Faculdade de Engenharia da Universidade do Porto

Building Mobile and Pervasive Applications in Second Life™

Pedro Ricardo Ferreira Teixeira de Jesus Leite

Thesis submitted in fulfillment of the requirements for the

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Major de Telecomunicações

Supervisor: Ricardo Morla (Professor)

July 25 of 2008

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES 2007/2008

A Dissertação intitulada

"Building Mobile and Pervasive Applications in Second Life"

foi aprovada em provas realizadas 15/Julho/2008

o júri

Presidente Professor Doutor Artur Pimenta Alves
Professor Catedrático da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Rui João Peixoto José
Professor Auxiliar da Escola de Engenharia da Universidade do Minho

Professor Doutor Ricardo Santos Morla
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

Ricardo Santos Morla

O autor declara que a presente dissertação (ou relatório de projecto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extractos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são correctamente citados.

Autor - Pedro Ricardo Ferreira Teixeira de Jesus Leite

Faculdade de Engenharia da Universidade do Porto

Abstract

Prototyping and simulation are concepts, that have over the course of mankind, played an important role, preventing errors and correcting details that certainly would have cost unaccountable amounts of resources.

With the boom of research within the decade old, Ubiquitous Computing field, due to the development of numerous mobile, sensing and other technologies, the models of human-computer interaction, and even of user-to-user relations, are shifting. This constitutes numerous opportunities in terms of even more scientific research, and possibly subsequent product development.

Simulating and prototyping complex ubiquitous computing scenarios and applications, that enabled such new models, has proved to be a difficult task, a fact underlined by several projects.

As the online metaverse Second Life was subject of a media and user-number boom, it has been proven to support more serious aspects, relaying on its scripting and object creating capabilities.

All these issues form the premise of the present thesis. Both Second Life™ and the open source version of the simulator, Opensim, were used as testing ground for simulation of mobile and pervasive applications within the ubiquitous computing scope, to evaluate their capabilities in this domain.

Within this context, two simulations were conducted: a social proximity application and a smart room environment, which exposed Second Life and Opensim's simulation capabilities along with some disadvantages. Also, performance levels were measured, allowing to compare Second Life™ and Opensim's processing capabilities and subsequence stability when running the simulations previously mentioned.

The overall results show that both simulators have the main characteristics desired in a ubiquitous computing simulator, with special focus on Opensim that, thanks to its open source nature, offers high flexibility.

Resumo

Prototipagem e simulação são conceitos, que durante o curso da humanidade, têm tido um papel importante, prevenindo a ocorrência de erros e optimizando a correcção de detalhes, que noutro caso teriam um incalculável custo de recursos.

Com a explosão do número de trabalhos de investigação na área da computação ubíqua, devido ao recente desenvolvimento de tecnologias de comunicação móvel, e redes de sensores entre outros, os modelos de interacção entre homem e máquina, e mesmo entre utilizadores estão a mudar. Este facto constitui enumeras novas oportunidades de investigação, e mesmo o desenvolvimento de novos produtos comercias.

Simular e prototipar complexos cenários dentro da filosofia de computação obíqua que suportam este novos modelos, é uma tarefa que tem sido provada também ela complexa, facto sublinhado por várias investigaçes.

O próprio Second Life^{TM} foi sujeito de uma grande expansão e alvo de grande atenção mediática, mas este simulador tem provado suportar aspectos mais sérios de computação, baseado na sua capacidade de permitir programar intrinsecamente e criação de objectos.

Todos estes pontos formam a premissa da presente tese. O Second Life^{TM} e o Opensim (a versão "open-source"do simulador) foram usados como ambientes de simulação para aplicaçes moveis e ambientes inteligentes dentro do prisma da computação ubíqua, avaliando assim as suas capacidades neste domínio e revelando possíveis falhas.

Dentro deste contexto, duas simulações foram efectuadas: uma aplicação social baseada em proximidade e um ambiente inteligente (smart room), que expuseram as capacidades de simulação do Second Life^{TM} e do Opensim. Paralelamente, os níveis de performance em termos de capacidade de processamento dos simuladores foram medidos e comparados, usando diversos cenários de simulação, permitindo julgar a estabilidade dos mesmos.

O conjunto de resultados permitiram concluir quem ambos os simuladores retêm as características necessárias de um simulador de computação ubíqua, com especial foco no Opensim que, graças à natureza "open-source"da sua implementação, oferece uma flexibilidade inigualável.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Ricardo Morla, whose expertise, understanding, support and unmeasurable patience made not only this project possible, but also enriched my graduate experience.

I must also acknowledge INESC Porto for providing the infrastructures necessary, and enabling a great work environment, that contributed for the wellbeing of this thesis.

Also, a special thanks to Professor Paulo Frias (Professor of Communication Science at the University of Porto) and responsible for the University of Porto island in Second Life, for providing the space for implementing a virtual research lab within Second Life™.

I would also like to thank my family and friends for the support they provided me through my entire life and in particular, I must acknowledge my girlfriend and best friend, Susana, whose love, encouragement and support, pushed me to thrive for more.

The Author

"Computing is not about computers any more. It is about living."

Nicholas Negroponte

Contents

List of Figures

List of Tables

Glossary

With this glossary we intend to expose, in a brief and objective manner, the meaning of all important terms and expressions used throughout this dissertation enabling the reader to find any unknown term.

- **Avatar** - An avatar is a visual representation of an agent within Second Life^{TM}.
- **Agent** - An agent is a client's presence within a simulator.
- **HUD (heads-up display)** - 8 attachment slots that are visible only to the user to which they're attached and were intended to allow scripters to create user interface elements.
- **LinkSet** - several prims that are linked together, are called a "linkset"(also link or linked set) and they that act as (or, by its definition, are) a single object.
- **LSL** - The Linden Scripting Language (LSL) is used for creating interactive content in Second Life^{TM}.
- **Object** - An object is a 3-dimensional geometric entity in Second Life^{TM} that can consist of a just a single primitive or multiple prims linked together.
- **Owner** - The owner of an object is the user who rezzed it.
- **Prim** - A primitive (prim for short) is the basic building block (made up of polygons) of Second Life^{TM}.
- **Rez** - When you create an object within SL, you "rez"it.
- **Sim or Simulator** - The Second Life world consists of many, interconnected, uniquely-named simulators (also referred to as sims or regions).
- **UUID (Universal Unique IDentifier)** - are 128-bit numbers assigned to any asset in Second Life^{TM} and are guaranteed to be unique inside their scope.

Chapter 1

Introduction

1.1 Context

As new mobile and sensing technologies emerge from the world's hunger for information, new computing paradigms rise, creating numerous opportunities in hardware and software development. Ubiquitous Computing [1, 2] offers the concept of a more natural interaction between human and machine, that is only possible as these technologies reach a wider range of consumers.

This ubicomp (**ubi**quitous+**comp**uting) philosophy introduces a rather different model of human-machine interaction, when comparing with the one foreseen throughout our everyday experience. Discarding the "intimate computer"concept suggested by Kay[3], ubicomp defines a notion of invisible computing, where hardware is embedded into the environments and "effectively invisible to the user"[1], focusing on the task itself and not on the hardware necessary or how to use it. This concept of invisibility introduces a natural interaction between user and machine, where computers are moulded to serve our everyday needs and even magnify or created new realms of possibilities. Like Adam Greenfield mentioned [4], the fact that a user simply does not know how to interact with a particular application, is caused by a design flaw in that system and should never intimidate the user. In the ubicomp future, computer interaction will be as natural as drinking a glass of water.

To pursue this dream of Ubiquitous Computing, many mobile and pervasive applications have been designed and developed [5, 6]. Due to their mobile nature [7] these equipments have interactions with the environment that often can only be tested in the real world. Pervasive environments are often too complex to simulate with current tools, thus research into prototyping and simulation tools for ubicomp systems being a relevant topic.

Using Second Life™ as the simulation backbone, may offer many advantages. This is the main subject of this thesis, and perhaps also taking SL further, as the Web 3.0 interface.

1.2 Objectives

The goal of this thesis is to evaluate the extent to which Second Life^{TM} can be used for prototyping and simulating mobile and pervasive applications.

Although some authors refer to Second Life^{TM} [8] as a new breaking environment with many educational and prototyping possibilities, support beyond basic Second Life^{TM} primitives to simulate a full pervasive room or a urban/social scenario [7] with multiple users is yet to be tested.

To this end two case studies were conducted. The first is in the field of pervasive environments, using both Second Life's^{TM} built-in primitives and functionalities, and OpenSim's possibility of developing new primitives to evaluate the simulator's performance, stability and complexity. As mentioned earlier, Opensim is the open source version of the SL simulator. Experiments were conducted not only on the simulated application but also in a mixed reality system supporting dual mode (off-world and in-world) communications.

The second simulation focus on social mobile applications, and is intended to expose Second Life's^{TM} limits in terms of software application complexity and evaluate if it's own social network could be used to test applications that by nature need a wide range of users to fully function, as a pre-study before real life implementation - prototyping - [9, 10].

1.3 Results

The work developed revolved on two case studies conducted, that allow us to conclude that both Second Life^{TM} and Opensim have the built-in capabilities to support simulations in the ubiquitous computing domain, more specifically in mobile-social and smart room environments applications.

In the course of the two main simulations scenarios, we were successfully able to implement an Publish/Subscribe middleware that supported a management application, allowing to control an array of synchronized touch screens, a Bluetooth middleware to support communication for a social proximity application focusing on friends, current neighbours and previous visitors to certain hot spots. It was also implemented a graphical user interface for that application, running on a simulated mobile device.

Parallel to this evaluation of simulation complexity, the pervasive environment case study, implemented in Second life^{TM} and Opensim in a wider range of test scenarios, led us to concluded on the greater performance and efficiency of the Opensim simulator, allowing the development of new primitives into the server's open source code. In fact Opensim's delay times in the delivery of the pub/sub simulation were 140 times smaller than on Second Life^{TM} using an in-world server, and a surprising 1320 times faster than Sl with an off-world server based on PHP website with an SQL database.

As it will be explain further into the thesis, Second Life^{TM} proved to support the development of software for simulation and prototyping purposes to a certain point of

complexity. Beyond this mark, Opensim is the only one to allow complex scenarios, based on the development of software structures right into the simulator's own code, and although not offering a easy usability as Second Life™, it has proven far more flexible.

1.4 Thesis Structure

This thesis is divided into 7 chapters and two appendix sections. The first chapter is the present introduction.

In the second chapter basic notions of ubiquitous computing and simulation and prototyping are presented, so the reader can better understand all the basic concepts necessary to comprehend this thesis, along with the research work already developed in the correspondent fields.

The third chapter introduces the two main technological tools used in this project, Second Life™ and Opensim, presenting their capabilities and major characteristics.

The fourth and fifth chapters report the two applications that were simulated, a smart room application and a social mobile appliction, with a corresponding contextualizing of their domain of application, and architecture, implementation issues functionalities.

The sixth chapter reports the simulations results with the correspondent analysis, that are compared with initial objectives on the seventh and final chapter, concluding this thesis.

Lastly, two appendix sections present the most relevant sections of code developed, regarding the two simulations.

Chapter 2

Related Work

2.1 Introduction

To better understand the premise of this thesis as a whole, it is necessary to know the existing work on simulation environments within the ubiquitous computing field, and the important concepts of this domain.

This section serves this purpose, by introducing specific definitions and existing projects related to ubiquitous computing, simulation, and Second Life™. Of course all the notions related to these themes will be fully explained, using when possible references from other authors.

The first section reports on ubicomp notions and general concepts, as an introduction point to this research area. The second section travels through the simulation environments previously developed in this field.

2.2 Ubiquitous Computing

2.2.1 Introduction

Ubiquitous Computing, as referred in chapter 1, is the next generation computer-user interaction paradigm, where information processing and manipulation are thoroughly integrated into everyday objects and activities, making this interaction as natural as possible, e.g. a smart glass can automatically detect when the client's drink is empty and call the waiter to fill it up.

Many contemporary devices already fall in to the ubiquitous computing category like mobile phones with digital audio players and RFID (radio-frequency identification) technology.

Since Mark Weiser coined the phrase "ubiquitous computing", during his tenure as Chief Technologist of the Xerox Palo Alto Research Center (PARC), much research has been done in this field, including distributed computing, mobile computing, sensor networks, human-computer interaction, and artificial intelligence.

2.2.2 Context-aware pervasive systems

Mobile systems that can sense their physical environments, i. e., their context of use, adapting themselves accordingly. Such systems and applications are an important component of a ubiquitous computing (or pervasive computing if you will) environment. By context these systems subtend: physical location, nearest users and nearby resources.

Although location is a primary capability, it does not necessarily capture all things of interest that are external to a mobile application. Context-aware in contrast is used more generally to include nearby people, devices, lighting, noise level, network availability, and even the social situation, e.g. , whether you are with your family or a friend from school.

Dodgeball[11] is an example of mobile social software, using a context-aware approach. Whenever a user is near popular places, like certain bars or restaurants, or even known friends, the system based on a global database informs the user of the presence of friends or of parties near his present location. The notifications use the Short Message Service (SMS) .

ScienceSifter[12], yet another example, uses RSS Feeds, to enable communication between colleagues, throughout the scientific community and introducing a notion of context grouping to better implement the information database. There are even projects involving managing human activities in space operations [13]. And concerns about awareness versus privacy[14] have been risen in similar projects.

Another important aspect is enabling context aware characteristics to user profiles, within social applications, as referred in [15]. Enriching profile information with context aware data, allows a more refine profile matching, which translates into more accurate social encounters with social peers with similar personalities and tastes.

2.2.3 Mobile Social Systems

With wireless technologies and communication protocols having a major impact in everyday life, mobile social applications are bound to become more and more embedded in people's routine. Indeed wireless information exchange is widely use, due to technologies like Wi-Fi, RFID, GPRS and Bluetooth. The mobile success is also credited to computer progressive under sizing[5]. Dodgeball, as referred earlier, is a good example.

Another example is the MediaAlert[16] system that provides a automated system for multimedia content monitoring and alerting. It is also important to acknowledge that most context and activity systems, like referred early, use mobile technologies.

Genie[17] is a social mobile application that enables information flow between nearby users. A user posts a series of questions to be answered and when two or more users pass near each other, they can view and possibly answer other user's questions. This application allows social networking, as personal profiles are exchanged. Pirate[17] is yet another example, specialized in matching the users' own personal music taste. WALID[17] as the same functionality basis, but developed to task interchange, as users post a series of

real-world tasks, that other users can view and offer to realize, on a basis of a friendly negotiation. Note that all these three application relay on Bluetooth technology to detected and communicate with other users.

To develop mobile applications, some authors started with a publish/subscribe abstraction and later introduced a notion of location-aware subscriptions [18]. The Pervaho platform[19] provides a middleware layer which can be used to developed ubiquitous mobile applications, e.g. the Traffic Jam Warning system and Ubiquitous Flea Market, using a context and location based publish-subscribe philosophy.

2.2.4 Ubiquitous Applications

As referred early, many current hardware and software applications fall into the ubiquitous category. But many more are still on research. The Ubi-Finger[20] is a prototyping example of next generation user interface with real world ubicomp applications. Using an IR transmitter, touch, bending and acceleration sensors, and a network connection the Ubi-Finger presents various possibilities including Control of Information Appliances, Efficient Window Scrolling of Computers and Dynamic Presentation. The Ubi-Pen [21] is another ubicomp research example.

Wearable computing also plays a important role in the ubicomp paradigm, potentially integrated with various research on mobile sensor networks[22, 23]. Research into wearable computers has allowed the exploration of different applications, from monitoring vital signs to accessing the internet in your clothes.

2.2.5 Conclusion

Ubicomp is here to stay as the future computing philosophy and the current research projects point to a future were the user will not adapt itself to the computer, but just the opposite. Everyware, as referred in the Adam Greenfiel presentation [4], is the future, and simulating/prototyping such applications is of the most importance.

2.3 Simulation and Prototyping

2.3.1 Introduction

Prototyping is the process of quickly putting together a working model (a prototype) in order to test various aspects of a design, illustrate ideas or features and gather early user feedback. This process is often treated as an integral part of the system design process, where it is believed to reduce project risk and cost.

Often one or more prototypes are made in a process of iterative and incremental development where each prototype is influenced by the performance of previous designs, in this way problems or deficiencies in design can be corrected.

When the prototype is sufficiently refined and meets the functionality, robustness, manufacturability and other design goals, the product is ready for production.

But to ensure an even more refined product and reduce project related costs, a first phase of the prototyping process can be considered via computer simulation, using a simulation environment. This prototype simulation can result in a major advantage, mainly due to the complexity and hardware costs involving developing mobile and pervasive applications.

2.3.2 Simulation Environments

The use of simulation technology in ubiquitous computing, and other fields in general, is of particular importance to product developers and researchers all over the globe. This fact arises from a number of reasons, including the ones given in the prior 2.3.1 sub-section.

Pervasive environments and mobile applications often demand a wide range of test scenarios, which may require a high number of simulated devices of various kinds. In a smart office, for example, mobile devices have to co-exist with local servers and fixed interfaces like projectors, flat screens and much more. This simple proposition implies a complex relation between all devices, so a robust software model has to be modulated, but also the individual difficulties of every type of hardware, and finally the communication backbone, including wire and wireless technologies. So evaluating these scenarios, applications, communication protocols and so forth, is clearly achieved, in a cost effective manner, with rigorous simulation and prototyping tools.

Various ubiquitous computing simulation tools - the simulators - have been developed over the years. Upon a closer look at the related projects, two major groups emerge: the truly ubicomp simulators, which are specifically designed tools for this domain; and the derived ones, from other related domains like sensing and networks. So a further analysis of each major simulators, of both groups, will be presented next, to define comparative goals to Second Life™ and Opensim's simulation capabilities.

2.3.2.1 Dedicated Ubiquitous Computing Simulators

Several native ubiquitous computing simulators have emerged in the past years. The most referenced is UBIWISE [24], that was developed at Hewlett Packard by Barton et al. This simulator consists of two simulators: UbiSim, which generates 3D virtual space based on the Quake-Arena III rendering engine and serves to simulate the first person view of the simulated environment; and WISE, which displays the 2D graphical interface of mobile devices, using a Java-base interface. UbiWise allows hardware creation of various kinds of devices with a wide range of capabilities but it doesn't complement network simulation aspects. The WISE device view permits realistic user interaction (one of the simulator's most important achievements). This distinction between device and environment views

along with complex user interaction (like manipulating a digital photo camera) proves an in-world challenge to be resolved.

A similar is [25], using Quake III Arena to simulate sensors and actuators when evaluating and testing mobile services.

TATUS[26] was developed with a similar methodology by using the Half-Life 3D game engine to offer ubiquitous computing environments simulation. This program focuses more on user interaction with the simulated environment than UBIWISE, but on the under hand it also doesn't simulate any aspects related to communications networks. To this purpose integrating with wireless networks simulators was developed and implemented with the TOSSIM TinyOS sensor network simulator. So, in fact, TATUS is a user-interaction module, which can be used with other simulation applications to cover all ubiquitous computing major aspects.

UbiREAL [27], unlike UBIWISE and TATUS, has its own 3D environment, and provides a virtual test bed for ubiquitous applications in a Smartspace context. It also enables communication simulation among devices, from MAC to Application layers and physical quantities representation, like temperature or humidity, and their relation with some devices, e.g. a radiator affecting the room temperature. As the name implies, the UbiREAL (Realistic Smartspace Simulator for Systematic Testing) simulator has a testing mechanism which systematically generates possible contexts and checks whether the system runs expectedly, along with a GUI for the virtual Smartspace design, simulators for physical quantities and the already mentioned network simulator.

Second Life already provides this GUI design views, but network and physical quantities simulation along with testing mechanisms simply don't exist.

The simulator described in[28] is yet another approach to ubiquitous computing simulation. This simulator in part devoted to context-aware behaviours, is based on a distributed architecture, providing separate interfaces for network and location simulation and flexibility by a Web Services based API. It supports real application code, and interface with the ns network simulator. A remote medical monitoring application was used to exemplify the simulator's performance.

Finally, the UbiWorld[29] is a earlier study on the concept of ubicomp simulation (note the date of release, 1996, previous to all the above simulators) . This experimental system combined "virtual reality, advanced networking and supercomputing to explore the implications of ubiquitous computing". Based on the CAVE (CAVE Automatic virtual Environment) it provided virtual environment simulation, but with the hardware limitations associated to that time. This visual problem, does not happen in Second Life, has object creation is already provided. Important to acknowledge from this project, are the technical challenges identify which can almost summarize this project's requirements.

2.3.2.2 Derived Ubiquitous Computing Simulators

Several sensor network simulators have been developed over the years, and SENS[30] is an example. This application-oriented simulator offers models for various sensors, actuators, environments and a framework for application testing. The simulator consists of several simulated sensor nodes interacting with the environment, and each node consists of application, network and physical components, fact that creates some extensibility to the simulator, but in terms of ubiquitous computing, the environment model does not seem to suggest possible future simulation capabilities. But wireless communication characteristics, in particular wave propagation, are fairly modulated in this application, along with rich simulation information access. This wireless model may play a fair role in our project.

The TOSSIM[31] TinyOS simulator, previously mentioned, is a discrete event simulator for OS sensor networks. Its main objective is to provide high fidelity simulation for TinyOS applications, focusing more on this applications execution, and discarding complex real world environment simulation. TinyOS is an open-source and free operating component-based system and platform targeting wireless sensor networks (WSNs), written in the nesC programming language. This OS has developed to incorporate the smartdust model which defines a hypothetical tiny microelectromechanical gadget, like a sensor, robot or other devices with wireless communications capabilities for physical characteristics measurement. The smartdust is one of the ubiquitous computing milestones, which will allow more complex applications within the context-aware theme.

Other examples of sensors network simulators are SensorSim[32] and GloMoSim[33]. This last simulator has a wide range of simulation parameters, from the physical to the application layer in terms of communication protocols, offering a wide range of choices.

A different kind of simulator, are robot simulators. Although this domain may appear to fall outside the project's goal, it's important to see the current options. Robot simulation and related aspects often share the same requirements from their simulators, as more pure ubiquitous computing simulations.

Webots™ [34] is an example of mobile robotics simulation software that provides you with a rapid prototyping environment for modelling, programming and simulating mobile robots, including even direct program transfers to LEGO™ Mindstorms™ via RCX cross-compilation of Java WebotsTM controllers based on LeJOS, a java-based alternative firmware for the RCX. Robo Sim [35] is another 3D robot simulation environment example which uses Java.

Lastly, a number of technical simulations have been developed on Second Life™. These projects include Cristina Lopes' simulation of a transit system, called the SkyTran, proposed by the Irvine-based transportation company Unimodal Inc, and smaller projects like the Vital Lab[36] of the University of Ohio, artificial intelligence projects using bot-like avatars and even medical simulations.

2.3.3 Conclusion

Simulation, along with prototype development, plays a major role in both academic and industrial applications/products research. So using Second Life™ as a inexpensive simulation tool, unlike the often complex, costly, and very specific simulation environments, would represent a breakthrough, proving to be an important tool to researchers.

Chapter 3

Technologies and tools

3.1 Introduction

In this section the two main technological tools, Second Life™ and Opensim, are presented with detail and contextualized, allowing the reader to understand the project's implementation and orientation, the premises behind the work that was developed, and subsequent results with the up most inside knowledge.

3.2 Second Life™

3.2.1 Introduction

Second Life™ (often abbreviated as SL) is an online virtual community, based on a 3D world where users interact with each other through motional avatars that enable a complex web of social encounters resulting in advanced social networking combined with general aspects of a metaverse. Second Life came to international attention via mainstream news media in late 2006 and early 2007.

The term metaverse was first used in Neal Stephenson's 1992 novel Snow Crash, and is now used to describe immersive 3D virtual spaces, as Second Life™. The novel referred a virtual environment, where the physical limitations of real life didn't stress multi-user communication and social-economical interactions. Residents can explore the virtual world, meet other Residents, socialize, participate in individual and group activities, create and trade items (virtual property) and services from one another. The Second Life™ metaverse consists of individual islands that can, or cannot, be related to each other.

Although all the islands are supported by the Linden Labs servers - company responsible by SL's development - each island is bought by individuals or groups of users to build it's infrastructure as they wish. Businesses like avatar clothing or virtual housing thrive within the community, reaching nowadays profits of several thousand of American Dollars. Some islands are dedicated to educational purposes and group discussions, as others

offer more recreational assets like live music, that include real-life artists performing in real-time via web-streaming to Second Life, and game based themes.

Access to the Second Life™ universe is free and requires a basic user registration procedure along with the free download of the Second Life Viewer, the client program that enable "Residentes"to log on to the virtual world.

Often referred to as a game, this description does not fit the Second Life™ experience. It does not have points, scores, winners or losers, levels, an end-strategy, or most of the other characteristics of games. In fact, as referred earlier, in-world residents can get jobs and earn Linden Dollars (L$) - the in-world official currency - to spend on whatever items or services they wish.

3.2.2 Building

The great breakthrough of Second Life™, is it's freedom to be creative. With the right know-how residents can develop almost everything.

This creative freedom is mainly associated with the possibility of creating 3D objects using primitives, commonly referred as prims, that enable the construction of any given real-life object. Prims are the most basic building blocks in Second Life™, that are common geometrical figures, e.g., cubes or cylinders. With the right manipulation these primary geometrical shapes, when combined together, can form various complex objects that almost appear real. But these constructions would fall much behind for not the possibility of applying textures to objects and even avatars. Textures are simply JPEG, TGA, PNG or BMP images that users can upload to the metaverse at price of 10L$ per upload.

Object creation is simple and easy, although complex objects require both knowledge and art. The creation of a simple prim cube is shown in figure 3.1.

After the prim creation, it's parameters like cut, hollow, twist, taper, shear, size, colour, textures and position can be modified at will. Also, a very important aspect, is the physical parameter of each prim, that when enabled applies the physics engine rules to that prim, and thus forces like gravity and collisions now affect the prim. Although the the most used physics engine in SL is the Havok 1.8.3, some sims are already using a beta version of the Havok 2 Physics engine[37] that has proven to be superior in various aspects.

Various prims can be logically linked via the Link option on the Second Life™ viewer or via script. A set of objects linked to each other are called a linket and act as a single object. Every linkset has a parent prim which is defined by being the last selected prim before the link command. Linked prims have the advantage of being able to communicate to each other via a special kind of message called the link message that will be explained in the next sub-section.

Using scripts, two special features can be applied to prims that give a more natural feel. The first one is called the Particle System and is used to simulate non-object visual

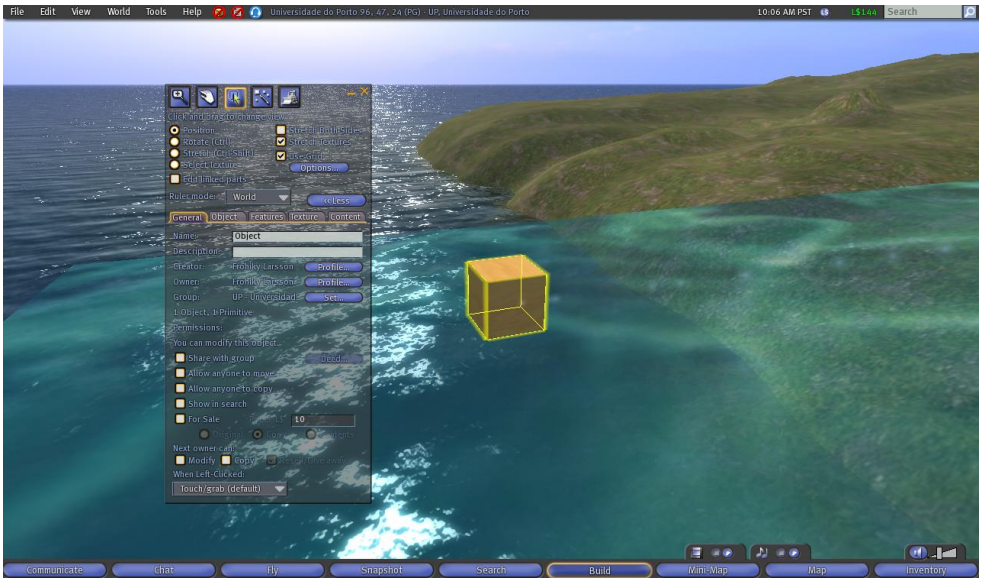

Figure 3.1: Creating a Prim

effects like water, fire, smoke, sparks and others. This effect not also allows applying textures to a particle but also doesn't contribute to the sim's (island) load, to a certain point, as its implementation is entirely on the client-side (the SL viewer). The second are the lighting effects that are generated from a prim via script or the sim's day cycle lighting, generating shadow and other features.

