TATUS

A Ubiquitous Computing Simulator

Eleanor O'Neill

A dissertation submitted to the University of Dublin, Trinity College

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

September 2004

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Eleanor O'Neill

Dated: September 13, 2004

Permission to Lend and/or Copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Eleanor O'Neill

Dated: September 13, 2004

Acknowledgements

Many thanks are due to my supervisor, Declan O'Sullivan, for his invaluable advice and for the benefit of his experience throughout the course of the project. Also, thank you to Dave Lewis who acted in earnest as a second supervisor, providing many constructive ideas towards the final design. Finally, thanks to Tony O'Donnell for the time and feedback he provided when testing the simulator.

To my family and friends, thanks for your continued support and encouragement. To my NDS classmates, thanks for making this a year to remember.

Eleanor O'Neill

University of Dublin, Trinity College

September 2004

Abstract

"... we are trying to conceive a new way of thinking about computers in the world, one that takes into account the natural human environment and allows the computers themselves to vanish into the background" - Mark Weiser, 1991 [1].

Weiser was ahead of his time with this visionary idea of a ubiquitous computing environment that would enhance and improve daily life. His ideas involve a paradigm shift away from the constraints of the one-on-one personal computer situation which is now a common part of daily life. Although technology has not reached Weiser's adventurous predictions, progress to advance state-of-the-art in ubiquitous computing is under-way.

Furthermore, current research projects within the Knowledge and Data Engineering Group (KDEG) closely resemble some of Weiser's early visions. Software is being developed here to implement intelligent environments. Inside these smart worlds, doors open automatically but only to authorized persons, rooms identify people on entry and the environment as a whole works to recognise the behavioural patterns and intentions of those living and working in the space. Universal progress on the development of ubiquitous computing technologies has been hindered by a commonly recurring set of problems involving cost and logistics when implementing suitable test environments.

This dissertation describes TATUS, a ubiquitous computing simulator aimed as overcoming these cost and logistical issues. Based on a 3D games engine, the simulator has been designed to maximise usability and flexibility while minimising working knowledge of the game engine.

Contents

List of Figures

Contents

Chapter 1

Introduction

This dissertation describes a simulator called TATUS that has been developed to support researchers writing software to control ubiquitous computing environments. A peripheral piece of software-under-test (SUT) connected to the simulator assimilates exported state in order to develop its own representation of the world. Based on the view the SUT holds of the environment, it makes decisions to change the world in reaction to user movements and behaviour. The overall effect is to allow the SUT to control the TATUS virtual environment such that the environment behaves intelligently according to the experimental goals of the SUT. The graphics and network connection features of a game engine have been exploited to support the core simulator.

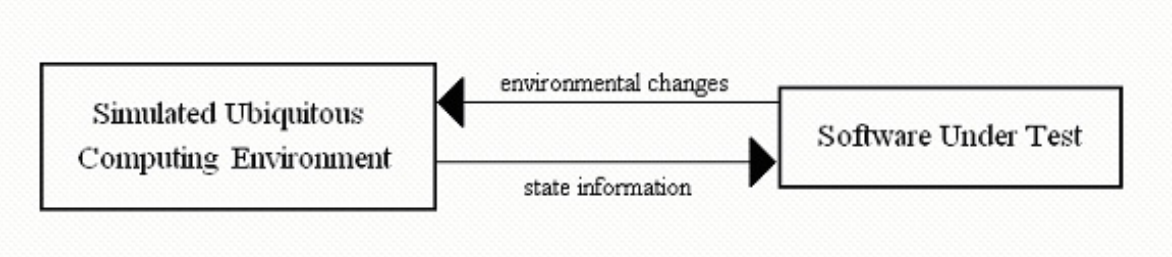

Figure 1.1 High-level simulator overview.

1.1 Project Motivation

When developing devices, protocols and software to support a ubiquitous computing environment, the common problem set encountered involves cost, logistics and location. Acquisition of adequate resources, such as embedded and hand-held devices, often prohibits experimentation by exceeding the project budget. In the case that resources are freely available, the logistics of managing large scale projects and instructing a test user community are complex and time-consuming.

In addition, most ubiquitous computing applications require testing and experimentation in their target location for valid and beneficial results to be produced. Through careful device positioning, a standard office can be adapted to resemble another environment such as a living-room, however it is difficult to simulate large or specialised locations, for example an airport.

In response to these issues, TATUS provides a virtual ubiquitous computing environment. Cost is no longer a problem because all TATUS ubiquitous computing environments are assumed to be fully equipped and configured. Complex logistics, although not eliminated are alleviated because the test-arena is configured and monitored from a single desktop machine. Finally the virtual world is built in accordance with experimental requirements which means a suitable location is always available.

Currently within the Knowledge and Data Engineering Group (KDEG), there are a number of researchers developing software intended to control ubiquitous computing environments. The software is being designed with a view to providing the intelligence that supports smart environments. The simulator provides a test-bed to execute experiments that test the logic and rules implemented within the SUT code. Prior to this, the research group has lacked a ubiquitous computing test-bed, hindering developers as they try to validate their work.

The envisaged benefits of the simulator include:

- Provision of a research test-bed to experiment with SUT.
- Experimentation without the cost and logistical issues presented by physical installations.
- Flexible experiments because multi-player game settings allow multiple researchers to

interact in a single experiment while non-player-character (NPC) Artificial Intelligence (AI) allows a single researcher to work independently.

- Replication of experiments using saved settings.
- Experiment review by rerunning recorded and logged experiments.

1.2 Project Objectives

The objective of the project was to develop a 3D simulator to satisfy the following objectives:

1. Allow researchers to connect software-under-test (SUT) to the simulator.
2. Allow researchers to select simulator events of interest for notification to the SUT.
3. Provide researchers with an instruction protocol allowing the SUT to control actions within the simulator.
4. The simulator must be flexible to handle diversity in research projects and cope with multiple SUT connections in parallel.
5. The simulator must be usable. In particular, this means a straightforward intial setup procedure and an easy mechanism to configure and run tests.
6. The simulator must realistically reflect the current state of ubiquitous computing technology e.g. pressure sensors and RFID (Radio Frequency Identification) tags. This version of the simulator is not aimed at developing new device technology, communication protocols or wireless media.

1.3 Dissertation Structure

Chapter 2: State-of-the-Art Chapter 2 notes diversity in ubiquitous computing testbeds. The chapter specifically focuses on UbiWise [2], a 3D ubiquitous computing simulator that is well recognised. The second section of the chapter provides the reasoning behind supporting TATUS with a game engine. This section also covers the process of selecting an appropriate game. The final section of the chapter introduces Half-Life, a first-person-shooter (FPS) game.

Chapter 3: Design Although from the outset, there were clear high-level goals for the final deliverable, the requirements for the project were drawn-up based on meetings with potential users. The first part of this chapter presents those requirements. The second part of the chapter discusses the TATUS design.

Chapter 4: Implementation Chapter 4 explains how components of the system are implemented by modifying the game engine. The chapter first presents the relationships between system components followed by the implementational details behind each system element.

Chapter 5: Results, Evaluation & Discussion Chapter 5 presents results, evaluation and discussion of TATUS. Results are divided into two sections according to two separate simulator prototypes. The section titled evaluation is a theoretical comparison between the final implementation and the original objectives. Finally the chapter finishes with a discussion that compares TATUS to its counterpart UbiWise.

Chapter 6: Further Work & Conclusions Chapter 6 presents ideas for future development of the simulator followed by conclusions about the success of its design and implementation.

Chapter 2

State of the Art

The following chapter is divided into two sections. The first section discusses some existing technology that has been implemented to test and evaluate ubiquitous computing environments. The second section of the chapter focuses on the 3D game engine which is used to render the virtual ubiquitous computing environment inside TATUS.

2.1 Ubiquitous Computing Simulators

In the paper "User Study Techniques in the Design and Evaluation of a Ubicomp Environment" [3], Consolvo and Arnstein make the claim that techniques for evaluating and assessing ubiquitous computing environments have not yet been well-established. A variety of practices are currently used to test ubiquitous computing environments, with many research groups developing test arenas specifically tailored towards their own experiments. The following section presents research projects currently active along with the methods of testing and experimentation employed to evaluate their work.

The Sentient Computing Project [4] is moving away from the conventional view that human-computer interaction is all premeditated and involves explicit deliberate actions with a computer interface. The project is working to develop applications that can model a true representation of the world so that a person's natural surroundings become in essence a user interface. The natural movements and gestures of people occupying the space become the input commands to the application controlling the environment.

The Sentient Computing Project has set up a physical test-environment in the Engineering

Figure 2.1 The Sentient test environment.

Department at Cambridge University. Figure 2.1 shows the real-world environment with the graphical representation drawn up by the application alongside. The application received input from sensors and actuators embedded in the room. The red colour applied to the phone in right-hand image of Figure 2.1 demonstrates the application's ability to recognise when the phone is use.

Earlier this year Ricardo Morla and Nigel Davies evaluated a location-based ubiquitous computing application using a hybrid test environment [5]. A remote medical monitoring system was implemented as the test application. Using existing network and context simulators the team simulated the potential conditions that occur in a user's home e.g. temporary disconnection from the network. From this they were able to verify that the application does perform reliably under target conditions.

TCD's KDEG research group is currently researching applications similar in theory to those being developed by the Sentient Computing Project at Cambridge. However, unlike Sentient, KDEG does not have a suitable location to setup a similar physical test environment. Neither is the approach adopted by Morla and Davies using network and context simulators suitable to satisfactorily test and evaluate the software being developed.

The ideal solution for KDEG researchers is a high quality 3D interactive ubiquitous computing simulator. Research has shown that only one such simulator, UbiWise [2], has

successfully been developed to date. Its design is discussed in the following section and an explanation is presented as to why the simulator is unsuitable for use as a test-bed for KDEG research.

2.1.1 UbiWise

UbiWise targets the development and testing of hardware and low-level software for ubiquitous computing devices. The mutual dependency between developing these two technologies had been hindering real-world development of these types of devices. The UbiWise project aimed to simulate the existence of devices in order to develop the systems that would run on them. UbiWise emerged from an amalgamation of two existing simulators, WISE and UbiSim.

2.1.2 UbiSim

Initially developed as QuakeSim the early goals were to produce context information in real-time using a semi-realistic environment. Its overall goal was to fill the void for testing and demonstrating context aware services without the high overhead of installing real physical devices and operators of those devices. QuakeSim worked by taking raw simulated data outputted from the Quake III Arena (Q3A) [26] gaming environment and processing it in the Context Toolkit. The context server was also capable of inserting data produced by real-world sensors and using the result to deliver meaningful context to applications and services.

When Barton and Vijayaraghavan took over QuakeSim from Bylund and Espinoza they extended it to become UbiSim. The most obvious changes they made allowed wireless devices to be displayed rather than weapons for use by the user. So the game Q3A was further tailored towards the world of ubiquitous computing.

2.1.3 WISE

WISE is not a ubiquitous computing simulator in its own right. The initial motivation behind it was to provide an arena for demonstrating protocols at the application layer without full-scale development and deployment. On a grander scale it was hoped that WISE would encourage invention and development of services specifically targeting digital devices connecting directly to the Internet e.g. wireless cameras or PDAs. The results produced from

WISE aimed to guide developers in creating the interfaces and protocols required by such devices in the face of problems such as latency and connectivity issues.

The Wise environment consists of a 2D world or interface which displays an image of a simulated device. The user manipulates the device to communicate with real-world Internet services or other simulated devices. The underlying supporting software is a Web Client that connects to the desired Internet services and interacts with the outside world via the HTTP protocol.

2.1.4 UbiWise: UbiSim + WISE

UbiWise makes use of the graphical interfaces provided by each tool. UbiSim provides the 3D model of the simulated world that the user can navigate around in the first person manner as is normally done when playing a first-person shooter game. This is called the physical environment view. In this view, the user's current weapon becomes the user's current wireless device.

The second view provided by the system is called the Device-Interaction View. This is a Java window as part of the WISE system where screen areas are mapped to particular device buttons so the device can be controlled by a mouse and keyboard.

UbiWise offers three usage roles. The first role, the user, interacts with the simulated environment when running an experiment or playing out a scenario. This involves navigating around the 3D world using the game controls or using the mouse and keyboard to interact with the WISE interface.

The second role is that of a researcher where the user adjusts the simulated environment, pre-run time, setting up the world to suit a particular scenario. Generally when using this type of simulator it is necessary to consider in advance the actions that will be carried out. For example a meeting requires a conference room with appropriate facilities e.g. large table but to execute a lecture there must be projection facilities in the room.

The third role is that of a developer, it is the most technical role and is filled by anyone extending the simulator to improve the ubiquitous computing environment. This includes incorporation of new devices and wireless media. As a developer the user understands the background processes to the UbiWise system.

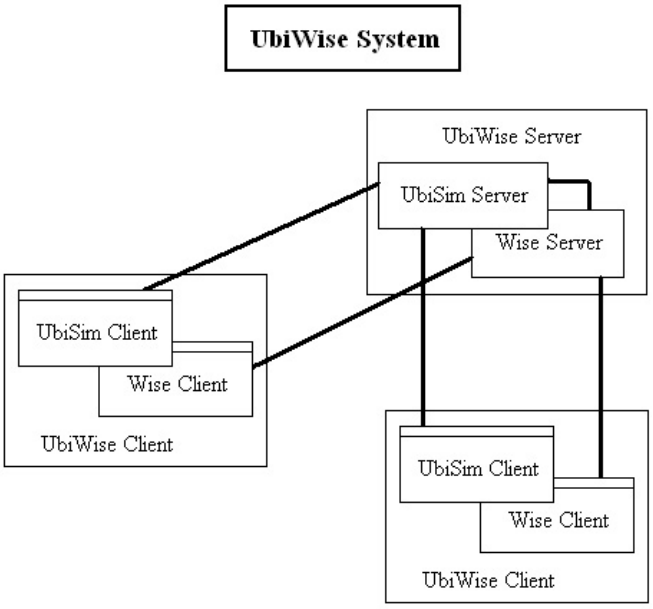

Figure 2.2 Client-Server layout of WISE and UbiSim.

2.1.5 UbiWise vs. KDEG Requirements

Based on the project objectives presented in chapter 1, UbiWise will be analysed in terms of its suitability for KDEG research. The original objective is noted in bold at the start of each item on the list below.

1. **Allow researchers to connect software-under-test (SUT) to the simulator.** Applications or software tested using the UbiWise simulator must first be developed inside the UbiWise tool. Experimentation with external software is not supported.

2. **Allow researchers to select simulator events of interest for notification to the SUT.** From the information presented in the UbiWise paper there is no reason to believe this feature is supported. Further UbiWise targets testing and development of hardware and low-level software. KDEG research targets development of high-level intelligent software capable of controlling smart environments based on the input it receives from embedded sensors and actuators.

3. **Provide researchers with an instruction protocol allowing the SUT to control**

actions within the simulator. From the information presented against requirement 1, UbiWise does not support testing of external software and as such no instruction protocol is in place for use by the SUT.

4. **The simulator must be flexible to handle diversity in research projects and cope with multiple SUT connections in parallel.** UbiWise is tailored very specifically towards testing devices; the tool's flexibility focuses on device design and configuration.

