Virtual Paintball Game
implemented using the symbiosis
between Java and AmbientTalk

Submitted in partial for the Final Project Degree
Hèctor López Escobar

Promotor: Prof. Dr. Theo D’Hondt
Advisors: Elisa Gonzalez Boix
Dr. Jessie Dedecker

March 2008
Abstract

This report introduces Virtual Paintball Game, a social game for mobile devices with a large-scale location system, situated in the novel gameplay field of location-aware games. Our project combines many key research themes such as mobile social gaming, mobile ad hoc networks and the impact of ambient-oriented programming languages when working with embedded systems. The observations and the results of trialling Virtual Paintball Game in the real world are presented in this report.
Dedication

I would like to dedicate this project to my grandparents Pedro and Isabel. They started life from nothing after the Spanish Civil War, and if they were alive today they would be very proud of me for making one of their dreams come true.

I still think of you every day...
Acknowledgements

With these lines I would like to express my gratitude towards all the people who helped me in many different ways to make this project possible. I would never have completed this report without their support.

Prof. Dr. Theo D’Hondt, for promoting this project.

My advisors Elisa Gonzalez and Dr. Jessie Dedecker, for giving me the opportunity to realize this first AmbientTalk game and guiding me through the project. Special thanks to Elisa on one hand, for commenting on the text, and Jessie on the other, for helping me every time he could.

My student room neighbors Matthias Stevens and Yves Vandriessche for their last-minute comments and suggestions on the text and for helping me in everything they could on one hand, and everyone in general from the Programming Technology Lab for providing me a perfect and comfortable working environment on the other.

My parents, German and Maria Isabel, for giving me the opportunity to go to study Computer Engineering to the Universitat Politecnica de Catalunya of Barcelona and also for giving me their support and encouragement to go the Vrije Universiteit Brussel of Brussels to make this project possible. You always gave me the support I needed to continue with everything.

My best friend Marta, for sharing her life with me and staying almost always next to me from day to day no matter if times were good or not.

My closest friends from Reus and Barcelona and all the people I’ve lived with during all my stay in Barcelona as a student: Carles, Diana, Joe, Joan, Mesqui, Nestor, Ruben, Roberto, Imma, Gerard, Mireia, for helping me to be the way I am and listening to me whenever I had the need.

Dani, Ori, Elena, Sergi, Marisa, Maite, Pep, Nadia, Rai, Pere, Oscar, Laia, Luque, Laura, Xavi, Raquel, and all my other friends from the Facultat d’Informatica de Barcelona, for sharing uncountable hours of good times and studying at the library.

Ainara, Closca, Eloi, Annalisa, Valentina, Judith, Cayetana, Gilda, Laura, Marco, Veronica, Nikki, Patrizio, Betty, and all the other great erasmus people I’ve meet here in Brussels during my stay, for helping me to not think only about work during the first months...
Contents

1 Introduction ......................... 1
  1.1 Context ................................ 1
  1.2 Motivation: Gaming at the campus .... 4
     1.2.1 Idea ................................ 4
     1.2.2 Motivation ............................ 4
  1.3 Objectives ........................... 5
  1.4 Methodology ......................... 5

2 Ambient-Oriented Programming in AmbientTalk .......................... 6
  2.1 AmbientTalk Background .............. 6
  2.2 Running Example ..................... 7
     2.2.1 AmbientTalk Variables, Functions and Objects ... 8
     2.2.2 Concurrent and Distributed Programming in AmbientTalk .... 11
        2.2.2.1 Actors ................................ 11
        2.2.2.2 Far References ......................... 13
        2.2.2.3 Isolates ................................ 14
        2.2.2.4 Futures ................................ 15
     2.2.3 Exporting Objects and Service Discovery ............... 16

3 Virtual Paintball Game .................. 19
  3.1 Introduction ........................ 19
  3.2 Game-Play Overview .................. 20
  3.3 Design and Architecture .............. 21
     3.3.1 Functional Requirements .......... 23
     3.3.2 Non-Functional Requirements ........ 24
     3.3.3 Modules .............................. 24
        3.3.3.1 General Overview ................... 24
  3.4 Implementation ...................... 26
     3.4.1 Preview of the GPSSimulator and the Localizer in Java .... 27
        3.4.1.1 GPS Data Standard NMEA 0183 .......... 27
        3.4.1.2 GPS Client-Server Communication .......... 29
        3.4.1.3 gspDataManager module ............... 31
3.4.1.4 Player's Graphical User Interface .......................... 34
3.4.2 Localizer in AmbientTalk using the symbiosis with Java ........ 35
3.4.2.1 gamePlayer module in AmbientTalk ....................... 35
3.4.2.2 gameServer module in AmbientTalk ....................... 38
3.4.3 Future Implementation Work ................................. 40

4 Experiments and Results ........................................ 42
4.1 Deploying the game on a PDA .................................. 42
4.2 Reading Real GPS Coordinates from the PDA .................. 43
4.2.1 Handling GPS Data Through the Serial Port ............... 43
4.3 Difficulties in Deploying Applications on Mobile Devices ...... 45

5 Conclusions ....................................................... 47
5.1 Summary ....................................................... 47
5.2 Future Work .................................................... 48
5.3 Personal Valorization and Acquired Experience ............... 49
5.4 Conclusion ..................................................... 49

Bibliography ....................................................... 51
Chapter 1

Introduction

1.1 Context

Games have appeared in the history of humans since the dawn of the humanity. In excavations from very primitive periods (even earlier than 3,000 years B.C.) very simple games and toys have been found. In paintings, it's possible to see children in recreational activities, which makes us think about games like an essential activity of the human being as a learning exercise, as a rehearsal and improvement for later activities.

From birth, human beings are an inexhaustible source of activity: looking, touching, manipulating, experimenting, inventing, expressing, discovering, communicating, imagining, are game activities that generate pleasure and happiness in children. Playing means participating in an interpersonal situation where emotions, expressions, communication, movements and intellectual activity are present. The interests and the personal decisions are the facts that make the recreational activity become real. In this sense, nobody should be forced to play because this may cause the lost of the essence of playing.

Before computers existed, games where invented and played out in the physical world, using the spaces between real-world objects and physical objects themselves. Interactions in games before the computer game boom consisted on two basic elements: adaptation of humans to the physical world and communication between humans.

Video games differ from traditional ones because they introduce new technologies, where people interact with elements of a program. A video game can be defined as a game that involves interaction with a user interface to generate visual feedback on any type of display device [38]. These devices can vary from large computers to smaller mobile devices such as mobile phones or PDA's. Nowadays, computer games have become a dominating form of entertainment which moves a big market.
Computer games have some characteristics that distinguish them from traditional ones. They add a non-real factor impossible to find in real-world games, but, at the same time they tend to eliminate many of the social aspects of traditional game playing, turning it into an individualistic activity. Even in the non-individualistic game plays, computer games exchange the face-to-face human interaction for a computer screen. These are the main reasons why computer games are sometimes more badly considered than others that have been played in the streets for years.

Another newer genre of games, known as virtual games, combine computers with interactions with the real world by means of physical objects such as game boards. This new field of games has been an emerging genre during the last ten years, with the proliferation of mobile devices equipped with wireless communication technology. Some representative examples of this new genre of location-based games can be: *Pirates!*\(^1\) [3], an adventure game that takes place in a fantastic archipelago, where players play the role of captains of a ship and can explore islands, solve missions, gain goods and fight for treasures with other players; *La Mosca* [29], a chase game with three policy teams and an escaped gangster, that converts a real city into a fantasy world; *RealReplay* [35], a racing game that allows players to create race tracks, to share them with other players and run on them to beat each other scores; *The Shroud* [36], an incoming role playing game, where players initially will have to champion their own farming communities, to later become heroes, protectors of their local towns; or *Monopoly Live* [28], a game settled in London that represents a live version of the traditional one, allowing players to buy properties and place apartments and hotels.

One inspiration to develop these kind of games can be found in the research field of ubiquitous computing. As defined by Mark Weiser [21], ubiquitous computing corresponds to the vision of omnipresent computational resources, that are sensitive and responsive to the presence of people, made continuously available to people in their everyday lives. This domain is where the Ambient Intelligence (Aml) vision (recently termed this way by the European Council’s IST Advisory Group [12]) has been built on.

Ambient Intelligence represents a new vision of distributed computing that grew up parallel with the increasing availability of mobile networks and mobile devices equipped with wireless communication technology. The related technologies are expected to combine concepts of ubiquitous computing and intelligent systems, *putting humans in the centre of technological developments* [10] and facilitating their contact. There are a lot of terms implied within the key technologies required for Aml to become a reality: miniaturization, nanotechnology, smart devices, sensors, wireless networks, service-oriented architecture, interoperability, service discovery, intelligent agents, context awareness, etc. In spite of that, the Aml paradigm is characterized by systems (also called *Ambient Systems*) and technologies which, according to [38], are:

\(^1\) Not be confused with the Commodore 64 game Pirates!, copyright by Microprose.
- **Embedded.** Many networked devices are integrated into the environment.

- **Context-aware.** Devices can recognize people and their own situational context.

- **Personalized.** Devices can reach and correspond to concrete human needs.

- **Adaptive.** Devices can modify their states or behaviors in response to people's needs.

- **Anticipatory.** Devices can anticipate to human needs or wishes without conscious planning.

From a software engineering developing point of view and, according to [9] [10], the Aml vision highlines how hard it can be to develop software in ordinary programming languages when working with mobile networks. The main part of the existing programming languages lack abstractions to deal with the mobile hardware characteristics, what makes them treat remote communication failings with exception handling sentences more as a rule than as an exception. When users are surrounded by a wireless network, they can make it change its configuration as they move around with their devices. This generates a natural evolution from isolated applications into cooperative ones, to make them able to interact with their environment (other devices that are present in their surroundings).

To develop applications that run in these dynamically demarcated networks, we need these Ambient Systems to take into account the new network topology which differs from the traditional stationary one. As it is described in [6], the mobility of the devices, added to the characteristics of wireless networks, leads these kind of applications to work with software entities which spontaneously detect one another, engage in some collaborations, and may disappear in the same way they appeared.

The properties of mobile ad hoc networks (usually wireless connections) have a direct impact on software development of applications running on them. The following is a list of properties that, according to [6], distinguish mobile networks from other types of networks and, in one way or another, may affect the way of dealing with communication:

- **Volatile connections.** In mobile ad hoc networks we can not assume that every time entities start a cooperating task together they will have a stable connection available. Wireless connections are less reliable than the wired ones and they also have a limited communication range for the devices. This implies that disconnections, unexpected in the wired networks, are not unusual in the wireless ones. Frequently the network partitions should not affect the application, allowing the communication to resume when the connection is reestablished.

