A Library for Managing Spatial Context Using Arbitrary Coordinate Systems

Thomas Schwarz, Nicola Hoenle, Matthias Grossmann, Daniela Nicklas
Institute of Parallel and Distributed Systems, Universitätsstr. 38, 70569 Stuttgart, Germany
<firstname>.<lastname>@informatik.uni-stuttgart.de

Abstract

Since location is an important part of context, the management of spatial information is important for many context-aware applications, e.g. the position or the extent of users, sensors, rooms or buildings. Coordinates always have a coordinate system (CS) associated to them. Numerous CSs exist and a lot of them are commonly used, thus conversion becomes a necessity. We introduce a library that implements the OGC Simple Feature Specification and can dynamically cope with different CSs, enabling interoperability between applications, middleware components and data providers. We illustrate functions and features, describe a common CS determination algorithm and point out our lessons learned: avoid transformations, use existing standards but dare to extend them when needed.

1. Motivation

Context information is vital for ubiquitous computing applications. Especially location is an important part of context information: applications need to know about the position or the spatial relations of users, devices, sensors and actuators [1], [2]. This information comes from different sensors or external models, and the application embeds it in its local model. The representation of location can be symbolic (just names), topological (explicit modeling of spatial relations) or geographical (coordinates). A geometric or geographical model is most flexible: with spatial predicates and queries, lots of tasks can be solved (inside, overlap, nearest neighbor), and symbolic or topological information can easily be embedded into the model. Thus, the management of spatial information is a common building block for context-aware applications and middleware components.

Coordinates always have a coordinate system (CS) associated with them, see Figure 1. Numerous CSs exist and many of them are commonly used, e.g. WGS84 or NAD83. Indoor applications often use a local CS whose origin is a point within the room or building. Outdoors, the position can be obtained from a Global Positioning System (GPS) sensor in WGS84 format. Spatial databases, geographical information systems or map data use even more

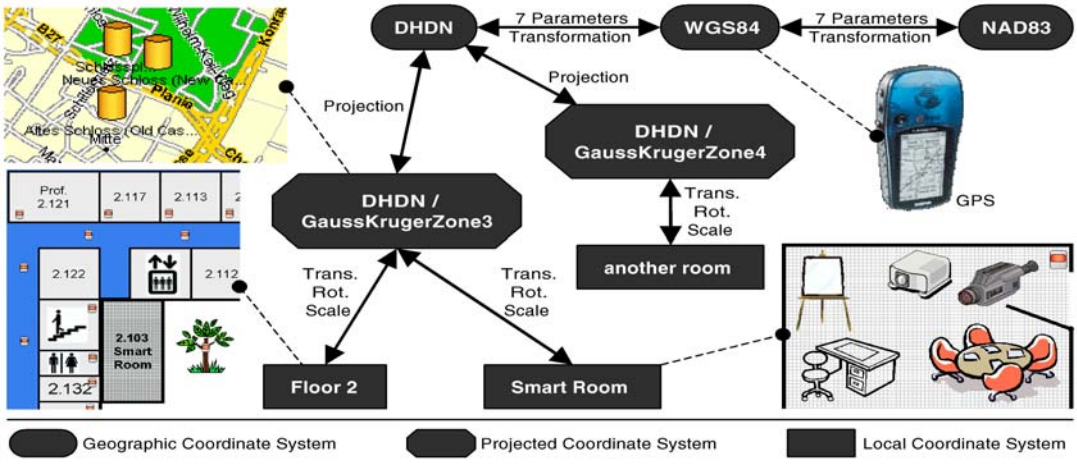

Figure 1. Various data sets may utilize different coordinate systems (CSs). CSs are the nodes in the transformation graph, linked by transformation rules (e.g. projection).

CSs, e.g. one of the 2824 different CSs contained in the EPSG database [10].

