Location Based Services on mobile devices using 3D images

Omar Choudary

University “Politehnica” of Bucharest

Computer Science department

Group 354 C3

June 19th 2008
Contents

Introduction ................................................................................................................................................. 5

State of the art .............................................................................................................................................. 6

LBS .............................................................................................................................................................. 6

GPS ............................................................................................................................................................... 6

GSM .............................................................................................................................................................. 8

Wireless Networks ....................................................................................................................................... 10

TDOA ........................................................................................................................................................... 10

Photo Localization ....................................................................................................................................... 10

Hardware ....................................................................................................................................................... 11

iPhone .......................................................................................................................................................... 11

BlackBerry Bold .......................................................................................................................................... 11

HTC Diamond ............................................................................................................................................. 12

Nokia N95 8GB .......................................................................................................................................... 13

Software ....................................................................................................................................................... 14

Symbian Operating System .......................................................................................................................... 15

Project Specifications ................................................................................................................................. 16

Mobility ........................................................................................................................................................ 17

3D interface ................................................................................................................................................ 18

Accuracy ...................................................................................................................................................... 19

Project Plan. System architecture ............................................................................................................... 19

Project timetable .......................................................................................................................................... 19

Application Overview .................................................................................................................................. 21

Application UI ............................................................................................................................................ 21
Network Engine ................................................................. 22
Application View .......................................................... 23
Location Map ................................................................. 24
Development Platform ...................................................... 24
Symbian C++ ................................................................. 24
S60 FP1 ....................................................................... 27
Carbide C++ ................................................................. 28
Design Patterns ............................................................. 28
Model-View-Controller .................................................... 28
Factory Pattern ............................................................ 29
Second Constructor (Symbian C++) .................................... 29
Implementation ............................................................. 30
Active Objects ............................................................... 30
Network Engine ............................................................. 32
Logger .......................................................................... 33
GPS (LBSInfo) ............................................................. 35
WLAN (WLANInfo) ...................................................... 36
GSM (GSMInfo) .......................................................... 37
Location Map ............................................................... 38
Stadium model ............................................................. 41
Conversion between GPS and 3D model coordinates ............. 44
Stadium Views ............................................................. 46
Conversion between 3D and 2D coordinates ......................... 47
Application UI.................................................................49
Deployment and Use Case................................................49
Signing and building .........................................................49
Installation ............................................................................50
Use Case ............................................................................50
Display Information..........................................................51
Mobile guide commands ..................................................51
Test and evaluation............................................................54
GPS coordinate test ..........................................................54
Logger test ..........................................................................56
Coordinate to Image coordinate test.................................57
Mathematic Testing............................................................57
Visual Testing .....................................................................58
Conclusion ..........................................................................59
References .........................................................................60
Introduction

The scope of this project was to use most of the information available to a mobile device (Wireless LANs, GPS data, GSM network) in order to locate it with the higher accuracy possible and to express this information in a useful way.

The term LBS (Location Based Services) refers to the collection of services and applications used to locate a user in a specific geographical place. This is one of the latest subjects in Computer Science and it has been a work very complex, interesting and full of experiences.

This project was done in collaboration between the “Institute National Polytechnique de Toulouse” and Immersive-Solutions, a company specialized in 3D modeling of real venues (e.g. football stadiums).

In order to make the project useful, we adapted it to the needs of Immersive-solutions and we used 3D stadium images to show the results of LBS to the user.

At the beginning the work has been focused on the state of the art and finding the right hardware and software to implement the application. The work continued in creating a logger that could retrieve all possible information about WLAN, GSM and GPS. After that was done, we tried to bring 3D content (images) to the application in order to improve the user experience. This was a bigger challenge than what it was supposed because there are many transformations and computations involved, from converting between GPS (WGS84) and 3D coordinates to compute best path in a stadium map. At the end the work was concentrated in putting all the pieces together and making some real tests in the stadium to see how the application actually works.

At the end of the project we have also written a short article that has been submitted to the ACM Multimedia 2008 international conference. This paper is attached as an appendix. In the paper we propose new methods for the visualization of position and environment on mobile guide applications.

The result of this project is an application running on a mobile device (Nokia N95) that uses GPS to guide a client (person that buys a ticket to a football match) from the entrance of the stadium to his exact seat.
State of the art

In this section we present the state of the art in LBS, current hardware for mobile devices and the current operating systems and development platforms for these devices.

LBS

As we mentioned in the introduction, Location Based Services (LBS) is a highly discussed topic nowadays as many applications such as mobile guides (ex. Google Maps or Nokia Maps) have become very popular.

LBS can be thought as a set of services that provide location information to an application. Usually this information is expressed in geodetic coordinates using the WGS84 datum (explained later). In the next paragraphs we present some of the possible and usual technologies used in LBS.

GPS

GPS is an US implementation of the Global Navigation Satellite System (GNSS). It is the only fully functional implementation at this time. Other projects are the GLONASS (incomplete as 2008) and the European Galileo that will be available on 2013. This is the main technology used in this project in order to get location information.

Method of Operation

A typical GPS receiver calculates its position using the signals from four or more GPS satellites. Four satellites are needed since the process needs a very accurate local time, more accurate than any normal clock can provide, so the receiver internally solves for time as well as position. In other words, the receiver uses four measurements to solve for 4 variables - x, y, z, and t. These values are then turned into more user-friendly forms, such as latitude/longitude or location on a map and then displayed to the user.

Although four satellites are required for normal operation, fewer may be needed in some special cases. For example, if one variable is already known (for example, a sea-going ship knows its altitude is 0), a receiver can determine its position using only three satellites. Also, in practice, receivers use additional clues (Doppler shift of satellite signals, last known position, dead reckoning, inertial navigation, and so on) to give degraded answers when fewer than four satellites are visible.

WGS84

This is the geodetic system (datum) used mainly in GPS applications as it contains the latest revisions (in 1984).

World position is expressed in terms of latitude, longitude (degrees) and altitude (meters).
Latitude ranges from -90 degrees south (South Pole) to +90 degrees north (North Pole). 0 degrees is situated on equator line.

Longitude ranges from -180 degrees west to +180 degrees east (both points coincide in the same place). 0 degrees longitude is situated 100m east of the Prime Meridian at Greenwich, UK.

Degrees can be represented in 3 main ways:

1. DMS (Degree Minute Second), e.g. 1° 26’ 0.04”
2. DDM (Degree Decimal Minute), e.g. 1° 26.006
3. DDD (Decimal Degree), e.g. 1.43334423

To convert from DMS to DDM and DDD take last part (ex. seconds), divide by 60 and you have minutes, after that add to the number of minutes you had and again divide by 60 and add to the number of degrees to obtain DDD.

**GPS System Architecture**

The GPS system is composed of 3 segments: Space Segment (SS), Control Segment(CS) and User Segment(US):

- **Space Segment - Satellites**: currently, GPS uses 31 MEO satellites on 6 plans centered on Earth. The satellites (or Space Vehicles - SV) orbit at an altitude of approximately 20,200 kilometers. Each SV makes 2 complete orbits in a sidereal day. Each MEO (Medium Earth Orbit) satellite orbits the earth faster than the rotation of the earth so during a day from a fixed point on earth we can detect different satellites above our position. This is a link to a page that describes the different orbits: [http://en.wikipedia.org/wiki/Image:Orbits_around_earth_scale_diagram.svg](http://en.wikipedia.org/wiki/Image:Orbits_around_earth_scale_diagram.svg)

- **Control Segment**: this segment is composed by stations that monitor the satellite’s position and activity.

- **User Segment - Receiver**: in general, GPS receivers are composed of an antenna, tuned to the frequencies transmitted by the satellites, receiver-processors, and a highly-stable clock (often a crystal oscillator). Actual receivers have between twelve and twenty channels so they can monitor simultaneously 12-20 SVs. As of 2006, even low-cost units commonly include Wide Area Augmentation System (WAAS) or A-GPS (Asisted GPS) receivers. These type of receivers greatly increase accuracy, integrity and availability.

The protocol used for communication between a GPS receiver and a PC or other device is usually NMEA. A free daemon that can be used to get data from a GPS device is GPSD.
Navigation Signals
Each GPS satellite continuously broadcasts a Navigation Message at 50 bit/s giving the time-of-day, GPS week number and satellite health information (all transmitted in the first part of the message), an ephemeris (transmitted in the second part of the message) and an almanac (later part of the message). The messages are sent in frames, each taking 30 seconds to transmit 1500 bits.

Each satellite sends its position with at least 2 codes, the Coarse / Acquisition (C/A) code, which is freely available to the public, and the Precise (P) code, which is usually encrypted and reserved for military applications. The mainly used frequency is L1 (1575.42 MHz).

Accuracy
The accuracy of a GPS receiver is about 15m on simple GPS operation. Accuracy can be improved with:

- Augmentation, like WAAS, EGNOS or Assisted-GPS
- Precise monitoring through multiple frequency monitoring (L1, L2 and L5) and Carrier-Phase Enhancement (CPGPS) which together with DGPS can give 30cm absolute accuracy
- Relative Kinematic Positioning (RKP). In this approach, determination of range signal can be resolved to a precision of less than 10 centimeters.

Errors
GPS accuracy errors can be determined by one of the following:

- **Atmospheric effects**: these errors come from the ionosphere being traversed by the GPS signal
- **Multipath effects**: errors due to signal reflecting into buildings
- **Ephemeris and clock errors**
- **Selective availability**: this is not a problem anymore. It was used by the US to insert errors in Civilian signal.
- **Relativity**: clocks in space tick slower than in Earth.
- **Sagnac distortion**: problems that come from the fact that GPS is defined in inertial system but observations are Earth-based which are not inertial.
- **Interface with natural and artificial sources**

GSM
This type of location technology uses the GSM network in order to give an estimate location to the user. It is less accurate than GPS and only by itself it needs a database with cells positions or the agreement of the network operator to provide location services to the client. The main methods of location using the GSM network are as follows:
Network Based

Network-based techniques utilize the service provider’s network infrastructure to identify the location of the handset. The advantage of network-based techniques is that they can be implemented non-intrusively, without affecting the handsets.