- **Zero infrastructure.** Zero infrastructure refers to the fact that devices have to collaborate without relying on a predetermined infrastructure such as a server, which provides naming service to discover other devices in the environment. Every entity is
supposed to be able to work as an autonomous computing unit. If any autonomous computing unit changes its location, it may temporally turn some resources in the environment into available or unavailable. For this reason, in contrast with stationary networks, these service resources have been denominated or labeled as ambient. Zero infrastructure advocates the user of peer-to-peer approaches. Without a higher pre-established role on the network, concurrency becomes a natural phenomenon in software for mobile networks.

To deal with the hardware phenomena at a programming level, the Ambient-Oriented Programming (AmOP) paradigm was proposed [9]. It gives a model for the design and implementation of high level programming technology. This technology facilitates the development of Ambient Systems, introducing network failure events in a very deep level of their semantic building blocks. AmbientTalk [6], the chosen language to develop this project, is an ambient-oriented programming language, that works with mobile ad hoc networks as a normal mode of operation, providing software developers with abstractions to deal with the characteristics of the mobile networks. Deeper details about the language will be given in the Chapter 2 of this report, while explaining the first experiment we did with it.

1.2 Motivation: Gaming at the campus

1.2.1 Idea

The main idea for this project is to develop a game merging the use of computers (or small handheld devices) and user interaction. The game will merge aspects of traditional games with new age ones, trying to bring the virtual computer entertainment back to the real world or, in other words, increasing real-world game amusement with computing functionalities. The game will allow multiple players to play simultaneously together through a wireless local area network, allowing them to move all around an outdoor delimited area. The game will also include the use of real world properties such as locations and states of co-location between players and virtual items.

1.2.2 Motivation

Our main motivation is to develop a game which integrates the hardware phenomena of mobile ad hoc networks explained in Section 1.1 in its design. To this end, we need to take care about aspects that are not usually analyzed, such as interactions triggered by the physical proximity of the players. The virtual game will be able to run on PDA’s and will be implemented using AmbientTalk, an AmOP language which provides us with language support to deal with the hardware phenomena of MANETs (mobile ad hoc networks).
1.3 Objectives

The goals of this project are twofold:

- **Gaming at the campus.** Building a virtual game to exploit the existing network facilities in the campus of the VUB.

- **MANETs restrictions.** Developing a game that takes into account the hardware phenomena of MANETs. This is the reason to use AmbientTalk as development platform to implement the game since it contains dedicated language constructs to deal with MANETs.

1.4 Methodology

To achieve this main goal, it is essential to achieve other smaller sub-objectives. For this reason the project has been structured in three very different parts that are going to lead us to our main goal through an iterative approach:

1. Learning the AmbientTalk language by developing of a first simple game application called *Paper-Scissors-Stone Game*. This application will be also used to explain the language in the Chapter 2 of this report.

2. In a second phase, we will develop the ambient game by means of an iterative approach. We will first develop the part that shows the position of other players in the VUB campus (called *Localizer*). This module will simulate all the GPS data transmissions that are usually done when working with a PDA with an incorporated GPS. The design and implementation of the virtual items is also included in this second sub-objective.

3. The last step is jumping from the Localizer and all the simulated GPS data to the real life, installing AmbientTalk on the PDA and migrating the application from the MAC Computer to the PDA.
Chapter 2

Ambient-Oriented Programming in AmbientTalk

The aim of this chapter is twofold. On one hand, we give an overview of the programming language used in this project, going from its basic building blocks to its advanced distribution model. On the other hand, we intend to structure the chapter so as to make it easily readable and useful to any person who might be getting started with AmbientTalk. In view of that, we introduce the language by means of code samples (extracted from our first experiment). The main reason to choose AmbientTalk to develop this project is that due to the abstractions the language provides, it decreases the complexity of interacting in mobile networks, simplifying the development of distributed applications.

2.1 AmbientTalk Background

As it is described in [6], AmbientTalk ¹ is a distributed object-oriented programming language especially thought to develop applications deployed on mobile networks. It treats network partitions as a normal mode of operation. The language is actor-based and follows an event-driven concurrency model. This, combined with his publishing/subscribe service discovery abstractions, makes it easier to compose service objects across a mobile network.

Although AmbientTalk can be considered as a domain-specific language for distributed programming, it is also a complete object-oriented language which inherits most of his language features from languages like Self, Scheme and Smalltalk. The language has true lexically scoped closures (inherited from Scheme); an expressive block closure syntax, representation of closures as objects and the use of block closures for the definition of control structures (inherited from Self and Smalltalk); and is classless, slot-based objects using delegation as a reuse mechanism (his object model has been derived from Self). [6]

¹ AmbientTalk refers to the version AmbientTalk/2, a successor of the first AmbientTalk language developed in 2005 by Jessie Dedecker (AmbientTalk/1).
2.2 Running Example

We will describe AmbientTalk by means of a simple example ad hoc networking game, called Paper-Scissors-Stone, that we developed with the purpose to get experienced with the language constructs and features. Our application is a decentralized version of this traditional game that allows multiple players to play this traditional game simultaneously.

The rules of the game are quite simple: in the traditional version, an open hand represents the paper; a closed fist, only showing the index and middle fingers extended and separated represents the scissors; and a closed fist represents the stone. The rules to determine winners and losers are the following:

- **Paper** wins stone because it can wrap it.
- **Scissors** win paper because they cut it.
- **Stone** wins scissors because it brakes them.

The oldest version of the game only allowed two players to play at the same time. In our ad hoc network version of the game we allow multiple players to play simultaneously together, assuming that the probabilities of having a draw are exponentially increased in proportion to the number of players. In this particular multiple player version of the game the scores are calculated as follows:

- In the case that all three possible figures appear between player’s choices, the game will be tied.
- If player’s choices are all between two of the three subsets of choice, winners and losers will be determined by the traditional way.

Our ad hoc network version of Paper-Scissors-Stone is a very simple command line based application which has been designed under a main assumption: all players have access to the same network and have always full visibility of each other. To deal with the fact of not having a server and developing a decentralized application, we created the concept of a session.

A session will be created every time two players discover each other and neither of them is already playing in a session or is engaging with another player to create one. To decide which of the two players has to create the session, we proposed the concept of a leader player. The fact of having a leader player is also useful to avoid having mismatches of information in our distributed system, because such player will be the only one able to update the status of the session.

When two players discover each other, if neither of them is already the leader, nor have they
joined an already existing session, they have to decide who is going to be the leader of the game. Once a leader has been found, he will start a session which all the later incoming players will eventually receive from him.

The game session is structured by rounds, which are composed by two different states: a starting state (where the results of the round are displayed) and a playing state (where players are asked to choose their options). The current leader of the game is the one who dictates the states of every round, using a timer.

Players are allowed to enter the game at any time but, depending on which state players apply to join in, they will immediately be able to play (if joining during the starting state) or they will have to wait until next round (if joining during the playing state). When players join the session, they are asked to choose a figure (i.e. paper, scissors or stone). After a certain time, when the state changes, they will get their result of the round and will be asked to choose their new option for next round.

Even though we took network failures into account when designing the game, we did not implement the case when the leader player is disconnected. By the way, the easiest way to deal with the problems such disconnection might generate, would be making the session able to determine a new leader player on the first round the current one stops playing.

The objective of this section is not to exhaustively explain how this small game has been implemented, but to try to explain the AmbientTalk features using code from our experiment. For this reason, the general explanation above (we have not introduced AmbientTalk features yet), combined with the code samples that are going to be shown, may not be enough information for the reader to understand the complete perspective of the whole implementation of the game, missing some functionalities or some logic parts of it. However, we bear in mind that our main goal for this section is to explain the AmbientTalk language features as good as possible.

### 2.2.1 AmbientTalk Variables, Functions and Objects

AmbientTalk is an object-based language built on a small grammar with a very simple expression language and a flexible syntax. As in many other functional languages, in AmbientTalk we can define variables, to later make assignments and references to them. Variable definitions are made with the keyword **def**:

```plaintext
def timeoutStart := nil;
def timeoutPlay := nil;
def state := nil;
def round := nil;
def leaderP := nil;
```

**Box 2.1:** Variables of the object PSSSession
Even though AmbientTalk has some native data types: numerical data types (numbers, fractions); texts (called strings in other languages); tables (also known as arrays), and booleans, variables will never have an initial type, but they can contain any value because AmbientTalk is a dynamically typed language. Values can be immediately or later assigned when declaring the variable.

An example on working with text native data types in AmbientTalk can be found in one of the methods of our experiment, used to decide which of the two players engaged in creating a session would be the leader. The following is a method that parses a text unique identifier (as it is the ID of a far reference – see subsection 2.2.2.2 –) returning a numerical data type:

```plaintext
def myremoteID := parseID(print: remoteFacade);

::
def parseID (objID)
{
  objID := objID.replace: "<far ref to:" by: { |el| "" };
  objID := objID.replace: "<object:" by: { |el| "" };
  objID := objID.replace: ">" by: { |el| "" };
  objID := objID.parseNumeric();
  objID;
}
```

**Box 2.2:** Parsing player's identifiers

By parsing the identifiers of both players and comparing them, players will be able to determine who has to be leader. The player with the higher objID will become the leader.

The language also supports multiple variable assignments. The only restriction is that we need to assure that the number of variables on the left side of the assignment token matches the number of variables or expressions on the right side:

```plaintext
def [future, resolver] := makeFuture();
```

**Box 2.3:** The method makeFuture() has to return two values

Rather than instantiating objects from classes, in AmbientTalk objects are created ex-nihilo or by cloning and adapting other existing objects. Computation works by sending messages from one object to another.

The following code sample shows a standard declaration of an ex-nihilo created object in AmbientTalk:

```plaintext
def InfoPlayer := object: {
  def name := nil;
  def info := nil;
  def init(n, inf) {
    name := n;
    info := inf;
  }
  def getName() {name};
}
```
In the code above an anonymous object \(^2\) is defined and assigned to the variable InfoPlayer. In a similar way as it was explained in [6], the object is a prototypical item object that defines two fields (known as the state of the object) to store the name and some information about a player and two functions (known as the behavior of the object) to get them. Functions in AmbientTalk are implemented as named closures (anonymous functions objects that contain some structured code). Block closure's syntax is the following: |arguments| body, where arguments are the arguments the closure receives as parameter and can be empty. The declared object can be instantiated to create new items in the following way:

```
def info := InfoPlayer.new(username, "scissors");
```

Box 2.5: Objects instantiation

As you can see in the previous example, AmbientTalk's object instantiation is similar to the usual instantiations in class-based languages. Every time the message `new` of an object is called, the `init` method is invoked with exactly the same arguments that were passed to `new` (as it is done with the constructors in class-based languages). The main difference between AmbientTalk and class-based languages instantiation is that in AmbientTalk an object is a clone of the prototypical object rather than a new object allocated from a class.

