University of Cape Town
Deparment of Computer Science

Honours Project Report

Real-World Spatial Features in Location-Based Mobile Games

Author:
Rainer Dreyer

Supervisor:
Dr. James Gain

	Category	Chosen	Total
1	Requirement Analysis and Design	15	
2	Theoretical Analysis	0	
3	Experiment Design and Execution	0	
4	System Development and Implementation	15	
5	Results, Findings and Conclusion	10	
6	Aim Formulation and Background Work	15	
7	Quality of Report Writing and Presentation	10	
8	Adherence to Project Proposal and Quality of Deliverables	10	
9	Overall General Project Evaluation	5	
Total Marks		80	

October 31, 2011

Abstract

This thesis argues for the need to build interactions between players and complex spatial features in Location-Based Mobile Games (LBMGs). Existing LBMGs such as Foursquare and MyTown leverage only point-based data, such as player coordinates and real-world establishments' positions.

A platform providing access to complex spatial features, such as building outlines, roads and reservoirs, is designed and implemented using only open source software, such as Hibernate and PostgreSQL. The thesis also discusses problems and solutions related to Geodesy and map projections.

Finally a rudimentary proof-of-concept game is built and tested using this platform and complex spatial data on the University of Cape Town's campus.

Contents

List of Figures

List of Tables

Listings

Glossary

3G Third generation mobile telecommunications standards, often used to describe high-speed mobile Internet / data access.

A-GPS See Assisted GPS.

Assisted GPS A GPS receiver assisted by a computationally powerful server or additional downloadable orbital data, usually accessed through a mobile phone's data connection.

DBMS Database Management System, database software and tools to manage databases.

DOM Parser A Document-Object Model Parser constructs an in-memory tree representation of an XML document.

EDGE A 2nd generation mobile telecommunications standard, characterized by much slower data transfer rates than 3G.

EPSG The European Petroleum Survey Group compiled and disseminated the EPSG Geodetic Parameter Set, a widely used database of Earth ellipsoids, geodetic datums, geographic and projected coordinate systems, units of measurement, etc.

FOSS Free and open source.

Geodesy The branch of mathematics dealing with the shape and area of the earth or large portions of it.

GeoTools An open source GIS toolkit written in Java.

GIS Geographic Information Systems, a system for storing and manipulating geographical information on computer.

GPS Global Positioning System, may also refer to the GPS receiver, which receives GPS signals for the purpose of determining the device's current location.

GPS receiver A stand-alone device or chip in Mobile Phones that provides the user or device with GPS position updates.

Hibernate An open source Java persistence framework.

Hibernate Spatial Spatial extensions for Hibernate, which provide the developer with Spatial Queries and results returned as JTS Geometry.

JDBC Java Database Connectivity, a set of Java classes and interfaces for client-side connections to databases.

JTS Java Topology Suite, a collection of Java geometry classes implementing the SFSQL standard.

JOSM An OSM Editor, written in Java. It allows OSM contributors to create and modify spatial features in their vector form.

JVM Java Virtual Machine, the Java language's runtime environment.

LBMG See Location-Based Mobile Game.

Location-Based Mobile Game A mobile game which uses players' positions in the real world as a gameplay element.

Mapnik A python / C++ GIS toolkit, offering map rendering functionality. Used as the default renderer on openstreetmap.org and by many community projects.

Object/Relational Mapper A software library used to transform relational database records into native programming language objects.

OGC See Open Geospatial Consortium.

Open Geospatial Consortium An international voluntary consensus standards organization for GIS and spatial data, originated in 1994.

OpenStreetMap A free & open source, wiki-style world map.

ORM See Object/Relational Mapper.

OSM See OpenStreetMap.

Osmosis A Java toolkit to store, manipulate and transform OSM data from and into different sources and formats.

OSM XML OSM data formatted in XML as provided by OSM APIs.

POI Point of Interest, a real world location represented by a point or coordinate. POIs are usually of touristic relevance or amenities, such as restaurants, shops and police stations.

POJO A plain-old Java object.

QGIS Quantum Gis is a GIS toolkit, useful to display and modify data from a large variety of spatial data sources.

SAX Parsers Simple API for XML Parsers allow the sequential processing of XML pieces using an event-based system of callback methods.

SFSQL Open Geospatial Consortium Simple Features for SQL, a standard defined by the OGC, specifying a set of geometry classes and operations these should support.

SQL Structured Query Language, a language used to query relational databases. Most relational databases define their own dialect of SQL.

Tile server A server specialized to serve map tile "images".

WGS-84 World Geodetic System, 1984 revision, is the coordinate system used by GPS devices and defines a standard coordinate frame for the earth, a standard spheroidal reference surface for raw altitude data, and a gravitational equipotential surface that defines the nominal sea level.

WKB The binary version of WKT.

WKT Well Known Text, part of the SFSQL standard, describes spatial features in standard text. This format is meant to allow spatial objects to be shared and instantiated in different SFSQL implementations.

XML Extensible Markup Language is a language to describe structured data in a machine-readable and human-readable format.

Chapter 1

Introduction

Location-Based Mobile Games (LBMGs) are entertainment applications that run on mobile phones and use spatial data to create gameplay elements. They form a sub-genre of pervasive games, which either "augment traditional, real-world games with computing functionality or bring purely virtual computer entertainment back to the real world."[18] Many of these games are also commonly classified as Augmented Reality (AR) applications[32], but a stricter definiton of AR requires information to be super-imposed on live video from a device's camera, which most LBMGs do not provide.

Since 2000 there has been a sudden explosion in research covering location-based games [4], all of which opted to use spatial data in differing ways. Some of these games were designed for solo-play; others were designed for "social-play" and some to scale to a "massively multiplayer" user base. Some games track mobile users and others are based on points of interest. The technology used to establish locations also varies from using GPS devices and GSM triangulation[1] to short range radio frequency proximity sensors. Some game concepts are defined by the spatial data they rely on, and some games create new spatial data.

With the proliferation of GPS-enabled smart phones in the mobile consumer market and mobile games fuelling the app economy, LBMGs seem like a logical successor to the wave of touch and accelerometer interaction-based games that have done well on mobile app stores. Indeed the success of games like MyTown by Booyah, a location-based social game with over 3 million users [31], and the success of location-based social networks, such as Fousquare and Gowalla, seems to suggest that LBMGs have the potential to be "game changing" mobile applications.

Because of the large variety of games developed, many varying solutions for creating (or importing), storing and transmitting spatial data have been created for each specialized gaming application, but none of these solutions have been described in detail in publicly available papers or open-sourced.

This project's main goal is to build an open-source platform for developers of LBMGs. For this to be successful, the resulting platform needs to be flexible enough to be of use to developers of many different types of LBMG. The different types of LBMG are described in detail in section 2.3 - Taxonomy of LBMGs. In

[1] The technique of using mobile network base stations to estimate a phones position.

Figure 1.1: Sample OpenStreetMap vector data, centered on UCT

addition, the project explores the platform's ability to facilitate gaming using higher complexity real world spatial features, since no existing LBMGs leverage data other than simple coordinates.

1.1 Complex spatial data

As further discussed in section 2.3, no current LBMG uses spatial data other than points. Players are represented by moving points and shops or other places of interest (POI) are represented as static points. This thesis treats this kind of point-based data as "simple data".

"Complex data" is data based on all non-point spatial features. These features can include lines and polygons. Streets are commonly represented as lines and buildings are a classic example of polygons.

Figure 1.1 shows the full range of spatial data available from some sources, such as OpenStreetMap. Points include bus stops, individual trees, automatic teller machines, etc. The more complex data includes fields, parking lots, reservoirs, forests, etc.

1.2 Research question

> Using more accurate positioning hardware now available on mobile phones, such as GPS and Assisted GPS (A-GPS), we can build more fine-grained location-based interactions with game worlds.

> While the distance players are from one another in previous mobile games could only be estimated by the distance they were from the cell towers they were connected to, we can now define interactions based on much more accurate metrics. More importantly we can now place players on roads and estimate their distance from points of interests.

The previous paragraph describes the current state of the art in LBMGs: GPS or A-GPS data is used to let players interact with each other (represented as coordinates internally) or interact with POIs, such as restaurants and other Check-In locations (which are also represented as coordinates in the game engine). These games using only coordinate or point data can be said to use "simple" real-world spatial data.

We propose to push the state of the art to LBMGs offering interactions with complex real-world spatial data, which can include streets and areas, such as fields, parking lots, tunnels, bridges, etc. and we are hoping to offer this ability to other developers as an easy-to-use and powerful platform.

The main research question investigated in this document is whether it is possible to use complex real-world spatial data in LBMGs and whether a flexible and easy to use platform can be built that allows game developers to leverage said data in their own games.

This question can be further broken down into the following sub-questions:

1. Can we find a source of spatial data that is accurate and dense enough to be used in a demonstational game?

   We need the spatial data to be correct, up-to-date and not too sparse to be useful in a game, i.e. players should not have to walk too far to find spatial game objects to interact with.

2. Can we convert complex real-world features such as steps, parking lots and hollow buildings into game objects?

   After procuring spatial data, we need to parse or convert it to objects in our framework. How can this be done efficiently?

3. Can we serve this data to game clients within an acceptable amount of time and handle a large number of clients?

   While this paper is not directly concerned with networking between the game server and its clients, the component described here needs to perform spatial queries rapidly and efficiently enough to serve players of our demonstrational game. If queries by this component are too slow and the networking component's latency is too high, players might be able to move into an area that appears empty and get startled by suddenly appearing spatial objects.

4. Can we make this component easy enough to use to be useful to game developers not experienced with handling spatial data?

   Most game developers can easily define interactions between objects and players in 2D and 3D space, often using abstractions such as tiles, rooms or waypoints. The system we need to build should not expect these game developers to understand much about cartograhy, reference coordinate systems (WGS-84) or map projections.

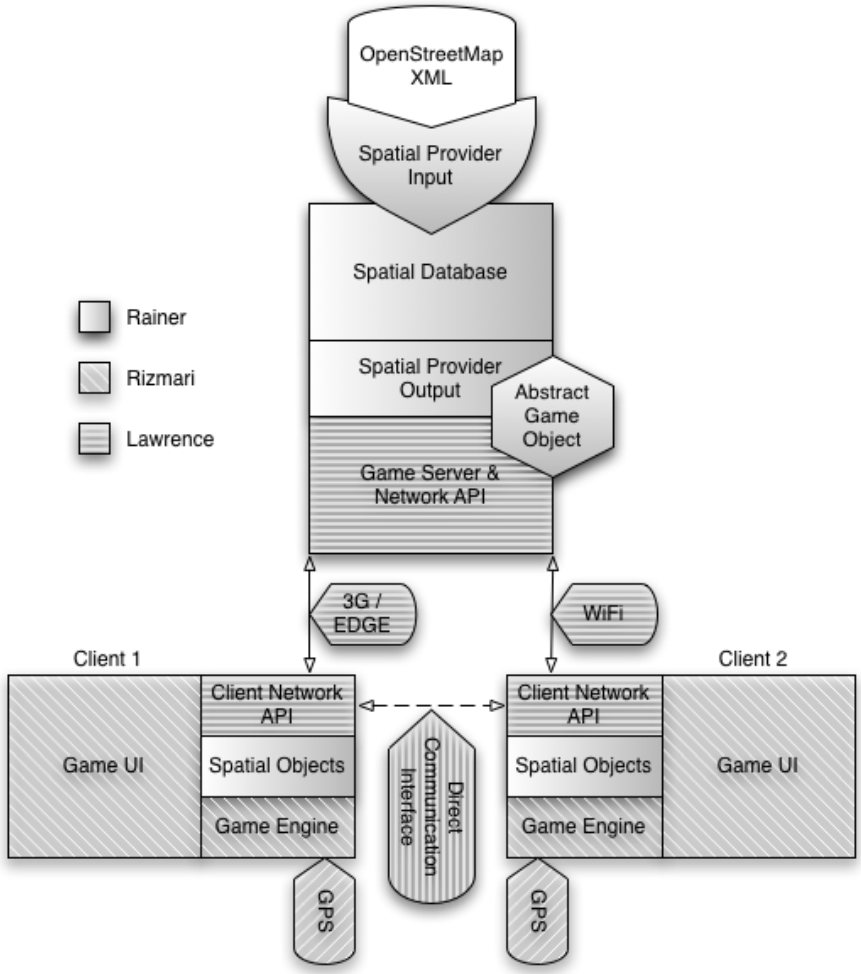


Figure 1.2: System diagram

1.3 System overview

The system used to test our project's research questions can be best summarized by looking at Figure 1.2. The system is divided into two major components: the server and the client.

The server is split between the Spatial Provider component and the game server, which incorporates the Mobile Networking component. The clients are based on a game engine, which uses the Mobile Networking Client component and Spatial Objects and is extended by a game-specific UI.

This paper describes the Spatial Provider component and the Spatial Objects passed between the components. The Spatial Provider is further subdivided into three logically separate components:

1. A parsing & processing layer, which reads and stores data from a spatial data source, such as OpenStreetMap.

2. A spatial database, which is used to permanently persist spatial objects and has the ability to perform efficient queries on said data.

3. An interface, which abstracts away the underlying database queries and provides easy to use spatial game objects to the Game Server component. These objects should be able to be passed to the game clients and provide utility functions, such as distance calculations and intersection tests.

1.4 Thesis outline

This chapter provides a brief overview of the Spatial Provider component's goals.

Chapter 2 first gives some background on Geographic Information Systems (GIS), then surveys the field of Location-Based Mobile Games from a historical perspective and finally builds a taxonomy of LBMGs, classifying them by their use of spatial data.

Chapter 3 covers the design of the Spatial Provider from an early stage Analysis Model in Figure 3.1 to a more detailed Design Model in Figure 3.6. It discusses the technology choices made to complete the Spatial Provider and justifies the component's feature requirements.

Chapter 4 outlines the Implementation phase of this project. It covers many of the pitfalls encountered while transforming the Design Model into a working component. It also covers issues encountered with geodetic conversions and map projections.

Chapter 5 demonstrates this project's results. It analyses the component's performance and comments on the implementation of the features required by this component. It briefly discusses the results from user testing the LBMG Lokémon.

Chapter 6 is a short conclusion and proposes some future work to be completed on this component and in this field in general.

1.5 Intellectual property

Since this component is meant to be usable in commercial games, the attached software license should be as permissive as possible. While the component could have been placed in the public domain, attaching the BSD license removes this team from all future liability and removes any intellectual property rights UCT could hold over this software.

Chapter 2

Background

2.1 LBS, GIS and Spatial DBMS

This section will explain the jargon surrounding Location-Based Services (LBS), Geographic Information Systems (GIS) and spatial Database Management Systems (spatial DBMS) by building understanding of spatial concepts from the ground up starting with the data and its varying representations, and finally covering the services built on top of it.

Spatial data can be represented using two data models: raster data and vector data [15]. Most users are more familiar with the former, since the final result of most spatial queries is represented as raster data i.e. an image. The quality of this representation is limited by the resolution of the resultant image, since all spatial data is encoded in the position and colour of individual pixels. The position of a pixel can be used to establish distances to other points in space and the colour of a pixel denotes the type of the real world object, i.e. a blue pixel typically corresponds to a water feature. This data is readily available for most applications, and sources of it include Google Maps and Yahoo Maps.

Vector data is a more flexible representation of spatial data. It consists of annotated geometric shapes, which are usually built up from simpler shapes, like polylines, lines and points[19]. The simplest shape (i.e. a point) usually contains spatial information like latitude and longitude. The highest level shapes usually represent geometric features such as roads, buildings and railway lines and are distinguished from one another through the data with which they are annotated. This data is often used to generate (or render) rasterised representations to be displayed to a user. Vector data is created through GPS traces of roads and paths and through the manual identification of geographic features on satellite imagery[11]. The OpenStreetMap project is a typical source for free to use geographical vector data [34].

Raster data is harder to manipulate than vector data. This is because programmatically changing an annotation or a points position in raw vector data is easier than changing text in an image or changing the colour of a feature, which could be disjoint or intersect other features of the same colour [13]. Thus transformations from vector data to raster data tend to be irreversible for complex data. For these reasons vector data is usually the initial and intermediate representation of all spatial data and only the final product gets rasterized or

rendered[15].

Spatial data needs to be easy to store and query to be useful to higher order information systems. Raster data can usually be handled by static file servers, whereas more complicated solutions are needed for vector data. Most solutions incorporate spatial data with their existing relational database models. For this purpose many Database Management Systems (DBMS) added location fields and spatial queries like distance comparisons in some extensions, making them spatial DBMS or geoDBMS[9]. Example extensionsinclude PostGIS (for PostgreSQL), MySQL Spatial, GeoCouch (for CouchDB) and Oracle8i Spatial.

Some research has recognised that traditional DBMS have not been designed to efficiently update and query non-static (i.e. moving) spatial data [28]. Many of the data structures used in spatial DBMS are optimized for fast queries of static data[21] and assume few insertions, updates and deletions of indexed spatial data. This has motivated research on designing new data structures and algorithms to index moving spatial data. One resulting solution might be to build Spatio-Temporal DBMS extensions, which include a projected path or trajectory and a timestamp as part of a moving points spatial data[1, 7]. Using this new information, moving objects need only be updated in the DBMS once their trajectories change, but querying moving objects is a more computationally expensive operation [1]. Trajectories might be easy to predict for moving points such as cars and planes, and much harder for pedestrians (or general mobile phone users).

The information systems built on top of spatial DBMS often abstract away much of the underlying data model and present only high level geographic features to their users. These systems are grouped under the term Geographic Information Systems (GIS). GIS are used by map servers, in the geographical sciences and in Location Based Services[15].

Location Based Services (LBS) are usually mobile phone based services that make use of users' locations to offer context-specific services, information or recommendations. These systems usually rely on GIS to provide nearby points-of-interest (POI). defines them as any service or application that extends spatial information processing, or GIS capabilities, to end users via the Internet and/or wireless network.

Location-Based Mobile Games can be classified as specialised LBS, since they tend to leverage GIS and mobile networking to offer an entertainment service to users [17].

2.2 History of LBMGs and related games

2.2.1 Early LBMGs

Sotamaa [29] describes three of the earliest location-based games. We choose these as benchmarks, since almost all recent Location-Based Mobile Games (LBMG) still use the same spatial data sources and spatial DBMSs for persistence.

Sotamaa claims that the history of technology-assisted location- based gaming started with the invention of geocaching. This game consists of players with GPS devices hiding caches containing a few small items outdoors, and informing all other players of the caches positions. Some players then go treasure-

hunting for these caches and either record their finds in the caches or replace items. The simple geospatial data that this game requires - GPS coordinates - was initially shared between players via email and later recorded on a website [26]. Now multiple websites are used to visualize this data in the form of GPS points on maps. Since this data is not time critical and players do not need to be connected to any form of network during play, the technology developed to share this data is no different from the technology used in online map services.

Next generation early mixed-reality location-based games like Pirates! used relative positioning and proximity sensors to create interactions between players and their (indoor) environments [8]. Because all interaction was governed by the concept of proximity, no spatial data had to be saved in any form of persistence layer in the game servers architecture during gameplay. This particular system uses short range radio frequency proximity sensors placed inside a building and on the players to simulate islands and pirate ships. This allowes the game administrators to start a game in any location without setting up geospatial data first [8]. This second wave of location-based gaming again did not use specialized Geographic Information Systems (GIS) to manage the positions of players or game objects.

The third wave of location-based gaming according to Sotamaa uses mobile GSM networks to triangulate players. The earliest paper on the concept describes Botfighters, a mobile phone game based on SMS messages and CELL ID. In this game a player can locate other players and move towards them until both players are in the same GSM cell, at which point they can engage in battle (using SMSs) [24]. The game uses a website, but only to upgrade the players bots. The technology used to enable this GSM based gaming has been derived from more general Location Based Services (LBS). The three games described by Sotamaa [29], can all be classified as LBMGs under De Souza e Silva and Hjorth [4]'s classification scheme, which distinguishes between Urban Games, LBMGs and Hybrid Reality Games (HRGs). Pirates! could be viewed as an exception, because it uses proximity sensors as well as mobile computers instead of mobile phones, which might disqualify it from being a pure LBMG.

Hybrid Reality Games have higher spatial data requirements than most LBMGs, since they require a virtual representation of the real world. Some of these games offer standard 2D maps, but others choose to generate 3D representations of the real world for the virtual players. These games have resulted in a lot of research focusing on 3D Geographic Information Systems and how to generate this 3D information from various data sources [35, 10, 23]. Although players real world positions are updated on the virtual players maps (and sometimes vice-versa), much of the spatial data is static and map features do not influence gameplay.

2.2.2 Modern LBMGs

Most modern location-based mobile games make use of GPS chips built into smartphones or WiFi triangulation (geolocation) to improve the accuracy of the earlier CELL ID based games[4], but still use the same underlying spatial Database Management Systems (spatial DBMS or geo-DBMS).

One modern reincarnation of an older game is Gowalla, which is a social network mixed with geocaching[6]. Players still use GPS to locate something, but in this case the real-world cache or stash is turned into a virtual badge or

stamp. Instead of recording a find on the website or in a log- book in the cache, users can show off their stamps in Gowallas social network. The GPS is integrated into the mobile phone, and communication with the game server / social network is instant via Edge or 3G connections. The spatial data requirements are still very similar to the requirements of a game of geocaching a list of GPS coordinates.

