noaa.gov/PUBS_LIB/inverse.pdf