As AmbientTalk is a class-less language, there is a type tag system for categorizing objects (in class-based languages objects are usually tagged by the class itself). A type tag is a "label" (that works as identifier of the object to which it is attached) which can be defined (as shown in the Box 2.6 below) or assigned to an object (see the last line of Box 2.4):

```
def type PSSPlayer;
```

Box 2.6: Definition of a type tag

One object may be tagged with more than one tag and type tags themselves can also be a subtype of one or more other type tags.

Type tags can best be compared to empty Java interface types. Such empty interfaces are sometimes used in Java purely for the purposes of marking an object. Examples are `java.io.Serializable` and `java.lang.Cloneable`. An empty interface type can be implemented by any class (object) and hence only serves the purpose of distinguishing objects by type (by means of `instanceof` in Java) [24].

\(^2\) An anonymous object is an object which does not have a related name itself
After having explained how objects are created and instantiated in AmbientTalk, coming back to our game experiment, this may be a good point to introduce how to work with different modules in AmbientTalk, by explaining the use we did of the timer object.

```python
import ./at.support.timer;
...
def bootPSStimer() {
  def StartingCode() {
    state := 0;
    round := round + 1;
    leaderP.sendAlarm();
    when: timeoutStart elapsed: PlayingCode;
  };

  def PlayingCode() {
    state := 1;
    leaderP.sendAlarm();
    when: timeoutPlay elapsed: StartingCode;
  };
  when: timeoutStart elapsed: PlayingCode;
}
```

Box 2.7: Using the timer from the PSSSession

As we described in Section 2.2, the leader player creates a session. That session contains a timer which, after every certain time, tells (we will explain later how sentences such as when: elapse: work) the leader into which state is the round running. By this way, is how the leader is able to update the session every time a change of state is done. To work with an external module as it is the timer, we need to explicitly import it in the PSSSession lexical scope, as is shown in Box 2.7.

It is possible to see the code of the timer object downloading AmbientTalk from the official site (the object implementation can be found in the package ./at.support.timer).

### 2.2.2 Concurrent and Distributed Programming in AmbientTalk

#### 2.2.2.1 Actors

Rather than having a thread model, AmbientTalk implements concurrency by means of actors. AmbientTalk’s actors follow an event-driven concurrency model and are represented as event loops, which are threads of execution that continuously process events from their queues by invoking an event handler. As it is explained in [6], event loop concurrency avoids deadlocks and certain race conditions because of its concurrency control properties:

- **Serial Execution.** Incoming events are processed from its event queue in a strictly serial order.

- **Non-blocking Communication.** All the communication between event loops is done by asynchronous event notifications, avoiding the suspension of current executions to wait for another event loop to finish a computation.
• **Exclusive State Access.** Every event loop it's the only one that can access to its state.

Instead of an event queue, in AmbientTalk each actor has a message queue, where all the messages are stored. As each object can only be owned by one actor, it will be the only one able to execute its methods. Event notifications are traduced as asynchronous message sends and event handlers are represented as the methods of the objects.

In AmbientTalk two objects are denominated *local* when they are owned by the same actor and *remote* when they are owned by different actors. For this reason, as one virtual machine can host multiple actors executing concurrently, we can have remote objects within a single virtual machine even though we will never have partial failures.

![Diagram of local and remote objects]

**Figure 2.1:** *local* and *remote* objects

Local objects (owned by the same actor) can communicate using standard, sequential message passing or asynchronous message passing while remote ones (owned by different actors) can only communicate using asynchronous message passing, via *far references* (see subsection 2.2.2.2). AmbientTalk has a different syntax for both communication means:

```plaintext
remoteShell<-getSession();
```

**Box 2.8:** Asynchronous message passing

Coming back again to our experiment, as we previously explained, when one player discovers another, he will ask the other for its session, to know if he was already playing or not. The way how a player asks for the session to another one is by asynchronous message passing, as shown in Box 2.8.

```plaintext
if: (remoteID==parseID(print: remoteSession.getLeaderP())) then:
```

**Box 2.9:** Sequential message passing

Once that player receives the session, he will be able to call its methods locally, as is shown in Box 2.9. After knowing that a remote player already joined a session, players will be able
to identify if the remote player is the leader or not, in order to apply to join the session.

As remote objects can only communicate by sending asynchronous messages, the application’s control flow will never be blocked because such objects can never reach a deadlock or a blocking state, waiting for a remote method to finish.

The event loop threads of the actors continuously read the messages from their message queues and synchronously execute the corresponding methods on the actor’s owned objects. When communication with another object is required, a message is asynchronously sent via a far reference to the object, that enqueues it in its message queue.

In asynchronous message passing, when the sender and the receiver are both owned by the same actor, messages are just enqueued to the actor’s message queue and parameters are passed by reference (as is usually done with synchronous message sending). For asynchronous message passing between objects residing on different actors, objects and closures are parameter-passed by far reference, while native data types are always passed by copy.

2.2.2.2 Far References

As it has been explained in the previous section, it is possible for objects owned by an actor to refer to objects owned by other actors. The references that allow communication between different actors are named far references. Far references are a referencing model inherited from language E [18], which AmbientTalk uses for asynchronous communication both locally and remotely. Such references only allow asynchronous access to the referenced object.

Far references suitability to network failures is another of the properties that turns AmbientTalk’s distribution model into a perfect environment to deal with ad hoc networks partial failures. When a network failure takes place, all the far references to offline remote objects store all the messages that were sent to them, to later, when the network connection is restored, re-send them in the same exact order. Note that AmbientTalk doesn’t ensure that messages will be received in the same order at the other side.

The following are some code samples of our experiment showing all the asynchronous message that are sent in our application, that may draw a little bit about the distributed protocol of the game and the different remote references:

```plaintext
//PSSession
def broadcastChooseOption() { ... currentPlayer < choosePSSoption(round); ...};

def broadcastUpdateSession(s) { ... currentPlayer < sendSession(s); ... };

def broadcastGetResult() { ... currentPlayer < endOfRound(); ... };    

def removeSleepingPlayers() { ... currentPlayer < sendMessage("Timeout: You didn't throw in round: +(round-1)); ... ");

//PSSPlayer (leader)
remoteShell < sendSession(session);
```
2.2.2.3 Isolates

In order to reduce the number of asynchronous messages sends and to avoid wrong behaviors when network failures occur, we want players of our game scenario to be able to evaluate the result of every round locally by themselves. To achieve that purpose, we need to send them copies of the session every time it has been updated. This is what isolates allow us to do. Because of regular objects are passed by reference (it would be needed the whole lexical scope to pass them by copy), AmbientTalk introduced these objects called isolates, which are "isolated" from their lexical scope.

```
def PSSSession := isolate: {
  ...
  def PlayingList := jlobby.java.util.Vector.new();
  def WaitingList := jlobby.java.util.Vector.new();
  def PlayerThrowList := jlobby.java.util.Vector.new();
  ...
  def broadcastUpdateSession(s) {
    if: !(PlayingList.isEmpty()) then: {
      o.to: PlayingList.size() do: {
        i.index() def currentPlayer := PlayingList.elementAt(index);
        currentPlayer <-- sendSession(s);
      }
    }
   ...
  }
```

---

Box 2.11: PSSSession isolate

It is possible to create an isolate by means of tagging it with taggedAs: [./at.types.Isolate], as it has been done in Box 2.4, or with the primitive isolate: (which is the syntactic sugar way), as it has been done in the example above.

As shown in Box 2.11, players get the updated session by means of an asynchronous message sent by the session itself. As we previously explained, the leader is the only player that updates the session, every time the change into the starting state occurs:

```
if: (state==STARTING) then: {
  listener.display("Start State...");
  when: session.removeSleepingPlayers() becomes: {
    lack! session.broadcastUpdateSession(session);
    session.broadcastGetResult();
    session.addWaitingPlayers();
    session.broadcastUpdateSession(session);
  };
}
```

---

Box 2.12: Update of the session
During the update of the session, as shown in Box 2.12, the players that did not chose any option for the last round are moved from the PlayingList to the WaitingList shown in Box 2.11, with the method `removeSleepingPlayers()`. Players get the session with the updated lists just explained; they get the results of the last round; and they receive again the updated session, after the new incoming players that were waiting to join the game have been moved from the WaitingList to the PlayingList.

### 2.2.2.4 Futures

When using the futures module, asynchronous message sends do not return any value (nil), but they immediately return what is known as futures: objects that represent the future value of an asynchronous message sending. When the return value is computed and sent back, the future object is replaced by the current value (at this moment the future is said to be resolved).

In AmbientTalk, futures require to be imported because they are not native to the language. For this reason, we need to import the futures module into our lexical scope, as we did with the timer in the example of Box 2.7. We also need to explicitly enable future’s behavior in our code to work with them as follows:

```plaintext
import ./at.lang.futures;
...
enableFutures(true);
```

**Box 2.13: Importing and enabling futures**

As a future can be eventually resolved with a return value or ruined with an exception when the computation raises an exception, AmbientTalk allows expressions that work like event-handlers. These expressions are useful to control player states or behaviors, without having any possibility to reach a deadlock state (working in a distributed programming context we need to avoid the possibility that an actor can wait for a computation of another actor residing on a different virtual machine). Note that in AmbientTalk gives the possibility to put observers, which are triggered also asynchronously. Once again, AmbientTalk implements non-blocking futures as `E`.

To show how future event-handlers work in AmbientTalk we can return to our *Paper-Scissors-Stone* game again. Every time an actor without a session (a non-playing actor) discovers (see subsection 2.2.3) another one, he sends an asynchronous message asking for his session. With the `when:becomes:catch:` function, the actor asking for the session won’t be blocked waiting until the answer of the asynchronous message is back, but we will be able to specify what he has to do when the future is resolved:
Box 2.14: when: becomes: catch: function behavior

If the future is resolved to a valid value, the closure passed as an argument to the becomes: will be called with the resolved value as a parameter. If the method called asynchronously throws an exception, the catch: closure parameter will be applied. As it is explained in [24], this is how AmbientTalk, in a similar way that sequential languages deal with exceptions using the try and catch, provide applications to catch exceptions that were asynchronously thrown.

As shown in Box 2.14, any player will be able to play until he finds the leader, in order to apply to join the session (see the code of the closure of the second when: becomes:). The code in the closure of the first when: becomes: shows the only exception to that affirmation, where the co-leader (player who engaged with the leader in the creation of the session), automatically receives and joins the session without applying for it.

2.2.3 Exporting Objects and Service Discovery

To make objects reachable to remote actors, an actor can export objects to the network, by means of tags (previously explained in subsection 2.2.1). Type tags are used to give a description of the service the device can provide to other actors. As AmbientTalk service discovery is based on a publish/subscribe service discovery protocol, an export of an object by means of a type tag is known as a publication. As long as an object is published, it can be discovered by other objects searching the provided service, as well as it can be cancelled.