Multiple location-based mobile games using the same type of static data (with no moving game objects) have been created[16]. Foursquare is a check-in based game and social network with the motivation stemming from being the mayor of a real world location[3]. Players can become mayors by checking into a location more than any other player[6]. MyTown by Booyah is another such check-in based application[3, 30], but extends the idea with a Monopoly-like virtual currency used to buy real-world properties virtual representations. In this game, players pay rent when visiting a location they do not own.

Both these social networks heavily rely on users creating the locations they like to visit (creating the spatial data) and curating the virtual representation of their city by removing duplicates, adding data about places, etc.

Some games offer a platform to extend the basic check-in based social gameplay. Foursquare offers an API to use their check-ins and locations in other games. Games based on this API, like Kingpin, foursqWAR and oust.me attach some form of strategic value to Foursquares check-in locations and allow players to capture or defend these. The games leverage players positions, but only when said players are checked-in somewhere they become vulnerable or can interact with the location. The spatial data is still point-based and can be stored in spatial DBMS.

Other modern LBMG allow players to fight over territory, but the underlying map data is only used as a visual reference and does not influence gameplay. Examples of this type of game are Parallel Kingdom[14] and Own This World. These games use simple partitioning schemes to split up the real world into chunks a player can take over. Chunks in the case of Own This World are simple grid squares. In Parallel Kingdom chunks are created as circular areas around flags. If these areas overlap a straight line runs through the overlap to crop the chunks. These lines give the appearance of complicated polygon-based spatial partitioning, but the real partitioning is simply based on areas around points.

Some games recreate the idea of location-based combat ala Botfighters. Mini Stockholm created a LBMG, called 'Getaway Stockholm 2010', as an advertising campaign: The game simulates a virtual Mini Cooper, which players can steal from each other, when they are within 50 metres of it. A player who manages to hold onto the virtual Mini for a week wins a real Mini Cooper. This game heavily relied on accurate real-time location data of all players and the player holding the virtual Mini, but does not make use of more complicated spatial data types than points which represent players.

Coke Zero released a new form of LBMG (Livecycle) to promote the movie Tron Legacy: When players join the game they get paired in a grid-based virtual environment overlaid on their real world. Because players get joined in a virtual grid, they do not have to be physically close to participate in the same game level. They create virtual paths similar to the Light Cycles in the movie by travelling across the city and if one player moves through another players path they 'get derezzed' (or die). The spatial data generated by this game forms

a new category, since it uses accurate absolute positioning to generate relative positions between players in a virtual space. Since the game server generates new virtual levels for each pair of players, past positions can be deleted after each round of play and the server does not need to query a GIS for player locations or other game objects.

One game deserves a special mention because it was based on real world locations, even though it is not a mobile game and not location-based in the sense of the players location being important. Hasbro launched a game called Monopoly City Streets (Monopoly CS) in September 2009, in which players could play Monopoly on Hasbros website, with the Monopoly streets based on real world street data extracted from OpenStreetMap (but overlaid on top of a google map). The game was discontinued due to scalability issues and other bugs in December 2009. The main difference between this game and MyTown is that locations in MyTown are venues generated by players, whereas Monopoly City Streets used OpenStreetMaps spatial data to generate game elements.

2.3 Taxonomy of Spatial Games

2.3.1 Introduction

The research performed suggests a new classification scheme for Location-Based Games. It was designed to accommodate not only Location-Based Mobile games, but all games leveraging spatial data to create gameplay elements.

2.3.2 Static points, untracked players

Many LBMGs use this spatial data model. Examples are Gowalla, MyTown and Foursquare (and all games built upon Foursquare's API). Players can interact with fixed locations (which are usually real-world public locations like shops, restaurants, parks, etc.) if they are close enough to them, but their positions are not tracked. In this category all spatial data can easily be stored and queried using normal GIS and spatial DBMS solutions[12].

This data model has the advantage of being easy to implement. Google Places, for example, offers an API to query for static locations of interest. It is trivial to extend gameplay using these and GIS to store any game-specific data. The main disadvantage of this data model is that it does not allow for much player- versus-player gameplay, since players are not tracked and games revolve around locations rather than player s.

2.3.3 User-generated points, untracked players

Geocaching is one example game that falls into this category. The main difference between Geocaching and static POI-based games lies in the player's ability to create points that other players interact with. These points are rather static, and do not move or get destroyed frequently, so the solution used for static points still applies.

Unless games in this category evolve to create or update locations rapidly, the advantages and disadvantages remain the same as in the previous category: this type of game is easily implemented using normal GIS, but might not be player-centric enough to support all gameplay.

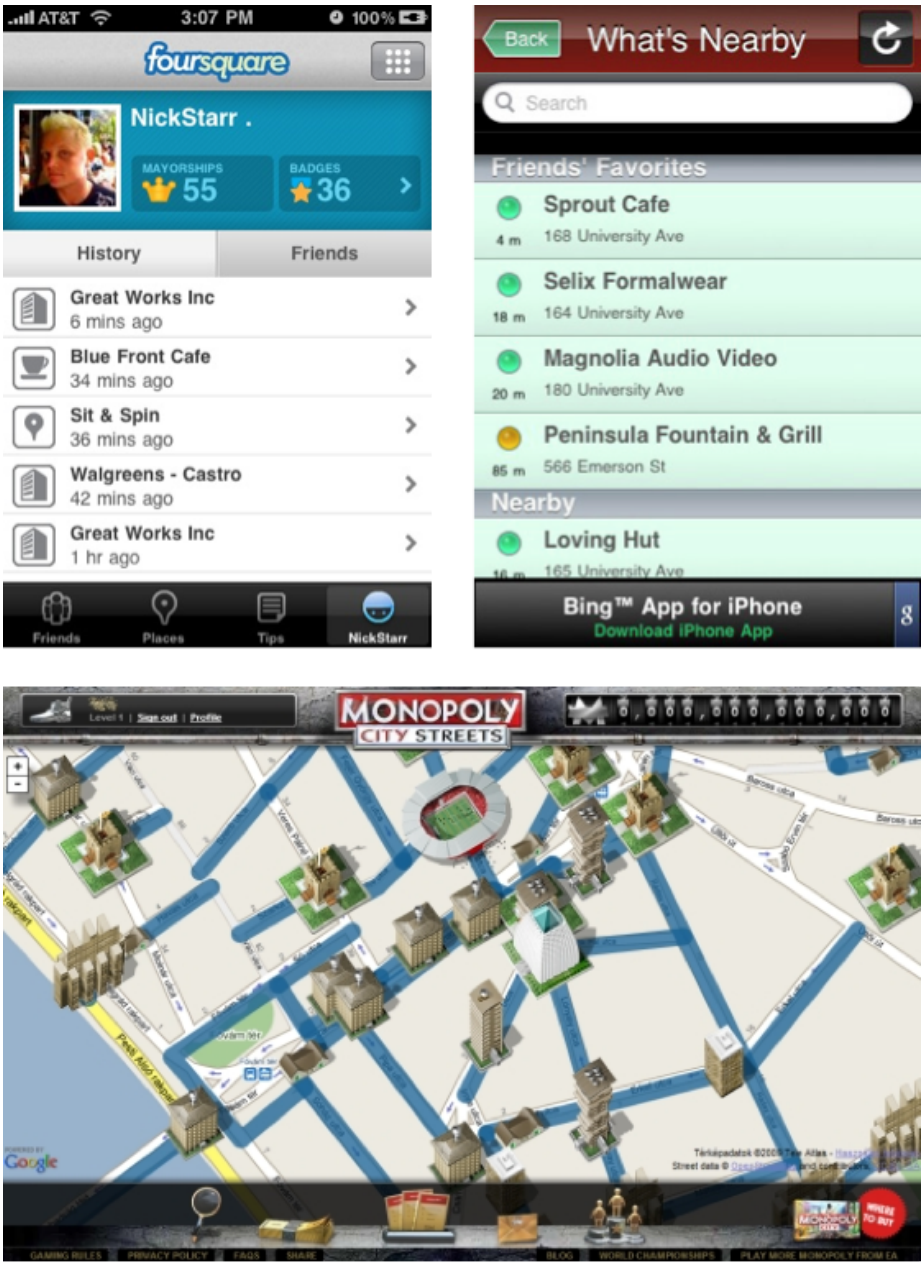


Figure 2.1: Foursquare, MyTown and Monopoly City Streets

2.3.4 Static features, untracked players

The only example in this category is Monopoly City Streets, which opted to use more complicated interactive geographic features than static points, but player locations were completely irrelevant to the game play. The game probably used the same GIS and spatial DBMS tools as games in the previous categories, but nothing has been published about the underlying technologies used.

This category could become quite popular, since it is simple to get the underlying spatial data from sources like OpenStreetMap. It also requires less expensive hardware on the client-side than other categories, since the players can interact with game objects independent of their actual positions. A disadvantage is the technical risk associated with extending this data model with tracked players, since it has not been attempted before.

2.3.5 Regular points, tracked players

Some territory-based games like Own This World fall into this category. The concept of controlling territory necessitates spatial data that is more complex than points, but in practice the division of the real world into grid cells is as simple as creating static points since connected rectangular cells can be described using just one corner point. Since the points are generated using regular spacing, this game can probably get away with a custom data structure that is much simpler than a spatial DBMS, but nothing about the underlying technologies has been published, yet.

Own This World generates troops, depending on which cell the player is in, so it is one of the first games to continuously track the player. The actual location of players does not have to be finely tracked, though, since they can simply be assigned to a cell.

The main advantage of this category is that it starts to embody the best of all worlds, by giving developers static spatial data and player locations to build game concepts with. The main constraint is that regularly spaced points do not facilitate more than simple territory-based game concepts.

2.3.6 Virtual space, tracked players

Livecycle is the first game to abstract away underlying geography and create a virtual world for the players to interact in. It probably does not use any form of geo-spatial persistence and is likely to track relative positions of pairs of competing players and past positions in-memory on the game server. No literature has been published on the technologies used in Livecyle.

This category has the advantage of removing the real-world in favour of a more flexible virtual world. This allows game designers to put players close to one another, even if they are not close in reality, for example. A disadvantage is that the player's connection to a real-world that can be bought or fought over is lost, and location-based games turn into motion-based games, which might be strenuous to play.

2.3.7 No data, tracked players

These are games that do not rely on any important locations or territory and are purely player-vs-player. Examples are Botfighters and Getaway Stockholm

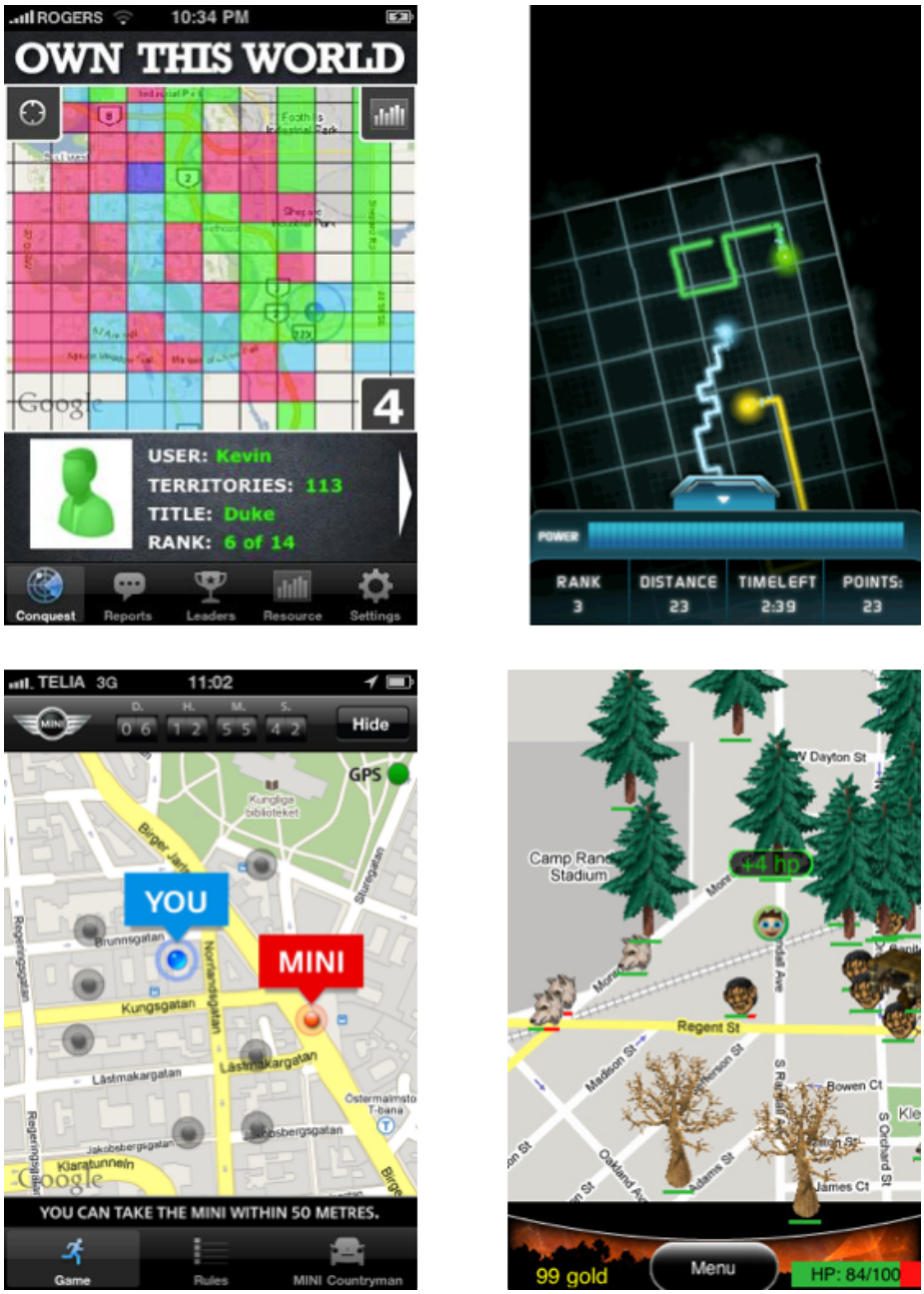


Figure 2.2: Own This World, LiveCycle, Getaway Stockholm 2010 and Parallel Kingdom

2010. These games need to track many moving points, and never store any static spatial data.

Botfighters used CELL ID technology and did not provide more granular location information; it probably relied on LBS technology from the 1990s. Future LBMGs are likely to require a solution that provides more accurate location data.

Getaway Stockholm 2010 probably used a custom software solution, since the game showed every players location to every other player and all positions were continuously updated. (A spatial database and advanced queries would have added no value.) With 11 000 participants (not online at the same time) a game server probably held these positions in memory, but currently no literature on the implementation exists.

The disadvantage of this model is that the real world turns into an obstacle, that inhibits player movement and does not constitute any value towards the gameplay otherwise. An advantage is that player positions are fairly simple spatial data, which is easy to manipulate.

2.3.8 User-generated points, tracked players

The only game in this category is Parallel Kingdoms (PK). Much of the generalisations of Geocaching still apply: Players generate points infrequently (erecting a structure is expensive); these points do not get destroyed frequently (attacking a structure is a lengthy and expensive process) and players can only erect or attack one structure at a time.

Players can see other players travel around the game / real world in real-time, using technology similar to No data, tracked players. The game limits players' travel ability to either their real-world position or within certain circular areas around structures, so a query for structures in the area can also return players.

2.3.9 Conclusion

None of the surveyed location-based mobile games made use of map data for anything other than as a background image for building a visual reference to find other players or virtual objects in the real world. Monopoly CS is different in that it was the first game to generate game objects (in this case tradable streets) from real world spatial data, but it did not use players positions in gameplay and was not playable on mobile devices.

This paper tries to justify the need to build a platform for LBMGs, that is flexible enough to facilitate the use of all types of spatial data, if the game designer so desires. If the platform allowed real-world map features to be turned into assets, ala Monopoly CS, this would enable novel gameplay in LBMGs. The proposed platform should also be able to continuously update the positions of many moving players with high accuracy.

Chapter 3

System Design

3.1 Introduction

3.2 Constraints

3.2.1 UCT's Campus

The demonstrational game we are using to evaluate our platform's performance and game's playability will be tested by volunteers on UCT's Upper Campus. We requested volunteers who identify themselves as casual gamers, but did not disclose that we were evaluating a location-based version of the game and a tap-to-move control version.

It is essential that the game - or the Spatial Provider component specifically - does not require the location-based version's users to walk too far to interact with specific game objects, since UCT's campus has a large number of steps, which "casual gamers" will find tiring. The Spatial Provider might also need careful balancing to provide a fair distribution of game objects of differing types around campus.

3.2.2 Data Networks

UCT's campus also brings the problem of bad cellular reception, since many buildings and lecture venues have been radio-proofed to eliminate students receiving calls during lectures. High buildings at the base of a mountain also impair mobile reception in general. The platform needs to handle clients having very patchy reception and a slow data connection, which means that the Spatial Provider component should not return overly large data responses.

WiFi network coverage on campus is also very patchy and the team decided to drop support for playing the game over WiFi during our user testing, since all WiFi networks on campus are secured by a non-permissive and non-transparent proxy with non-standard Windows ISA authentication.

3.2.3 Android

The project requires us to develop for the Android platform. While all members of this team have developed games for Apple's iOS, we welcome the challenge of

developing on a non-familiar platform. Android might qualify as a constraint, since we believe that the documentation, existing game frameworks and community support are better when developing for games for iOS.

3.3 Requirements

Since one of our main goals is to provide a reusable and general framework, a logical approach is to summarize requirements gathered from all previous LBMGs. To thoroughly analyze these past games, we split the "spatial data" and "player tracking" aspects of game design into two separate sections of requirements.

3.3.1 Spatial data requirements

These requirements include all aspects of receiving or querying and storing data the game server might need.

Static points, untracked players

From this LBMG type we can extract the following spatial data feature requests:

1. Request closest POIs, based on client position
2. Display rasterized data as a background for interactions

To serve "check-in"-based games, such as Foursquare and MyTown, all the client needs is a list of close establishments and shops and a background map to render them on. The client will determine its own GPS position to request this list and map, but the server will not track this exact position, since friends in social games are only interested in the last check-in location, not the user's actual GPS fix.

To implement this we do not necessarily need a spatial database, since we can request these POIs directly from POI APIs, such as Google Places and SimpleGeo Places, from the client device. Even OpenStreetMap's APIs can be restricted to return only specific types of nodes to emulate these pure-POI APIs.

Most of the games in this category use a rasterized "slippy map" to display these POIs and provide some contextual map information, but MyTown opts to provide only a list of close-by establishments with distance metrics to each. All other types of game types we explore, except "Virtual space, tracked players", also require feature 2, so we will not be explicitly repeating the requirement.

To provide this feature we can use map views (or activities, in Android jargon) provided by the Operating System, in the case of iOS, or the platform, in the case of Android, to request rasterized map data from Google or OpenStreetMap.

User-generated points, untracked players & User-generated points, tracked players

From these LBMG types we can extract the following spatial data feature requests:

3. Request POIs / Point-based Game Objects

4. Store Point-based Game Objects

Games, such as Geocaching, produce points and query for points. These games tend to produce points infrequently, and the points themselves are static and do not get destroyed often. These points are not necessarily close to the player: Geocaching allows the player to explore maps of caches before actually travelling to them.

Some APIs, such as OSM's API, allow us to create points and store them, but OSM's terms of service would not allow us to create game-specific points in the global map database. SimpleGeo offers a commercial service to do this, which is out of the question for our open-source (& free) platform.

At this point we need to definitely introduce a spatial database to store and efficiently query for points ourselves.

Static features, untracked players

From this LBMG type we can extract the following spatial data potential feature request:

5. Request and modify more complex spatial objects than points

This feature is actually one of the main goals of our platform, which is why we included a non-mobile game, such as Monopoly City Streets, in our LBMG history and classification. Monopoly City Streets is the only game we found that allows players to buy non-point-based spatial objects. Theses objects are annotated with the current in-game owner, the number of buildings built on them - which is dependant on the real world length of the street - and so forth.

We are hoping to introduce more advanced game object interaction than just using distance queries from POIs. Once the platform has the ability to accept non-POI data, we should be able to perform "Within" queries on arbitrary shapes and points, be able to determine the area of shapes, such as parking lots, sports fields or forests and potentially build complex interactions between real-world objects.

This feature requires us to use a non-point-based spatial database, but does not add anything else to the architecture yet.

Regular points, tracked players

From this LBMG type we can extract the following spatial data potential feature request:

6. Request and modify regular areas / points

Since games, such as Own This World, only build areas from regular grid-steps, we do not really need to add any special "area data type" to our database. Which "area" a point is in can be determined using standard division and modulo operators, and data about these regions can be saved in standard arrays for fast access.

Using the complex data-type database we mentioned in the last requirement, we do have the option of storing actual areas in a database already and can modify these with additional information, such as the presence of a player or their troops, so this can be modeled using a complex spatial database quite easily.

Virtual space, tracked players

From this type of LBMG we cannot extract any spatial data feature requests, since the space the game takes place in is created by the game and purely virtual, i.e. does not correspond to any real world map location for which map features would be available.