We built a simple house prim to exemplify the simplicity of building SL. This construction project took about 40 minutes to complete - note that this was the first time the author attempted to build complex housing.

As you can see in figure 3.2, a fair result was accomplished, with very little effort, showing that Second Life has easy to use construction capabilities, making environment manipulation and generation accessible to most users, although advanced content requires both art and ingenuity. Apart from existing in the sim virtual physical space, prims can be attached to the avatar, giving the feeling that the user is holding that same object, like mobile phones or simply a glass of water. This attachment that either be forced by an option in the object menu, or via script.A special kind of attachments are the HUDs (Heads-up Display), as shown in the top left corner of figure 3.3, that consist of 8 attach points with the particular characteristic of being visible only to the user to which they're attached. They were design to allow users to develop interfaces elements since, apart from a small number of functions, scripts run on HUDs.

Finally another important aspect of building is, much like real life, the building rights. Like previous mentioned the islands are privately owned, and this fact can pose as problem to Residents that don't have much money to spend on their second life. But fortunately, all around the SL universe (or more correctly metaverse) there all special parcels of land,

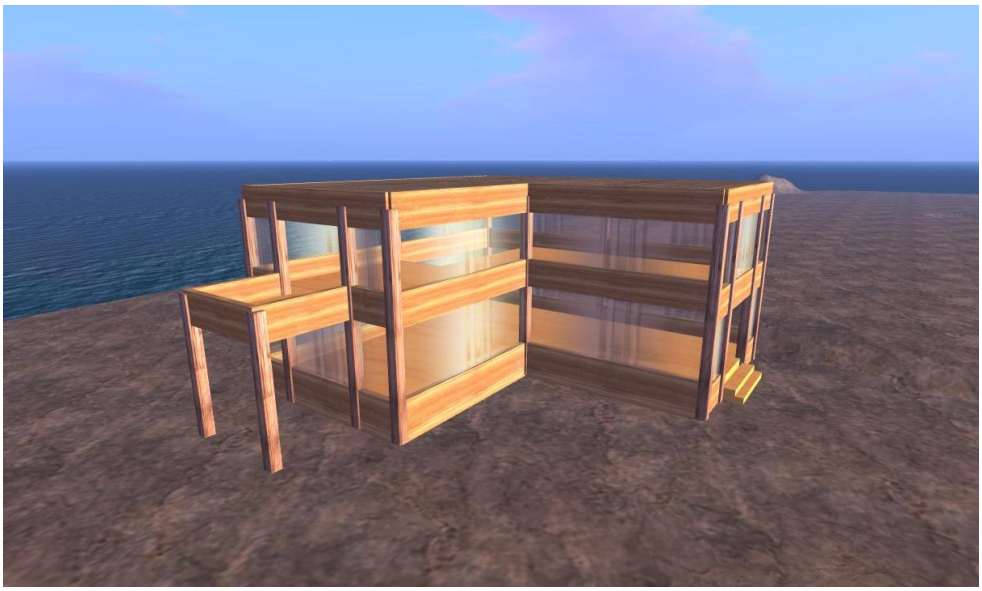

Figure 3.2: Building a House

mainly called Sandboxes that offer free work space to build and test scripts.

3.2.3 Scripting

The ability to build on Second Life™ is alone a great advantage to the user, opening creative doors that previous social network based applications didn't offer. But cleverly Linden Labs also offered within its metaverse the possibility of applying scripts to objects, introducing creativity dynamic that has insured SL's current success.

These scripts are written in LSL (Linden Scripting Language), a programming language created specifically to be used in Second Life™, to control the behaviour of in-world objects, among other things.

LSL (somewhat similar to the C language) is a state-event driven scripting language, in the sense of a finite state machine. A script consists of variables, function definitions, and one or more named states. Each state contains a description of how to react to events which occur while the program is within that state. The system sends events to the script, such as timers, movement, chat (from other agents), email, and collisions (with objects in the virtual world). Scripts can change most aspects of the state of the object and communicate with other objects and agents. As soon as a script is added to an object, it begins to execute. The universal "hello world "LSL script is shown next.

```
default
{
  state_entry()
  {
```

Figure 3.3: Prim Attached as HUD

```
    llSay(0, "Hello, Avatar!");
  }

  touch(integer total_number)
  {
    llSay(0, "Touched.");
  }
}
```

The next few sub-sections explain in more detail the more important aspects of LSL.

3.2.3.1 States and Events

When a script is created, the default script is automatically created and consists of the universal "hello world". Also previous referred, LSL is a state-event driven scripting language, which in practice means that every script has a default state - much like the mandatory main() function in the C language. The default state is where all LSL scripts start, but using the expression:

```
state user_define_state;
```

the script will jump to the specified state, where another set of instructions can be executed and events managed.

Important to acknowledge is that every time the a script is compiled, reset or loaded, it will execute the "default"state. After the default state definition, additional states can be declared and the respective event handlers.

Within states LSL works based on events that are triggered by scripts or the sim itself, e.g., if configure to listen, a chat message detected within range, will trigger a listen() state in the script. Unlike states, events can contemplate arguments that define the type of event, e.g., using the listen, the identification and the message is retrieved using variables passed as arguments of the event. Pure scripted events can be trigger by timers for example.

3.2.3.2 Data Types

LSL offers a medium range of data types, especially design to manipulate the in-world objects and are as shown in table 3.1.

Table 3.1: Data Types

Type	Description
Integer	A whole number ranging from -2,147,483,648 to 2,147,483,647
Float	A decimal number ranging from 1.175494351E-38 to 3.402823466E+38
Vector	Three floats in the form <x , y , z >
Rotation	A quaternion rotation, made up of 4 floats, <x , y , z , s >
Key	A UUID (specialized string) used to identify something in SL, 36 characters long
String	A sequence of characters, limited only by the amount of free memory available to the script.
List	A heterogeneous list of the other data types.

The integer, float and string types are in all similar to its homonymous in the C or Java languages. The vector and rotation are specials data types, as they were introduced to manipulate all issues regarding positioning, rotation and movement within the sim. The key type, also unique when comparing with other programming languages, is one of the more important types. It consists of a specialized string that constitutes a UUID (Universal Unique IDentifier) for an agent, object, sound, texture, other inventory item, or dataserver request in Second Life. Here is and example of a typical key:

```
"66864f3c-e095-d9c8-058d-d6575e6ed1b8"
```

Last, instead of arrays, LSL uses lists. This type consists of a heterogeneous list of the other data types, and although insertion is completely dynamics, i.e. the LSL command for inserting in a list is:

```
myList = myList + [new_item];
```

regardless the data type of the “new item”. Retrieving a element, unfortunately doesn’t share this transparency. The user must know exactly what type constitutes that element, as every data type has a different function, like shown in table 3.2.

Table 3.2: List Return Functions

Function	Description
llList2Float	Returns the float element
llList2Integer	Returns the integer element
llList2Key	Returns the key element
llList2Rot	Returns the rotation element
llList2String	Returns the string element
llList2Vector	Returns the vector element

3.2.3.3 Functions

Apart from the native 332 native functions embedded in LSL, this language allows the scripter to build other functions avoiding useless coding repetitiveness. The function syntax is similar to the one used in the C programming language as shown here:

```
return_type function_name (type argument1, type argument2 , (...) )
{
  function code;
  (...)
  return value;
}
```

Regarding the LSL native functions, the most relevant ones, will be subject of a brief explanation and contextualisation in the following sub-section, giving the reader a richer inside knowledge of the work developed, discussed in the next chapters. To allow a better understating, they will be aggregated into modules.

Communications These functions allow scripts to communicate with other in-world scripts, users and even programs outside the Second Life^{TM} metaverse and regarding in-world communication, there are three main scenarios, as explained next.

The first one is script to-script communication where objects can exchange messages via chat messages, when not on the same link set. The chat message is sent using the llWhisper, llSay, llShout and llRegionSay. All four functions take two arguments, an integer that defines the chat channel used, and the message to be transmitted with the following syntax:

```
llSay(integer channel, string message);
```

The 0 channel is the common chat used by avatar to communicate with each other, and is often referred as the public channel, so it can be also used to script-user and user-script communication. All other 4,294,967,293 channels are called “private ”channels, requiring special commands. The four previous chat functions only differ in the range in which the message is received, as the reader can observe in table 3.3.

Table 3.3: Communications In-World Distance

Function	Range(radius in meters)
llWhisper	10
llSay	20
llShout	100
llRegionSay	Broadcasts to the whole sim

To receive these chat messages, the recipient script has to use the llListen function with the following syntax:

```
integer llListen(integer channel, string name, key id, string msg);
```

It creates a listen filter, to enable the reception of chat messages on a particular channel or from a particular object identified by the name or id argument. An integer is returned that identifies the handler, to enable future modification or even removable of that particular listen filter. After the function is invoked, a chat message received that meets the filter parameters, will trigger a listen event:

```
listen(integer channel, string name, key id, string message)
{
   used_code;
   (...)
}
```

All arguments are used to identify the sender, and the message itself. Note that if a Listen() is invoked, but the listen event is not properly declared, the chat message will be received by the script-engine's event queue, but the LSL script will not.

On the other hand, if the prims are in the same linket, message exchange can be implemented using the llMessageLinked function:

```
llMessageLinked(integer linknum, integer num, string str, key id)
```

Not only is the linked message a much secure way of communicating, it allows the exchange of a string, but also an integer and key which maximize the prim to prim communication. The integer linknum is used to refer which prim is the recipient, as every prim in a linkset was a link id number. Another advantage of the linked message, is that is does not need any kind of a listen filter function. When a linked message is received, the script automatically generates the link message event, which is received using the following LSL event:

```
link_message(integer sender_num, integer num, string str, key id)
{
   code;
```

```
    (...)
}
```

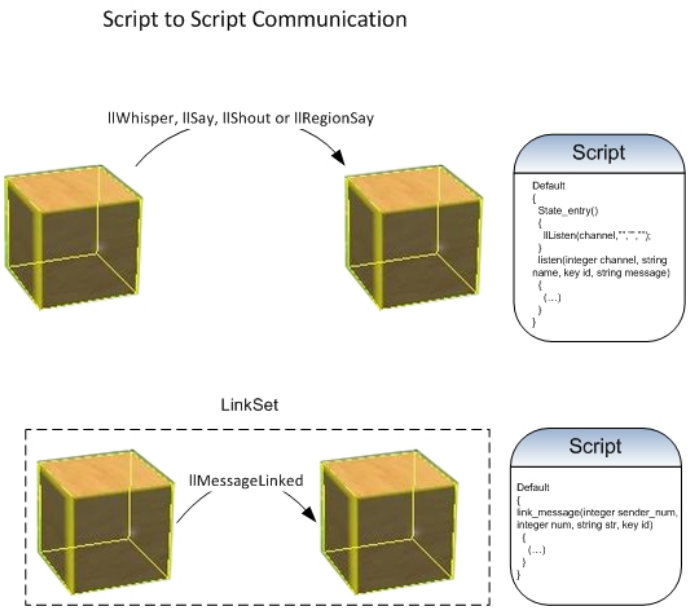

Figure 3.4: Script to Script Coommunication

Regarding script to user communication, a script can send messages to a resident using the llSay on the public chat channel, like previous mentioned. More secure options are the llOwnerSay(string message) that sends a message directly to the owner of that object/prim and the llInstantMessage(key user, string message), sending a IM to the user with the specified key. In the opposite direction - user to script - a user can send a chat message on the public chat channel using the chat line on the SL Viewer. To use a specific channel, the users can type:

```
/channel_number message
```

and the script needs only to be listing on that particular channel.

To implement off-world communication, a script can send messages outside the Second Life^{TM} metaverse invoking the function:

```
key llHTTPRequest(string url, list parameters, string body);
```

The page specified in the url can receive data, i.e., passed by php arguments and fetched using the $_GET[] array.

On the opposite side, a program can send data to a Second Life script using a XML remote procedure call (XML-RCP). To enable this, a script opens a data channel using the llOpenRemoteDataChannel(), accessible off-world by the url:

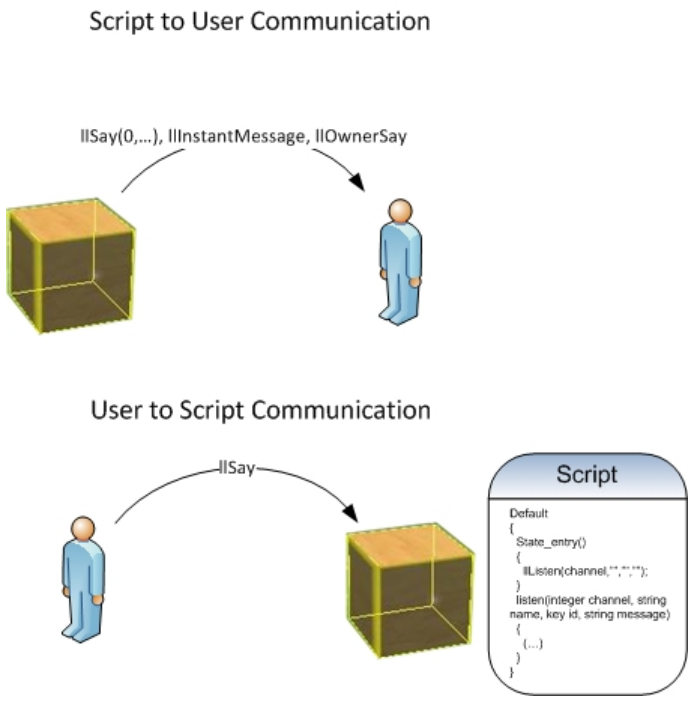

Figure 3.5: Script to User Coommunication

```
xmlrpc.secondlife.com
```

and specifying the UUID of the channel. This will create a remote data event on the script:

```
remote_data(integer type, key channel, key message_id, string sender,
integer ival, string sval)
{
   code;
   (...)
}
```

Sensor The llSensor and llSensorRepeat function allow the script to detect nearby objects and/or avatars and have the following syntax:

```
llSensorRepeat(string name, key id, integer type, float range, float arc,
float rate);
```

This function will every rate seconds scan the area define by arc and range, for the name and id, and generate a sensor event if found or a no_sensor if not. Note that the syntax of llSensor only differs in the rate argument, as it does not exist in this function.

Within the sensor event, the detected objects/avatars are available using the detection functions, e.g., llDetectedName and llDetectedKey.

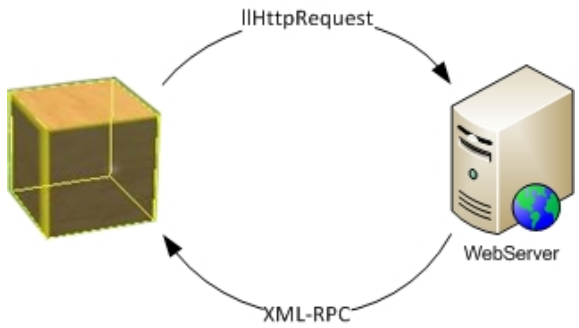

Figure 3.6: Off-World Communication Coommunication

Primitive/Objects and Agent/Avatar As mentioned in sub-section 3.2.2, prims have numerous attributes, which can be modified via script. Functions like llSetColor, llSetAlpha and llSetTexture are an example of some that directly change the prim's layout. Another important function is the llSetText, which displays a string just above the prim, and is very useful in terms of message output to users in general.

3.2.3.4 The Future of LSL

Currently Linden Labs is trying to use Mono as the next generation scripting engine within Second Life™.

Mono is an open source implementation of Microsoft's .NET development platform, including an implementation of an ECMA/ISO Common Language Infrastructure (CLI) standard virtual machine.

The CLI is designed to allow multiple languages to be compiled to a Common Intermediate Language (CIL), to be run safely on any Common Language Runtime (CLR) implementation like Mono or .NET and to be able to interoperate by using a Common Type System (CTS).

In practice this means that, if the Mono engine is accepted, LSL scripting compiling will be much faster (up to 200 times faster) and maybe even allow outside programming languages to be used in Second Life™.

3.3 OpenSim

3.3.1 Introduction

Opensim [38] is a open source server for hosting virtual worlds, and can be used to simulate Second Like™ like environment. This BSD licensed virtual worlds server is written in C# and can be executed under the Microsoft .NET and Mono frameworks, which means that any OS can be used. As previous mentioned, Mono [39] is another open source project,

designed to develop and run .NET client and server applications on Linux, Solaris, Mac OS X, Windows, and Unix.

OpenSimulator features two modes of operation: standalone, where a single process controls the simulation, and is intended to private use, and the grid mode, were several processes control the simulation, i.e., user, assets, inventory and islands are run on separate serves. This mode contemplates several sims simultaneously, that can be connected to other public grids.

In terms of social network, there are already a list of public grids, that can be used by everyone, much like Second Life's™ main public grid (owned by Linden Labs).

As this thesis is being written, the current state of Opensim is considered to be in an alpha code version. In practice this means that some Second Life™ features, like scripting, building and physics engine are not fully operational.

3.3.2 Building

Like Second Life's™ main grid, OpenSimulator allows users to build and create objects at will. Although minor bugging issues have clouded the public grid of the open source application, there are no major problems in the current version. But in a major advantage point, the private owned sims allow building without the commercial restrictions present in the Linden Labs grid, i.e., with OpenSim every user can create one or more islands, allowing the building permissions to be as big as the server memory and processor capacity.

The version used also contemplated a simple but effective physics engine that prevented major unrealism behaviour, present in previous releases, like the avatar passing through objects. Further versions are expected to include Havok engines.

3.3.3 Scripting

Like Second Life™, OpenSim enables user scripting, but not only using LSL, but also C#, VB.Net and JScript.Net. All these new languages allow a more liberal use of scripting, but their behaviour still is based on LSL, in the sense that residents still have to use ll-functions like llSay and handle LSL events, like the C# script shown next:

```
public void default_event_state_entry()
{
    llSay(0, "Hello World.");
}
```

Regarding the LSL implementation itself, a pre-study of OpenSim's code revealed that not all aspects of LSL scripting are functional in Opensim version used in this project.

Early tests showed that the list type as not been fully developed, along with state changing, confirmed by the data on the Opensim's webpage[38]. Also regarding off world communication functions, more specifically the ones involving XML-RPC, are yet to be

implemented, as well as several others. This posed immediately a problem and it's discussion and resolution, along with several others, will be demonstrated on the next chapter.

But most importantly, aside from lack of full LSL implementation, Opensim due to its "open"nature, allows the development of new LSL primitives, that will be fully discussed in chapter 4.

3.4 Viewers

To connect to the Second Life^{TM} public grid or a private one, running on the Opensim server, the user needs a viewer application to observe and interact with all the aspects regarding the metaverse chosen.

Along with the implementation of their main grid, the Linden Labs company developed the Second Life^{TM} Viewer as mentioned in section 3.2.1, which is used by the majority of the SL residents.

Throughout the history of Second Life's^{TM} development, various versions of the viewer were released, to combat ever rising issues and bugs. At a certain point, Linden Labs started two deploy to extra viewers, along with the basic one, called the Release Candidate and the WindLight viewers. The Release Candidate Viewer (also known as the RC Viewer) is in fact a beta-tester-viewer, that contains the latest bug fixes and/or soon-to-be-released new features, which have not yet been included in the basic viewer. The WindLight viewer, on the other hand, included enhanced graphics options that pushed Second Life's^{TM} appearance to realistic levels. But since February of 2008, the WindLight version was terminated, and the advanced graphics options were included in the Release Candidate Viewer.

Parallel to Linden Labs' viewers, a number of other viewers, based on the official one, have appeared over the internet. This is because, since January 8th of 2007, Linden Labs released the source code of their viewer, allowing the development of the previously mentioned viewers.

Because the viewer plays an important role in terms of using Second Life^{TM} and Opensim as simulation environments for ubiquitous computing applications, specifically contemplating the metaverse's usability, a brief explanation of the most relevant viewers will be presented next.

3.4.1 Second Life^{TM} Basic and Release Candidate Viewers

As shown in figure 3.7 the interface of the basic Second Life^{TM} viewer is fairly simple. The majority f the screen is occupied in the representation of the sim where the user is currently at. On the button of the interface, a series of buttons allow the user to interact with the metaverse that includes chatting, building, check the user's inventory or simply take a snapshot of the current view.

Figure 3.7: Second Life^{TM} Viewer

As the reader can observe from figure 3.8, the Release Candidate Viewer is almost identical to the basic version, deferring only on the enhanced graphics and in-world new features, e.g., allowing a prim to use a url as texture, which allows visualization of the specified webpage in the prim, within the metaverse.

Figure 3.8: Second Life^{TM} Release Candidate Viewer

Both viewers are simple to interact with, and allow easy manipulation of the metaverse, essential to the objectives of this thesis. In terms of system requirements, both viewers demand middle equipped computers, where the graphics broad plays an important role along with the available RAM memory. The RC version is even heavier, and calls for a well equipped personal computer. To serve as a comparative model, the computer used to run the Release Candidate in a smooth manner, is a 2.2GHz dual core laptop with 2GB RAM equipped with a graphics broad with 515 MB of RAM.

Finally, these viewers were primarily designed to connect to Second Life's^{TM} main

grid, as connecting to other grids requires adding an extra argument when executing the viewer. This argument is:

```
-loginuri http://secondlife5.inescporto.pt:9000
```

as the viewer connects to the grid available running on the server specified by the url and port.

3.4.2 RealXtend

The RealXtend viewer was the first major implementation of an unofficial viewer specified to work with several grids. As the reader can observe in figure 3.9, this viewer contemplates an extra option on the entry screen, that defines the url of the grid. Although a simple feature, this option does not exist in the Lind Labs' viewers (only using the extra argument as previously referred).

Figure 3.9: RealXtend Viewer

Regarding in-world experience, this viewer is in all similar to the official versions, but extra features are expected in the future.

Other non-official versions of viewers are available, like the AjaxLife which is an ajax-based application implemented to allow all Second Life options, except metaverse 3D representation. Using a simple website to run, this viewer was specifically built to allow communication with other users in computers that do not support the official viewers requirements. The other viewers don't have any new options, which make them relevant to mention.

3.5 Conclusion

In conclusion, LSL is a programming language which offers great development possibilities and with a C-like syntax, it's quick to understand. Comparatively speaking, although

Second Life^{TM} offers more LSL stability, due to its maturity, Opensim allows more freedom, by its open source nature, in terms of complex code development to support the existing API functionality regarding the development of ubiquitous computing applications.

Also, the easy building properties of Second Life^{TM} and Opensim enable fast content development, that achieve a realism, that few other simulators offer.

Chapter 4

Smart Room Middleware

4.1 Introduction

Following the ubicomp direction, pervasive environments, with embedded hardware - sensors, smart displays and physical user interfaces - are the subject of numerous discussions and research.

But physically implementation of such systems has proven to be of high cost, financially and in terms of time/human resources. Thus effective prototyping and simulation can potentially shorten significantly this high resource demand. By simulating software models, possible issues can then be predicted and resolved, avoiding implementation problems in the real life model.

With this purpose, the first main simulated scenario of this project, is a control application based on a pervasive environment context. The smart room model is based on the iRoom [40], a research project with the objective of prototyping an interactive workspace model enabling easy control and data flow within the room.

In terms of hardware infrastructure, the iRoom contains three touch sensitive whiteboard displays along the side wall and a custom-built 9 megapixel, 72 inch (1,80 meters) diagonal display called the interactive mural. In addition, there is a table with a 36 x 48 inch (0,9 x 1,22 meters) display that was designed to look like a standard conference-room table. The room also has cameras, microphones, wireless LAN support, and several wireless buttons and other interaction devices [41].

The most significant contribution of this project was possibly the Event Heap, a middleware model based on the tuple space paradigm [42]. Message tuples are stored in the tuple space at the request of an application and queried out from that space at the request of another application. This allows natural and fluid interaction between equipments, and the room's backbone control and management applications.

While we simulated the large displays and a simple touch screen interaction between user and the displays, we focused mostly on the simulation of the enabling middleware.

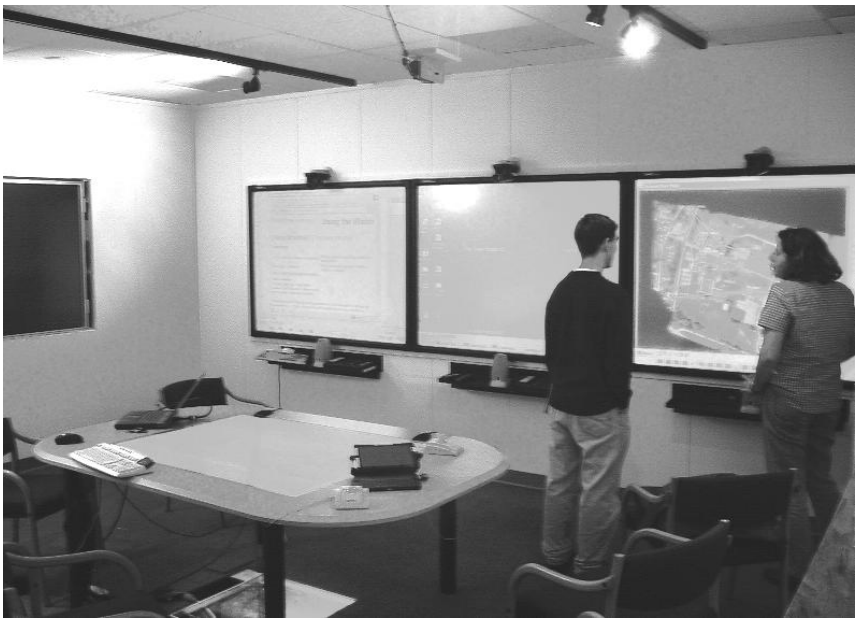

Figure 4.1: The iRoom

Rather than building a tuple space, we started by a simple publish-subscribe middleware on top of which a tuple space middleware could be built.

With this simulation, our goal was not only to evaluate Second Life's™ simulation capabilities and unmask any possible fragilities, i.e., if the scripting and building properties enable prototyping and simulation within this software context, but also measure its communication performance, comparing it with off-world and mixed reality scenarios.

The following sub-sections report the in-world simulation and its correspondent off-world application, along with a mixed reality experiment, using Second Life™ and Opensim, with a focus on the architecture and implementation. The results of the evaluation that we conducted in this case study is presented in chapter 6.

4.2 Architecture

After a pre-study on the architecture of the iRoom's software backbone and tested applications [40], it was defined that the simulated system would have a centralized server, in charge of all events and notifications. This communication between server and future peripherals would be based on the iRoom's architecture.

Due to the nature of LSL scripting, it was defined that the simulated system would break away from the iRoom on this point and instead implement a pure Publish/Subscribe architecture. This option will be explained later in this sub-section.

4.2.1 Publish/Subscribe

A centralized publish/subscribe system (figure 4.2) is in fact an asynchronous messaging model where senders (or publishers) are not aware of possible recipients (or subscribers) but still exchange various forms of data with each other. This is achieved by the characterization of the data into classes (or events) and enables the decoupling of publishers and subscribers which subsequently allows for greater scalability and a more dynamic network topology within the system.

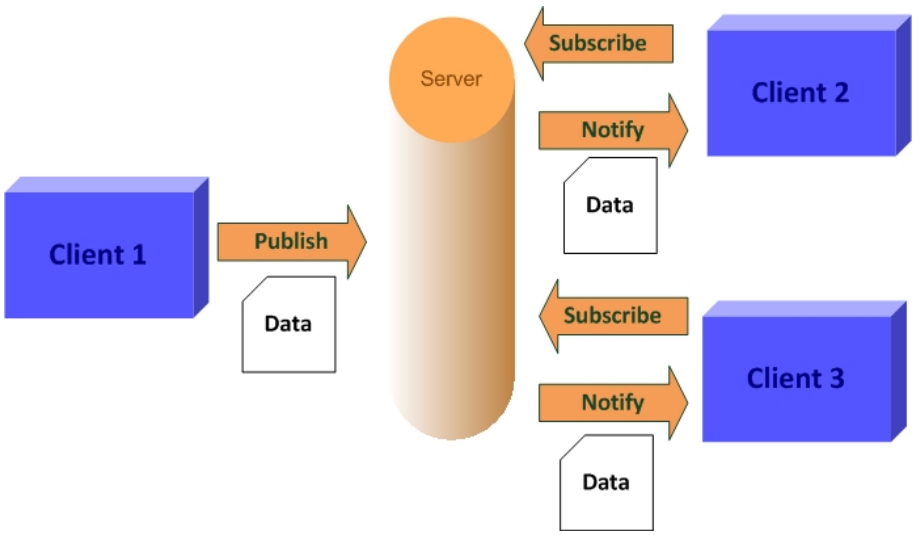

Figure 4.2: Publish/Subscribe Model

With the centralized Publish/Subscribe server (or Pub/Sub), various peripherals can communicate with each other, without knowing the existence of one and other.