Coming back to our Paper-Scissors-Stone running example again, players need to discover each other to be able to communicate when they discover each other(Paper-Scissors-Stone
players all run the same code). For this reason, we created an object which is going to be published on the network by each player:

```python
def remoteFacade := object {  
  def getName() {username};  
  def getOption() {option};  
  def setOptionToNil() {option := nil};  
  def getSession() {session};

  def choosePCOption(round) {listener.displayPCSOptions(remoteFacade, round)};
  def sendSession(ss) {session := ss};
  def sendMessage(msg) {listener.display(msg)};
  def sendAlarm() { ... };  
  def joinSession(player) { ... };
  def endOfRound() {session.evaluateResult(remoteFacade, option)};
};
```

Box 2.15: Player’s facade to export

Once an object has been published on the network and it becomes discoverable by other object’s actors, remote objects can receive a reference to the exported object when a service object matching the required service is found. The correspondent syntax for publishing objects into the ad hoc network is the one that follows:

```python
defetype PSSPlayer;
...
export: remoteFacade as: PSSPlayer;
```

Box 2.16: Exporting an object by means of type tagging

In AmbientTalk, a subscription is known as the availability of an exported object in the mobile network. The following is the part of code that subscribes to a PSSPlayer type tag availability notification:

```python
whenever: PSSPlayer discovered:
  |remoteShell| def remoteID := parseID(print: remoteShell);
  ...
  remoteShell<=getSession()
  ...  
  );
```

Box 2.17: Discovering services across the network

The whenever:discovered: function returns a publication object to cancel the publication. The block of code in the discovery receives as parameter the far reference remoteShell. From this moment on, the player who discovered the publication object, will be able to communicate with the remote object as in the example above, sending asynchronous messages via the far reference.

In AmbientTalk network access is disabled by default. To enable network access and simulate connection and disconnection events, AmbientTalk provides the network object. The methods to enable/disable network access into an AmbientTalk virtual machine are: net-
work.online() and network.offline. To make actors able to discover all another’s presence and their discoverable objects in the environment, we require to add the following line before the last one in the code of our experiment:

```
... network.online();
self
```

**Box 2.18: Starting a network**

When the method online() is called, all discoverable objects (e.g. the object remoteFacade from our game scenario) are going to be exported to the environment for all the virtual machines temporarily found on there, because of the build in service discovery broadcasts their presence on the network. On the other hand, when the network method offline() is called, the virtual machine will be disconnected from the network, breaking all connections pre-established with other virtual machines.
Chapter 3

Virtual Paintball Game

In this chapter we are going to describe the design and implementation of the game we are going to develop. The chapter includes its full design and its detailed implementation strategy, together with some code samples.

3.1 Introduction

Our Virtual Paintball Game is a multiplayer game ment to be played on an outdoor area, concretely all around the Campus Etterbeek of the Vrije Universiteit Brussel. It is intended to use the wireless local area network ambienttalk, a private sub-net within Urbizone, a wireless network that can be freely used by students, personnel or visitors.

Our game basically consists on a finite game where multiple teams compete to achieve a common goal. The main objective of the game is to kill as many enemies as possible, trying to be in the survivor team. To achieve that objective, players can collect many different kinds of virtual items that are hidden all around the game area such as weapons or health packages, which become visible in their maps when they pass close by.

Players have PDA devices equipped with GPS receivers, where they can see their own locations and the one of their team partner’s as well, as they move about the campus.

The location of a player can be technically determined by many different ways (using proximity sensors, ultrasonic emitters, GPS satellite signals, etc.). e.g. Pirates! (a location-aware game already described in Chapter 1), was implemented using short-range radio frequency proximity-sensors, which makes positioning easier and allows to play indoors, but hardly in a very big area. For our game we decided to work with GPS signal reception, which allow us to play outdoors.
3.2 Game-Play Overview

To determine when an item has to become visible for a player we calculate the euclidean distance between player’s positions and the item’s one. Players have a discovery-radius which partly determinate whenever an item is visible for them or not. The center of the item requires to be in the area delimited by the discovery-radius of the player to become visible. Also virtual items have a pick-up-radius to determinate when an item is able to be picked up or not, and also the center of a player needs to be in the area delimited by the item’s pick-up-radius to be able to be picked up. The figure below illustrates such concepts:

![Figure 3.1: Player’s proximity-triggered actions with items](image)

When a virtual item is able to be picked up because of his co-location with a player, such player receives a notification. At this point, the player is able to pick up the item by double clicking on it. Players can consult their owned items and operate with them through their items panel.

Depending on the kind of virtual item, players will be able to use them for themselves, to interact with other players by switching them on, or just throw them to the floor to make them discoverable and available to be picked up again by other players. All items are classified as throwable (throwing an item doesn’t always mean switching on his functionality), but not all of them are transferable (exchangeable between players). The virtual items for the game are:

- **Health packages**(transferable item): A player can use a health package for himself or transfer it to another player. When health packages are used they totally fulfill the health of the player who uses it.

- **Mines**(transferable item): When one player throws a mine in a location, the virtual item will be still visible by other players and could be picked up again. If the mine is switched on by the player, the virtual item won’t be visible anymore and will be detonated when another player passes within a concrete radius (comparing the damage-radius of the item to the euclidean distance between the item and all the players).
• **Bombs** (non-transferable item): When a bomb is thrown, the virtual item is automatically switched on and becomes invisible for all the other players, meaning that it cannot be picked up again. After throwing the bomb, the owner can detonate it with a remote control, whenever he thinks it is useful. When the bomb is detonated it will hurt all players inside the explosion radius of the item.

• **Traps** (transferable item): As mines, traps can also be thrown to the floor or switched on. When a trap is switched on, it also becomes invisible for all the other players and cannot be picked up again. If a player passes close by the trap, within a concrete radius, he remains trapped in that location, until another player of his team moves there to rescue him. When a player gets trapped, virtual movements in the screen are temporarily disabled, forcing the player to stay there, in a short range area, waiting to be rescued. If he moves out of the range while trapped, he won't be able to continue playing until he comes back to the place where the trap was set.

![Health packages, mines, bombs and traps game icons.](image)

Figure 3.2: Health packages, mines, bombs and traps game icons.

For the items that require to be switched on (mines or traps) or are automatically activated (bombs) to interact with players, there are some restrictions on the proximity-triggering aspect. If a player switches on a virtual item, it will not hurt anybody who is inside the radius of the explosion immediately. There is an established time for the people who is inside the hurt-range to go out of this area. The people who is not inside the radius of explosion can make the item interact immediately going inside the hurt-range area.

When one player decides to throw one of his items to the floor, this item will appear on the screen of the other players of his team, without any proximity restriction. The players of the other teams can also pick up this virtual item, but the only difference is that they will have to pass close by to discover it. This is the only way to exchange items between players of the same team without them being close in space.

### 3.3 Design and Architecture

In the very beginning of the design process we thought that a client-server architecture could be useful to deal with some aspects of the game we were specifying. Even though the game has been designed as a nomadic ad hoc network game, such server will also be available, but letting players interact in a peer-to-peer way without constantly depending on the server.
Our game also takes into account network failures in its design. As we described in Chapter 1, one of the motivations for this project was getting more experience in the ubiquitous and context aware computing study area by developing new applications on that field. In Chapter 2 we proved that AmbientTalk is a language that makes easier the communication between mobile devices. Due to its discovery mechanisms it is also a good choice when dealing with device service discovering and it can help us to design the connection controlling we aim to achieve. Due to the abstractions it provides when dealing with the problems raised when developing applications for mobile ad hoc networks it also can result a good way to avoid the use of large amounts of code that would be required by other languages to develop an embedded system such as our.

When connection/disconnection events occur, the system is able to perceive which players have disconnected or reconnected, and it sends this information to the other players. This way, each player can see in his PDA screen which members of his team are connected at that moment. Players are also able to detect when they are disconnected themselves, as they will loose connectivity with the server, which is supposed to be always turned on. Apart from taking into consideration network failures, another thing that influences the design of our game is working with GPS accuracy. In the Figure 3.3 below you can appreciate a graphical representation of the basic architecture of the game:

![Figure 3.3: General Architecture](image)

We first developed the game in a desktop machine simulating GPS coordinates and, in a later stage, we deployed it in a PDA and switched it to real GPS coordinates. The simulator had two parts: one (called GPPSimulator) that basically checked that the reading of the data,
using a concrete standard, sent from the satellites would be successful, and the other (called Localizer) that tracked all players movements all around the game area, also simulating an automatically generated GPS data of their positions on-time. The purpose of this simulator was also to facilitate the future game work and testing phases, making easier the future jump to the reality.

In this subsection we have introduced the GPSSimulator and the Localizer in a short way, trying to explain why are they useful to develop our game with the restrictions we had. All the modules developed will be explained in more detail in Section 3.4, where the strategy followed to achieve the design, structure and implementation of the final Figure 3.3 above is also explained step by step.

3.3.1 Functional Requirements

- Server allows manual input through a Graphical User Interface to set the following parameters before booting:
  - Number of teams.
  - Maximum number of players.
  - Initial number of virtual items.

- Players have a Graphical User Interface that allows them to:
  - Visualize the map of all the game area in a frame.
  - Choose which team do they want to join when connecting with the server.
  - Receive notifications of their current state and the one of their team partners.
  - Visualize their own current position represented with a point of a random color with their username on it.
  - Follow their team partner’s movements all around the map.
  - Visualize their team members disconnections (players are painted in black) and reconnections (players are repainted with its original color) on-time.
  - Consult on their items panel the virtual items they collected during the game and operate with them (throw them or activate them).
  - Do zoom in on the map, focusing on the area they are walking through.
  - Do zoom out on the map to return to the initial view of the game area.
3.3.2 Non-Functional Requirements

Apart from the debugging difficulties it implies, the fact of having to develop a game making it able to run simultaneously in a desktop machine and also in a mobile device implies many restrictions under different technological points of view:

- On one hand, the AmbientTalk/2 interpreter has been written in pure Java and requires a regular J2SE Java Virtual Machine supporting version 1.3 or higher, but on the other, the application requires to be implemented using the Java JDK Compiler Compliance with the source compatibility and the generated .class files compatibility 1.3. and not higher, to be able to run on PDA’s.

- Also because of running the application on PDA’s, between all the possible choices in Java, only awt Java Graphics Technology can be used to develop the Graphical User Interface.

- We can not use the main part of the network related packages that are packaged in java.net, in the J2ME Generic Communication Framework [13] [11] (in Java 2 Standard Edition (J2SE)), because it doesn’t result so suitable for the wireless applications that run on certain PDA’s and two-way pagers. There is thus no implementation of the connection interfaces at the CLDC level. The actual implementation is left to MIDP.