This heterogeneity of coordinate systems becomes a problem, when you

- want to have two or more applications to interoperate,
- want to integrate different location sensors (like infrared beacons, GPS),
- want to integrate data from different sources (spatial databases, directories,...), that are surveyed by different people, organizations and providers,
- want to build a middleware for context-aware applications (see above points).

We ran into this problem while developing a distributed middleware for context-aware applications, that consists of several different components, and applications for it [3]. After two years, there were five different packages to represent and manage geographic data. To harmonize this, we looked for a standard library for geo data, and found the Java Topology Suite (JTS) [8] implementing the Simple Features Specification (SFS) of the Open GIS Consortium (OGC) [5]. Unfortunately, this library does not care about different CSs. Using a single global CS for all spatial data provided by the platform does not solve this problem. All cartesian CSs are only usable within a small area on the earth's surface (see Section 2.2), so the global CS had to be geographic, which is not supported by the JTS library.

Our solution was to develop a library for geodata management based on the JTS, that fully implements the OCG standard and furthermore allows an easy and transparent usage of different CSs in parallel. The conversion policy for coordinates is "as seldom, as late as possible" and original data is always kept as long as possible. This library can be used in applications and middleware components and leads to better interoperability and maintainability.

In this paper, we illustrate the functions and features of the library (Section 2), describe a common coordinate system determination algorithm in Section 3 and give a conclusion on the lessons learned and the next steps.

2. Functions and Features

The library is based on the OGC SFS for Corba [4] which defines geometric data types, functions and operations on them, and spatial reference systems. Points can be used for objects with no relevant extent, e.g. users or small items like sensors or actuators. Polygons represent areas, like the extent of a room or the transmission range of an infrared beacon. Routing information or roads can be modeled using linestrings. In this chapter, we will describe the functions and features of the library.

2.1. Basic geometry types and functions

Geometry data can be categorized by its dimension: Points are zero-dimensional, curves are one-dimensional, and surfaces are two-dimensional. Three and more dimensional geometric values are not considered in the SFS. Beside simple geometries like individual points, curves, and surfaces geometry collections are defined. Geometry collections are generally heterogeneous, however there are specialized subclasses restricting the entries to a specific single geometry type. We implemented this class hierarchy in Java. Our internal representation of geographic objects includes the geometry itself (geometry type and coordinates), the associated coordinate system (CS) in which the coordinates are defined, and optionally the last created JTS object (see next section) and the CS associated to the JTS object. All coordinates can be given in three dimensions. However, the third coordinate is ignored by all computational geometry algorithms of the JTS.

For geometries, there are functions testing spatial relationships like Equals, Disjoint, and Overlaps, based on Egenhofer's Nine Intersection Model [6]. New geometries can be computed using spatial operators like Intersection, Difference, and Union.

In our implementation, all geometry calculations are delegated to the JTS. Input geometries for JTS methods have to be in the same cartesian CS. Our wrapper satisfies this requirement by choosing a cartesian CS, converting the geometry data to the selected CS if necessary, and constructs JTS objects from the converted geometry data before calling JTS methods. If more than one geographic object takes part in the same geometric operation (e.g. Intersection), the same CS must be chosen for all objects. The created JTS objects are cached within the geographic objects for further calculations. Choosing the right CS and converting the geometry data into this CS is hidden from the applications.

2.2. Coordinate Systems

Because the SFS is targeted at representing geometries of real world objects, there are two main spatial reference system types: Geographic CSs approximate the shape of the earth as an ellipsoid. The coordinates are specified by latitude and longitude (e.g. WGS84). Projected CSs define a cartesian coordinate plane. Points on the earth's surface are mapped to points on the plane using the projection rule associated with this projected coordinate system (e.g. DHDN/Gauss-Kruger zone 3). Indoor applications often use a local cartesian CS which can be seen as a special case of a projected one.