No data, tracked players

From this type of LBMG we cannot extract any spatial data feature requests either, since it uses only feature 2 to provide a background map for moving players's reference, but offers no interactive map-based features.

Conceptually this type of LBMG is the exact opposite of Static points, untracked players:

- While this game type displays only moving players on a map, Static points, untracked players displays only static POIs (and the player's position, which is not received from the server).
- All interaction is with other player objects, whereas in Static points, untracked players all interaction is with spatial data.

3.3.2 Player tracking requirements

Some games do not track players at all, the other game types usually want to share moving players' positions with other players in real-time.[1]

7. Track and broadcast players' real-time positions

This feature can not necessarily be implemented using a spatial database, therefore we would like to show that spatial databases are not required to implement this feature. We will discuss alternate designs used by two common player tracking games.

Case study: Getaway Stockholm 2010

Pure real-time massively multiplayer Player vs. Player (PvP) games, such as Getaway Stockholm 2010, rely on broadcasting player positions with the lowest possible latency. Getaway Stockholm 2010 displayed all online players to each other. This is a strong piece of evidence suggesting that the game server(s) held all players' positions in memory during the game, since querying a database for all players to perform a position broadcast to a new player is infeasible.

We believe that its games servers were possibly sharded by region, which is entirely feasible with 17 000 registered players. The reason we suspect that the dataset was sharded is not because keeping 17 000 players' positions in memory is challenging, but because of the "C10k problem"[2]. Keeping connections to more than a few thousand clients open from a single server is still challenging.

[1] We did not discover a game server that relied on exact player positions, but did not share them with other players immediately. Many of the POI-based games do establish exact player positions, but only to request spatial data around said position and do not store the position on the server.

[2] The C10k problem refers to the problem of optimising web server software to handle a large number of clients at the same time (hence the name C10k - ten thousand connections).

We are also assuming that only players close to each other saw each other's positions update in near-real-time.

Case study: LiveCycle by Coca Cola

In real-time PvP games with smaller numbers of competing players, such as Livecycle, keeping all players in memory rather than in a database is even easier. In LiveCycle the current and historic distance travelled and bearing (2D vectors) of players are the only important components to the gameplay and are streamed between mobile phones in near-real-time. The server does not store this information after a game is finished. Again this is evidence suggesting that the game servers form a "room" for each match and broadcasts position updates to all clients in said room directly, with no need for a spatial database or persistence.

Conclusion

We conclude that gameplay of this nature involving no spatial data, other than players - in limited size PvP scenarios or pure PvP scenarios - is best left up to the game server to implement. The game server might want to leverage the "point geometry classes" we intend to offer to achieve this, but most Java 2D point and vector classes should provide the features the server needs.

Large scale games providing a mix of geographic spatial data and player locations, might want to write a custom in-memory player position update component, too, while this Spatial Provider component provides the game server with geographic spatial data.

This custom component to manage players and perform distance checks can be a simple grid, a quad-tree, or any other datastructure that partitions players and removes the $O(n^2)$ comparisons required for naive distance checks between players. Most game developes should be able to effortlessly implement a datastructure they prefer in the game server.

3.3.3 Conclusion

The analysis of other LBMGs leaves us with the following seven requirements:

1. Request closest POIs, based on client position
2. Display rasterized data as a background for interactions
3. Request POIs / Point-based Game Objects
4. Store Point-based Game Objects
5. Request and modify more complex spatial objects than points
6. Request and modify regular areas / points
7. Track and broadcast players' real-time positions

We will now analyze, generalize and simplify this list:

Feature 1 is a less general version of feature 3: Feature 3 caters to POIs and more general point-based features and can return these around any position or

from any bounding-box rather than just around the player. A game developer can easily use feature 3 to provide the functionality requested by feature 1. Therefore we can safely eliminate feature 1.

Feature 2 can not be eliminated as easily, but developers can use the Android map "activity", which uses Google's map tile API, to provide this functionality. The disadvantage of this approach is that it requires a data connection to download individual map tiles at every zoom level. One alternative is to emulate GPS software, which often stores country or state-wide maps on the GPS device to remove this mobile data dependency. Cloudmade provides Garmin and Navit map files, which are generated from OSM data. Since this feature is mainly a client-side feature we do not count it as a feature of the server-side Spatial Data Provider, but we inted to support client-side developers with implementing this feature.

Feature 4 requires our component to offer write support on top of the read / query support requested by features 1, 2 & 3. This feature is offered by all spatial databases, but if the developer wanted to use a custom index to store these points, some of the same limitations of feature 7 apply, if points get created and removed (or move) rapidly.

Features 3 & 4 can be handled by feature 5, if feature 5 is implemented using a spatial database that can handle complex spatial data *and* simple points.

Feature 6 can be implemented using either the spatial database required by feature 5 or by a simpler spatial databases, such as the one required for features 3 & 4, if the developer is willing to do some algebra to simulate grid cells by using simple points.

Feature 7 is a feature requirement we plan to make easier to implement for developers. Our reserearch suggests that deciding on the exact implementation is best left to the game developers however, since requirements can be vastly different from game to game and issuing player position updates is usually performance and game-play critical. In our demonstrational game this feature was implemented by Lawrence to Rizmari's specification.

This leaves us with the following simplified and generalized list of features required of a general Spatial (Data) Provider component:

1. (Support Game Client) Display rasterized map data as a background for spatial game interactions
2. Request and modify complex spatial objects - a superset of point-based objects
3. (Support Game Server) Track and broadcast players' real-time positions

Feature 2 from this refactored list is therefore the only feature the actual server-side Spatial Data Provider needs to support. The data returned by this feature should provide utility functions to use in conjunction with the Game Servers in-memory player locations.

3.4 Initial design of Spatial Provider

The requirements can be summarized into an Analysis Model of the Spatial Provider component and supporting spatial data functions as shown in Figure 3.1:

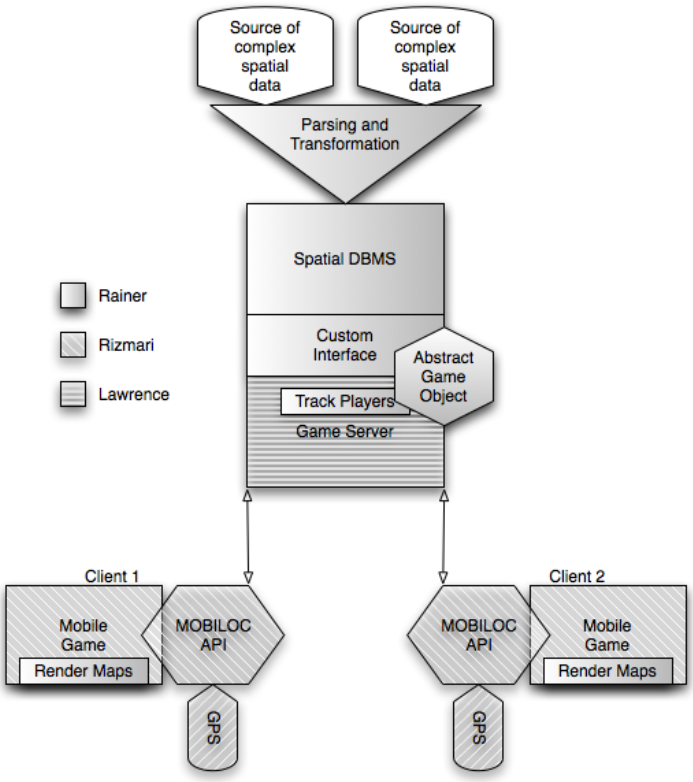


Figure 3.1: Analysis Model

The component needs to use complex spatial data from one or more sources, parse and process it to provide spatial data relevant to game developers, then store said data for efficient queries in a spatial database.

This is represented by tag-shaped input bubbles, leading into a funnel-shaped converter component, which outputs data into a block-shaped database.

It then needs to provide an API layer that abstracts away complex queries and exposes the spatial data in easy-to-use Abstract Game Objects to the game developers' game server. These game objects should also "play nice" with the game servers' internal representation for tracked players.

We represent this by a block shaped adapter sitting at the bottom of the database, which shares hexagonal objects with the game server. These objects also overlap the little block shaped datastructure the game server uses to store moving players in memory.

The component should also offer some client-side library to facilitate the display of rasterized map data as a background for the actual game. This is again represented as a small block inside the larger game client component.

3.4.1 Abstract Game Objects

Abstract Game Objects should be flexible enough to represent arbitrary in-game objects. The game server should have all its requests to the Spatial Provider answered in the form of these objects, and they need to be lightweight enough to be able to be passed to the client and used & modified on the client.

These objects might not necessarily be spatial of nature, but if they are, they need to support spatial functions, such as sorting by distance, intersections, unions and differences.

3.5 The choice of language: Java

The first choice we made as a team was the choice of programming language. We decided that our platform had the highest potential of success if we wrote all components in the same programming language. With the Android phones we were given as development targets, the natural choice for the team was Java.

The main thoughts that led us to choosing Java are the following:

1. The team had a lot of experience with using Java and our combined experience with the language probably outweighed the combined experience we have using any other programming language.

2. We believe that Android game developers will have a much easier time using our platform if all components are authored in the same language, and native Android applications are written in Java, making this a default language for Android game developers to know.

3. Keeping all components in the same languge simplifies maintenance and allows us to easily verify how our components are used in each others code.

4. By writing all of our components in Java, we expected to be able to easily integrate components by importing one from another, such as importing the Spatial Provider component in the Game Server and importing the

Networking Client in the Game Engine on the Android device. This saved us from having to implement protocols and clients in other languages between components.

5. We figured that passing game objects between components and over the network would be greatly simplified by using Java's Object Serialization API. [3]

During the period we made our language choice, I was aware of a few other key advantages Java held for my component specifically:

1. My initial survey of the GIS software scene led me to GeoTools, the open source Java GIS Toolkit, which made me feel confident that GIS tools existed for the Java language.

2. My initial survey of the OpenStreetMap developer wiki pages linked to OSM tools, such as Osmosis, written in Java. This supported Java as a language to work with OSM data.

3. I had used Java Database Connectivity (JDBC) drivers before and knew of, but had not used, Hibernate - an open source Java persistence framework, for which I soon discovered a GIS extension called Hibernate Spatial.

3.6 The choice of geometry: JTS

Complex spatial features can include POIs, roads, barriers, traffic lights, fountains, buildings (even hollowed out by courtyards), concave and convex areas such as fields and parking lots, disconneted areas such as UCT's campus, etc. Representing this complex spatial data is non-trivial.

Ideally the spatial data representation should offer convenience functions, such as the distance between a point and a road, checks to see whether two roads cross or whether a point is enclosed by a building.

Implementing all these interactions could itself qualify as an Honours or Masters project, therefore trying to find a reusable Java package that offers this functionality was paramount.

3.6.1 OGC Simple Features for SQL

The Open Geospatial Consortium (OGC) has published multiple standards for GIS software. One such standard is the "Simple Features for SQL" (SFSQL) standard. This standard defines a set of geometry classes and functions that these geometries should support. Some of these geometry classes are:

GEOMETRY An abstract superclass, from which all other geometry inherits.

POINT A single 2D (or 3D) coordinate.

LINESTRING A set of two or more coordinates, with a linear interpretation of the path between them.

LINEARRING A connected LINESTRING with at least three coordinates.

[3] This assumption did not hold between the Networking layer's server and client.

POLYGON An outer LINEARRING with optional inner LINEARRINGs to define holes in the geometry.

MULTIPOINT A set of points.

...

A small sample of the functions these geometry classes need to support are the following:

GeomFromText Return a specified ST_Geometry value from Well-Known Text representation (WKT)

Intersection Returns a geometry that represents the shared portion of geomA and geomB.

Area Returns the area of the surface if it is a polygon or multi-polygon.

AsGeoJSON Return the geometry as a GeoJSON element.

AsBinary Return the Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.

RotateX Rotate a geometry rotRadians about the X axis.

...

These functions are meant to serve as a small, but representative sample of what the SFSQL standard tries to make possible with geometry objects. While this OGC standard is technically "for SQL", it would obviously be beneficial if a library of Java geometry classes supported the same functions. This would also facilitate a cleaner interface between SFSQL compliant databases and our code.

3.6.2 ESRI's ArcObjects

ESRI offers a commercial solution to represent GIS Geometry in Java (and a slew of other languages), called ArcObjects. Since this is commercial software, we did not consider it an option.

3.6.3 GDAL/OGR

GDAL/OGR is a Geospatial Data Abstraction Library written in ANSI C & C++, but provides language bindings in a large variety of languages, including Java. GDAL itself was originally designed to handle raster data only. OGR was later inspired by the OGC SFSQL spec and got added to the GDAL source tree.

We decided to not use GDAL/OGR, since it is not a pure-Java implementation, and we did not want to introduce the complexity of compiled C code into the project tree.

3.6.4 Java Topology Suite

We found the Java Topology Suite (JTS) to be a complete and powerful implementation of the geometry classes this project requires. It is built as a general geometry library, but provides high enough accuracy operations to be used in GIS applications.

JTS is popular enough to have spawned multiple open source ports into other languages. If this project was using C++ it could leverage GEOM, a C++ port of JTS, and if we were developing for the .NET platform we could use the "Net Topology Suite".

Advantages of JTS

JTS provides implementations for spatial indexes, such as quad-trees and STR-trees, sparing the game developer from having to write custom datastructures to store and query spatial objects in-memory. These spatial indexes alone provide all the functionality the server would require for an in-memory massively-multiplayer PvP game such as Getaway Stockholm 2010.

The Geometry classes provided by JTS also fully-implement the OGC Simple Features spec. This means that if we later choose to use an OCG "Simple Features for SQL" compliant database, we have full support for "complex, non-point" spatial data.

Disadvantages of JTS

One of the drawbacks of JTS is that it provides general geometry classes, but no GIS specific geodetic conversions. Geodetic conversions are conversions from one coordinate (reference) system to another, to provide area and distance results in other units. We had an email conversations with the main developer of JTS, which established the infeasibility of us implementing these for this project ourselves, see section B.1. This is discussed in more detail in section 4.5.

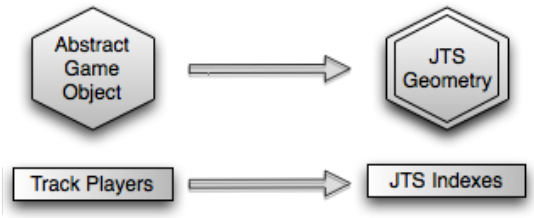
3.6.5 Conclusion

The Java Topology Suite was chosen due to its open source license, feature completeness for standard geometry operations and ease of use. As we will see it later sections it also interfaces with the Object Relational Mapper, Hibernate Spatial, very well.

The Abstract Game Object can now be described as a POJO with a geometry attribute. The geometry attribute will be the abstract geometry base type provided by JTS. In our diagram this is represented as an object-in-object hexagon, with the core object being JTS Geometry.

Additionally, the existing spatial indexes provide game developers with the required datastructures to easily write the in-memory component of their game server.

This choice provides our Spatial Provider component with its first two building blocks:



3.7 The choice of spatial data source: OpenStreetMap

Through research we discovered the following sources of spatial data: the Google Maps API family, SimpleGeo's Places API and OpenStreetMap's APIs and OpenStreetMap .osm files.

3.7.1 Map data formats

Much like digital images, digital maps can be stored and edited in two formats: As raster data and as vector data.

Raster data

Raster data is the more conventional format used for online maps. The reader has probably noticed blocks of map data downloaded by online maps such as Google maps. These blocks of data are called tiles and are displayed by a "slippy map" renderer.

Raster data is often compressed, which reduces its size on disk and during network transfers, but it is still comparatively larger than "raw" vector data, which is a clear disadvantage for mobile applications.

Research note Building a custom tile server is a great educational experience. AListApart [2] provides a detailed introduction to map rendering and open tile server platforms. It is a recommended read for readers, who are new to GIS and spatial data.

Vector data

Vector data is much less common than raster data and the only free source of global vector map data at a high resolution is OpenStreetMap. Other sources exist, such as VMAP and TIGER, but these are either not free, only available for low resolutions or only contain data for a limited number of countries.

Rendering vector data requires the reprojection of each vector in the data for each zoom level and the manual rendering of lines connecting these vertices and filling polygons. This makes vector data more expensive to render than raster data, which can be too computationally expensive on mobile devices with limited processing power.

Conclusion

A developer can imagine having to OCR a rasterized map to extract street names and having to reverse engineer streets' lengths from lines on a map. While possible, extracting features from a rasterized map is more error prone and may lead to a loss of detail compared to simply using vector data to form game objects.

Both vector data and raster data have their disadvantages, which is why existing systems usually combine both: Most games render raster data, like standard Google maps, overlaid with vector data - in the form of simple points. The raster data forms just a background image, whereas all interaction is based on vector data.

3.7.2 Google Maps APIs

Google offers a whole family of APIs grouped together as "Google Maps APIs".

One well known API, the Maps Image API, allows web developers to embed a static map image focused around a point or street address in their websites. Other APIs can be used to embed full slippy maps in developers' websites. A sample is provided as Figure 3.2. The data provided by these Map APIs is all rasterized.

The API most interesting for our project as a source of spatial data that can be converted to game objects is the Google Places API. This API returns a list of POIs around a location, encoded in either JSON or XML formats. A sample "place" is provided in section A.1.

While this data is intersting, no Google API provides more complex vertex data than simple POIs. This means that building interactions based on vegetation types, road types or land use using Google Maps data is very hard.

3.7.3 SimpleGeo API

SimpleGeo provides a "Places API", much like Google's. Some of these POIs or "Places" around UCT's campus and Rondebosch central are shown in Figure 3.3. The data provided about each POI is similar to Google's data.

What makes this data more interesting than Google's is the "geometry" field in each POI, see section A.2. At this point in time the API documentation does not mention geometry types other than "Point", but the field indicates SimpleGeo's intention to expand the API to more complex geometry types in the future, at which point SimpleGeo might become more interesting for developers of complex-data LBMGs.

3.7.4 OpenStreetMap APIs

OpenStreetMap provides a family of APIs, much like Google's. The OSM APIs are less well documented and some have overlapping functionality, such as the main API and the XAPI.

Both these APIs return vector data, structured as XML, with the main difference being that the "Extended API" (XAPI) allows queries over larger ranges and with more input parameters, but doesn't offer the facility to modify OSM data, whereas the normal API allows developers to write tools that can

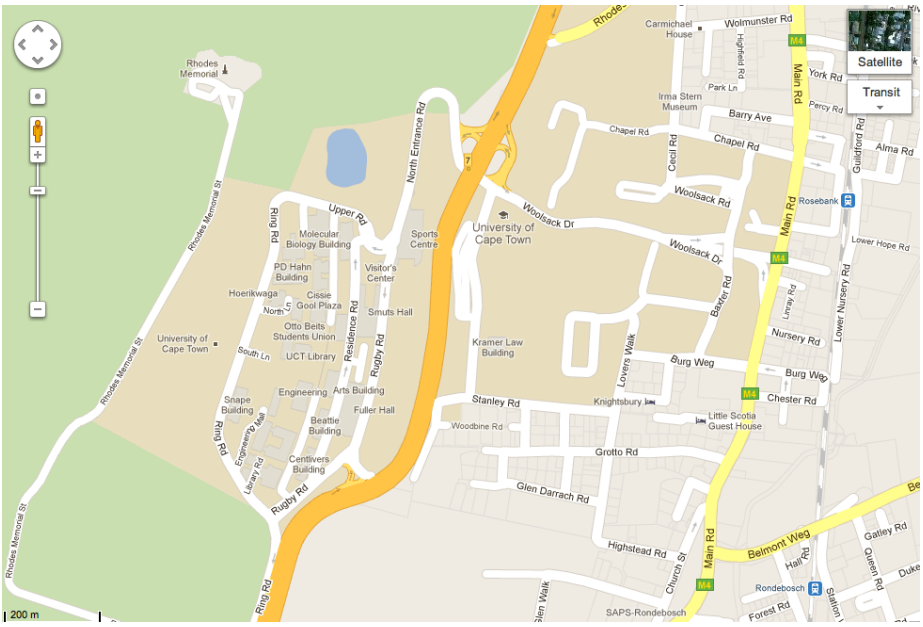


Figure 3.2: Sample Google map raster data, centered on UCT

Figure 3.3: Sample SimpleGeo POI data, centered on UCT

modify the actual OpenStreetMap data that is offered to other developers and rendered as the OpenStreetMap map on the homepage.

CloudMade

Another OSM API option is offered through a commercial third-party provider, called CloudMade. CloudMade offers more reliable, and better documented APIs to read spatial data, much like the XAPI. It also offers language bindings for these APIs in many popular languages.

CloudMade also offers a service tailored specifically to game developers, called Local Discovery for Games. This service can offer the following information in-game:

- The city the player is in.
- The weather in the player's current city.
- The type of location the player is in.
- Distances to the closest locations in a set of location types, such as closest hospital, closest school, pub, store, etc.

The game can then build custom interactions around this information or change background graphics in the game, etc.

Because CloudMade is not free and open-source we did not further consider using its APIs in the Spatial Provider component.

The planet dump

Another entirely different option OSM offers is the "planet dump". The dump file is a complete dump of the global OpenStreetMap database. This file, too, is structured as XML, henceforth refferred to as OSM XML and described in subsection 4.3.3.