The Pub/Sub model contemplates four basic operations that form the core (or primitives) of communication within the system, and are as follows:

- **Subscribe**: subscribes the client to a particular event, linking that event to the client, which will generate a notification when the event is submitted.
- **Unsubscribe**: unlink the client from a particular event, thus not receiving notice on that same event.
- **Publish**: when a client wishes to send data or any other action, it must publish a event to the server.
- **Notify**: upon reception of a particular event, the server checks the subscription list and if any link is found, it will generate a notify message to the subscribed clients.

The first three operations are transmitted from a client, as the notification occurs on the server's side, when it receives a publish request from a client, and there are subscribers registered, like figure 4.2 indicates.

4.2.2 Server

The server is the central piece of this model, where all communications and associated data will pass through. The simulated server, regardless of its location (off or in-world) will contemplate the modules specified in figure 4.3.

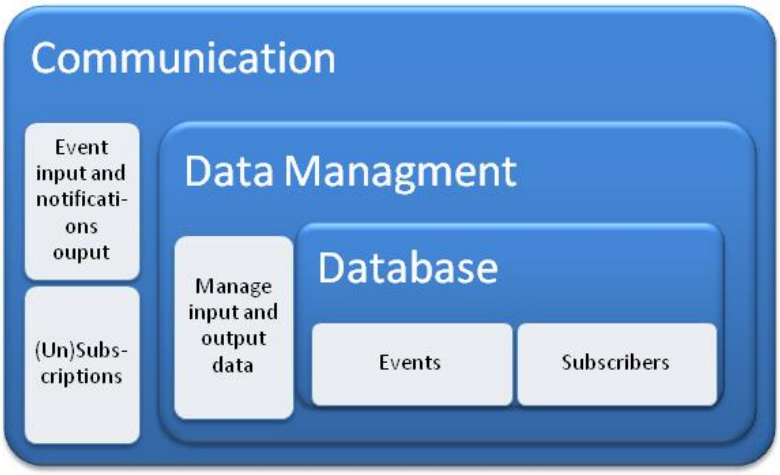

Figure 4.3: Server Modules

The communication module is responsible for receiving and sending messages, directly reporting to a data manager which will format the message, and access a database where all relevant operations are stored, e.g., the subscription list.

4.2.3 Client

Regarding the publish/subscribe paradigm, clients have a very light pub/sub layer, unlike the server.

Figure 4.4: Client Modules

As shown in the picture above, the client's layer regarding the publish/subscribe model is solely formed by a communication module that sends data with the model's message format and generates events whenever a notification is received.

4.2.4 Event Format and Messaging Protocol

The basic event format used in this architecture, is simple, roughly the event can simply be composed of its name.

```
["event name"];
```

Due to system's flexibility, data fields can be appended to the event, with virtually no limit of number:

```
["event name", "data", "other arguments" , (...) ];
```

This flexibility is enabled by the fact that there is no coded limit to the number of arguments, and only limited to the memory available for the simulation. This feature, combined with the asynchronous nature of the message delivering system where sender and recipient do not need to be directly aware of each other, allows the system to be used for any kind of data exchange.

The sender unique ID is appended to the event at a lower level of the software pile model, invisible to the Pub/Sub layer.

Regarding the subscribing and unsubscribing messages, these were created using a special kind of events called simply the "Subscribe" and "Unsubscribe" events. This option allowed maximizing the event format and message protocol already defined. These special events have the following format:

```
[(Un)Subscribe , event name];
```

These trigger a special treatment by the server and do not affect other clients.

With this event format, a simple messaging protocol, described in figure 4.5 was defined to allow more effective communications between clients and server. This protocol stipulates that any client, before sending any one of the previous defined events, must send a special low-level handshake event, again self-describing on the broadcast channel. At this point the client, by default, listens to all messages travelling on the publish/subscribe channel. After sending the handshake event, the serves responds with its unique identifier, which on the other hand causes an immediate change on the client. From this point on, the client only listens to the server identified by the special event, ignoring all other messages on the publish/subscribe channel.

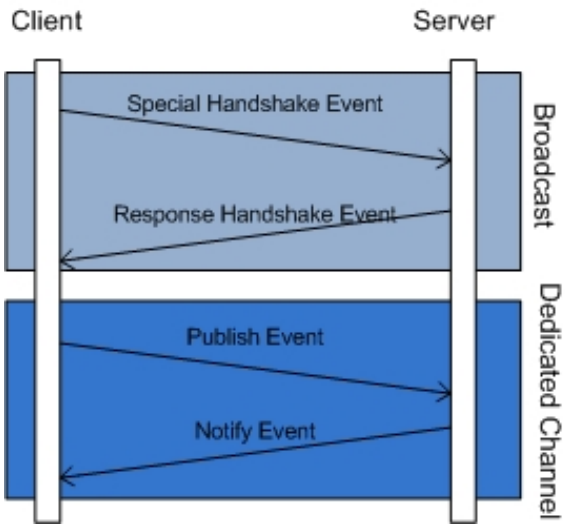

Figure 4.5: Publish/Subscribe Message Protocol

4.2.5 Subscribers List Vs Event List

As mentioned previously, an important decision regarding the type of data stored in the server was made that really defines the system's architecture. Switching from a event heap to a publish/subscribe model is mainly explained by the memory limit imposed on every LSL script in Second Life's main grid - 16KB when tested and referred by SL's LSL wiki[43]. An event list, instead of a simple subscribers list, would quickly burst the scripts limits, as a pre-test proved. In fact a simply outburst of events generated on one client, quickly crashed the server.

With this architecture as basis, the following simulation scenarios were defined:

1. In-world clients and server;
2. In-world clients and off-world server;
3. Off-world clients and server;

Along these 3 main simulations, other use-cases were tested, with mixed reality characteristics. The respective implementations will be presented with detail in the next sections.

4.3 In-World

The following sub-sections report on in-world simulations only, i.e., tested scenarios that use only Second Life's main grid or private grid based on Opensim. A a supplementary

scenario, where the server is implemented off-world and the clients in-world, has also been developed.

4.3.1 Second Life Implementation - In-world Server

The in-world implementation using Second Life's main public grid, with the architecture defined in the previous sub-section, would have to be both simple and effective, respecting the goals proposed.

With the publish/subscribe model as the foundation of the simulated environment, all applications should be built using the pub/sub primitives. This lead to a layered software structure, conceptualized with the TCP/IP and OSI protocols models in mind.

Regarding the simulation parameters and architecture, the layer model would be as shown in figure 4.6, with followed detail explanation.

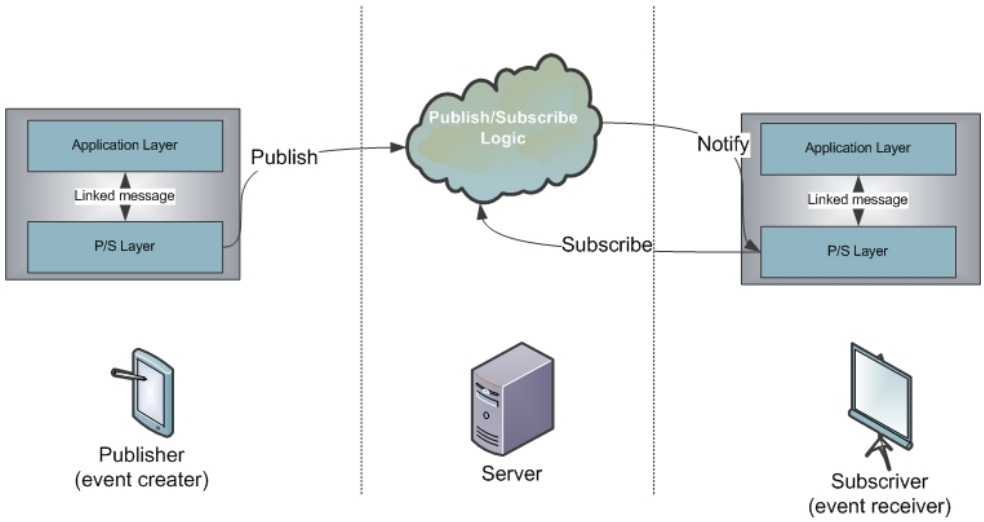

Figure 4.6: In-World Publish/Subscribe Implementation

4.3.1.1 Publish/Subscribe Layer

As mentioned previously the publish/subscribe layer is responsible for the asynchronous communication model between peripherals. The various aspects that constitute this layer are explained in detail next.

Communication Regarding LSL communications module, previously referred in chapter 3, the client-server communication on the pub/sub layer was implemented with the llSay() function on a private channel, defined in the layer, and all pub/sub inter-object communication is on this private chat channel. An important notion to retain is that sender information does not need to be implemented as a pub/sub event argument, because the listen LSL event generated when a object receives a message that complies

with the message filter defined by a previously invoked llListen() function gives that same information with great detail.

But using llSay() as the communication's backbone could pose a problem, as not only the server is listening that channel, but also all clients in the premises. This issue was foreseen and a messaging protocol was defined in the system's architecture.

In terms of the simulation, this means that all objects upon entering the smart room or simply rezzed, they will send a special event, called the Server ID. This event is constituted by one argument, a string self-naming the event. When the server receives this special pub/sub event, it processes it and responds with another ServerID event, but this time with one other argument, its UUID. The client(s) then receive this response and modify their llListen filter, to only listen to the object specified by the UUID in the ServerID pub/sub event. Note that whenever a client sends the special request, all the clients update their listen filters.

Event Data Concerning the pub/sub event format, in LSL is a simple list type where the first element is mandatory, known as the event name. The following elements are known as the data arguments, and have no software imposed limits, in terms of number and have a non mandatory nature.

```
[ event-name, arguments, (...) ];
```

When a client or server needs to transmit a particular event, the function used - llSay - doesn't support lists, which means that the event must be converted to a CSV (comma-separated values) string, and back again when received. These operations are supported in LSL by the llList2CSV and llCSV2List functions.

Server As specified in the architecture sub-section, the server is the centre piece of the system and with no application layer, its logic is purely based on the publish/subscribe model.

The server has two states. The first is called the "off-line"state, and like the name indicates the server does not receive or send any type of data when on this state. But with a simple touch by the user's avatar, the server changes to the "online"state. When on this online state, the server is always listening for incoming events. When it receives a particular event, it is processed, and checks if corresponds to a special event. If not it will check the Subscriptions list for any subscribers to that particular pub/sub event, and if any found, it will send the event to the specified targets.

If, on the other hand, the event received, constitutes a special kind of event, it be target of special processing. Being a subscribe or unsubscribe event, the server will check the Subscriptions list to verify the events validity and perform the desire action. Lastly, if the event is a ServerID one, like previously mentioned, the server will respond with its UUID.

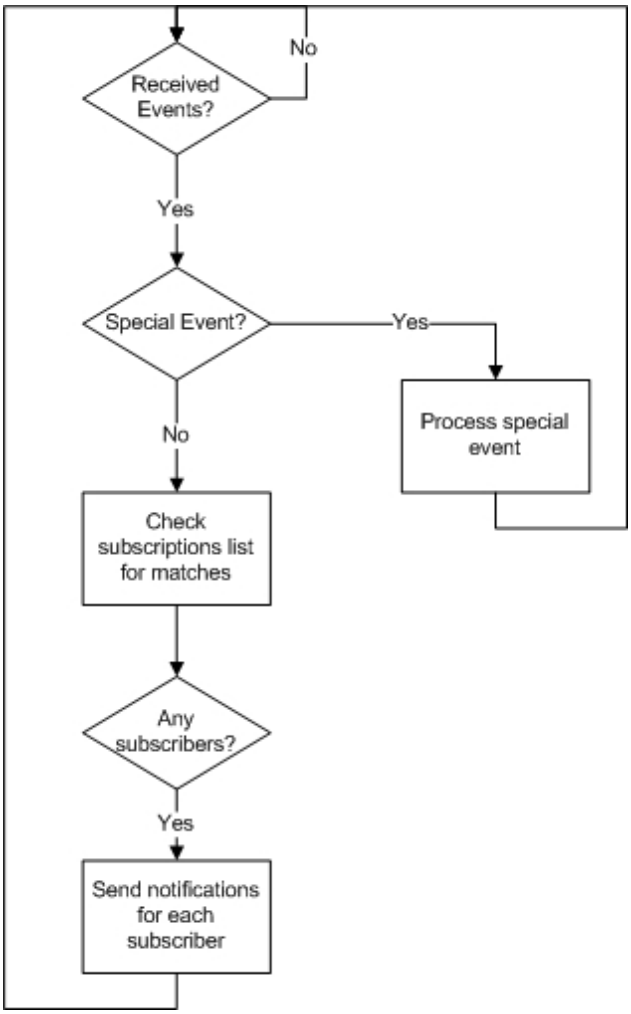

Figure 4.7: In-World Server Fluxogram

Clients Unlike the server, the Publish/Subscribe layer on the clients does not need any kind of data storage. The layer was developed with the pub/sub primitives in mind, and allows the following functions:

- **Subscribe**: sends a special "Subscribe"event, with the specified event name;
- **Unsubscribe**: functions the same as subscribe primitive, but for unsubscribing a event;
- **Publish**: send a particular event to the server with optional data;

On the other hand, the P/S layer will generate a "LSL event", when a notification is received from the server, that's not considered a special notification.

If a special notification is detected, in this case a ServerID notify, the client will format its llListen filter parameters, to only listen to the object referenced by the UUID passed as argument of the pub/sub event.

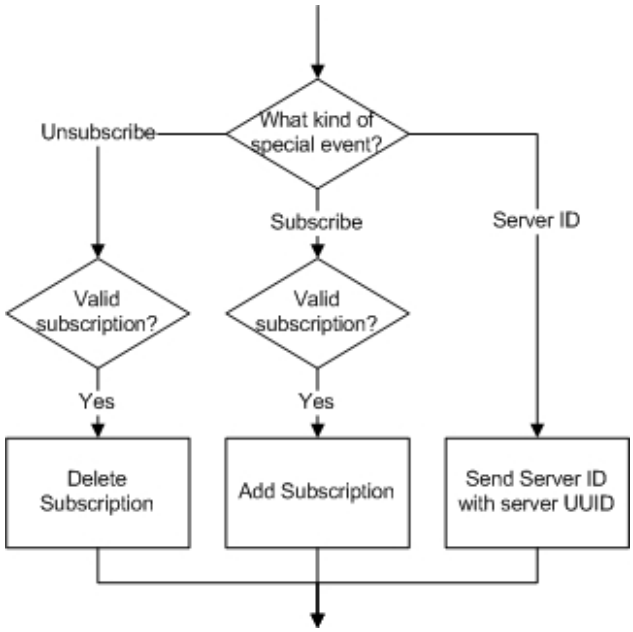

Figure 4.8: In-World Server Fluxogram - Special Events

4.3.1.2 Inter-Layer Communication

As shown in the pic 4.2, the publish/subscribe logic was implemented in a layered model, to allow maximum scalability to the system. With this methodology, any given application can be developed above the pub/sub layer, using its primitives to communicate to various objects. But, this model obliges the definition of an inter-layer communication protocol.

After studding all communications functions within LSL, the obvious choice pointed to the llMessageLinked. Although the main purpose of this function is to allow communications within a linkset of prims, it also enables message exchange between scripts of the same prim. As the model defined contemplated two layers, which one corresponding to a separate script, this function fulfills the requirements.

As explained in the previous chapter, the llMessageLinked function has the following syntax:

```
llMessageLinked(integer linknum, integer num, string str, key id);
```

The linknum argument, in this context, uses the LINK_THIS constant, which means the linked messages will only be transmitted to the "prim script is in"[44].

The number of different linked messages is directly related to the number of primitives of the pub/sub layer, so the model defines four different types of messages. The num argument was chosen to define each type of linked messages, and the *str* and *id* as the data exchanged between layers with the syntax defined in the 4.1 table.

So an example of this logic can be seen here:

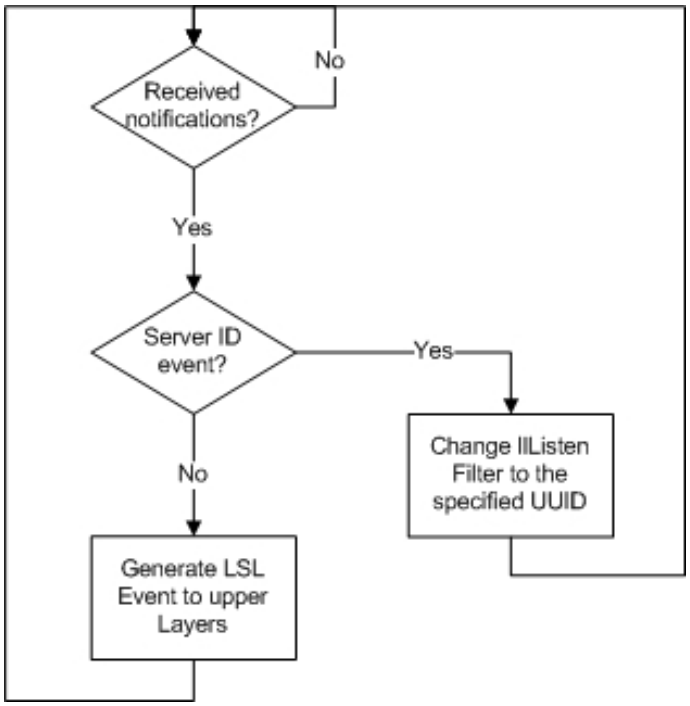

Figure 4.9: In-World Client Fluxogram

```
touch_start(integer total_number)
{
  llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
}
```

This object, when touched by an avatar, will send a linked message to the publish/subscribe layer with the identifier "3", which corresponds to a publish action. Then the pub/sub layer will redirect the event to the server, following the data flow described earlier.

If linked messages type "1", "2"and "3"are directed downwards, the type "4"message as a ascending nature. This message is created in the pub/sub layer and used to transmit any notifications received from the server to the application layer.

4.3.1.3 Application Layer

The simulated application to verify the pub/sub's functionality was a simple power management to control the state of a given number of displays.

The original layout of the iRoom was designed using Second Life's™ building capabilities as shown in figure 4.10.

Table 4.1: Inter-Layer Messaging Protocol

Type	Arguments	Id Num
Subscribe	"Subscribe" + Target	1
Unsubscribe	"Unsubscribe" + Target	2
Event	EventType + Target + Sender	3
Notify	Target + EventType + Sender	4

Using the pub/sub logic, all five displays subscribed a event called the "Button", and the button object, when touched, published a event with the same name. The correspondent notification to the displays would cause a change in their state (on/off), visible in the simulated smart room, as figure 4.11 shows.

4.3.2 Second Life Implementation - Off-world Server

This in-world implementation is a variation from the previous one presented. They differ in the fact that the pub/sub server is located outside the Second Life's™ metaverse on this simulation, and although the server is located off-world, we present this work in this section because all clients are in-world.

To implement this architecture the communications module of the publish/subscribe layer was modified to use the appropriate functions.

These changes, along with the off-world server development and in-world client adaptation are presented next.

4.3.2.1 Publish/Subscribe Layer

Communication Like mentioned above, in this implementation, this module suffered the most drastic changes, when comparing to the server in-world simulation. Because the server is located off-world, specifically a PHP server associated with a MySQL database, the llSay based communication of the server in-world simulation was not adequate. Instead two new functions were used, according to the origin of the data communication - off-world or in-world.

Regarding data leaving the Second Life metaverse, the function used, and the only one available to this purpose, was the llHTTPRequest, which is explained in sub-section 3.2.3. Using this function to communicate with the outside server, the pub/sub layer transforms the event syntax to arguments passed in the following form:

```
SERVER_URL+"op="publish+"&key="+(string)gChannel+"&type="event2send+"&subs="
+llGetObjectName()
```

Being the SERVER_URL variable the server's URL, the llHTTPRequest sends a GET request to the specified web address that invokes the PHP server, which fetches the passed arguments with the $_GET[] array.

Figure 4.10: Implemented Smart Room

Data entering the main grid of Second Life is a more complex situation. To enable such a scenario, an XML_RPC channel has to be created for each client, using the llOpenRemoteDataChannel function, again explained in more detail in chapter 3.

When objects enter the smart room logic domain, they subscribe to whatever events they require. On this event, the objects invoke the previous mentioned function to open a input data channel, which the server will use to send any notifications.

Server As referred earlier, the off-world server was developed in PHP, connected to a MySQL database, where the subscriptions list is stored, as the PHP file itself could not hold any data. Like its in-world homonymous, the PHP server receives the subscribe, unsubscribe and publish messages, and generates notifications.

Relative to the server functionality, the cycle is equal to the in-world one, only discarding the ServerID special event, because the pub/sub layer was hard coded the url location of the off-world server.

Clients, Inter-Layer Communication and the Application Layer Apart from the changes in the communication module, the P/S layer of the clients is exactly the same as the first simulation. This is also valid to the inter-layer communication protocol and the application layer, as no modifications were necessary.

4.3.3 Opensim Implementation

After a analysis of Opensim's server code, and it's alpha state, specifically regarding LSL implementation as explained in sub-section 3.3.3, a rather different approach to the

Figure 4.11: Implemented Smart Room - Displays On

publish/subscribe model had to be idealized.

Since the pub/sub LSL code could not be used in Opensim, even with major alterations, the solution found was to embed the publish-subscribe layer into the Opensim's own code, respecting architecture previously established.

4.3.3.1 Publish/Subscribe

Because Opensim was developed in C#, instead of the LSL lists, a more dynamic model would use the language capabilities. So a Subscription Class was modulated with the information that defines a subscription, and a Subscriptions one, that consists in a list of subscriptions and the respective methods. This implementation is very similar to the one in LSL, but without the issues associated with that scripting language.

Both classes were inserted in the Opensim's code, specifically in the class LSL_BuiltIn _Commands, that defined all the supported LSL functions off the script engine used by the server. Because this class is used by every script engine - one engine per object in the sim - a special class Pub/Sub was created to centralize all subscriptions.

With this implementation, the server object developed in the first case seized to exist and is fully embed in the Opensim's code.

4.3.3.2 LSL Integration

After the development of the Publish/Subscribe logic, a code interface was necessary to allow access to the pub/sub list by any resident.

This interface was achieved using the LSL built in structure - note that the deployment of the pub/sub classes in the LSL functions implementation file was no coincidence. In

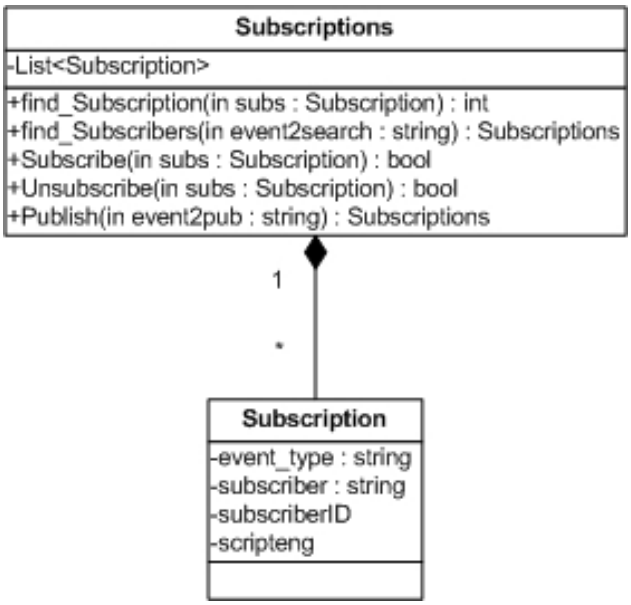

Figure 4.12: Subscriptions Class Diagram

practice this meant adding new LSL functions and events to the script engine, which proves not only possible but very effective and relatively easy for users.

This liberty of code could not happen in SL's public grid, due to the closed nature of Linden Labs server's code.

Using the model defined earlier in this report, there were created three new LSL functions and one LSL event. The three functions, much like in the Publish/Subscribe layer defined in the previous mentioned implementation, report to the Subscribe, Unsubscribe and Publish primitives, and have the following syntax:

- **llPSSubscribe(string event2target)**: like the name suggests, this functions subscribes the prim, in which the script is running, to the pub/sub event specified in the string argument;
- **llPSUnsubscribe(string event2target)**: if the prim is subscribed to the specified event, it deletes the subscription;
- **llPSPublish(string llevent, string argument)**: this functions publishes the specified event, along with a argument string used to pass data between clients. The internal server will check for any subscribers to that particular pub/sub event and inform them of the subsequent notification.

This notification process consists on a actual LSL event, much like the link_message() or listen() presented in the previous chapter, called the "publish" event and has the following syntax:

```
publish(string publisher, string event_type, string data )
{
  code;
}
```

The automatic generation of this event, was not as complex as we feared, due to the nature of the Opensim's code, which provides excellent support to incorporate changes and new assets.

4.3.3.3 Application

With the publish/subscribe layer fully developed and embedded in the Opensim's code, the final milestone was to simulate the previous implemented application, to validate the pub/sub logic implementation.

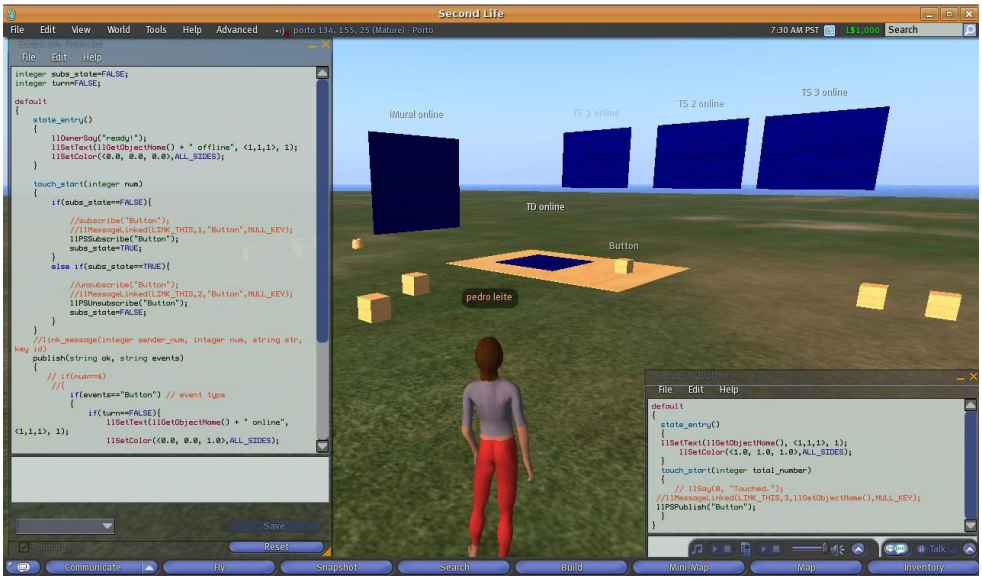

Figure 4.13: Opensim Smart Room Simulation

The LSL application code was adapted from the one developed in the first implementation. The migration only implied changing the llLinkedMessages, used in the public grid, to the new LSL pub/sub primitives and the replace the link_message LSL event to the newly created publish LSL event.

The simulation validated the new pub/sub implementation and further results will be presented in the next section.

4.3.4 Performance

With the publish/subscribe model fully implemented, its communication performance was measured using the application layer.

This evaluation consisted on measuring the communication delay between the publish action on a particular client, and the respective notification on a subscriber of that event, on all the three previous presented simulations.

For each scenario, the delay was measured with the subscribers list with only the client used, and with two elements, being the client tested on the last position of the list.

As shown in the next figure, the total delay time measured each time includes communication times between:

1. the Application and P/S layers of the event sender via Link Message;
2. the P/S layers of the sender and server;
3. the server's own processing time;
4. the P/S layers of the server and target;
5. the P/S and Application layers of the target.

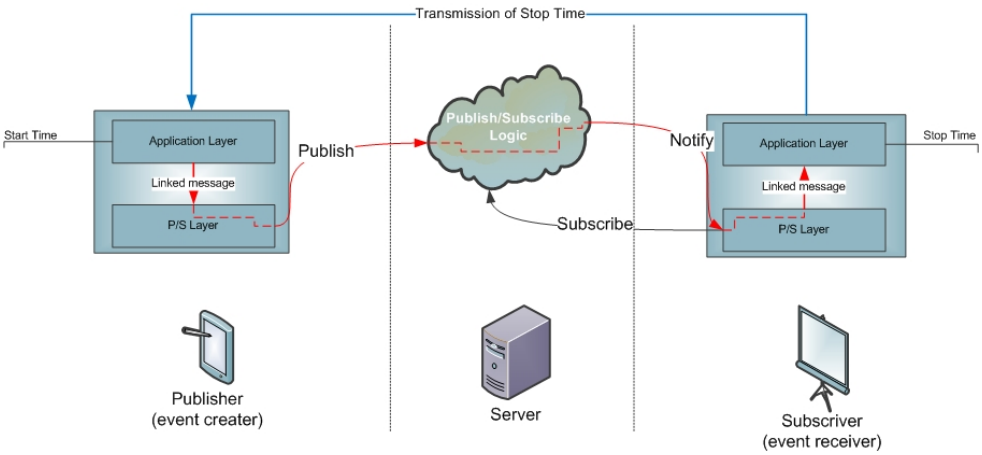

Figure 4.14: In-World Cycle Test

The publisher is responsible for calculating the time difference between the departure and the arrival time on the target client. This client communicates the stop time to the publisher, which calculates the delay time. In all these three tests, the subscriber relays the stop time information to the publisher using the llSay function.

