Assessment of Geospatial Data Access and Operations of the Multi-Platform Open-Source HaptiMap Toolkit


Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.
Spatial Data Computations in an Accessibility Improving Toolkit for Mobile Applications

Janne Kovanen*, Paul D. Kelly**, Stuart Ferguson**, L. Tiina Sarjakoski*, Tapani Sarjakoski*

* Finnish Geodetic Institute, Dept. of Geoinformatics and Cartography, P.O. Box 15, FIN-02431 Masala, Finland
{firstname.lastname}@fgi.fi
** Queen’s University Belfast, School of Electronics, Electrical Engineering and Computer Science, Ashby Building, Stranmillis Road, Belfast BT9 5AH, Northern Ireland
{paul.kelly, r.ferguson}@ee.qub.ac.uk

Abstract. The HaptiMap toolkit is an open-source library enabling location-based mobile application developers to include accessibility-enhancing features into their own applications. This paper presents the structure of the geospatial data access of the HaptiMap toolkit. In addition, an evaluation is carried out on the solution and processing performance of the toolkit. The operation performances are compared with the corresponding operations of a leading open-source library, which supports computational geometry calculations. The results indicate that implementing a custom optimized algorithm for the toolkit’s data model is a feasible approach when the algorithms are specific or simple enough, while complex algorithms should be accessed through wrapped functions.

Keywords: Geographic information, computational geometry, accessibility, open-source, audio, tactile

1. Introduction

The HaptiMap toolkit is a cross-platform, open-source library that enhances accessibility for mobile location-based applications. The target group of the library is applications that want to enhance the perception of visually impaired users, and the applications are meant especially for pedestrian navigation. The toolkit is built as part of the European Commission-funded HaptiMap project (Magnusson et al., 2009), the
The overall objective of which is to make maps and location-based applications more accessible by using touch, hearing and vision senses. To reach a wide audience, the toolkit can be used on a variety of popular mobile platforms, including Android, iPhone, Windows Mobile, Meego and Symbian.

The toolkit is a collection of application programming interfaces (APIs). Some of the interfaces are only available internally to support others, and the interfaces have different levels of complexity and functionality. Hence, general API development rules and principles, such as robustness, durability, modularity and ease of use, apply to the toolkit. The toolkit design concentrates especially on addressing the usability of the toolkit by its audience. For instance, the intended target group of the internal APIs is mainly composed of human-computer interaction designers who want to share their knowledge on how to make mobile applications more accessible for navigation, whereas the public APIs are for application developers who might only have a moderate understanding of geographic information or how to channel this information to visually impaired end users. Thus, to succeed, the toolkit interfaces should follow the ideal functionality described by Henning (2007): the right call for a particular job is available at just the right time, can be found and memorized easily, is well documented, has an interface that is intuitive to use, and deals correctly with boundary conditions.

The toolkit aims to support several modalities and types of human-computer interaction in order to help the user perceive the spatial information. The applications could use visual feedback for visually impaired people or the elderly through common visualization rules. Tactile feedback for blind or otherwise visually overloaded users, such as bikers, can be based on vibrations. Vibrations may come from the in-built vibrators of smart phones. The toolkit core supports access to the vibrators. Vibrations may also come from external devices; for instance, Anastassova & Roselier (2010) defined preferred vibration patterns for a handheld haptic device called VIFLEX (Roselier & Hafez, 2006), which can be accessed through the toolkit. Similarly, vibrations may be given by a tactile waist belt (Erp. et al. 2005; Pielot & Boll, 2010) or wristbands (Bosman et al. 2003). For toolkit support, these actuator devices would need their own output drivers in the toolkit core.

Audio-based guidance can be based on spoken instructions, as has been presented by several systems, like Chittaro & Burigat (2005). The toolkit supports text-to-speech, but auditory feedback is not restricted to speech because non-verbal descriptions of the surroundings may also be given through other means, such as earcons (Blattner et al. 1989), spearcons (Walker et al., 2006), auditory icons (Gaver 1986) or soundscapes (Schafer...
1977). Similarly, directions and distances may be given with audio patterns or metaphors, such as the Geiger counter (Holland et al. 2002), Hearcons (Klante et al. 2004), gpsTunes (Strachan et al. 2005), Soundcrumbs (Magnusson et al. 2009) and AudioBubbles (McCookin 2009). These functionalities are not part of the toolkit, but similar and broader ones may be implemented in the future as part of the toolkit.