Community sites and Cloudmade offer extracts of the planet dump for download. Some sites offer per-city downloads for European cities, but Cloudmade opted to split files on a per-continent, per-country and per-state basis. We downloaded South Africa's Western Cape province for testing purposes and found that the size of the BZIP compressed *western_cape.osm.bz2* was only 11.3 MB (accessed June 2011).[4] When decompressed the actual data's size equals 155.6 MB of XML.

OpenStreetMap's Geometry Model

OpenStreetMap describes all map features in three distinct classes, also referred to as "elements":

1. Nodes: A node is the most basic element and models a geospatial point. It has a latitude and longitude and an optional altitude value, all of which are floats.

   Nodes are used to model POIs, but are also components of the other element types, so do not necessarily carry any POI information.

[4]When we downloaded the file again in October 2011 it had grown to 12.7 MB, so South Africa's OSM data still seems to be getting more detailed.

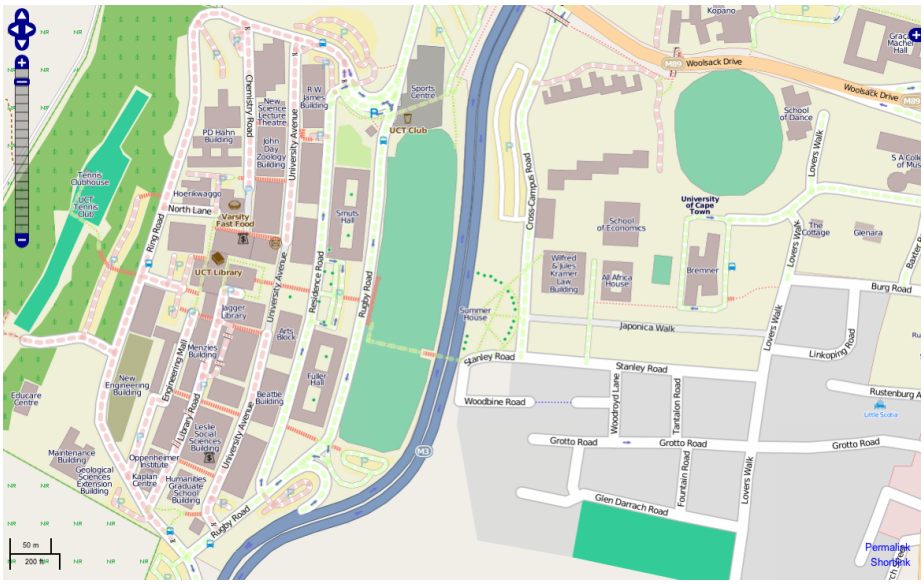


Figure 3.4: Sample OpenStreetMap map raster data, centered on UCT

2. Ways: Ways are a collection of nodes, referencing the nodes by ID.

   Ways were named ways to reference all types of streets, roads and paths. They were conceptually expanded to also model areas: A way that starts and ends on the same node is considered closed and models an area.

3. Relations: Relations are collections of the other two types, also referencing them by ID. Relations' members are called "members" and these members can have an optional "role" attached.

   Relations can be used to describe any real-world geometry that does not conceptually map to nodes or ways. A country might be made of two or more unconnected areas such as the USA, which is modelled as a relation with multiple members which are (connected) ways.

   Another use case for relations is a case we can observe on UCT's campus by looking at Smuts and Fuller Hall. Both these residences have two courtyards, which hollow out their "building ways". This is mapped by having multiple ways as members of the building relation, one of which is assigned the role "outer" and some of which are assigned the role "inner".

All three element types have IDs attached, which are unique integers within their element class. Most APIs also attach user & timestamp keys to indicate who last edited an element and when.

3.7.5 Conclusion

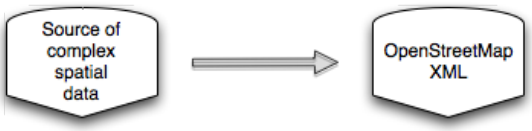
If one compares Figure 3.4 to Figure 3.2, it is clear which map prodives the higher level of detail. Many building outlines are accurate on OpenStreetMap, but do not feature on the Google Map at all. While the residences like Smuts

Hall are deformed polygons on the Google Map, OpenStreetMap even has the palms growing in their courtyards tagged.

OpenStreetMap's point data is more detailed than both SimpleGeo's and Google's, since OSM data also exposes points which are not "Places", such as ATMs, fountains, trees, etc, and tends to have roughly the same number of actual POIs registered as both competitors.

Since OpenStreetMap won in both categories, we decided to use OpenStreetMap as the only data source provided by default, but leave other developers the option to implement more or other datasources at a later stage.

This choice provides our Spatial Provider component with its second building block:



3.8 The choice of spatial database: PostGIS

During the early phases of this project we considered pitting a number of databases against one another and running some benchmarks to see how well they performed. We considered using SQL and NoSQL databases.

3.8.1 Spatial NoSQL databases

Many NoSQL databases claim to scale better than traditional SQL databases. Some claim to return queries faster, some can perform writes faster than reads. It would be interesting to test some of these claims using complex spatial data.

Since so-called NoSQL databases are a still relatively new paradigm in computing, it did not come as a big surprise that not many of them support spatial functions. We discovered rudimentary spatial support in MongoDB and a spatial extension for CouchDB, called GeoCouch.

GeoCouch is the least feature-rich database we encountered during our research. It allows the developer to attach points to CouchDB JSON documents, define an index on the point field and perform a bounding-box query. While this functionality is nowhere near allowing us to manipulate roads, perform "within building" queries, etc, it might be sufficient for "check-in"-based games such as Foursquare.

MongoDB's support for spatial objects is limited to points as well. Currently it can index points embedded in documents and can do bounding box queries, like GeoCouch. Additionally it supports "near" queries, which take a center point and return a set of documents near said point. MongoDB does not save or index any more complex spatial objects. Again, this might work well for point-based games.

ReadWriteWeb [25] shows a fascinating talk from an engineer working for SimpleGeo, the provider of a POI API discussed earlier in subsection 3.7.3, extending the distributed NoSQL datastore Cassandra with a spatial index. The talk goes into great detail on how to do geohashing using space filling

curves. Unfortunately the actual code is proprietary and Cassandra does not provide any spatial indexing out-of-the-box.

An article by DirectionsMagazine [5] confirms and summarizes some of our conclusions: NoSQL databases work well for very large scale games with POI-centric gameplay.

3.8.2 Spatial SQL databases

OpenGeo [22] provides a detailed overview of (SQL) Spatial Database options. The most feature complete options are "Oracle RDBMS with Spatial or Locator", "Microsoft SQL Server with Spatial" and "PostgreSQL with PostGIS".

Microsoft SQL Server is obviously non-free and non-open, so we exclude it from our list of potential databases.

Oracle's database is proprietory, too, but at least offers a free license. This license however limits the database's RAM use to one Gigabyte and uses only one CPU core. This gives Oracle Spatial a major disadvantage in performance tests and would probably deter game developers from deploying it as part of our free platform.

The article[22] goes on to describe a less feature-rich option - "MySQL Spatial". It follows the OGC SFSQL specification, but does not offer many of the spatial functions offered by its competitors. We decided to exclude it from the potential DBMS roundup.

A last option is "ArcSDE", but this is part of a commercial "ArcGIS Server" and is therefore of little interest to this free & open source platform.

This leaves us with the winner by elimination - PostGIS.

PostGIS

PostGIS is a free and opes source (FOSS) GIS extension for the popular FOSS PostgreSQL database. It is one of the most feature-rich spatial extensions available and fully OGC SFSQL compliant.

The official OpenStreetMap wiki recommends the use of PostGIS with OSM data over other spatial databases and the Osmosis toolkit has first-class PostGIS import and export support.

Geometry PostGIS provides standard Geometry and a separate class of spatial objects called Geography. Geography is less feature rich and computationally more expensive, but accurate globally and provides some geodetic conversions, such as from WGS-84 degrees to metres, without requiring the user to understand spatial reference systems and reprojections.

Indexes PostGIS builds interesting and powerful indexes: PostgreSQL supports B-Trees to sort data, such as numbers and strings, along one axis. Since spatial data needs to be sorted along (at least) two axes for meaningful queries, it uses R-Trees instead.

Because R-Trees are limited to a maximum key size of 8K for their indexing scheme, and for performance reasons, PostGIS builds its own R-Tree implementation on top of a standard PostgreSQL GiST (Generalized Search Tree) index.

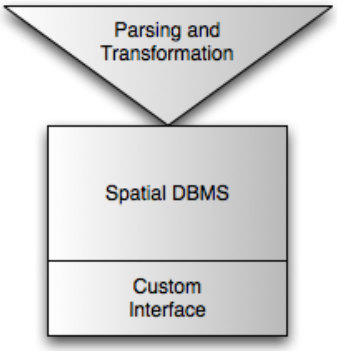


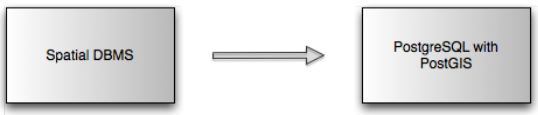
Figure 3.5: Interface between Spatial Provider and database

To do this it can construct a "lossy" boundary box around very large spatial features, which keeps the indexes' memory foothold low and leads to faster queries.

3.8.3 Conclusion

After reviewing all spatial database options, we were left with just PostGIS. PostGIS is very powerful and provides complex operations on complex spatial objects.

This step further completes our System Analysis diagram and gets us closer to a full System Design.



3.9 The choice of ORM: Hibernate (Spatial)

With the choice of language, geometry objects and database fixed, the next question is how to interface with said database from Java code.

The Spatial Provider component has to interface with the database in two key places: when spatial data is entered by the parsing / transformation stage and when data is queried from the database or updated by the game server. This is best illustrated by looking at Figure 3.5.

Each game developer will attempt to store game-specific information with the spatial data. Monopoly City Streets, for example, would need to store the current value of a street with the streets spatial information, whereas Foursquare would save the name of the current mayor at a location. To facilitate the storage of widely different game information with each spatial feature, we will have to provide extendable spatial game objects.

3.9.1 JDBC

In Java it is possible to use the JDBC drivers to write client-side SQL code by hand and to directly operate on databases. The purpose of the JDBC is to wrap different SQL database drivers in a common set of APIs, but it does not compensate for the different "SQL dialects" spoken by different databases.

Object input and output

The JDBC also does not provide any persistence-wrapping for whole Java objects. This means that it is necessary to write a method for each type of object that the developer want to be persistent, which builds a SQL insert or update query by inspecting every attribute that is meant to be persisted.

This by itself would constitute quite a lot of work already, but if the object were non-final and shared between clients, each client would also have to periodically check the database entry to see whether another client modified said object. This check would then have to cope with inconsistencies and conflicts, if the object was modified locally.

Extensibility

If a game developer has to modify a game object to add an extra attribute, say the number of houses built on a street in Monopoly City Streets, he would have to also modify the SQL code he previously wrote to save, query and update every object. This means that the Spatial Provider platform's game objects would only provide a blueprint for how to write SQL code, but most of the work would have to be done by the game developer.

Spatial support

Another disadvantage is that standard JDBC drivers do not have spatial input and return types defined. The developer would have to convert a spatial object to WKT or WKB, then call a FromWKT function on the spatial database, to input objects in a geometry database field. The reverse is true to output spatial objects.

Conclusion

This leaves us with the impression that using pure PostgreSQL language and JDBC drivers would result in a lot of additional work, for both the platform developer and the game developer.

3.9.2 Object Relational Mapping

One solution for many of the problems mentioned in the pure-JDBC interface implementation is to use an Object-Relational Mapper for Java. Object Relational Mappers define the mapping from an object to SQL to facilitate persistence and often provide additional features, such as transactions and schema genereation.

Wikipedia currently lists 21 ORM Software projects for Java, most of which are open source. Due to the overwhelming choice available we focussed our research on finding an ORM solution that provides Spatial support. The research revealed DataNucleus' JDO Spatial and Hibernate Spatial.

DataNucleus Access Platform (JDO)

Java Data Objects (JDO) is a persistance specification, which uses external XML files to specify how to persist "plain old java objects" (POJOs). This is fully transparent peristence, ie it does not require the Java classes to implement any interface or import any extra code, which means that the classes can be safely passed around without introducing external dependencies.

JDO is just a specification, however, and is implemented by a number of vendors. One such implementation is the "DataNucleus Access Platform", an open source project, which has Spatial JDO classes with JTS support.

Using DataNucleus' JDO, the game developer can very easily extend our existing, non-persistend classes, add a few XML mappings files and add persistence using a bytecode-enhancer. After making the classes persistent, the developer can use simple calls to PersistenceManager.makePersistent() to automatically have all changes to the objects reflected in the database.

Hibernate (JPA)

Hibernate is a popular open source ORM and our research revealed Hibernate Spatial, an extension for Hibernate providing support for storing and querying JTS Geometry using the PostgreSQL JDBC driver.

Hibernate leverages a different Java persistence specification, called the Java Persistence API (JPA). JPA is not fully transparent, unlike JDO, which means that classes implementing JPA have dependencies on the javax.persistence package. This is a disadvantage when the game designer plans on passing light spatial game objects to clients, which might not provide javax.persistence.

Queries One of the advantages of Hibernate is the Hibernate Query Language. Queries are constructed by chaining JPA Criterium objects together. This allows us to write very finegrained queries, such as "all objects within a certain bounding box that overlap a street's geometry and are within X meters of a player".

Annotations JPA persistence can be defined using annotations. This means that a class annotated with "@Entity" will be made persistant automatically. Attributes can be annotated with annotations such as "@Id" to define a primary key or "@Transient" to define attributes that do not need persistence.

Spatial dialect Hibernate can be extended with "dialects". One such dialect is provided by the Hibernate Spatial extension. Once the spatial dialect is imported in Hibernate's configuration, it automatically registers all JTS geometry as Geometry tables in PostGIS.

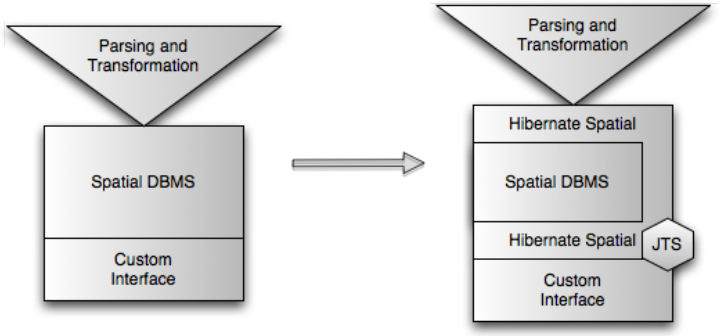
Schema inheritance Hibernate offers multiple inheritance schemes, such as "table-per-class" and "discriminator column", to deal with inheritance, all of which behave very well with queries. An example: If Pub and Foodcourt entities inherit off the POI entity, a query for POIs will return POIs, Pubs and Foodcourts. A query for Pubs will obviously return only Pubs.

3.9.3 Conclusion

We chose to use Hibernate for a few reasons:

- We heard about it during recrutitment talks from S1 and BSG and it sounded like an industry standard
- Hibernate's community and documentation seemed more active and helpful

Using Hibernate Spatial the main Spatial Provider component can be redefined as the following:



This diagram illustrates how the database is never directly accessed by the component we are implemting. Instead everything passes through Hibernate Spatial, which fully understands JTS Geometry as a data type.

3.10 The choice of map renderer: MapsForge

One feature requirement we have not yet fulfilled is that of helping a game designer with getting rasterized map data as a background for game interaction on the device.

3.10.1 Android Google Map activity

The first solution we described involved using Android's native map activity in conjunction with Google Maps.

We mentioned a drawback being that to render these maps the player needs to be online whenever their map view moves into a new map tile or the player zooms the view in and out to different view levels.

No custom rasterized map source

Another drawback we did not mention earlier is that the "look and feel" of Google Maps is fixed. It is impossible, for example, to request a Google Map where all roads are green, rivers look like red lave, etc.

The developer has the ability to create custom overlays on top of a Google Map, which can be used to theme the map to a certain extent, but unless the developer completely fills the overlay view with objects, there will always be a non-themed Google Map visible underneath.

Conclusion

Using the Android map activity is still a valid solution, since many games require a data connection to be active for multi-player features to work in any case. It also has the major advantage of being very little additional work for this project, since the codebase is fairly mature and there is a lot of community support online.

3.10.2 Offline map renderers

We also mentioned the possibility of imitating the functionality of GPS devices, such as TomToms and Garmins, which pre-download the necessary maps for a large area (a country or state) and store this map data on the device, to be rendered as the need for it arises.

We found two open source map view activity libraries which implement exactly this functionality. Both of these use OpenStreetMap vector data, which can be stored on the device and rendered at runtime.

OSMDroid

OSMDroid is an open source replacement for the native Android map activity, leveraging OpenStreetMap or Cloudmade map tiles or custom tile servers.

We evaluated OSMDroid's offline rendering capabilities, but found them inferior to the alternative, MapsForge. OSMDroid simply allows the developer to package up pre-rendered map tiles at different zoom levels in a zip archive and can read these from Android storage.

MapsForge

MapsForge tries to closely emulate the native Android map activity's behaviour as well, but takes a different approach to rendering data: It allows the developer to use a custom binary format to store OSM data with pre-defined theming information.

Instead of simply zipping rasterized images, developers use an Osmosis plug-in which has the ability to write these specialized ".map" files. This allows the data to be compressed to a more compact format than standard rasterized map data, but does not render the data as quickly. A large advantage however, is that it allows the developer to specify render styles and product tiles customized for the theme of the game.

Much like the other map views, MapsForge maps can be overlaid with overlays to provide game-specific information, such as a HUD and map markers to display game objects or other players.

If the developer does not provide MapsForge with enough map data is can fall back to requesting standard rasterized maps from a provider such as Google Maps, Cloudmade or a custom tile server.

Reusing spatial data We had an email conversation with the maintainers of MapsForge, see section B.2, to establish whether the underlying binary map data can also be used in a game, instead of having to separately load (possibly the same) spatial data to build game interactions. Unfortunately this does not seem possible, since MapsForge's data format is highly optimized for rendering purposes. (The data is not modifiable, for example.)

Conclusion

Even without access to the underlying spatial data, MapsForge most closely matches our requirements for an in-game map renderer. While the Spatial Provider component itself does not rely on displaying map data, we can use MapsForge to provide rendered spatial data on the client.

This completes another choice on the Analysis Model:

Research note As a part of our design phase we did install a simple map rendering and tile server stack: osm2pgsql to import OSM data into a PostGIS database and "mapnik" to render tiles. We found that setting up this system to produce custom map tiles is not very challenging, and many good community tutorials are available, in case the game developer would rather choose a different renderer.

3.11 Design Model & Conclusion

Using these components, we can now refine the Analysis Model to the Design Model (Figure 3.6).

We replaced as many vague component names with concrete technology choices as possible:

- The sources of complex spatial data are now just one source: OpenStreetMap XML. We leave space for the implementation of future sources, though.
- The Spatial DBMS turned into PostGIS, wrapped in Hibernate for easy Object Relational Mapping of our input objects.
- The "Abstract Game Object" that had to be better defined, is now a POJO that contains a geometry attribute, which is of the JTS Geometry type.

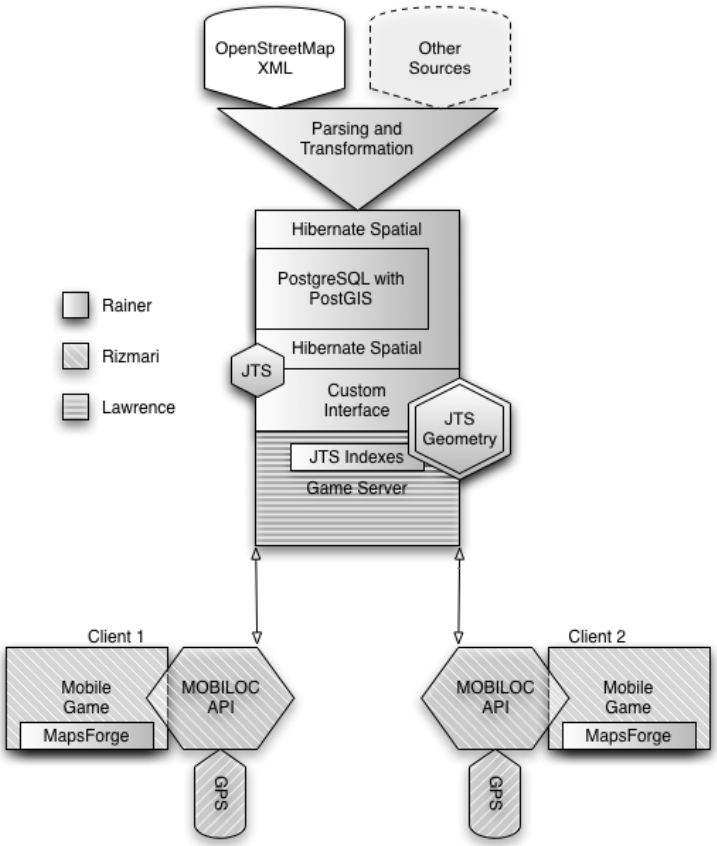


Figure 3.6: Design Model

- The in-memory datastructure to store moving players in the game server is now one of the spatial indexes offered by JTS.
- On the client we now offer an Android mapping activity by MapsForge.