To push the middleware to stress levels, the delay times were measured using a cycle. When the publisher receives the stop time from the subscriber, it publishes the event again, for a N number of times. After a series of tries and discussion, the testing parameters were defined and concluded that each test cycle would generate one hundred events. After these one hundred events travelled through the logical path defined previously, the average delay time was calculated for the cycle. This cycle was repeated fifty times and calculated a final average time of the whole process.

The results concerning the three simulation scenarios previously mentioned will be presented in the next subsections.

4.3.4.1 SL - Server In-World

This testing module was inserted in the application layer of two clients: a subscriber and a publisher. The test implied publishing events that changed the state of one display in the smart room. Relative to delay issues, all times were achieved using the llGetGMTclock function. Running with the parameters specified earlier, the results are shown in table 4.2.

Table 4.2: Server In-World Performance

Description	Average Delay (s)	Standard Deviation
One subscriber on queue	0,140	0,021
Two on queue - Last place	0,251	0,015

As the reader can observe in this simulation there is no much delay between messages, but the lag doubles as the server was loaded with more subscriptions. This indicates that in much larger scale, searching algorithms would have to be embedded in the server code, when checking the subscriptions list.

About half way of the testing procedure, a visual delay between the current task in progress and the state of the display was subtle but undeniable. This may suggest an overflow of information on the sim server, overflowing its processing capabilities.

An important information to retain is that the sim used - Universidade do Porto island - is generally a rather stable simulator, with very few lagging issues, which means that this simulation, and the following ones, were realized within excellent conditions, and the lag observed is almost entirely due to the simulations processing requirements.

4.3.4.2 SL - Server Off-World

Regarding the simulation model, where the publish/subscribe server is located outside the Second Life^{TM} metaverse, the test results are shown in table 4.3.

Table 4.3: Server Off-World Performance

Description	Average Delay (s)	Standard Deviation
One subscriber on queue; Sleep(15) every 15 times	1,320	0,021
Two on queue - Last place; Sleep(15) every 15 times	1,330	0.028

As expected the off-world location of the server, translated into a much slower communication between objects, which makes it a valid only when the server has to support great quantities of data.

An important revelation of this simulation was the limit imposed by the Linden Labs servers on the number HTTP requests. In fact

"requests are throttled to a maximum of 25 requests per 20 seconds"

[44]. In practice this limit appears to be even lower, and in order to allow the simulation to run without crashing, a sleep of fifteen seconds was imposed every fifteen cycles. Note that a outside overflow of information may cause the cycle to crash, which was observed when the number of visitants of the island peaked at a particular moment.

Another important conclusion is the fact that the MySQL server responded well in both scenarios of list occupancy, which leads us to conclude that data searching is much more effective using an outside database.

4.3.4.3 Opensim

Much of the test cycle code used in the two previous tests was used in this simulation, as the application layer is almost the same, and translated into the results shown in table 4.4. The only exception was a small modification in the Opensim code, regarding the llGetTimestamp used to retrieve arrival departure times. This issue was due to the incompatibility of the date and time formats, along with the wrong use of the correspondent time zone. This adaptation re-enforces Opensim strength in resolving implementation issues.

Table 4.4: Opensim Performance

Description	Average Delay (s)	Standard Deviation
One subscriber on queue	0,001	0,0004
Two on queue - Last place	0,003	0,0002

The fact that the Opensim server is local and no external factors intervene, allowed such small delays as shown in table 4.4. Much quicker than Second Life™, the Opensim implementation also allows future development of more effective searching algorithms, which would lower the 0.003 seconds of delay in the second experiment, without using outside databases.

Of course linking the Opensim pub/sub code to a SQL database would be much easier and effective, when compared to Second Life™, because this bridge could be hard coded, bypassing the HTTP restrains of Second Life™.

4.4 Off-World

This section regards all off-world implementation, i.e., a real-life scenario to use as case test for comparing directly to in-world implementations and performance.

4.4.1 Implementation

Unlike in Second Life, off-world there's no predefined communication module ready to use, so a slightly different approach was imperative.

With the mixed reality simulation already in the horizon, the most functional choice was to use a web server as the publish/subscribe server. By using this approach not only the main logic of the publish/subscribe server could remain the same, but also enable server communication from any physical and logical location.

Regarding the development platform, the logical choice was the .NET and Mono frameworks, using the C# language, because Opensim's code and the publish/subscribe model incorporated were already developed in this programming language.

An important goal of this implementation is stability and being able to run on multiple operating systems, which since the development of the Mono framework, has been a doable task.

With the architecture model defined previously, the off-world system consists on a publish/subscribe centralized web server, communicating with clients on multiple workstations, regardless of the operating system.

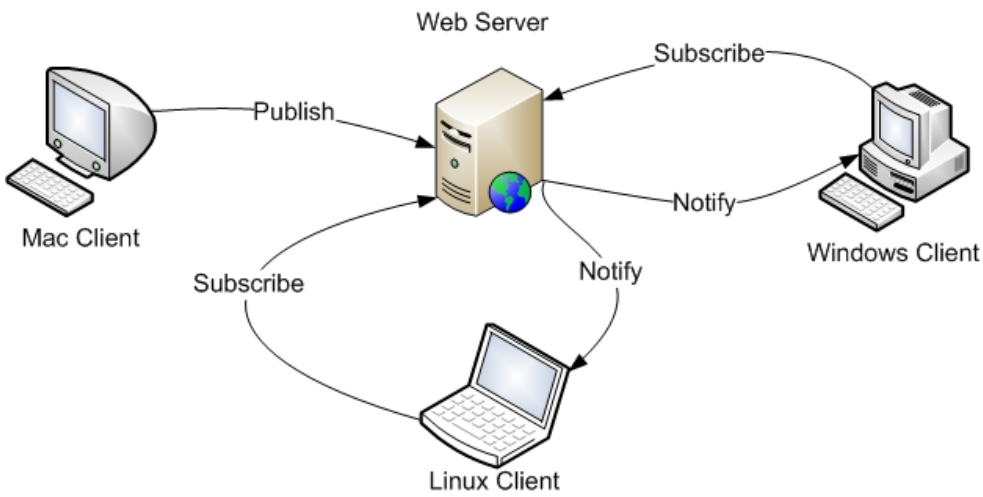

Figure 4.15: Off-World Publish/Subscribe System

A more detail explanation on the server and clients' implementation, along with the communication interface between them, will be presented next.

4.4.1.1 Pub/Sub Layer

Server To incorporate the publish/subscribe logic, migrating the Subscription and Subscriptions classes from the Opensim implementation, the first step was to developed a fully functional web server.

This web server was developed using two main classes, the HttpServer and HttpProcessor. The HttpServer constitutes the main class, and receives HTTP requests on the corresponding 80 port. For each request, the program launches a new thread, invoking the HttpProcessor class, which will handle the incoming request. By using threads, virtually

any incoming connection can be accepted, which constitutes a more stable nature to the server.

The web server has the expected behaviour of a HTTP/1.1 regular server, with the exception of some special cases. The special requests constitute the publish/subscribe primitives and handled by the previously mentioned HttpProcessor class. Much like the LSL server developed in Second Life's public grid, and the embedded server logic in Opensim, the web server receives the special requests, Subscribe and Unsubscribe, acknowledging the respective clients, and the Publish primitive, notifying any registers subscribers.

Client On the clients' side, a pub/sub layer was developed using the same logic as the previous implementations. The Pub/Sub class contemplates the three basic primitives, and enables a listening mechanism for any incoming notifications.

When a notification is received, a event is generated, to allow a more structured manipulation by the user developing applications on the upper layers, again like the first set of simulations.

Communication Regarding this off-world (or real-world) implementation, the model of communication between clients and server was the most complex to implement, as the logic behind the server and clients was already defined, which even allowed to use the pub/sub classes previously developed, embedded in the web server model.

Since the server is based on the HTTP/1.1 model, the client to server primitives had to be encapsulated as regular HTTP requests, and were achieved using a HttpClient class, that sent simple GET requests. These requests were defined as shown in table 4.5, and were based on the off-world server/in-world clients test scenario implemented previously

Table 4.5: HTTP Requests

Primitive	HTTP Request
Subscribe	Subscribe?event=EVENT_NAME& subscriber=SUBSCRIBER_DNS_NAME
Unsubscribe	Unsubscribe?event=EVENT_NAME& subscriber=SUBSCRIBER_DNS_NAME
Publish	Publish?event=EVENT$_NAME\&publisher = PUBLISHER_DNS_NAME\&arg0 = DATA_0\&()argX = DATA_X$

Regarding server to client notifications, there was no need to use HTTP requests, was this model would required a HTTP server in each client. A simpler and effective solution was to use simple TCP connections between them, based on the DNS name passed as argument on the subscription and un-subscription actions. When a client enters the pub/sub logic, the corresponding layer starts immediately listen on a given port for incoming notifications from the server.

4.4.1.2 Application Layer

The final step of this implementation was to develop the application to validate the pub/sub model. Using the same parameters as the in-world simulations, two clients subscribed to a particular event, publish by a third one. When the subscribers receive the notification, the monitors of those workstations were turn on or off, conforming to it's previous state.

Regarding the workstations used, four personal computers were used. The server ran on a Windows Vista laptop, and the clients used were two Macminis with Mac OS X Leopard (v10.5) and a destop runing on Linux Ubuntu 7.1 .

The application validated the communication model and the pub/sub logic, and was tested on two separate scenarios, using wired and wireless networks. Performance results and evaluation on the code developed will be presented in chapter 6.

4.4.2 Performance

After the validation of the publish/subscribe model off-world, the next step was to evaluate the system's performance in terms of communication delays, to compare with the values of the in-world simulations.

Using the exact same parameters of the in-world tests, on the application layer a delay test was developed. Regarding the stop time relay, the simplest solution was to use a different event, as shown in figure 4.16.

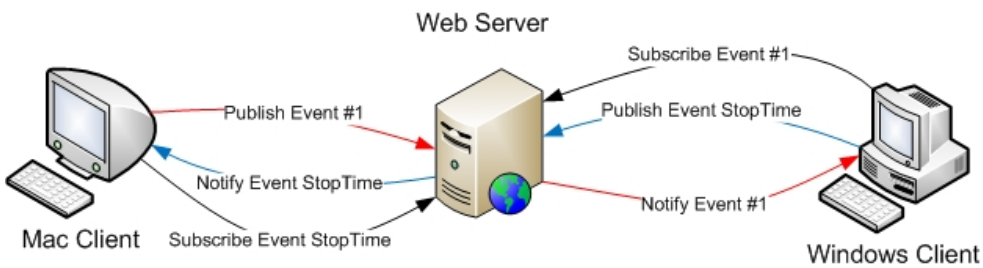

Figure 4.16: Off-World Pub/Sub Performance Cycle Test

The test event (red), when received by the test subscriber, publishes the stop time using a event called "Stop Time"(blue) , which will be received by the original publisher.

On the course of the test development an issue emerged, that did not exist in simulation using Second Life^{TM} and Open. This problem is the clock synchronism of the workstations involved in the system, since there were several machines used, with different operating systems, making impossible a clock synchronism by regular means.

After some research, the solution found was to synchronize the internal clocks of the personal computers used, using a NTP (Network Time Protocol) server. INESC network infrastructure offers such a server and due to the small number of hops (Stratum 2), this was the one used.

With the clocks synchronized by the NTP server, the following values, presented in table 4.6 were measured using the test cycle application.

Table 4.6: Off-World Performance

Description	Average Delay (s)	Standard Deviation	NTP Error(ms)	Average Delay Without Error(s)
All 3 Workstations on wired network	0,402	0,0007	394	0,007
All 3 Workstations on wireless network	1,112	0,0035	9,710	1,103

The NTP error column is the combined offset of the subscriber and publisher, relatively to the NTP server.

Without the NTP offset, the model using a wired network offers results similar to the ones given by the Opensim simulation, proving the simulators great performance. Also remarkable, is the great difference between wired and wireless networks, which indicates that future implementations and simulations should always use wired networks, if available.

4.5 Mixed Reality

This last section of the current chapter, exposes a mixed reality implementation of the publish-subscribe system, combining Opensim and real life events.

4.5.1 Implementation

With the in-world and off-world simulations implemented and their performance evaluated, a dual mode of the publish-subscribe system was developed to compare with the previous cases in terms of performance, and evaluate Opensim's mixed reality capabilities.

The core of this implementation is to join the off-world publish/subscribe server developed in C# and the embedded server in the Opensim simulation. This architecture allows events to travel from the Opensim metaverse to real life and vice-versa without any restrictions.

Regarding implementation off the servers, both were already fully developed, due to the previous simulations and implementations. Thus it was just necessary to create a bridge between the servers.

4.5.1.1 Inter-Server Communication

The first model for this implementation was to use a central server that received all events, off-world and in-world. This would require that the embedded pub/sub layer of Opensim would have to be reformulated to send events to this central server, which should be like the real world one, a web server.

But a quicker and easier solution was found. Using the synergy from the previous two implementations and bridging them together, the pretended functionality of the server would not be affected and offers a flexibility that the one central server model didn't.

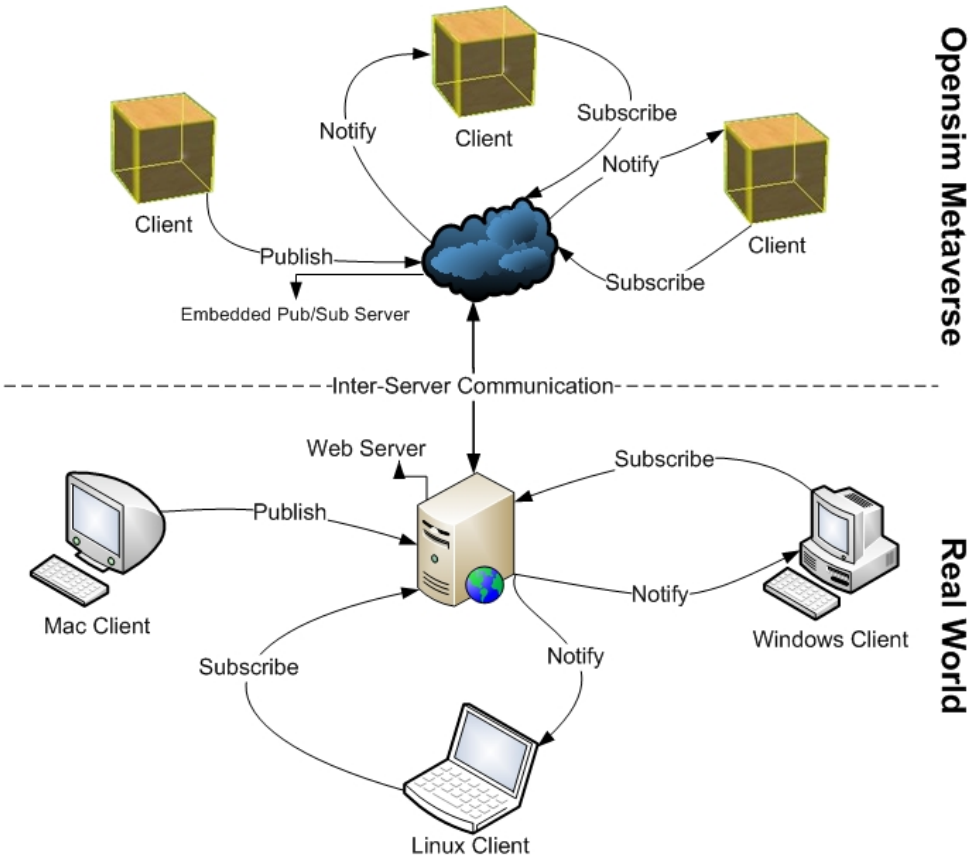

Figure 4.17: Mixed Reality System

This architecture allows both servers to communicate with each other but also run independently from one and other.

Since the real world server, receives HTTP requests, and responds using lower TCP communications, this model would constitute the bridge communication protocol, and both ways of data flow will be explained next.

In-World to Off-World Communication When the internal publish/subscribe server of Opensim receives publish events from any given object within the metaverse, it would check the subscriptions list embedded in the code and notify any registered subscribers. To allow the bridge between in to off world objects, the pub/sub server now sends a special notification to the off-world web server. Because the web server receives HTTP requests, this special notification is encapsulated into a HTTP Get request, equal to the publish request that off-world objects send to the web server.

The web server does not make distinction from the special publish and the regular publish actions outputted from real world objects, because is does not need to. This fact allows 100% use of the web server code previously developed, with absolutely no alteration whatsoever.

To implement this feature, the llPSPublish primitive developed in the Opensim simulation was modified to check the internal subscriptions listings and also send a HTTP request to the off world server. This request was sent using a HTTP client class, already developed on the client's side in the real life implementation, which meant inserting this class to the pub/sub class and adapting it to the new functionality, which proved to be quick and easy.

Off-World to In-World Communication Like the previous section referred, when the off-world server receives any publish action, it checks its subscribers list and uses a TCP connection to notify each (if any) subscribers.

To allow maximum use of the code already developed, the communication of the off-world server to the in-world one would respect this model.

So when the web server receives a publish action by a client, it will send a special notification to the workstation where the Opensim server is runing, using a TCP connection. To allow this, a TCP mini-server was embedded into the Opensim code, which translated into a TCP client class, on permanent lookout for incoming notifications. This class is used by the main pub/sub class.

When a notification is received by the mini TCP server, the pub/sub class will check the subscriptions list, and notify any subscribed object, as if an in-world object had published an event.

An important aspect to retain is that this model would easily create an infinite cycle because if anyone of the servers (off or in world) receive a notification it alerts the other one. This situation was fully expected and avoid, using a notification flag on the pub/sub in-world server.

4.5.2 Performance

The final step of the smart room simulations was to evaluate the communication performance of the system in dual mode. To this purpose two scenarios were tested, an off-world object communicating with a in-world one, and vice-versa. Like the off-world implementation test, the stop time relay is given to the original publisher using a pub/sub event, demonstrated in the 4.18 figure.

With the code developed on the previous tests, it was only a matter of adapting it to the mixed mode. During the in-world to off-world performance test, a conversion problem occurred when Opensim could not cast from string to float(double). In order to resolve this problem and avoid using int - this would introduced a greater margin of error - a new LSL primitive was created, using the previous methodology, called the llString2Float(string arg) which converts a string to a float number, respecting the comma. This simple but effective solution, once again underlines Opensim's versatility in resolving development issues.

With all both tests fully working, the output of results were as the 4.7 table presents.

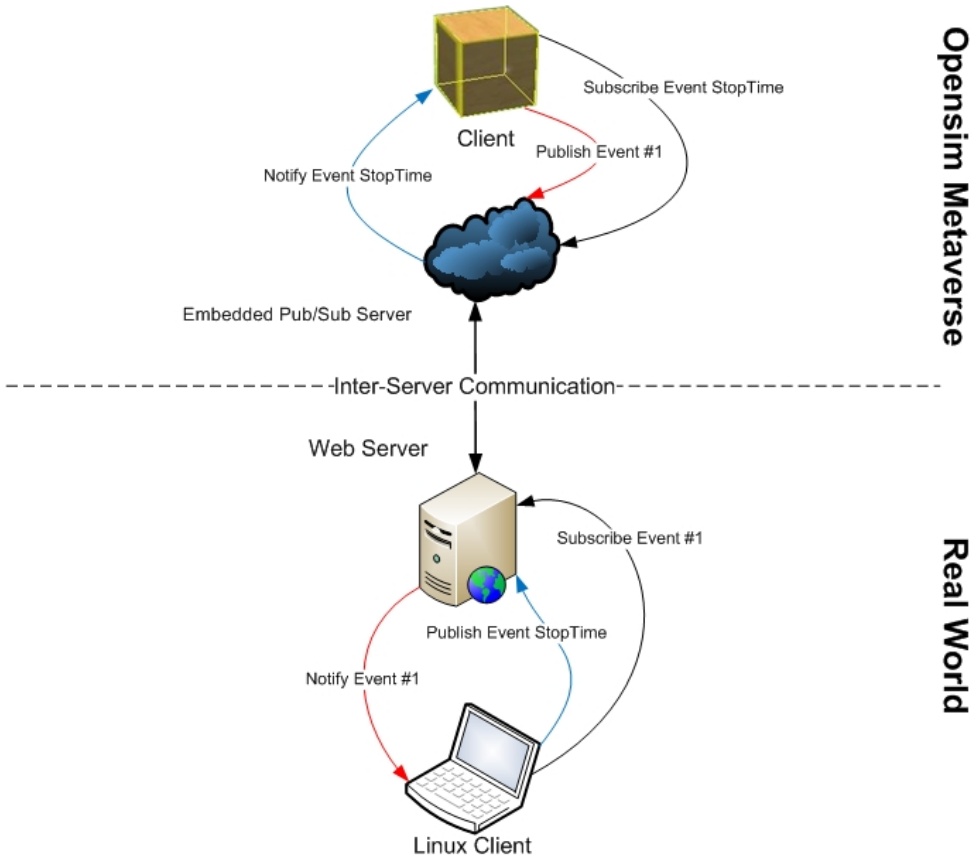

Figure 4.18: Mixed Reality Performance Test

Again, using the NTP offset to extinguish synchronism errors, the results in table 4.7 show that both test obtained a fair result, with very little delay in the communications on either way.

4.6 Overall Performance

When comparing the overall performance results of all the simulations, the Opensim scenarios offers by far the smallest communication delays, as table 4.8 exposes.

In fact, when comparing the simplest simulation scenario - on subscriber in queue - Opensim is 140 times more faster than Second Life using the in-world pub/sub server and 1320 times faster than the Second Life^{TM} clients and off-world server simulation, which indicates that Opensin offers a efficiency in processing the simulation parameters to which Second Life^{TM} cannot compete.

The Linden Labs servers have to overcome the difficulties of managing a growing network of sims with millions of users worldwide and billions of scripted prims. This of course justifies much of the surplus delay in SL's main grid, when compared to Opensim,

Table 4.7: Mixed System Performance

Description	Average Delay (s)	Standard Deviation	NTP Error(ms)	Average Delay Without Error(s)
In-world Publisher - Off-world Subscriber	0,439	0,017	393.45	0,046
Off-world Publisher - In-world Subscriber	0.190	0,005	143.331	0.047

but also the remoteness of those serves, opposing the local nature of the Opensim server used.

Table 4.8: Opensim Results Comparison

Simulation Scenario	Description	Average Delay compared with Opensim's
Second Life	One subscriber on queue - In-world Server One subscriber on queue - Off-world Server	140 $\times slower$ 1320 $\times slower$
Off World	All 3 Workstations on wired network All 3 Workstations on wireless network	7 $\times slower$ 1103 $\times slower$
Mixed Reality	In-world Publisher - Off-world Subscriber Off-world Publisher - In-world Subscriber	46 $\times slower$ 47 $\times slower$

These facts make the Opensim platform the primordial simulator, offering processing capabilities that Second Life^{TM} is unable to deliver, along with more security and stability, because the Opensim server can run on any personal computer, meaning the simulation depends only on that computer's processing capabilities, independent of any external factors.

Comparing the Opensim simulation to the real-life implementation, the open source server generated results very close to real world delays being even faster, proving the simulator's great processing capabilities.

Regarding the two simulations within Second Life, the one using the off-world server embedded in a PHP web page connected to a SQL database, is slower than the in-world server, built using a LSL script, which is 9.4 times faster. The off-world server model is only justifiable when the simulation requires a database of large proportions, passing the 16KB per script limit imposed in SL's main grid. The large difference in numbers is not only justified by the communications delay, in the HTTP requests and in XML-RPC channels, in traveling the Internet, but also because of the imposed limit in HTTP requests of LSL. The "maximum of 25 requests per 20 seconds"[44] limit the application's performance to a "unusable"state, compromising the stability of the model.

The dual mode system (mixed reality) proved also to reliable in terms of communication and data processing, offering delay times similar to the real life model (6.6 times slower), proving Opensim's stability and flexibility within mixed reality experiments.

Chapter 5

Social Proximity Application

5.1 Introduction

The development of wireless communications has lead to the booming of a market, currently evaluated in billions of dollars, reaching the four corners of the world. Of course, we are talking about the mobile device business.

As more and better hardware reaches a wider range of users, new software models are subject of intensive research, in hope to deliver new applications that take full advantage of all the new technologies available, and capture the users' attention with all new functionalities. This domain of research is a perfect example of next generation ubiquitous computing development, where new applications thrive to enable more natural interaction between man and machine, and even build new bridges in terms of social networking between users.

With the number of potential clients growing exponentially, software prototyping and simulation in complex interaction scenarios, using mobile devices, are becoming a struggle in terms of implementation - cost and time. So, prototyping and simulating these software models with the hardware specifications using a suitable platform can save researchers and companies' time and money that might not have.

Taking all these facts in account, the second main simulation scenario is a social proximity application (SPA), using Bluetooth technology as the communication backbone. Although the application developed was not based on one specific, like the first simulation was almost entirely based on the iRoom project, the model was based on a group of SPAs.

Scent [10] is an example of social mobile software, where on real-world physical proximity two or more user's contacts list are compare and if a predefine similarity parameter is satisfied, the respective profiles are sent to the other parties, enabling new social encounters. Serendipity [45] offers a more complex profile matching. The mobile device uses the GPRS network to connect to an Apache web server via a PHP interface with the BTID (Bluetooth unique interface identifier) as the identifier. The server accesses a database of individual profiles that links a device's BTID to its owner. Upon the profile fetching, the

matching occurs that will determine the possibility (or not) of the social interaction based on Instant Messages and others. DigiDress [9] on the other hand doesn't offer any profile matching, but enables social encounters based on physical proximity with a higher level of customization in the profiles, including personal picture and graphic icons.

With these examples, the application to be developed in Second Life's main grid will use a simulated Bluetooth middleware, that will allow social encounters based on the proximity of other users and fixed points. The application uses only local resources, discarding any centralized database and enables a friend's list, profile editing and interaction with Bluetooth fixed points, marking important landmarks or objects, with the intuition of enabling information exchange. These functionalities will be explained in greater detail in the next section. Regarding the simulated hardware, it consists on a hand-held PDA or a simple mobile phone, with Bluetooth.

This simulation has the main purpose of evaluating Second Life's prototyping and simulation capabilities, reporting any issues regarding software development in the social software domain. The adopted architecture and implementation, along with the application's functionalities will be presented in the next sections.

5.2 Architecture

To define the system's architecture, a set of basic requirements were defined, as explained next.

The user must be able to produce a personal profile, based on a few basic informations, which will be compared using metadata tags on each field. Using the Bluetooth technology, users will be able to discover other nearby users with the corresponding profile matching value, which they can add to their friends list. The application also enables interaction with fixed points equipped with Bluetooth devices, allowing information exchange from each side.

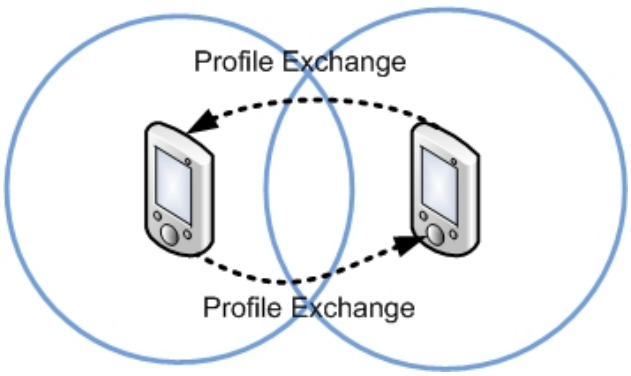

Figure 5.1: Profile Exchange with Local Database

From the study of social proximity applications based on the Bluetooth technology, two main system architectural models emerged:

- the local database model where each device stores the info related to it's user, e.g. the DigiDress[9] ;
- and the remote database model where, upon a social encounter the various devices use a unique mark to identify the peer, e.g. the BTID of a Bluetooth capable hardware, to fetch the user's info, e.g. Serendipity[45].