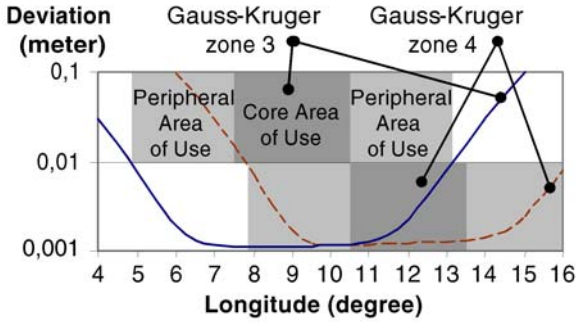

Figure 2. Deviations at varying longitudes after round-trip conversion: WGS84 -> DHDN/Gauss-Kruger zone 3/4 -> WGS84

2.3. Coordinate transformations

The transformation between our current coordinate systems WGS84 and DHDN/Gauss-Kruger zone 3 is calculated by a transformation from WGS84 into the DHDN geographic CS, followed by a projection of the DHDN geographic CS into the DHDN/Gauss-Kruger zone 3 projected CS.

Projections and transformations are reversible, but not without loss, i.e. there are slight differences between the original coordinates and the result coordinates after one forward and one backward conversion step. Figure 2 shows this deviation between the original point and the round-trip converted point depending on the longitude of the point. All points have the same latitude of 51° North, and at other latitudes a similar deviation can be observed. In this figure we can see two things: first, every conversion inevitably incurs some imprecision, the lowest attainable deviation being about 1 millimeter. Secondly, only within each projected CS's core area of use is the deviation minimal. Outside this core area the deviation increases exponentially. Within the peripheral area of use the deviations are tolerable.

Conversions should be computed rarely. Therefore we compute conversions as late as possible and the system caches converted versions of geometries after comparisons or calculations for further operations on them.

Additional CSs, transformations and projections can be defined by adapting the relevant classes, e.g. for integrating an indoor local CS.

2.4. Input and output of geometries

Geometry data can be serialized in order to export or import data. We do not use the Java serialization mechanism because the serialized objects are not compatible with systems such as spatial databases.

The input and output format of geometries in our library is Well Known Text (WKT) and the XML-based Geographic Markup Language (GML) [7].

In GML, the identifier of the coordinate system is part of the geometry:

```
<gml:Point srsName="4326">
  <gml:coordinates> 10 20
</gml:coordinates>    </gml:Point>
```

WKT is also part of the SFS specification and a standard for geo data exchange, all databases with SFS-compliant spatial extensions support WKT, e.g. DB2 Spatial Extender and Oracle Spatial. The WKT format encodes only the coordinates of the geometries but not their CS. Therefore we enclose WKT geometries into an XML element `<wkt>`, which has an attribute `srs` to denote the CS:

```
<wkt srs="4326"> POLYGON (10 10, 10 20,
20 20, 20 15, 10 10) </wkt>
```

3. Common coordinate system determination algorithm

As we introduced in Section 2.1, geometries need to be converted to a common cartesian coordinate system in order to be processed or compared by the JTS library. At first glance this seems to be trivial, but after further consideration this involves two nontrivial tasks: First, to determine the most suitable common CS, and secondly, to transform both geometries into this CS.

In [10], a list of CSs and rules for direct transformations between pairs of CSs is maintained, but transformation rules are defined only for a few pairs of CSs. The list of transformation rules can be interpreted as a graph with CSs as the nodes and existing transformation rules as the edges, see Figure 1. It is possible to calculate a transformation from a source CS into a target CS, if a path in the transformation graph between the two CSs exists. If more than one path exists, the shortest path should be chosen because transformation calculations always incur imprecisions.

A local CS can be easily incorporated into this transformation graph by defining a transformation rule relating it to a well-known projected CS. Such a transformation rule can be composed of a translation, a rotation and a scaling factor to map local coordinates to projected ones. If this is not sufficient, our library allows to include custom transformation code as well. Now, coordinates in a local CS (e.g. Smart Room) can be compared to coordinates in a different local CS (e.g. Floor 2), or to coordinates in any other CS (e.g. WGS84) that is reachable in the transformation graph starting from the local CSs node.