This leaves us with two components we need to implement ourselves:

- The input stage, which takes OpenStreetMap XML and transforms it into Hibernate Entites, which it then persists in the PostGIS database.
- The "custom interface", which abstracts away the Hibernate API and allows the game server to easily query for spatial data and save custom data.

We describe the detailed implementation of these components in the following chapter.

Chapter 4

Implementation

4.1 Set-up procedures

This section describes the software used to implement the Spatial Provider component and the methods for setting up a GIS development environment. subsection 4.1.1 describes the basic tools used in the development of the platform and identifies software that is useful for GIS inspection and debugging purposes. After laying the groundwork with these basic tools, the setup process for more complex software, such as PostgreSQL and Hibernate, which is central to the component is described in detail.

4.1.1 Tools

The following is a list of tools which were invaluable in the development process:

Eclipse, Netbeans & Maven

Eclipse and Netbeans are two full-featured IDEs. Maven integration is a feature of these IDEs which is valuable during the development of a complex Java project. Maven is a Java dependency management and build system, that lays out a basic project structure and dependency file, and manages the complete build process for the developer.

Managing the build process includes downloading the necessary dependencies and dependencies of dependencies recursively, then placing these on the class path used in the build process. The build process can be customized to produce fully packaged versions of a project. The packaged version of this component, with dependecies included, weighed in at a hefty 8.8MB.

Packaging the component for this project with Maven allowes other developers, such as the developer of our Network component, to simply import a single *.jar* file into their project and automatically have all libraries used available in their code.

JOSM & QGIS

JOSM and QGIS are GIS applications used to inspect spatial data. JOSM is specialised to view and modify OSM data, either using the OSM API or OSM

XML files as input and output. JOSM is a useful tool for debugging in certain cases, for example: when identifying why a specific OSM node did not get parsed. Figure 1.1 is generated using JOSM.

QGIS is a more general GIS toolkit. It is focused on reading and modifying a large number of spatial data sources. QGIS can directly read PostGIS database tables and recognizes their geometry columns, if any are present. QGIS has been used extensively to visually inspect the state of this project's database contents.

Pitfall The most recent version of QGIS (1.7) has a broken OSM file viewer plugin, the project therefore used QGIS 1.6.

pgAdmin3

'pgAdmin3' is a visual DBMS tool for PostgreSQL databases. It is useful to inspect database table structures manually, dump random sample records and query for specific data.

Osmosis with MapsForge MapWriter

Osmosis is a command-line OSM processing application. It is able to read and write OSM data from a variety of sources, such as databases and files, and is extensible via plugins. One such plugin is the MapsForge MapWriter used to create the compact binary map files used by MapsForge's Android offline map renderer.

4.1.2 PostgreSQL & PostGIS

Setting up a PostGIS enabled PostgreSQL database is much more complex than setting up the tools mentioned in the last section and is paramount to using the Spatial Provider component, which explains the separate section dedicated to its setup.

Installation

Over the course of this project the combination of PostgreSQL and PostGIS have been installed on three platforms: Mac OS X (Snow Leopard & Lion), Windows 7 and Ubuntu Server 11.04.

Initial testing was done on an EC2 instance running Ubuntu Server 11.04. This operating system & hosting combination was chosen because it is easy to use the aptitude package manager to install the necessary PostgreSQL binaries and PostGIS libraries, without going through a compilation step and the high availabitily and scalability of an EC2 system.

The installation process creates an unix user called "postgres", who owns the database. Creating unpriviledged users to run long-running services with public sockets is a best-practice security measure on many unix systems. This mechanism works by ensuring that exploited services are exploited at the unpriviledged level of the user who is running them.

To set up the database an application-specific user called "mobiloc" was created. PostgreSQL's createuser & createdb commands were issued as the postgres user to create a database, called "gis", for the mobiloc user.

PostGIS was then enabled on the database by running a set of *.sql* files distributed with PostGIS. These files create a table containing map projections and SRIDs in the specified database and create the SQL functions which are then usable in the database.

The separate installation of PostGIS is easier when using Windows systems: A spatially enabled database can be created using the "StackBuilder.exe" shipped with PostgreSQL in Windows. It offers a simple visual wizard that lets the user create a new database and tick "spatial extensions".

The pitfalls in section 4.1.2 led to testing PostGIS locally on a Macbook Air. The installation was similar to the installation on Ubuntu, with the exception of not having a package manager. Some community members bundle PostgreSQL and PostGIS in standard Mac "dmg" installers. Unfortunately the user still has to use command line tools to create database users & databases and install PostGIS extensions manually for each database created.

Testing

The database was tested using osm2pgsql and Mapnik. Osm2pgsql is a small utility script used to import a bzip2 compressed planet.dump extract (South Africa's Western Cape) into the database. This process took 20 minutes, which is fairly quick to load the map data of a $129,462km^2$ province.

After installing additional python libraries, such as PIL, and changing the boundary-box parameters in the mapnik script to the target area, it was possible to render a map of UCT's campus using Mapnik and the data contained in the database.

Pitfalls

The necessary ports were opened using EC2's security groups and PostgreSQL was set to accept connections from non-localhost IPs to inspect the OSM data using QGIS. It became apparent that the Western Cape is too large to be opened in QGIS from a remote PostGIS server. QGIS queries for the complete data source (table) by default, to display an overview of all the data. While this is sufficiently fast for small databases on local systems, the query needs to be restricted to a smaller area using PostGIS' SQL functions and QGIS' query designer for a larger remote database, such as the Western Cape.

4.1.3 Hibernate Spatial

The Hibernate documentation recommends using Maven to set up Hibernate with its necessary dependencies. This process consists of setting up Maven, following Maven's project structure and adding a few lines to the Maven project-specific configuration files. A simpler alternative is to use an IDE, such as Netbeans, with full Maven support built in.

The Spatial Provider component was created by starting a Maven project from within Netbeans and adding Hibernate as a dependency in Maven's pom.xml project property file.

Hibernate, like Maven, is configured through an XML project file, which defines the database connection to be used and any classes that need to be mapped.

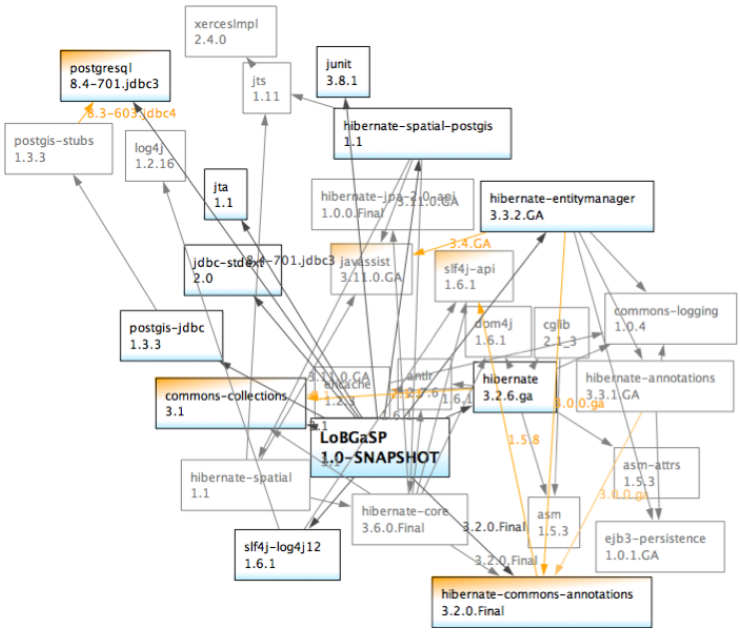


Figure 4.1: Dependency Mess

Every class' mappings can be defined either through annotations or through more XML, in either the main project XML file or via smaller XML files, included in the main XML file.

Pitfalls

Annotations It seems counterintuitive, but even when using Hibernate annotations in entity classes, they are still required to be listed in the XML configuration file manually. This makes the component harder to use, since developers cannot simply inherit off one of the Spatial Provider's entities to have their own entity class without modififying the XML file that is packaged within the component's *.jar* file.

One workaround is to scan the whole project's classpath with the comically named "ClassPathScanningCandidateComponentProvider" class (part of the Spring Framework). The problem with this is that importing Spring would have made this component even larger. One could also write a class to scan the class path for entities manually.

Another annotations related pitfall is the need to place annotations on either attributes or getters, but not mix the two, otherwise Hibernate will not persist one of the two.

Dependencies & size bloat Copying Hibernate requirements into Maven's project file is easier than manually downloading a large number of ".jar" libraries, however it creates an incredibly complex dependency graph and a very large distributable jar (over 8 MB). Figure 4.1 shows the mess created.

Excessive logging In order to turn off certain log messages one needs to include a different version of the log4j library to the one that is used in all Hibernate tutorials. Since much of our project evaluation relied on the game server collecting clean server logs doing this was vital.

4.2 Designing Abstract Game Object

The design of the abstract game objects that would be sent between the component and the game server and client was crucial to this component's success in achieving usefulness through generality. The component has been designed around the following assumptions:

- The game developer only passes objects to be persisted to the spatial provider, since non-persistent objects can stay in the game server's memory.
- The abstract game objects should be serializable to pass them directly to the client with minimal effort and introduce few extra dependencies on the client.
- Extending the provided spatial objects with custom attributes and persisting them should be easy to implement.
- If desired the game developer should be able to specify a mapping between pre-defined real world entities and game specific entities; if a parking lot is represented as an urban zone, the provider should be able to map between "urban zones" and "parking lots" transparently.

4.2.1 Abstract base object

The base object is a persistend database entity, but it is not necessarily spatial. This allows the developer to build a large hierarchy of custom game objects parallel to the tree of spatial objects. These non-spatial objects share a database with the spatial provider, but are not managed by it.

The only predefined field on the base object is an ID collumn, which is assigned a unique (in this database) ID by the databse, when the object is first persisted. Creating some unique primary key is a useful strategy to use Hibernate's queries and chaching efficiently.

A base object with no information other than an ID is absolutely useless. Due to this the base class was made abstract to prevent the accidental creation of unnecessary database entities.

Other non-spatial entities

No other non-spatial entities were created using this Spatial Provider so far, but it is possible to do so. Due to the fact that the Spatial Provider cannot predict all potential use cases for non-spatial objects, no API to save or query them is provided.

Therefore, the developer should define queries and idexes for these obejects separately, since they will not form part of the same spatial hierarchy, they will not be indexed on a geometry column. Doing this should be trivial using

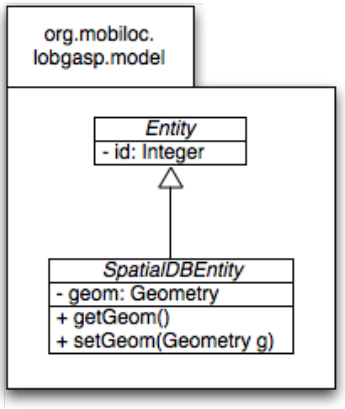


Figure 4.2: Abstract Base Objects

standard Hibernate annotations to define the persistence and the query API to create game-specific "views" of the data.

4.2.2 Abstract spatial object

The abstract spatial object inherits off the simple base object and contains an attribute of the JTS.geometry class, called geom. While it is possible to define multiple geometry attributes on the same object, there are very few use cases for doing so[1]. This one geom attribute is used throughout the whole inheritance tree to specify a consistent interface.

This class is abstract, exactly like its parent, because having a feature only in 2D space with no type information attached is of little use.

4.2.3 General inheritance strategy

On top of the abstract spatial base object, a set of other inheritance trees can be defined. One logical approach is to follow the OSM structure and create a tree for points, a tree for (potentially cyclic) lines and a tree for anything more complex.

Object Orientation

This structure has the advantage of clean object orientation: a point can declare some utility methods on its base class, which make sense for points only, such as getX() and getY() or getLon() and getLat().

Higher complexity objects, such as lines (roads) and cyclic lines (areas / buildings) could have a getX() method[2], it would just not be immediately clear to a developer what the purpose of this method is.

[1] One such case is specifying separate collision geometry and display geometry, but larger collision checks can be simulated by increasing the distance from an object that counts as a collision instead.

[2] It could return the x component of the feature's center point, for example.

Implementation

The component is meant to deal with a variety of sources and as such each source should implement its own "model" sub-namespace to extend the SpatialDBEntity. This makes it possible to cherry-pick how to match source objects on a source by source basis to game objects during the parsing stage.

Figure 4.2 shows the base objects in their root namespace. The attributes are private, but can be accessed through public getters and setters. This is not recommended for the ID attribute as it functions as a database index.

4.3 Designing Parser & Transformer component

The purpose of the Parser & Transformer stage of the Spatial Provider is to read data from a chosen source or sources and transform it into game objects. These can then be persisted in the database to be provided to game clients in the relevant regions later via database queries.

If the platform's goal was to produce a platform for games, such as 'Geocaching', in which all game objects (POIs) are user-generated, the parser and transformer component would be unnecessary. For all other game types it is necessary to start with spatial data that will form game objects.

Figure 4.3 shows the main strategy. The user defines a set of mappings; the spatial data source is matched against these mappings to create game specific spatial objects.

4.3.1 Exposed API

The parser stage simply returns a full set of features found in the data source. The transformer stage iterates through this set to select the spatial features required by the developer and saves them to the database, with slight modifications, if necessary.

The API is simple and consists of only two functions: the developer specifies mappings from real-world features to game objects, then provides a source of data. Figure 4.4 illustrates this sequence of events.

4.3.2 Selecting an OSM data source

In section 3.7 the choice to use OSM data as the primary data source is justified, and the OSM data model is described briefly. The section does not further explain how to obtain OSM data programmatically or how to parse the XML efficiently. This section presents the method chosen to obtain OSM XML data, explains pitfalls assosciated with the format.

Planet.osm file

One source of data is the planet OSM XML file. It contains the entire planet's OpenStreetMap data and is currently 18 GB when compressed and over 250 GB of XML when uncompressed.

Working with this file directly has some disadvantages:

- 18 GB is a massive download.

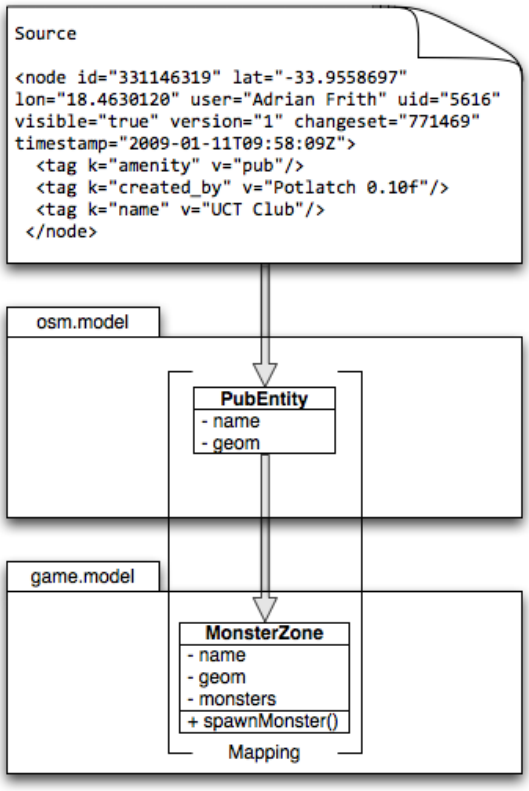


Figure 4.3: Parser & Transformer Strategy Model

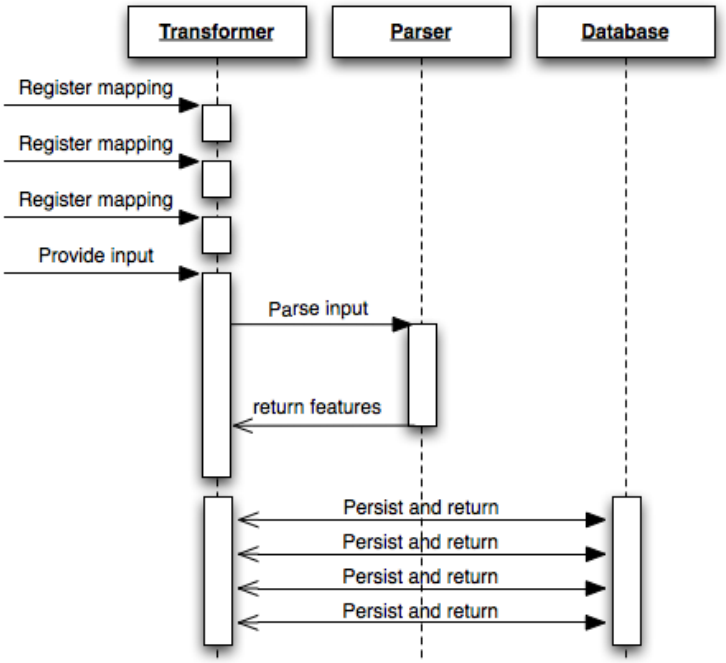


Figure 4.4: Parser & Transformer Sequence Diagram

- Inspecting the contents is tricky while it is compressed, but it takes a large amount of disk space to uncompress.
- Many tools, such as text editors, will simply crash rather than opening the file partially.
- Finding an item in the file to modify or inspect takes a very long time.
- When the file is out of date, it has to be replaced with a new large file.

The OSM community invented a few workarounds:

- The community offers changesets, which can be used to update an existing planet.osm file.
- The Osmosis tool can be used to transform this file into more useable "extracts".
- It is possible to recompress the file using gzip, which increases file size, but decreases CPU processing time.

Using planet OSM and changesets is time and bandwidth consuming and requires a fast server and a lot of disk space to perform processing tasks quickly.

Planet.dump extracts

The second option is to use smaller extracts of the planet.osm file provided by community members.

South Africa's OSM extract provided by CloudMade is currently 58.1 MB when compressed & the Western Cape is only 12.7 MB. Even uncompressed the files are still managable at 839.7 MB and 185.9 MB on disk.

One disadvantage of this approach is that the extracts are not always up-to-date. At the time of this writing (26 October), the extracts are exactly 2 months old.

OSM API

Another option is to input data, as it is needed by the game server, using the OSM API or XAPI.

The advantages of this method are that the Spatial Provider will waste very little bandwidth downloading large osm files and that this method could occasionally update the saved spatial data with new data from the OSM API.

One disadvantage of this method it that the data still has to be parsed, transformed (filtered) and persisted before it is available to the client and that this can lead to lag between a player joining the game and the spatial data becoming available.

OSM's APIs are not continually available, which can cause problems with this approach. If a player in a new region joins the game while the OSM API is unavailable, the game would have to turn the player away.

Another disadvantage is the additional complexity the Spatial Provider would need to handle this data source: The component has to check whether each query is in the bounds of the previously requested queries to see whether or not it needs to query additional data from the OSM APIs.

Discussion

The CloudMade extracts are well suited to this particular project's requirements. The data is easy to download, easy to decompress and can still be readily explored using command-line tools such as "more" and "grep".

Finding all references to UCT in South Africa, for example using "time", "cat" and "grep" takes only 6.503s of real time and produces 11 results.

The data does not have to be 100% up-to-date, because not much has changed on UCT's campus in the last two months. The main disadvantage of using the extracts, the possibility of data being too old, hereby becomes absolete.

4.3.3 OpenStreetMap's XML format

API callbacks returning XML, such as the main API and the XAPI, share the same structure as planet.osm files and its extracts. These are the following:

1. A standard XML header tag.
2. A OSM root tag, containing API/Format version and generator attributes. This tag contains all following tags.
3. An optional bounds tag, containing bounding box elements, such as minlat and maxlon to describe the area the data covers.

4. A number of “node” tags, which have id, lat & lon attributes and additional attributes such as a version number, the author who created them, ...

5. A number of “way” tags, each of which has “nd” child-tags, containing “ref” attributes to node ids.

6. A number of “relation” tags, each of which has “member” child-tags, containing “type” and “ref” attributes.

7. “node”, “way” and “relation” tags can additionally contain “tag” tags, which are key-value pairs describing their parent tag. These tags contain valuable information for potential game designers, such as a road’s type, a building’s or shop’s name, the type of amenity provided in case of restaurant nodes, etc.

A complete XML file, describing the UCT Pub, can be inspected in section A.3. This demonstrates the tag necessary to identify a pub as an element with a <tag k=”amenity” v=”pub”/> tag.

Parsing problems

The fact that all elements of the same type are grouped together seems to make the file better structured, but it complicates parsing. This is a disadvantage of OSM’s XML format.

- Because ways and relations contain only references to their members and not the actual member data, constructing a complete geometry requires multiple lookups in some temporary datastructure containing all previous elements.

- Similarly, because nodes do not have the same tag information attached as their parents, filtering nodes that will later belong to uninteresting ways / relations is impossible during the node-reading stage.

- Parsing very large files becomes difficult as the datastructures sizes increase beyond memory limits.

Due to this unfortunate structure OSM tools have to define large disk-backed datastructures or use databases to parse the very large data sources, such as planet.osm or continent sized extracts.

DOM & SAX parsers

DOM parsers build a Document-Object Model to represent the complete document as a traversable tree in memory. They are not suitable for parsing very large XML files, such as planet.osm, because they will generally parse a complete XML document until they find the end-tag and create a large number of nodes in their tree datastructure. Some DOM Parser implementations can fall back onto disk memory if the system’s main memory runs out.