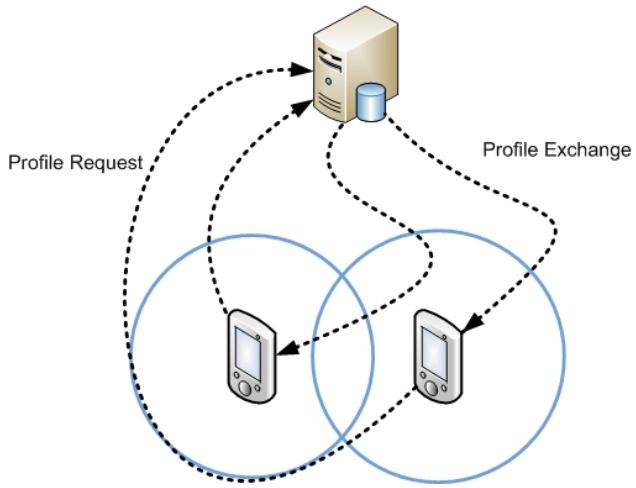

Figure 5.2: Profile Exchange with Remote Database

To provide a more mobile nature, based on the system's requirements, the local database model was chosen, discarding any type of central database.

As previously mentioned, there are two basic types of devices, the mobile which are the common PDA or cell phone where the application will allow the user to interact with parties, and the fixed which is used to share information about a particular landmark and the history of users that have been in that particular place and their feedback, within the context of the application

5.2.1 Mobile Client

As the reader can acknowledge from figure 5.3, the application is separated into three logical modules. The communication module is in charge of all Bluetooth primitives, that enables device to device message exchange.

The client also has a data management module where interactions are supervised from a higher level and data is formatted to be stored to or sent from the database module, where the friend's list, and own profile is stored. More detailed information about the profile module will be presented in sub-section 5.2.5.

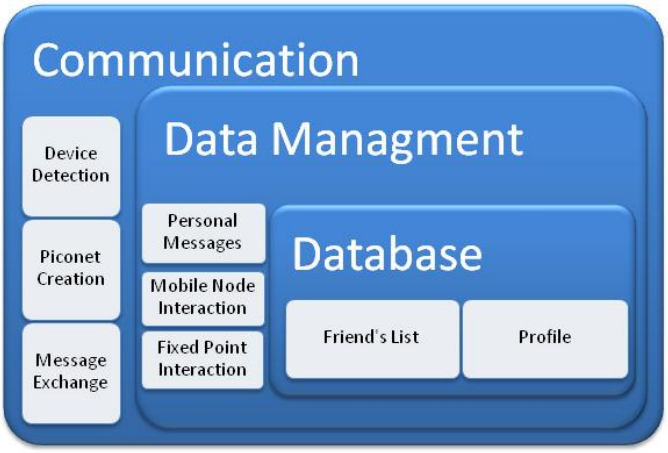

Figure 5.3: Mobile Client Architecture

5.2.2 Fixed Point Client

As it mobile homonymous, the static client has the basic communication module, responsible for all Bluetooth actions.

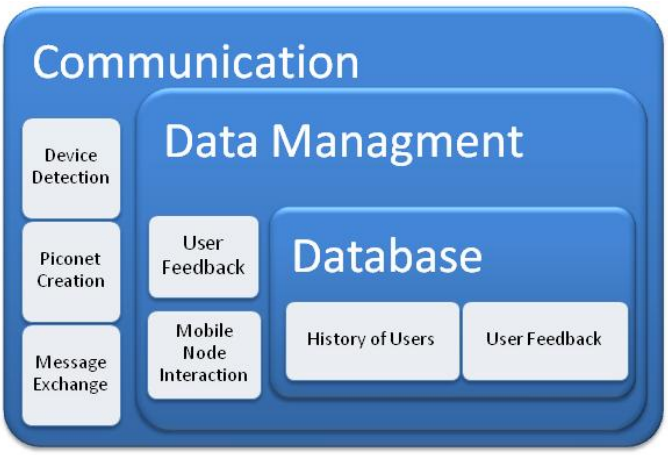

Figure 5.4: Fixed Point Client Architecture

The database module keeps a history of users that have either used this client or have been near it, depending on the application context, along with possible feedback. These informations are accessible through the data management module.By default, the fixed point client will always act as the master, in terms of creating the Bluetooth communication infrastructures, although this feature could be changed in different applications or contexts.

5.2.3 Bluetooth Middleware

Based on the actual model of the Bluetooth implementation, a middleware layer was stipulated to support the social application layer (much like the pub/sub layer in the first simulation). From this model four primitives emerge:

- **Scan**: used to detected all Bluetooth devices within a 20 m radius;
- **Create Piconet**: creates a private network were the devices can exchange messages;
- **Kill**: deletes a previous created piconet;
- **Send Message**: sends a message through a piconet previous created.

Respecting the real life protocol, in order to communicate with each other, a master device first must detect all nearby Bluetooth devices using the previously mentioned Scan function. Then a special primary handshake message is sent to acknowledge the state and compatibility of a particular device found in the scanning stage. If the potential slave responds, the master generates a random channel, and final special message is sent, indicating the creation of a piconet between the two devices, on that random channel. If no issue is found, the slave responds already on the random channel defined by the master. This offers a private channel, where the devices can exchange messages, like figure 5.5 shows.

When one of the devices wishes to terminate a connection, it sends special message to kill that same piconet.

5.2.4 Piconet

The piconet concept ensures the secure nature of all communications within the Bluetooth layer, as a particular randomly created channel is only used by the allowed devices (at least a master and one slave), in the geographical area supported by the protocol. This is insure by the fact that not only the master checks the viability of a channel generated, in the piconet creation stage, but also by the slave when a piconet creation message is received. If the slave is already using that channel it will inform the master to generate a different one. This feature allows the simulation to test security issues, which can prove very useful for later real life implementations.

5.2.5 Profile Characteristics and Matching

To simulation purposes, the profile information would be kept to its more basic form, with a nick field, which can (or not) correspond to the users real name, a welcome message, which will be displayed along with the nick on profile sharing, and a picture. Aside from this basic information the user may also be able to define its occupation, home town and likes, i.e., hobbies or simply a book the user recommends.

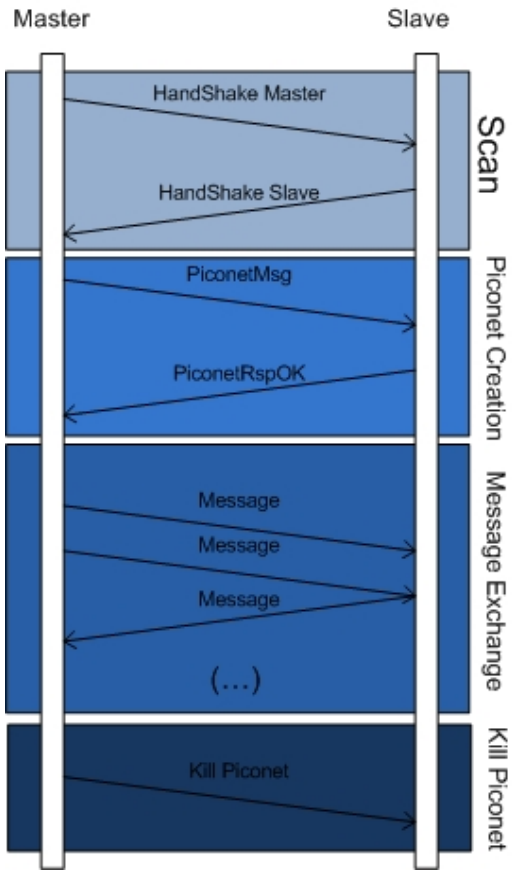

Figure 5.5: Middleware Bluetooth Protocol

Parallel to this profile information, a list of tags and correspondent user feedback will allow the previously mentioned profile matching. Each tag was associated a user personal information on that theme, e.g., in "comedy movies" tag the user may post "I' crazy about the simpsons!". The application defines a static list of tags, which define the matching core of al users, without discarding the possibility of adding other tags.

5.3 Implementation

The first step implementing the social proximity application in Second Life's^{TM} main grid, was to developed the Bluetooth middleware, to support the simulated application communications module. Then, the application layer was built, along with a graphical user interface, to simulate the software behaviour on a small mobile device. Both will be explained in detail in the next sub-sections.

5.3.1 Bluetooth Middleware Layer

The Bluetooth middleware was developed with the objective of being the most realistic possible, when comparing with the real life protocol, as shown in the architecture section.

As defined previously, when a Bluetooth device wishes to communicate with another, the first step is to scan the surrounding area for any devices present. For this purpose, the middleware layer is equipped with a scanning function called BlueScan(). This function uses the LSL sensor to detect other scripted objects within a pre defined radius. To every object detected the master device will send a handshake special message, to which any valid detected Bluetooth simulated devices will respond with another special handshake response, as shown in figure 5.5.

These first set of inter devices messages are transmitted using the llSay function in a private channel designated as the handshake channel (also known as the broadcast channels). This channel is where all communications take place, before a piconet is created between devices.

When the master receives a positive response to the first handshake message, from a slave, the CreatePiconet(key slave) function is invoke, and is responsible for creating a piconet with the slave identified by the *key* argument, as the name suggests. This function generates a random channel identifier, and checks if that particular channel is not already in use. If not, the master will send on the broadcast channel a special message, telling the salve to start using this random private channel, and starts to listen to that same channel. The slave will respond successfully, already in the new private piconet channel, pending that channel was not already in use, in another piconet, thus creating the piconet.

After this procedure, the two devices can now send messages to each other, enjoying a private communication bridge between them. To ensure the privacy and security of these piconets, every device stores the key of the devices it is connected two, along with the piconets channels (and listeners filters identifiers) used.

To stop using a particular piconet, a device sends the other one a special message, the indicates the termination of that channel, liberating it for future communications.

5.3.2 Application

With the Bluetooth middleware layer validated, an inter-layer communication protocol was defined, to enable the Bluetooth primitives to be used by the application layer.

5.3.2.1 Inter-Layer Communication

Based on the methodology of the smart room simulation, the inter-layer communication on this simulation is based on link messages.

As the Bluetooth layer was designed to offer four basic primitives, there were defined eight link messages, four for each layer, are presented in tables 5.1 and 5.2.

Table 5.1: Bluetooth Inter-Layer Protocol - Primitives

Type	Arguments	Id Num
Scan	Null	1
Create Piconet	Key Device	2
Kill Piconet	Key Device	3
Send Message	Key Device$_{Target}, StringMessage$	4

Table 5.1 reports the four primitives offered by the Bluetooth layer, as table 5.2 refers the messages generated and sent to the upper layers, to indicate Bluetooth events.

Table 5.2: Bluetooth Inter-Layer Protocol - Events

Type	Arguments	Id Num
Scan	String Device_Detected [or] NO_DEVICES	5
Create Piconet	String OK [or] DEVICE_NOT_AVAILABLE	6
Kill Piconet	String KILLED [or] ERROR	7
Send Message	Key Device_Sender, String Message [or] ERROR	8

These linked messages allow the application layers to take full advantage of the Bluetooth middleware, enabling the creation of secure communication channels through the piconet model mentioned previously.

5.3.2.2 Interaction with Mobile Devices

Profile With the inter-layer protocol defined, the pretended functionalities of the application were implemented. The first step was to define the profile module where all the user info is stored. As the only data structure available in LSL is the list type, this module was implemented using unbounded data fields. Regarding the tags and user feedback, two separate lists were used, where the same index relates the tag to the correspondent information.

The other desired aspect of the profile module was a personal picture. As mentioned in chapter three, uploading a picture to the Second Life^{TM} environment has a 10L$ fee. Although a symbolic price, based on the current currency rate, we did not want this fact to uphold any restriction on the application. This was bypassed combining the application profile picture with Second Life's^{TM} own profile picture. Every user within SL may wish to define a picture to their profile, among other things, which automatically creates a web page with the entire user's info, accessible using the UUID. As mentioned in chapter three, all objects in SL have a unique identifier, which is also valid for residents and pictures. So to apply the picture to any object - to allow the user to visualize it - the script needs only to know the picture's UUID, which is accessible through the resident's SL profile.

To fetch the picture's key, the application obtains the Bluetooth slave device owner's key, and using the llHTTPRequest function, downloads the user's SL profile web page.

Within this html document, is defined the personal picture key (if available), which through a parsing procedure is easily obtained.

Sniff Searching for nearby users (or Sniffing) is one of the key aspects of simulated application. When the user wishes to scan the surrounding area, the application layer informs the middleware to sniff for any Bluetooth mobile devices. If a device is found, the application connects the two using a new piconet, allowing the application to request the new device's user profile. The information exchanged consists on the nick, welcome message and the owner's key.

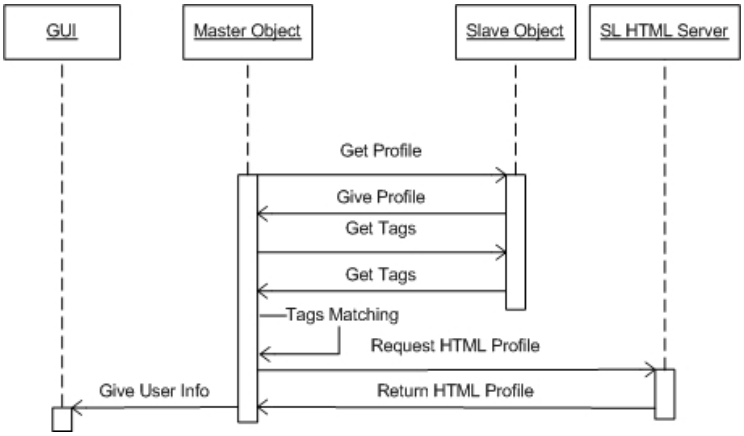

Figure 5.6: Profile Exchange Message Sequence

As figure 5.6 shows, using the owner's key, the application retrieves the photo, and calculates the number of tags matching. The final step is to relay all this information to the GUI object, presented in figure 5.7. Along with the sniffed profile information and tag matching, the Bluetooth object also sends the user's current friends list, to allow the interface to check if the sniffed user is already on the user's friends list. If not, the top right button allows the user to add that particular to his friends list.

5.3.2.3 GUI

The graphical user interface (GUI) is simulated using a link set of prims attached as HUD, as shown in figure 5.7.

The main prim in the link set and the Bluetooth device object communicate directly with each other, using the llSay function on a private channel. Whenever the Bluetooth object receives any kind of data, it processes it and relays to the HUD interface. Regarding the link set of prims that constitute the GUI, they communicate using link messages.

The graphic layout is based on a PDA interface, where each button is prim, along with the display screen, the main on the link set. The interface contemplates a main menu and three sub-menus, corresponding to the three main functionalities as seen in figure 5.8.

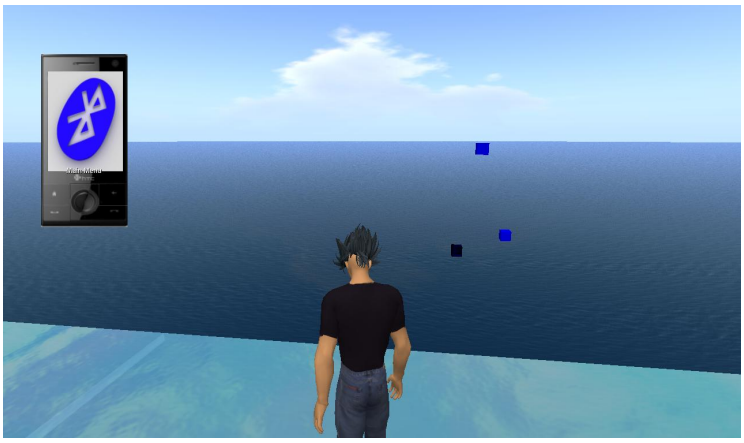

Figure 5.7: Graphical User Interface Layout

Note that in terms of LSL scripting, the three sub-menus correspond to three separate states.

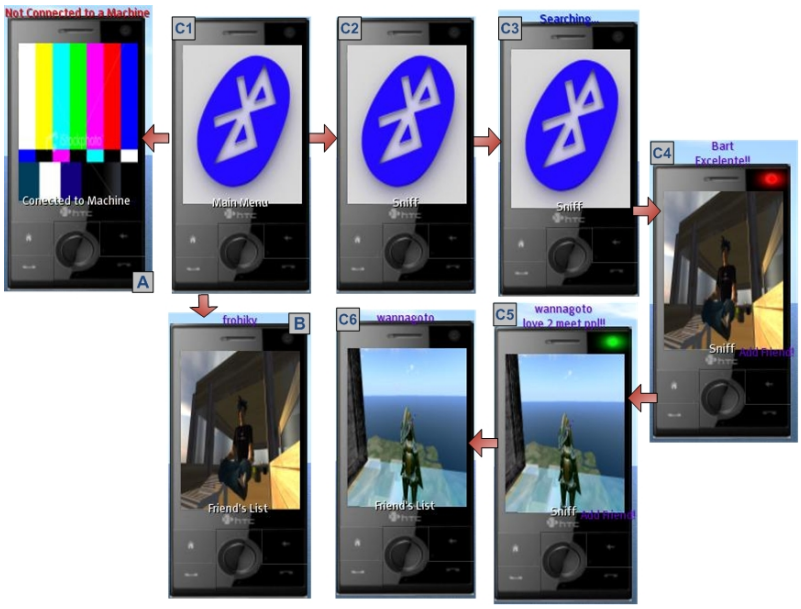

Figure 5.8: Menu Layout

From the main menu, using the navigation key (in the middle) the user changes through the sub-menus. The first one 5.8(A) corresponds to the interaction with static Bluetooth points, which in this case as the reader can observe, the Bluetooth mobile device is not connected to any valid point. The second sub-menu 5.8(B) displays the user's current friends list, using the two bottom buttons to browse through each one. The third and final sub-menu 5.8(C) corresponds to the sniffing mode, that when enabled allows, with on touch on the displays, to check for nearby users as previously explained. The users

detected will be displayed, through their picture nick and welcome message 5.8(C4 and C5). Along with this information a light on the top right corner will signal the matching between the current user, and the sniffed one, which will be green 5.8(C5) if any tags are in common, or in the contrary red (5.8C4). The user can also add the sniffed user to his friends list, if not already present 5.8(C6).

5.3.2.4 Interaction with Fixed Point

As mentioned previously, one of the key aspects of this simulation was to prototype the interaction between the user holding a mobile device, and a fixed Bluetooth access point, marking an important point regarding the location context, e.g., a painting in a museum, or a workout machine in a gym setting.

This last example was the scenario chosen, where users when using a particular machine, would be presented with the history of users that have used this machine and their respective feedback on it, and allow the user's own feedback as his presence is automatically stored. Embedded with a sensor, the machine detects whenever it is in use and connects to the user's Bluetooth device to allow a flow of information exchange.

In terms of Bluetooth methodology, the fixed machine is the master, i.e., starts the communication procedure, connecting to the user's mobile device. For each time the machine connects, it transmits a history of users that have used it (automatically storing the user's presence), along with the possibility of storing user feedback. The user may send a special comment message when using that particular machine, using his Bluetooth device that will be stored along with nick on the machine history memory.

Figure 5.9: Interaction with Fixed point - History of Users

This history of users is immediately sent to a new user, when connected to the machine, as figure 5.9 shows.

When there is no information to share, a special "no data" message is to sent, to inform the user that no one has ever used that machine, shown in fig 5.10.

Figure 5.10: Interaction with Fixed point - No Data Available

The visualisation of the history of users is much similar to the list of friends or sniffed devices, showing the user profile picture, nick and feedback on that machine, if any.

Apart from the obvious advantage of information exchange, that allows any user to evaluate the machine's utility, judging if it is the right one for him/her, the other main asset of this functionality is the ability to add friends when searching the history list, which creates numerous social encounters, based on the user's comment on that machine, or perhaps even the simply empathy given by the nick/picture.

So, apart from friends of friends and nearby devices, the application allows social networking based on past locations and feedback, which may be parallel to the user's personal taste.

Chapter 6

Analysis

In the course of the two simulations implementation, Second Life™ and Opensim's virtues and flaws were exposed, in terms of application prototyping and simulation within the ubiquitous computing domain, specifically pervasive environments and mobile social networks.

As referred in [46], the four main characteristics of a simulator within the ubiquitous computing scope are:

- Flexibility
- Usability
- Scability
- Stability

First, the diversity of simulation scenarios imposes the simulator to be flexible and the most generic possible, enabling multi-technological, large scale environments. This means the simulator must offer high level ubiquitous software simulation capabilities, but also enable prototyping lower level (middleware), to support those higher layers.

Second, usability in simulation context is as simple as it has to be easy for the users to get what they want. This means the user should not lose time in tasks that do not concern the simulation requirements, and easily prototype the system, and develop the applications desired.

Third, scalability in an ubicomp context, is the possibility of simulating many types of devices. In this project's context hardware is primarily based on mobile devices, and smart environments. A simulated device must support various types of applications, depending on the development of the middleware. Scalability also means that the simulator must support a large number of devices, e.g., a mobile social system, within a large physical area.

Fourth and finally, as the simulation scenarios often include a large number of devices or perhaps have a heavy load of information flowing through the communication channels, the simulator must be able to cope with these processing requirements, thus preventing any unwanted crash events while running the simulation. These aspects define the notion of simulator stability and are essential to the correctness and accuracy of any given simulation, in this field of research.

These fourth main characteristics combined, form the foundation of any ubiquitous computing simulator. With these notions in mind, the next two sections expose the evaluation supported by the two simulations developed in this project on the two environments, and a third and final section where the two are compared.

6.1 Second Life™ Evaluation

On the course of the work developed in Second Life™, the sim used offered us, almost ideal working conditions in terms of latency, which means the results discussed in the performance section are the most optimistic they can get.

As referred in chapter 3 the fact that Second Life™ runs on remote servers privately owned, subjects the user to those owner's will. Specifically this means random periods of downtime of those servers, which can undermine a whole days work. In the course of this project, the longest time of inability to connect to the main grid was of 6 hours, which include most of the designated work period of a day. Although not a critical aspect of Second Life™, it is important to retain and does undermine the simulator's stability.

As far as usability Second Life™, as referred in 3.2.2, is simple to use in terms of building and scripting, needing only a soft pre study, gratuitously found through the support wikis available online.

Concerning software complexity, i.e., the level that simulated software can achieve, SL scripting language (LSL) is fair as development tool but clouded by stability and scability issues. The first big problem encountered was the rigorous 16KB memory limit per script, which annuls any possibility of developing applications that demand great amounts of data storage. Logic components like databases are then almost impossible to test with reliable case studies, i.e., testing a database with only a couple of entries is rather pointless. As the reader observed in chapter 4, this impediment can be bypassed using an off-world server/database, relaying on "LSL-to-the-outside-world" communication bridges, HTTP requests and XML-RPC channels. But these bridges were proved to be quite unreliable, offering very little stability. This occurs due to another limit imposed by the Linden Labs company - in a effort to reduce server to client latency - concerning the number of HTTP allow in a small period of time, that proved to be too rigorous to allow any realistic and reliable access to outside applications in order to serve as a dependable tool, and the impossibility of using XML-RPC from the inside to off-world locations, parallel to a latency far to great to allow communications in resolvable times.

The issues concerning outside communication also undermine the possibility of Second Life^{TM} working as a mixed reality scenario, in systems that require a lot of data exchange, in short periods of time.But in less demanding applications, Second Life has proven to be fairly stable, has a pre-thesis simple simulation of a tagged RFID system, combined with a outside database, showed. This system simulated an entry point of a building, where personnel checked their entry time, by passing a RFID reader. Because, no more than 10-15 persons entered at the same time, the simulation had no crashing issues.

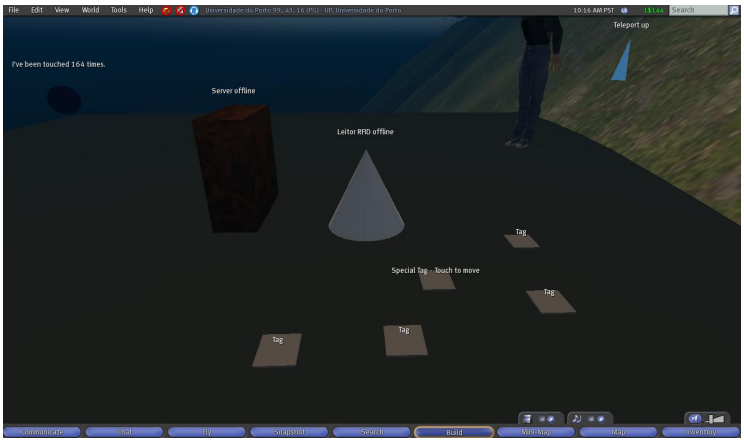

Figure 6.1: Simulated RFID System

Developing graphical user interfaces, to simulate user interaction with developed applications, has been proved possible although not reaching an optimal point. This component was simulated in the social proximity application simulation using prims attach to HUD points. Although the prims were embellished with texture to give a more natural appearance, i.e., a PDA image cropped and divided to build the display screen and a couple of buttons, this interface is directly connected to the main prim - the Bluetooth device - because attached prims do not offer all the LSL functions, e.g., sensing functions. This means that a communication channel was opened to allow the interface and the device to communicate with each other, sharing data. Although functional, this model generates dense coding, unusable in the real life implementation.

Despite all these issues, Second Life^{TM} proved to be able to support application development to a certain point. And offers a testing ground that few other simulators can offer, which is a dense and very larger social network, where social applications can be distributed, to return very valuable user feedback. As referred in [9, 10, 47], user feedback on a particular application requires a large community with certain demographic characteristics, i.e., to own a particular range of electronic devices that support that application. Second Life offers^{TM} a middle ground test for such applications, that can safe precious time and money to the developing entities.

6.2 Opensim Evaluation

As mentioned throughout the thesis, Opensim is an open source simulator, fact alone that gives a enormous advantage, when compared with Second Life™.

Although Opensim is currently in a early age of development, which translates into unfinished features, e.g., the incomplete implementation of all the aspects concerning LSL, that degrades the quality of the simulator, its open code nature allows a flexibility, in terms of simulation implementation and prototyping, unlike any other. The smart room implementation in this simulator, showed exactly this flexibility. Confronted with the fact that LSL's list type was not fully implemented, the solution was to embed the middleware layer into the simulator's own code, that reached a performance to which Second Life™ cannot simply compete.

Opensim also offers an important asset, which is security. Being a private server, the user has total control over the entire simulation environment, that not only justifies the great performance levels achieved, due to the isolation of the sim meaning that external factors do not weigh on the server, but also creates a physical frontier to outside networks, decreasing the probability of security breaches.

Of course, this isolation may be an unwanted factor, e.g., in a social application field test a social network is needed. But this is no issue to this simulator, because of its possibility to run on Grid Mode, which means that our private server can be reachable to outside users and even connect to other grids, building a main public grid, similar to the Second Life metaverse. In fact, the possibility of connecting Opensim servers to the SL public grid is under development as this thesis is being written.

In terms of usability, installing and configuring Opensim is fairly simple, although some issues may raise that might escape the knowledge scope of the most basic user. Despite this, it is our believe that the majority of the interested parties will not be caught off guard with the configuration procedure. Embedding new assets in the simulator's own code may at first be confusing, due to the large scale of the implementation, but by no means an impossible task. In fact recently documentation has been posted in the simulator's wiki [44], that facilitates this task, but unfortunately was not available in the time of the smart room implementation. In-world, the user interaction is basically the same comparing with Second Life™, which is simple and intuitive.

Software complexity in this simulator is only limited by the user's imagination and knowledge of the C# programming language. As the smart room simulation and mixed reality experiment showed, almost every software model can be embedded into the native code, and even outside communication bridges can be assembled, using mid level protocols like TCP. This means that programming issues related to LSL limitations and server imposed performance limits do not enter the Opensim equation.

6.3 Opensim Vs Second Life™

When comparing the two simulators, the obvious choice pends to Opensim, because the open source server offers an unbeatable processing performance, stability, better security characteristics, and programing possibilities that Second Life's™ native LSL implementation simply cannot deliver. Embedded coding in Opensim has been proved to be more effective and allow programming strategies that simply cannot exist in Second Life™.