3.3.3 Modules

In this subsection, being conscious of the restrictions we have, we introduce a first general overview of the design of all the modules that conform our game simulator. Later on, in Section 3.4 all the modules are explained with more detail.

3.3.3.1 General Overview

The initial modules of our game simulator are listed below:

- GPSSimulator modules:

  - GPSServer (GPS data sender simulator).
  - GPSClIENT (GPS data receiver simulator).
  - GPSSDataStandardModule (GPS data standard module used to simulate the parsing).

- Localizer modules:

  - gameServer.
  - ServerGUI (used to set up the game parameters).
- `gamePlayer`.
- `PlayerGUI` (used to manage player functionalities and data simulations).
- `GPSDataManager` (used by the `PlayerGUI` to make coordinates translations and to generate and extract GPS data in a correct format).

![Diagram](image)

**Figure 3.4: General Architecture**

The two independent parts drawn in Figure 3.4, which will be unified in the future, come up then in the first version of the simulator (to initially separate these two parts also makes the working of the system more understandable and testable for future error detections). The `GPSServer` is the one that, as we said before, establishes the communication between a `GPSServer` (sender) simulator and a `GPSClient` (receiver) simulator, to send a prototype of data in a format normally sent by GPS receivers. The `Localizer` is the one that, through the Graphical User Interface of every player, generates GPS data on a concrete standard every-time players are manually dragged through the screen, simulating as if they were moving about the campus. Such data is sent from the player which is moving to his team members, allowing them to have its localization updated on their screens.

Later on, in a future stage, the `GPSServer` module will be replaced by real satellites, which will supply real GPS information, as well as the `GPSClient` module will be integrated within the `PlayerGUI` module, introducing the necessary modifications.
3.4 Implementation

In this section we are going to explain how the designed simulator has been implemented step by step. We will start explaining how from the beginning our modules have evolved and got connected until we reached our final game.

As we described in the previous section, in the design of our game there are two different parts: the GPSSimulator and the Localizer. The first one has been exclusively done in Java while the second has been implemented using the symbiosis between Java and AmbientTalk, using Java only to develop the Graphical User Interfaces for the players and the server, and the management of GPS data. As we will better see later, it is possible to take AmbientTalk objects down to the Java level so that the Java language can be used to send messages and create new AmbientTalk objects. On the other direction, as AmbientTalk has been totally implemented in Java, it is also possible to instantiate Java classes and send messages to Java objects from within AmbientTalk.

The logical part of the game itself and the part that deals with all the communication between the mobile devices and manages the discovery mechanisms between the entities has been fully done in AmbientTalk (game server and game players). In short, AmbientTalk has been used for the distributed communication. The figure below shows again the basic architecture of our game, but this time showing which roles the chosen technologies play into it:

![Diagram of the general architecture of the game](image)

**Figure 3.5:** General Architecture
3.4.1 Preview of the GPSSimulator and the Localizer in Java

The more basic design of the GPSSimulator module is the one showed in the Figure 3.6 below:

![Diagram of GPSSimulator scheme](image)

**Figure 3.6:** GPSSimulator scheme

These independent modules have been developed for two reasons, which together share the same final objective. On one hand, we previously needed to assure that GPS data could be successfully sent, received and parsed. One the other, we also needed to be able to simulate the complete behavior of all the data transactions that will be done when a player will be moving around the campus with his PDA, generating a simulation of the real GPS coordinates. The shared objective of both modules is to make it easier to read real coordinates in the future and enable communication between different players through a wireless network. The GPSSimulator module will help us to achieve the first part, and the Localizer will help us to achieve the second one. In spite of that, in both modules, we needed to assure that the data sent had the appropriate format. In subsubsection 3.4.1.1 there is the result of the research we did on the existent standards in the data sent by a GPS, exposing the more appropriate for our game.

3.4.1.1 GPS Data Standard NMEA 0183

NMEA is a protocol used when sending GPS coordinates from satellites for marine and terrestrial navigation. We decided to choose this standard because it tell us exactly what we need to know about players while they are moving around the campus: the date, the latitude and the longitude. This list of information is the required to localize a player at one time. Once a GPS knows the geographical coordinates of the receiver’s position, this information can be sent to another device. Apart from the geographical coordinates and the date, the standard also tells the current velocity, the destination way-point from which satellites is receiving, the intensity of the signals that are received, the position of the satellites, the depth of a sensor down in water, etc. The NMEA Standard is the current state of the art standard in GPS systems.
The following is a sample of the common format of all the sentences of the standard:

$GPWPL,4807.038,N,01131.000,E,WPTNME*5C$

- Data items (in ASCII text) separated by commas, contained in a single line.
- Each sentence begins with a $ and ends with a checksum field plus two digits\(^1\).
- Two letter prefix after the beginning symbol $ that defines the device that uses that sentence type (GP for GPS receivers).
- Three letter word that defines the sentence contents, called data type, that defines the interpretation of the rest of the sentence.

**Figure 3.7:** Example of a sentence in the NMEA 0183 Standard from [30]

There are many sentences in the NMEA Standard for all kinds of devices that may be used in similar applications to the one we are developing. After exploring some of the data types of the standard we decided to use the GGA sentence, which provides the essential fix data. The main reason is that GGA sentences include the latitude, the longitude and also the time. This information is the only one we require for our system. Even other sentences may repeat some of the same information while supplying new data, though decided to keep our system as simple as possible.

What follows is an example of a GGA sentence extracted from [30], which illustrates the format of the only sentence data type we are not going to discard while reading simulated GPS coordinates in our GPSSimulator, and the format of sentences our players are going to send to their team members to update their locations in our Localizer:

$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47$

Where:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GGA</td>
<td>Global Positioning System Fix Data</td>
</tr>
<tr>
<td>123519</td>
<td>Fix taken at 12:35:19 UTC</td>
</tr>
<tr>
<td>4807.038,N</td>
<td>Latitude 48 deg 07.038' N</td>
</tr>
<tr>
<td>01131.000,E</td>
<td>Longitude 11 deg 31.000' E</td>
</tr>
<tr>
<td>1</td>
<td>Fix quality:</td>
</tr>
<tr>
<td>0</td>
<td>invalid</td>
</tr>
<tr>
<td>1</td>
<td>GPS fix (SPS)</td>
</tr>
<tr>
<td>2</td>
<td>DGPS fix</td>
</tr>
<tr>
<td>3</td>
<td>PPS fix</td>
</tr>
<tr>
<td>4</td>
<td>Real Time Kinematic</td>
</tr>
<tr>
<td>5</td>
<td>Float RTK</td>
</tr>
<tr>
<td>6</td>
<td>estimated (dead reckoning) (2.3 feature)</td>
</tr>
<tr>
<td>7</td>
<td>Manual input mode</td>
</tr>
<tr>
<td>8</td>
<td>Simulation mode</td>
</tr>
<tr>
<td>08</td>
<td>Number of satellites being tracked</td>
</tr>
<tr>
<td>0.9</td>
<td>Horizontal dilution of position</td>
</tr>
<tr>
<td>545.4,M</td>
<td>Altitude, Meters, above mean sea level</td>
</tr>
<tr>
<td>46.9,N</td>
<td>Height of geoid (mean sea level) above WGS84 ellipsoid (empty field)</td>
</tr>
<tr>
<td></td>
<td>Time in seconds since last DGPS update</td>
</tr>
</tbody>
</table>
3.4.1.2 GPS Client-Server Communication

In the NMEA Standard there are no commands to indicate that the GPS should do something different. There is no chance to give any instructions or send some messages back to the unit to make it know that the message has been read correctly or to request the unit to send again some of the data that might be lost. For this reason, we are only going to make unidirectional communication for the data sending simulation, from the GPSServer module to the GPSClient. Usually, receiving units ignore the data if the checksum is not right, expecting data to be sent again sometime later. To implement our GPSClient module of the simulator we used two libraries (GCF.jar and nmea.jar) and the following classes, taken from [37]: NMEAReader, NMEAReader, NMEAServer, NMEAServer, NMEAEmitter, NMEAEmitter, NMEAReader, NMEAEmitter, NMEAReader, NMEAEmitter.

By extending the classes NMEAServer and NMEAReader we will be able to read all the GPS data, but ignoring the main part of it, which is not required, only taking the GGA sentences, as we explained before.

The figures box 3.1 and box 3.2 below show how the data is sent by the server, and received and parsed by the client:

```java
public class GPSServer {
    private StreamConnectionNotifier server = null;
    private StreamConnection client = null;
    private String URL = "serversocket://:8080";
    public GPSServer() {
        try {
            server = (StreamConnectionNotifier) Connector.open(URL, Connector.WRITE, true);
        }
        listen();
    }
    public void listen() {
        while (true) {
            if (server != null) {
                client = server.acceptAndOpen();
                if (client != null) {
                    sendData(client);
                    client.close();
                }
            }
        }
    }
    ...}
```

Box 3.1: GPSServer class
As you can see in the previous box, the GPSClient module uses a class called CustomReader. That class is the one that establishes connection with the GPSServer module and where we can select which of the read data we want parsed:

In the figure box 3.4 you can see a sample of some of the random NMEA sentences sent by the GPSServer module. The corresponding output that generates the GPSClient module after parsing such sentences is shown in the figure box 3.5:
3.4.1.3  **gpsDataManager module**

After finishing the communication test to simulate the real GPS data reading in the NMEA Standard from a Com port, we need to develop a module to deal with the simulation of all the GPS data generation and extraction that is done when a player is moving around the campus. We need a module that allow us to convert from planar coordinates of the picture map on our PDA to the real-world polar coordinates and vice-versa.

As described in [39], the polar coordinate system is a two-dimensional coordinate system in which each point on a plane is determined by an angle and a distance. As the coordinate system is two-dimensional, each point is determined by two polar coordinates: the radial coordinate and the angular coordinate. The radial coordinate denotes the point's distance from a central point known as the pole (equivalent to the origin in the Cartesian system). The angular coordinate (also known as the polar angle) denotes the positive or anti-clockwise angle required to reach the point from the 0 deg ray or polar axis (which is equivalent to the positive x-axis in the cartesian coordinate plane). Planar coordinates, are coordinates that specify a position in a plane. The most common sort of planar coordinates use an (x,y) pair. They are frequently used to describe positions on maps. The translation from one system to the other can be found through trigonometric formulae.