5. **The simulator must be usable. In particular, this means a straightforward intial setup procedure and an easy mechanism to configure and run tests.** Since UbiWise does not satisfy objective 1, it cannot meet objective 5.

6. **The simulator must realistically reflect the current state of ubiquitous computing technology e.g. pressure sensors and RFID (Radio Frequency Identification) tags. This version of the simulator is not aimed at developing new device technology, communication protocols or wireless media.** UbiWise aims to develop and advance device technology. This goal is completely orthogonal to the objective 6. From requirement 2, the events generated in such an environment cannot be guaranteed to exactly reflect the current state of ubiquitous computing technology.

In chapter 5 a full comparison of UbiWise and TATUS is presented as part of the projects evaluation.

2.2 3D FPS Games

Many of the 3D first-person-shooter network games released for PCs since the late 1990's have also released SDKs which allow programmers to modify the game through the inclusion of new rules, physics, weapons and characters. The term mod is appropriately used to refer to the games resulting from such adjustments.

The general aim in choosing to use one of these games is to exploit the 3D graphics engine while mapping the projects requirements into the SDK code to provide a virtual 3D ubiquitous computing environment as a test-bed for researchers. In addition the LAN style implementation of these games provides potential for multiple researchers to interact

in a single experiment. Finally, the SDK also provides limited AI and scripted sequences to include non-player-characters (NPCs) allowing a single researcher to run tests independently.

When choosing the game for this project the potential candidates were Half-Life [17], Quake III Arena [**?**, qke]nd Unreal Tournament [29]. Tables 2.1, table 2.2 and table 2.3 show profiles for Half-Life, Quake III Arena and Unreal Tournament respectively. These games are designed on the same basic principles and in fact Half-Life is derived from Quake with about 30% of the original code remaining at its core. Research for these games has determined that Quake III Arena is less mod-friendly than the original Quake release. Mods such as Quake III Rally [28] have published this as the reason for discontinuing development work on their mods.

Each of the games would have sufficiently met the project requirements, however Half-Life outshone the others for two main reasons.

- Firstly, the Half-Life SDK [18] uses a C/C++ combination accommodating object-oriented modelling and also providing the most accessible implementation language of the three. Quake uses an adaptation of C called QC, this does not allow object-oriented style programming, while Unreal Tournament uses an object-oriented language called UnrealScript.

- Secondly, although Half-Life has not officially been documented by its creators, Valve [17], the tutorials, articles, forums and advice available online is far superior in content and comprehensibility than the equivalent texts for either of the other games. For the most part this was made possible by Botman [21] a programmer who put huge effort into exploring and documenting the SDK code when the first release was made available.

2.3 Introduction to Half-Life

Half-Life (HL) uses a client-server architecture allowing up to 32 players to compete in a single game. The server machine runs either in dedicated-mode, full 32 clients, or player-mode, 15 clients (16 players) in total. Each client has enough built-in artificial intelligence to estimate player movements in the case of lost messages from the server, correcting to the true picture of the world when contact is re-established.

Half-Life	
Developer	Valve Corporation
Distributor	Sierra Entertainment, Inc [20]
Release Date	31 October 1998
Platform	PC only, game modification requires C compiler
Mod development	Valve HL SDK v 2.3
Editor Microsoft	Visual C v 6.0 (as used by Valve)
Map Editor	Valve Hammer Editor [19]
Language	C/C++ inc object-oriented modeling

Table 2.1 Profile for Half-Life.

Quake III Arena	
Developer	id Software
Distributor	id Software
Release Date	December 1999
Platform	PC/MAC
Mod development	Quake III SDK [27]
Editor	Microsoft Visual C
Map Editor	Q3Radient [27]
Language	QC - A Quake adaptation of C.

Table 2.2 Profile for Quake III Arena.

Unreal Tournament	
Developer	Epic Games [29]
Distributor	Atari [30]
Release Date	November 1999
Platform	PC/MAC
Mod development	UnrealScript source files, UnrealED 3.0
Editor	UnrealED 3.0
Map Editor	UnrealED 3.0
Language	UnrealScript featuring object-oriented modeling

Table 2.3 Profile for Unreal Tournament.

A *total-conversion* of HL involves creating new maps, weapons, characters, physics and rules. The simplest method to modify the game uses map creation alone. To adjust or add new instances of the remaining items, the HL SDK must be reprogrammed. Both techniques are discussed in the following section.

2.3.1 Map Creation

New maps or levels can provide the illusion of an entirely new game. Every world is a combination of basic shapes but through careful application of textures the map's terrain can be varied dramatically. Valve released a map editor called Hammer which is a drawing tool for building maps. Hammer saves maps in the BSP (Binary Space Partitioning) format used by Half-Life.

BSP files have been designed to improve game play by minimising the calculations involved at run-time when drawing the environment. The BSP file saves the topology of a map as a binary tree. Objects that are geographically close in the map are stored in neighbouring nodes within the tree. In addition to the BSP file, Hammer produces a MAP file which provides a textual representation of the world. It lists information about objects in the world such as their name, type, size and coordinates.

Due to the size of images for this section, all screenshots are stored in Appendix G. A view of Hammer can be seen in Figure 6.6.

2.3.1.1 Brushes & Entities

Figure 6.7 shows a map under development in Hammer. Brushes are the three dimensional solid objects that represent the physical structure of the room e.g. walls, doors, furniture. Textures are applied to these blocks to create the door, whiteboard, carpet and walls. Entities, on the other hand, are neither visible nor physically tangible during game-play. They exist only through the effects they supply to a map e.g. sound/light. Hammer shows the positioning of entities through the use of icons e.g. the light bulb at the centre of Figure 6.7. Entity-Brushes are the result of selecting a brush and associating an entity with it using a technique called *tying*. When *tied*, the combination provides a functional object e.g. a door or button.

Entities in Hammer are selected from drop lists on display in Figure 6.7. The lists are populated using a text file called the FGD file. This is mentioned here because it becomes

an important feature in the final design for TATUS. The FGD file can be edited to add or remove Hammer's selection of entities. Every entity added to a map can be configured through Hammer's Properties dialog box. An important value supplied as part of setting up an entity is a string value called its *targetname*. Targetnames are used by HL to identify entities in a map and are not required to be unique but where entities share a name, the engine will identify them as a group.

2.3.1.2 Triggers

Triggers are essentially *entity-brushes*, however unlike the example of an entity-brush that is a door, triggers are invisible during game-play. They are used to generate events based on a player's movements and location. For this reason, they must be invisible so that it is not possible to consciously avoid them. When a player enters a region of a map occupied by a trigger the associated event is activated e.g. a door is opened.

As a result, a normal entity alone cannot act as a trigger since the boundary of the trigger must be detectable to the game engine. In Figure 6.7 triggers can be seen as the purple blocks surrounding the door. In this instance the triggers are present to open the door when a play approaches. During the setup process for the trigger, the door's *targetname* is stored as the target for the trigger. At runtime the engine can perform a lookup using the target value to search for the entity to be activated.

The following section discusses game modification at the next level using the SDK to reprogram Half-Life.

2.3.2 HL SDK Coding

Coding refers to re-writing the HL SDK by modifying two key dynamic-linked-libraries (DLLs). The hl.dll controls games rules and entity behaviour while the client.dll is responsible for screen rendering. The HL SDK divides its code into six folders according to functionality:

- **cl_dll** contains the client.dll code for screen rendering.
- **common** contains header files built into both the hl and client dlls.
- **dll** contains the hl.dll code for physics and rules.

- **engine** contains code that allows the hl and client dlls to communicate with the core engine.
- **pm_shared** contains code to handle player movement.
- **utils** contains code to examine maps and graphical models.

The SDK allows new weapons, characters, physics, rules and graphics to be introduced. The SDK does not provide access to the underlying support infrastructure for running a game. This means there is no access to the I/O connection between the game and its peripherals, namely the player controls and the network connection. This posed a problem for Design 1 which will be discussed in Chapter 3.

Within the SDK's code, the class hierarchy for entities all stem from a class called CBaseEntity. This determines that all entities have two important common features. The first is a data structure called the *pev*, its full contents are listed in Appendix B. The pev contains attributes that are relevant to all entities regardless of their derived class. For example the *targetname* supplied through the Properties dialog of Hammer is common to all entities and is stored in the variable *targetname* of the pev structure.

The second important feature of the CBaseEntity class is a set of five functions common to all entities. These functions are *Use*, *Think*, *Touch*, *Spawn*, and *KeyValue*.

- **Use** is invoked to activate an entity e.g. call Use to open a door. As its parameters it takes pointers to two entities, *activator* and *caller*. Activator is the entity initiating the sequence of events e.g. the player that walked through the trigger. Caller is the trigger invoking the door's Use function.
- **Touch** is invoked when two entities collide, usually due to a player or NPC walking into another entity. This function will most commonly be talked about in relation to a player walking through a trigger. A single argument to the function is an entity pointer to the second entity involved in the collision.
- **Think** is invoked at regular time intervals and is used to give the impression of *thinking*. For example a monster often uses its Think function to intelligently change a player through a map by invoking its AI at regular intervals.
- **Spawn** is called when loading a map to initialise each entity in the map.

- **KeyValue** extracts the data provided through Hammer's Properties dialog and uses it to populate an entity's data structures.

2.4 Conclusion

Chapter 2 discussed state-of-the-art in terms of a variety of test-beds employed to evaluate ubiquitous computing environments. A 3D interactive simulator called UbiWise was presented and reasons were provided why UbiWise does not satisfy the objectives for this project. Finally Half-Life was introduced and the main features of its development environment were described.

Chapter 3

Design

Chapter 2 introduced the HL SDK and although in retrospect it is easy to isolate modification of the game's code to a limited set of files, the discovery stage and learning curve were both difficult and time-consuming. Two main factors affected the learning process.

Firstly, although the HL SDK was chosen because its tutorials and articles were by far the most comprehensible, information in this format is no substitute for well-documented code. No single website within the Half-Life community condenses all aspects of the engine into a single source. The second impediment is the SDK's size. Chapter 2 mentions the directory structure that stores the SDK's files. Modification of the SDK targets two specific dll files, *hl.dll* and *client.dll*. Between them, these libraries implement over 250 files and this number greatly increases when the dependencies on header files stored in the remaining folders are resolved.

Faced with these two problems, each mutually exacerbating the other, a rapid-prototyping approach was adopted. Chapter 3 begins its discussion of the simulator's design by setting out the project requirements that were drawn up for TATUS.

3.1 Project Requirements

The high-level goal for this project was to provide a convenient and flexible 3D virtual ubiquitous computing environment that researchers can use to test ubiquitous computing applications currently under development. Chapter 1 gave a broad overview of some key objectives for the simulator which in the initial stages of the project were outlined in only the most

general and vague terms.

Specific design requirements were derived by combining these initial key objectives with requests gathered from departmental researchers. Prospective users attended a workshop in January 2004 for the opportunity to suggest potential features for the final simulator. The full potential feature set drawn up at this meeting are documented in Appendix D.

In accordance with the finite time available to complete this project, a subset of features were selected for implementation. Although many items on the original list are important characteristics of a ubiquitous computing environment, it was necessary to choose features that would be most useful in the short term. As a result, items such as state extraction were chosen before issues such as modeling network connectivity. The selected items are listed under the Project Requirements List.

3.1.1 Project Requirements List

1. **3D Graphical Interface**

   Provision of a 3D interactive graphical user interface using an off-the-shelf and modifiable game engine such that a researcher can navigate through the virtual ubiquitous computing environment.

2. **Separation of HL and SUT**

   The project design must not require that the SUT be developed or integrated as part of the modified game engine. Instead, extraction of state information will allow the SUT to control the environment from an external position.

3. **Realism**

   The simulated ubiquitous computing environment must realistically model the equivalent real-world physical implementation as closely as possible. It is crucial that Half-Life's complex data system is not exploited in an unreasonable manner.

4. **Flexibility**

   The simulator is being developed to test a range of software. Its design must be generic and not tailored to provide specific state or to interface to a particular piece of software.

5. **Usability**

   The connection procedure, resulting from requirement 2, between the simulator and SUT must be straight-forward and quick to implement. This also supports requirement 4, flexibility, by encouraging re-use of the simulator.

6. **Extensibility**

   Given the finite time allocated for this project, a subset of potential features for the simulator have been drafted, see requirement 7. This extensibility requirement provides for extending the feature set at a later date.

7. **Short List**

   (a) **Information Extraction:** State information is streamed out of the HL environment to the SUT. The SUT uses the data to build its own picture of the world relevant to the decisions it has been programmed to make about the ubiquitous computing environment.

   (b) **State Selection Service:** Researchers are provided with a mechanism to select a subset of the state information most suited to the goals of the SUT. This is to avoid a full state dump, potentially containing surplus data and wasting machine resources.

   (c) **SUT Influence:** The SUT is provided with an interface or API to impose changes and decisions on the simulated ubiquitous computing environment. Where control of events has been designated to the SUT it is important that Half-Life's game rules and physics do not influence the state of the environment.

   (d) **Scenario Creation:** Researchers can set up a test environment most fitting to SUT experiments, for example a lecture scenario requires a presentation room.

The following sections of this chapter discuss developing a design to meet these criteria.

3.2 Simulator Design

In the initial stages of the rapid-prototyping approach, the high-level design for the simulator was generalised and vague as shown in Figure 3.1. The modified game engine and SUT needed

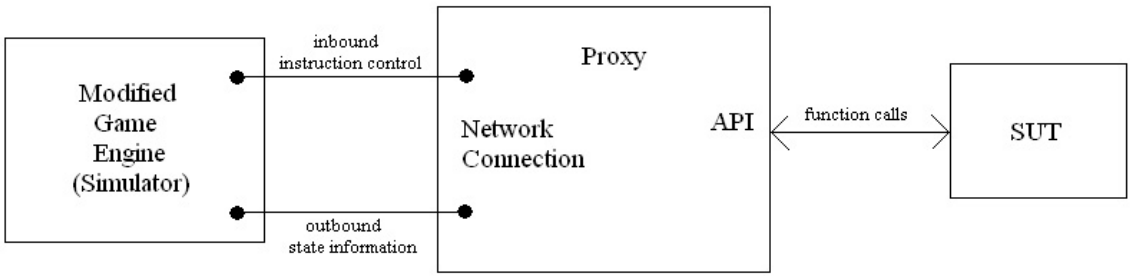

Figure 3.1 High Level Design Overview.

some method of communication to exchange information. Messages travelling outbound from the simulator contain state information about the simulated environment. Messages travelling inbound to the simulator contain instructions to adjust the simulated environment.

There are two specific features introduced as part of this high-level overview. These features specifically target requirements 2, 4 and 5, the *HL/SUT separation*, *flexibility* and *usability* and are listed below.

1. **Network Connection**

   The network connection allows the simulator and SUT to run on separate computers. This is important because when both programs are run in parallel on a single machine the simulator's graphics absorb the entire screen. In addition, the keyboard and mouse are dominated by Half-Life's player controls. Running each program on a separate machine means a researcher can view and control both programs concurrently. This is particularly relevant when debugging test software.

2. **Proxy**

   The Proxy removes any need to integrate a network connection into the SUT code by providing a ready-made link to the simulator. This is supplied with a view to reducing set-up time when initially connecting new SUT to TATUS. The Proxy also provides an API that offers function calls to send and receive messages to and from the simulator.