SAX parsers work by emitting events for every opening and closing XML tag they discover during parsing. The developer using a SAX parser can define event

handlers that track state. By knowing the containing XML tag, the developer can ignore / instantiate objects as the XML file gets parsed.

The amount of memory used by a SAX parser is directly related to the size of the deepest-level XML element, which is smaller or equal to the size of the whole document by definition.

Discussion

Using a SAX parser appears to be advantageous, as it allows data processing to be interweaved with the IO operations to read in the data. It also requires less memory than a DOM parser, since OSM XML elements are small and never deeper than 3 levels.

4.3.4 Mapping OSM data to Entity inheritance strategy

The OSM data model defines only three classes of objects: Nodes, Ways and Relations. These are tagged to virtually transform them into other objects, during map-render-time. This is the main purpose of OSM data: to be rendered.

Rendering schemas

Tools built primarily for rendering, such as osm2pgsql, build only three database tables: nodes, ways and relations. Each one of these tables contains a column for every possible tag-key, and most of these columns are empty.

This simple data model works very well for map renderers: The map renderer is not interested in a specific type of object, it wants to query for all objects within a certain tile. It will then order them, ignore some details if the map scale is large, and finally draw the features.

This project using a standard schema created by a tools such as Osmosis or osm2pgsql was considered. This would have been advantageous in several ways:

- These tools are stable and likely to have fewer bugs than the parser / transformer component being created.
- These tools are built to handle large OSM data such as the planet.osm file.
- Using these may have resulted in less work during this project.

Unfortunately the database schemas these tools create were not suitable for this project for a number of reasons:

- In order to search for pubs and fast food restaurants, for example, the amenity column would have to be indexed. To search for forests, the landuse column would have to be indexed.

  Eventually this would lead to every column being indexed, which is not how databases are designed to be used. The indexes would most likely not all fit in main memory for a database generated from the full planet.osm file.

- The table-per-feature-type scheme is not extensible enough to cater for arbitrary game objects, as well as map features. The game developer might want to define an extra five persistent attributes on each game object. If the table grows by five columns per extra game class defined, the size increase would be enormous.
- The Hibernate entities would have had to be reverse-engineered versions of the schemas created by Osmosis or osm2pgsql. Creating these entities to map perfectly to the schemas, would have constituted a significant amount of additional work which would have nullified the last of the advantages above.

Based on the above the schemas created by these tools do not fit game development needs. It is therefore more logical to create a schema specifically for gaming applications.

An OSM schema for games

A game developer is concerned with creating different types of objects. If only player objects appear and disappear and the client already has the static spatial data, why query for everything, filter by player objects and then send these only? It does not make sense to query for everything within a certain region.

Figure 4.5 shows a few classes of the OSM schema created for games. These classes are Hibernate entities and use the table-per-class inheritance scheme. This inheritance scheme works by creating a database table for each class of object. Other schemes can make different objects share a database table, by adding a discriminator column containing an object-type enum.

The concept behind this schema was to group related entities together, so they could easily be queried using Hibernate's table-per-class union feature. The table-per-class schema is very useful, because it creates no extra "column bloat" in classes which do not share each others' exact attributes and it creates small tables, containing only one class of object.

Advantages This scheme facilitates Object-Oriented coding practices. Entities can share everything defined in their super-classes; and by using this inheritance scheme a lot of boilerplate code has been eliminated.

A good example of eliminating boilerplate can be found by comparing the osm.model.POIs.PubEntity and osm.model.POIs.FastFoodEntity. Both classes inherit from a NamedPOIEntity class, which automatically sets their names and provides point access functions. The only difference between the two is in the xmlRule() function, see Listing 4.1.

4.3.5 Parser implementation

The parser component is a separate open source project, found on google code. The authors were contacted and asked about refactoring it into the Spatial Provider's namespace, which was permitted, see section B.3. Their names remain in the source code as attribution.

Some minor modifications were necessary to make the parser more fully featured: The transformer stage persists objects using their geometry, but the

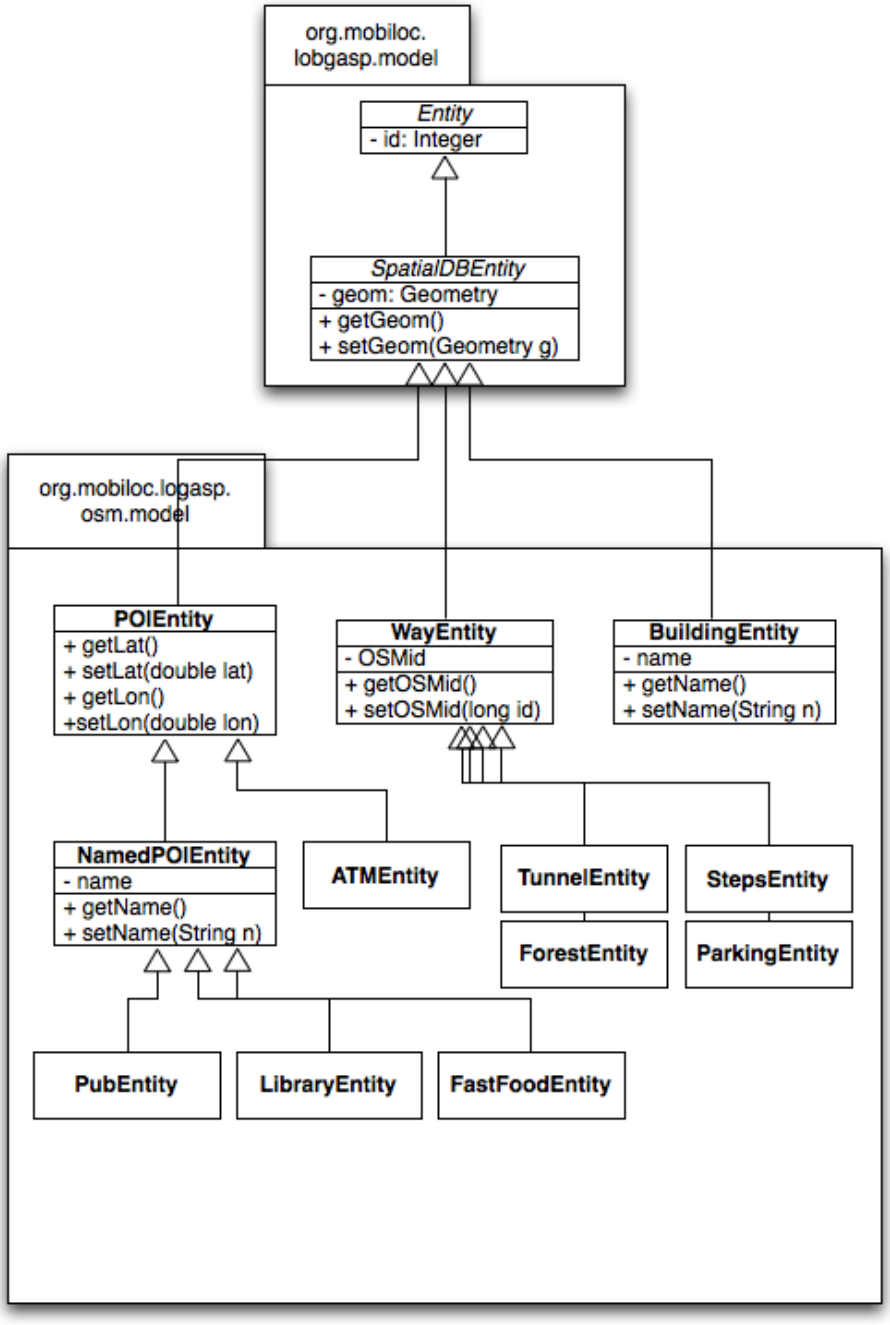


Figure 4.5: OSM Entity Hierarchy

```
PubEntity.java:
@Override
public boolean xmlRule(AbstractNode in) {
    return (in.tags.containsKey("amenity") && in.tags.get
        ("amenity").equalsIgnoreCase("pub"));
}

FastFoodEntity.java:
@Override
public boolean xmlRule(AbstractNode in) {
    return (in.tags.containsKey("amenity") && in.tags.get
        ("amenity").equalsIgnoreCase("fast_food"));
}
```

Listing 4.1: Low boilerplate classes: PubEntity.java & FastFoodEntity.java

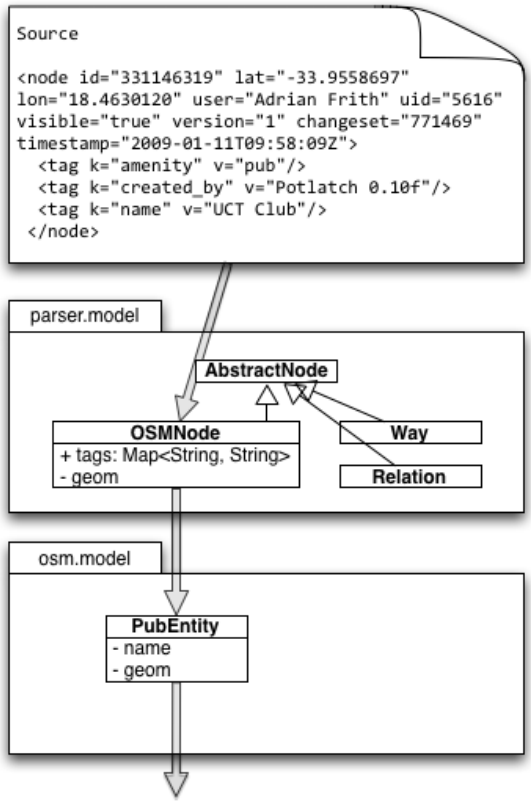


Figure 4.6: Parser step in transformer strategy

original parser nodes were only returning Well Known Text (WKT) representations of their geometry. To correct this simple getGeom() methods were added.

Figure 4.6 shows the classes that this parser provides as an additional step in the strategy model defined in Figure 4.3.

Memory use problems

This component is a DOM parser and as concluded earlier SAX parsers are much more efficient.

To illustrate the problem an attempt to parse the Western Cape (186 MB of XML) on a Macbook Air with 2 GB of RAM was made. The first attempt quickly caused an OutOfMemoryError (Heap Space). This is due to the Java Virtual Machine (JVM) assigning only 16MB of RAM to the heap of a running program. By default this heap can expand to 124 MB.

After raising the memory allocated to the JVM to 1024 MB, the parser still runs out of memory. When the maximum memory is raised to 1536 MB, the parser runs for a full 15 minutes, while the system is thrashing (swapping virtual memory pages into and out of RAM) rapidly, before the same error occurs again.

Solution A simple solution to this problem is to split the XML into smaller regions and run the Parser & Transformer component multiple times. Some other APIs, such as the Electronic Thesis and Dissertation Library API used in this year's WWW course practical, offer resumption tokens to request the next part of a large XML request. This makes client-side parsing much easier and results in the API not serving as much XML on each request.

For this particular project to work the area required was just UCT's Upper Campus. The final area used during experimentation also included Rhodes Memorial, all of Middle & Lower Campus and areas within Rondebosch. The parser was able to parse this area within the 124 MB default JVM memory limit and took an average of 4.5 seconds.

Discussion While this DOM parser is not ideal for the project, it works well for the small area required for testing. Reimplementing this as a SAX parser is one option, looking at Osmosis' code or importing Osmosis' parser is another option.

4.3.6 Transformer implementation

The transformer takes a mapping from source spatial features to game objects, then uses the parser to create a list of source features. These source features are passed through a list of the mapped game objects to see whether they are valid input for a game object. At the end the game objects found are stored in the spatial database.

Defining mappings

Mappings are defined using a number of function calls to the transformer's "register()" method specifying a source feature and an output game object. These mappings are later used to:

(a) decide whether a parser item is a spatial feature of interest or not.

(b) instantiate a matching game object.

The register method simply takes mappings and stores them in a Java HashMap. A later optimisation added some complexity to the register method by defining multiple HashMaps, one for each type of OSM data. This reduced the number of comparisons required to find a match (see section 4.3.6) by a constant factor.

Matching parser output to spatial features

Matching parser output to spatial data classes is not as trivial as simply looking up a key in a HashMap. The parser returns a pure Java representation of the XML input, see Figure 4.6, which means that a building is still returned as a Way class with a dictionary of tags containing a key-value pair of building=true.

To link objects from the OSM Entity Hierarchy to simple OSM features as returned by the parser, each of the objects implements a method called xmlRule(), which accepts an AbstractNode from the parser and returns a boolean value to indicate a match.

The rest of the matching procedure is implemented by using the xmlRule() method to compare each feature returned by the parser to the feature mappings registered. Once a match is found the loop continues to match the next feature.

Non-deterministic outcomes

This system can result in non-deterministic outcomes if the developer is not careful:

- A super-class can match a feature that was meant to be matched by the sub-class.

  > It might be logical to specify something like "all roads are 10 MBit links, all highways are 1 Gigabit links". To implement this the developer might match "WayEntity" to "10MbitLink" and "HighwayEntity" to "1GbitLink". The more general WayEntity class also matches features that should be mapped by HighwayEntity.

  To solve this issue the developer has to add exceptions to each super-class, to never match any of its childrens' matches. This renders a super-class useless as a catch-all matcher.

- Any two classes can match on the same tags. This results in the class to be compared to a feature first to dominate the results. The order of the classes is non-deterministic and related to the size of the HashMap used and the hashes of the class types.

  A possible work-around is to make the loop continue and match all possible mappings to each feature. This would create duplicates (multiple game objects) for each feature that was mapped multiple times by overlapping xmlRule() methods.

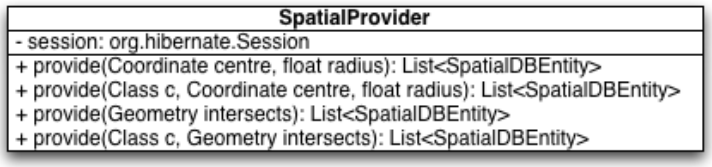


Figure 4.7: Spatial Provider Class Diagram v1

> For the demonstrational game the loop in the Spatial Provider was changed to allow multiple matches. This is used to match steps and more general ways. The ways are useful for debugging purposes, but not used in-game, which is why the "overlapping matching" is no problem.

Solution A solution not yet implemented, but considered, is to use an order-preserving datastructure to store mappings. This uses the order of register() function calls as the factor determining the order of matches.

This can still cause buggy behaviour, if not used right (by declaring a catch-all super-class before its sub-class), but would at least behave the same for any given order of register() calls.

Conclusion It is currently the developer's duty to not map any OSM Entities with overlapping xmlRule()s to different game objects. To do this the developer needs to manually compare xmlRule()s and not match a super-class and a sub-class at the same time.

4.4 Designing Spatial Provider component

The actual Spatial Provider functionality in the Spatial Provider component is extremely simple.

4.4.1 Exposed API

With the Entity inheritance structure already defined, the Spatial Provider needs to provide only a single funcion: "provide()". The class diagram in Figure 4.7 shows this function and its overloaded variants.

The functionality these functions provide is the following:

- Provide everything around the point specified within a certain radius
- Provide all objects of a specific class and its children around the point specified within a certain radius
- Provide everything within an area specified by a JTS Geometry
- Provide all objects of a specific class and its children within a JTS Geometry

```
1  public List<SpatialDBEntity> provide(Class<?extends
       SpatialDBEntity> source, Coordinate p, double radius)
       {

3      Point point = GeometryFactory.
           createPointFromInternalCoord(p, example);
       Geometry poly = point.buffer(radius);
5
       Session s = HibernateUtil.getSessionFactory().
           getCurrentSession();
7      Transaction tx = s.beginTransaction();

9      Criteria query = s.createCriteria(source);
       query.add(SpatialRestrictions.intersects("geom", poly
           ));
11     List list = query.list();

13     tx.commit();
       return list;
15 }
```

Listing 4.2: Example Spatial Provider query

Internally the "coordinate with radius" method variant constructs a circle, which it passes to the more general function which takes any geometry as input. Similarly the methods not specifying a class, simply internally call the version with a class parameter, by using SpatialDBEntity (the highest inheritance-level spatial base class) as the class parameter.

4.4.2 Implementation

Listing 4.2 shows a simple query for a specific class of objects, around a point & within a certain radius.

This function can be decomposed into a few simple steps to demonstrate the inner workings of the Spatial Provider:

1. A Point is constructed from the coordinate supplied. This means that JTS adds Spatial Reference information to the point using a sample Geometry object.

2. The constructed point is widened to a circle using the buffer() method and a radius.

3. The Hibernate Session is used to start a new Transaction.

4. A Criteria is created, using the source class as the SQL "FROM" parameter.

   Hibernate actually offers a more sophisticated "FROM" than standard SQL: Knowing the inheritance stratgy, it performs multiple SQL unions,

to combine the result of the query with the same query on all children of the source class.

5. A SpatialRestriction is added, which can be interpreted as the "WHERE" in a standard SQL query. It takes the circle we have defined in step 2 as the target area.

6. The query is executed and returns a list.

7. The transaction is committed (closed) and the result is returned.

This whole function is an extremely simple way of querying for spatial objects. It corresponds to the notion that a query for POIs in an area should return all sub-classes of POI, for example.

4.4.3 Pitfalls

Within & Intersection

During the game testing some of the largest features, such as the Table Mountain National Park, were never returned by any queries. The problem was related to the spatial predicate used to define a search query.

The spatial predicate functions defined by SFSQL can be named confusingly at times. A "within" query performs only strict withins. This means that a road that is partially in the query area and partially outside of it will not be sent to the client. We would have expected within to return all objects partially within an area.

The next predicate tested was "overlaps". This did not return anything within our query area, but instead returned only items that were partially inside the area and partially outside of it.

Solution "Intersects" turned out to be the predicate the Spatial Provider needs: It returns all objects within the query area and any objects partially outside of it. The naming is peculiar, since intersects sounds like it would only return "overlapping" elements.

Serializing database entities

Initially one of the requirements was to easily pass spatial game objects from the server to the client. Serialization is a simple process to turn Java objects into bytestreams and these streams can easily be sent over the network.

One disadvantage of the current system is that each returned object is a database entity, and each class includes *javax.persistence*. This adds additional requirements if the client is to use these classes without modification. Refer back to Figure 4.1 to inspect the complicated dependency mess of ORM-related code. Passing dependency-rich objects to the client is definitely undesirable.

Solution attempt 1 An additional Java interface was implemented to circumvent this problem. The interface is named "ToSimple" and the implementing classes need to provide a "toSimple()" method, which returns a simple plain-old-Java-object (POJO). This POJO is no database entity, but contains the same attributes as its creator.

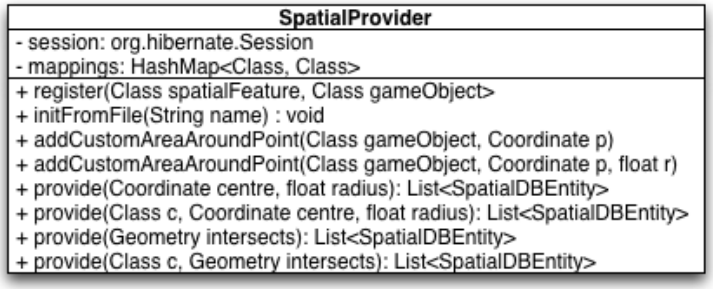


Figure 4.8: Spatial Provider Class Diagram v2

This method is highly undesirable, as it involves creating a complete copy of each class of the game objects, simply to not make them database entities.

This solution did not work correctly, as Geometry classes still contained references to Hibernate's GeometryFactory, which created them.

Solution attempt 2 A simpler solution was suggested by StackOverflow's users: After removing all annotations and references to *javax.persistence* from the Spatial Provider's models and doing all ORM Mapping in XML, the classes should again be standard POJOs, and ready for serialization.

It was additionally recommended to use Data Transfer Objects (DTOs), rather than plain serialization. Other sources describe DTOs as an "anti pattern" that should be avoided.

Final solution Serialization turned out to be unnecessary, since the Networking component defined its own spatial game objects to be effectively packed instead of serialized, using protostuff.

This solution is actually very close in spirit to the "use DTOs to send objects" solution recommended by StackOverflow's users and completely removed the need to serialize game objects directly provided by the database.

4.4.4 Final API design

Since the Spatial Provider offers only four overloaded versions of the same function to the game developer and the Transformer component requires only two methods, "register()" and "init()" the APIs for both were combined in the SpatialProvider class.

As an additional feature the developer was given the control to create extra spatial objects manually, using the *addCustomAreaAroundPoint()* method. This extended version of the Spatial Provider component is shown in Figure 4.8.

Since this changes the separate Transformer component and Spatial Provider component into a single API, the overall System Diagram got revised, see Figure 4.9. It now displays these two components as the input and output stage of the same Spatial Provider API.