3.1. Choosing a common CS

Projected, and therefore cartesian, coordinate systems are more convenient for the calculation of geometric predicates, e.g. if two polygons are overlapping. Efficient algorithms for computing the spatial predicates exists, and also (Java-) implementations for these algorithms. We are using the JTS, which requires the coordinates to be cartesian. We want to minimize the number of conversions, because conversions always lead to imprecisions.

Parameters of projected CSs are determined to get the best attainable precision for a specific section of the earth's surface. Outside of this specific section the precision decreases constantly (and distortions increase). Thus a projected CS is characterized by a core and a peripheral area of use, with little projection errors in the core area, tolerable errors in the peripheral area, and significant errors outside the peripheral are. In Figure 2 the core area of use of the DHDN/Gauss-Kruger zone 3 CS is between 7°30' and 10°30' East (and between 47° and 56° North). Core and peripheral areas of use are given as bounding boxes in WGS84 coordinates.

A common cartesian CS for two given objects is chosen as follows: If the two objects already have a common cartesian CS (original or cached), then this CS is the common CS and no transformations are necessary.

If this fails, then check if the original or cached CS of one object can be used. This CS has to be cartesian, and the other object has to lie within the peripheral area of this CS. If more than one original or cached CS fulfill these requirements: Select the original CS.

If this fails also, determine a new common cartesian CS. For each object build a list of suitable cartesian CSs in the following four incremental steps

1. Start with all CSs where the object lies inside the core area.
2. Add all CSs where the object overlaps with the core area.
3. Add all CSs where the object lies inside the peripheral area.
4. Add all CSs where the object overlaps with the peripheral area.

After each step check if both lists contain a common CS. Continue with next step if none is found. If more than one common CS is found, choose the one with the shortest path in the transformation graph. If after these four steps still no common CS has been found, no such one is available for these two objects. Throw an error.

A possible extension of this algorithm is to consider the preferences of an application while choosing the common CS.

Up to now we use only two CSs as described above, so DHDN/Gauss-Kruger zone 3 is always chosen as the common cartesian CS. But we want to extend our system to import databases with CSs and transformation rules as for example defined in [10].

4. Related work

There exist several other Java-based implementations of the SFS, JTS and deegree being the most complete ones. The JTS implements only the geometry part of the SFS, including all the computational geometry functions. It lacks, however, any capability to deal with different CSs.

The deegree project [9] is the successor of the sfcorba2java project. While the latter one intended to implement the geometry part of the SFS, the deegree project now implements several of the newer web standards proposed by the OGC like the Web Map Service Implementation Specification (WMS) and the Web Feature Service Implementation Specification (WFS). Some packages of this project provide a functionality that is quite similar to what is specified in the SFS. This includes the geometry package, the feature package, the CSs package, and the coordinate transformation package. The class hierarchy within the geometry package resembles the SFS structure, however, the only supported computational geometry operation is "intersects". Geometries having different CSs are not automatically converted into a common one. Not even an error is thrown in such a case which essentially allows one to compare apples and pears. Instead, the application itself needs to initiate all conversion work, and it also needs to find a suitable common CS by itself. We are currently investigating if we can include their coordinate transformation package into our library, in a similar way as we did with the JTS.

In the Spatial Part of the SQL99 standard the function ST_Transform is defined, which is supposed to perform arbitrary coordinate conversions. However, it is not always implemented like this. IBM's Spatial Extender, e.g., can only do (inverse) projections, but not transform from one geographic CS into another one. Local CSs can be used in the database, but they exist in isolation and cannot be related to any other CS.

The transformation policy is rather simple. Functions, which take two geometries as arguments usually convert the second geometry into the CS of the first.