Two designs were considered for implementing TATUS.

3.3 Design 1:Exploiting HL's Client-Server Message System

Network games such as Half-Life use a client-server model to distribute and share information. Potentially this is a goldmine of opportunity for capturing the state of the environment. The intended approach involved two steps:

1. Produce a *Reduced Client* by removing the screen rendering code from the SDK. The SUT determines no useful information from the image printed to the screen of a computer. Figure 3.2 shows how the SUT interacts with the environment through the reduced, non-graphical client while researchers continue to make use of the standard Half-Life client.

2. Capture state as it arrives from the server at the client-side. Primarily the SUT is interested in changes to the environment and so it will interpret state information as a delta value calculated from the current and new views of the world. Based on the calculations the SUT will make decisions and provide instructions that change the state of the ubiquitous computing environment.

The success of this approach depended on the feasibility of both steps listed above. A discussion of each follows:

1. Part one of this approach, although never implemented is believed to be feasible. The HL SDK adopts a split-share approach to its code. All user interface code, such as graphics and sound, are built into the *client.dll* file. The rules and physics that support the environment are part of the *hl.dll* file. Both dlls work from shared data structures and variables.

   The hl.dll runs AI code to update the shared data. The client.dll reads the shared data in order to render the world on screen. From this it seems reasonable to assume that the internal representation of the environment will persist if client.dll is removed from the project.

2. Chapter 2 discussed both the organisation of files within the SDK's directory structure and the access rights of programmers to the core engine code. Part 2 of this design fails because Valve does not allow modification to the game's supporting infrastructure. In

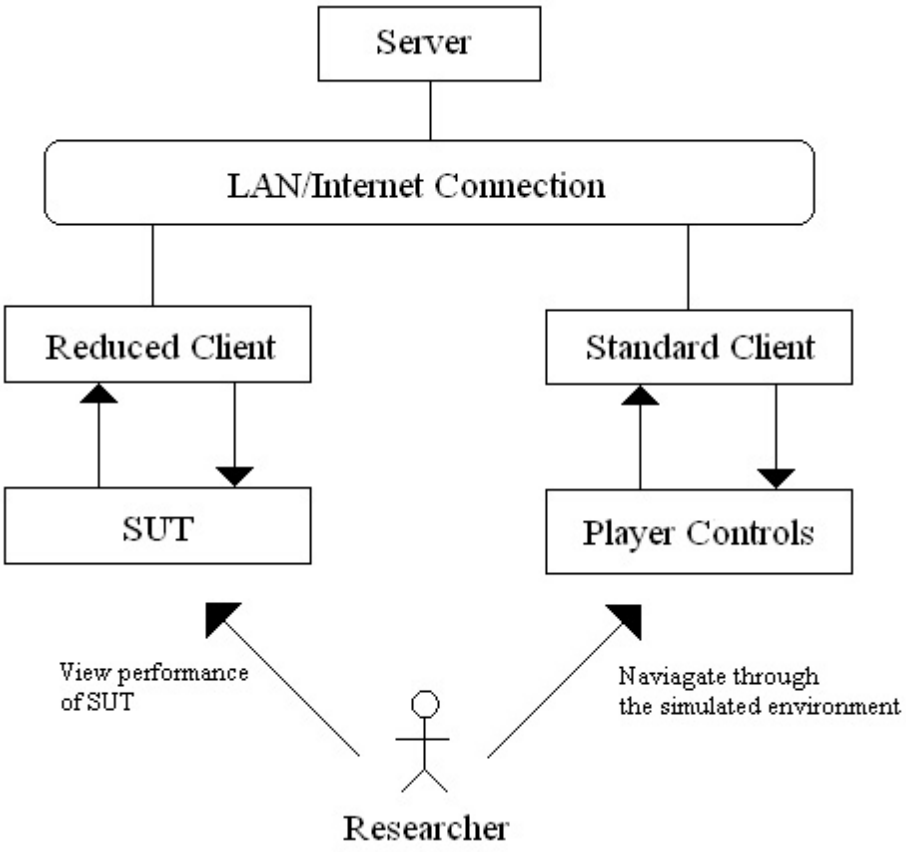

Figure 3.2 Overview of Design 1.

particular the I/O with peripheral resources such as player controls and the network connection is protected. Access to the network connection is required to capture state as it is delivered from the server.

Intercepting messages at this level would have allowed the simulator to capture the full view of the environment in order to pass it on to the SUT. Instead a mod developer can only manipulate this data at a much higher level after the messages have been parsed and their contents distributed to the appropriate variables and data structures.

Overall it was not possible to implement Design 1. Design 2 discusses an alternative approach sparked by exploiting *triggers* to generate event-driven messages.

3.4 Design 2: Exploiting Triggers

Based on the information presented about the HL SDK in chapter 2, it is evident that triggers are used to generate events. There is a definite parallel between this and objective 2 presented in chapter 1. Objective 2 states the simulator must allow researchers to select simulator events of interest for notification to the SUT.

However, according to requirement 7(c), the SUT must be designated to have complete control over the outcome of these selected events. The rules and physics in the game engine cannot intervene or supplement the result. As such, the event's information must be passed on to the SUT, which in turn will decide whether or not an environment change is required. Design 2 began with the adaptation of a standard trigger to capture state information and stream it out to the SUT.

3.4.1 Ubiquitous Computing Trigger & Information Extraction

In an early prototype experiment for the simulator, a character enters a room, the event is noted by the SUT and the lighting is switched on. Figure 3.3 shows a map for the scenario. A trigger has been placed across the doorway to register the event when the player enters the room. In this scenario the *targetname* for the *light-entity* is supplied as the *target* value for the *trigger*.

Under standard game conditions the light's *Use* function is invoked for every activation of the trigger. Removing this function call takes control of the event away from the game. For

Figure 3.3 Intelligent lighting scenario.

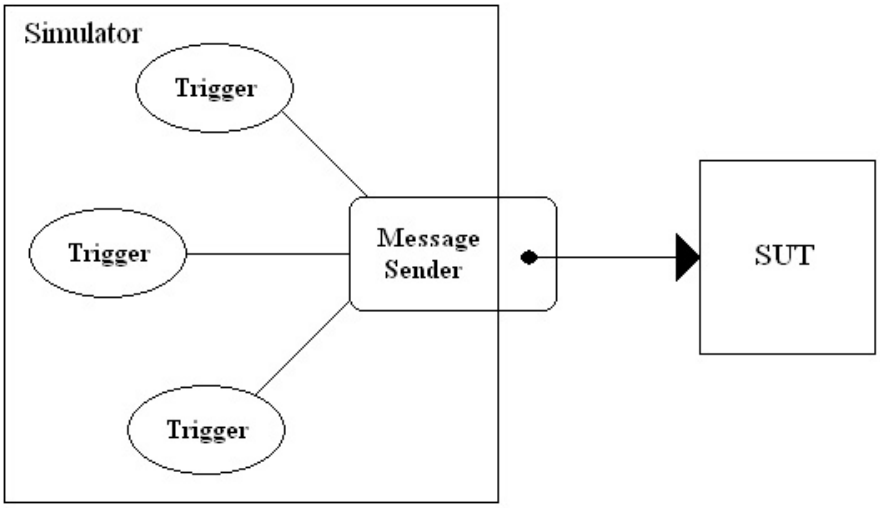

Figure 3.4 Trigger and Message Sender combination.

the moment this satisfies half of requirement 7(c) by preventing the game from interfering with the outcome of the event. To further meet the requirement, information about the event must be gathered and delivered to the SUT.

The expected language choice for implementing SUT is Java. As a result, the connection mechanism between the simulator and SUT must permit Java and C++ to successfully communicate. Due to the complex nature of HL, techniques such as RMI (Remote-Method-Invocation) and technologies such as CORBA are too heavy-weight to be incorporated into the SDK's code. In addition, investigation determined that the C++ socket class, CSocket, is also beyond the tolerance of the HL SDK due to conflicting libraries. Instead a low-level socket implementation, using winsock.h, proved viable.

Providing each trigger with its own socket connection is not practical in terms of efficiency or scalability. Such an implementation would require one port for each trigger added to a map. A second entity, called a *Message Sender*, is included in the design to host a single socket connection to the network, see Figure 3.4. Each trigger stores a pointer to the Message Sender object so that when activated the messages generated can be sent to the SUT through the Message Sender.

The combination presented in Figure 3.4, shows how a set of triggers and a Message

Sender work together to pass information out of the simulator and on to the SUT. Streaming data out of the simulator in this manner is a near fulfilment of requirement 7(a). As was mentioned in section 3.2, part of this design includes a proxy that hosts a network connection on behalf of the SUT. The Proxy is responsible for delivery of messages to the SUT which completely satisfies requirement 7(a). The Proxy's design will be discussed in section 3.5. The design so far has partially satisfied requirement 7(c), to completely meet its requisites an instruction protocol and inbound delivery service is needed.

3.4.2 SUT Influence

An asynchronous messaging system between TATUS and the SUT is used for two reasons:

1. The HL engine will not tolerate a synchronous messaging system whereby game progress is expected to halt while the SUT performs its processing.

2. Although the SUT is using game events to build up its own view of the world, it may not want to act on every event that it is notified about. An asynchronous design provides the SUT with the option to ignore uninteresting events.

From this a separate inbound connection is required from the SUT to TATUS. It has already been established that the low-level socket implementation used by winsock.h is the best solution for connections between the SUT and the simulator.

A *Message Distributor* entity is included in this design to fill two roles:

1. It hosts the second socket added to the simulator.

2. It manages the redistribution of instructions that arrived from the simulator to their intended entity.

The Message Distributor is designed to use its *Think* cycle to check the socket for incoming messages. Based on the *targetname* supplied in the message, the Message Distributor will call on an engine look-up function to get a pointer to that entity and thus deliver the message.

3.4.3 Polling Body

This section temporarily revisits the issue of state extraction. The *ubiquitous computing trigger* has fully satisfied the requisites for requirement 7(a) however according to the event-style messaging used, it is conceivable that the SUT may endure extended periods of data starvation due to an absence of events. A second entity is introduced to extract data from the simulator.

A *Polling Body* imitates the *Message Distributor* in its exploitation of the *Think* cycle. The Polling Body also imitates the ubiquitous computing trigger in its mechanism for sending messages to the SUT. The combined effect is an entity capable of collecting information at regular intervals on behalf of the SUT. The introduction of the Polling Body allows this design to surpass the potential offered by Design 1 mentioned at the start of the chapter. Design 1 was confined to fixed state deliveries from the central game server. This second design offers both timed and event-driven state delivery to the SUT. Figure 3.5 shows the overall design for the 3D simulator component of TATUS.

3.4.4 Messaging Protocol

The messaging protocol provides the vocabulary between the simulator and Proxy. Messages are written in XML [8] and defined using two DTDs [9] created for both inbound and outbound messages. All messages are sent in string format, with inbound messages prefixed by their length to assist dynamic memory allocation on the HL receiver side. DTDs for TATUS are held in Appendix C.

XML was chosen to structure messages because it fills the two basic criteria required:

1. Messages must be strings or character based for socket based communication.

2. Messages must incorporate semantics about data leaving and instructions entering the simulator.

Outbound DTD

Outbound messages are produced by two separate types of entity so two separate types of message exist, *event* and *poll*. Since event messages are mainly concerned with entities

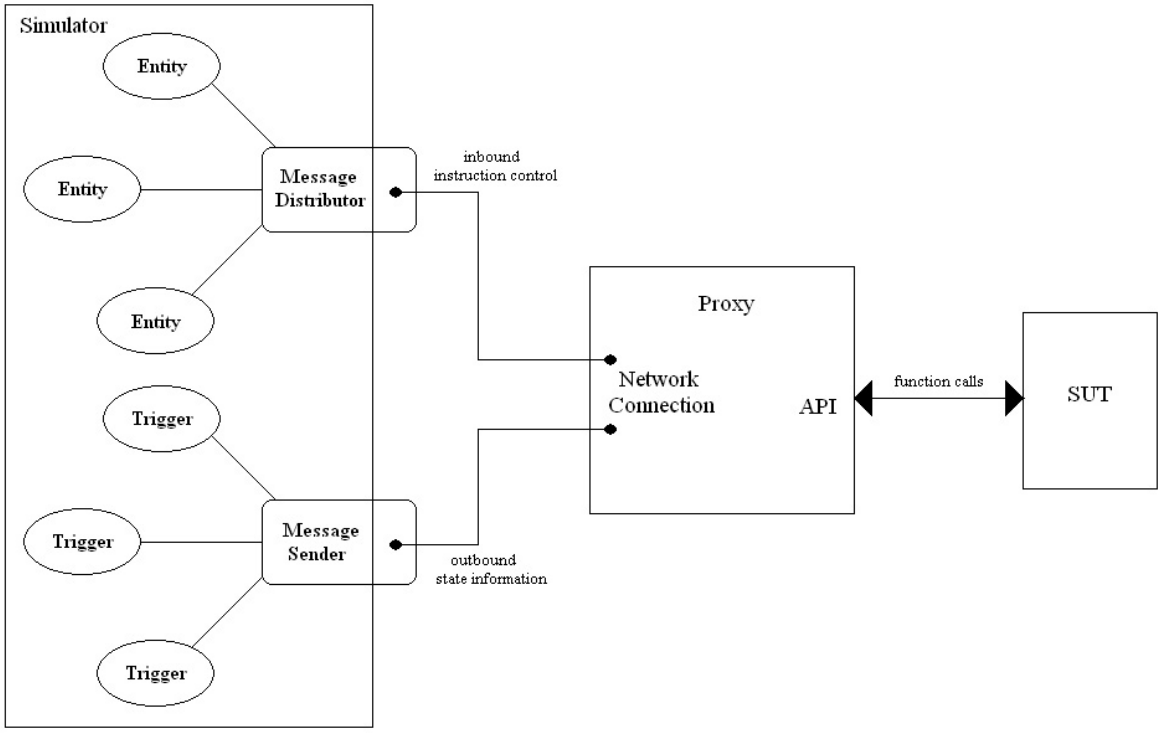

Figure 3.5 Core 3D simulator design.

and attributes involved in a trigger's activation, the event message elements are designed to describe information about the *activator*, *target* and *trigger*. The *globals* element stores information sourced from the *gpGlobals* data structure which is included in Appendix B.

Poll messages use five elements to enclose data, *instance*, *class_set*, *targeting*, *radius* and *globals*.

1. **Instance** elements contain the *targetname* of a map entity along with a subset of *pev* attributes from the named entity.

2. **Class_set** elements are the most sophisticated element. They contain a *classname* along with both *pev* and *local variable* attribute values. It is not possible to include local variables for other elements since their classname cannot be predetermined outside run-time. This element gathers information about all entities of this class.

3. **Targeting** elements contain the *targetname* of a map entity along with a subset of *pev* attributes. Information is collected for every entity in the map that is *targeting* the named map entity.

4. **Radius** elements contain the *targetname* of a map entity, an integer value denoting *radius* along with selected *pev* attributes. The *radius* supplied marks out a sphere surrounding the named entity. Information is collected about every entity that lies within that sphere.

Inbound DTD

Inbound messages contain instructions to change the environment. The main commands used to do this call on an entity's *Touch*, *Use* and *Think* functions. The Inbound DTD contains three major elements to accommodate these function calls.