Several toolkit modules support the human-interaction modules. One of the support modules handles computational geometry tasks. The module is made available for internal and external use. The demand for such a module was incontrovertible; however, whether or not to implement the necessary algorithms, especially for the toolkit, or to use existing open source geometry engines was not self-evident. The solution was to implement the functions that had mostly been applied from the beginning under the licensing model of the toolkit, and to make the rest of the algorithms available through wrapping them inside functions that perform the necessary data and geometry type conversions. The fact that not all of the required functionalities were available in computational geometry engines had a substantial impact on the decision. This paper presents an assessment of the solution, and compares it with the option of only using wrapped functions.

2. The Architecture of the HaptiMap toolkit

2.1. Core, mantle, crust

The toolkit is composed of three principal modular layers: core, mantle and crust. All modules can access the public interfaces of the same layers as well as the lower layers. Furthermore, a set of plug-ins is used to read geospatial data into the internal data model of the toolkit. The core layer has two main functions. Its first task is to connect to external sensor hardware, such as an accelerometer, for input or a tactile vibrator belt for output. To make use of platform-specific operating system interfaces, some parts of the core are also platform specific. The second task of the toolkit core is handling and caching geographical data in a simplified data model. Geographical data is stored within a custom database based on memory-mapped disk files. Memory-mapped disk files reduce processing requirements by presenting disk files as virtual memory blocks (Scheiber 1997).

The mantle layer is situated on top of the core. This platform-independent layer acts as a middle layer between the core and the toolkit utilizing software components – the crust. The modules of the mantle layer are building blocks that involve analysis and processing functionalities. Consequently, the mantle layer modules contain the primary accessibility-
enhancing logic. For instance, a module might employ the text-to-speech functionality of the core together with access to the vibrations to signal the direction to a target location. The text-to-speech functionality would be provided using the Flite synthesis engine and the vibrations through the in-built vibrator of the mobile device.

The modules of the mantle are written in pure ANSI-C. Both the core and mantle layers are published under the Lesser General Public License (LGPL). The crust components may just as well be called 'convenience-modules', because they present alternative ways to a mobile application user for how to realize the functionality of the core and mantle. For instance, the crust can include view activities and fragments for the Android platform, while for the iPhone platform the crust can contain view controllers. A view controller could, for example, visualize how passable a route is for a physically restricted person, by providing the starting and end points, or a reference to the route. The crust functionality might also simply be used to modify the mantle behaviour, such as to provide settings for vibration patterns. The crust layer can contain modules in any suitable license. Similarly, the functionality in the crust is written in a platform-specific language.

2.2. The data model and internal coordinate reference system

The abstraction level of the toolkit was a compromise between the degree of functionality versus its level of simplicity. The solution was to keep the abstraction high, because most of the accessibility-enhancing software implementations do not need a complex geographical data model and it is anticipated that their developers will have only meagre experience with geographical data; thus, an overly complex model would be unpropitious. Consequently, the toolkit may be easy to learn, easy to use and hard to misuse; however, it does not allow for enhancing accessibility using a

![Figure 1. The ISO Simple feature access geometry model (left) and HaptiMap toolkit geometry model (right).](image-url)
comprehensive geometry model.

The geometry model of the toolkit is composed of two principal geometrical primitives: points and linestrings. In addition, a linestring may be used to form a planar surface, a polygon, without any interior holes. Formalising a linestring as a polygon is performed through the geographical data storage API. A comparison of this geometry model with a model of the ISO Simple feature access standard (ISO 2004) is presented in Figure 1. The information model of the toolkit allows the geometries to be linked with any number of attributes. The attributes are key-value encoded. The keys are integer values whose meaning is defined in resource files. The value of an attribute may be an integer, integer array or string.

We performed another simplification in order to increase efficiency on mobile devices. We made the decision to use centimetres as the internal unit of measure and store all co-ordinate values as 32-bit integer values. The whole numbers reduce round-off errors (i.e., they increase robustness) and slightly increase performance; however, during spatial operations, a conversion to floating point numbers is commonly required to avoid integer over- and underflows. Similarly, while reading data into the internal data model, we needed to convert the data into centimetres. As the unit of measure for angles, we decided to use degrees in public interfaces because most platform APIs use them for digital compass bearings.

The coordinate reference system of the toolkit can only be a compound or projected coordinate reference system. In both cases, the two-dimensional part is confined so that it has a metric-map projection; thus, for instance, Pseudo Plate Carrée would be unsuitable as a projection. The second part of the compound system is a gravity-related coordinate reference system; in other words, height values are normal or orthometric. The coordinate reference system is based on a leading data source; however, for most operations, it is irrelevant what the actual system is. It is enough to have coherent units and to know that even the height axis is almost perpendicular to the two-dimensional plane.