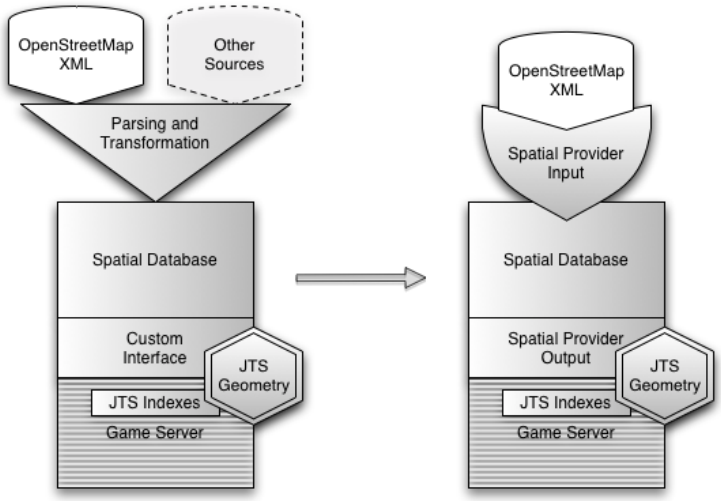


Figure 4.9: Spatial Provider component in System Diagram

4.5 Geodetic conversions & Projections

The JTS Geometry classes work in only one base unit. This means that if we specify Geometry in metres, there is no conversion functionality to yards. This also means that distance & area calculations are limited to this base unit, i.e. metres and square metres.

The GPS module's API on the phone obviously provides the game with (longitude, latitude) position updates. The base unit for these values is in degrees. Conversions between these degree values and useful units to measure distances in (metres) are non-trivial.

4.5.1 The haversine formula

One method to convert between two WGS-84 points (longitude & latitude) is to use the haversine formula. This formula can be used to find the "great-circle distance" between two points. It is based on using a perfect sphere to model the earth's shape, but is accurate "enough" for short distances.

> Accuracy: since the earth is not quite a sphere, there are small errors in using spherical geometry; the earth is actually roughly ellipsoidal (or more precisely, oblate spheroidal) with a radius varying between about 6,378km (equatorial) and 6,357km (polar), and local radius of curvature varying from 6,336km (equatorial meridian) to 6,399km (polar). 6,371 km is the generally accepted value for the Earths mean radius. This means that errors from assuming spherical geometry might be up to 0.55% crossing the equator, though generally below 0.3%, depending on latitude and direction of travel. An accuracy of better than 3m in 1km is mostly good enough for me, but if you want greater accuracy, you could use the Vincenty formula for calculating geodesic distances on ellipsoids, which gives results

```
1  double haversine(double lat1, double lng1, double lat2,
       double lng2)
   {
3          double dLat = Math.toRadians(lat2 - lat1);
       double dLon = Math.toRadians(lng2 - lng1);
5      double a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
           Math.cos(Math.toRadians(lat1)) *
           Math.cos(Math.toRadians(lat2)) * Math.sin(dLon /
               2) * Math.sin(dLon / 2);
7      return 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a))
           * 6371; // 6371 = average earth radius
   }
```

Listing 4.3: Simple Haversine formula in Java

accurate to within 1mm.

MovableTypeScripts [20]

A simple haversine formula in Java[33] is shown in Listing 4.3.

Since this formula specifies the earth's radius in kilometres, the returned distance will be in kilometres. The resulting distance is quite accurate for a spherical model of the earth. No better solution is available without using the non-sperical geoid defined in WGS-84 to reproject data.

Unfortunately it is difficult to integrate this conversion with every possible distance calculation between differing geometry types, as established by emailing the JTS maintainer, see section B.1.

4.5.2 Reprojecting the data

Another solution is to reproject the WGS-84 data into a map projection that has metres as its base unit. Since the earth is not flat, these alternative projections only work for small areas, such as countries or states.

Reprojecting the input data is complex, but not too complex to be done server-side. This would allow the Spatial Provider to do very accurate distance queries in metres.

Unfortunately, this also means that all other results would be returned in a non-WGS-84 spatial reference system, which means that the data would need to be projected back to WGS-84 to be useful on the client device (no map rendering libraries for Android would accept non-WGS-84 input).

Again, this means that if the client wishes to do its own distance tests without relying on the server, it would also need to reproject from WGS-84 to a local coordinate reference system in metres.

This solution of creating multiple reprojections in different parts of the system, client-side and server-side, is too complicated to achieve simple distance queries.

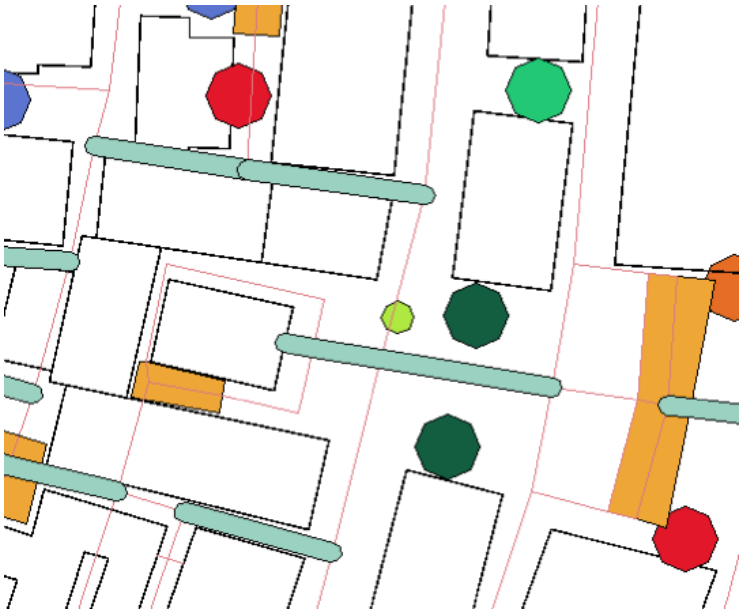


Figure 4.10: Geometry generated on campus, reduced number of vertices

4.5.3 Solution using 18th century Geodesy

This project adopted a very simple solution for the time being. To understand this solution here is a little history of Geodesy:

In 1791, the French Academy of Sciences defined $10^7 metres$ to be be equal to the distance between the equator and the North Pole (going along the Paris meridian (line of longitude - y axis), of course). If we divide this by 90° of latitude we arrive at 111111.1 metres per degree of latitude.

Degrees of longitude are a little more complex. As the degree of latitude approaches the poles, each degree of longitude becomes shorter in terms of real metres. If we use a sperical model of the earth again, a degree of longitude at the equator is exactly as long as a degree of latitude, 111111.1 metres. At the poles each degree of longitude equals 0 metres. In between, we can calculate the length of each degree of longitude by multiplying 111111.1 metres by the cosine of the latitude (in radians, unless the programming language provides a version of cosine accepting degrees).

UCT's campus lies at a latitude of roughly $-33.957°$. The cosine of this equals 0.829457009, therefore a degree of longitude at UCT is only $111111.1m * 0.829457009 = 92161.9m$ long. A degree of longitude is 17% shorter in Cape Town then a degree of latitude.

Accuracy

One test of this method shows that its worst-case result is off by 8.6 metres on a 2 000 metre distance calculation (at a latitude of 81 degrees). The results are better on all other latitudes, even when getting closer to the pole.

In all LBMGs surveyed the normal range of interaction is between 50 metres and 200 metres, with the shortest possible interactions around 10 metres - these

are limited by the accuracy of GPS receivers in mobile phones.

Since these distance calculations are one or two magnitudes shorter than the above mentioned test, the error of a distance calculation would also be one or two magnitudes smaller, i.e. less than one metre.

This solution, whilst more inaccurate than the other possible solutions, is less computationally intensive and the errors are still much smaller than the inaccuracies introduced by the GPS receivers on the client devices, which justifies this methods use for location-based gaming applications.

Result

Figure 4.10 shows some circles generated on campus. The circle generating function of JTS Geometry does not take differences in latitude and longitude into account, therefore causing circles to appear squashed along the circle of latitude (on the x axis). Using 18th century French Geodesy we can establish that the circles' radius is 17% shorter on the x axis than it should be. This was accepted as a minor enough error for our project on these small scales.

4.5.4 Projections

It has to be noted that the pseudo-circular geometry, generated using the Geodesy trick above, looks even less circular once projected into flat 2D map space using an OpenStreetMap or Google Map projection.

Google Maps and most other web-based maps use a projection invented at Google, and originally given the unofficial EPSG code 900913 (= google). The European Petroleum Survey Group managed and catalogued the EPSG Geodetic Parameter Set, a database of Earth ellipsoids, geodetic datums, geographic and projected coordinate systems, units of measurement, etc.

The EPSG body rejected Google's projection with the following statement[27]:

> We have reviewed the coordinate reference system used by Microsoft, Google, etc. and believe that it is technically flawed. We will not devalue the EPSG dataset by including such inappropriate geodesy and cartography.

The original result shown in Figure 4.10 is projected in EPSG:4326, which is the standard geographic projection for WGS-84 data.

Figure 4.11 is what actually appears on the client (and any map using Google's map projection), which we believe is more horizontally distorted. This projection was later accepted by the EPSG body and is now named EPSG:3857.

The user-generated pseudo-circular geometry is not the only geometry affected, and we believe that the buildings and other features look less desirable in EPSG:3857 in general, but this is the standard projection for online maps and the only one available on Android devices.

4.6 Conclusion

The implementation of the Spatial Provider component closely followed the design specified in the last chapter. Most chosen software components were easy to use during this implementation phase.

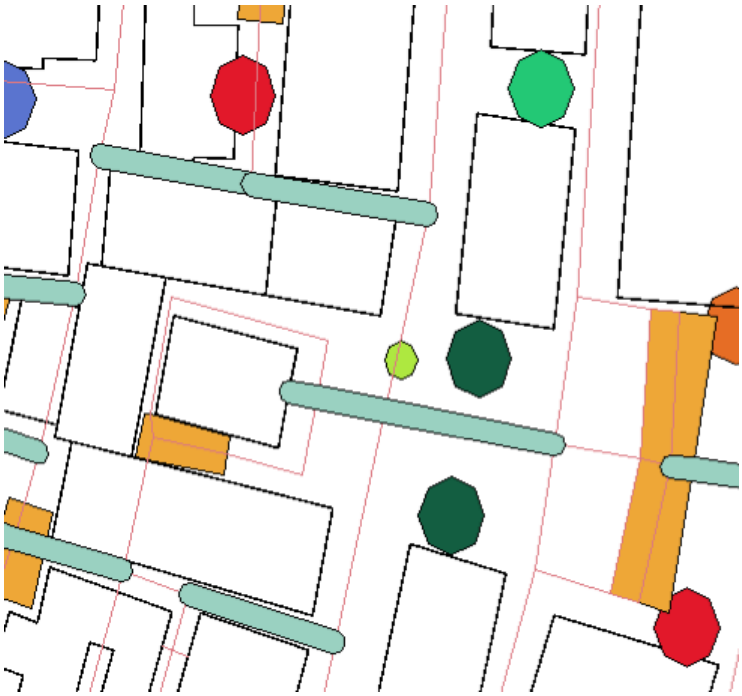


Figure 4.11: Geometry projected to Google's Pseudo-Mercator projection

Map projections and geodetic conversions require some more attention in the future. Current distance functions provide a rough estimate, with the accuracy dependant on the bearing of the distance vector. If JTS distance functions could be rewritten to return a vector decomposable into separate x and y components, the geodetic estimate described in subsection 4.5.3 could prove to be accurate enough for many game applications.

Chapter 5

Results

5.1 Introduction

To evaluate this project's success the performance of the Spatial Provider was investigated. The use of PostgreSQL is first justified using official benchmarks, then confirmed using the LBMG Lokémon.

Further metrics of success, such as the ease of use of the Spatial Provider and the usefulness of the spatial objects provided, are evaluated using Lokémon's implementation and user testing sessions.

5.2 Query performance

A database-backed system obviously behaves differently under production load than under a test load. Many blogs have published unofficial benchmarks comparing PostgreSQL performance to MySQL performance using some rudimentary benchmarks, such as the performance of a web framework. Results were usually in favour of MySQL, since PostgresSQL needs configuration optimizations to run efficiently, whereas MySQL's default settings are already optimized.

PostgreSQL's development turned more performance-centric since version 8.1 (November 2005). Version 8.4 (July 2009) is at least 7.5 times faster than version 8.0, and is 10 times faster on a pure read load.

5.2.1 Official benchmarks

In 2007 SUN and the Spec performance publication organization published a peer-reviewed official PostgreSQL benchmark, *813.73 SPECjAppServer2004 JOPS@Standard*, which beat MySQL's performance, which is the only other open source relational database that was contending PostgreSQL. PostgreSQL performed especially well on multi-core production servers, where MySQL's performance increased by only 37% when scaling from 1 to 4 cores, but PostgreSQL's performance increased by 226%.

PostgreSQL additionally beat Oracle server in terms of the total cost over performance. Where Oracle outperformed PostgreSQL by less than 15%, the Oracle licenses made the system almost 200% more expensive.

These benchmarks validate PostgreSQL as a powerful open source database choice.

5.2.2 Lokémon performance

The Lokemon "proof of concept" game was used to provide some "real world use" benchmarks. This test had multiple obvious shortcomings:

1. The maximum number of concurrent users of this system was 4 users, since that was the number of phones this project had available for testing.

   Any real-world system would aim to serve player numbers in the hundreds per database server. Testing on this scale was infeasible using this project's resources.

2. The system was tested running on a Macbook Air and an EC2 Micro instance.

   Both, a low-powered laptop and Amazon's smallest cloud instance, are the lowest possible hardware configuration this system could have been tested on. A production server would have much more than 2 GB or 512 MB of RAM and test results would look differently.

3. The database used was very small. These queries were run on a database containing only 1.5 km x 1.2 km of data.

   A real world server would obviously allow the players to play in a much larger area than just on UCT's campus.

4. The game prototype simplified the problem, and requested the whole dataset, rather than requesting incremental updates.

   Due to time and code complexity limits, the game queried and stored the complete set of spatial data on the client during each (client startup) request, and did not query for more data as players moved around.

   This results in larger queries, but a smaller number of queries.

5. The PostgreSQL & PostGIS setup is "out of the box", i.e. unmodified. Given more time the author would have probably been able to improve on the query time results.

The server logs, provide some useful insight, none the less. Using the limited data at our disposal, the following performance traits can be observed:

- Figure 5.1 shows that a large number of queries was faster than 80 milliseconds.

  During session 4 the query provider was clearly not performing at its best, possibly due to "background" server load.

- The first query after a server restart is much slower than the remaining queries. This is shown in the *ST* column in Table 5.1.

  A possible explanation for this is the Hibernate session creation overhead. The game server reuses a Hibernate session once it has been established. The process of first creating a session clearly takes about 2270 milliseconds and is followed by a normal 100 millisecond query.

Session	Date	ST	QT	s	N
1	05 Oct	2494	97	32.1	29
2	06 Oct	2334	100	36.6	35
3	07 Oct	2308	92	26.8	16
4	11 Oct	2302	104	30.4	12
Average:		2359	98	31.5	23

Table 5.1: Server logs from the most recent gaming sessions

Legend:

ST	Startup Time (ms)
QT	Average Query Time (ms)
s	Sample Standard Deviation
N	Total Number of Queries

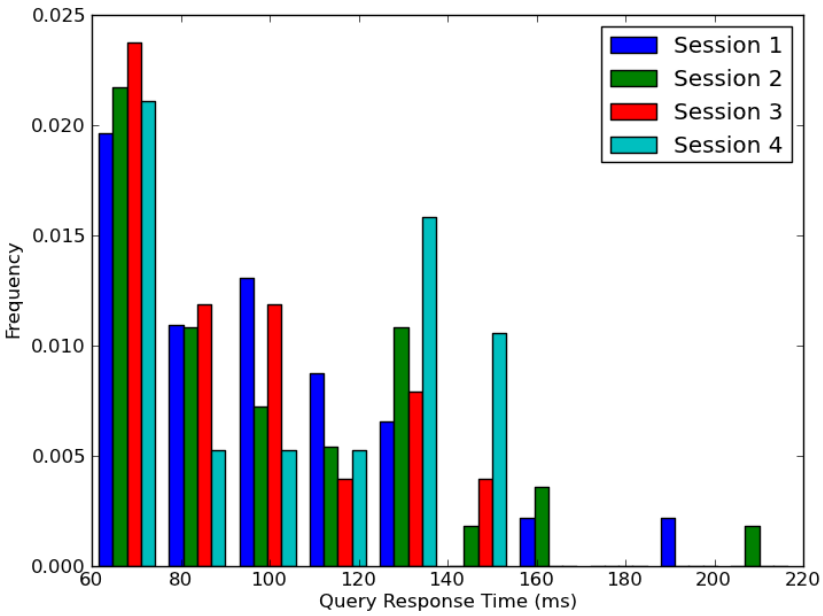


Figure 5.1: Histogram of Spatial Query Time

This histogram has been normalized to compare sessions as if they had the same number of queries. The first query of each session was removed, since it was a consistent outlier.

- Almost all queries return 88 spatial objects, since they take place during player-creation. They are all querying around the same location and using the same radius. They take on average 100 ms to complete.

  A very small number of queries (2 queries) clearly exploit a bug in the game logic and might be fired when the player successfully reaches the edge of the game world: They return only 12 spatial objects and complete in 9 ms and 10 ms.

  This discovery leads us to assume that smaller queries complete much more quickly, which does seem intuitively correct.

- Some sessions seem to perform much better than the others, with average queries being up to 2.5 times faster during the "best" session, compared to the slowest session.

  This might be due to the nature of EC2 Micro instances, which share CPU cores between instances and might be under load from another instance during the testing.

5.3 Implementation Case Study: Lokémon

This component has not been evaluated in a real game other than Lokémon, which was produced by this project team. Therefore not many details on general implementation issues are available yet. Instead this thesis will describe implementation from the Lokémon developers' perspective.

5.3.1 User testing & feedback

The evaluation of the system as a whole, including the Spatial Provider, Networking component and Client-side Game Engine, involved the use of a Flow State Scale (FSS) questionnaire during a number of evaluation sessions. The same FSS questionnaire was administered to a control group and a location-based group of players.

The location-based version of the game was the standard full-featured LBMB, using complex spatial features and allowing real-time interactions between players. The control group's version was limited by disabling the GPS in the game, and activating movement via gestures. The other gameplay mechanics remained exactly the same.

The user feedback extracted from the game testing session was overwhelmingly positive. Some users complained about wanting more spots to heal their monsters, but everyone enjoyed the game. Many inquired about getting the game for their own phones after the testing was done, which is a clear indicator of the game being very enjoyable.

The FSS questionnaire showed a slight increase in the "fun factor" metric in the location-based group over the control group. This increase was not statistically significant, due to the very small sample sizes. This test did not contrast any features of the Spatial Provider between the standard group and the control group.

It was used to collect the GPS usage and query performance metrics for the statistics in the rest of this chapter, however.

Figure 5.2: Lokémon Screenshots

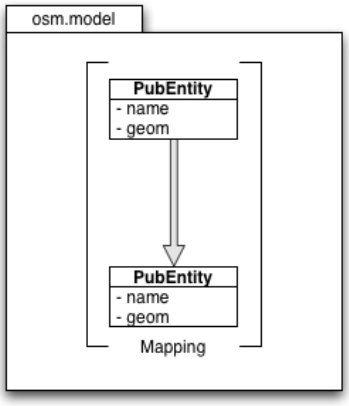


Figure 5.3: Simple Mapping using only features

5.3.2 Mapping of Game Objects

Lokémon does not use the Spatial Provider quite as intended. Instead, it never defines custom GameObjects and simply queries for spatial features. This was anticipated[1] and supported on purpose by providing mappings directly from a feature to itself. Figure 5.3 illustrates a mapping created from a feature to itself.

Using this style of mapping is convenient for developers, who wish to only query for spatial features, and not define persistent game objects to be used instead of the standard features.

Lokémon defines the following set of spatial objects:

Cave Mapped to real-world tunnels.

Forest Mapped to real-world forests.

Grassland Mapped to real-world sports fields.

Mountain Mapped to real-world Nature Reserves.

Rough terrain Mapped to real-world steps.

Urban Mapped to parking lots.

"Water's edge" Mapped to reservoirs and fountains.

Pokémon Centre Mapped to fast food restaurants and pubs.

Pok'eMart Mapped to automatic teller machines (ATMs).

The specification written by the game developer wanted these spatial objects to be defined as a single class, discriminating between area type using a Java "enum" construct.

While this could have been implemented using the Spatial Provider's *register()* method, it was implemented using a *switch statement* in the game server

[1]Some developers might not want to map from a street to a game object, they just need street geometry!

itself. The *switch statement* is used to compare the objects returned by the Spatial Provider and proceeds to create the simple object with the correct enum set as specified by the game developer.

It would have been more beneficial (for this evaluation) to use the built-in functionality of the Spatial Provider, but a developer of open source software cannot control how their software gets used by other developers. Therefore this example implementation is a likely real-world use-case scenario.

5.3.3 OSM XML Input

The input is a small 1.5 km x 1.2 km extract retrieved directly from the OpenStreetMap website's export function. A few zones were manually placed on campus, these are all pseudo-circular. The final result can be inspected in Figure 5.4.

The Spatial Provider created 88 spatial objects to be used as game objects. The density is quite high on campus with 3 to 7 objects within a 100 metre radius of any point.

While using a larger map was considered, it was not necessary for testing this proof of concept game. The parser & transformer input stage was separately evaluated, using larger maps instead.

5.3.4 Spatial Queries

The game used a very simple scheme to query, and always retrieved the same 88 spatial objects during client startup time. This is unfortunate, since it means that the only query data collected comes from players starting or restarting the game. Luckily this still occured around 30 times in the first two sessions and about 10 times in the final two sessions.