- **Touch** names the entity receiving the instruction. It also names the other entity that was supposedly involved in the collision.

- **Use** names the entity receiving the instruction. It also names the function's activator and caller. Finally it specifies a *usetype*, either switch on, switch off, toggle or set. On and off commands switch the entity on or off respectively and toggle switches between on and off. Classes interpret the set command individually.

- **Think** names the entity receiving the instruction.

Message Format DTD

A third DTD included in the design satisfies requirement 7(b) which states that researchers must be provided with a mechanism to select a subset of the state information most suited to the goals of the SUT. It is specifically aimed at defining the attribute set to be collected by *ubiquitous computing triggers* and *polling bodies* with a view to avoiding a full state dump.

The elements of this DTD correspond to the elements of the Outbound Message DTD with all semantics persisting. It will be explained later in this chapter how a researcher links the XML files according to this DTD with the ubiquitous computing triggers and polling bodies added to maps.

3.4.5 Message Definition Tool

Hand-writing XML is inconvenient and tedious and although requirement 7(b) has been fully satisfied with the inclusion of the Message Format DTD as part of the design, this Message Definition Tool is introduced as a convenience feature for the researcher with a view to improving general usability of the tool as laid out in objective 5.

This message definition tool provides a Java Swing interface displaying the available attribute set for each of the major elements within poll and event messages. Using the interface, a researcher can select a subset of attributes for each element. The tool will generate the appropriate XML content when the document is saved. The tool also loads existing XML format files to allow reselection or refinement of previously used message formats.

In addition to relieving the researcher from the tedious task of writing XML this further improves usability by giving easy access to the HL data store. As a result of keeping developers outside the HL environment they are not familiar with the variables used by the engine. Presenting the available attributes in this manner improves usability of the ubiquitous computing simulator system as a whole.

It was mentioned in chapter 2 that *Hammer* parses a FGD file to populate the lists of entities and attributes, or properties. It was also noted that the FGD file is extensible to accomodate extra entities added as part of a new game modification. In accordance with this, the Message Definition Tool uses the same FGD file, along with the pev and gpGlobals

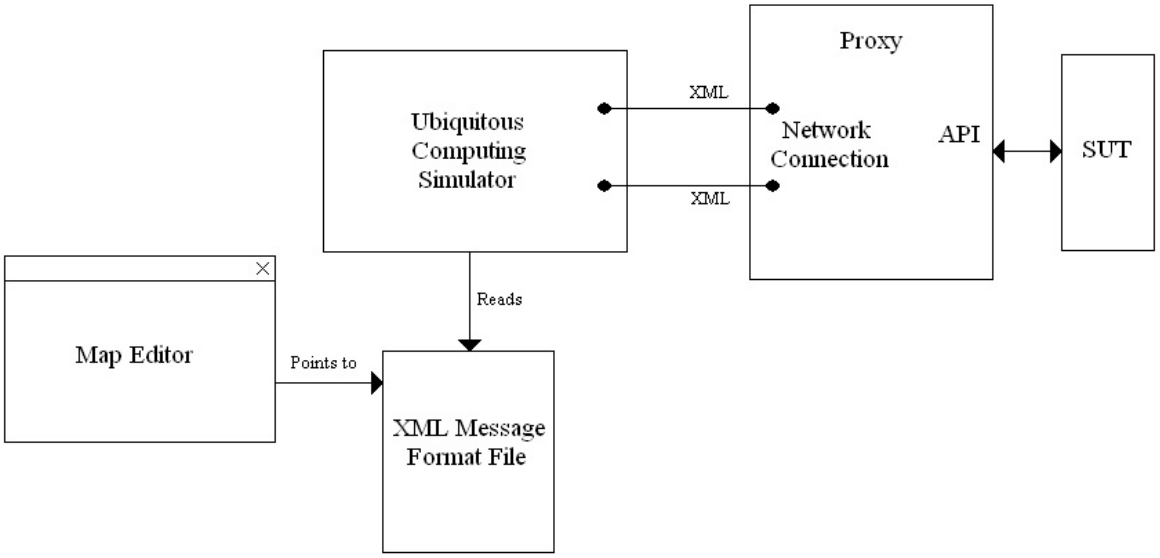

Figure 3.6 Role of XML message format file.

structures, to populate the attribute sets on display. As a result, the Message Definition Tool will automatically update itself to correspond with future extensions to TATUS and hence satisfying requirement 5, *extensibility*.

3.4.6 Extension to Hammer (Map Editor)

According to requirement 7(d), the researcher requires a mechanism to create scenarios most suitable for running experiments designed for the SUT. Hammer provides the solution to this requirement. The FGD file has been extended to include the ubiquitous computing trigger, the polling body entity, the Message Sender and the Message Distributor. As a result Hammer is fully compliant with maps for TATUS.

3.5 Java Proxy

In response to requirements 2, *Separation of Half-Life and SUT*, and 5, *Usability*, a proxy is included in the system design. It hosts a network connection to the simulator which is accessible to the SUT using an API. The Proxy minimises the adjustment made to the SUT during the initial setup procedure by providing a ready-made network connection. The API

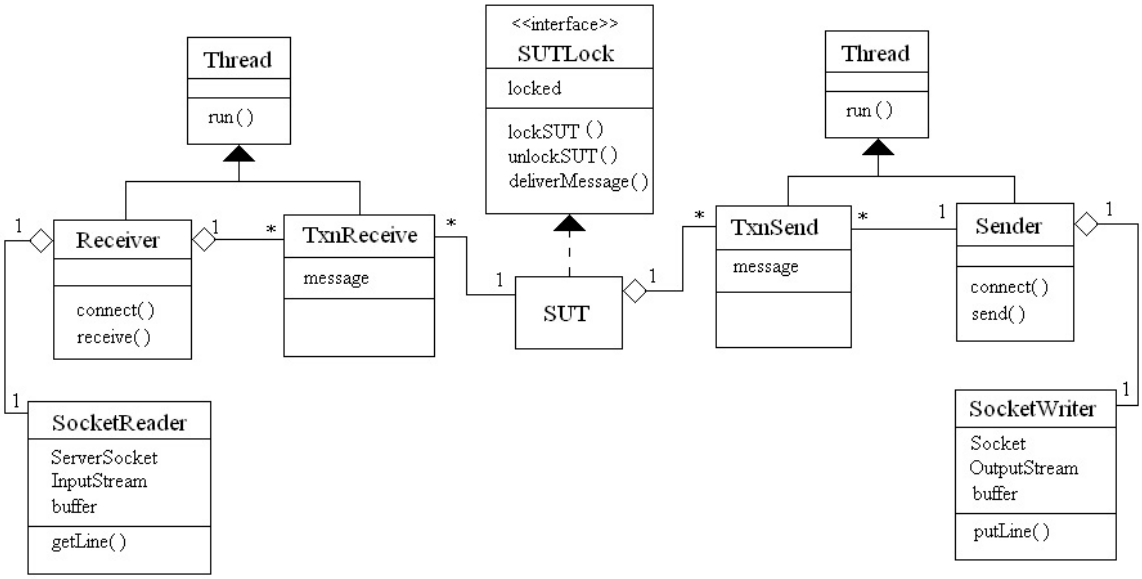

Figure 3.7 Proxy class system overview.

uses overloaded functions to allow the SUT to send instructions in either XML or DOM format. Figure 3.7 displays the relationships between the classes designed to implement the Proxy.

The remainder of this section discusses the role of each class in allowing the proxy to operate.

- **SocketReader** hosts the inbound socket, and provides a method to read one line of text from the socket at a time.
- **Receiver** manages messages retrieval through iterative calls to SocketReader's getLine method and thus ensures messages are received in full. Receiver will spawn a TxnReceive thread to deliver every message that is removed from the network.
- **TxnReceive** threads will persist until they successfully obtain the lock on the SUT and deliver their message.
- **SUTLock** is an interface that the SUT must implement as part of the connection procedure to the Proxy. The SUTLock prevents interference between threads attempting to deliver messages to the SUT.

- **TxnSender** threads are spawned by the SUT and are supplied with an instruction for the simulated environment. These threads will attempt to obtain the lock on the Sender object which will forward the instructions across the network.
- **Sender** manages dispatching instructions to the simulator. SUT working under light loads may wish to avoid using threads to send messages so it is also possible to make a direct function call to the Sender object to send an instruction.
- **SocketReader** hosts the outbound connection to the simulator and provides a method to put one line of text through the socket at a time. Each line of text should correspond to a full message based on the receiving mechanism implemented on the simulator side which is described in chapter 4.

3.6 Conclusion

The final design for TATUS is laid out in Figure 3.8 featuring the following system components:

1. Core 3D Simulator
2. Proxy
3. Message Definition Tool
4. Map Editor

The role of each component is synopsised below:

1. The Core 3D simulator provides an interactive simulated ubiquitous computing environment. Ubiquitous Computing Triggers and Polling Bodies generate event-driven and polled messages respectively. All messages contain simulator state information and are sent to the SUT. A Message Sender object hosts a socket connection as an outbound access point to the network connection. A Message Distributor hosts a corresponding inbound socket and delivers arriving instructions to the appropriate entities.

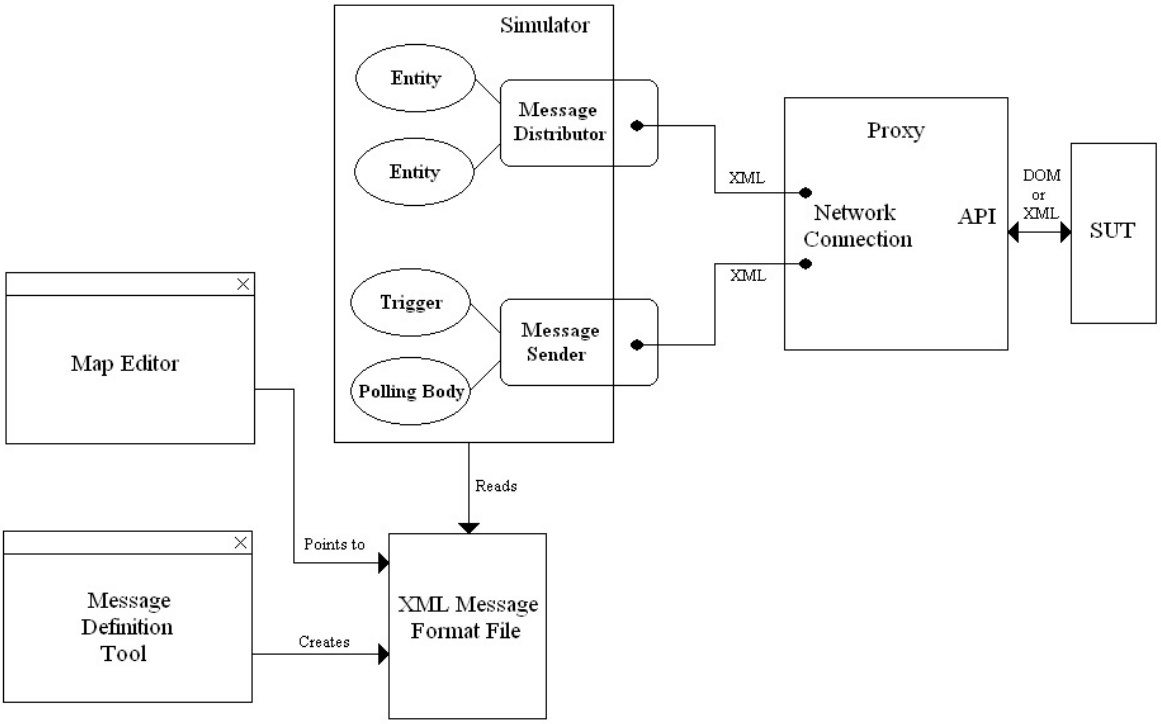

Figure 3.8 Final system design for TATUS.

2. The Proxy hosts the network connection to the simulator so that the connection setup procedure between SUT and TATUS is minimised. The Proxy's classes provide an API to the SUT as an easy method to send and receive messages.

3. The Message Definition Tool generates and saves the XML Message Format File. The contents of the file are defined by the researcher using the tool's GUI.

4. The Map Editor offers a toolset for creating maps and saves them in a BSP/MAP format so they can be loaded into the simulator.

Chapter 4

System Implementation

4.1 Introduction

This chapter describes the implementation of the design as presented in chapter 3. The chapter is divided into two sections. Firstly an overview of TATUS provides a high-level view of the relationships between the components that make up the system. The second part of the chapter will discuss the implementation of each component in isolation starting with the *3D Simulator*, followed by the *Java Proxy*, the *Map Editor* and finally the *Message Definition Tool*.

Before progressing any further, it is worth noting the following terminology. Entities within the game can be identified by two separate sets of names depending on whether they are being discussed in relation to the map editor or SDK's code. For example, taking the ubiquitous computing trigger, in the map editor it is known by the name tcd_message_trigger while in the SDK the class representing this entity is called TcdTriggerMessage. A translation function, shown below, lets the game engine map between the two entity names and hence the engine can successfully create a model of the world in memory using C++ objects.

```
LINK\_ENTITY\_TO\_CLASS( tcd\_message\_trigger, TcdTriggerMessage);
```

Table 4.1 shows the name mapping for all the ubiquitous computing entities. As far as possible throughout this document, map names will be used when discussing entities in maps and similarly, SDK class names will refer to code implementation.

Map Name	SDK Class Name
tcd_message_trigger	TcdTriggerMessage
tcd_polling_body	TcdPollingBody
tcd_message_sender	TcdMessageSender
tcd_message_distributor	TcdMessageDistributor

Table 4.1 Translation between map names and SDK class names.

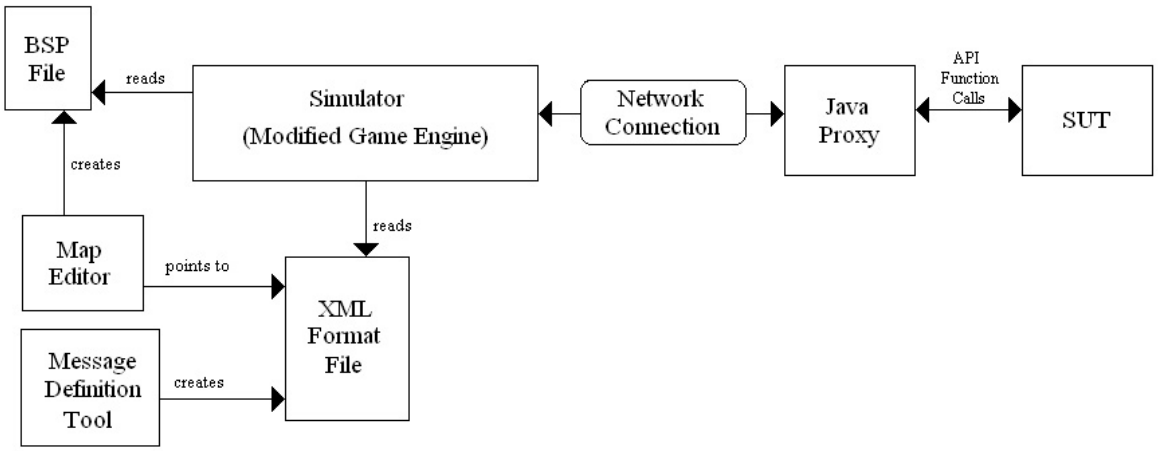

Figure 4.1 An overview of TATUS.