The accuracy of network-based techniques varies, with cell identification as the least accurate and triangulation as the most accurate. The accuracy of network-based techniques is closely dependent on the concentration of base station cells, with urban environments achieving the highest possible accuracy.

One of the key challenges of network-based techniques is the requirement to work closely with the service provider, as it entails the installation of hardware and software within the operator’s infrastructure. Often, a legislative framework, such as E911, would need to be in place to compel the cooperation of the service provider as well as to safeguard the privacy of the information.

Handset Based

Handset-based technology requires the installation of client software on the handset to determine its location. This technique determines the location of the handset by computing its location by cell identification, signal strengths of the home and neighboring cells or the latitude and longitude, if the handset is equipped with a GPS module. The calculated location is then sent from the handset to a location server.

The key disadvantage of this technique is the necessity of installing software on the handset. It requires the active cooperation of the mobile subscriber as well as software that must be able to handle the different operating systems of the handsets. Typically, only a smart phone, such as one based on Symbian or Windows Mobile, would be able to run such software.

One of the proposed workarounds is the installation of embedded hardware or software on the handset by the manufacturers. However, the obvious difficulty of convincing different manufacturers to cooperate on a common mechanism and to address the cost issue means that this avenue has not made any significant headway.

Another difficulty would be to address the issue of foreign handsets that are roaming in the network.

Hybrid

Hybrid-based techniques use a combination of network-based and handset-based technologies for location determination. One example would be Assisted-GPS, which uses both GPS and network information to compute the location. Hybrid-based techniques give the best accuracy of the three but inherit the limitations and challenges of network-based and handset-based technologies.
Wireless Networks
As with GSM, Wireless networks can be used to obtain the location of a mobile device. This method can lead to very good results in terms of accuracy (less than 5 meters) if good algorithms are employed. The best location method is to constantly analyze current WLAN access points (AP) and determine the position of the mobile device based on the current receiving signal strength of all Wireless stations.

For this method a database has to be made with all wireless APs in an area and record its position along with its BSSID (MAC address of access point). Mobile devices can choose to have the database in memory in order to apply the location algorithm or they can send the information about detected WLANs to a server that applies the location algorithm and sends back the location of the device.

TDOA
Time difference of arrival (TDOA) is an efficient method of localization. It is being applied in the US to locate the users using the GSM network as a federal regulation has been emitted saying that a user calling to the 911 (emergency call) must be located in any circumstances.

If a pulse is emitted from a platform, it will arrive at slightly different times at two spatially separated receiver sites, the TDOA being due to the different distances of each receiver from the platform. In fact, for given locations of the two receivers, a whole series of emitter locations would give the same measurement of TDOA. Given two receiver locations and a known TDOA, the locus of possible emitter locations is a one half of a two-sheeted hyperboloid.

In simple terms, with two receivers at known locations, an emitter can be located onto a hyperboloid. Note that the receivers do not need to know the absolute time at which the pulse was transmitted - only the time difference is needed.

Consider now a third receiver at a third location. This would provide a second TDOA measurement and hence locate the emitter on a second hyperboloid. The intersection of these two hyperboloids describes a curve on which the emitter lies.

If a fourth receiver is now introduced, a third TDOA measurement is available and the intersection of the resulting third hyperboloid with the curve already found with the other three receivers defines a unique point in space. The emitter's location is therefore fully determined in 3D.

Photo Localization
This method is based on images made by the user of the mobile device as most devices currently have a small camera. In this method a user can send an image taken from a particular place (e.g. a cathedral) to a server and the server might reply with the position of the place in the image. This method involves a big database on the server, having the coordinates of each image. Also a complex algorithm that is capable to match an image sent by the mobile user with the images on the database must run on the
server in order to compute the position of the device. We don't know current implementations of this method but it might be a good way to locate the user on demand.

**Hardware**

In this section we present some mobile devices that represent the state of the art in mobile technology.

**iPhone**

The iPhone is an Internet-enabled multimedia mobile phone designed and marketed by Apple Inc. It has a multi-touch screen with virtual keyboard and buttons. The iPhone's functions include those of a camera phone, portable media player (iPod), in addition to text messaging and visual voicemail. It also offers Internet services including e-mail, web browsing, and local Wi-Fi connectivity. It is a quad-band mobile phone that uses the GSM standard, and hence has international capability. It supports the EDGE data technology. This device runs its proprietary iPhone Operating System.

![iPhone](image)

**BlackBerry Bold**

Modern BlackBerry handhelds incorporate an ARM 7 or ARM9 processor, while older BlackBerry 950 and 957 handhelds used Intel 80386 processors. The latest GSM BlackBerry models (8100 and 8700 series) have an Intel PXA901 312 MHz processor, 64 MB flash memory and 16 MB SDRAM CDMA BlackBerry smart phones are based on Qualcomm MSM6x00 chipsets which also include the ARM 9-based processor and GSM 900/1800 roaming (as the case with the 8830). The devices are very popular with some businesses, where they are primarily used to provide e-mail access to roaming employees.
integrate the BlackBerry into a company’s systems, the installation of BlackBerry Enterprise Server (BES) is required. This device runs the RIM Blackberry Operating System.

**Figure 2**

**HTC Diamond**

With HTC’s vibrant touch-responsive user interface, TouchFLO™ 3D, and ultra-fast HSDPA internet connectivity... the HTC Touch Diamond offers a rich online experience to rival a notebook computer, allowing you to interact with Google, YouTube, and Wikipedia as freely as you would with a broadband connection.

A 3.2 megapixel auto-focus camera will help you capture the perfect moment in style and with a massive 4GB of internal storage you can keep all the files you need. The integrated ultra-sensitive GPS will help you find your destination as quickly and efficiently as a dedicated satellite navigation unit.

This device runs the Windows Mobile 6.1 Operating System.

**Figure 3**
Nokia N95 8GB

This is the smartphone used in this project. It has many hardware capabilities, including internal Assisted-GPS receiver, WLAN and graphic accelerator. Some other hardware capabilities very important are the dual-processor ARM 11 at 332 MHz and the big memory (8 GB). This device also has an accelerometer that can be used in any application in order to detect the position and rotation of the phone. It also has an incredible Camera Carl Zeiss Optics of 5 megapixels.

This phone runs the Symbian Operating System v9.2. For development it supports the Nokia S60 3rd Edition Feature Pack 1. This development framework offers many APIs to access all of the phone's capabilities, including the graphic accelerator, internal GPS, camera and wireless.

We used this phone after many researches because it offered us the needed APIs to work with GPS, GSM and WLAN information and also because of the large developer community (http://forum.nokia.com).

Figure 4

Figure 5
Software

In this section we present an overview of the operating system architecture for mobile devices and we provide a better insight of the Symbian Operating System that the Nokia N95 8GB uses.

The following image presents the overview of operating systems and development platforms for mobile devices. This diagram is the result of an extensive research on current mobile phones:

![Diagram of Operating Systems and Development Platforms](image-url)
Symbian Operating System

Symbian OS is the advanced, open operating system licensed by the world’s leading mobile phone manufacturers. It is designed for the specific requirements of advanced 2.5G and 3G mobile phones. Symbian OS combines the power of an integrated applications environment with mobile telephony, bringing advanced data services to the mass market.

Symbian OS supports a wide range of device categories with several user interfaces, including Nokia Series 60, UIQ and the NTT DoCoMo common software platform for 3G FOMATM handsets. The commonality of Symbian OS APIs enables development that targets all of these phone platforms and categories.

Key features of Symbian OS v9.2:

- Platform security – system defense mechanism based on granting and monitoring application capabilities. Infrastructure to allow applications to have private protected data stores. In addition, full encryption and certificate management, secure protocols (HTTPS, SSL and TLS) and WIM framework
- Rich suite of application services – the suite includes services for contacts, scheduling and messaging, OBEX for exchanging appointments (vCalendar) and business cards (vCard); integrated APIs for data management, text, clipboard and graphics
- Comprehensive Java support – supports the latest wireless Java standards, including CLDC 1.1, MIDP 2.0, JTWII (JSR185), Mobile Media API (JSR135), Bluetooth (JSR082), Wireless Messaging (JSR120), Mobile 3D Graphics API (JSR184), Personal Information Management and FileGCF APIs (JSR075) and Content Handling API (JSR 211)
- Hard real-time capabilities – a real-time, multithreaded kernel provides the basis for a robust, power-efficient and responsive phone and enables support for single core architecture
- Support for the latest hardware – supports latest CPU architectures, peripherals and internal and external memory types
- Complete messaging capabilities – Internet email based on Post Office Protocol (POP3), Internet Message Access Protocol (IMAP4) and Simple Mail Transfer Protocol (SMTP) including attachment support. Also support for IMAP4 IDLE command. Email format is based on MIME (Multipurpose Internet Mail Extensions). Support for Short Message Service (SMS) and Enhanced Message Service (EMS)
- Rich multimedia capabilities – audio and video support for recording, playback and streaming; image conversion
- Powerful graphics – direct access to screen and keyboard for high performance; graphics accelerator API; increased UI flexibility (support for multiple simultaneous display, multiple display sizes and multiple display orientation)
• Broad support for communications protocols – wide area networking stacks including TCP/IP (dual mode IPv4/v6), RTP, RTCP and SIP, and WAP 2.0 (Connectionless WSP and WAP Push); personal area networking support including infrared (IrDA), Bluetooth and USB; support for multi-homing and link layer Quality-of-Service (QoS) on GPRS and UMTS networks; support is also included for bearer mobility

• Optimized for mobile phones – Symbian OS v9.2 is optimized for the 3G market with support for CDMA (3GPP R5); GSM circuit switched voice and data (CSD and EDGE CSD) and packet-based data (GPRS and EDGE GPRS); CDMA circuit switched voice, data and packet based data (IS-95 and 1xRTT); SIM, RUIM, UICC Toolkit; other standards can be implemented by licensees through extensible APIs of the telephony subsystem; 3GPP R5 IMS; Smart Card access APIs

• CDMA-specific features including CDMA network roaming, third party OTA API, NAM programming mode, CDMA SMS stack, NAI handset identification, interfaces to enable Mobile IP and bridge and router gateway modes of operation

• International support – supports the Unicode Standard version 3.0

• User-friendly data synchronization – over-the-air (OTA) synchronization and PC-based synchronization (over Bluetooth, infrared and USB) support using OMA Data Sync 1.2; a PC Connectivity framework providing the ability to transfer files and synchronize PIM data

• Lowering cost of ownership through Device Management/OTA provisioning capabilities – OMA DM 1.2 compliant, OMA Client provisioning v1.1

• Rich set of options for developing for Symbian OS – content development options include: C++, Java (J2ME) MIDP 2.0 and WAP; tools are available for building C++ and Java applications; reference telephony abstraction layer for 2G, 2.5G and 3G provided.