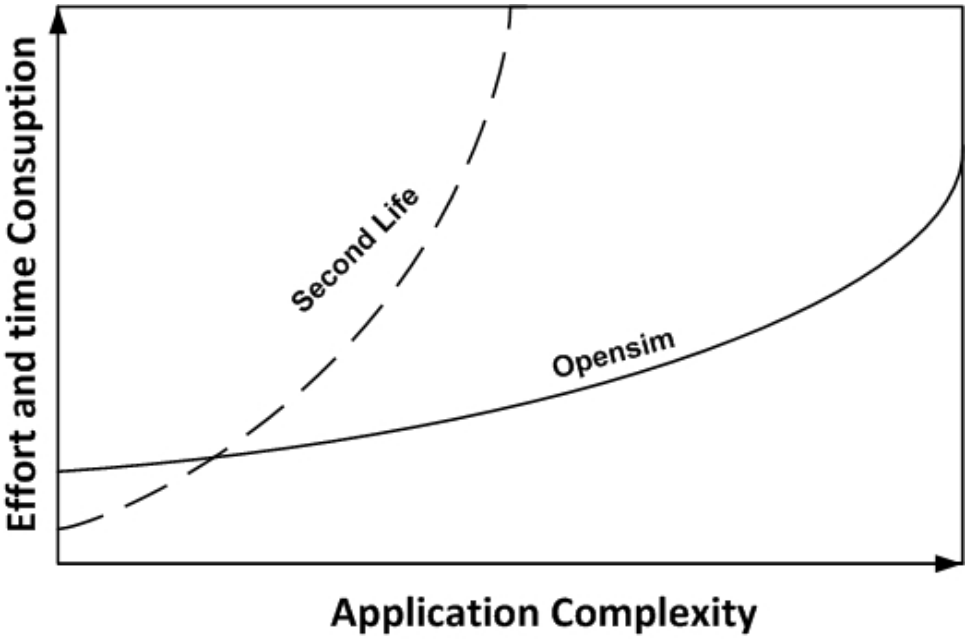

Figure 6.2: Second Life™ vs. Opensim

The social network of Second Life™ could be an advantage, when compared to Opensim, but even this is changing, due to the creation of numerous public grids running on Opensim and accessible to anyone.

Second Life™ only wins in the usability department, because it's server-client architecture is already available and fully configured, as in Opensim the user is required to configure it himself. But this fact by no means diminishes Opensim's overall greater performance levels and far wider possibilities in terms of application simulation in the ubiquitous computing field.

Choosing which simulator to use is directly related to the type of application subject of simulation and to what extent it will developed, as shown in figure 6.2. For more basic solutions, Second Life's™ simulation capabilities and performance levels suffices delivering a reasonable result. If a more complex level of simulation is required, only Opensim offers the flexibility and stability to support heavier requirements.

Chapter 7

Conclusion

7.1 Review of Contributions

As stated in the first chapter, the main goal of this project was to determine Second Life™ and Opensim's prototyping and simulation capabilities in terms of mobile applications and pervasive environments.

Evaluating the software complexity allowed in these simulators, along with performance levels, flexibility and usability, two major simulation scenarios were conducted. The first was a smart room environment with a publish/subscribe middleware backbone, supporting higher level applications. Using both Second Life and Opensim, various test cases were conducted, as referred in chapter 4, measuring the processing performance of the simulators, specifically in delivering events through the publish/subscribe simulated system.

The results exposed Opensim's greater stability, with the smallest delay times of all the simulated scenarios, and in the course of the implementation, it's flexibility was proved, allowing to developed new LSL primitives, characteristic not available in Second Life™. Despite this, Second Life™ proved able to support ubicomp application simulation to a certain point.

The second part of the work developed was the implementation of a mobile social application based on a Bluetooth middleware. Built on Second Life™, this scenario proved once again SL's capabilities to support ubicomp simulation, and perhaps even serve as a testing ground, using the dense social network already available.

In conclusion, Second Life™ and Opensim offer the requirements needed for a ubiquitous computing application simulation enviroments. Second Life™, more easier to use and little configuration requires, can support software development only to a certain point in complexity. Beyond that, only Opensim offers the flexibility and stability to support complex simulation of ubicomp applications.

7.2 Future Work

Following the work developed on the course of this project, further simulations would enlighten undiscovered issues, which the tested scenarios simply did not address. Specifically, an extensive simulation regarding sensing issues for example, would certainly help to reformulate the judgement about Second Life™ and Opensim's prototyping and simulation capabilities, because sensing and sensor networks play a key role in ubiquitous computing, as referred throughout this thesis, especially on the first two chapters. Although an experiment, prior to this project, regarding a RFID detection system was implemented, further development would be necessary to allow more profound conclusions, as the two main simulations scenarios gave.

Another important aspect that would enrich the quality of the simulations developed, and could be subject of future implementation are crash scenarios, introducing a statistical based error module. Although complex, this feature would allow Second Life™ and Opensim to achieve a more realistic simulation environment.

In the final stage of the smart room implementation, a mixed reality scenario was developed, and along with great performance results, this experiment opened numerous possibilities of dual systems, not only in terms of simulation but also in supervising and remote management applications. Further work on this mixed reality setting, introducing new test parameters and architectures in different ubicomp contexts, would allow a more precise evaluation on Second Life™ and Opensim's mixed reality capabilities. An interesting simulation could even involve mixed sensing, joining real life sensors and simulated ones, as already referred in this section.

The social application could also be developed into a mixed experiment, where real life Bluetooth devices could communicate with simulated ones fading the barrier between the two worlds.

Appendix A

Smart Room Simulation

A.1 In-World Implementation

A.1.1 Second Life - In-World Server

A.1.1.1 Publish/Subscribe Layer - Clients

```
integer comChannel= 666;
integer server_listen_handle;
integer first_listen_handle;
key ServerID;
//event-message parsing
string List2TypeCSV(list input) { // converts a list to a CSV string
with type information prepended to each item
  integer     i;
  list        output;
  integer     len;
  len=llGetListLength(input); //this can shave seconds off long lists
  for (i = 0; i < len; i++) {
    output += [llGetListEntryType(input, i)] + llList2List(input, i, i);
  }
  return llList2CSV(output);
}
list TypeCSV2List(string inputstring) { // converts a CSV string created with
List2TypeCSV back to a list with the correct type information
  integer     i;
  list        input;
  list        output;
  integer     len;
  input = llCSV2List(inputstring);
```

```
  len=llGetListLength(input);
  for (i = 0; i < len; i += 2) {
    if (llList2Integer(input, i) == TYPE_INTEGER) output += (integer)llList2String
    (input, i + 1);
    else if (llList2Integer(input, i) == TYPE_FLOAT) output += (float)llList2String
    (input, i + 1);
    else if (llList2Integer(input, i) == TYPE_STRING) output += llList2String(
    input, i + 1);
    else if (llList2Integer(input, i) == TYPE_KEY) output += (key)llList2String(
    input, i + 1);
    else if (llList2Integer(input, i) == TYPE_VECTOR) output += (vector)llList2String(
    input, i + 1);
    else if (llList2Integer(input, i) == TYPE_ROTATION) output += (rotation)llList2String
    (input, i + 1);
  }
  return output;
}
//publish-subscribe primitives
send_event(list event2send)
{
  list hEvent_sending= event2send;//["Subscribe","Button"];
  string message2send= List2TypeCSV(hEvent_sending);
  llSay(comChannel, message2send);
  //llOwnerSay("sent: " + message2send);
}
subscribe(string target)
{

  send_event(["Subscribe",target]);
  //subs_state=TRUE;

}
unsubscribe(string target)
{
  send_event(["Unsubscribe",target]);
  //subs_state=FALSE;

}
default
{
```

```
  state_entry()
  {
    send_event(["ServerID"]);
    first_listen_handle = llListen(comChannel, "", NULL_KEY, "");
  }
  link_message(integer sender_num, integer num, string str, key id)
  {
    if(num==1)    {
      subscribe(str);
    }
    else if(num==2)
    {
      unsubscribe(str);
    }
    else if(num==3)
    {
      send_event([str]);
    }
    else if(num==4)
    {
      //notify
    }
  }
  listen( integer channel, string name, key id, string message)
  {
    list hEvent_received = TypeCSV2List(message);
    //checks if the event is for this object and sends the event type
    //and source via lm
    if(llList2String(hEvent_received,0)==llGetObjectName())
      llMessageLinked(LINK_THIS,4,llList2String(hEvent_received,1),id);
    else if (llList2String(hEvent_received,0)=="ServerID")
    {
      llListenRemove(first_listen_handle);
      llListenRemove(server_listen_handle);
      ServerID = llList2Key(hEvent_received,1);
      server_listen_handle = llListen(comChannel,"",ServerID,"");
      // llOwnerSay("got key from server: " + (string)ServerID);
    }
  }
}
```

The Hello World! program in Java.

A.1.1.2 Publish/Subscribe Layer - Server

```
integer comChannel= 666;
integer managerChannel = 999;
integer listen_handle;
// timing
integer ts;
integer tr;
string List2TypeCSV(list input) { // converts a list to a CSV string with
 type information prepended to each item
  integer     i;
  list        output;
  integer     len;
  len=llGetListLength(input); //this can shave seconds off long lists
  for (i = 0; i < len; i++) {
    output += [llGetListEntryType(input, i)] + llList2List(input, i, i);
  }
  return llList2CSV(output);
}
list TypeCSV2List(string inputstring) { // converts a CSV string created with
List2TypeCSV back to a list with the correct type information
  integer     i;
  list        input;
  list        output;
  integer     len;

  input = llCSV2List(inputstring);

  len=llGetListLength(input);
  for (i = 0; i < len; i += 2) {
    if (llList2Integer(input, i) == TYPE_INTEGER) output += (integer)llList2String
    (input, i + 1);
    else if (llList2Integer(input, i) == TYPE_FLOAT) output += (float)llList2String
    (input, i + 1);
    else if (llList2Integer(input, i) == TYPE_STRING) output += llList2String
    (input, i + 1);
    else if (llList2Integer(input, i) == TYPE_KEY) output += (key)llList2String
    (input, i + 1);
    else if (llList2Integer(input, i) == TYPE_VECTOR) output += (vector)llList2String
```

```
    (input, i + 1);
    else if (llList2Integer(input, i) == TYPE_ROTATION) output +=
(rotation)llList2String
    (input, i + 1);
  }
  return output;
}
list Event_Heap;
integer add_event(list new_event)
{
  if(find_event(new_event)==-1){
    integer old_list_size = llGetListLength(Event_Heap);
    list temp = ["["] + new_event + ["]"];
    Event_Heap+= temp;
    return old_list_size;
  }
  else
  {
    return -1;
  }
}
integer remove_event(integer index)
{
  if(llList2String(Event_Heap,index)!="[")
  {
    //error
    return -1;
  }
  else
  {
    integer i=index+1;
    while(llList2String(Event_Heap,i)!="]")
    {
      i++;
    }
    Event_Heap=llDeleteSubList(Event_Heap,index,i);
  }
  return 0;
}
integer find_event(list event_ptr)
```

```
{
  list new_event = ["["] + event_ptr + ["]"];
  //llOwnerSay("procura: " + llList2CSV(new_event));
  return llListFindList(Event_Heap,new_event);
}
list find_subscriptions(list template)
{
  integer i;
  integer j;
  list subscrivers;
  for (i=0;i<llGetListLength(Event_Heap);i++)
  {
    if(llList2String(Event_Heap,i)=="[")
    {
      j=i+1;
      if(llList2String(Event_Heap,j)=="Subscribe" && llList2String
(Event_Heap,j+1)==llList2String(template,1))
        subscrivers+= llList2String(Event_Heap,j+2);
    }
  }
  return subscrivers;
}
integer find_subscription_ptr(list template)
{
  integer i;
  integer j;
  for (i=0;i<llGetListLength(Event_Heap);i++)
  {
    if(llList2String(Event_Heap,i)=="[")
    {
      j=i+1;
      if(llList2String(Event_Heap,j)=="Subscribe" && llList2String
(Event_Heap,j+1)==llList2String(template,1))
        return i;
    }
  }
  return -1;
}
list getEvent(integer index)
{
```

```
  if(llList2String(Event_Heap,index)!="[")
  {
    //error
    return [];
  }
  integer ptr=index;
  while(llList2String(Event_Heap,ptr)!="]")
  {
    ptr++;
  }
  return llList2List(Event_Heap,index,ptr);
}

integer del_subs(list template)
{
  integer ptr = find_subscription_ptr(template);
  integer error;
  //llOwnerSay("ptr do find: " + (string)ptr);
  if(ptr!=-1)
  {
    if(remove_event(ptr)!=-1) return 0;
    else return -1;
  }
  else return -1;
}
integer del_event(list event_ptr)
{
  integer ptr = find_event(event_ptr);
  integer error;
  // llOwnerSay("ptr do find: " + (string)ptr);
  if(ptr>-1)
  {
    if(remove_event(ptr)!=-1){ llOwnerSay("deleted!!"); return 0;}
    else return -1;
  }
  else return -1;
}
print_event_list()
{
  llOwnerSay("all: " + llList2CSV(Event_Heap));
```

```
}
send_event(list event2send)
{
  list hEvent_sending = event2send;//["ServerID", llGetKey()];
  string message2send = List2TypeCSV(hEvent_sending);
  llSay(comChannel, message2send);
}
//work flow
default
{
  state_entry()
  {
    llSetText("Server Offline", <1,1,1>, 1);
    llSetColor(<1.0, 1.0, 1.0>,ALL_SIDES);
  }
  touch_start(integer num)
  {
    state online;
  }
}
state online
{
  state_entry()
  {
    llSetText("Server Online", <1,1,1>, 1);
    llSetColor(<0.0, 0.0, 1.0>,ALL_SIDES);
    listen_handle = llListen(comChannel, "", NULL_KEY, "");
    // the server listens from any object, expcting events
    send_event(["ServerID", llGetKey()]);
  }
  touch_start(integer num)
  {
    print_event_list();
  }
  listen( integer channel, string name, key id, string message )
  {
    //   tr=llGetUnixTime();
    list hEvent_received = TypeCSV2List(message);

    if (llList2String(hEvent_received,0)=="ServerID")
```

```
    {
      //create and send event to target objects
      send_event(["ServerID", llGetKey()]);
    }
    else if(llList2String(hEvent_received,0)=="Subscribe")
    {
      hEvent_received+=name;
      if(find_event(hEvent_received)==-1)
        add_event(hEvent_received);
      else llOwnerSay("subscriver: " + llList2String(hEvent_received,1) +
" alreday subscribed!");
    }
    else if(llList2String(hEvent_received,0)=="Unsubscribe")
    {
      list event_temp = ["Subscribe",llList2String(hEvent_received,1),name ];
      del_event(event_temp);
    }
    else //if (llList2String(hEvent_received,0)=="Button")
    {
      //create and send event to target objects
      list template=["Subscribe",llList2String(hEvent_received,0)];
      list target=find_subscriptions(template);
      if(llGetListLength(target)>0)
      {
        integer j;
        for(j=0;j<llGetListLength(target);j++)
        {
          send_event([llList2String(target,j),"State",llList2String
(hEvent_received,0)]);
          //      ts=llGetUnixTime();
        }
      }
    }
  }
}
```

A.1.1.3 Application Layer - Button

```
integer NUM;
integer counter;
```

```
integer owner_listen;
integer object_listen;
integer CYCLE_CHANNEL = 999;
string CYCLE_MESSAGE = "time=";
float time;
float start;
default
{
  state_entry()
  {
    llSetText(llGetObjectName(), <1,1,1>, 1);
    llSetColor(<1.0, 1.0, 1.0>,ALL_SIDES);
    owner_listen=llListen(0, "", llGetOwner(), "");
    counter=0;
    counter=0;
    time=0;
    start=0;
  }
  touch_start(integer total_number)
  {
    //llSay(0, "Touched.");
    //llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
    if(NUM>0)state testing;
  }
  listen( integer channel, string name, key id, string message )
  {
    //llSay(0,"u said: " + message );
    if(llSubStringIndex(message, "num=")!=-1 )
    {
      list arguments = llParseString2List(message,["="],[]);
      llSay(0,"Defined number of cycles: " + llList2String(arguments,1));
      NUM=llList2Integer(arguments,1);
    }
  }
  state_exit()
  {
    llListenRemove(owner_listen);
  }
}
state testing
```

```
{
  state_entry()
  {
    counter=0;
    llSetText(llGetObjectName() + "....Testing!", <1,1,1>, 1);
    llSetColor(<1.0, 1.0, 1.0>,ALL_SIDES);
    object_listen=llListen(CYCLE_CHANNEL, "",NULL_KEY, "");
    llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
    start=llGetGMTclock();
  }
  listen( integer channel, string name, key id, string message )
  {
    //llSay(0,"u said: " + message );
    if(llSubStringIndex(message, CYCLE_MESSAGE)!=-1 )
    {
      list arguments = llParseString2List(message,["="],[]);
      float time_now=llList2Float(arguments,1)-start;
      // llOwnerSay("start: "+(string)start + " final: "+
(string)llList2Float(arguments,1) + " dif: " + (string)(time_now));
      time+=time_now;
      counter++;
      if(counter==NUM)
      {
        state default;
      }
      else
      {
        llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
        start=llGetGMTclock();
      }
    }
  }
  state_exit()
  {
    float median=(time/NUM);
    llOwnerSay("time of "  + (string)NUM +" cycles: "+(string)median);
    llListenRemove(object_listen);
    counter=0;
    time=0;
    start=0;
```

```
  }
}
```

A.1.1.4 Application Layer - Display

```
integer subs_state=FALSE;
integer turn=FALSE;
integer CYCLE_CHANNEL = 999;
string CYCLE_MESSAGE = "time=";
default
{
  state_entry()
  {
    llOwnerSay("ready!");
    llSetText(llGetObjectName() + " offline", <1,1,1>, 1);
    llSetColor(<0.0, 0.0, 0.0>,ALL_SIDES);
  }
  touch_start(integer num)
  {
    if(subs_state==FALSE){

      //subscribe("Button");
      llMessageLinked(LINK_THIS,1,"Button",NULL_KEY);
      subs_state=TRUE;
    }
    else if(subs_state==TRUE){

      //unsubscribe("Button");
      llMessageLinked(LINK_THIS,2,"Button",NULL_KEY);
      subs_state=FALSE;
    }
  }
  link_message(integer sender_num, integer num, string str, key id)
  {
    if(num==4)
    {
      llSay(CYCLE_CHANNEL,CYCLE_MESSAGE + (string)llGetGMTclock());
      if(str=="State") // event type
      {
        if(turn==FALSE){
          llSetText(llGetObjectName() + " online", <1,1,1>, 1);
```

```
          llSetColor(<0.0, 0.0, 1.0>,ALL_SIDES);
          turn=TRUE;
        }
        else if(turn==TRUE){
          llSetText(llGetObjectName() + " offline", <1,1,1>, 1);
          llSetColor(<0.0, 0.0,0>,ALL_SIDES);
          turn=FALSE;
        }
      }
    }
  }
}
```

A.1.2 Second Life - Off-World Server

A.1.2.1 Publish/Subscribe Layer (LSL) - Client

```
integer comChannel= 666;
integer server_listen_handle;
integer first_listen_handle;
//key ServerID;
string SERVER_URL="http://paginas.fe.up.pt/~ee02200/sl/rpc/server.php?";
key     gChannel; // my llRemoteData channel

DEBUG(list out) { // output debug info
  llSay(0, llList2CSV(out));
}
//publish-subscribe primitives
send_event(list event2send)
{  llHTTPRequest(SERVER_URL+"op="+llList2String(event2send,0)+
"&key="+(string)gChannel+"&type="+llList2String(event2send,1)+
"&subs=" +llGetObjectName(),[],"");
  llOwnerSay("sent" );
}
subscribe(string target)
{

  send_event(["Subscribe",target]);
  //subs_state=TRUE;
}
unsubscribe(string target)
```

```
{
  send_event(["Unsubscribe",target]);
  //subs_state=FALSE;
}
default { // initialize XML-RPC
  state_entry() {
    llOpenRemoteDataChannel(); // create an XML-RPC channel
    // this will raise remote_data event REMOTE_DATA_CHANNEL when created
  }
  remote_data(integer type, key channel, key message_id, string sender,
integer ival, string sval) {
    if (type == REMOTE_DATA_CHANNEL) { // channel created
      DEBUG(["Channel opened", "REMOTE_DATA_CHANNEL", channel, message_id,
sender, ival, sval]);
      gChannel = channel;
      llSay(0, "Ready to receive requests on channel \"" + (string)channel + "\"");
      state online; // start handling requests
    } else DEBUG(["Unexpected event type", type, channel, message_id,
sender, ival, sval]);
  }
}
state online
{
  link_message(integer sender_num, integer num, string str, key id)
  {
    if(num==1)
    {
      subscribe(str);
    }
    else if(num==2)
    {
      unsubscribe(str);
    }
    else if(num==3)
    {
      send_event(["Publish",str]);
    }
    else if(num==4)
    {
      //notify
```

```
    }
  }
  remote_data(integer type, key channel, key message_id, string sender,
integer ival, string sval) {
    if (type == REMOTE_DATA_REQUEST) { // handle requests sent to us
      //DEBUG(["Request received", "REMOTE_DATA_REQUEST", channel,
message_id, sender, ival, sval]);
      // handle request
      // llOwnerSay(sval);
      llMessageLinked(LINK_THIS,4,sval,message_id);
    } else DEBUG(["Unexpected event type:", type, channel, message_id,
sender, ival, sval]);
  }
  http_response(key id, integer status, list metadata, string body){
    if(status==200){
      // llOwnerSay("Ok");
    }
    else{
      //    llOwnerSay("Ocorreu um erro com o website, tente novamente.");
    }
  }
  state_exit() {
    llCloseRemoteDataChannel(gChannel);
    // clean up when you no longer want to handle requests
  }
}
```

A.1.2.2 Application Layer (LSL) - Button

```
integer NUM;
integer VALUES=50;
integer counter;
integer n_values;
integer owner_listen;
integer object_listen;
integer CYCLE_CHANNEL = 999;
string CYCLE_MESSAGE = "time=";
float time;
float start;

default
```

```
{
  state_entry()
  {
    llSetText(llGetObjectName(), <1,1,1>, 1);
    llSetColor(<1.0, 1.0, 1.0>,ALL_SIDES);
    owner_listen=llListen(0, "", llGetOwner(), "");
    counter=0;
    counter=0;
    time=0;
    start=0;
  }
  touch_start(integer total_number)
  {
    //llSay(0, "Touched.");
    //llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
    if(NUM>0)state testing;
  }
  listen( integer channel, string name, key id, string message )
  {
    //llSay(0,"u said: " + message );
    if(llSubStringIndex(message, "num=")!=-1 )
    {
      list arguments = llParseString2List(message,["="],[]);
      llSay(0,"Defined number of cycles: " + llList2String(arguments,1));
      NUM=llList2Integer(arguments,1);
    }
  }
  state_exit()
  {
    llListenRemove(owner_listen);
  }
}
state testing
{
  state_entry()
  {
    counter=0;
    n_values=0;
    llSetText(llGetObjectName() + "....Testing!", <1,1,1>, 1);
    llSetColor(<1.0, 1.0, 1.0>,ALL_SIDES);
```

```
    object_listen=llListen(CYCLE_CHANNEL, "",NULL_KEY, "");
    llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
    start=llGetGMTclock();

  }
  listen( integer channel, string name, key id, string message )
  {
    //llSay(0,"u said: " + message );
    if(llSubStringIndex(message, CYCLE_MESSAGE)!=-1 )
    {
      list arguments = llParseString2List(message,["="],[]);
      float time_now=llList2Float(arguments,1)-start;
      // llOwnerSay("start: "+(string)start + " final: "
+ (string)llList2Float(arguments,1) + " dif: " + (string)(time_now));
      time+=time_now;
      counter++;
      // llOwnerSay("counte " + (string)counter);
      if(counter==NUM)
      {
        n_values++;
        if(n_values>VALUES)
        {
          state default;
        }
        else
        {
          float median=(time/NUM);
          llOwnerSay((string) n_values + " "+(string)median);
          time=0;
          counter=0;
          llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
          start=llGetGMTclock();
        }
      }
      else
      {
        start=llGetGMTclock();
        if(counter%15==0){
          llSay(0,"sleeping");
          llSleep(10);
```

```
        }
        llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
      }
    }
  }
  state_exit()
  {
    llListenRemove(object_listen);
    counter=0;
    time=0;
    start=0;
    n_values=0;
  }
}
```

A.2 In-World and Mixed Reality - Opensim

A.2.1 Publish/Subscribe Classes

```
public class TCPclient
    {
        private TcpClient client;
        public TCPclient(TcpClient client)
        {
              this.client = client;
        }
        public string[] read()
        {
            NetworkStream ns = client.GetStream();
            String request = "";
            int numberOfBytesRead = 0;
            byte[] myReadBuffer = new byte[100];
            do
            {
                numberOfBytesRead = ns.Read(myReadBuffer, 0, myReadBuffer.Length);
                request =
                String.Concat(request, Encoding.ASCII.
                GetString(myReadBuffer, 0,
                numberOfBytesRead));
            } while (ns.DataAvailable);
```

```
            ns.Close();
           Console.WriteLine(request);
          string[] parser = request.Split(new char[] {'$'});
    return parser;
        }
     }
  public class Httpclient
  {
    private string SERVER_URL = "194.117.26.131";
    public Httpclient(string url)
    {
      if(url.Length>0)
        this.SERVER_URL=url;
    }
    public void http_request(string page_url)
    {
               string responseFromServer;
                try
                {
                //Console.WriteLine("page: " + page_url);
                // Create a request for the URL.
                HttpWebRequest request = (HttpWebRequest)
WebRequest.Create("http://" + SERVER_URL + "/" + page_url);
                // If required by the server, set the credentials.
                request.Credentials = CredentialCache.DefaultCredentials;
                           // Get the response.
                HttpWebResponse response = (HttpWebResponse)request.GetResponse();
                // Display the status.
                Console.WriteLine(response.StatusDescription);
                // Get the stream containing content returned by the server.
                Stream dataStream = response.GetResponseStream();
                // Open the stream using a StreamReader for easy access.
                StreamReader reader = new StreamReader(dataStream);
                // Read the content.
                responseFromServer = reader.ReadToEnd();
                // Display the content.
                //Console.WriteLine(responseFromServer);
                // Cleanup the streams and the response.
                reader.Close();
                dataStream.Close();
```

```
                response.Close();
              }
              catch (WebException e)
              {
                  //return "error";
              }
    return;
    }
  }
  public class PubSub
  {
    public bool flag=false;
    private static PubSub ps=null;
    private Subscriptions sub_list = null;
    private Httpclient client = new Httpclient("");
    public delegate void NotifyDelegate(string[] arguments);
    public event NotifyDelegate NotifyEvent;
    private Thread thread;
    private int port =9500;
    public void send_event(string evt, string publisher,string argument){
    client.http_request("Publish?event="
+ evt + "&publisher=" +publisher + "&arg=" + argument);
    }
    public  void start_th()
    {
           thread = new Thread(new ThreadStart(this.listen));
      thread.Start();
    }
    public static PubSub getPS ()
    {
      if(ps==null){
        ps = new PubSub();
        ps.start_th();
      }
      return ps;
    }
    public Subscriptions getList()
    {
      if (sub_list==null)
        sub_list= new Subscriptions();
```

```
    return sub_list;
  }
  public void listen()
        {
              try
              {
                  IPAddress addr = IPAddress.Parse("194.117.26.129");
                  TcpListener server = new TcpListener(addr, port);
                  server.Start();
                  while (true)
                  {
                        TcpClient client = server.AcceptTcpClient();
                        TCPclient tcp = new TCPclient(client);
                        //Thread thread = new Thread(new ThreadStart(tcp.read));
                        Thread thread = new Thread(delegate()
                        { NotifyEvent(tcp.read()); });
                        thread.Start();
                  }
              }
              catch (ArgumentNullException e)
              {
                  Console.WriteLine("ArgumentNullException: {0}", e);
              }
              catch (SocketException e)
              {
                  Console.WriteLine("SocketException: {0}", e);
              }
              //this.notified = response;
              //return response;
        }
}
public class Subscription
{
  private string eventType;
  private string subscriber;
  private LLUUID subscriberID;
  private LSL_BuiltIn_Commands scripteng;
  //private uint localID;

      public Subscription(string eT, string subs, LLUUID UUID,
```

```
LSL_BuiltIn_Commands se)
        {
       this.eventType=eT;
      this.subscriber=subs;
      this.subscriberID=UUID;
      this.scripteng=se;

    }
          public static bool operator ==(Subscription a, Subscription b)
          {
                // If both are null, or both are same instance, return true.
                if (System.Object.ReferenceEquals(a, b))
                {
                    return true;
                }
                // If one is null, but not both, return false.
                if (((object)a == null) || ((object)b == null))
                {
                    return false;
                }
                return a.eventType == b.eventType && a.Subscriber
== b.Subscriber && a.UUID == b.UUID;
          }
          public static bool operator !=(Subscription a, Subscription b)
          {
                return !(a == b);
          }
          public string EventType
            {
          get{return eventType;}
          set{eventType=value;}
        }
        public string Subscriber
        {
                get{return subscriber;}
          set{subscriber=value;}
        }
    public LLUUID UUID
    {
      get{return subscriberID;}
```

```
  set{subscriberID=value;}
}
public LSL_BuiltIn_Commands ScriptEng
{
  get{return scripteng;}
  set{scripteng=value;}
}
  public class Subscriptions
  {
private List<Subscription> SubsList= new List<Subscription>();

      public Subscriptions()
      {
 }
private int find_Subscription(Subscription subs)
{
  for(int i=0;i<SubsList.Count;i++)
  {
    if(subs==SubsList[i])
    return i;
  }
  return -1;
}

      private List<Subscription> find_Subscribers(string event2search)
{
            List<Subscription> list = new List<Subscription>();
  for(int i=0;i<SubsList.Count;i++)
  {
               if (event2search == SubsList[i].EventType)
      list.Add(SubsList[i]);
  }
  return list;
}
public bool Subscribe(Subscription subs)
{
  if(find_Subscription(subs)==-1)
  {
    SubsList.Add(subs);
    return true;
```

```
      }
      else return false;

    }
    public bool Unsubscribe(Subscription subs)
    {
      int ptr;
      if((ptr=find_Subscription(subs))!=-1)
      {
        SubsList.RemoveAt(ptr);
        return true;
      }
      else return false;
                //return SubsList.Remove(subs);
    }

    public List<Subscription> Publish(string event2pub)
    {
                List<Subscription> subscribers = new List<Subscription>();
                subscribers = this.find_Subscribers(event2pub);
      return subscribers;
    }
  }
```