4.2 Component Relationships

Figure 4.1 displays an overview of the components that make up the system supporting the ubiquitous computing simulator. A key feature of this diagram shows the relationship between the XML format file and the system components that use it. The Message Definition tool is responsible for creating the file and writing to it the XML content generated by the researcher's chosen attribute set. The Map Editor saves the name of the XML file into the BSP file. When the simulator reads the BSP file it also reads the list of XML files and loads them into DOM objects for easier accessibilty.

4.2.1 3D Simulator

On start-up, the engine reads the BSP file and loads its own model of the environment into memory. Part of this process is to determine the set of XML files relied on by entities inside the level. Also during this set-up phase entities are spawned, a term used to denote creating an instance of an entity class, similar to running the constructor in a standard object-oriented language. In the case of TcdMessageSender and TcdMessageDistributor, this involves establishing a network connection to the proxy. TcdMessageTrigger and TcdPollingBody use the procedure to obtain a reference to TcdMessageSender.

4.2.2 Message Definition Tool

The purpose of this tool is to generate the XML format files. At the user's discretion, this should be added to Half-Life's working directory under the path `.\tcdUbiSim\xmlFiles`. The simulator will search this folder when starting a new level.

4.2.3 Map Editor

When inserting a new tcd_trigger_message or tcd_polling_body into a map, it is required that an XML format file name be supplied as one of the object properties, see Figure 4.2. Tcd_polling_bodies are also supplied with their *think interval* through the same dialog.

4.2.4 Java Proxy

Java Proxy supports the network connection to and from the simulator. The proxy program must always run before the simulator start-ups to successfully allow spawning of TcdMessageSender and TcdMessageDistributor entities when a level is loading.

4.3 Component Implementations

In the following section the implementation of each system component will be covered in detail, starting with the 3D simulator, followed by the Java Proxy, the Map Editor and finally the Message Definition Tool.

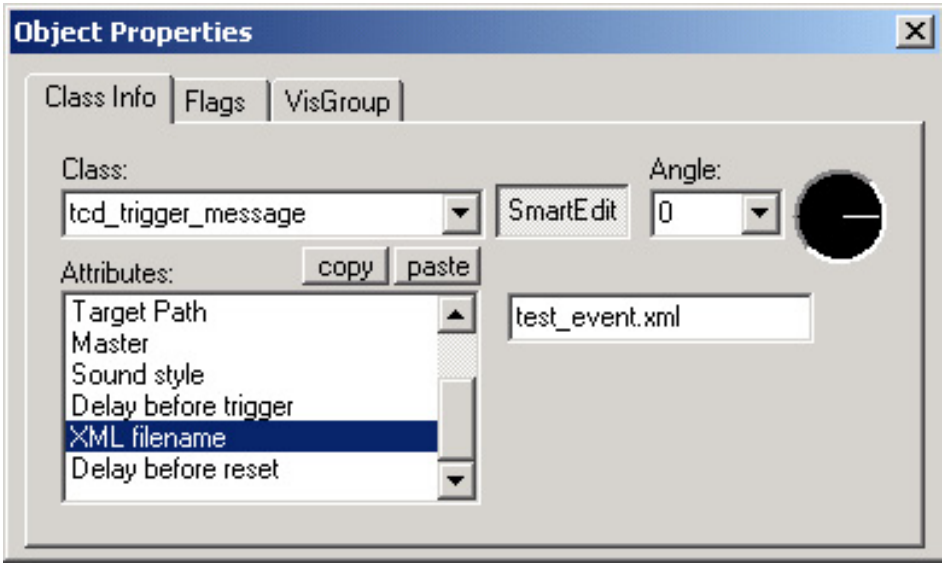

Figure 4.2 Adding an XML filename property to a ubiquitous computing trigger.

4.3.1 3D Simulator

Figure 4.3 shows the interactions involving entities inside the simulator, both amongst themselves and with system components external to the simulator. This implementation section will focus on the four ubiquitous computing entities added to the SDK as part of the game to ubiquitous computing simulator conversion.

TcdMessageSender: Implementing an Outbound Connection

As discussed in chapter 2, *entities* in Half-Life exist either independantly or tied to *brushes*. It was also mentioned that *entity-brushes* are physically tangible and reactive to *touch*. Standard entities or point-entities on the other hand are not. For this reason TcdMessageSender has been derived from *CPointEntity*, the point-entity base-class, with a view to preventing player and NPC interaction or interference.

The outbound network connection managed by TcdMessageSender is set-up as a client socket running on port 5060. For reasons mentioned in chapter 3 the low-level sockets available from winsock.h are used to implement this connection. Figure 4.4 shows TcdMessageSender's setup method. The parameters to this function are a port number and a string value identifying the server e.g.*localhost* could be supplied as this value. WSAStartup sets up

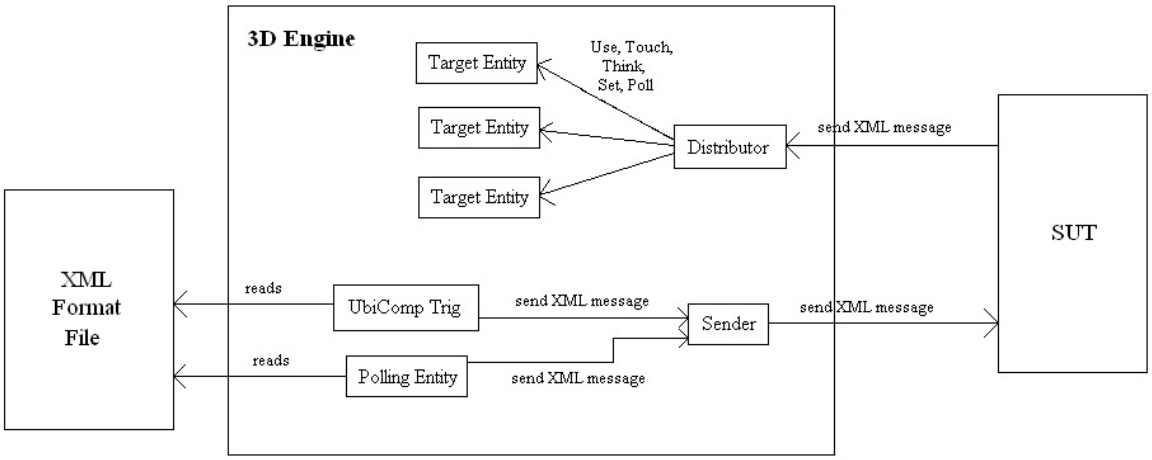

Figure 4.3 Core simulator design.

parameters for winsock while the function *gethostbyname* returns a data structure containing information such as the host's network address and the type network address used e.g. an IP address. Finally, an open socket is allocated for the connection.

TcdMessageDistributor & TcdMessageReceiver

Two classes share responsibility for the receipt and distribution of messages. TcdMessageReceiver is a stand-alone class, it does not inherit any features specific to the game e.g. from *CBaseEntity*. Similar to TcdMessageSender, during the Spawn cycle of TcdMessageReceiver, port 5061 is assigned to host the connection. However, this time a listening socket is employed because the asynchronous messaging system means that messages arriving via this port are all initiated at the proxy side.

In view of the fact that the SDK does not support string manipulation as defined in the standard C++ library ¡string.h¿, an alternative method to accommodate varying message lengths was required. Messages leaving the proxy are prefixed by four digits, denoting messages length, and so allowing messages to reach lengths up to 9999 characters. TcdMessageReceiver parses this value before taking the message content from the socket's buffer.

In addition to preventing a requirement for fixed length messages, this allows memory to be allocated and also retrieved at run-time, an important feature given the memory footprint for the game. The code shown in Figure 4.6 controls memory allocation and message retrieval,

```
int TcdMessageSender::setup(short p, const char* s){

	WSAStartup( 0x0101, &wsaData);

	strcpy(servername, s);

	portnum = p;

	/* get host address */
	h = gethostbyname(servername);
	if (h == NULL) {
		return -1;//exit(0);
	}

	/*  build socket structure  */
	memcpy((char *)&sa.sin_addr,(char *)h->h_addr, h->h_length);
	sa.sin_family = h->h_addrtype;
	sa.sin_port = htons(portnum);

	/*  allocate an open socket */
	soxdes = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
	if (soxdes == -1) {
		return -1;
	}

	return 0;
}
```

Figure 4.4 TcdMessageSender setup function, tcd_sender_socket.cpp.

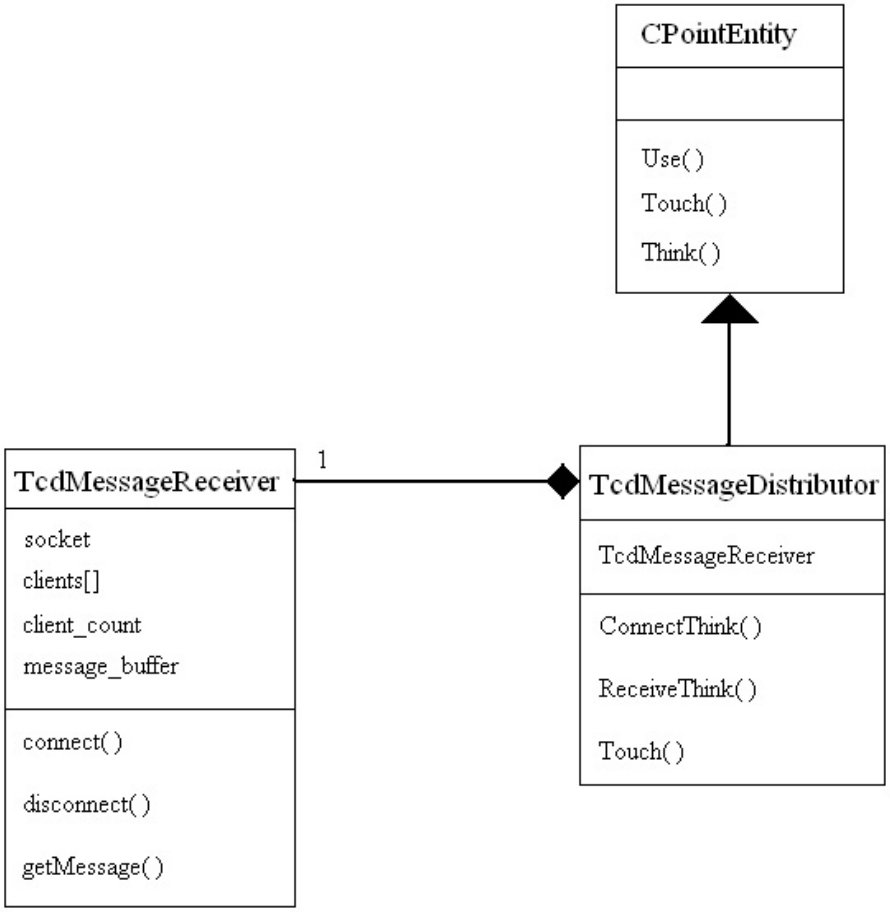

Figure 4.5 Class diagram for TcdMessageDistributor.

```
int prefixLen = 4;
bufferSize = new char[prefixLen];
recv(Client[s].ClientSocket, bufferSize, prefixLen, 0);
bufferLen = 0;

for(int i = 0; i< prefixLen; i++){
      bufferLen = bufferLen * 10;
      int nextDigit = (int) (bufferSize[i]-'0');
      bufferLen = bufferLen+nextDigit;
}

if(bufferLen > 0){
      buffer = new char[bufferLen];
      recv(Client[s].ClientSocket,buffer,bufferLen,0);
}
delete [] bufferSize;
```

Figure 4.6 Allocation of memory when receiving a message, tcd_receiver_socket.cpp.

it can be found in tcd_receiver_socket.cpp.

It should be noted that because the memory allocated to the buffer array must be referenced outside this method, *buffer* is stored as a global variable. Further a separate function called deleteBuffer will release the memory when the message has been parsed.

TcdMessageReceiver also provides a set of functions for use by TcdMessageDistributor which carry out actions such as connection set-up and message retrieval. Although TcdMessageReceiver provides connection set-up code at the level of initialising Winsock parameters, TcdMessageDistributor manages connection establishment ensuring the setup code is re-run until a successful outcome is achieved.

By exploiting the TcdMessageDistributor's Think function, TcdMessageReceiver's setup code is invoked at regular intervals until the network link is constructed. However, when this task is accomplished, the TcdMessageDistributor's cycle becomes redundant. Through the use of a *SetThink* function, the class can define two implementations of the Think function for itself, switching between the two as determined by the implemented logic.

Figure 4.7 displays the code for the ConnectThink() function. Initially it attempts to locate a client and make a connection. If it is successful in this, control is switched to the RetrieveThink() function and finally regardless of the connection outcome the time is set for the next Think invocation. RetrieveThink, as the name implies, calls on the methods from

```
void TcdMessageDistributor :: ConnectThink(){

      connected = server.Check_Connections();
      if(connected > 0){
            SetThink( ReceiveThink );
      }
      pev->nextthink = gpGlobals->time + 1;
}
```

Figure 4.7 TcdMessageDistributor's ConnectThink function, tcd_message_distributor.cpp.

TcdMessageReceiver to check the socket for any messages that are waiting to be delivered.

A drawback to this design means that messages can only be serviced on the simulator side as quickly as the think interval allows, since only one message is retrieved per cycle. However, easing this problem, the Inbound Message DTD does provide for the inclusion of more than a single instruction per message.

When a message arrives, its content is loaded into a DOM object using the XercesC [12] tool. TcdMessageDistributor need only manipulate the object to the point of determining which entity is the target of each instruction. Using the name supplied by the SUT, the following look-up function will obtain a pointer to the entity instance in question and from that the appropriate function, *Think*, *Touch* or *Use* can be called.

```
CBaseEntity *activator = UTIL_FindEntityByTargetname(NULL, activatorname);
```

Should the case occur that more than one entity has been entered in the map under the same targetname, a while loop will apply the instruction to all entities saved under the provided name. This is a useful feature when the map creator wants a single trigger to target more than one entity e.g. all the lights in a single room.

TcdTriggerMessage

TcdTriggerMessage inherits from the CBaseTrigger to maintain standard trigger characteristics i.e. reaction to touch and invisibility. At spawn time it locates the XML format file named in its properties and loads the content into a DOM object, again using XercesC. A

separate standard game function, *KeyValue*, is used to extract property information from the map and populate the data variables of entity objects. As such, the filename will already have been supplied to TcdTriggerMessage before Spawn() is called.

The second action performed at Spawn time is the retrieval of a reference to the tcd_message_sender. Previously, targetnames have been discussed as means of identifying entities using the *UTIL_FindEntityBy* TcdMessageSender is identified by its classname instead, since a well-formed map will only ever have one instance of the class included in it. The next piece of code is an equivalent lookup function that uses the classname of an entity instead of a targetname.

```
sender = TcdMessageSender::Instance(FIND\_ENTITY\_BY\_CLASSNAME
                                    ( NULL, "tcd\_message\_sender" ));
```

TcdPollingBody

Implemented with the same principles in mind as the tcd_message_trigger, however working on its Think function rather than its Touch function, the entity gathers data on a timed cycle. Similar to TcdMessageSender and TcdMessageDistributor, it inherits from *CPointEntity* making it invisible and untouchable.