Project Specifications

In this section there will be presented the project specifications, explaining what was the target of the project, the constraints and the needs of the application.

We start by showing a diagram of the system:
As it can be seen from the image, the goal of the application was to implement Location Based Services on a mobile device (Nokia N95) in order to provide a mobile guide application to Immersive-Solutions. The main utility of this application is to guide a user in a stadium, providing him the path from the entrance of the stadium to his seat. Another enhancement that could be done is to provide a solution for two persons to find each other in the stadium.

**Mobility**

The mobility is the key factor of the project. The application must run on a mobile device thus serving as a mobile guide to the user. This also implies that the user doesn’t have to do anything else besides using this application. There should be no need to connect to a server, download intensive data, having an
account or any difficult navigation menu. The application should be as simple as possible but it should provide specific and useful information to the user.

**3D interface**

The objectives of Immersive-Solutions are the 3D models in general and this is why the mobile application should display a 3D-like view to the user. For these reason in this project we have to integrate a 3D view in the mobile device.

We want to show the user different perspectives of the stadium, showing him its position as well as his destination and path to follow into these views. Here it is an example of such view:

![Figure 8](image)

Figure 8

In the application as we will show later we used these images and we provided the possibility to zoom in/out and to scroll left, right, top and bottom in order to give the sensation of immersion into the stadium. As it can be seen from the previous image this type of visualization is much more intuitive and closed to reality than simple 2D maps like this:
Accuracy
A very important need of the application is to give an accurate position to the user. That is why we tried to create a position engine that was able to get useful information from GPS, WLAN and GSM networks in order to provide increased accuracy. GPS gives accuracy between 20 and 200 meters, depending on the surrounding environment. In the city it gives a bad accuracy of 100–200 m while in rural areas it can give accuracy of about 20m or less. In our project we created a position logger that retrieves information from WLAN, GSM and GPS but in the end we only used GPS for positioning. Our application works fine using only GPS for positioning as what is needed with precise accuracy is the target position while current position can be an approximation for computing the path between current and target positions. In a future version of the application we could use WLAN and GSM information that we already have in order to provide a more accurate position.

Project Plan. System architecture
In this section the system architecture and the project plan are presented. We start by showing an overview of the application and the timetable and then we present the modules used by the application. In the next section these modules are presented in more detail.

Project timetable
The project timetable is presented in the following Gant diagram:
Application Overview

In the next diagram we can observe the application overview. Each component is explained below.

![Application Overview Diagram]

**Figure 11**

**Application UI**

The application user interface (UI) is the main component, that handles input commands from the user and sends them to the application view or to the network engine as it is the case. It is also the component in charge of resizing the screen or changing screen orientation if the device requests it (the Nokia N95 can display both on landscape and vertical). This is the first class (object) that is loaded when the application starts. It is responsible for loading all the other components, resources and for creating new instances of the Application View and Network Engine.
Network Engine

The Network Engine is the module used to get information from the GPS, GSM and WLAN. It is also the class used to create the LBS Logger that records the GPS, GSM and WLAN information into an XML file each time the current position (given by the GPS) changes by a specific amount.

LBSInfo

This is the module (class) used to get GPS information. The result is an object containing the current position expressed in latitude, longitude (decimal degrees) and altitude (meters) using the WGS84 datum. There is possible to obtain also the current time, speed or accuracy.

GSMInfo

This module is used to obtain information about the GSM network. It can only retrieve the information of one cell at a time and only for the current carrier operator. The information provided is:

- MCC – Mobile Country Code
- MNC – Mobile Network Code
- LAC – Location Area Code
- Cell ID – The ID of the current connected cell from the GSM network
- Long Name – The full name of the carrier operator (e.g. Vodafone)
- Signal Strength – The strength of the received signal, very useful for positioning

WLANInfo

This module is used to retrieve information about currently detected wireless networks. Using this module we can obtain the following information:

- ESSID – SSID name of the network
- Mode of operation – Ad-hoc or Infrastructure
- Security Mode – WEP, WAP, WPA2 or NONE
- BSSID – MAC address of the access point or host in ad-hoc mode
- Signal Strength – received signal strength measured in negative dBm values (same for GSM)

Logger

We use this module to log all available information (WLAN, GPS and GSM). With this information we can create a database useful for better positioning. It is also possible to create a radio (shown in the following image) in order to visualize better the covered area of networks. In our application a log is done each time the current position changes with an amount K from latest logged position. This is a parameter that can be changed in the application or some other method for logging can be used (e.g. using a timer). Each entry log is represented in XML format as it will be shown in the implementation section.
The Application View is the module that handles visualization. It is the part that renders the visual content on the screen. This module is also responsible for initializing the Location Map class used to handle the stadium maps, views and transformations between GPS and pixel coordinates. The application view can communicate directly with the Network Engine in order to get updates on position and redraw the position on the screen. It also takes the input from the Application UI to zoom in/out, move the position and change the views.
Location Map
The Location Map is an abstract class that defines the methods that need to be implemented by each venue that wants to register to the system as a model. Each venue can have several views that can be used to display the location, target and path to the user. This class will be described more in detail in the next section. For the moment it should be known that this class is the responsible for the different venues and views used in the application, it handles the conversions between GPS and pixel coordinates.

Development Platform
The development platform used for the project was composed of the following:

- Symbian C++: the native language used in Symbian applications
- S60 3rd edition Feature Pack 1 development library
- Carbide C++ IDE

Symbian C++
This is the native programming language used in Symbian applications. All of the applications and features provided by phones that use the Symbian OS are written in Symbian C++. This provides the best solution for fastest applications and it also consumes the least amount of RAM memory.

The most important drawback of this development platform is its particularity. In order to create an application using Symbian C++, besides having a good knowledge in C++ we need to learn all the conventions and features of Symbian C++. In the next paragraphs we discuss these particularities.

C classes, T classes, R classes
You will notice that class names in Symbian begin with different letters. The letter with which they begin depends on the type of the class. If the name begins with T then it is a so called T class. Such classes cannot own external data - i.e. they cannot have pointers to data that is not owned by a different object. Such classes usually don’t need a destructor. They can be created on the stack (i.e. as automatic variables) or inline in a class.

If a class owns some external data, then it must be a subclass (direct or indirect) of CBase and its name will begin with C - this will be a C class. The CBase provides a virtual destructor, so even after casting to CBase * the right destructor will be called. Another nice feature of C classes is that it fills the data area with 0’s, so you don’t need to initialize variables to 0 or NULL (what is the most common case). Because the C classes will be pushed on the cleanup stack (which will be described later), C classes cannot be created on the stack - they must be created with the new operator (usually new (ELeave) as will be explained later).
Somehow analogical to T and C classes are the structs and classes in C#. Symbian uses also classes which are analogical to Java or C# interfaces. They are prefixed with M and can contain only pure virtual functions. Symbian discourages multiple inheritances unless all the super classes except at most one are M classes.

The last commonly used prefix is R. These classes access to some external resources (e.g. files, sockets) and needs to be closed before exit. These classes don’t have a common ancestor and there is no single closing function. The closing function is most often called Close, sometimes Dispose or Delete.

There is also a convention for variables names - the class fields have names starting with ‘i’ while functions arguments have names beginning with ‘a’.

Note that although the T classes can be created on the stack it is not always the best idea. The stack on the device is very small - by default 8kB. Some T classes are relatively big - e.g. a TFileName uses more than 512 bytes. The emulator uses the Windows NT stack (which can grow up to 8MB by default), so you will notice the overflow only when testing on the device.

Another Symbian feature is that DLLs cannot contain static data. The problem is that applications (*.app files) are DLLs, so when writing an application you should not use static data. Static data can be used in servers (*.exe).

**Defined Types**

In Symbian C++ there are a number of most used types that an application should use. Instead of a usual “int” an application should use the “TInt” or the bit variants “TInt16” and “TInt32”, representing an integer on 16 and respectively 32 bits. It is also the same for “TBool” (instead of “bool”), “TReal” (double) and so on.

**Leaves and the cleanup stack**

Symbian doesn’t use C++ exceptions because it was written before the exceptions where standardized and they require lots of resources. Instead Symbian uses a mechanism of leaves very similar to exceptions.

A leave is generated with User::Leave (errorCode). The error code is a TInt which carries the information about the reason of the leave (there are many predefined error codes, like KErrNotFound). It is like the throw instruction in C++ with the difference that the information about the failure must be contained in the error code, not in an arbitrary object. The User::LeaveIfError (errorCode) is also very useful.