2.3. Geospatial data work-flow

The toolkit is built to handle only geographical vector data, because it takes far too much time to perform analysis on raster data on a mobile device while in a hurry. The data is loaded into the toolkit from external data sources using logically separate plug-ins for different types of data sources, such as the open and free Open Street Map API or NAVTEQ's commercial MapTP API. A plug-in is required to map the feature type and attributes into the toolkit information model. Additionally, the plug-in may have to generalize the input geometries to make them suitable for the toolkit, which
may involve operations such as removing polygon holes and splitting a multi-geometry into individual geometries. Separating the plug-ins outside the core and mantle allows them to be written under a more diverse software license than exists for the core and mantle. Besides, the plug-ins’ data may be added to the internal data storage through the public interface of the core layer. The application making use of the library may decide to only temporarily add data to the internal storage of the toolkit, while maintaining its own database. This scenario is especially suitable for applications where the toolkit library is included at a late stage of development, or as an optional extension. Alternatively, the application can, for example, let the toolkit store all required vector data while it stores raster data for base map visualization.

2.4. Geospatial operations

Human-computer interaction developers are, for the most part, not interested in the absolute positions of features in their applications; instead, relative measures between objects play an important role. For instance, pointing or scanning (or sweeping) types of applications, such as a geiger counter, hearcons or soundcrumbs, could be implemented by knowing only relative measures to the closest points. Hence, the distances and directions to nearby objects measured from the location of the user are of greatest importance for building the mental model of the surroundings of the user and guiding her or him to the final destination.

Operations such as distance and bearing are required to guide the user. For instance, an application could give, while using multiple modalities, the direction and distance to a visual or otherwise notable landmark. These operations, together with length and area, can also be used to describe the environment. To obtain the values, it is necessary to offer supporting operations for fundamental static computational geometry problems, like geometry intersection or Convex-Hull calculation. Similarly, support for search problems, spatial predicates and buffering is required to solve queries made by accessibility enhancing functionality. For instance, the point-line intersection function may be essential for determining the distance to the centre of a footpath, buffering may be applied to validate that the user is inside a passable safe zone, and ray-tracing can be used to verify how far it is to the next obstacle or attention point. The requirements for algorithms depend on the application at hand. At the moment, we can only implement our own applications and predict some of the problems that others might need to solve, because the library will evolve in the future and more human-computer interaction modules with new needs will emerge.
Table 1. Some operation interfaces of the HaptiMap toolkit.

```c
/* Metric methods */
HM_RESULT hm_geom_area(hm_t *hm, int lid, double *area);
HM_RESULT hm_geom_bearing(hm_t *hm, int pid1, int pid2, double *angle);
HM_RESULT hm_geom_distance(hm_t *hm, enum HM_GEOMETRY_TYPE gtype1, int fid1,
   enum HM_GEOMETRY_TYPE gtype2, int fid2, double *dist);
HM_RESULT hm_geom_distance_hausdorff(hm_t *hm, enum HM_GEOMETRY_TYPE gtype1,
   int fid1, enum HM_GEOMETRY_TYPE gtype2, int fid2, double *dist);
HM_RESULT hm_geom_length(hm_t *hm, int lid, double *length);

/* Spatial predicates */
HM_RESULT hm_geom_contains(hm_t *hm, int polyfid, enum HM_GEOMETRY_TYPE gtype, int fid, int *r);
HM_RESULT hm_geom_within(hm_t *hm, int fid, int polyfid, enum HM_GEOMETRY_TYPE gtype, int *r);
HM_RESULT hm_geom_intersects(hm_t *hm, enum HM_GEOMETRY_TYPE gtype1, int fid1, enum HM_GEOMETRY_TYPE gtype2, int fid2, int *r);

/* Overlay methods */
HM_RESULT hm_geom_intersection(hm_t *hm, enum HM_GEOMETRY_TYPE gtype1, int fid1, enum HM_GEOMETRY_TYPE gtype2, int fid2, enum HM_GEOMETRY_TYPE *r_type, int *r);

/* Buffering */
HM_RESULT hm_geom_buffer(hm_t *hm, int fid, enum HM_GEOMETRY_TYPE gtype, double buffer_width, int *r);

/* Generalisation etc */
HM_RESULT hm_geom_simplify(hm_t *hm, int lid, double tolerance, int *r);
HM_RESULT hm_geom_centroid(hm_t *hm, int lid, int *r);
HM_RESULT hm_geom_interior_point(hm_t *hm, int lid, int *r);
HM_RESULT hm_geom_convex_hull(hm_t *hm, int lid, int *r);
HM_RESULT hm_geom_mbr(hm_t *hm, int lid, int *mbr_id);
HM_RESULT hm_geom_ray_intersection(hm_t *hm, int lid, int pid, double angle, double *distance);
```