To store the data that will be generated, extracted and sent from one player to the server and to the other players we use the GEOLocation class shown in box 3.6 below:

```java
class GEOLocation {
    int degrees;
    int minutes;
    double seconds;

    public GEOLocation(int d, int m, double s) {
        this.degrees = d;
        this.minutes = m;
        this.seconds = s;
    }

    public String toString() {
        String trunkSec = Double.toString(seconds);
        int idxDot = trunkSec.indexOf(".");
        trunkSec = trunkSec.substring(0, idxDot+3);
        return (degrees + "\u00B0" + minutes + "\u0027" + trunkSec + "\u0027");
    }

    public double toOnlyDegrees() { return ((double)this.degrees + (double)this.minutes/60 + this.seconds/3600); }
}
```

**Box 3.6:** GEOLocation class

31
Knowing the dimensions of a standard screen of a PDA, we took the Campus Etterbeek map of the Vrije Universiteit Brussel from the well known application Google Earth\textsuperscript{2} with the appropriate scaling. We got the polar coordinates of the four corners of the map, with the purpose of being able to make the translation system. In the box 3.7 below you can see the initialization of the gpsDataManager class:

```java
import java.util.Date;
import java.awt.Point;

public class GPSDataManager {

    // Coordinates of the 4 points of the map
    private GEOLocation minNorth = new GEOLocation(50, 49, 32.29);
    private GEOLocation minEast = new GEOLocation(4, 23, 26.49);
    private GEOLocation maxNorth = new GEOLocation(4, 24, 11.70);
    private GEOLocation maxEast = new GEOLocation(50, 49, 00.82);

    // Width and height from the window to map
    final int X, Y;

    // Scale factors to map on the window
    private double incFactorX;
    private double incFactorY;

    public GPSDataManager(int x, int y) {
        this.X = x;
        this.Y = y;
        calculateFactors();
    }

    private void calculateFactors() {
        // Incremental factor for X coordinates
        Double initEast = minEast.toOnlyDegrees();
        Double endEast = maxEast.toOnlyDegrees();
        Double absEast = endEast - initEast;
        this.incFactorX = absEast/X;

        // Incremental factor for Y coordinates
        Double initNorth = minNorth.toOnlyDegrees();
        Double endNorth = maxNorth.toOnlyDegrees();
        Double absNorth = initNorth - endNorth;
        this.incFactorY = absNorth/Y;
    }
}
```

**Box 3.7: Initialization of the gpsDataManager class**

The factors incFactorX and incFactorY calculated in the method calculateFactors() above are essential for the translation functions from the polar system to the planar one and vice-versa. Both functions are attached in the box 3.8 below:

```java
public String translateToPolar(int x, int y) {
    Double newNorthCoord = minNorth.toOnlyDegrees() - y*incFactorY;
    Double newEastCoord = minEast.toOnlyDegrees() + x*incFactorX;

    GEOLocation northGeoLoc = fromOnlyDegrees(newNorthCoord);
    GEOLocation eastGeoLoc = fromOnlyDegrees(newEastCoord);

    String newPosition = northGeoLoc.toString() + "N " + eastGeoLoc.toString() + "E";
    return newPosition;
}
```

\textsuperscript{2} An application property of Google that combines the power of Google Search with satellite imagery, maps, terrain and 3D buildings to make the world's geographic information available for everybody

32
public Point translateToXY(String location) {
    int idxD = location.indexOf("°");
    Integer nd = Integer.valueOf(location.substring(0, idxD));
    int idxM = location.indexOf("'", idxD+1);
    Integer nm = Integer.valueOf(location.substring(idxD+1, idxM));
    int idxS = location.indexOf("", idxM+1);
    Double ns = Double.valueOf(location.substring(idxM+1, idxS));
    
    idxD = location.indexOf("°", idxD+1);
    Integer ed = Integer.valueOf(location.substring(idxD+4, idxD));
    idxM = location.indexOf("'", idxD);
    Integer em = Integer.valueOf(location.substring(idxD+1, idxM));
    idxS = location.indexOf("", idxM+1);
    Double es = Double.valueOf(location.substring(idxM+1, idxS));

    GEOLocation northGeoLoc = new GEOLocation(nd, nm, ns);
    GEOLocation eastGeoLoc = new GEOLocation(ed, em, es);

    Double decimalNorthCoord = northGeoLoc.toOnlyDegrees();
    Double decimalEastCoord = eastGeoLoc.toOnlyDegrees();
    
    Double y = (minNorth.toOnlyDegrees() - decimalNorthCoord)/incFactorY;
    Double x = (decimalEastCoord - minEast.toOnlyDegrees())/incFactorX;
    return new Point(roundUp(x),roundUp(y));
}

Box 3.8: System coordinates translation methods

Other useful methods in the gpsDataManager module are the following ones:

public void setMapCoordinates(String upperLeft, String downRight);
public void resetMapCoordinates();
public String generateGGAData(String location);
public String extractGGAData(String mainSentence);

private GEOLocation fromOnlyDegrees(Double deg);
private int roundUp(Double x);

Box 3.9: Headers of the gpsDataManager module

setMapCoordinates() allows an user of this module to change the values of the references of the four corners of the map, allowing him to change the whole map itself if necessary. Only the upper left and down right polar coordinates of the new map are required to change the system. This method is also useful for the zooming functionalities. When a player switches the zoom in option on, the viewer will focus into a single area of the map. The GPSDataManager has to be notified with the new polar positions that are going to be set on the corners of the screen of our PDA, to be able to properly recalculate the conversion factors to continue simulating proper data. When a player switches the zoom out function on, the method resetMapCoordinates() will have to be called in order to reset the map references and the map factors to the view of the whole game area.

Note that this changing of coordinates and all the GPS data generation will not be necessary when moving on to the real coordinates because we will directly receive the polar position with the GPS receiver incorporated in the PDA, but they are essential for our simulator.
The method that will still be very useful when jumping to the real coordinates will be the `extractGGAData(String mainSentence)`, which gets the polar position from a GGA sentence given in a String, returning also an String.

### 3.4.1.4 Player’s Graphical User Interface

In order to allow the GPSDataManager module explained in the previous subsubsection to calculate the scaling factors for coordinate’s system translations, we require to initialize it the with the size of the map viewer of the PlayerGUI:

```java
public class GPSGUI extends Frame implements MouseListener,
        MouseMotionListener {
    ...
    private GPSDataManager cc;
    private PlayerPoint me;
    ...
    cc = new GPSDataManager(mapWidth, mapHeight);
    me = new PlayerPoint("", 0, 0, 0, 0);
    ...
}
```

**Box 3.10:** Initialization of the GPSDataManager and the PlayerPoint

This first version of the Localizer has been exclusively implemented in Java, allowing only one single player, to make easier the testing of the NMEA sentences generation when players move around the campus. The GUI allows the user to select a starting location point for the player by clicking on the map. When clicking on the map, the player is represented with a colored point labeled by his username. Once the player has been located, the user can simulate player movements around the campus by dragging the PlayerPoint. Whenever a player is dragged through the GUI, the GPS data in the NMEA Standard is generated, corresponding to his current location:

```java
if (me.moving) {
    me.x = evt.getX();
    me.y = evt.getY();
    //Simulated NMEA sentences (GPS data generation)
    String polarDataPosition = cc.translateToPolar(me.x, me.y);
    String gpsDataPosition = cc.generateGGAData(polarDataPosition);
    System.out.println(gpsDataPosition);
}
```

**Box 3.11:** NMEA data generation when dragging a player through the screen

By interpreting the NMEA sentences that players will send every time they move, the path of every single player will be able to be reconstructed and also painted in the screen of the others. In the next stage, when adding to the Localizer the AmbientTalk modules for the distributed layer, the PlayerGUI will include the required methods to be able to store and update multiple player locations and also to reaction when disconnection events happen:
public void updatePlayerPosition(String usr, String nmeaSentence) {
    String polarPosition = cc.extractGGAData(nmeaSentence);
    Point pixelPosition = cc.translateToXY(polarPosition);
    PlayerPoint pp;
    if (players.containsKey(usr))
        pp = (PlayerPoint)players.get(usr);
    else
        pp = new PlayerPoint(usr, Math.round(pixelPosition.getX()), Math.round(pixelPosition.getY()), 2, 1);
    pp.setPolarPosition(polarPosition);
    players.put(usr, pp);
    imagePanel.repaint();
}

Box 3.12: NMEA data interpretation when an update message of a player position is received

After that, when deploying the application on a PDA, manual player movements on the GUI will be avoided and replaced by the part of code simulated in this first GPSClient version. Real GPS coordinates will be read from the satellites instead of the simulated ones in the GPSServer module.

3.4.2 Localizer in AmbientTalk using the symbiosis with Java

In this section we introduce the gameServer and the gamePlayer modules implemented in AmbientTalk; the changes this action implied in the other modules; and we are also going to explain how the symbiotic relationship between AmbientTalk and Java has been used.

3.4.2.1 gamePlayer module in AmbientTalk

Every player has his own Graphical User Interface, declared as follows:

    def newUser( playerGUI := jlobby.at.gps.PlayerGUI ) {
        ...
        def listener := playerGUI.new(localFacade);
        ...
    }

Box 3.13: Accessing our own Java classes from within AmbientTalk

All the classes available in the classpath of a running Java Virtual Machine are accessible from AmbientTalk through the jlobby object. By this way, all the methods from the Java GUI we developed can be called with the listener object we declared in our code, as if they were simple AmbientTalk objects. To instantiate a Java class from within AmbientTalk the behavior is similar to how AmbientTalk objects are instantiated. The arguments we pass to the new method are passed as arguments to the Java constructor. In the code above, when declaring the GUI, the object localFacade is passed as an argument to the Java class. The localFacade contains all the methods the GUI will call from the Java world:

35
```python
def localFacade := object: {
def initServiceDiscovery (usr) { lookForServer (usr) };  
def resetUsername (usr) { checkUsername (usr) };  
def setLocation (pos) { location := pos; broadcastOwnPosition(); };  
};
```

**Box 3.14: localFacade object**

When the AmbientTalk object localFacade is passed as an argument to the Java GUI new method, the proper conversions will be done from AmbientTalk to make the object appear to the Java world as a regular Java object. Because of the constructor method will be expecting an object of an interface type, the AmbientTalk object will be converted by the AmbientTalk Interpreter into a Java object implementing that interface:

```java
public class PlayerGUI extends Frame implements MouseListener,
   MouseMotionListener {
   ...
   private ATGPSSimulator atGPS_;
   ...
   public interface ATGPSSimulator {
   // used by the GUI to boot the service discovery mechanism and
   // setting the username
   public void initServiceDiscovery(String username) throws Exception;
   // used by the GUI to notify the gpsPlayer simulator that the user
   // changed the username
   public void resetUsername(String username) throws Exception;
   // used by the GUI to notify the gpsPlayer simulator that the user
   // changed the location
   public void setLocation(String location) throws Exception;
   }

   public PlayerGUI(ATGPSSimulator atGPS) {
      super("GPS Simulator");
      atGPS_ = atGPS;
   }
}
```