When a leave occurs then (similarly to exceptions) the control will return from all the functions (the stack will be unwinded) until it reaches a trap harness (which is like a try/catch block in standard C++). A trap harness is defined by TRAP:
Tint error;

TRAP (error, FunctionThatLeavesL());  // try

If (error==KErrNotFound)     // catch
{
    //...
}

If (error!= KErrNone)       // catch
{
    //...
}

There is also a TRAPD macro that automatically defines the variable in the first parameter.

When a leaves occurs all the data which is not owned by an object (i.e. data which is only pointed by an automatic variable in one of the functions we return from because of the leave) should be destroyed to prevent a memory leak. To achieve that Symbian uses a concept of Cleanup Stack.

If you have an object that should be destroyed in case of a leave push it on the cleanup stack with CleanupStack::PushL (object). When the leave occurs all the objects on the cleanup stack pushed after entering the trap harness will be destroyed. When an object should be removed from the cleanup stack, e.g. because it will be destroyed or the ownership of it will be taken by an object use CleanupStack::Pop to remove the top-most object from the stack (In the first case CleanupStack::PopAndDestroy is very practical). There is also a function CleanupStack::Pop (object) with checks if the popped object is the same as the pushed one.

The CleanupStack::PushL function has an overloaded version that accepts a CBase * class which has a virtual destructor. When a leaves occurs the destructor will be called and the owned data will be deleted. That’s why all objects that owns some external data should be descendant of CBase. The R classes don’t have a common ancestor so the closing function won’t be called automatically. However there are functions like CleanupClosePushL, CleanupReleasePushL and CleanupDeletePushL that, thanks to the magic of templates, will guarantee a call of Close, Release or Delete before destroying the object by the Cleanup Stack.
Strings
Symbian doesn’t use the C char * strings but provides classes that do range checking called descriptors. Overflows in a char str[n] strings are probably the most common mistake that allows an attacker to take control over a system. This is achieved by overwriting the stack frame. So it’s nice that Symbian libraries do all the range checking.

There are several classes of the descriptors. The TDesC is an abstract class which is an ancestor of constant strings. The TDes is a subclass TDesC and adds some functions to modify strings - it is an ancestor of all the modifiable strings. A TBuf<n> is a class that keeps n characters as a part of itself (thus the T prefix). It is usually used to keep some temporary strings as automatic variables. However because of the small stack in Symbian devices you should not keep too many strings on the stack.

TPtr objects store a pointer to a buffer (with information about the string length and maximum length) which it doesn’t own. E.g. a TPtr can point to characters 5 to 10 of a TBuf<64>. Destroying the TPtr will not hurt the data in the TBuf. However if you destroy the TBuf, the TPtr will become invalid. A TPtrC is a non-modifiable version of TPtr.

If you want to keep the string on the heap use an HBufC object. The HBufC objects cannot be kept on the stack but must be created on the heap like C classes but they also don’t need a virtual destructor as all the data is inline like in T classes. That’s why it has a special prefix - H. As the C suffix suggests this class is non-modifiable directly but there is a Des function which returns a TPtr to the data, which will allow you to modify the content.

S60 FP1
The Symbian C++ development platform gives the developer a large set of APIs and libraries to be used in applications. These cover most needs to interact with the mobile device. They also provide the necessary basic controls for the application UI but it doesn’t provide custom and specific controls. This is why Nokia provides its own set of controls and APIs for the UI, the S60 platform. Another specific platform on top of the Symbian platform is the UIQ, most used by the Sony-Ericsson devices. The main difference between UIQ and S60 is that S60 (and thus Nokia phones) doesn’t allow for touch screen functions and controls.

Besides UI controls the S60 Feature Pack 1 (FP1) gives a set of APIs specifically designed to work with phones supporting these functions. Each Nokia S60 device has a specifically S60 set of libraries that can be accessed by an application. That is why a developer must use the right development platform when developing for a specific device.

In our application we use the S60 Feature Pack 1 as this is the latest supported library by our phone, the Nokia N95. All the older versions are supported by new the phone, which means that S60 keeps compatibility with older versions.
Carbide C++

Carbide C++ is the Integrated Development Environment (IDE) used in our application. It is provided free of charges by Nokia. It comes with all the features, including phone debugging, for one month and after that if we don’t buy a license it reverts to standard features, which are enough to develop any application. The standard features include emulator debug. The emulator behaves very well and supports most of the APIs; the only problem is that it doesn’t provide WLAN or GSM emulation.

Design Patterns

This application represents an extensive project and uses a big number of classes. To maintain clarity, separate UI from Engine and to deal with multiple inheritances we used a series of design patterns presented in the following paragraphs.

Model-View-Controller

The Model-View-Controller (MVC) architecture is a widely-used architectural approach for interactive applications. It divides functionality among objects involved in maintaining and presenting data to minimize the degree of coupling between the objects. The architecture maps traditional application tasks—input, processing, and output—to the graphical user interaction model.

The MVC architecture divides applications into three layers—model, view, and controller—and decouples their respective responsibilities. Each layer handles specific tasks and has specific responsibilities to the other areas.

![Model-View-Controller Diagram](Figure 13)
A model represents business data and business logic or operations that govern access and modification of this business data. Often the model serves as a software approximation to real-world functionality. The model notifies views when it changes and provides the ability for the view to query the model about its state. It also provides the ability for the controller to access application functionality encapsulated by the model. In our application the Model is represented by the Network Engine as it can be seen from the application diagram (Figure 11).

A view renders the contents of a model. It accesses data from the model and specifies how that data should be presented. It updates data presentation when the model changes. A view also forwards user input to a controller. In our application the view is represented by the Application View module (class).

A controller defines application behavior. It dispatches user requests and selects views for presentation. It interprets user inputs and maps them into actions to be performed by the model. In a stand-alone GUI client, user inputs include button clicks and menu selections. A controller selects the next view to display based on the user interactions and the outcome of the model operations. An application typically has one controller for each set of related functionality. Some applications use a separate controller for each client type, because view interaction and selection often vary between client types. The Controller can be identified as the Application UI in our project.

Factory Pattern

The factory method pattern is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The factory method design pattern handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created. More generally, the term factory method is often used to refer to any method whose main purpose is creation of objects.

This model has been used to create the different LocationMap objects into our application. Instead of using the constructor of each view/venue we created a class that contains static methods for creating all the views. This way we only have to use one class (MapFactory) to create all the views we need, regardless of the name of the actual classes derived from LocationMap.

Second Constructor (Symbian C++)

This pattern is used in Symbian C++ to create safe constructors. This means that we hide the implementation of the real constructor (that must not leave, as explained before) behind a public method that can leave. This involves three steps:

1. Inside the public static method that can leave (usually NewL or NewLC) we create an instance of the object using its constructor (that must not have any code that leaves)
2. We push this object into the stack (CleanupStack)
3. We call the private ConstructL function of the object that can leave. This function is the actual constructor of the object, where all the heavy data is initialized.

**Implementation**

In this section we will present the details of the project implementation.

**Active Objects**

Each service in Symbian (file, graphics, wireless, GSM, etc...) is exposed to the applications as a server. In order to get the information needed an application must initialize a connection to the server, wait for the action to be completed on that server and get the results.

In order to get the results, we need a TStatus object that will be used by the server to announce our application that the operation has been completed or an error has occurred. There are two ways to deal with this framework. The first way, simple but inefficient is to block until the action has been completed on the server side. This is done like this:

```cpp
//Inside a function
TStatus aStatus;
RServer aServer;
RResult aResult;
aServer.Connect();
aServer.CallNeededFunction(&aResult, &aStatus);
User::WaitForRequest(&aStatus);
//do whatever with results
```

As I can be seen from this code the application will effectively block until the result is obtained or an error occurs. This is highly discouraged by Symbian and it is a bad way to get information in commercial applications (of course it is easy).

The second way is to make an asynchronous call to the server, and deal with the result in a callback function that will be called by the server when the result is ready. This is the recommended way and this is how we implemented all the calls in our application. To deal with the asynchronous calls and callback functions Symbian has the concept of Active Objects. An “Active Object” is a class derived from CActive. CActive defines the TStatus object needed to be passed in a server request and it also defines the following virtual methods:

- RunL: this is the callback function called by the server when the action has completed or an error occurred. In order to verify if everything completed fine or some error occurred we can check the iStatus member.
• DoCancel: this method is called when we want to cancel an active request using the Cancel() function
• RunError: this method is automatically called by the CActive class if the RunL function leaves. In this way we can trap any leave in our RunL function.

Before an Active Object can be used it has to be pushed into the CActiveScheduler. The CActiveScheduler is a kernel thread that runs jobs taken from a queue. We just have to add our object to the CActiveScheduler in order to be taken into consideration. After calling the request method on the server side with our iStatus object of the CActive object we must set the object to the “active” state by calling theSetActive function. Here it is an example code (the same approach is used in our implementation):

```cpp
class CMyActive : public CActive
{
private:
  CProvider* iProvider;
  ..
};

CM yActive::CM yActive()
  : CActive(CActive::EPriorityStandard)
  {
    CActiveScheduler::Add(this);
  }

void AsyncRequest()
  {
    iProvider->Request(iStatus);
    SetActive();
  }

void CM yActive::RunL()
  {
    // check the iStatus for possible errors and
    // resubmit the request if necessary
  }

void CM yActive::RunError(TInt aError)
  {
    // check the error and do whatever action is necessary
  }
```
Network Engine

The Network Engine is one of the most important parts in our application. As described in the architecture section, it is the module used to get available information for GPS, GSM and WLAN as well as the module used to make the log file.