The code extract in Figure 4.9 shows the polling body using the Xerces parser to get an instance of the XML format file represented as a DOMDocument (DOM) object. The genMessage function manipulates the DOM object and amends it by inserting corresponding attribute values for each of the attribute names. Following this, using the XercesC domWriter the DOM object is converted to an XML string before being forwarded to TcdMessageSender.

The *think interval*, as supplied in the map properties, is used to set the next time to call this Think function. The default think time is 30 seconds although it is preferable to increase this value where possible to reduce the load on the engine.

State Retrieval

Both TcdTriggerMessage and TcdPollingBody must retrieve data values from Half-Life. There is no existing function as part of the standard SDK that will perform this task. An approach was adopted to implement the inverse effect that is applied by the KeyValue function, introduced in chapter 2. This can be explained as follows.

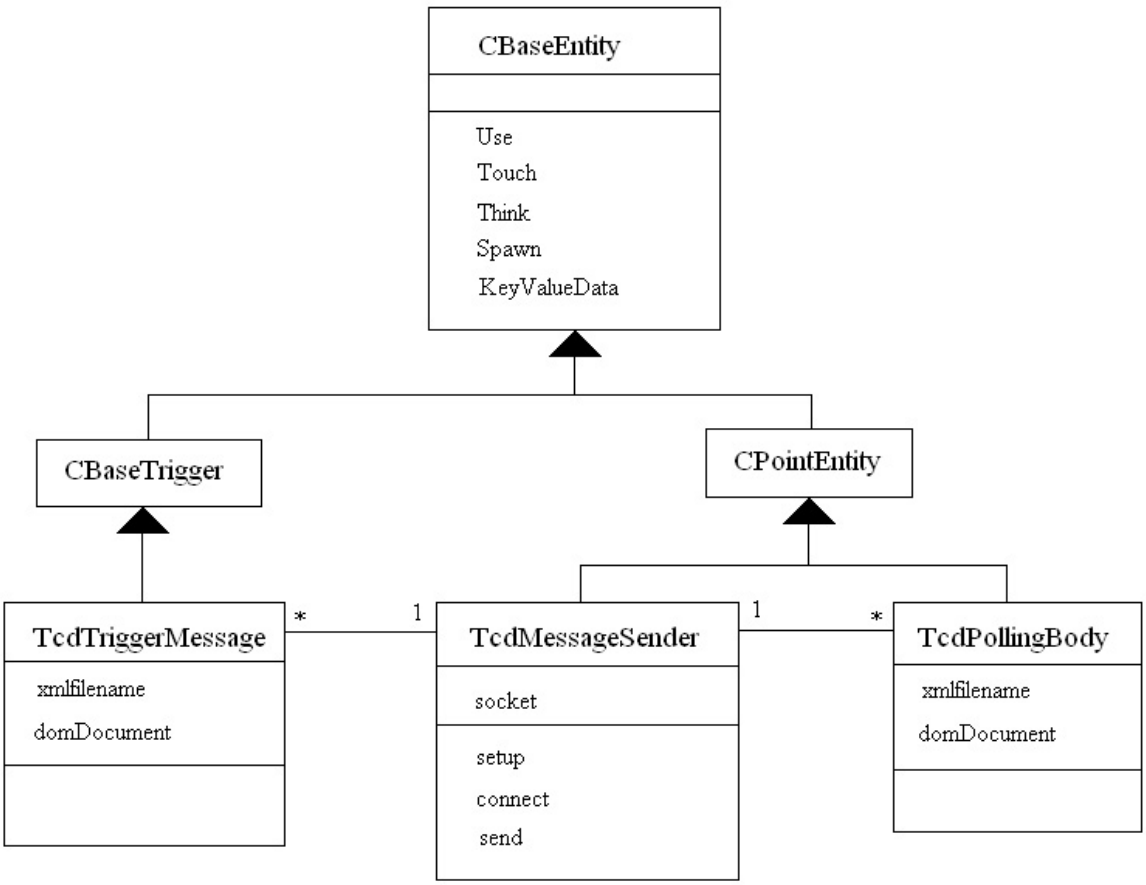

Figure 4.8 Relationships between TcdTriggerMessage, TcdMessageSender & TcdPolling-Body.

```
void TcdPollingBody::Think()
{
	DOMDocument *document = parser->getDocument();
	genMessage(document);
	XMLCh *xMessage = domWriter->writeToString(*document);
	StrX strXMessage = StrX(xMessage);
	const char *message = strXMessage.localForm();
	sender->med_send(message);
	pev->nextthink = (gpGlobals->time) + thinkInterval;
}
```

Figure 4.9 TcdPollingBody's Think function, tcd_polling_body.cpp.

```
void TcdTriggerMessage :: KeyValue( KeyValueData *pkvd )
{
	if (FStrEq(pkvd->szKeyName, "xmlfilename"))
	{
		pev->message = ALLOC_STRING(pkvd->szValue);

		const char* temp = STRING(pev->message);
		const char* temp2 = "tcdUbiSim\\xmlFiles\\";
		int len = strlen(temp)+strlen(temp2)+1;

		xmlFile = new char[len];
		strcpy(xmlFile, temp2);
		strcat(xmlFile, temp);

		ALERT(at_console, "EVENT XML FILENAME: %s \n", xmlFile);

		pkvd->fHandled = TRUE;
	}

	else
		CBaseTrigger::KeyValue( pkvd );
}
```

Figure 4.10 KeyValue function for TcdTriggerMessage, triggers.cpp.

The KeyValue function extracts the map properties from the BSP file loaded into a game of Half-Life. These properties are supplied through Hammer's properties dialog which is setup using the entity definitions stored in the FGD file. Similarly, the Message Definition Tool sources its attribute sets from the FGD file definitions. As a result, the data being retrieved by TcdTriggerMessage and TcdPollingBody corresponds to the data values that are populated using the KeyValue function.

Figure 4.10 shows the KeyValue function for TcdTriggerMessage at work. The if statement compares TcdTriggerMessage's only local variable, xmlfilename, against the KeyValueData structure that has been taken from the map information. If they match the data is saved into this variable, otherwise the function invokes the parent implementation of the same method. This can cycle all the way back to CBaseEntity's implementation of KeyValueData. CBaseEntity holds the parent implementation of this function for all classes and it is used to populate the generic data structure, pev.

Based on this information, a new function has been implemented throughout the class

```
const char* TcdTriggerMessage::getEntityData(const char *variable)
{
	if (!strcmp(variable, "xmlfilename"))
	{
		return STRING(xmlFile);
	}
	return CBaseTrigger::getEntityData(variable);
}
```

Figure 4.11 TcdTriggerMessage's getEntityData function, triggers.cpp.

hierarchy to emulate this technique. The function is called *getEntityData*. For example during an activation the TcdTriggerMessage may want to collect data about the Activator entity. TcdTriggerMessage calls the Activator's getEntityData passing the variable name as a parameter.

The getEntityData function compares the variable name to the local variables for the Activator's class. If no match is made, the getEntityData function defined for the Activator's parent class in invoked. Again this has a cyclical nature and can revert all the way to CBaseEntity level. Figure 4.11 shows the code for TcdTriggerMessage's getEntityData function.

Implementing this solution was a laborious task. However it was also a once-off task. Future entities that are added to the simulator will implement their own version of the function where required. Entities that do not define a unique set of local variables may disregard this function and the engine will automatically revert to the implementation inherited from a parent class. This is already accepted practice for mod development since it applies to each of the *Touch*, *Think*, *Use*, *Spawn* and *KeyValue* functions.

4.3.2 Java Proxy

Figure 4.12 displays the relationships between the Java classes that implement the Proxy. Chapter 3 discussed the role that each class played in transferring messages between the simulator and SUT. This section discusses some of the code implemented to support the interactions between the classes.

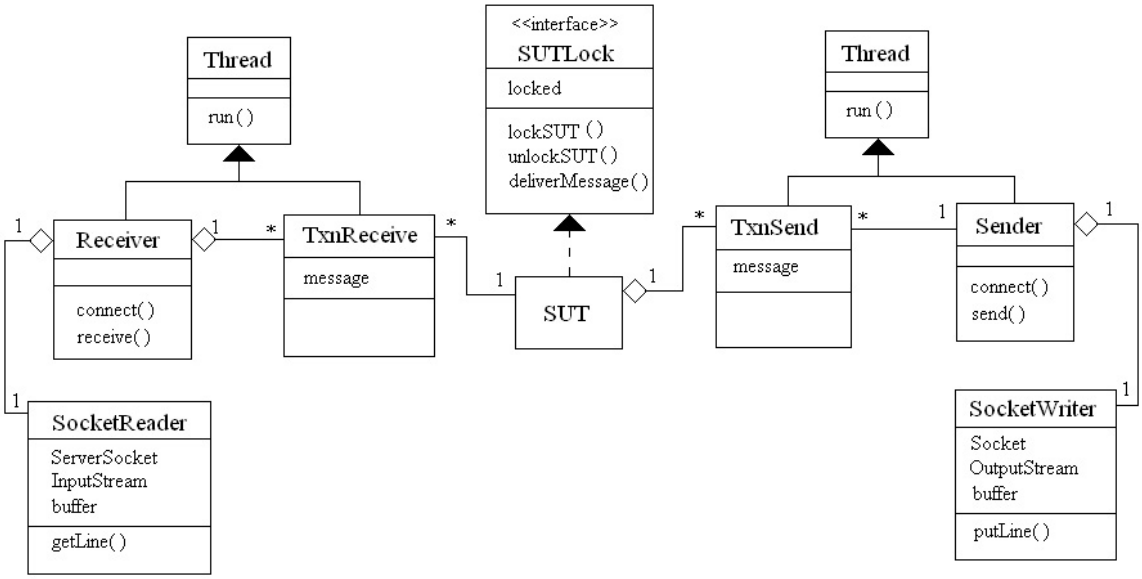

Figure 4.12 Proxy Class System Overview.

Messages Arriving at the Proxy

Three classes specifically handle messages arriving at the proxy. SocketReader houses the ServerSocket linking the two programs, Receiver manages the receipt of messages and finally TxnReceive objects are threads that take responsibility for a single message and ensure it is delivered to the SUT.

Receiver is implemented as a thread allowing it to run in parallel with the Sender object. Inside its run method it continuously invokes the SocketReader's getLine method concatenating the lines until a full message has been retrieved, shown in Figure 4.13.

When a full message has been identified, a TxnReceive thread is spawned taking the message as a parameter. At the centre of the Figure 4.12 it can be seen that the SUT must implement an interface called SUTLock. TxnReceive threads must call the lockSUT method in order to gain access to the object and hence pass on their message. When the SUT is finished processing the message it will unlock itself in preparation for the next message.

The pool of TxnReceive threads is adaptable with a view to keeping the sockets between the simulator and SUT as free as possible. It is not expected that this pool size will reach unmanageable proportions. The drawback to incorporating threads into message delivery is

```
while(true){
      nextLine = in.getLine();
      if(nextLine.equals(SocketReader.EOF)) return false;
      if(nextLine.startsWith("\0"))
      nextLine = nextLine.substring(1);
      message = message.concat(nextLine);
      if(nextLine.trim().endsWith("</message>")) break;
}
```

Figure 4.13 Code used by SocketReader to receive a message in full.

that delivery order is not guaranteed. This is not considered a problem since in a real-world network the same problem is often encountered when dealing with messages arriving at a device. However this was the motivation for time-stamping each messages as part of the DTD definition.

Messages Leaving the Proxy

In return when the SUT is ready to send an instruction into the simulator it has two options available. It can spawn a TxnSend thread, passing an XML instruction string and a reference to a sender object as the thread's parameters. This may suit SUT that is particularly busy. Alternatively, the SUT can make a direct function call to the Sender object.

Similar to the Receiver class, Sender is an extension of Thread, however its run method is implemented solely to manage the connection process. SocketReader hosts a standard Socket object and so behaves as a client. Under normal client behaviour, if the server is not up and running the client will fail. These are the circumstances under which the Proxy must create its outbound link. Catching and handling the IOExceptions generated by the failed link allows the client to keep trying with a sleep period in between attempts. When the run method has completed, the Sender object persists but its thread lifecycle is finished.

4.3.3 Map Editor

As mentioned in chapter 2, amending the map editor's *FGD* file allows a game developer to provide new game entities for inclusion in levels. To extend the FGD file a developer must know about three types of class:

```
@PointClass base(Target, Targetname) = tcd_polling_body: "TCD Polling Body"
[
      xmlfilename(string) : "XML filename"
      think_interval(integer) : "Polling Interval" : 30
]
```

Figure 4.14 FGD file entry for tcd_polling_body.

- **SolidClass:** Defines a brush based entity and its attributes.
- **PointClass:** Defines a point entity and its attributes.
- **BaseClass:** Defines a base set of attributes for reuse by either of the above classes.

In correspondence with the SDK code implementation of TcdMessageSender, TcdMessageDistributor and TcdPollingBody, PointClasses have been added to the FGD file for each. TcdTriggerMessage, on the other hand, has been included as a SolidClass. Figure 4.14 shows the FGD file entry for tcd_polling_body.

A well-formed ubiquitous computing simulator map will contain one TcdMessageSender and one TcdMessageDistributor. It will also contain the required info_player_start, the start-up location for the first-person character.

4.3.4 Message Definition Tool

The Message Definition Tool is used to generate the XML files that correspond to the Message Format DTD. As such, the interface implemented for this tool reflects the elements defined in the DTD. Chapter 3 discussed the motivation for the DTD's structure, it will not be reiterated here.

The tool's interface contains three tabbed panes, one each to define event and poll messages and a third to display the current XML contents of the document. Within the Event and Polled tabs are drop lists which contain the major message elements as taken from the DTD.

For a Poll message the major elements are:

- Class_Set
- Instance

- Targeting
- Radius
- Globals

For an Event message the major elements are:

- Activator
- Trigger
- Target
- Globals

Selecting an element from the list displays an interface tailored to the contents of the chosen element. These interfaces are put together using pev panels, class panels, global panels and text-fields.

For example in Figure 4.15, the Class_Set element for a poll message has been selected. From the Message Format DTD in Appendix C it can be seen that Class_Set elements contain both class and pev attributes. The interface in Figure 4.15 appropriately is constructed from a class panel, the uppermost panel and a pev panel, the lower panel labelled *Generic Attributes*.

Appendix G displays some further examples of the interface in use to illustrate the globals panel and text-fields.

Features of the Message Definition Tool

In order to manipulate the XML files the tool provides the standard set of file control commands provided by any editor:

- **New:** start a brand new file.
- **Open:** load an existing file.
- **Save:** save changes to current file.
- **Save As:** save current file under a new file name.
- **Close:** close the current document.
- **Exit:** exit the tool.