**Box 3.15: AmbientTalk interface**

In the example above, the Java interface ATGPSSimulator contains the three methods `initServiceDiscovery(String username)`, `resetUsername(String username)` and `setLocation(String location)` defined in the localFacade object. These three methods are the ones that are going to be called respectively from the GUI when the player introduces his username for the first time on the first hand; whenever the username is already in use or invalid on the second; and every time the player changes his location on the third.

```java
if (ableToConnect) atGPS_.initServiceDiscovery(username);
else atGPS_.resetUsername(username);
...
String polarPosition = cc.translateToPolar(x, y);
String nmeaSentence = cc.generateGGAData(polarPosition);
atGPS_.setLocation(nmeaSentence);
...
```

**Box 3.16: calls to the AmbientTalk module**
First time players introduce an username through the GUI, doesn’t matter if it is valid or not, players engage in server service discovery through the network, when the AmbientTalk method lookForServer() is called.

```ruby
def lookForServer(usr)
  deftype gameServer;
  whenever: gameServer discovered:
    |server| system.println("Server Found");
    when: server disconnected: { listener.setAllOnBlack(true); listener.display("Disconnected from the Server"); }
    when: server reconnected: { listener.display("Reconnecting to the Server..."); }
    serverRef := server;
    checkUsername(usr);
  ;

Box 3.17: Engagement in server service discovery
```

As you can see in the Box 3.17 above, when the server is discovered, players check if their username is valid or not. The method checkUsername(usr) sends an asynchronous message to the server to check the validity of the username. If the username is invalid, another call to the Java world will be done in order to notify the player to try with another username. If the username is valid, players will apply to join the game and, if the server is not full, they will randomly join one of the playing teams. When players apply to join the game a whole object called remoteFacade (Box 3.18 below) is passed as an argument to the server by far reference. The remoteFacade methods are the ones the server and the other team players will be able to call via asynchronous message sending.

```ruby
def remoteFacade := object: 
  def getName() { username }
  def getLocation() { location }
  def addNewPlayer(playerFacade) { MyTeamPlayersList.add(playerFacade); nil }
  def updatePlayerPosition(usr, loc) { listener.updatePlayerPosition(usr, loc); nil }
  def teamPlayerDisconnected(usr) { listener.setOnBlack(usr, true);
      listener.display(usr + " disconnected"); nil }
  def teamPlayerReconnected(usr) { listener.setOnBlack(usr, false);
      listener.display(usr + " reconnected"); nil }
  def receiveMessage(msg) { listener.display(msg); nil }

Box 3.18: remoteFacade object
```

When players successfully join the game, they are provided with the remoteFacades of all their team partners. Because of that, players can update all their team partners with their latest location every time they move. The gamePlayer module then just requires one abstract data type to store all the facades of the members of their team to later be able to broadcast the new position to all the remote players:

---

3 Even though it was designed to give players the opportunity to choose which team do they want to join, we decided to temporarily implement it this way.

37
def MyTeamPlayersList := jlobby.java.util.Vector.new();
...

def broadcastOwnPosition() {
  0.to: MyTeamPlayersList.size() do: {
    [index] def currentPlayer := MyTeamPlayersList.elementAt(index);
    currentPlayer.updatePlayerPosition(username, location);
  };
  serverRef.updateServerPlayerPosition(username, location);
}

Box 3.19: Broadcast of the location to the other team players and the server

When a player changes his position all through out the game area, he directly tells his
new location not only to his team partners, but also to the server. That means that when
disconnection and reconnection event occurs, and players re-apply to join the game, the
server will always be able to update them all with all last known locations of all their team
partners.

3.4.2.2 gameServer module in AmbientTalk

The server basically allows to set some game parameters; manages all new incoming players;
keeps track of player’s states (connected/disconnected); and records their latest locations.
The following are some abstract data types required by this module in order to store the
following information:

- All the teams that are dynamically created when parameters are set on (Box 3.20).
- All player’s usernames, remote facades and locations (Box 3.21).

def TeamsList := jlobby.java.util.Vector.new();
0.to: NTEAMS do: {
  [index] TeamsList.add(jlobby.java.util.Vector.new());
}

Box 3.20: Vector to store player’s information organized by teams

def infoPlayer := isolate: {
  def facade := nil;
  def username := nil;
  def connected := nil;

  def init(fac,usr,con) {
    facade := fac;
    username := usr;
    connected := con;
  }
};

Box 3.21: infoPlayer object to store player’s information

The server also has his own Graphical Interface, used to set up all the parameters designed,
and is declared as follows:
def settingsInterface := object:
   {
      def setTeams(nTeams) { NTEAMS := nTeams }; 
      def setPlayers(nPlayers) { NMAXPLAYERS := nPlayers }; 
      def setItems(nItems) { NITEMS := nItems }; 
      def bootServerSettings() { bootServer() }; 
   };
   def serverGUI := jlobby.at.gps.ServerGUI;
   def settingsManager := serverGUI.new(settingsInterface);

Box 3.22: Server Graphical Interface

Booting the server declares an object called remoteServer (Box 3.23) which later needs to be exported to be able to be discovered and remotely accessed by other actors:

def remoteServer := object:
   {
      ... 
      def validUsername(username) {...};
      def joinGame(name, playerFacade) {...};
      def updateServerPlayerPosition(user, loc) {...};
      def isAlreadyPlaying(playerFacade) {...};
      def broadcastConDiscPlayer(team, username, disc) {...};
      def managePlayerConDiscEvent(playerFacade, disc) {...};
      def managePlayersInRange() {...};
   };
   def type gameServer;
   export: remoteServer as: gameServer;
   network.online();
   remoteServer

Box 3.23: export of the remoteServer

After the server has been booted and when one player successfully joins the game, if the number of players is not higher than the maximum established, he gets the far references to all the other players of the team he has been assigned to. The other players of the team also get the far reference of this new player from the server. Apart from giving a key for the communication between players from the same team, the server also sends all player’s last known locations, which are stored in there:

if: (currentNumberOfPlayers<NMAXPLAYERS) then: {
   def currentTeam := TeamsList.elementAt(idxTeamToJoin);
   0.to: currentTeam.size() do: {
      index : def currentPlayerInfo := currentTeam.elementAt(index);
      currentPlayerInfo.facade<->addNewPlayer(playerFacade);
      playerFacade<->updatePlayerPosition(currentPlayerInfo.facade, 
         PlayerPositions.get(currentPlayerInfo.username, 
            PlayerPositions.get(currentPlayerInfo.username));
   };
   def infoEntry := infoPlayer.new(playerFacade, name);
   currentTeam.add(infoEntry);
   TeamsList.set(idxTeamToJoin, currentTeam);
   if: (idxTeamToJoin==NTEAMS-1) then: { idxTeamToJoin := 0 }
   else: { idxTeamToJoin := idxTeamToJoin + 1 };

   currentNumberOfPlayers := currentNumberOfPlayers + 1;
   listener.display("Player " + name + " is connected to the server and 
     joined the team "+idxTeamToJoin);
   true;

39
3.4.3 Future Implementation Work

Even though we gave the specification and some keys about the design of the virtual items, they have not been implemented. In this subsection of future implementation work, added to what we already explained, we give some more ideas and details we had in our minds for implementing the virtual items.

The main idea was creating a prototypical object, with the common properties and methods all virtual items have, as we previously discussed:

```ruby
def vItem := object {
  def location := nil;
  def icon := nil;
  def damage := nil;
  def damageRadius := nil;
  def pickUpRadius := nil;

  def init(loc) { location := loc; ... ;}
}
```

Box 3.25: Object prototype for vItem

Every single type of item will extend the previous object, adding its particular values and methods, depending on its game-play functions.

The according icons for each type of virtual item showed in Figure 3.2 will be implemented as numerical objects from 1 to 4 at the AmbientTalk level, as identifiers, but they will be implemented as Images in the PlayerGUI module at the Java level. A very similar structure to the one used for the PlayerPoint could be used to display them on the screen. The damageRadius and pickUpRadius will also be numerical objects, according to the description done in Figure 3.1, and so will be the damage variable. To make interactions become real in playing terms between players and virtual items we will also need to include some variables in the gamePlayer module, such as the discoveryRadius or life.
The following are some suggested values for such variables to reach an acceptable playability:

- **life**: from 1 to 100.
- **damage depending on the type of virtual item:**
  - mines: -40.
  - remoteBomb: -60.
  - trap: -5.
  - healthPackage: 25.
- **pickUpRadius < damageRadius < discoveryRadius**
Chapter 4

Experiments and Results

The aim of this chapter is to explain how the game was taken from the desktop machine where it was implemented to a PDA. We thereby switched from using simulated coordinates to real GPS coordinates. We will briefly explain the problems we met while doing the deployment of the application and the solutions we provided, when possible.

4.1 Deploying the game on a PDA

After having installed AmbientTalk in a PDA running Windows Mobile, we could start testing its behavior. Table 4.1 shows the specifications of the PDA we used, in comparison with the laptop were the application was previously deployed:

<table>
<thead>
<tr>
<th>Laptop</th>
<th>PDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make &amp; Model</td>
<td>Apple PowerBook G4</td>
</tr>
<tr>
<td>CPU</td>
<td>PowerPC G4, 1.5 GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>1GB</td>
</tr>
<tr>
<td>OS</td>
<td>Apple Mac OS X</td>
</tr>
<tr>
<td>Java VM</td>
<td>Sun J2SE 1.5.0_13</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of specifications

After many tests on the PDA, a very slow execution was observed, both when starting the AmbientTalk Interpreter, and the whole execution of the application. The loading of the background image representing the campus of the university, where the game takes place, required a lot more time than expected. The different players, running on different virtual machines and also on different devices, successfully discovered the Server (always booted on a desktop machine). After setting their usernames, they properly applied to join the game, and they got the far references of all their team partners. This part of the process of discovery and joining the game also had a slower execution than the expected. However,
the main problem we had to deal with on the PDA were the slow graphical operations. As the number of players moving about on the campus was increasing, more and more location updates were necessary. Visualizing all these location updates on the PDA screen was so slow that it affected the playability of the game for the PDA player. We analyzed the execution time of every single call of the painting methods and, after optimizing such methods as much as possible, the results were still not as fast as required.

4.2 Reading Real GPS Coordinates from the PDA

In parallel with the experiment we discussed in the previous section, we also introduced actual GPS coordinates – read from GPS receiver – into our application. To be able to read GPS coordinates from our PDA, we had to modify some of the first modules of the system we initially designed in order to deal with some new communication aspects and make GPS data flow through the communication port. In the first part of the project, as explained in subsection 3.4.1, we checked that GPS data could be properly sent and received via sockets, using the NMEA Standard. Now in this stage, the initial GPPServer module is finally replaced by connection to a serial port which read the data coming from real satellites, and some modifications have to be done to the GPPClient module to adapt to this.