5. Conclusion

Support for different coordinate systems (CSs) and transformations between them is an important feature of location-based applications. It simplifies the adoption of existing spatial data, because no transformation into a glo-

bal CS is necessary, as well as the acquisition of new data, e.g. by allowing the definition of a CS that is specifically tailored to a building whose rooms are to be acquired. Still each spatial object is comparable to every other one even if their CSs are different, if appropriate CS transformations are used.

Especially for representing the floors and rooms of a building, a topological approach would be even more suitable. In that approach, objects like floors and rooms are not represented by shapes with geographical coordinates, but the inclusion relationships between them are explicitly modeled. In conjuction with a symbolic naming scheme for spatial objects, this allows spatial queries to be processed without any geographical information.

The topological approach is very convenient for indoor scenarios, but for outdoor scenarios, the situation is different: a mature positioning system (GPS) which uses geographic coordinates is available, as well as data sets containing shapes of countries, cities or even buildings using geographical coordinates. Thus, for a universal platform for location-based services supporting both types of coordinates would be beneficial. The seamless integration of symbolic coordinates and topological relations with geographical coordinates is subject to ongoing research.

Adopting the OGC SFS turned out to be a great advantage. This allowed us to use the existing JTS library instead of implementing all geographical operations ourselves, thus reducing the implementation effort by around 72% of lines of code.

With the library, it is now possible to integrate data of any CS even dynamically without changing the application. This becomes especially important when your application should work at the borderline of different geographical zones, e.g. indoor and outdoor, or across state borders. This offers better flexibility of the application, if you want to transfer your application out of your test-bed to another environment.

The SFS is sufficient for modeling spatial aspects of context-aware applications. The only drawback is the lack of circles. Circles have some benefits: it is a straight-forward way to model accuracy of points, and as long as you stick on circles and points, inside and overlap predicates are easily computed. But if you start to mix circles with polygons, things are getting tricky: the result of the intersection of a circle and a polygon is none of both. A solution for this problem is to introduce a datatype circle that is converted into an approximated polygon if necessary. For this conversion, a similar conversion policy as for coordinate reference systems can be used.

We found that the library is a common building block for context-aware applications, middleware components and data providing services. It ensures interoperability and flexibility.

6. References

[1] Weiser, M.: The Computer for the Twenty-First Century. Scientific American, pp. 94-100, September 1991.

[2] Dey, A., Abowd, G.: Towards a better understanding of context and context-awareness. Georgria Tech GVU Technical Report, GIT-GVU-99-22, 1999.

[3] Nicklas, D., Grossmann, M., Schwarz, T., Volz, S., Mitschang, B.: A Model-Based, Open Architecture for Mobile, Spatially Aware Applications. Proceedings of the 7th International Symposium on Spatial and Temporal Databases: SSTD 2001; Redondo Beach, CA, USA, July 12-15, 2001

[4] Open GIS Consortium: Simple Features Specification for Corba. Version 1.0, 1998. `http://www.opengis.org/techno/sfr1/sfcorba_rev_1_0.pdf`

[5] Open GIS Consortium: Simple Features Specification for SQL. Version 1.1, 1999. `http://www.opengis.org/techno/specs/99-049.pdf`

[6] Egenhofer, Max J.; Herring, John R.: Categorizing Binary Topological Relationships Between Regions, Lines, and Points in Geographic Databases. Technical Report, Department of Surveying Engineering, University of Maine, Orono, ME, 1991.

[7] Open GIS Consortium: Geography Markup Language (GML) Implementation Specification. Version 2.0, 2001.
`http://www.opengis.net/gml/01-029/GML2.html`

[8] Vivid Solutions: Java Topology Suite. Version 1.2, 2002.
`http://www.vividsolutions.com/jts/jtshome.htm`

[9] Deegree Java framework for geospatial solutions, founded by the GIS and Remote Sensing unit of the Department of Geography, University of Bonn, and lat/lon.
`http://deegree.sourceforge.net/`

[10] European Petroleum Survey Group (EPSG) Geodesy Parameters V 6.3. `http://www.epsg.org/`