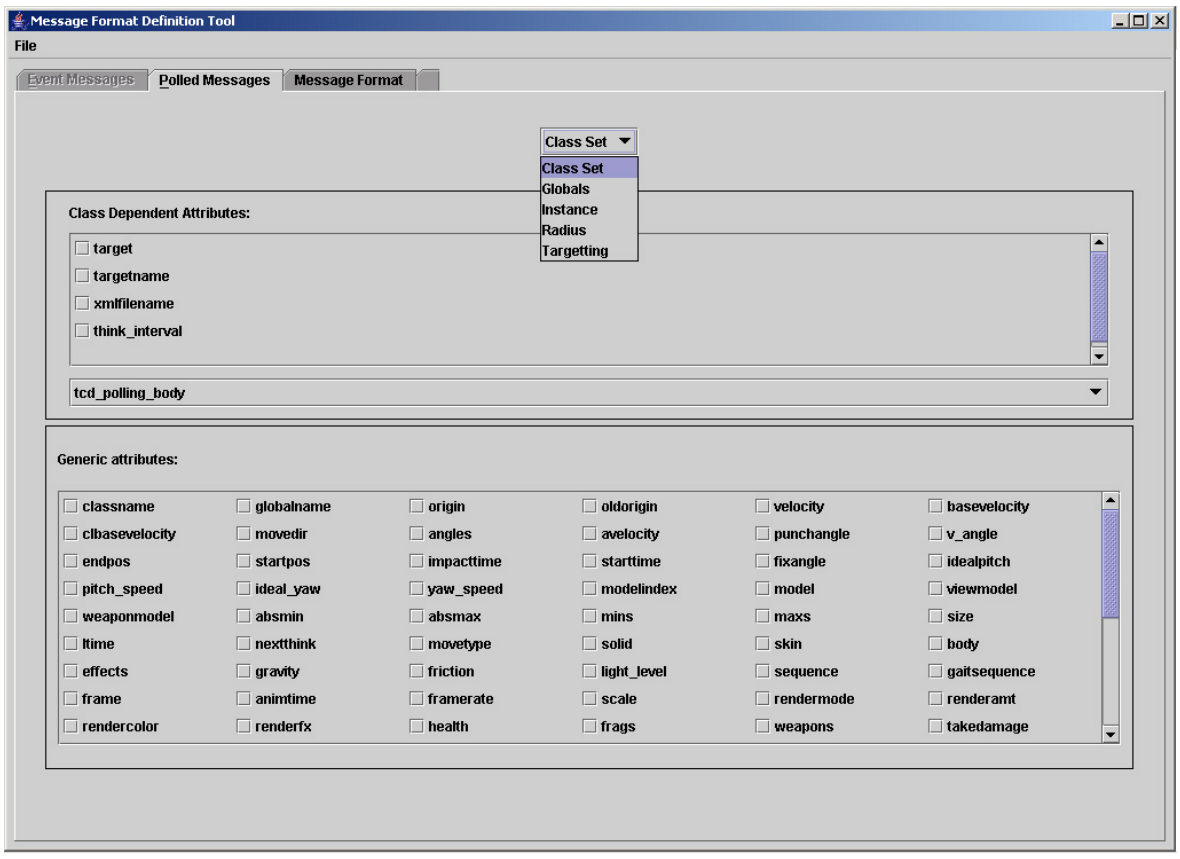

Figure 4.15 Message Definition Tool Interface, the Class_Set Interface.

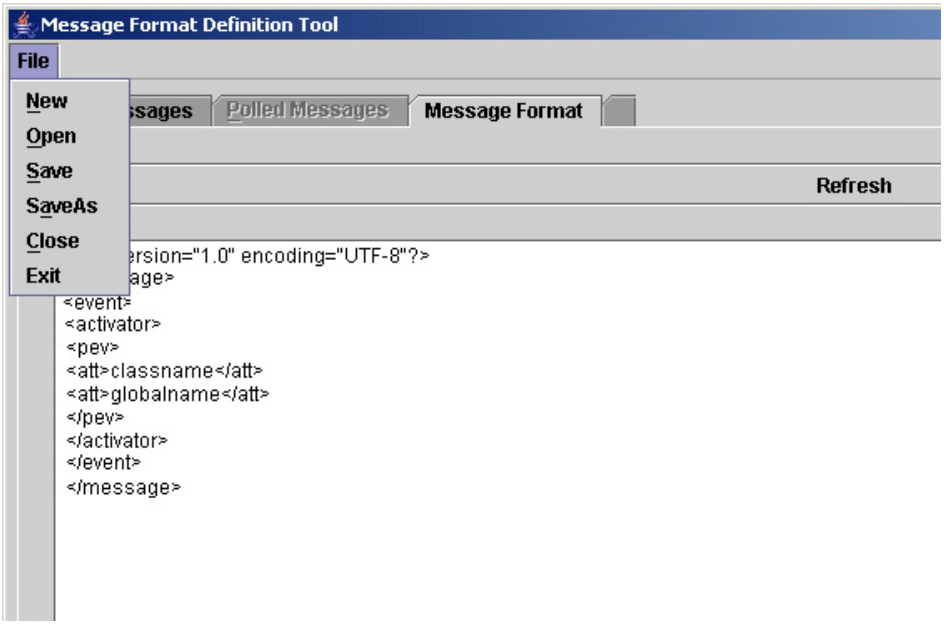

Figure 4.16 Message Format tab and file menu.

Figure 4.16 displays the tool's File menu along with its Message Format tab. The Message Format tab displays the document's current contents and can be updated using the Refresh button. Documents that are open in the tool are stored in a DOM format using the XercesJ [12] tool. This tool will parse an existing XML document into DOM format when opening a file and loading its contents into the tool. It will also generate the XML string from the DOM object when saving the file or outputting it to the display in the Message Format tab.

4.4 Conclusion

In this chapter the implementation of TATUS has been described. A high-level view of the relationships between system components was presented. Following this each of the four components were discussed in isolation, starting with the core 3D simulator, followed by the Java Proxy, the Map Editor and finally the Message Definition Tool. Chapter 5 presents results, evaluation and a discussion of TATUS.

Chapter 5

Results, Evaluation & Discussion

This chapter presents results, evaluation and a discussion of TATUS. The first result set is based on feedback from both KDEG and CIT (Cork Institute of Technology) researchers following an early demonstration of the project. The second set of results relates to the final implementation of TATUS. Feedback for this version was provided by a KDEG Ph.D. student, Tony O'Donnell. Experimentation scenarios based on Tony's work provided solid examples when extending the first prototype. These scenarios are documented in Appendix E.

The second section titled evaluation is a theoretical assessment comparing the simulator's design with the original project objectives as laid out in chapter 1. The chapter finishes with a discussion relating the design and implementation of TATUS with that of its most similar counterpart, UbiWise.

5.1 Results: TATUS v1

On the 8th July 2004, the first prototype of TATUS was presented to a group of KDEG researchers. The purpose of the presentation was to demonstrate the tool's usability from a user's perspective and so focused on running and creating experiments in the TATUS environment.

According to the test storyline presented, an audience will be sitting in a conference room waiting for a speaker to arrive and a lecture to begin. The role of the lecturer is played through the first-person game controls while the audience is made up of NPCs. When the simulation

commences, the lecturer is waiting outside the conference room door, as he approaches the room the door opens allowing him to enter.

From here the lecturer progresses to the podium, the smart environment recognises his movements and the room is switched to presentation mode, signified by dimming the lighting. During the presentation an audience member interrupts proceedings to ask a question. The room is taken out of presentation mode for the duration of the ensuing discussion. Recommencing the presentation returns the room to presentation mode. Finally, at the end of the presentation, the lecturer steps away from the podium signifying the completion of his talk. The room is restored to normal settings, the conference room door opens and the lecturer exits.

5.1.1 Demonstrating TATUS v1

To run the experiment the map shown in Figure 5.1 was created. Three couches are marked by the yellow rectangles, the heavy grey outline represents the room's walls and the green triangle is the speaker's podium. The purple lines denote triggers which register the movements of the player and NPCs. To raise a question an NPC must stand, in the process the trigger next to the middle couch is activated. Since the first-person control is already absorbed by the lecturer, a simple Java Swing interface is used to prompt the NPC to stand.

TATUS v1 preceded the communication protocol defined in the DTDs. To demonstrate state extraction, a fixed set of attributes were displayed in a Java Swing window, see Figure 5.2. The window represents the position held by the SUT in the simulator system. It also demonstrates the role that SUT plays in providing instruction to the environment. This implementation does not exactly resemble the asynchronous messaging system because TATUS v1 does not implement the message protocol that allows spontaneous instruction from the SUT client. Figure 5.3 shows the flow of messages to and from the simulator. The semantics behind the variables displayed in the SUT window are as follows:

Player Name: Name of the character touching the trigger.

Object Name: Trigger name.

Target Object: Name of the entity to be activated e.g. conference room door.

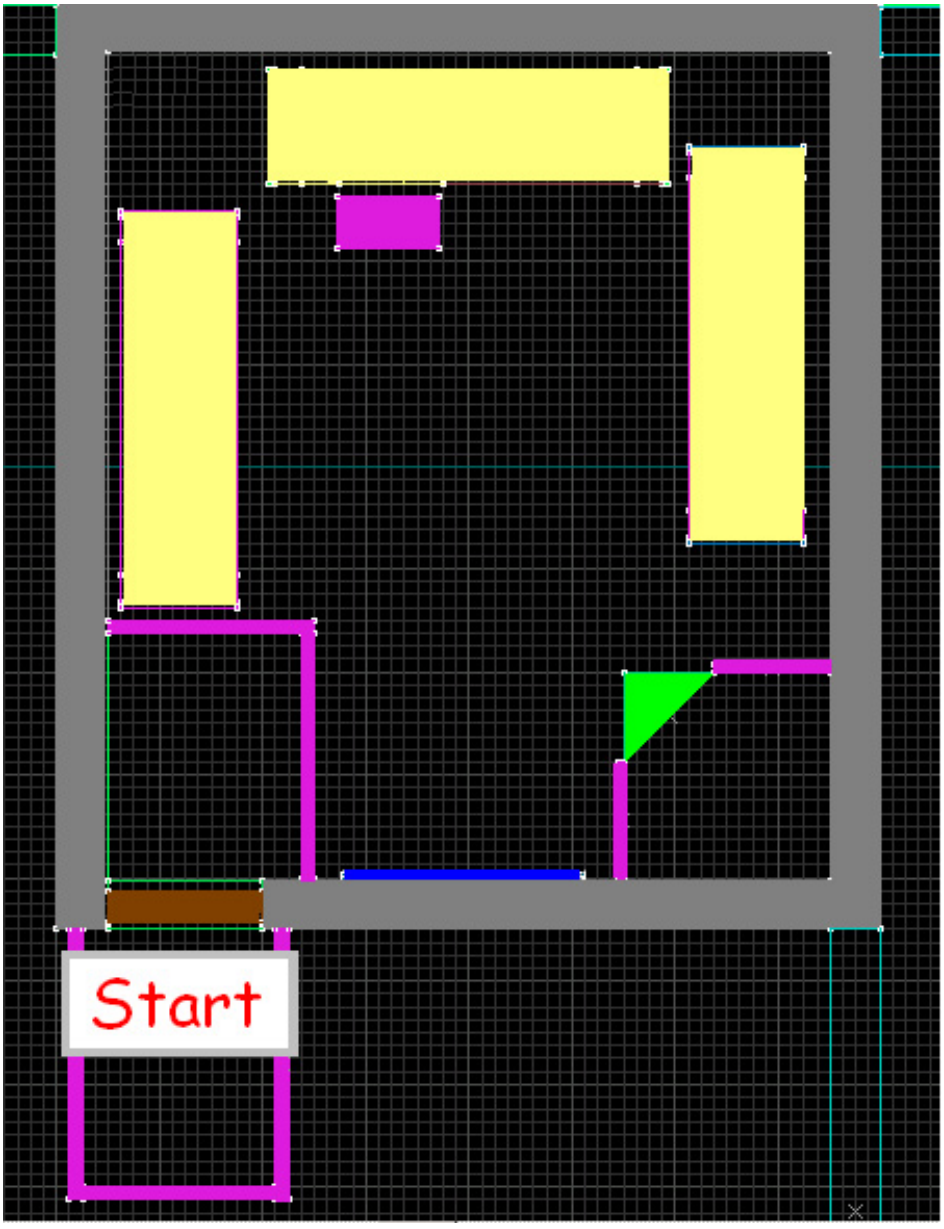

Figure 5.1 Overview of Presentation Scenario Map.

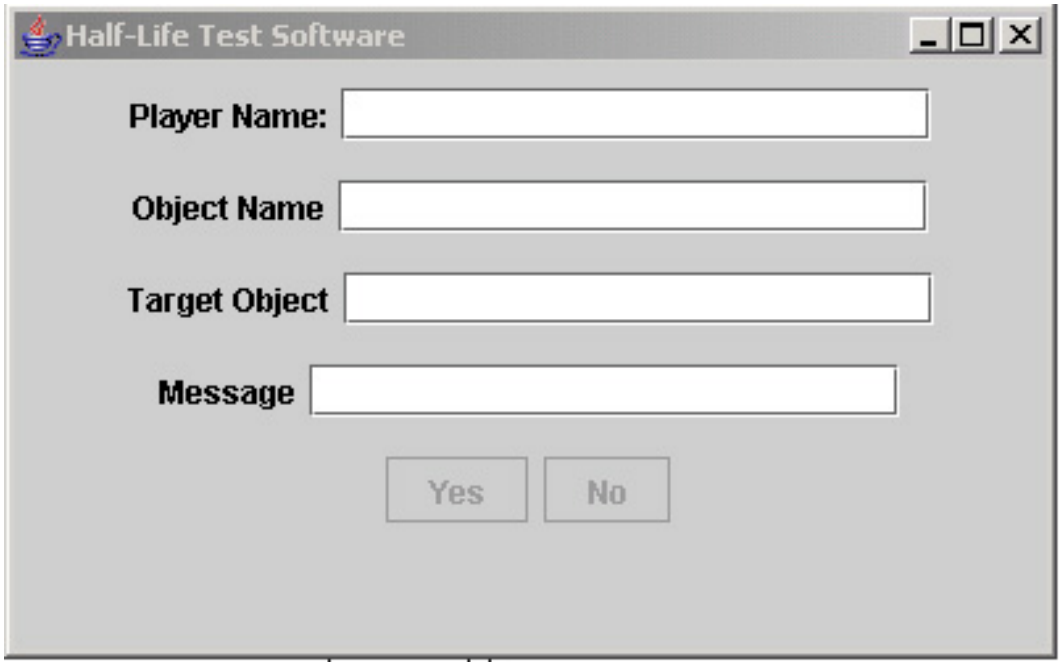

Figure 5.2 SUT place-holder.

Message: A question relating to the event e.g. Open conference room door?

5.1.2 Feedback on TATUS v1

Feedback from KDEG Researchers

KDEG researchers had two main concerns following the demonstration.

1. The first involved gaining access to Half-Life's internal data set. Both the outbound message protocol and the message definition protocols were under development but not established at this stage. Motivation for the Message Definition Tool was prompted by these concerns to provide researchers with an easy access route to define messages.