A single module in the mantle layer primarily performs geometry-related computations. The module is made available for the human-computer interaction modules in the mantle layer and all modules in the crust layer. The input geometries are directly read from the public core interface, and the calculations take into account the characteristics of the toolkit. The algorithms support common spatial operations for two-dimensional geometries. The implemented algorithms are not all robust. Some algorithms are affected by round-off errors, but typically the targeted types of human-computer interaction applications do not require the robustness. For instance, an application wants to inform the user how far the next obstacle. To calculate the intersection point a ray is cast from the user’s location to a certain direction. If the obstacle is a point or parallel line then due to floating point arithmetic the ray might pass the geometry even if it should intersect in reality.

The interfaces support only centimetres for distances and degrees for angles for coherency. Table 1 presents interfaces for some of the operations. All operations return a status code indicating whether they were successful or not. If the geometry type is not implicit in the operation, the geometries are given as references together with the type of geometry; for instance, length is not computed for a point feature. Possible output geometries are directly
written to the internal data storage and references to the identifiers of the new geometries are returned.

3. **Assessment of the Operational Performance of Geometry Handling**

3.1. **Fit for use and ready for testing**

We conducted unit tests for every implemented spatial function. The test cases are comprehensive, independent and can be run continuously while extending the interfaces or upgrading the functionality behind the interfaces. The unit tests also provide examples for HCI developers on how to use the APIs, that is to say, the test cases also act as documentation. We validated the unit testing values by comparing them to other geometry libraries; thereby, during performance comparisons, it was no longer necessary to assess the correctness of the library by making separate result comparisons.

The functions for testing were not randomly selected. We only tested and validated functions that were based on the same algorithms in GEOS and the geometry library. We performed the validation by going through the open source code of GEOS. For instance, both the library and GEOS implement convex hull calculations using Graham Scan (Graham 1972) with slightly different syntaxes.

3.2. **Operational processing power comparison**

We tested the spatial operations of the toolkit against one of the oldest, most widespread and stable open source geometry engines available, the GEOS (Geometry Engine – Open Source), which is a part of the Java Topology Suite (JTS). GEOS was originally written C++, but has also been wrapped in the C-language. We also designed the geometry engine to be wrapped for the most complex functions required by the toolkit. This work has not yet been finished.

We compared the operation performances with the corresponding operations of GEOS. The operations included the most common functions, such as distance calculation, but also more sophisticated algorithms, such as convex hull and intersection calculation. We performed the benchmarking on an iPhone mobile phone simulator, iPhone 4 device and iPad 2 tablet. The simulator was ran on a 2,5 GHz Intel Core 2 Duo, 4 GB RAM. We ran each operation several times (100-1000) in order to obtain realistic benchmark numbers and remove noise. In addition, each benchmark was repeated one hundred times with different random
geometries. We recorded standard deviations to validate how repeatable and systematic the results are. Three different cases with same data were benchmarked:

1. The implemented algorithm. Before benchmarking, all data is stored in the internal data storage of the toolkit. The algorithm is responsible for reading the geometries from the data storage based on the identifiers of the features.

2. Corresponding GEOS algorithm. The same geometries as in the first case are converted into the geometry model and structure of GEOS before benchmarking. GEOS is accessed through its C-interface.

3. GEOS algorithm wrapped inside the toolkit model compliant interface. The internal geometry model is converted into the GEOS geometry model and structure during benchmarking. Similarly, if the operation creates new geometries, these are added to the internal data storage during benchmarking.

3.3. Adjusting the size of the internal data storage

After the first benchmark iterations it became evident that something was wrong with the algorithms that created new geometries. The problem appeared as an unordinary high standard deviation and sluggish performance, which declined after each run. Figure 2 shows some of the results of the benchmarking. The algorithms affected in the figure are polygon centroid calculations and convex hull calculations, which both create new geometries as a result.
The reason turned out to be the large number of times the memory-mapped files (on which the toolkit internal data storage is based) were resized during the tests. This had a severe impact on the performance, and was caused by the insertion of a large number of new geometry elements. The increments by which the mapped file sizes were increased were not designed for the benchmarking, instead, for normally sized input data (a couple of hundred features at a time). Thus, to get more realistic results, we increased the reallocation limit. The observation suggests that the size increment might need to be a variable that can be adapted for different types of applications. For instance, if the application relies heavily on the internal data storage and an intense amount of features are added to the data storage, then the size increment should be large. In general, there is much scope for optimizing the manner in which geometry elements are inserted and deleted from the memory-mapped files in order to improve performance.