The Network Engine has an object for each sub-module:

- CGSMInfo *iGSMInfo: the iGSMInfo is an instance of the CGSMInfo class used to get GSM Information
- CWLANInfo *iWLANInfo: for WLAN information
- CLBSInfo *iLBSInfo: for GPS position. Maybe the name is not the best choice, we could have better used CGPSInfo instead but this is not a problem for the application
- CFileLogger *iFileLogger: used to add entries to the log

The most important methods implemented by the Network Engine are:

- void GetGSMInfoL(TDes &aNetworkInfo): used to get information in XML format from the GSMInfo module. aNetworkInfo is a descriptor used to hold the resulting information
- void GetWLANInfo(TDes& aWLANInfo): used to get information about wireless networks
- void LBSPositionUpdatedL(TPositionInfoBase& aPosInfo): callback function called by the LBSInfo module when a new GPS position is obtained. AddLogL is called in this function
- TPosition GetGPSCoordonates(): used to get the last GPS position obtained
- void AddLogL(const TPosition &aPosition): this function is called when the current GPS position obtained (via LBSPositionUpdatedL) differs with more than KGPSDifference (0.0001) degrees from the last logged position

The Network Engine is responsible for initializing all the sub-modules and handle errors, messages or callbacks from them. The initialization is done in the private ConstructL function like this:

...  
  iLBSInfo = CLBSInfo::NewL(*this);  
  iGSMInfo = CGSMInfo::NewL(this);  
  iWLANInfo = CWLANInfo::NewL(this);  
...

CLBSInfo class gets *this instead of this because of the implementation (uses a reference instead of a pointer) but the functionality and behavior is the same.
**Logger**

The logger was the first target of the project. We wanted to get all the possible information from the network (WLAN, GPS and GSM using the Network Engine) and store it in order to create a database that could have been used later for good localization.

To store the data we have used a file (LocationInfo.log) where we saved all the information in XML UNICODE format.

In Symbian C++ we can use the RFs file system to make a connection to the file server like this:

```c++
TInt err = iFs.Connect();
```

To open a file we create a RFile object (attention to the R leading letter which means it is a resource object – of course, the resource is the file) that we will use to actually open the file, write and read from it:

```c++
err = iFile.Create(iFs, aLogFile, EFileShareAny | EFileWrite);
User::LeaveIfError(err);
```

To write something using our CFileLogger class we use a local buffer implemented using RBuf. RBuf is the recommended Symbian object to use for dynamic buffers. It behaves like a TDes or TDesC and can be passed as a parameter to functions accepting those but it holds its content on the heap, thus allowing for dynamic increase without any stack size limitation. It dynamically increases its size by chunks of memory defined by the granularity parameter (it can be changed at construction). To create an RBuf we used the following code:

```c++
RBuf8 iLogBuffer;
iLogBuffer.CreateL(size);
```

where size is the maximum size that this RBuf will hold. RBuf8 is a version of RBuf that stores data as 8 bits values.

Writing to the file is simple, just call the Write function:

```c++
iFile.Write(iLogBuffer);
```

All the information in the log file is in XML format, as this gives an easy way of transferring and reading the information. Also it is very easy to convert to a database from XML.

A new entry (&lt;Information&gt;) is added to the log file each time our position changes with more than KGPSDifference degrees since the last log. The function used to make a log is AddLogL (const TPosition
\&apos;Position), where \texttt{aPosition} is the current GPS position (see the introduction for more details). The TPosition class is part of the Symbian Location framework and it exposes the following important methods:

- \texttt{Speed}(): used to get the speed, based on another (old) position
- \texttt{Time}(): returns the GPS time, which can be much more precise than phone's time
- \texttt{Longitude}(): in DDD (decimal degree) format, as a \texttt{TReal}
- \texttt{Latitude}(): same
- \texttt{Altitude}(): in meters

In the AddLogL function we first create a RBuf buffer to store all the entry information (GPS, GSM and WLAN information) and then we write the whole content to the file logger. The XML tags used in the logger are this:

- \texttt{<NetworkEngine>}: this is the parent tag for the entire log file
- \texttt{<Information>}: this tag represents a new entry in the log
- \texttt{<Time>}: the current time given by the phone
- \texttt{<GPS>}: next is information related to GPS
- \texttt{<Latitude>}
- \texttt{<Longitude>}
- \texttt{<Altitude>}
- \texttt{<GPSTime>}: the time returned by the GPS receiver
- \texttt{<WLAN>}: next is WLAN information
- \texttt{<GSM>}: next is GSM information

GSM and WLAN specific tags will be presented in their section.

In our application we didn't use the data obtained with the logger but the information can be very useful for future improvements or for other applications that need this information (as it will be seen in the test and evaluation section).

The following image represents a sample obtained from a log file created with our application:
The GPS module, implemented by the CLBSInfo class is used to get the GPS information from the internal GPS receiver of our device (Nokia N95). In order to get the GPS information we had to use the Location framework which gives a set of APIs used to obtain location information in Symbian C++. The
CLBSInfo class is derived from CActive as it has to make asynchronous requests to the positioning server which returns the position from the internal GPS receiver.

To start getting position updates we must do the following:

1. Connect to the positioning server (RPositionServer):
   ```cpp
   iPosServer.Connect();
   ```
2. Select the requested or available position technology. The N95 device supports Network localization (given by cellular network, but doesn’t work with GSM yet), GPS positioning (using standard GPS), Bluetooth positioning (based on known position of Bluetooth emitters, but we don’t rely on this) and A-GPS which combines standard GPS with position updates from the GSM network, resulting in faster position fixes and better accuracy. We try to get A-GPS if possible, if not we request standard GPS
3. Open a sub-session to the RPositionServer using a RPositioner object and the selected technology ID:
   ```cpp
   iPositioner.Open(iPosServer, modId);
   ```
4. Set the requestor information (these are some default values)
5. Set the update parameters (time to wait for update, update interval, timeout, etc.)
6. Start the process of getting position updates (as asynchronous calls, which use the RunL callback function):
   ```cpp
   iPositioner.GetLastKnownPosition(*iPosInfoBase,iStatus);
   SetActive();
   ```

In the RunL function we make sure we have a valid position, we store it as the last position obtained, we call the LBSPositionUpdatedL function to notify the Network Engine and then we make the asynchronous call again to constantly get position updates.

The LBSPositionUpdatedL function is defined in a separate class (MPositionObserver) which represents an interface (as the leading ‘M’ suggests). This interface is implemented by the Network Engine to get updates from the CLBSInfo class. The CLBSInfo constructor takes as a parameter an MPositionObserver object. This is why we have seen this initialization on the Network Engine:

```cpp
iLBSInfo = CLBSInfo::NewL(*this);
```

The Network Engine implements the MPositionObserver and thus can pass itself as a parameter. In this way the CLBSInfo class can call the LBSPositionUpdatedL function that will be handled by the Network Engine to process position updates and different messages. The same process is done with the other modules (WLAN, GSM).

**WLAN (WLANInfo)**

The WLANInfo module is implemented in the CWLANInfo class. This class is also derived from CActive as it has to make requests to the timer service.
The classes used by CWLANInfo to get WLAN information (CWlanMgmtClient and CWlanScanInfo) are part of the plug-in extensions package offered by Nokia for devices that use S60 3rd edition or later. The extension is not supported on older platforms. These extensions have been a great help in the information obtained from the wireless networks, as we retrieved the BSSID (MAC address of wireless access point). Without this extension it was impossible to retrieve this information or to get access to the wireless data frame.

A small drawback of the WLANInfo API (the plug-in extension for wireless information) is that it doesn’t provide a method to get the SSID of a wireless AP. For this reason we had to create a function that parses the wireless frame, using a pointer to the frame given by the API, and retrieve the SSID data. The function parses the wireless frame data searching for the Information Element (see IEEE 802.11 specification) with id 0 which gives the SSID value:

```c
while((error = scanInfo->NextIE(ie, length, &data)) == KErrNone) {
    if(ie == 0) {
        aSSID.Copy(data, length);
        return KErrNone;
    }
}
```

In order to get periodic updates of active wireless networks we used a timer (implemented by the class RTimer). To get the service of the timer we also have to make an asynchronous call. When the timer expires the RunL function is called. Here we save the wireless information and we restart the timer. This way we can obtain periodic updates on wireless networks.

The XML tags (and the information retrieved) used to represent wireless information in the log are:

- `<Network id="\d">`: used to mark a new wireless network that has been detected
- `<Security>`: specifies the security type of the network(open, wep, wpa)
- `<WLANType>`: the type of network (ad-hoc or Infrastructure)
- `<BSSID>`: BSSID of wireless AP
- `<SSID>`
- `<WLANStrength>`: power of received wireless signal, expressed in negative dBm values (see StrengthValue in GSM section)

**GSM (GSMInfo)**

The CGSMInfo class implements the GSM module. As with the other modules it is derived from CActive as it needs to make asynchronous calls to get GSM information.
The service used to get GSM information is exposed to Symbian C++ via the CTelephony API. This API provides a big number of methods to deal with the GSM network. We are interested in two methods:

- iTelephony->GetCurrentNetworkInfo(iStatus, iNwInfoPckg): this method retrieves information about current connected cell (MCC, MNC, LAC, CellID, LongName)
- iTelephony->GetSignalStrength(iStatus, iGSMPwrPkg): we use this method to obtain the strength of the received GSM signal. This is useful for GSM positioning (see Google MyLocation)

As it can be seen from the methods presented, both use the iStatus member and they are both asynchronous calls. The problem is that we only have an Active Object and only one callback to handle completion events. We also use a timer (RTimer) to get periodic updates from the GSM network thus having actually three different asynchronous calls to deal with.

To handle the three different asynchronous requests in a single RunL function we use a variable (TInt iTask) which stores the next task that should be done in RunL. At the beginning we start the timer. After the timer expires we call the first function to get MCC, MNC, etc... After RunL is called with the requested results we call the function to get the power strength. After we obtain the power strength we write the information in XML (see below) format to a local buffer (RBuf) and we restart the timer to continuously make the same process.