A.2.2 Publish/Subscribe-LSL Interface

```
  public void PS_Notify(string[] arguments)
  {
    m_host.AddScriptLPS(1);
    List<Subscription> subs= new List<Subscription>();
    subs=PubSub.getPS().getList().Publish(arguments[0]);
    for(int i=0;i<subs.Count;i++)
    {
      object[] param = new object[] {arguments[1], subs[i].EventType, arguments[2]};
      subs[i].ScriptEng.m_ScriptEngine.m_EventQueueManager.
AddToScriptQueue(subs[i].ScriptEng.m_localID, subs[i].ScriptEng.m_itemID,
"publish", EventQueueManager.llDetectNull, param);
    }
  }
  public void llPSSubscribe(string event2target)
  {
```

```
     m_host.AddScriptLPS(1);
     World.SimChat(Helpers.StringToField("subs: " + event2target),
ChatTypeEnum.Say, 0, m_host.AbsolutePosition, m_host.Name, m_host.UUID);
     IWorldComm wComm=m_ScriptEngine.World.
RequestModuleInterface<IWorldComm>();
     wComm.DeliverMessage(m_host.UUID.ToString(),
ChatTypeEnum.Say, 0, m_host.Name, event2target);
     PubSub.getPS().getList().Subscribe(new Subscription
(event2target, m_host.Name,m_host.UUID, this));
  }
  public void llPSUnsubscribe(string event2target)
  {
    m_host.AddScriptLPS(1);
    World.SimChat(Helpers.StringToField("unsubs: " + event2target),
ChatTypeEnum.Say, 0, m_host.AbsolutePosition, m_host.Name, m_host.UUID);
    IWorldComm wComm=m_ScriptEngine.World.
RequestModuleInterface<IWorldComm>();
    wComm.DeliverMessage(m_host.UUID.ToString(),
ChatTypeEnum.Say, 0, m_host.Name, event2target);
    PubSub.getPS().getList().Unsubscribe
(new Subscription(event2target, m_host.Name,m_host.UUID, this));
  }
  public void llPSPublish(string llevent, string argument)
  {
    m_host.AddScriptLPS(1);
    List<Subscription> subs= new List<Subscription>();
    subs=PubSub.getPS().getList().Publish(llevent);
    PubSub.getPS().send_event(llevent,m_host.Name,argument);
    for(int i=0;i<subs.Count;i++)
    {
      object[] param = new object[] { m_host.Name, subs[i].EventType,argument};
      subs[i].ScriptEng.m_ScriptEngine.m_EventQueueManager.
AddToScriptQueue(subs[i].ScriptEng.m_localID,
subs[i].ScriptEng.m_itemID, "publish", EventQueueManager.llDetectNull, param);
    }
  }
  public double llString2Float(string arg)
  {
    m_host.AddScriptLPS(1);
    return System.Double.Parse(arg);
```

```
  }
```

A.2.3 LSL Script - Button

```
integer NUM=100;
integer VALUES=50;
integer counter;
integer n_values;
integer owner_listen;
integer object_listen;
integer CYCLE_CHANNEL = 10;
string CYCLE_MESSAGE = "time=";
float time;
float start;
default
{
  state_entry()
  {
    counter=0;
    n_values=0;
    llSetText(llGetObjectName() + "....Testing!", <1,1,1>, 1);
    llSetColor(<1.0, 1.0, 1.0>,ALL_SIDES);
    object_listen=llListen(CYCLE_CHANNEL, "",NULL_KEY, "");
    llPSSubscribe("time4");
    llPSPublish(llGetObjectName(),"app");
    //llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
    start=llGetUnixTime();
  }
  publish(string sender, string events, string args)
  {
    if(events=="time4"){
      float time_now=(float)args-start;
      //time+=time_now;
      counter++;
      //string ok = DateTime.Now.ToString();
      // llOwnerSay("counte " + (string)counter);
      if(counter==NUM)
      {
        n_values++;
        if(n_values>VALUES)
        {
```

```
          llSay(0,"finished");
        }
        else
        {
          float median=(time/NUM);
          //llOwnerSay((string) n_values + " "+(string)median);
          llInstantMessage(llGetOwner(),(string) n_values + " "+(string)median);
          time=0;
          counter=0;
          llPSPublish(llGetObjectName(),"ok");
          //llMessageLinked(LINK_THIS,3,llGetObjectName(),NULL_KEY);
          start=llGetUnixTime();
        }
      }
      else
      {
        start=llGetUnixTime();
        llPSPublish(llGetObjectName(),"app");
      }
    }
  }
```

A.2.4 LSL Script - Display

```
integer subs_state=FALSE;
integer turn=FALSE;
integer CYCLE_CHANNEL = 10;
string CYCLE_MESSAGE = "time=";

default
{
  state_entry()
  {
    llOwnerSay("ready!");
    llSetText(llGetObjectName() + " offline", <1,1,1>, 1);
    llSetColor(<0.0, 0.0, 0.0>,ALL_SIDES);
  }
  publish(string ok, string hf,string arg)
  {
```

```
    // llSay(CYCLE_CHANNEL,CYCLE_MESSAGE + (string)llGetTimestamp());
    llPSPublish("time", llGetTimestamp());
    if(hf=="Button1") // event type
    {
      if(turn==FALSE){
        llSetText(llGetObjectName() + " online", <1,1,1>, 1);
        llSetColor(<0.0, 0.0, 1.0>,ALL_SIDES);
        turn=TRUE;
      }
      else if(turn==TRUE){
        llSetText(llGetObjectName() + " offline", <1,1,1>, 1);
        llSetColor(<0.0, 0.0,0>,ALL_SIDES);
        turn=FALSE;
      }
    }
  }
  touch_start(integer num)
  {
    if(subs_state==FALSE){
      llPSSubscribe("Button1");
      subs_state=TRUE;
    }
    else if(subs_state==TRUE){
      llPSUnsubscribe("Button1");
      subs_state=FALSE;
    }
  }
}
```

Appendix B

Social Proximity Simulation

B.1 Bluetooth Layer

```
//memory
list scanDevices;
//piconets
list piconets;
list piconetDevices;
list piconetsHandlers;
string  piconetMsg = "PICONETGO=";
string  piconetRspOK = "PICONETOK=";
string  piconetRspKO = "PICONETKO=";
string  PICONETKILL="PICONETKILL";
//handshakes...
integer handShakeChannel = 1000;
string  handShakeMessage_Master = "BLUETOOTHHELLO=";
string  handShakeMessage_Slave = "BLUETOOTHOK=";
//scanning
integer RADIUS=15;
integer SCAN_TIME=5;
// linked message strings
string KILLED="killed";
string ERROR="error";
string DEVICE_NOT_AVAILABLE="device not available";
string OK = "ok";
string OK_SLAVE = "ok_slave";
//rands
    integer randInt(integer n)
    {
```

```
        return (integer)llFrand(n + 1);
    }

    integer randIntBetween(integer min, integer max)
    {
        return min + randInt(max - min);
    }
//functions
integer blueScan()
{
    scanDevices = []; // clear detected devices list...
    llSensor("",NULL_KEY,SCRIPTED,RADIUS, PI);
    llSetTimerEvent(SCAN_TIME);
    return 0;
}
integer sendHandShake(key device) // sends a hand shake to
check if the device has a valid bluetooth hardware
{
    llSay(handShakeChannel,handShakeMessage_Master + "=" + (string)device);
    return 0;
}
integer respondeHandShake(key master)
{
    llSay(handShakeChannel,handShakeMessage_Slave + "=" + (string)master);
    return 0;
}
integer createPiconet(key device)
{
    integer channel = randIntBetween(1100,10000);
    if(llListFindList(piconets,[channel]) ==-1)
    {
        llSay(handShakeChannel,piconetMsg + (string)channel + "=" + (string)device);
        integer handler=llListen(channel,"",device,"");
        piconets = (piconets=[]) + piconets + [channel];
        piconetDevices = (piconetDevices=[]) + piconetDevices + [device];
        piconetsHandlers = (piconetsHandlers=[]) + piconetsHandlers + [handler];
    }
    else
        createPiconet(device);
    return channel;
```

```
}
integer respondePiconet(integer channel, key master)
{
    if(llListFindList(piconets,[channel]) ==-1)
    {
        llSay(channel,piconetRspOK  + (string)master);
        integer handler=llListen(channel,"",master,"");
        piconets = (piconets=[]) + piconets + [channel];
        piconetDevices = (piconetDevices=[]) + piconetDevices + [master];
        piconetsHandlers = (piconetsHandlers=[]) + piconetsHandlers + [handler];
        return TRUE;
    }
    else
    {
        llSay(handShakeChannel,piconetRspKO + (string)channel + "=" + (string)master);
        return FALSE;
    }
}
integer sendMsg(key device, string message)
{
    integer index;
    if((index=llListFindList(piconetDevices,[device])) !=-1)
    {
        llSay(llList2Integer(piconets,index),message);
        return TRUE;
    }
    else return FALSE;
}
integer killPiconet(key device)
{

    if (sendMsg(device,PICONETKILL)==TRUE)
    {
        integer index=llListFindList(piconetDevices,[device]);
        llListenRemove(llList2Integer(piconetsHandlers,index));
        piconets = llDeleteSubList(piconets, index, index);
        piconetDevices = llDeleteSubList(piconetDevices, index, index);
        piconetsHandlers = llDeleteSubList(piconetsHandlers, index, index);
        return TRUE;
    }
```

```
    else return FALSE;
}
default
{
    state_entry()
    {
        llSay(0, "Hello, Avatar!");
        llListen(handShakeChannel,"",NULL_KEY,"");
    }

    sensor(integer num)
    {
        integer i;
        for (i=0;i<num;i++){
            //send handshake
            sendHandShake(llDetectedKey(i));
        }
    }
    no_sensor()
    {
        scanDevices = [];
    }
    listen( integer channel, string name, key id, string message )
    {
        if(channel == handShakeChannel)
        {
            if(llSubStringIndex(message, handShakeMessage_Master)!=-1)
            // i'm a slave...received handshake from master...
            {
                list arguments = llParseString2List(message,["="],[]);
                if(llList2Key(arguments,1)==llGetKey())
                {
                     respondeHandShake(id);
                }
            }
            else if(llSubStringIndex(message, handShakeMessage_Slave)!=-1)
            {
                list arguments = llParseString2List(message,["="],[]);
                if(llList2Key(arguments,1)==llGetKey())
                {
```

```
                    scanDevices = (scanDevices=[]) + scanDevices + [id];
                }
            }
            else if(llSubStringIndex(message, piconetRspKO)!=-1)
            {
                list arguments = llParseString2List(message,["="],[]);
                if(llList2Key(arguments,2)==llGetKey())
                {
                    integer index = llListFindList(piconetDevices,[id]);
                    piconets = llDeleteSubList(piconets, index, index);
                    piconetDevices =
                    llDeleteSubList(piconetDevices, index, index);
                    piconetsHandlers =
                    llDeleteSubList(piconetsHandlers, index, index);
                    llMessageLinked(LINK_THIS,7,KILLED,id);
                    // send data to upper layers
                }
            }
            else if(llSubStringIndex(message, piconetMsg)!=-1 )
            // master wants to create piconet...
            {
               list arguments = llParseString2List(message,["="],[]);
               if(llList2Key(arguments,2)==llGetKey())
               {
                    respondePiconet(llList2Integer(arguments,1),id);
                    llMessageLinked(LINK_THIS,6,OK_SLAVE,id);
                }
            }
        }
        else
        {
            integer index;
            if(llSubStringIndex(message, piconetRspOK)!=-1 &&
llListFindList(piconets,[channel]) !=-1) //piconet
            {
                 llMessageLinked(LINK_THIS,6,OK,id);
                 // send data to upper layers
            }
            else if((index=llListFindList(piconets,[channel])) !=-1 &&
llListFindList(piconetDevices,[id]) !=-1) // msg
```

```
            {
                if (message==PICONETKILL)
                {
                    llListenRemove(llList2Integer(piconetsHandlers,index));
                    piconets = llDeleteSubList(piconets, index, index);
                    piconetDevices = llDeleteSubList(piconetDevices, index, index);
                    piconetsHandlers = llDeleteSubList(piconetsHandlers, index, index);
                    llMessageLinked(LINK_THIS,7,KILLED,id); // send data to upper layers
                }
                else
                {
                    llMessageLinked(LINK_THIS,8,message,id); // send data to upper layers
                }
            }
        }
    }
    timer()
    {
        if(llGetListLength(scanDevices)>0)
        {
            llMessageLinked(LINK_THIS,5,llList2CSV(scanDevices),NULL_KEY);
           // send data to upper layers
            llSetTimerEvent(0);
        }
        else
        {
            llMessageLinked(LINK_THIS,5,"none",NULL_KEY); // send data to upper layers
            llSetTimerEvent(0);
        }
        llSetTimerEvent(0);   // stop timer
    }
    link_message(integer sender_num, integer num, string str, key id)
    {
        if(num==1) // scan
        {
            blueScan();
        }
        else if(num==2) // create piconet
        {
            integer index;
```

```
            if((index=llListFindList(scanDevices,[id]))!=-1)
            // device is available
            {
                createPiconet(id);
            }
            else //source not found....
            {
                llMessageLinked(LINK_THIS,6,DEVICE_NOT_AVAILABLE,id);
                // send data to upper layers
            }
        }
        else if(num==3) // kill piconet
        {
            if(killPiconet(id)==TRUE)
                llMessageLinked(LINK_THIS,7,KILLED,id);
                // send data to upper layers
            else
                llMessageLinked(LINK_THIS,7,ERROR,id);
                // send data to upper layers
        }
        else if(num==4) // send msg
        {
            if(sendMsg(id,str)==FALSE)
                llMessageLinked(LINK_THIS,4,ERROR,id);
                // send data to upper layers
        }
    }
}
```

B.2 Application Layer

B.2.1 Mobile Client

```
integer DATACHANNEL=888;
key machine_using;
// profile data;
string nick="Master";
string ocupation="trainee";
string home="porto";
string favourites="soccer,movies and music!";
string welcome_msg="check out my profile!!";
```

```
//memory
list tags = ["horror movies","comedy movies",
"romantic movies", "documentaries"];
list info = ["I like the ring", "love the beed movie!",
"hate P.S. I love you!", "I like  Michael Moore"];
string tex;
//buddy list
list friends = ["b45c0b67-d4a9-4c17-bc92-184565c81f00|frohiky"];
//sniffs
list nicks;
list messages;
list reqs;
list buddie_reqs;
//devices  found by the sniff function
integer n_dev=0;
//messages
string GETPROFILE = "givemeprofile";
string GIVEPROFILE = "myprofile=";
string SETTEXTURE = "settexture=";
string CLEANTEXTURES = "cleantextures";
string GETTAGS = "givemetags=";
string GIVETAGS = "mytags=";
string SETPROFILES = "profiles=";
string GIVEBUDDIES = "buddies=";
string GIVEBUDDIE = "buddie=";
string MENU = "menu=";
//HUD
integer pos=0;
integer DisplayBanner = TRUE;
float Size = 2.50
float Height = 2.50;
list DefaultTexturePalette = ["7cfd684e-2141-941c-eac8-bd439f0d5a9f",
"02a9cc16-23e5-c355-ca17-a98fc662254f",
"9acff594-c423-40ff-79f1-6a79102cde14"];
ApplyDefaultTexture()
{
    ApplySelectedTexture
    ((key)llList2String(DefaultTexturePalette,
    (integer)llFrand((float)llGetListLength
    (DefaultTexturePalette))));
```

```
}
ApplySelectedTexture(key texture)
{
    llParticleSystem([
        PSYS_PART_FLAGS, 0,
        PSYS_SRC_PATTERN, 4,
        PSYS_PART_START_ALPHA, 0.50,
        PSYS_PART_END_ALPHA, 0.50,
        PSYS_PART_START_COLOR, <1.0,1.0,1.0>,
        PSYS_PART_END_COLOR, <1.0,1.0,1.0>,
        PSYS_PART_START_SCALE, <Size * 1.6 ,Size,0.00>,
        PSYS_PART_END_SCALE, <Size * 1.6,Size,0.00>,
        PSYS_PART_MAX_AGE, 1.20,
        PSYS_SRC_MAX_AGE, 0.00,
        PSYS_SRC_ACCEL, <0.0,0.0,0.0>,
        PSYS_SRC_ANGLE_BEGIN, 0.00,
        PSYS_SRC_ANGLE_END, 0.00,
        PSYS_SRC_BURST_PART_COUNT, 8,
        PSYS_SRC_BURST_RADIUS, Height,
        PSYS_SRC_BURST_RATE, 0.10,
        PSYS_SRC_BURST_SPEED_MIN, 0.00,
        PSYS_SRC_BURST_SPEED_MAX, 0.00,
        PSYS_SRC_OMEGA, <0.00,0.00,0.00>,
        PSYS_SRC_TEXTURE, texture]);
llSetTexture(texture, ALL_SIDES);
}
integer AddFriend(string nick)
{
    if(llListFindList(friends,[nick])==-1)
    {
        friends = (friends=[]) + friends + [nick];
        return TRUE;
    }
    else return FALSE;
}
key getPhoto(key user_key)
{
    key req;
    string URL_RESIDENT = "http://world.secondlife.com/resident/";
    req=llHTTPRequest( URL_RESIDENT + (string)user_key,[HTTP_METHOD,"GET"],"");
```

```
    return req;
}
sendBuddiesPhoto()
{
    integer i;
    key req;
    for(i=0;i<llGetListLength(friends);i++)
    {
        string URL_RESIDENT = "http://world.secondlife.com/resident/";
        req=llHTTPRequest( URL_RESIDENT +
        (string)llGetOwnerKey(getKey(llList2String(friends,i))),
       [HTTP_METHOD,"GET"],"");
        buddie_reqs = (buddie_reqs=[]) + buddie_reqs + [req];
    }

}
sendBuddiePhoto(string nick)
{
    key req;
    string URL_RESIDENT = "http://world.secondlife.com/resident/";
    llOwnerSay("buddie " + (string)getKey(nick));
    req=llHTTPRequest( URL_RESIDENT +
    (string)llGetOwnerKey(getKey(nick)),
    [HTTP_METHOD,"GET"],"");
    buddie_reqs = (buddie_reqs=[]) + buddie_reqs + [req];
}
string getNick(string str)
{
    if(str!="")
    {
        list arguments = llParseString2List(str,["|"],[]);
        return llList2String(arguments,1);
    }
    else return "";
}
key findNick(string name)
{
    integer i;
    for(i=0;i<llGetListLength(nicks);i++)
    {
```

```
        if(getNick(llList2String(nicks,i))==name)
            return getKey(llList2String(nicks,i));
    }
    return NULL_KEY;
}
key getKey(string str)
{
        if(str!="")
    {
        list arguments = llParseString2List(str,["|"],[]);
        return llList2String(arguments,0);
    }
    else return NULL_KEY;
}
Sniff()
{
    pos=0;
    n_dev=0;
    llMessageLinked(LINK_THIS,1,"","");
    llSay(665,MENU + "sniff");
    llSleep(0.5);
    llSay(666,CLEANTEXTURES);
    llSay(666,GIVEBUDDIES + llList2CSV(friends));
}
default
{
    state_entry()
    {
        llSay(0, "Hello, Avatar!");
        llListen(666,"","","");
        llListen(667,"","","");
        llParticleSystem([]);
       // llRequestPermissions(llGetOwner(),PERMISSION_ATTACH);
        pos=0;
        n_dev=0;
        nick = (string)llGetKey() + "|Master";

    }
    touch_start(integer total_number)
    {
```

```
        Sniff();
       // llRezObject("profiler",llGetPos() + <0,1,0>,
       ZERO_VECTOR, ZERO_ROTATION, n_dev);

    }
    listen( integer channel, string name, key id, string message )
    {
        if (channel==DATACHANNEL && id==llGetOwner())
        {
            if(llSubStringIndex(message, "nick=")!=-1 )
            {
                list arguments = llParseString2List(message,["="],[]);
                nick = (string)llGetKey() + "|" + llList2String(arguments,1);
                llOwnerSay("nick changed sucessfully");
            }
            else if(llSubStringIndex(message, "welcome_msg=")!=-1)
            {
                list arguments = llParseString2List(message,["="],[]);
                welcome_msg=llList2String(arguments,1);
            }
            else if(llSubStringIndex(message, "ocupation=")!=-1)
            {
                list arguments = llParseString2List(message,["="],[]);
                ocupation=llList2String(arguments,1);
            }
            else if(llSubStringIndex(message, "home=")!=-1)
            {
                list arguments = llParseString2List(message,["="],[]);
                home=llList2String(arguments,1);
            }
            else if(message == "SNIFF")
            {
                 Sniff();
            }
            else if(llSubStringIndex(message, "MSG")!=-1)
            {
                 list arguments = llParseString2List(message,[" "],[]);
                 llMessageLinked(LINK_THIS,4,"MSG=" +
                 llList2String(arguments,2) ,llGetKey());
            }
```

```
            else if(llSubStringIndex(message, "CommentMachine")!=-1)
            {
                 list arguments = llParseString2List(message,[" "],[]);
                 integer i;
                 string buf;
                 for(i=1;i<llGetListLength(arguments);i++)
                 {
                     buf+= llList2String(arguments,i) + " ";
                 }
                 llMessageLinked(LINK_THIS,4,"CommentMachine=" +
                 buf,machine_using);
            }
        }
        else if(channel==666)
        {
            if(llSubStringIndex(message, "ADDFRIEND")!=-1)
            {
                list arguments = llParseString2List(message,["="],[]);
                if(AddFriend(llList2String(arguments,1))==TRUE)
                {
                    llSay(666,"FRIENDADDED=" + llList2String(arguments,1));
                    sendBuddiePhoto(llList2String(arguments,1));
                }
                else
                    llSay(666,"FRIENDERROR=" + llList2String(arguments,1));
            }
            else if(message=="SNIFF")
             Sniff();
        }
        else if(channel==667)
        {
            if(message==GIVEBUDDIES)
            {
                buddie_reqs=[];
                sendBuddiesPhoto();
            }
        }
    }
    link_message(integer sender_num, integer num, string str, key id)
    {
```

```
if(num==5)
{
    llOwnerSay("detected-> " + str);
    list users = llCSV2List(str);
    integer i;
    n_dev=llGetListLength(users);
    for(i=0;i<n_dev;i++)
    {
        //create a piconet..and get profiles...
        llOwnerSay("creating piconet");
        llMessageLinked(LINK_THIS,2,"",llList2Key(users,i));
    }
}
else if(num==6)
{
   llOwnerSay("requesting profile");
   //get the users profile....the slave sends its UUID
   llMessageLinked(LINK_THIS,4,GETPROFILE,id);
   llMessageLinked(LINK_THIS,4,GETTAGS,id);
}
else if(num==8)
{
    if(str==GETPROFILE)
    {
      llMessageLinked(LINK_THIS,4,GIVEPROFILE +
      (string)llGetKey() + "=" + nick + "=" + welcome_msg,id);
    }
    else if(llSubStringIndex(str, GIVEPROFILE)!=-1 )
    {
        list arguments = llParseString2List(str,["="],[]);
        // key uuid of owner for photo..
        key req=getPhoto(llList2Key(arguments,1));
        reqs = (reqs=[]) + reqs + [req];
        nicks = (nicks=[]) + nicks + [llList2String(arguments,2)];
        messages = (messages=[]) + messages + [llList2String(arguments,3)];
    }
    if(str==GETTAGS)
    {
      llMessageLinked(LINK_THIS,4,GIVETAGS +
      llList2CSV(tags) + "=" + nick,id);
```

```
            }
            else if(llSubStringIndex(str, GIVETAGS)!=-1 )
            {
                list arguments = llParseString2List(str,["="],[]);
                list user_tags = llCSV2List(llList2String(arguments,1));
                integer i;
                list matches;
                for(i=0;i<llGetListLength(user_tags);i++)
                {
                    if(llListFindList(tags,[llList2String(user_tags,i)])!=-1)
                    {
                        matches = (matches=[]) + matches +
                        [llList2String(user_tags,i)];
                    }
                }
                if(llGetListLength(matches)>=1)
                {
                    llOwnerSay("matches found: " + llList2CSV(matches));
                    llSay(666,"match=" + llList2String(arguments,2));
                }
            }
        }
    }
    http_response(key req,integer stat, list met, string body)
    {
        integer index;
        if((index=llListFindList(reqs,[req]))!=-1)
        {
            integer s1 = 0;
            integer s2 = 0;
            integer s1l= 0;
            integer s2l= -3;
            s1 = llSubStringIndex(body,
            "<img alt=\"profile image\"
            src=\"http://secondlife.com/app/image/");
            s1l = llStringLength("<img alt=\"profile image\"
            src=\"http://secondlife.com/app/image/");
            s2 = llSubStringIndex(body,"\" class=\"parcelimg\" />");

            if(s1 == -1)
```

```
            {
                // selected AV doesn't have a profile picture,
so use the default instead
                ApplyDefaultTexture();
            }
            else
            {
                tex=llGetSubString(body,s1+s1l,s2+s2l);
                llSay(666,SETTEXTURE + tex + "=" +
                llList2String(nicks,index) + "=" +
                llList2String(messages,index));

            }
        }
        else if((index=llListFindList(buddie_reqs,[req]))!=-1)
        {
            integer s1 = 0;
            integer s2 = 0;
            integer s1l= 0;
            integer s2l= -3;
            s1 = llSubStringIndex(body,"<img alt=\"profile image\"
            src=\"http://secondlife.com/app/image/");
            s1l = llStringLength("<img alt=\"profile image\"
            src=\"http://secondlife.com/app/image/");
            s2 = llSubStringIndex(body,"\" class=\"parcelimg\" />");

            if(s1 == -1)
            {
                ApplyDefaultTexture();
            }
            else
            {
                llOwnerSay("sending buddy photo");
                tex=llGetSubString(body,s1+s1l,s2+s2l);
                llSay(667,GIVEBUDDIE + tex + "=" +
                llList2String(friends,index));
            }
        }
    }
}
```