The performance of returning 88 objects in roughly 100 milliseconds is not too slow, if compared to the performance of the official SimpleGeo API in Figure 3.3, which took over three seconds to return a smaller number and less complex features.

A direct comparison with SimpleGeo is not valid of course, since the SimpleGeo data set contains the whole planet, but their query performance can be used to establish that 100ms is "fast enough".

Ease of implementation

The Spatial Provider was implemented with the Networking component within a single afternoon. Most of the time was spent installing PostgreSQL and PostGIS, and bundling the Spatial Provider into a single convenient *.jar* file.

The actual implementation of the spatial queries from within the game server was completed in less than 50 lines of code, much of which is the custom conversion to simple game objects as requested by the game developer.

An implementation step completed in a single afternoon is satisfactorily short, and affirms the completion of the "easy to use by game developers" requirement. To properly confirm this a study using more games and game developers has to be undertaken, of course.

Figure 5.4: Campus transformed to Lokémon World

The indoor pub and food court were removed to not encourage players to wander into buildings and lose their GPS signal. The dark orange shop entities were never used in-game, but were provided as specified. Two small blue mountain entities were placed closer to the Computer Science building. Additional dark green grassland was created next to Jammie steps and a small light green forest was placed below the Computer Science building.

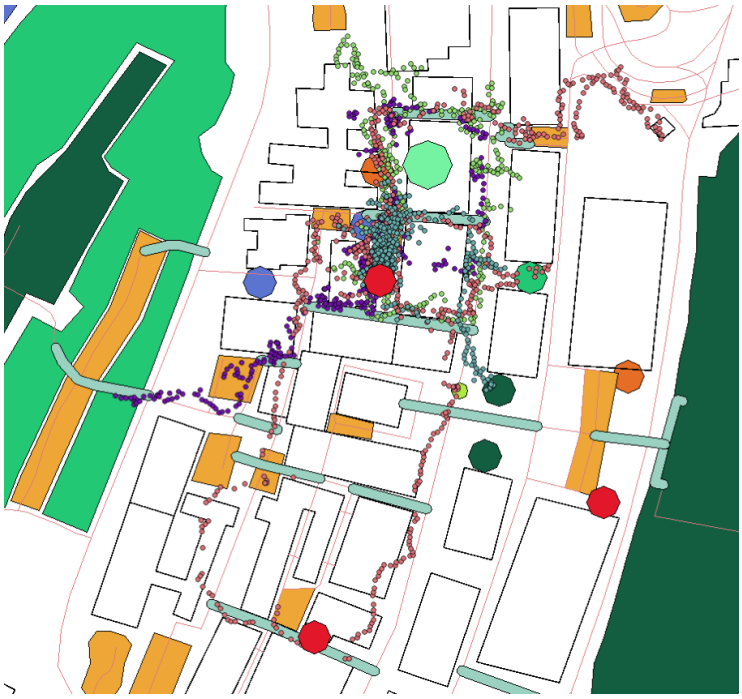


Figure 5.5: Players' use of Lokémon game world

Legend:

Phone	Colour	Distance $(m)^2$
1	green	2 146
2	purple	1 734
3	pink	2 609
4	blue	1 771

5.3.5 Use of Spatial Game Objects

The actual use of real world spatial features by players was established by plotting the GPS positions of four players during the location-based game evaluation. This data could be used to build an understanding of how far players are willing to move in the real world, which can be used by developers of future games to adjust the amount of movement their games require.

The data in Figure 5.5 shows the routes the players of the location-based version of Lokémon walked in the real world in one hour to explore the game world.

A large clump of activity can be seen under the red "food court" map marker. This is perfectly explainable using the mechanics of Pokémon style games, in which players need to return to PokéCenters regularly to heal. If more of these PokéCenters had been created at regular intervals, players might have travelled further.

[2]To arrive at distance metrics the GPS traces were reprojected to Hartebeesthoek94 / LO19 in QGIS, exported to shape files, imported as vector layers and finally measured using QGIS' Calculator. It is not possible to measure non-reprojected GPS traces.

Three of the four players wandered far from the computer science building and chose opposite directions to do so. Player three exhibited the most enthusiasm by walking over 2.5 kilometres during the one hour session. It is interesting to note that even the least enthusiastic players were still willing to walk over 1.5 kilometres.

This data is useful to developers of future LBMGs who have no data to support their decisions of how far apart to space spatial game objects.

5.4 Conclusion

A single game developer using a component is obviously no good indicator for a software component's ease of use. The evaluation game could have also made better use of the APIs provided by this component or tested the performance on a larger scale.

The GPS traces from the actual Lokémon test session and the user feedback suggest that all players really enjoyed the game. A study comparing larger samples of users to a "less location-based" control group, would have been more insightful.

This system was used to build an enjoyable, first-of-its-kind game, which is possibly the most important success metric.

Chapter 6

Conclusion

The use of complex spatial features in LBMGs is still in its infancy, since no large scale game has opted to use features other than POIs and many games are still using only player positions and no features at all. Once the first commercial game builds interactions between players and complex features it might be novel enough to change the way all future LBMG's players think about the link between games and the real world.

> During the course of this project another LBMG was released claiming to be location-based. "Geomon" uses exactly the same Pokémon inspired gameplay as this project's Lokémon.
>
> The author was disappointed to see that Geomon's location-based gameplay is limited to having the game world influenced by the time of day and weather at the players location.
>
> Geomon does not leverage the player's position to move an in-game player or provide any spatial features around the player. It does not even use POIs, such as Foursquare or MyTown to provide any kind of interaction with the real world.

It is fascinating to watch games claim to be "location-based" even if they are not. This is an exciting indicator of the potential presented by location-based gameplay.

This report has outlined one possible implementation of a system built to leverage complex features. The system implemented uses only open source software and is ready for production use on a small to medium scale.

The game created to evaluate the complete platform was received well by its players. It did not yet show its full fun potential, since it was tested by a rather small group of players. More players would have created more chance for interactions and more friendly competition.

The system has been fully open-sourced for future LBMG developers to find and use. It will hopefully inspire a great number of future game developers to build unique interactions between virtual game objects and the real world. This author plans on maintaining the platform even after the end of this degree programme.

6.1 Future work

The platform as a whole still needs to be tested extensively before it is ready to be used by a large game developer. The scalability of the Network component and Spatial Provider is largely untested. Once scalability metrics have been established, this platform could be used to serve a global audience of players.

The Spatial Provider component specifically could benefit from a different mechanisms to register mappings, which does not allow developers from registering ambiguous xml-matching rules. If developers are aware of this issue, it is easily circumvented and does not pose a major problem.

To build a more scalable input stage for the Spatial Provider, using a SAX parser rather than a DOM parser, a future project might inspect the source code of the Osmosis tool.

Bibliography

[1] P.K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '00*, 66:175–186, 2000.

[2] AListApart. A List Apart: Articles: Take Control of Your Maps, 2008. URL `http://www.alistapart.com/articles/takecontrolofyourmaps`.

[3] S. Büttner, H. Cramer, M. Rost, and N. Belloni. ϕ: Exploring physical Check-Ins for Location-Based Services. In *Ubicomp '10 Proceedings of the 12th ACM international conference adjunct papers on Ubiquitous computing*, pages 395–396, 2010.

[4] A. De Souza e Silva and L. Hjorth. Playful Urban Spaces: A Historical Approach to Mobile Games. *Simulation & Gaming*, 40(5):602–625, April 2009. ISSN 1046-8781.

[5] DirectionsMagazine. NoSQL Databases: What Geospatial Users Need to Know - Directions Magazine, 2011. URL `http://www.directionsmag.com/articles/nosql-databases-what-geospatial-users-need-to-know/164635`.

[6] M.R. Ebling and R. Cáceres. Gaming and Augmented Reality Come to Location-Based Services. *IEEE Pervasive Computing*, 9(1):5–6, 2010.

[7] M. Erwig, R.H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999. ISSN 1384-6175.

[8] J. Falk, P. Ljungstrand, S. Björk, and R. Hansson. Pirates: proximity-triggered interaction in a multi-player game. In *CHI'01 extended abstracts on Human factors in computing systems*, pages 119–120. ACM, 2001. ISBN 1581133405.

[9] B. Griglak and C. Caiseda. Evaluating query performance on object-relational spatial databases. *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 489–492, 2009.

[10] J. Haist, R. Schnuck, and T. Reitz. Usage of persistence framework technologies for 3D geodata servers. In *AGILE 2006: 9th AGILE International Conference on Geographic Information Science*, pages 43–50. AGILE, 2006.

[11] M. Haklay and P. Weber. OpenStreetMap: User-Generated Street Maps. *IEEE Pervasive Computing*, 7(4):12–18, October 2008.

[12] P. Klante, J. Krösche, D. Ratt, and S. Boll. First-year students' paper chase: a mobile location-aware multimedia game. In *Proceedings of the 12th annual ACM international conference on Multimedia*, pages 934–935. ACM, 2004. ISBN 1581138938.

[13] E. Klien. A Rule-Based Strategy for the Semantic Annotation of Geodata. *Transactions in GIS*, 11(3):437–452, June 2007. ISSN 1361-1682.

[14] J. Koolwaaij, M. Wibbels, S. Böhm, and M. Luther. Living virtual history. *The Visual Computer*, 25(12):1055–1062, August 2009. ISSN 0178-2789.

[15] A. Küpper. *Location based Services: Fundamentals and Operation*. Wiley Press, 2005.

[16] N. Li and G. Chen. Sharing location in online social networks. *IEEE Network*, 24(5):20–25, September 2010. ISSN 0890-8044.

[17] J. Lonthoff and E. Ortner. Mobile location-based gaming as driver for location-based services (LBS) Exemplified by mobile hunters. *Special Issue: e-Society*, 31(2):183, 2007.

[18] C. Magerkurth, A.D. Cheok, R.L. Mandryk, and T. Nilsen. Pervasive games: bringing computer entertainment back to the real world. *Computers in Entertainment*, 3(3):4, July 2005. ISSN 15443574.

[19] P. Mooney and P. Corcoran. Annotating Spatial Features in OpenStreetMap. In *Proceedings of the GISRUK 2011 Portsmouth, England*, 2011.

[20] MovableTypeScripts. Calculate distance and bearing between two Latitude/Longitude points using Haversine formula in JavaScript, 2011. URL `http://www.movable-type.co.uk/scripts/latlong.html#ellipsoid`.

[21] J. Myllymaki and J. Kaufman. *High-performance spatial indexing for location-based services*. ACM Press, New York, New York, USA, May 2003. ISBN 1581136803.

[22] OpenGeo. Spatial Database Tips and Tricks : Introduction, 2011. URL `http://workshops.opengeo.org/postgis-spatialdbtips/introduction.html`.

[23] M. Over, A. Schilling, S. Neubauer, and A. Zipf. Generating web-based 3D City Models from OpenStreetMap: The current situation in Germany. *Computers, Environment and Urban Systems*, 34(6):496–507, June 2010. ISSN 01989715.

[24] O. Rashid, I. Mullins, P. Coulton, and R. Edwards. Extending cyberspace: location based games using cellular phones. *Computers in Entertainment (CIE)*, 4(1):4–es, 2006. ISSN 1544-3574.

[25] ReadWriteWeb. Video: How SimpleGeo Built a Scalable Geospatial Database with Apache Cassandra, 2011. URL `http://www.readwriteweb.com/cloud/2011/02/video-simplegeo-cassandra.php`.

[26] B Schlatter and A Hurd. Geocaching: 21st-Century Hide-and-Seek. *Journal of Physical Education, Recreation & Dance*, 76(7):28–32, 2005.

[27] SharpGIS. Spherical/Web Mercator: EPSG code 3785, 2008. URL `http://www.sharpgis.net/post/2008/05/SphericalWeb-Mercator-EPSG-code-3785.aspx`.

[28] P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings 13th International Conference on Data Engineering*, pages 422–432. IEEE Comput. Soc. Press, 1997. ISBN 0-8186-7807-0.

[29] O. Sotamaa. All the worlds a botfighter stage: Notes on location-based multi-user gaming. In *CGDC Conference Proceedings*, pages 35–45, 2002.

[30] C.T. Tan. Augmented Reality Games: A Review. *Proceedings of GAMEON-ARABIA, EUROSIS*, 2010.

[31] TechCrunch. Booyah's MyTown Hits 3.1 Million Users, 2010. URL `http://techcrunch.com/2010/08/17/booyahs-mytown-hits-3-1-million-users/`.

[32] S.J. Vaughan-Nichols. Augmented Reality: No Longer a Novelty? *IEEE Computer*, 42(12):19–22, December 2009. ISSN 0018-9162.

[33] Wikipedia. Haversine formula, 2011. URL `http://en.wikipedia.org/wiki/Haversine_formula`.

[34] D. Zielstra and A. Zipf. A comparative study of proprietary geodata and volunteered geographic information for germany. In *AGILE 2010: 13th AGILE International Conference on Geographic Information Science*, volume 1. Springer Verlag, Guimaraes, Portugal, 2010.

[35] S. Zlatanova and E. Verbree. Technological developments within 3D location-based services. In *International Symposium and Exhibition on Geoinformation*, pages 13–14. Citeseer, 2003. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.2634&rep=rep1&type=pdf`.

Appendix A

Sample API responses

A.1 Sample Google Places API response

```
{
  "status": "OK",
  "result": {
    "name": "Google Sydney",
    "vicinity": "Pirrama Road, Pyrmont",
    "types": [ "establishment" ],
    "formatted_phone_number": "(02) 9374 4000",
    "formatted_address": "5/48 Pirrama Road, Pyrmont NSW,
        Australia",
    "address_components": [ {
      "long_name": "48",
      "short_name": "48",
      "types": [ "street_number" ]
    }, {
      "long_name": "Pirrama Road",
      "short_name": "Pirrama Road",
      "types": [ "route" ]
    },
      ...Cut administrative level & postal code for
         brevity
      ],
    "geometry": {
      "location": {
        "lat": -33.8669710,
        "lng": 151.1958750
      }
    },
    "rating": 4.5,
    "url": "http://maps.google.com/maps/place?cid
        =10281119596374313554",
    "icon": "http://maps.gstatic.com/mapfiles/place_api/
        icons/generic_business-71.png",
```

```
      "reference": "A LONG HASH, removed for formatting
          reasons",
      "id": "4f89212bf76dde31f092cfc14d7506555d85b5c7"
    },
    "html_attributions": [ ]
}
```

A.2 Sample SimpleGeo Places API response

```
{
    "geometry":{
        "type":"Point",
        "coordinates":[-122.4063,37.772614]
    },
    "type":"Feature",
    "id":"afcfa085-7ceb-4c28-a367-aabd2d1e1618",
    "properties":{
        "address":"41 Decatur St",
        "attribution":"Factual Inc.",
        "categories":[
            "Real Estate & Home Improvement",
            "Contractors"
        ],
        "closed":false,
        "country":"US",
        "license":"http://www.factual.com/tos",
        "locality":"San Francisco",
        "name":"Architectural Builders",
        "phone":"+1 415-558-9396",
        "postcode":"94103",
        "region":"CA"
    }
}
```

A.3 Sample OpenStreetMap API response

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
 <bounds minlat="-33.9559340" minlon="18.4629540" maxlat=
     "-33.9558400" maxlon="18.4630720"/>
 <node id="331146319" lat="-33.9558697" lon="18.4630120"
     user="Adrian Frith" uid="5616" visible="true" version
     ="1" changeset="771469" timestamp="2009-01-11
     T09:58:09Z">
  <tag k="amenity" v="pub"/>
  <tag k="created_by" v="Potlatch 0.10f"/>
```

```
    <tag k="name" v="UCT Club" />
  </node>
</osm>
```

Appendix B

Email conversations

B.1 Extending JTS Geometry

```
from    Martin Davis xxxxxxxxxx@xxxxxx.net via lists.
    sourceforge.net
to
cc      jts-topo-suite-user@lists.sourceforge.net
date    9 September 2011 02:32
subject Re: [Jts-topo-suite-user] distance in meters

Well, yes, if you're using the more general distance
    functionality in JTS then injecting Haversine into
    that is going to be quite tricky.

Someday hopefully JTS will support geodetic operations in
     a convenient, general way...


On 9/8/2011 5:18 PM, Rainer Dreyer wrote:
Hi Martin

thanks for the quick reply! I was reluctant to use one of
     the Haversine's I found because the geom.distance()
    function works between more complex objects and
    selects the closest points for me. Having to wrap the
    Haversine in multiple overloaded versions to do
    different geometries is a bit painful.

I will test importing geotools with my team mates working
     on the Android side, that might be the simplest
    solution to use the JTS Geom's I'm getting from
    Hibernate Spatial.

Thanks again!
Rainer
```

```
21
   On 9 September 2011 02:11, Martin Davis wrote:
23 You're right that if your coordinates are in WGS84, then
       the JTS distance functions return answers in degrees.
        For small distances you should be able to convert
       this to metres and get an approximately correct value.

25 When doing the conversion you need to account for the
       fact that the length of a degree varies depending on
       the latitude and the azimuth (angle). This is why you
        probably can't just multiply by the earth radius -
       that only works if you are at the equator. Thus the
       need for a more sophisticated formula such as
       Haversine.

27 If I were you I would just grab one of the numerous
       implementations around on the web and use it.

29
   On 9/8/2011 5:01 PM, Rainer Dreyer wrote:
31 Hi everyone

33 I'm new to this list, i'm hoping that copy-pasting the
       topic title will append to the right thread.

35 Sorry, I'm still very new to using JTS and GIS concepts
       in general and am having some fundamental concept
       issues:

37 As far as I could gather from somewhere on StackOverflow,
        if I am using standard WGS-84 GPS coordinates in JTS,
        the geom.distance(geom) function returns central
       angle degrees. I thought getting "accurate enough"
       distances in meters would simply involve converting
       the distance to radians and multiplying by the earth's
        radius. This semi-works, but my answers are off by 20
        metres on a 200m distance.

39 What am I missing? Why do I need haversine's formula (or
       something more accurate)?

41 PS: I could just use GeoTools to convert, but I'm trying
       to make a reusable (by team mates) component that
       should work on Android, so I'm trying to keep the
       dependencies as low as possible.

43 Thanks!
   Rainer
```

B.2 Extracting spatial data from MapsForge's binary

```
from xxxxxxxxxx@inf.fu-berlin.de muehlberg@inf.fu-berlin.de
reply-to mapsforge-dev@googlegroups.com
to mapsforge-dev@googlegroups.com
date 17 September 2011 14:05
subject Re: [mapsforge-dev] Quick hello from the University of Cape Town

Hello Rainer,

sorry for the late answers. I am not sure if i understand your
requirements. The map library is only for rendering map tiles
(rectangular parts of the map) from OSM data. To do that we compact all
necessary data into a binary file (see:
https://code.google.com/p/mapsforge/wiki/SpecificationBinaryMapFile).

However, this binary format is not designed to serve as a general
purpose OSM data container (like PBF). Any information not needed for
map rendering is not stored in the file, for example the original OSM
ID. Polygons get clipped and some way points removed. The format is also
not designed to do nearest neighbor queries or POI searching.

Starting with the next release it will be possible to adjust the map
rendering via XML files (see:
https://code.google.com/p/mapsforge/issues/detail?id=31). So you can
define if and how an "amenity=pub" node (or way) will be displayed. If
you have additional data that you want to display, we have an overlay
API for that (see: https://code.google.com/p/mapsforge/wiki/OverlayAPI),
which can also handle user interaction.

Does that answer your question?

```

```
Greetings,
Thilo


On 05/09/11 01:43, Rainer Dreyer wrote:
> Hi guys
>
> I'm currently doing my Honours in Computer Science at the University of
> Cape Town and am working on a Location-Based Multiplayer Gaming platform
> for Android. I'm working on writing the OpenStreetMap-related
> (server-side) code, one of my team mates is working on networking code
> (client-server, maybe even between phones (bluetooth)) and the other one
> is using location on the device and is our chief game designer (and will
> be happy to use mapsforge).
>
> Once we have finished writing the platform code we want to make a
> Location-Based Pokemon clone. I'm currently parsing a subset of OSM XML,
> saving it in PostGIS and provide an API for the game server to query
> gameobjects in a certain location (area of interest filtering). (Shops
> on campus will be transformed to shops in-game, parking lot areas can
> turn into "tall grass" == standard location for pokemon catching, etc.)
>
> Does your renderer make much of the underlying OSM data available on the
> device? Ie. could I transform an {amenity: pub} into an in-game shop on
> the device and do a distance query to the player, if I wanted to?
>
> Danke!
> Rainer
```

B.3 Using Java OSM parser

```
from    Mrcio Aguiar Ribeiro XXXXXXX@gmail.com
to      Rainer Dreyer
```

```
cc       willy tiengo
date     8 July 2011 22:00
subject Re: java osm parser

Hi Rainer,

be free to modify the parser in anyway you want.
    Unfortunately, we no
longer maintain it, although we still use it. It still
    have plenty
room for improvement.

Cheers.

On Thu, Jul 7, 2011 at 7:29 PM, Rainer Dreyer wrote:
> Hi!
> I just wanted to thank you for the OSM parser you're
    sharing on google code.
> I'll use your code as part of my honours project (
    making a server for
> location based mobile games - with game objects made
    from real OSM nodes). I
> might refactor the namespace, and put the parser under
    my util folder, I
> hope that's ok.
> Thanks!
>
```