The XML tags used to register GSM information into the log file are:

- <MCC>: Mobile Country Code
- <MNC>: Mobile Network Code
- <LAC>: Local Area Code
- <CellID>: the ID of the current cell (where the phone is connected)
- <Cell>: for each different cell, in case we have more cells (it is not the case in our application as we can only retrieve the information for one cell at a time)
- <LongName>: the name of the GSM operator
- <StrengthValue>: the value obtained from the CTelephony API for the strength of the GSM signal. This value represents a negative value in dBm (decibel value of power referenced to 1 mW). This measure is often used to represent radio signal power
- <StrengthLines>: this value represents the number of lines that the phone should display for the strength of the GSM signal (from 0 to 7)

**Location Map**

This module is used to represent the current position, target position and path in different views of a venue. In our application we have used the TFC Stadium from Toulouse and we used different views taken from a 3D model of the stadium. The 3D model and the views have been created by Immersive-Solutions. Together with the Network Engine, this module represents one of the most important parts of
the application, as it has to make all the conversions between GPS and pixel positions and it also has to compute the path between the current and the target position as well as display it in the different views of the same venue.

We used a hierarchical model in order to handle different venues and views as well as to allow a scalable application. The hierarchy used is presented in the following figure:

![Diagram](image)

**Figure 15**

The LocationMap is represented in our application by the class CLocationMap. This is an abstract class that defines the methods that need to be implemented by the derived classes which represent the different venues/views. CLocationMap also implements some general methods as it will be presented in
the following paragraphs. In our application we used one venue (the TFC stadium) and six different views (four perspective views, a front view and a top view).

The methods implemented by the CLocationMap are:

- **LoadBitmapL (const TFileName &aFileName):** this is a protected method that can be used only by derived classes to load a bitmap file to be used as a view. The CLocationMap class is derived from CActive and uses the class CImageConverter, given by the Symbian API, to convert different image file formats (.jpg, .gif, etc...) into the Symbian specific format (MultiBitmap files)
- **void SetMapParameters ():** this function is called after successfully loading an image to set the image parameters (width, height)
- **TRect GetBitmapRect ():** this function returns the current used rectangle from the image. This rectangle is used to display a specific part of the image into the device screen. This allows for zoom in/out and scrolls left/right/up/down thru the functions ZoomIn(), ZoomOut(), MoveLeft(), MoveRight(), MoveUp() and MoveDown()
- **TCoordinate GetPositionCoordinates() and TCoordinate GetTargetCoordinates():** these functions are used to retrieve the stored GPS coordinates of current and target positions. As presented in the introduction, the LocationMap module is part of the Application View which in turn communicates with the Network Engine to get position updates. In this way we can always use the last position when displaying it to the screen
- **void SetNewPosition(TPosition &aPosition):** is used to set a new position when an update is received from the Network Engine
- **void CenterRectangle():** this function centers the current rectangle in the place where the current position is displayed
- **TPoint ConvertMapToRectangle(const TPoint &aPoint):** this function is used to convert a coordinate relative to the image into a coordinate relative to the rectangle size

Any class that is derived from CLocationMap and wants to be used by the application must implement the following virtual methods:

- **void LoadL(const TFileName &aPath, const TInt aID):** This function should load all the information about the venue/view; aPath is the directory where the files used by the application reside; aID is an ID given to each view. This ID is used by the Application View to change views and load them
- **TPoint ConvertCoordToMap(const TCoordinate &aCoord):** the derived class is responsible for implementing the transformation between GPS coordinates and pixel coordinates on the loaded image. In our application, as we will see in the next sections, we use two conversions, one between the GPS coordinates and real 3D coordinates of the stadium (implemented in the
CToulouseMap class) and another conversion between 3D coordinates and 2D coordinates for each view of the stadium

- TInt GetNearPointID(TCoordinate aPoint): for a given GPS coordinate this function returns the ID of closest point in the algorithm used to compute the shortest path between current and target positions. The points and the IDs will be explained later.

- void RecomputePath(TCoordinate aTarget): recomputes the shortest path for a new target. This is very useful if we want to change our target, e.g. to find another friend.

- void GetPath(RArray<TPoint> *aPath, TInt aStart, TInt aEnd): this function is used to return the shortest path (computed by a call to RecomputePath) between the points with IDs aStart and aEnd. aEnd should be the target point as the algorithm was run for that point. The result is an array (class RArray in Symbian C++) of points, each object containing the pixel coordinate of the point in the image. Using these points we can draw the shortest path between current and target positions.

### Stadium model

In our application we defined a class named CToulouseMap that extends the CLocationMap class. This class doesn’t implement all the virtual methods from CLocationMap but instead it also defines new virtual methods that must be implemented by the different views that derive from this class. We used this approach to use the 3D stadium model. As the methods presented will show, using this method we can model the way we want to handle the different views and the conversions that must be made to get from GPS coordinates to pixel coordinates using the 3D model of the stadium.

In the class diagram presented in *Error! Reference source not found.*, we can observe our implementation for the LocationMap module. As we explained before, CToulouseMap doesn’t implement all the virtual functions from CLocationMap and it also defines new virtual methods, thus it cannot be initialized. Instead, derived classes (TFCViewA ... TFCViewF) from CToulouseMap must implement all the unimplemented functions from CLocationMap and the virtual methods defined in CToulouseMap.
Figure 16

CLocationMap

public:
  // implemented functions
  ....
  // virtual functions
  void Load(TFileName) = 0
  ConvertCoordToMap(TCoordinate) = 0
  GetNearPoint3D(TCoordinate) = 0
  RecomputePath(TCoordinate) = 0
  GetPath(BArray<TPoint>) = 0

CToulouseMap

public:
  GetNearPoint1D(TCoordinate)
  RecomputePath(TCoordinate)
  GetPath(BArray<TPoint>), ....

protected:
  // implemented
  LoadVenueInfo(TFileName)
  ConvertCoordToPoint3D(TCoordinate)
  GetNearPoint1D(TReal3DPoint)
  RecomputePath(TReal3DPoint)
  GetPath(BArray<TReal3DPoint>), ....

  // virtual
  Convert3DToMap(TReal3DPoint) = 0

private:
  LoadTarget(TFileName)
  LoadMapInfo(TFileName)
  LoadShortPathInfo(TFileName)
  ComputeShortestPath(TInt)

CTFCViewX (X = A...F)

public:
  Load(TFileName)
  ConvertCoordToMap(TCoordinate)

protected:
  Convert3DToMap(TReal3DPoint)
Now we present a description of the functions used in CToulouseMap:

- **TInt GetNearPointID(TCoordinate aPoint)**: implements the virtual method from base class. First, this function converts a GPS coordinate into a 3D coordinate that matches the 3D model of the stadium (conversion is explained in next section) and then it finds the closest 3D point that was defined for the shortest path algorithm. For this we just use the norm between two 3D points: $D = \sqrt{(x1-x2)^2 + (y1-y2)^2 + (z1-z2)^2}$

- **void RecomputePath(TCoordinate aTarget)**: implementation of virtual method. This function converts the GPS coordinate into 3D coordinate and then calls the ComputeShortestPath method.

- **void GetPath(RArray<TPoint> *aPath, TInt aStart, TInt aEnd)**: implementation of virtual method. This function returns the shortest path that has been computed with the ComputeShortestPath function between two points. The end point should be the target which has been used for the computation of the shortest path as we are using the Dijkstra algorithm which computes all the shortest paths for a given node (the target in our case). To return the shortest path we make use of a iParents matrix where we store the parent for each link between 2 nodes (see Dijkstra algorithm).

- **void LoadVenueInfoL(const TFileName &aPath)**: this function is used to load all the necessary information about the stadium, target position and image file. This function must be called by the derived classes in the LoadL function. It calls LoadTargetL, LoadMapInfoL and LoadShortPathInfoL.

- **virtual TPoint Convert3DToMap(const TReal3DPoint &aPoint)**: this is a virtual method that must be implemented by derived classes (the different views in our application). In this way we force our derived classes to implement a method that converts between our 3D coordinates to the 2D pixel coordinates of the image used for the respective view. This conversion is actually a projection as it will be explained later.

- **TReal3DPoint ConvertCoordToPoint3D(const TCoordinate &aCoord)**: this method implements the conversion between a GPS coordinate and a 3D point coordinate that matches the 3D stadium model. This conversion is explained in detail in the following sections.

- **void LoadTargetL(const TFileName &aTargetDir)**: this function reads the GPS coordinates of our initial target from a file. The file name is known for each venue, in our case the TFC stadium.

- **void LoadMapInfoL(const TFileName &aPath)**: this function loads the GPS parameters of our venue from a file. The parameters read are the maximum and minimum latitude and longitude.

- **void LoadShortPathInfoL(const TFileName &aPath)**: this function loads the graph information for a set of known points of the stadium from a file. We used the following format for the file:

  N
  id(1)  x1  y1  z1  V  id1  c1  id2  c2  ...  idV  cV
  ...


id(N)  xN  yN  zN  V  id1  c1  id2  c2  ...  idN  cN

N represents the number of points used in the shortest path algorithm. N-1 is the last valid ID for a point (this is the ID used in the functions like GetNearPointID). Each of the following lines has the id of the point in that line which is useful if we want to add a point or remove another at a given point, especially if we have a big file. Following the id of the point there is the position of that point (x, y, z) in the stadium, and then the number of connections (V) and the IDs and costs for these connections. The cost is used for the computation of the shortest path and we consider cost[a][b] = cost[b][a]. The graph is stored in a matrix as we have an acceptable number of points (around 200) and we want a fast access to information. If we would use lists the access will take longer time. Our phone has big amount of heap memory and we didn't experience any problem with this approach

- void ComputeShortestPath(TInt aStart): this is the method used to compute the shortest path between the given ID (aStart) and all the other points. The ID used in our application is normally the ID of the closest point to the target. This permits to obtain the shortest path between any point and the target without any recalculation, we just have to find the path between the target and the current point and reverse it

Conversion between GPS and 3D model coordinates

In this section we present the algorithm used to convert between GPS coordinates received from the Network Engine and 3D coordinates of the stadium model used for the different views. Having the 3D coordinates of a point in the stadium model is very efficient as we can compute the shortest path algorithm for this model and then we can project it in any view of that model without any other computation besides a projection.