B.2.2 Fixed Point Client

```
// profile data;
string nick ;
string welcome_msg="come in and work out!";
//memory
integer sitting;
string tex;
string text;
list users;
list comments;
string using;
string comment;
//devices  found by the sniff function
integer n_dev=0;
//messages
string GETPROFILE = "givemeprofile";
string GIVEPROFILE = "myprofile=";
string SETTEXTURE = "settexture=";
string GETTAGS = "givemetags=";
string GIVETAGS = "mytags=";
string SETPROFILES = "profiles=";
string HISTORY = "history=";
integer pos=0;
integer DisplayBanner = TRUE;
float Size = 2.50;
float Height = 2.50;
list DefaultTexturePalette = ["7cfd684e-2141-941c-eac8-bd439f0d5a9f",
"02a9cc16-23e5-c355-ca17-a98fc662254f",
"9acff594-c423-40ff-79f1-6a79102cde14"];
ApplyDefaultTexture()
{
    ApplySelectedTexture
    ((key) llList2String(DefaultTexturePalette,
    (integer)llFrand(
    (float)llGetListLength(
    DefaultTexturePalette))));
}
ApplySelectedTexture(key texture)
```

```
{

    llParticleSystem([
        PSYS_PART_FLAGS, 0,
        PSYS_SRC_PATTERN, 4,
        PSYS_PART_START_ALPHA, 0.50,
        PSYS_PART_END_ALPHA, 0.50,
        PSYS_PART_START_COLOR, <1.0,1.0,1.0>,
        PSYS_PART_END_COLOR, <1.0,1.0,1.0>,
        PSYS_PART_START_SCALE, <Size * 1.6 ,Size,0.00>,
        PSYS_PART_END_SCALE, <Size * 1.6,Size,0.00>,
        PSYS_PART_MAX_AGE, 1.20,
        PSYS_SRC_MAX_AGE, 0.00,
        PSYS_SRC_ACCEL, <0.0,0.0,0.0>,
        PSYS_SRC_ANGLE_BEGIN, 0.00,
        PSYS_SRC_ANGLE_END, 0.00,
        PSYS_SRC_BURST_PART_COUNT, 8,
        PSYS_SRC_BURST_RADIUS, Height,
        PSYS_SRC_BURST_RATE, 0.10,
        PSYS_SRC_BURST_SPEED_MIN, 0.00,
        PSYS_SRC_BURST_SPEED_MAX, 0.00,
        PSYS_SRC_OMEGA, <0.00,0.00,0.00>,
        PSYS_SRC_TEXTURE, texture]);

    llSetTexture(texture, ALL_SIDES);

}
getPhoto(key user_key)
{
    string URL_RESIDENT = "http://world.secondlife.com/resident/";
    llHTTPRequest( URL_RESIDENT + (string)user_key,[HTTP_METHOD,"GET"],"");

}
Sniff()
{
    text="";
    pos=0;
    n_dev=0;
    llMessageLinked(LINK_THIS,1,"","");
}
```

```
string getNick(string str)
{
    if(str!="")
    {
        list arguments = llParseString2List(str,["|"],[]);
        return llList2String(arguments,1);
    }
    else return "";


}
key getKey(string str)
{
        if(str!="")
    {
        list arguments = llParseString2List(str,["|"],[]);
        return llList2String(arguments,0);
    }
    else return NULL_KEY;
}
vector target = <1.0,0,0.3>;
rotation targetRot = <0,0,0,0.5>;
default
{
    state_entry()
    {
        llSetText("Free to use",<0.0, 1.0, 0.0>, 1.5);
        llSitTarget( target, targetRot );
        llSay(0, "Hello, Avatar!");
        pos=0;
        n_dev=0;
        nick = (string)llGetKey() + "|Machine 1";
    }
    touch_start(integer total_number)
    {
        llOwnerSay(llList2CSV(users));
        llOwnerSay(llList2CSV(comments));
    }
    changed(integer change)
    { // something changed
```

```
        if (change & CHANGED_LINK) { // and it was a link change
            llSleep(0.5); // llUnSit works better with this delay
            key sitter;
            if ((sitter=llAvatarOnSitTarget()) != NULL_KEY) {
            // somebody is sitting on me
                llOwnerSay("sitting");
                Sniff();
                sitting=TRUE;
            }
            else
            {
                //store data
                llMessageLinked(LINK_THIS,4,"FREE",getKey(using));
                sitting=FALSE;
                comments = (comments=[]) + comments + [comment];
                users = (users=[]) + users + [using];
                using="";
                comment="";
                llSetText("Free to use",<0.0, 1.0, 0.0>, 1.5);
                              //     llStopAnimation("hover");
               // llStopAnimation("sit_to_stand");
            }
        }
    }
    link_message(integer sender_num, integer num, string str, key id)
    {
        if(num==5)
        {
            llOwnerSay("detected-> " + str);
            list users = llCSV2List(str);
            integer i;
            n_dev=llGetListLength(users);
            for(i=0;i<n_dev;i++)
            {
                //create a piconet..and get profiles...
                llOwnerSay("creating piconet");
                llMessageLinked(LINK_THIS,2,"",llList2Key(users,i));
            }
        }
        else if(num==6)
```

```
        {
            llOwnerSay("requesting profile");
            //get the users profile....the slave sends its UUID
           llMessageLinked(LINK_THIS,4,GETPROFILE,id);
        }
        else if(num==8)
        {
            if(llSubStringIndex(str, GIVEPROFILE)!=-1 )
            {
                list arguments = llParseString2List(str,["="],[]);
                // key uuid of owner for photo..
                using=llList2String(arguments,2);
                text+= "Using: \n" + getNick(llList2String(arguments,2)) + "\n";
                llSetText(text, <1,1,1>, 1.5);
                llMessageLinked(LINK_THIS,4,"GIVEMEINFO",getKey(using));
                if(llGetListLength(users)>0)
                    llMessageLinked(LINK_THIS,4,HISTORY + llList2CSV(users)
                    + "=" + llList2CSV(comments),getKey(using));
                else
                    llMessageLinked(LINK_THIS,4,HISTORY + "empty",getKey(using));
            }
            else if(llSubStringIndex(str, "MSG=")!=-1 )
            {
                    list arguments = llParseString2List(str,["="],[]);
            }
            else if(llSubStringIndex(str, "CommentMachine=")!=-1 )
            {
                    list arguments = llParseString2List(str,["="],[]);
                    comment=llList2String(arguments,1);
                    llOwnerSay("Thank you for your feedback: " + comment);
            }
        }
    }
}
```

B.2.3 GUI - Main Prim

```
list menus_title = [ "Sniff", "Friend's List", "Machine"];
list menus=["sniff","friends","machine" ];
integer sub_menu_listener;
string SETTEXTURE = "settexture=";
```

```
string CLEANTEXTURES = "cleantextures";
string COMMENTMACHINE = "CommentMachine=";
string HISTORY = "history=";
string GIVEBUDDIES = "buddies=";
string MENU = "menu=";
string GIVEBUDDIE = "buddie=";
string CLEAN = "clean";
list textures_sniff;
list textures_history;
list textures_friends;
list nicks;
list messages;
list friends;
list match;
list comments;
list history;
integer connect2machine=FALSE;
integer pix = 0;
integer menu_ptr=0;
string getNick(string str)
{
    if(str!="")
    {
        list arguments = llParseString2List(str,["|"],[]);
        return llList2String(arguments,1);
    }
    else return "";
}
key findNick(string name)
{
    integer i;
    for(i=0;i<llGetListLength(nicks);i++)
    {
        if(getNick(llList2String(nicks,i))==name)
            return getKey(llList2String(nicks,i));
    }
    return NULL_KEY;
}
key getKey(string str)
{
```

```
        if(str!="")
    {
        list arguments = llParseString2List(str,["|"],[]);
        return llList2String(arguments,0);
    }
    else return NULL_KEY;
}
default
{
    state_entry()
    {
        llSetText("",<0,0,0>,0);
        llListen(665,"","","");
        menu_ptr=0;
        llMessageLinked(LINK_ALL_CHILDREN,0,"selection=Main Menu","");
        llMessageLinked(LINK_ALL_CHILDREN,0,CLEANTEXTURES,"");
        llSetTexture(llGetInventoryName(INVENTORY_TEXTURE,1),ALL_SIDES);
    }
    touch_start(integer total_number)
    {
        state friend;
    }
    listen(integer channel, string name, key id, string message)
    {
        if(channel==665){

            if(llSubStringIndex(message,MENU)!=-1)
            {
                list arguments = llParseString2List(message,["="],[]);
                integer index;
                if((index=llListFindList(menus,[llList2String(arguments,1)]))!=-1)
                {

                    if(llList2String(arguments,1)==llList2String(menus,0))
                        state sniff;
                    else if(llList2String(arguments,1)==llList2String(menus,1))
                        state friend;
                    else if(llList2String(arguments,1)==llList2String(menus,2))
                    {
                        connect2machine=TRUE;
```

```
                        state machine;
                    }
                }
            }
        }
    }
    link_message(integer sender_number, integer number,
    string message, key id)
    {
        if(message == "menu")
        {
            if (menu_ptr==(llGetListLength(menus)-1))
                menu_ptr=0;
            else
                menu_ptr++;
            if(llList2String(menus,menu_ptr)==llList2String(menus,0))
                state sniff;
            else if(llList2String(menus,menu_ptr)==llList2String(menus,1))
                state friend;
            else if(llList2String(menus,menu_ptr)==llList2String(menus,2))
                state machine;
        }
        else if(message=="home")
            state default;
    }
}
state sniff
{
    state_entry()
    {
          sub_menu_listener=llListen(666,"","","");
          llMessageLinked(LINK_ALL_CHILDREN,0,"selection=Sniff","");
          llSetTexture(llGetInventoryName(INVENTORY_TEXTURE,1),ALL_SIDES);
    }
    touch_start(integer n)
    {
        llSay(666,"SNIFF");
        llSetText("Searching...",<0,0,1>,1);
    }
    link_message(integer sender_number, integer number, string message, key id)
```

```
    {
        if(message == "next")
        {
            if (pix==(llGetListLength(textures_sniff)-1))
                pix=0;
            else
                pix++;
            integer index;
            if((index=llListFindList(friends,[llList2String(nicks,pix)]))==-1)
                llMessageLinked(LINK_ALL_CHILDREN,0,"NOT-FRIEND","");
            else
                llMessageLinked(LINK_ALL_CHILDREN,0,"FRIEND","");
            if(llListFindList(match,[llList2String(nicks,pix)])==-1)
                llMessageLinked(LINK_ALL_CHILDREN,0,"NOT-MATCH","");
            else
                llMessageLinked(LINK_ALL_CHILDREN,0,"MATCH","");
            llSetTexture(llList2Key(textures_sniff,pix),ALL_SIDES);
            llSetText(getNick(llList2String(nicks,pix)) + "\n" +
llList2String(messages,pix) ,<0.4, 0.0, 0.8>,1);
        }
        else if(message == "back")
        {
            if (pix<0)
                pix=(llGetListLength(textures_sniff))-1;
            else
                pix--;
            integer index;
            //conditions check
            if((index=llListFindList(friends,[llList2String(nicks,pix)]))==-1)
                llMessageLinked(LINK_ALL_CHILDREN,0,"NOT-FRIEND","");
            else
                llMessageLinked(LINK_ALL_CHILDREN,0,"FRIEND","");
            if(llListFindList(match,[llList2String(nicks,pix)])==-1)
                llMessageLinked(LINK_ALL_CHILDREN,0,"NOT-MATCH","");
            else
                llMessageLinked(LINK_ALL_CHILDREN,0,"MATCH","");

            llSetTexture(llList2Key(textures_sniff,pix),ALL_SIDES);
            llSetText(getNick(llList2String(nicks,pix)) + "\n" +
            llList2String(messages,pix), <0.4, 0.0, 0.8>,1 );
```

```
        }
        else if(message == "menu")
        {
            if (menu_ptr==(llGetListLength(menus)-1))
                menu_ptr=0;
            else
                menu_ptr++;
            if(llList2String(menus,menu_ptr)==llList2String(menus,0))
                state sniff;
            else if(llList2String(menus,menu_ptr)==llList2String(menus,1))
                state friend;
            else if(llList2String(menus,menu_ptr)==llList2String(menus,2))
                state machine;
        }
        else if(message=="ADDFRIEND")
        {
            llOwnerSay("add: " + llList2String(nicks,pix));
            llSay(666, "ADDFRIEND=" + llList2String(nicks,pix));
            //update in this list....
        }
        else if(message=="home")
            state default;
    }
    listen(integer channel, string name, key id, string message)
    {
        if(channel==666)
        {
            llOwnerSay("listen");
            if(llSubStringIndex(message,SETTEXTURE)!=-1)
            {
                list arguments = llParseString2List(message,["="],[]);
                textures_sniff = (textures_sniff=[]) + textures_sniff +
                [llList2Key(arguments,1)];
                nicks = (nicks=[]) + nicks + [llList2String(arguments,2)];
                messages = (messages=[]) + messages +
                [llList2String(arguments,3)];
                llSetTexture(llList2Key(textures_sniff,0),ALL_SIDES);
                llSetText(getNick(llList2String(nicks,0)) + "\n" +
                llList2String(messages,0),<0.4, 0.0, 0.8>,1 );
                if(llListFindList(friends,[llList2String(nicks,0)])==-1)
```

```
                llMessageLinked(LINK_ALL_CHILDREN,0,"NOT-FRIEND","");
            else
                llMessageLinked(LINK_ALL_CHILDREN,0,"FRIEND","");

            if(llListFindList(match,[llList2String(nicks,0)])==-1)
                llMessageLinked(LINK_ALL_CHILDREN,0,"NOT-MATCH","");
            else
                llMessageLinked(LINK_ALL_CHILDREN,0,"MATCH","");
        }
        else if(message==CLEANTEXTURES)
        {
            textures_sniff=[];
            nicks=[];
            messages=[];
            match=[];
            pix=0;
        }
        else if(llSubStringIndex(message,"FRIENDADDED")!=-1)
        {
            list arguments = llParseString2List(message,["="],[]);
            friends = (friends=[]) + friends +
            [llList2String(arguments,1)];
        }
        else if(llSubStringIndex(message,"FRIENDERROR")!=-1)
        {
            //
        }
        else if(llSubStringIndex(message,"match=")!=-1)
        {
            list arguments = llParseString2List(message,["="],[]);
            match = (match=[]) + match + [llList2String(arguments,1)];
            llOwnerSay("match: " + llList2String(arguments,1));
        }
    }
}
state_exit()
{
    llListenRemove(sub_menu_listener);
    llMessageLinked(LINK_ALL_CHILDREN,0,CLEAN,"");
    llSetText("",<0.4, 0.0, 0.8>,0);
```

```
    }
}
state friend
{
    state_entry()
    {
        llSetTexture(llGetInventoryName(INVENTORY_TEXTURE,1),ALL_SIDES);
        llSetText("Searching Friend's List...",<0,0,1>,1);
        llSay(667,GIVEBUDDIES);
        sub_menu_listener=llListen(667,"","","");
        pix=0;
        friends=[];
        textures_friends=[];
        llMessageLinked(LINK_ALL_CHILDREN,0,"selection=Friend's List","");
    }
    listen(integer channel, string name, key id, string message)
    {
        if(llSubStringIndex(message,GIVEBUDDIE)!=-1)
        {
            list arguments = llParseString2List(message,["="],[]);
            textures_friends = (textures_friends=[]) + textures_friends +
            [llList2Key(arguments,1)];
            friends = (friends=[]) + friends + [llList2String(arguments,2)];
            llSetTexture(llList2Key(textures_friends,0),ALL_SIDES);
            llSetText(getNick(llList2String(friends,0)),<0.4, 0.0, 0.8>,1 );
            llOwnerSay("friend added: "+  llList2String(arguments,2));
        }

        else if(llSubStringIndex(message,"FRIENDADDED")!=-1)
        {
           // list arguments = llParseString2List(message,["="],[]);
            //friends = (friends=[]) + friends + [llList2String(arguments,1)];
        }
        else if(llSubStringIndex(message,"FRIENDERROR")!=-1)
        {
            //
        }
    }
    link_message(integer sender_number, integer number, string message, key id)
    {
```

```
        if(message == "next")
        {
            if (pix==(llGetListLength(textures_friends)-1))
                pix=0;
            else
                pix++;

            llSetTexture(llList2Key(textures_friends,pix),ALL_SIDES);
            llSetText(getNick(llList2String(friends,pix)),<0.4, 0.0, 0.8>,1);
        }

        else if(message == "back")
        {
            if (pix<0)
                pix=(llGetListLength(textures_friends))-1;
            else
                pix--;

            llSetTexture(llList2Key(textures_friends,pix),ALL_SIDES);
            llSetText(getNick(llList2String(friends,pix)), <0.4, 0.0, 0.8>,1 );
        }
        else if(message == "menu")
        {
            if (menu_ptr==(llGetListLength(menus)-1))
                menu_ptr=0;
            else
                menu_ptr++;
            if(llList2String(menus,menu_ptr)==llList2String(menus,0))
                state sniff;
            else if(llList2String(menus,menu_ptr)==llList2String(menus,1))
                state friend;
            else if(llList2String(menus,menu_ptr)==llList2String(menus,2))
                state machine;
        }
        else if(message=="home")
            state default;
    }
    state_exit()
    {
        llListenRemove(sub_menu_listener);
```

```
        llMessageLinked(LINK_ALL_CHILDREN,0,CLEAN,"");
        llSetText("",<0.4, 0.0, 0.8>,0);
    }
}
state machine
{
    state_entry()
    {
        sub_menu_listener=llListen(666,"","","");
        llSetText("", <0.4, 0.0, 0.8>,0);
        //history=[];
        //textures_history=[];
        //comments=[];
        llMessageLinked(LINK_ALL_CHILDREN,0,"selection=Conected to Machine","");

        if(connect2machine==FALSE)
        {
            llSetText("Not Connected to a Machine",<1,0,0>,1);
            llSetTexture(llGetInventoryName(INVENTORY_TEXTURE, 0),ALL_SIDES);
        }
        else
        {
            llSetText("Connecting...",<0,1,0>,1);
            llSetTexture(llGetInventoryName(INVENTORY_TEXTURE,1),ALL_SIDES);
        }
    }
    touch_start(integer n)
    {
        llOwnerSay(llList2CSV(textures_history));
        llOwnerSay(llList2CSV(history));
        llOwnerSay(llList2CSV(comments));
    }
    listen(integer channel, string name, key id, string message)
    {
        if(llSubStringIndex(message, HISTORY)!=-1)
        {
            list arguments = llParseString2List(message,["="],[]);
            if(llList2String(arguments,1)=="empty")
            {
                llSetText("Machine Has No Data",<1,0,0>,1);
```

```
                llSetTexture(llGetInventoryName(INVENTORY_TEXTURE, 0),ALL_SIDES);
            }
            else
            {
                    textures_history = (textures_history=[]) + textures_history +
                    [llList2Key(arguments,1)];
                    history = (history=[]) + history + [llList2String(arguments,2)];
                    if(llGetListLength(arguments)>=4)
                        comments = (comments=[]) + comments +
                        [llList2String(arguments,3)];
                    else
                        comments = (comments=[]) + comments + [" "];
                    llSetTexture(llList2Key(textures_history,0),ALL_SIDES);
                    string text = getNick(llList2String(history,0));
                    if(llList2String(comments,0)!=" ")
                        text+= "\n" + llList2String(comments,0);
                    llSetText(text,<0.4, 0.0, 0.8>,1);
            }
        }
        else if(message=="FREE")
        {
            llSetText("", <0.4, 0.0, 0.8>,0);
            history=[];
            textures_history=[];
            comments=[];
            pix=0;
            connect2machine=FALSE;
            state default;
        }
    }
    link_message(integer sender_number, integer number,
    string message, key id)
    {
        if(message == "next")
        {
            if (pix==(llGetListLength(textures_history)-1))
                pix=0;
            else
                pix++;
```

```
            llSetTexture(llList2Key(textures_history,pix),ALL_SIDES);
            string text = getNick(llList2String(history,pix));
            if(llList2String(comments,pix)!=" ")
                text+= "\n" + llList2String(comments,pix);
            llSetText(text,<0.4, 0.0, 0.8>,1);
        }
        else if(message == "back")
        {
            if (pix<0)
                pix=(llGetListLength(textures_history))-1;
            else
                pix--;

            llSetTexture(llList2Key(textures_history,pix),ALL_SIDES);
            string text = getNick(llList2String(history,pix));
            if(llList2String(comments,pix)!=" ")
                text+= "\n" + llList2String(comments,pix);
            llSetText(text,<0.4, 0.0, 0.8>,1);
        }
        else if(message == "menu")
        {
            if (menu_ptr==(llGetListLength(menus)-1))
                menu_ptr=0;
            else
                menu_ptr++;
            if(llList2String(menus,menu_ptr)==llList2String(menus,0))
                state sniff;
            else if(llList2String(menus,menu_ptr)==llList2String(menus,1))
                state friend;
            else if(llList2String(menus,menu_ptr)==llList2String(menus,2))
                state machine;
        }
        else if(message=="home")
            state default;
    }
    state_exit()
    {
        llListenRemove(sub_menu_listener);
        llSetText("",<0.4, 0.0, 0.8>,0);
    }}
```

References

[1] Mark Weiser. Hot topics: Ubiquitous computing. *IEEE Computer*, October 1993.

[2] Mark Weiser. The world is not a desktop. *Interactions*, pages 7–8, January 1994.

[3] Alan Kay. Computers, networks, and education. *Scientific American*, pages 138–148, September 1991.

[4] Adam Greenfield. *Everyware*. March 2006.

[5] Roy Want. Carry small, live large! *Pervasive Computing, IEEE*, 6(3):2–4, July-Sept. 2007.

[6] T.; Fox Kindberg. System software for ubiquitous computing. *Pervasive Computing, IEEE*, 1(1):70–81, Jan.-March 2002.

[7] Jennifer Thom-Santelli. Mobile social software: Facilitating serendipity or encouraging homogeneity? *Pervasive Computing, IEEE*, 6(3):46–51, July-Sept. 2007.

[8] C. Edwards. Another world. *Engineering and Technology*, 1(9):28–32, Dec. 2006.

[9] Blom J. Persson, P. and Y Jung. Digidress: A field trial of an expressive social proximity application. *UbiComp*, LNCS 3660:195–212, 2005.

[10] Blom J. Jung, Y. and P. Persson. Scent field trial - understanding emerging social interaction. *8th International Conference on Human Computer Interaction with Mobile Devices and Services (Mobile HCI06)*, Sep 12-15 2006.

[11] http://www.dodgeball.com/home.

[12] K.K.; Martinez M.L.B.; Hussell J.A.T.; Luce R.E.; Collins, L.M.; Mane. Sciencesifter: facilitating activity awareness in collaborative research groups through focused information feeds. *e-Science and Grid Computing, First International Conference on*, page 8 pp., Dec. 2005.

[13] R.P.; Schreckenghost, D.; Bonasso. Multi-agent system for managing human activities in space operations. *Aerospace Conference, IEEE*, page 8 pp., 4-11 March 2006.

[14] Min Kyung Kim; Hee-Cheol Kim;. Awareness and privacy in groupware systems. *Computer Supported Cooperative Work in Design, 10th International Conference on*, pages 1–5, May 2006.

[15] M.; Riener A.; Schmitzberger H.; Franz M.; dos Santos Rocha M. Ferscha, A.; Hechinger and A Zeidler. Context-aware profiles. *International Conference on Autonomic and Autonomous System*, July 2006.

[16] B.; Chen Y.; Jana R.; Huang H.; Begeja L.; Gibbon D.; Liu Z.; Wei, B.; Renger and B. Shahraray. Mediaalert - a broadcast video monitoring and alerting system for mobile users. *International Conference On Mobile Systems, Applications And Services*, pages 25 – 38, 2005.

[17] G. Kortuem and Z. Segall. Wearable communities: augmenting social networks with wearable computers. *Pervasive Computing, IEEE*, 2(1):71– 78, Jan-Mar 2003.

[18] F. C. ; Kasten O.; Fiege, L.; Grtner and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. *th ACM/IFIP/USENIX International Middleware Conference*, page 103122, June 2003.

[19] Beno Eugster, Patrick Th.; Garbinato and Adrian. Holzer. Location-based publish/subscribe. *UNIL Technical Report*, pages 279–282, July 2005.

[20] Y. Koji, T.; Michiaki. Ubi-finger: Gesture input device for mobile use. *Transactions of Information Processing Society of Japan*, 43(12):3675–3684, 2002.

[21] Jun-Seok; Kyung, Ki-Uk; Park. Ubi-pen: Development of a compact tactile display module and its application to a haptic stylus. *EuroHaptics Conference, 2007 and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, pages 109 – 114, 22-24 March 2007.

[22] B.; O'Gorman-D.; Moyna N.; Diamond D.; Brady, S.; Carson. Combining wireless with wearable technology for the development of on-body networks. *Wearable and Implantable Body Sensor Networks, International Workshop on*, 3-5 April.

[23] A.; Nakajima-T.; Hanaoka, K.; Takagi. A software infrastructure for wearable sensor networks. *Embedded and Real-Time Computing Systems and Applications, 12th IEEE International Conference on*, pages 27 – 35, 2006.

[24] J.J.; Barton and V. Vijayaraghavan. Ubiwise, a simulator for ubiquitous computing systems design. *Technical Report HPL-2003-93*, 2003.

[25] M. Bylund and Espinoza F. Using quake iii arena to simulate sensors and actuators when evaluating and testing mobile services. *Conference on Human Factors in Computing Systems*, pages 241 – 242, 2001.

[26] M.; Lewis-D.; O'Donnell T.; O'Sullivan D. O'Neill, E.; Klepal and D. Pesch. A testbed for evaluating human interaction with ubiquitous computing environments. *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 60 – 69, 2005.

[27] S.; Tamai-M.; Nishigaki K.; Kitani T.; Shibata N.; Yasumoto K.; Nishikawa, H.; Yamamoto and M. Ito. Ubireal: Realistic smartspace simulator for systematic testing. *Ubicomp*, 2006.

[28] N.; Morla, R.; Davies. Evaluating a location-based application: a hybrid test and simulation environment. *Pervasive Computing, IEEE*, 3(3):48 – 56, July-Sept. 2004.

[29] M.E.; Disz, T.; Papka. Ubiworld: an environment integrating virtual reality, supercomputing, and design. *HCW '97*, pages 46–57, April 1997.

[30] G.; Sundresh, S.; Wooyoung Kim; Agha. Sens: a sensor, environment and network simulator. *Simulation Symposium, Proceedings. 37th Annual*, pages 221 – 228, April 2004.

[31] P. Levis and N. Lee. Tossim: A simulator for tinyos networks. Spt. 2003.

[32] A. Park, S.; Savvides and M. B. Srivastava. Sensorsim: a simulation framework for sensor networks. *Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104 – 111, 2000.

[33] M.; Ahuja R.; Tang B.; Bagrodia R. Bajaj, L.; Takai and M. Gerla. Glomosim: A scalable network simulation environment.

[34] O. Michel. Webotstm: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):39–42, March 2004.

[35] H.; Speck, A.; Klaeren. Robosim: Java 3d robot visualization. *Industrial Electronics Society, 1999. IECON '99 Proceedings. The 25th Annual Conference of the IEEE*, 2:821 – 826, 29 Nov.-3 Dec 1999.

[36] http://steam.cs.ohiou.edu/vitalwiki/index.php/Second_Life_Development.

[37] http://secondlife.wikia.com/wiki/Havok.

[38] http://opensimulator.org/wiki/Main_Page.

[39] http://www.mono-project.com/Main_Page.

[40] A.; Winograd T.; Johanson, B.; Fox. The event heap: An enabling infrastructure for interactive workspaces. 1999.

[41] A.; Winograd T.; Johanson, B.; Fox. The interactive workspaces project: experiences with ubiquitous computing rooms. *Pervasive Computing, IEEE*, 1(2):67 – 74, April-June 2002.

[42] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[43] http://wiki.secondlife.com/wiki/LSL_Portal.

[44] http://www.lslwiki.net.

[45] N. Eagle and A. Pentland. Social serendipity: Mobilizing social software. *IEEE Pervasive Computing, Special Issue: The Smart Phone*, pages 28–34, April-June 2005.

[46] Vinny Reynolds, Vinny; Cahill and Aline Senart. Requirements for an ubiquitous computing simulation and emulation environment. *ACM International Conference Proceeding Series*, 138, 2006.

[47] P. Persson and Y. Jung. Nokia sensor: From research to product. *Conference on Designing for User eXperience (DUX)*, Nov 3-5 2005.

Errata

Errata for the first printed version of "**Building Mobile and Pervasive Applications in Second Life**™", thesis submitted on June 30 of 2008, by Pedro Ricardo Ferreira Teixeira de Jesus Leite - 020503200.

Ordered by appearance in the thesis, and updated on July 6, 2008.

Front Page, entire page.
Front page translated in English and added "**Provisional Version**"to the cover.

Front Page (back), Header.
Unknown error number ("**-2**") on the top left corner deleted.

Page xviii, Header.
Incorrect section name ("**List of Tables**") on the Glossary deleted.

Page 3, last paragraph of section 1.4.
Changed "**The two appendix sections present the important sections of code developed, regarind the two simulations**"to "**Lastly, two apendix sections present the most relevant sections of code developed, regarding the two simulations.**".

Page 9, second paragraph from bottom.
Changed "**The simulator The simulator**"to "**The simulator**".

Page 25, second paragraph of section 3.4.
Unmarked reference. Changed "**XXX**"to "**3.2.1**".

Page 46, Table 4.3.
Aligned table.

Page 51, Table 4.6.
Aligned table.

Page 55, Table 4.7.
Aligned table.

Page 55, Table 4.8.
Aligned table and deleted "**Test Results compared with Opensim Simulation**" unnecessary cell.

Pages 77 - 91, Sections A.1.1 and A.1.2.
Code layout changed to more appealing form.

Page 94, Section A.1.2.3.
Deleted the section related to the "**PHP Off-World Server**".

Page 99, Section A.1.2.4.
Deleted the section related to the "**SQL Database - Off-World Server**".

Pages 107 - 109, Sections A.2.3 and A.2.4.
Code layout changed to more appealing form.