4. Results

The results clearly show that implementing the algorithms from the beginning using the same formulas led to, in general, better performances in comparison to GEOS. Nevertheless, in cases with a high degree of variation, like the length calculation, the performance was poorer. Error! Reference source not found. presents the results of one benchmarking on the simulator. The same functions are shown in Figure 2 before the memory allocation fix. A comparison between the benchmark numbers of the figures gives away the impact of the memory size increment. Figure 4 presents the results on the iPad tablet. The
benchmark results were slightly better in comparison to iPhone 4, which is reasonable considering the processor differences. In general the results on the simulator were five times better than the results from devices.

Our most significant finding was the difference between toolkit functions and functions wrapping GEOS functionality. The result was independent of the media. In each benchmark, the difference was at least twice as high for the wrapped functions, but it could be up to ten times higher. The simpler the algorithm and the larger the percentage of time taken for geometry model conversions, the higher the difference became. The difference between the time taken to perform the GEOS algorithms versus the time for performing wrapped functions reveals the time needed for the conversions.

We obtained a third result by comparing the time needed to implement the algorithms. It took much more time to implement complex algorithms because of the required data structures and the number of boundary conditions that needed to be taken into account. However, the time difference between running these algorithms diminished.

### Figure 4. Benchmarking results on the iPad 2 tablet.

<table>
<thead>
<tr>
<th></th>
<th>HaptiMap toolkit</th>
<th>GEOS geometry...</th>
<th>HaptiMap toolkit +...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>mean 0.2799</td>
<td>mean 0.1156</td>
<td>mean 1.3575</td>
</tr>
<tr>
<td></td>
<td>stddev 0.0684</td>
<td>stddev 0.0075</td>
<td>stddev 0.0174</td>
</tr>
<tr>
<td>Centroid (of polygon)</td>
<td>mean 0.0389</td>
<td>mean 0.0060</td>
<td>mean 0.2069</td>
</tr>
<tr>
<td></td>
<td>stddev 0.0064</td>
<td>stddev 0.0175</td>
<td>stddev 0.0031</td>
</tr>
<tr>
<td>Area</td>
<td>mean 0.0337</td>
<td>mean 0.0178</td>
<td>mean 0.1551</td>
</tr>
<tr>
<td></td>
<td>stddev 0.0060</td>
<td>stddev 0.0052</td>
<td>stddev 0.0028</td>
</tr>
<tr>
<td>Distance (point - point)</td>
<td>mean 0.0031</td>
<td>mean 0.0315</td>
<td>mean 0.0659</td>
</tr>
<tr>
<td></td>
<td>stddev 0.0007</td>
<td>stddev 0.0026</td>
<td>stddev 0.0025</td>
</tr>
<tr>
<td>Distance (point-line/...</td>
<td>mean 0.0119</td>
<td>mean 0.0954</td>
<td>mean 0.1347</td>
</tr>
<tr>
<td></td>
<td>stddev 0.0019</td>
<td>stddev 0.0069</td>
<td>stddev 0.0099</td>
</tr>
<tr>
<td>Convex hull (of a linestring)</td>
<td>mean 0.2113</td>
<td>mean 0.4453</td>
<td>mean 0.5973</td>
</tr>
<tr>
<td></td>
<td>stddev 0.0723</td>
<td>stddev 0.0250</td>
<td>stddev 0.0255</td>
</tr>
</tbody>
</table>

5. Conclusion

In this paper we have provided an overview of the geospatial data handling of the open-source multi-platform HaptiMap toolkit. The toolkit enables mobile map-based applications to enhance accessibility. The toolkit is composed of three logically separate layers and a set of plug-ins for external data sources. One of the layers contains a module to solve computational geometry problems.

In addition we have performed benchmarking to gain confidence in our solutions. In this paper we have presented the results of the benchmarking, which proves our approach to be significantly faster as the alternative of using wrapped functions. Moreover, through the approach it is possible to
take into account special requirements for geometry computations. However, providing own algorithms takes a greater toll in form of work the more sophisticated the algorithms become, and at the same time the performance addition decreases.

Acknowledgments This survey was performed as part of the HaptiMap-project (www.haptimap.org) that is coordinated by the Rehabilitation Engineering Research Group of Lund University’s Department of Design Sciences. The authors are grateful to the European Commission, which co-funds the HaptiMap-project (FP7-ICT-224675).

References


Bosman S, Groenendaal