This approach is an exact approach that uses 2 conversions, first from WGS84 geodetic system (latitude, longitude and altitude) to Earth Centered Earth Fixed(ECEF) coordinates and then from this system to local East, North, Up (ENU) coordinates that match a plane tangent to earth’s surface. From this coordinates we apply a rotation and scale to match exactly our 3D model.

WGS84 to ECEF

We will use the following symbols for latitude, longitude and altitude:
- latitude: F
- longitude: ?x
- altitude: h

To make this conversion we first need to define some constants for our system (WGS84):

Semi-major axis ?ü a = 6378137.0 m
first eccentricity squared \( e^2 = 0.00669437999 \)
\[ R = \frac{a}{\sqrt{1 - e^2 \sin(F^4)^2}} \]

With this conventions we can compute \( x, y \) and \( z \) in the ECEF system like this:
\[
\begin{align*}
    x &= (R + h) \cdot \cos(F) \cdot \cos(\phi) \\
    y &= (R + h) \cdot \cos(F) \cdot \sin(\phi) \\
    z &= (R + h - e^2 R) \cdot \sin(F)
\end{align*}
\]

**Figure 17**

**ECEF to ENU**

We'll keep the same notations for latitude, longitude and altitude. In order to convert to local east, north, up (ENU) we need a reference point on the plane tangent to the earth (see Figure 17). For this we transform the WGS84 coordinates of the center of our plane (in my application the center of the stadium) to ECEF. We'll call these coordinates \( XR, YR \) and \( ZR \) (on axis \( X, Y \) and respectively \( Z \)).
After converting a point P from WGS84 to ECEF we have a point EP with coordinates XP, YP and ZP on axis X, Y and respectively Z. Having XP, YP, ZP, XR, YR, ZR, latitude(F) and longitude(?) (of point P) we can compute the local ENU coordinates like this:

\[
x = -\sin(?) \cdot (XP - XR) + \cos(?) \cdot (YP - YR) + 0 \cdot (ZP - ZR)
\]

\[
y = -\sin(F) \cdot \cos(?) \cdot (XP - XR) - \sin(F) \cdot \sin(?) \cdot (YP - YR) + \cos(F) \cdot (ZP - ZR)
\]

\[
z = \cos(F) \cdot \cos(?) \cdot (XP - XR) + \cos(F) \cdot \sin(?) \cdot (YP - YR) + \sin(F) \cdot (ZP - ZR)
\]

**From ENU to 3D model**

In our model x and y axis are not the same as east and north axis used by ENU model. Also the 3D model of the stadium doesn’t have the exactly same size as the real stadium that we use to get coordinates (longitude, latitude, altitude). For this reason we must make a rotation and a scale for each point in ENU to match exactly the point in the 3D model. Here are the steps to follow:

1. Find the rotation angle needed to match Y axis of stadium with north axis of tangent plane on ENU. We call this α.
2. Create rotation matrix \( R = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \)
3. Find scale factor on X and Y axis. Let’s call them Sx and Sy, where:
   \( Sx = \text{(3D model stadium width / real stadium width)} \)
   \( Sy = \text{(3D model stadium height / real stadium height)} \)
4. Create scale matrix \( S = \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix} \)
5. For each point P (XP, YP, ZP) in the ENU model rotate and scale:
   \[ [xr; yr] = R \cdot [XP; YP] \]
   \[ [xs; ys] = S \cdot [xr; yr] \]

Final coordinates on stadium model are: [xs, ys, ZP].

Another way of converting between GPS and 3D coordinates is using an affine transformation I we have a set of known correspondences between GPS and 3D. But this method requires the computation of the affine transformation and the knowledge of the correspondence points while giving a worse result. We think the presented method is very appropriate.

**Stadium Views**

As it can be seen in Figure 16, each view has its own class that is derived from CToulouseMap. Each view is responsible for the transformation between 3D to 2D coordinates. Also, each view has to load its specific image from a file, implementing the LoadL function. In all the views this means calling the method LoadVenueInfoL from the base class, passing as a parameter the name of the file where this view is stored. The conversion between 3D to 2D coordinates is explained in the next section. All we
need is the projection matrix that can be obtained from the parameters of the camera that has been used to make the view from the 3D model.

Conversion between 3D and 2D coordinates

In order to convert from 3D point(x,y,z) to 2D point(u,v) we use a mathematical approach:

\[
\begin{bmatrix}
  si*ui \\
  si*vi \\
  si
\end{bmatrix}
= \begin{bmatrix}
  P_{11} & P_{12} & P_{13} & P_{14} \\
  P_{21} & P_{22} & P_{23} & P_{24} \\
  P_{31} & P_{32} & P_{33} & P_{34}
\end{bmatrix}
\times
\begin{bmatrix}
  xi \\
  yi \\
  zi \\
  1
\end{bmatrix}
\]

where:

• ui, vi are the 2D pixel coordinates; if si is a non-zero scale factor, then ui = (si*ui) / si
• P11 to P34 are the parameters of the projection matrix (P) that we want to find
• (xi, yi, zi) are the coordinates from the 3D model

We can reconstruct the P matrix (P11 – P34) using the camera properties, obtained from the 3D model used to create the view for which we need the projection matrix (P).

We need:

• Camera center (XC, YC, ZC) : where the center of the camera is located in the 3D model
• Camera target (XT, YT, ZT) : where the camera is pointing at in the 3D model
• Camera horizontal FOV (field of view) (fx): the aperture (in radians) of the camera on horizontal
• Camera vertical FOV (fy): the aperture (in radians) of the camera on vertical axis (relative to camera)
• Image(2D) width (w): size in pixels of the image's width
• Image(2D) height (h): size in pixels of the image's height
C = [XC, YC, ZC];
T = [XT, YT, ZT];
CT = T - C; camera direction vector
NCT = norm(CT); normal of CT (that is sqrt((XT-XC)^2 + (YT-YC)^2 + (ZT-ZC)^2)
zcam = CT / NCT; direction vector normalized
Z = [0, 0, 1];
xicam = cross(Z, zcam);
ycam = xcam / norm(xcam);
R = [xicam; ycam; zcam]; R should be an orthogonal matrix (R*Rt = I, det(R) = 1); rotation
t = -(R*Ct); where Ct is C transposed (as a column vector instead of line vector); translation
ax = w / (2 * NCT * tan(fx/2)); factor ax is number of pixels in image per meter in the 3D model on X axis
ay = h / (2 * NCT * tan(fy/2)); factor ay is number of pixels in image per meter in the 3D model on Y axis
K = [ax * NCT, 0, w/2; 0, ay * NCT, h/2; 0, 0, 1]; Camera internal parameters
P = K * [R, t]; This is the projection matrix that we needed

Computing ax (for ay use h and fy instead of w and fx):

\[ \tan(\frac{fx}{2}) = \frac{L}{NCT} \quad (1) \]
\[ ax = \frac{w}{\text{width (pixels per meter)}} = \frac{w}{2 \times L} \quad (2) \]
\[ (1), (2) \Rightarrow ax = \frac{w}{2 \times NCT \times \tan(\frac{fx}{2})} \]

We must only project points that are in front of the camera. For this reason we must assure that points behind the camera won't be projected (even if they would be if we multiply by P). To put this condition we compute \( s = P(3,:) \times [Tt, 1] \), where \( P(3,:) \) is the last row of \( P \) and \( Tt \) is the column vector of \( T \) (Tt
means $T$ transposed). All points in front of the camera will have the scale factor $(s_i)$ with the same sign as $s$.

**Application UI**

The Application UI, implemented in our project by the class CLocationInfoAppUi, is the Controller of the application. It is responsible for handling user commands (key events), passing them to the Application View and also it has to handle events from the Application View (which in turns handles events from Location Map) and from the Network Engine. The Application UI is also in charge of the communication between the Network Engine (Model) and the Application View (View). They can also communicate directly but it is better if the Controller handles all the communications.

The CLocationInfoAppUi is also the class that creates all the controls (Canvas, Forms and Menus) used in the application. In the User Case section we will explain all the controls and their functionality.

**Deployment and Use Case**

**Signing and building**

For the application to run on the phone it must be signed as it uses several capabilities (Location, User Data, etc...) and the Symbian OS 9.2 doesn’t allow applications that are not signed to run on the mobile device.

To sign a Symbian application we need a certificate. This certificate is given by Symbian and can be either a Developer Certificate (which we use) or an Enterprise Certificate (which is obtained after the application is ready for production and has been submitted for testing to Symbian). More details about Symbian signed can be found on their website:

[http://www.symbiansigned.com](http://www.symbiansigned.com)

Using our IDE (Carbide C++ v1.3) we can build and sign our application at the same time. There are two possibilities for signing the application using a developer certificate:

1. **Self signing the application.** This means that we use a custom certificate made by carbide and then we send the application to the Symbian website in order to obtain the signed application. The problem with this solution is that each time we make a modification to the project and we recompile it we need to send the application to the website in order to get it signed. This is the only cost-free solution

2. **Signing the application using a developer certificate.** To obtain a developer certificate we first had to buy a publisher ID and then go to Symbian website and order a certificate. With this developer certificate we can sign our application and install it on the phone
Installation

After successfully building our application we obtained a .sisx file. To install this file (the application) to the phone we just have to double-click on while having Nokia PC suite installed and the phone connected. A message will popup asking if we want to install the application. If we click “Yes” the application will be transferred to the phone and we can finish the installation on the device.