4.2.1 Handling GPS Data Through the Serial Port

To have data flowing through the communication port, we need to identify which communication port must be opened, because it may vary from one machine to another (Com ports mostly vary from 1 to 4). If a wrong port is chosen, an IOException is thrown communicating that the system can't find the specified file. If the chosen port is busy, the IOException just tells that the handle is not valid. Apart from that, as each GPS receiver sends data at a concrete baud rate, we also need to assure that the baud rate of our GPS receiver corresponds to the one we put in our URL String to open the StreamConnection. After some tests, we set the port as follows:

```java
private final String URL = "comm:4;baudrate=57600";
```

**Box 4.1:** URL to read from a Com port at a certain baud rate

To avoid getting parity errors because of the speed of the data flow, we also started reading 131072 bytes instead of a single one in every loop. The NMEARender class (superclass of the one we are implementing) allows us to send as many bytes as we require to the parser:
StringBuffer sb = new StringBuffer();
byte[] b = new byte[131072];

while (true) {
    int size = is.read(b);
    sb.append(new String(b, 0, size));
    super.fireDataRead(new NMEAEvent(this, sb.toString()));
    ...
}

Box 4.2: Reading and parsing of the sentences

As the major part of GPS receivers do not work indoors, all experiments have been done outdoors, going in and out of the building when changes in the code needed to be done. Looking at the results obtained, we saw that some data in the NMEA Standard was received, but the main part of the fields of the sentences were empty, including the latitude and the longitude ones. The reason for this is that to determinate a concrete position on earth it is required to receive signal from at least three satellites, and sometimes it can take some time before a GPS finds a lock position (also called a fix). Once a GPS receiver knows the geographical coordinates of its position, they can be sent to another device. For this reason, until a lock is established, some of these fields of the sentences can remain empty.

The fact that such NMEA sentences with the fields of latitude and longitude are sent being empty, did not affect the parser we already tested in the first part of the project, to correctly parse only the GGA sentences, but led us to modify our GPSDataManager module to be able to interpret NMEA sentences with empty fields, and being able to filter them to paint our player’s locations on the screen only when full sentences are received. To adapt to the reading of the real coordinates we dropped all mouse events from our original PlayerGUI module, which allowed us to generate NMEA sentences when dragging the player through the GUI, and reply them for the following code, which is called every time an NMEA sentence is received:

```
...
String polarPosition = cc.extractGGAData(nmeaSentence);
if (polarPosition!="emptyLocation")
{
    Point XY = cc.translateToXY(polarPosition);
    if (firstTime)
    {
        firstTime = false;
        me = new PlayerPoint(username, (int)XY.getX(), (int)XY.getY(), 2, 1);
    }
    else ( me.x = (int)XY.getX(); me.y = (int)XY.getY(); )
    imagePanel.repaint();
}
...
```

Box 4.3: Filtering the empty sentences from the PlayerGUI module

When testing the game with multiple PDA’s, the following code line will have to be included, as it was in the simulated version, in order to tell the player team members that they need to update its location:
Once we have been able to make run the parser and the filter together, to read real GPS coordinates all around the campus, and locate the player's position on the screen, the still remaining problem of repainting locations in an acceptable timing in the PDA did not allow us to go further.

4.3 Difficulties in Deploying Applications on Mobile Devices

We have found many obstacles while deploying the game we initially designed and partially implemented on a desktop machine to the PDA. Specially the fact of having to develop a game which had to be able to run simultaneously in a desktop machine and also in a mobile device, lead us to confusions many times. The constant apparition of new errors during the deployment, turned debugging into the phase which consumed the main part of the time of the development cycle of our project.

By integrating and testing our software in an embedded system we found that we had the added difficulty of having no proper debugging tools in AmbientTalk. The lack of such debugging tools also affected the deployment of the application on the PDA, forcing us to debug the errors in our mobile device by using the plain old input and output statements. Because of that, debugging errors on a mobile device can become a very slow and painstaking task.

To debug the behavior of the application on the PDA, we had to work in long cycles, e.g.:

- 1). Change code on a desktop machine.
- 2). Compile and package binaries.
- 3). Copy and install the binaries on the PDA.
- 4). Run the binaries to test the application behavior and to check for errors.
- 5). Go back to point (1) to try to fix bugs...

In our particular case, step (4) often required us to leave the building to allow the GPS receiver acquire a lock position. Obviously this further slow down the debugging process.

Virtual Machines also play a very important role in the performance of embedded applications such as our game. They have a direct impact in the execution time and memory
consumption. As it was proved in [1], every Java Virtual Machine may act differently depending on the application type. On one hand, after looking at the results of the experiment explained in Section 4.1, we can wonder whether Java is a good choice to work with graphics on mobile devices with limited memory space, but on the other, we just saw that it is not trivial to figure out conclusions from a Java application running on an a mobile device, because there are many different aspects that may influence its performance.
Chapter 5

Conclusions

5.1 Summary

In this project we proposed the development of a location-aware game, taking advantage of the increasing integration of location-aware technologies in mobile devices such as proximity sensors or GPS satellite signals. Our intention was to develop a game which could mix some face-to-face properties of traditional games with the non-real factor that can be provided by computers. The aim of such game was twofold.

First, we built a virtual game which allows playing outdoors exploiting the existing network facilities in the campus of the VUB.

Secondly, we took advantage of a novel language called AmbientTalk for developing a game that takes into account the hardware phenomena of mobile ad hoc networks. By decreasing the complexity of interacting in mobile networks, AmbientTalk greatly simplifies the development of distributed applications, such as Virtual Paintball Game.

The main tasks that have been done during the realization of this project can be grouped in the following sections:

- Implementation of a virtual version of the Paper-Scissors-Stone game to familiarize ourselves with AmbientTalk.
- Design of the logic of the modules of the Virtual Paintball Game.
- Implementation and testing of the modules simulating real game's behavior.
- Deployment of the application on a PDA.
5.2 Future Work

In this section we give an overview of some future directions and future work that could build upon the implementation we discussed in our report. As we saw in the previous chapter, we noticed a slow execution running the game on the PDA on one hand, but we can not assure that the slow flickered update of the graphics is a direct consequence of that on the other. This leads us into two different future directions:

- The first one is build upon the assumption that such lack of resources, which clearly affects the performance of the application, also affects some of the Java Graphics methods, which may run in separate threads affecting the flow of the others, making the repainting have a weird behavior and also making impossible to play. An immediate way to verify such assumption would be to test the current application on a more powerful mobile machine and compare its behavior.

- The second resides in the assumption that, as we explained in Section 4.3, both of the two Java Virtual Machines we tested to run our embedded application, have a so strong impact into the performance of our application, making impossible to play. In this direction, we should do some investigation into which Java Virtual Machines would reveal the best behavior while executing our game.

Other ideas for future work are:

- Implement the possibility of choosing which team players want to join.

- Finalize the design and do the implementation of virtual items, adding new ones such as temporal invisibility packages or a flag virtual item to add the possibility to play the well known game capture the flag.

- Implement new game playing modes by:
  - Adding the possibility for teams to select up an initial strategy battle which reminds to the players what do they have to do.
  - Add a league system with scores, rounds and rankings, that could also be uploaded to the Internet.
  - Introducing the role of a traitor player, which plays in one team, but visualizes and is visualized by the opponents as if he was one of his team members.
  - Creating the role of a leader player for each team, which has to be defended by his team partners, and killed by the opponents to beat the other team.

- Add a speed limitation for players to not let them use bikes or skates to cheat.

- Improve the Graphical User Interface by adding sprites or 3D models.
• Recording of the game from the server.
• Create a team speaking radio channel using the micro of the PDA.

5.3 Personal Valorization and Acquired Experience

I feel that these last months have been a very interesting and intensive learning period for me. I started working in a totally new area for me as it is the domain of distributed programming and mobile ad hoc networks. I developed an application which had to run reliably on a mobile device, and I did it taking advantage of the latest advances in networking and location-aware technologies at the same time. I took profit of the abstractions that AmbientTalk provides to deal with the hardware phenomena of mobile ad hoc networks on one hand, and the incorporation of location-aware technologies such as GPS receivers on mobile devices on the other.

To do the work I just described I also familiarized myself with a new working environment such as Mac OS X. I learned how to work with Java Graphics, and how to use the Eclipse SDK platform [25], which I used to develop and debug the Graphical User Interfaces implemented using the awt Java technology. Apart from all these new domains, applications and technologies, at the end of the project, I also learned how to produce book quality text of scientific and technical works with one of the most powerful formatting programs called \LaTeX [15], [27], which I used to write this report.

5.4 Conclusion

By developing Virtual Paintball Game, we accomplished one of our initial objectives: gaining more experience in implementing more realistic AmbientTalk applications.

AmbientTalk, and the Ambient-Oriented Programming paradigm in general, proposes a novel model to deal with the characteristics of distributed applications and mobile ad hoc networks. By taking advantage of these new abstractions, we think we have been able to reduce the complexity of the implementation of the game, in comparison with an implementation using a traditional OO language.

Our initial expectations of developing our Virtual Paintball Game have been affected by some problems we ran into during the implementation and debugging phases of the project. In spite of that, our objectives have been mostly accomplished, leaving for future work few implementation aspects and the performance issues that limited playability on the PDA.

We succeeded in developing the simulator for our application and also in reading actual GPS coordinates, creating the first application ever done in AmbientTalk that incorporates
location-aware technologies.

Mobile devices that interact with location systems are becoming more and more popular everyday. We even expect that location-aware technology could generate similar demands in the games market as advanced graphic cards did a decade ago. We believe our Virtual Paintball Game presents a new prototype for this new field of games. Furthermore, the location-aware technology we experimented with has applications in distributed systems far beyond the scope of games.
Bibliography


51


URL: http://prog.vub.ac.be/amop/.

[23] AmbientTalk/2 Language Reference [online].
URL: http://prog.vub.ac.be/amop/at2langref/.


[26] Java Technology [online].

[27] LaTeX Tutorial Site [online].
URL: www.chem.uga.edu/cgso/dissertation/LaTeX.cfm.

URL: http://www.monopolylive.com/.

[29] La Mosca citygames: met GSM en GPS op avontuur in de stad [online]. 2005.
URL: http://www.lamosca.be/.

[30] NMEA Standard Reference Site (1) [online].
[31] NMEA Standard Reference Site (2) [online].
URL: http://www.gpsinformation.net.

[32] NMEA Standard Reference Site (3) [online].

[33] NMEA Standard Reference Site (4) [online].

[34] Official NMEA Standard Site [online].
URL: http://www.nmea.org.

URL: http://realreplay.mopius.com/.


[37] SourceForge.net [online].
URL: http://sourceforge.net.

[38] Wikipedia (1) [online, cited 12/02/2008].