Use Case

In this section we will explain how to use the application.

The following image shows the interface of the application which we will discuss in the next paragraphs:
Display Information

In the bottom left of the screen there are displayed the latitude and longitude of the current position or “Position Unknown” if there is no known current position (e.g. you don't have GPS signal) as well as the latitude and longitude of the current target (destination).

In the bottom right part of the screen there is a message “AutoCenter” which can be set to “ON” or “OFF”. If it is “ON” then the image will automatically center on the current position each time the application receives an update on the current GPS position.

The upper part of the screen (80% of the total screen) displays the selected view, position and path between current and target position.

Mobile guide commands

Using the different keys of the phone we can interact with the application by change views, set the AutoCenter parameter or moving the viewing position as it is explained here.

Change views

To select between the different views press keys 1 to 6. Each key (1, 2, 3, 4, 5 or 6) is used to select that view. If it is the first time we select a view, it will start loading and an information note will pop up saying that the application is loading the view. When the view has been loaded it will be displayed.

The possible views are presented in this image:
Change AutoCenter option
To switch on/off the AutoCenter option you can press the “0” key. The command will also be visible in the bottom-right part of the screen where the new value of AutoCenter should appear.

Zoom
To zoom in or out in any view press keys “*” (zoom in) or “#” (zoom out).

Scroll Image
To scroll left, right, top or down the visible image use the joystick on the phone (the rectangle in the middle, just below the screen).

Network and logging commands
The application also gives the possibility to retrieve information about GSM, GPS and WLAN as well as the possibility to add a new entry in the log file (if the logger was enabled at compilation time).
To access all this information, select Options from the bottom-left menu, using the left button as shown in the following image:

The menu displayed after pressing the Options button has the following options:

- Cell ID: the information about current GSM cell is displayed in XML format
- WLAN APs: information about current wireless networks is displayed, also in XML format
- GPS Position: gives the last position obtained from the GPS receiver
- Help: shows the help message of the application, explaining how to use it
- About: shows the about dialog
- Exit: used to exit the application. This can be done also using the bottom-right option by pressing the right button in the main screen
- Log Position: use this option to create a new entry in the log file with the current position. When pressing this button a new form appears with the following options:
  - Log Text: this is the text that will appear as an attribute in the &lt;Information&gt; tag of the log file. Enter the desired text here using the Options-&gt;Edit option
Test and evaluation

In this section we describe the methods used to test our application and the observed results.

GPS coordinate test

It is easy to test a static application on a PC, that only needs data from the PC or from the internet, but for us it was a hard work to test the application with real coordinates as we didn't afford to go to the real stadium (to get real coordinates) each time we wanted to obtain its coordinates and test the application. For this reason we used Google Earth, which provides accurate GPS positions all over the world.

In order to use the GPS receiver for testing, while keeping our real position in the institute (we didn't have to travel to the TFC stadium and back) we overlaid an image on Google Earth like the one in the following image:
For our application we always needed two known GPS positions of the stadium, in order to convert between GPS and 3D coordinates as explained in previous sections. Using the overlay approach we could rebuild our application in order to make it function anywhere.

We did test our application and we retrieved a number of points from the actual TFC stadium. In the next table we present the difference between the recorder position at the stadium and the GPS position obtained from Google Earth for the same points:

<table>
<thead>
<tr>
<th>Point (latitude, Longitude)</th>
<th>Point1</th>
<th>Point2</th>
<th>Point3</th>
<th>Point4</th>
<th>Point5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Earth</td>
<td>43.583763</td>
<td>43.583380</td>
<td>43.583294</td>
<td>43.583194</td>
<td>43.582836</td>
</tr>
<tr>
<td></td>
<td>1.4336111</td>
<td>1.4348000</td>
<td>1.4340472</td>
<td>1.4332916</td>
<td>1.4344861</td>
</tr>
<tr>
<td>GPS receiver</td>
<td>43.583702</td>
<td>43.583336</td>
<td>43.583240</td>
<td>43.583249</td>
<td>43.582868</td>
</tr>
<tr>
<td></td>
<td>1.4336089</td>
<td>1.4347252</td>
<td>1.4340170</td>
<td>1.4333846</td>
<td>1.4344429</td>
</tr>
</tbody>
</table>
The points used are presented in the following figure:

![Figure 25](image)

As we can observe from Figure 24 the maximum error is 0.00007 which is a very good result that proves two things:

1. Google Earths can be used as a trusted source for GPS positions all over the world
2. The GPS receiver on the mobile device gives a position accurate enough for positioning in the stadium field. Some problems have been detected on the tribunes, as they are covered and the GPS signal is weaker, resulting in worse accuracy

**Logger test**

We tested the logger module of our application in the stadium as well as in the center of Toulouse. We logged positions automatically as we walked through a determined path (see image below) and we also recorded the information at various known points. This information has been used by a company to create an application that uses GSM, GPS and WLAN localization in order to locate the user in Toulouse center with the high accuracy possible. The fact that our data has been used for a commercial application proves that our application provides good results and these results can be used in different
applications. In the following image we present the path that we followed in Toulouse center to register GSM, WLAN and GPS information using our logger:

![Image of a map showing the path followed in Toulouse center to register GSM, WLAN and GPS information using a logger.]

The resulted log file has 500 KB and it recorded over 500 positions. A screenshot of this XML file can be observed in Figure 14.

**Coordinate to Image coordinate test**

In order to test the precision of our method to convert the GPS coordinates into the different views as pixel coordinates we used two different approaches, a mathematical approach and a visual approach.

**Mathematic Testing**

To test our method by examining the resulting values of the transformations we used Matlab. This powerful tool was used also to compute the projection matrixes and the rotation and scale parameters needed for the conversion between GPS and 3D.
The absolute error for the entire conversion (GPS to 3D and 3D to 2D) was of 2 pixels, which represents an excellent result, as only the GPS receiver introduces a bigger error. The accuracy of the GPS receiver is between 2 and 40 m, and the stadium has 110 m in length. That means that an error of 2 m will be reflected in our images as an error of more than 30 pixels (our images have a width of 2000 pixels). This also means that the application should be improved in the future and we should use other methods for better accuracy (e.g. WLAN which gives accuracy of less than one meter).

The following Matlab script was used to create the projection matrix for any view:

```matlab
function P = TFC_Projection(C, T, W, H, fovX, fovY)
% P = TFC_Projection(C, T, W, H, fovX, fovY)
% This function returns the projection matrix used to project a point
% from the 3D world view in TFC stadium into a 2D point in the image
%
% C - camera center (x,y,z)
% T - camera target (x,y,z)
% W - image width (pixels)
% H - image height (pixels)
% fovX - horizontal FOV (degree)
% fovY - vertical FOV (degree)
CT = T - C;
NCT = norm(CT);
zcam = CT / NCT
Z = [0, 0, 1];
xcam = cross(Z, zcam)
xcam = xcam / norm(xcam)
ycam = cross(zcam, xcam)
R = [xcam; ycam; zcam]
t = -(R*C')
fx2 = fovX / 2;
fxr = fx2 * pi / 180;
fy2 = fovY / 2;
fyr = fy2 * pi / 180;
ax = W / (2*NCT*tan(fxr))
ay = H / (2*NCT*tan(fyr))
K = [ax * NCT, 0, W/2; 0, ay * NCT, H/2; 0, 0, 1]
P = K * [R, t]
```

Visual Testing

After proving that our conversions were reliable we implemented them in the application and then we compared the results for different views. As it can be seen in the next images, the same path between two points is projected well in all the perspectives, offering a 3D-like perspective to the mobile user:
This proves that our application works and the images and positions displayed are correct.

**Conclusion**

In this project we aimed at creating a solution for localization of the user using the mobile device and LBS in the form of WLAN and GSM localization as well as GPS positioning. In the middle of the project we took a small turn and focused on displaying rich visual content on the mobile device in order to give a better perception of the position to the user. Finally we obtained a mobile guide application, focused on the TFC football stadium, which uses different images taken from 3D models. This application can guide a client from any point of the stadium to his seat by showing him his position, the target position and the exact path that he must follow in the stadium, taking into consideration the possible paths to walk and the fences inside the stadium.

The resulting application provides good results and can be used very well in the real venue. Some things should be improved in order to provide a fully functional application. Among them we list:

- Position accuracy: WLAN positioning or other type of positioning system should be used together with GPS to increase the accuracy of the application
- Dynamic target: in our application we used a static target for the purpose of demonstrating the functionality of the application. In a commercial implementation this should be changed and maybe the application should communicate with a server in order to get the position of our target
- Friend Finder: the application could add a feature which allows for two friends to find in a stadium. The modifications are not big, is it enough that one of the friends send a message (SMS or over internet) to the other with his position and the second one sets the target to the first

I would like to thank the following persons for their valuable contributions in this project:

- Prof. N. Tăpus from the University “Politehnica” of Bucharest for all the help in administrative and decisional problems and also for giving me the opportunity of doing this project in the INPT
- Prof. R. Grigoras, Prof. V. Charvillat, Prof. Pierre and dr. C. Dehais from the Institute National Polytechnique de Toulouse for all the support, help with mathematical problems and their ideas over the entire period as well as for the guidance provided
- Romuald Rouhier, Olivier Maffrand and Damien Brasset from Immersive-Solutions for all the help involved in the 3D model of TFC stadium
References


III. T. Hastie and Friedman. The Elements of Statistical Learning. Springer-Verlag, 2001


ACM


VIII. J. Zhu. Conversion of earth-centered earth-fixed coordinates to geodetic coordinates. Aerospace and

IX. Electronic Syst., IEEE Trans. on, 30(3), 1994

X. http://www.forum.nokia.com/

XI. http://www.navcen.uscg.gov/gps/default.htm

XII. http://www.astro.uni.torun.pl/~kb/Papers/ASS/Geod-ASS.htm