2. The researchers' second concern was a translation service between Half-Life terminology and SUT terminology. Using the string values provided through the map editor seemed a suitable solution. Half-Life already has built-in code to relate these strings to the objects and data structures it stores. On the other-hand, the researcher is responsible

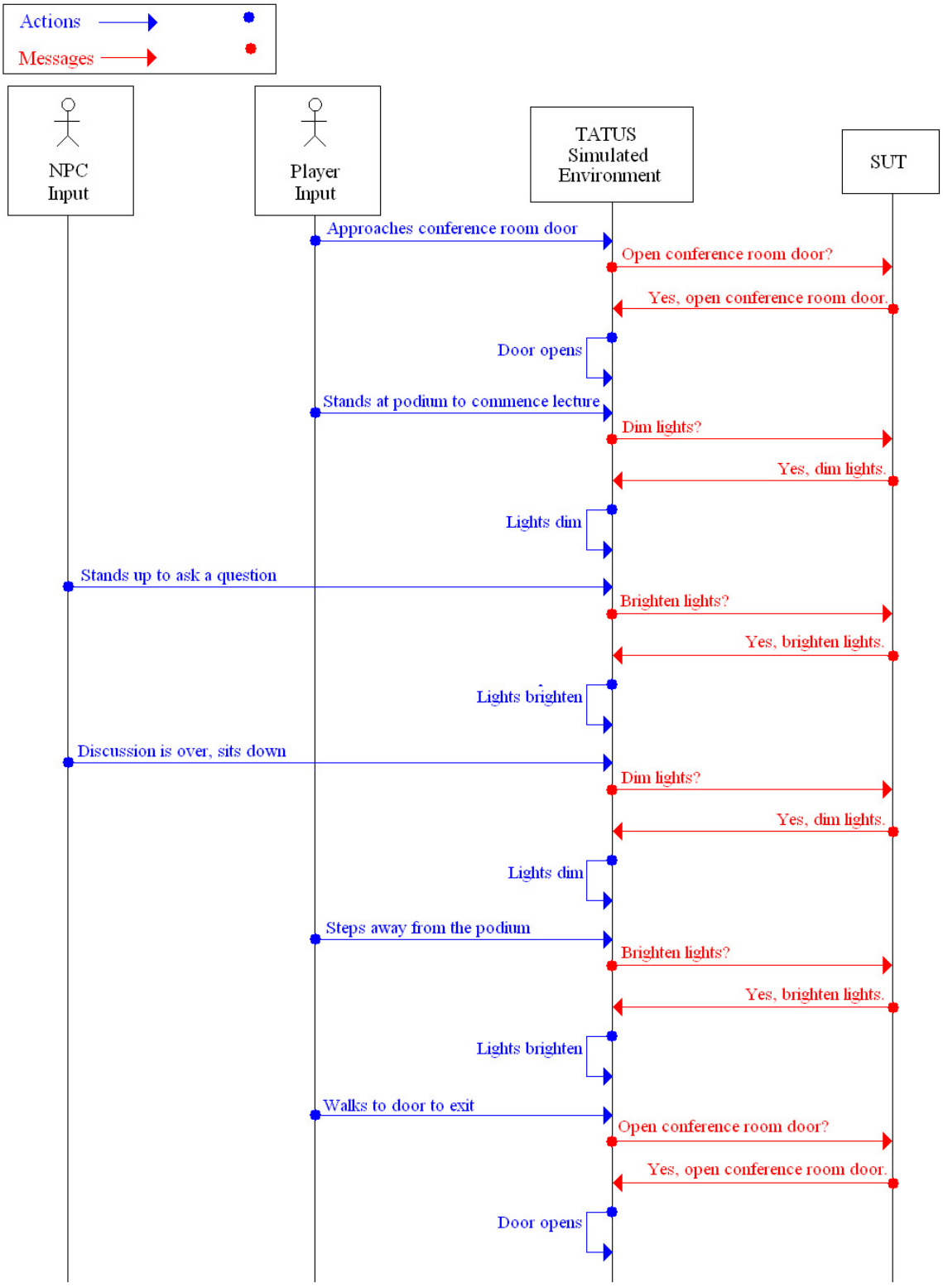

Figure 5.3 Actions driving messages between TATUS and SUT.

for populating an entity's property fields at map-creation time. As such, it is within their control to choose values best suited to the SUT.

In addition to the concerns put forward, it was proposed that the simulator also be used for designing and planning the layout of ubiquitous computing environments. It is expected that in the coming year, a room will become available to KDEG researchers for physical implementations of experiments. Prior to setting up the devices, researchers could predetermine their best positioning inside a simulated room.

Feedback from CIT Researchers

On the 15th July, a similar presentation was given at a meeting between CIT and TCD researchers. CIT are currently developing a 2D network simulator specifically aimed at modelling the communication channel characteristics in a sensor network. The meeting focused on linking the two simulators and using them to model a single environment.

By relating the results from each simulator it is hoped that a TATUS user will experience varying signal levels due to fluctuations in Bit Error Rate, Throughput and Quality of Service. CIT represent their results in SVG [10] (Scalable Vector Graphics) format. TATUS takes its view of the world from the BSP and MAP files produced by the map editor. A relationship derived between these files is the expected path forward.

5.2 Results: TATUS v2

The results presented in this section correspond to the final implementation of TATUS. To demonstrate the system's capabilities the previous scenario is reused, however a new Java Swing window clarifies the outbound and inbound messaging protocols as well as the asynchronous messaging system. In addition, the Message Definition Tool is introduced as an element of version 2.

5.2.1 Demonstrating TATUS v2

Figure 5.4 displays the improved Java GUI. The window is divided into two panels. The uppermost panel displays state information arriving from the simulator, the lower panel

accepts XML file names that list instructions to the simulator. This is a more precise imitation of SUT behaviour.

A set of drop-lists in the State Information Display Panel display the names of attributes for the Activator, Target, Trigger and Globals elements. As can be seen in the screenshot, the Trigger list has been supplied with the attributes target, targetname and classname. Selection of any of these attributes displays the corresponding value in the text-field below. This interface was designed to display message contents in the most flexible manner possible, however it has been tailored for event messages which are easier to demonstrate during a presentation than poll messages.

The Instruction Panel accepts the name of an XML file which contains a pre-defined instruction message. Prior to the presentation, a set of these files were written and saved. Each file provided instructions to open the conference room door, switch the lights on, switch the lights off or prompt the NPC audience member to stand/sit. The flexibility of the interface closely resembles asynchronous messaging.

5.2.2 Experimenting with TATUS v2

In addition to the demonstration of TATUS v2, the simulator was installed on a second machine for testing by a KDEG Ph.D. student, Tony O'Donnell. Two major achievements were realised as a result. Firstly, TATUS was successfully installed and used on a second machine in the college. Secondly, with minimal instruction and guidance, Tony has successfully created and run experiments independently using the full toolset: Message Definition Tool, Map Editor and 3D simulator. Figure 5.5 displays Tony's work using the map editor.

5.2.3 Feedback on TATUS v2

Through his experience with the TATUS, Tony O'Donnell provided feedback about the simulator. Overall, TATUS is considered a usable tool with user-friendly features such as the Map Editor and Message Definition Tool. In addition, the type and quality of information produced is considered useful to the research proposed for testing inside the environment. In response to his experience with TATUS, the following two ideas have been proposed to improve usability for the tool:

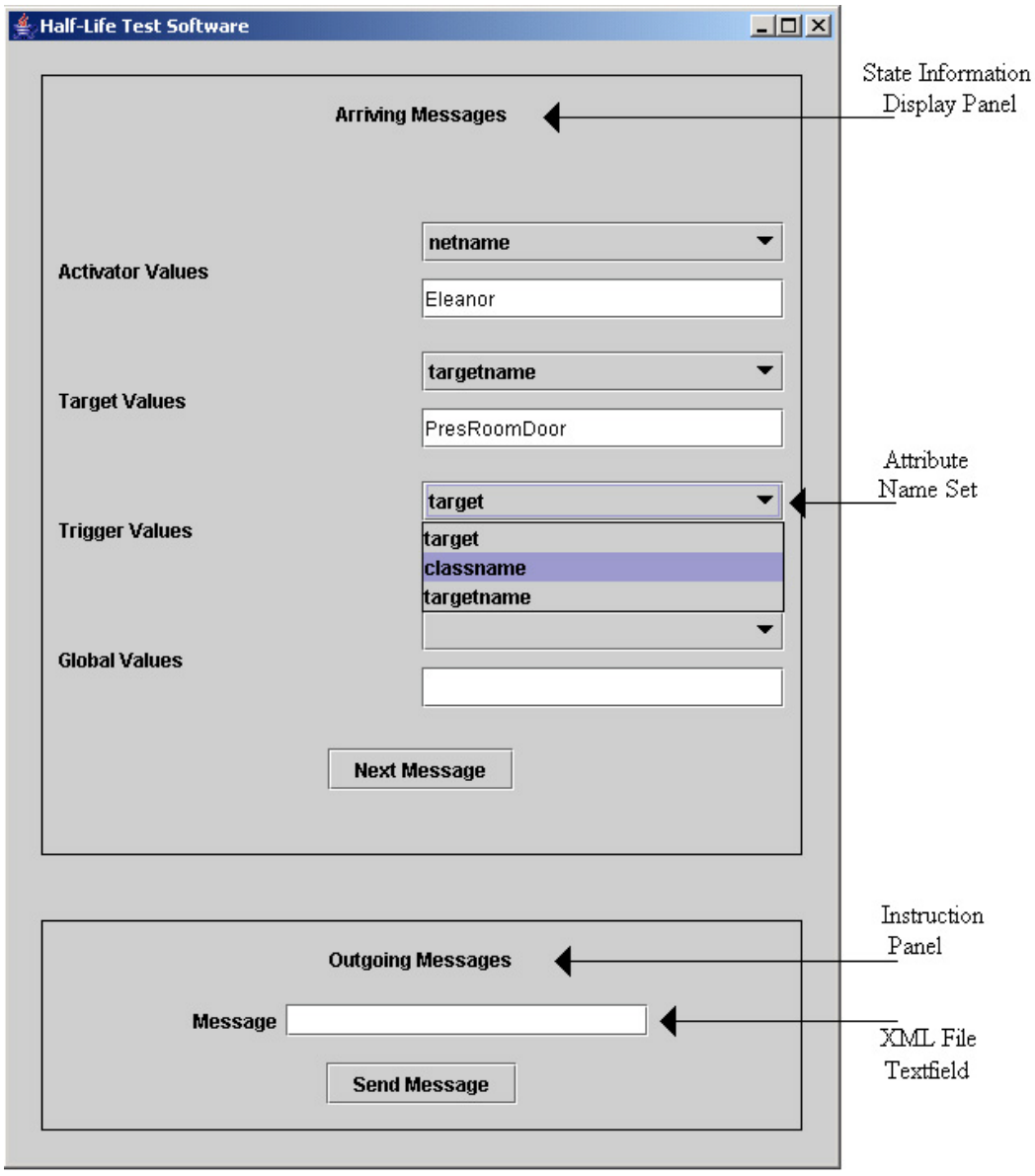

Figure 5.4 Improved Java Swing Interface.

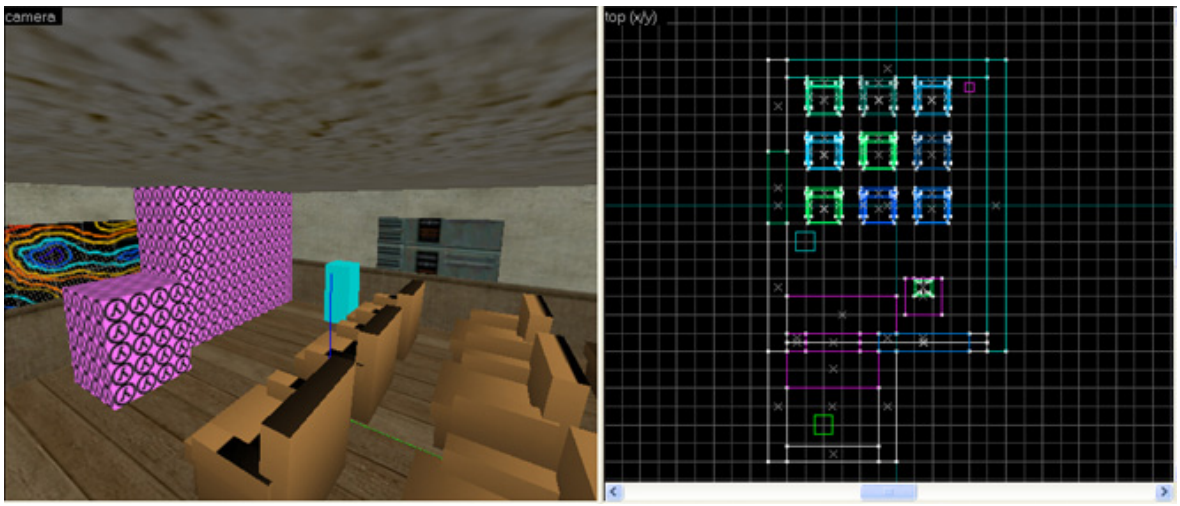

Figure 5.5 Ubiquitous computing environment created by Tony O'Donnell.

1. Currently, the start up procedure for TATUS requires the proxy to run first followed by the simulator, each invoked by a separate command. For convenience a single command incorporating both steps might be implemented.

2. The set of attributes presented by the Message Definition Tool is vast and disorderly. Providing a *basic* view that display only a subset of values would be helpful, especially for new users. The full attribute set could be part of an *advanced* option. This suggestion is similar to the proposed **Profiles for Message Definition Tool** feature presented as further work in Appendix F.

5.3 Evaluation

5.3.1 Project Objectives vs. Project Implementation

The following section compares the original objectives as set out in chapter 1, with the final deliverable presented in chapters 3 and 4. The original objective is noted in bold at the start of each item on the list below.

1. **Allow researchers to connect SUT to the simulator.** The Java Proxy interfaces the SUT to the simulator. In the process it shields the SUT from the network connection

code and, where requested, will convert the XML message protocol into Xerces Java DOM objects. Researchers must implement the Java SUTLock interface inside their own code to allow the Proxy's threads to deliver messages. The SUT uses an instance of the Proxy's Sender class to supply the environment with instructions.

2. **Allow researchers to select simulator events of interest for notification to the SUT.** Using TcdTriggerMessage entities, the researcher marks physical locations where interesting events will occur. Instances of TcdTriggerMessage forward messages when activated, so notifying the SUT about significant events.

3. **Provide researchers with an instruction protocol allowing the SUT to control actions within the simulator.** According to the Inbound Messaging DTD, the SUT can invoke the Use, Touch or Think function for any entity identifiable by a targetname.

4. **The simulator must be flexible to handle diversity in research projects and cope with multiple SUT connections in parallel.** TATUS implements both event driven and polled messaging for state extraction, providing a flexible means of gathering data. In addition, it is at the user's discretion to define the most appropriate messages for their software using the Message Definition Tool. Finally, TATUS has provisions as part of its socket implementation to accept more than one SUT as a client.

5. **The simulator must be usable. In particular, this means a straightforward intial setup procedure and an easy mechanism to configure and run tests.** The Java Proxy's API minimises the alterations required to the SUT for successful connection with the simulator. The SUT does not have to contain client and server sockets required to communicate with the simulated environment. The Message Definition Tool and Map Editor simplify the set-up procedure for experiments and are accessible, as proved by Tony O'Donnell's work to test the simulator.

6. **The simulator must realistically reflect the current state of ubiquitous computing technology e.g. pressure sensors and RFID (Radio Frequency Identification) tags. This version of the simulator is not aimed at developing new device technology, communication protocols or wireless media.** Inline with objective four, TATUS does not impose a veto on any data leaving the simulator, for

this reason some responsibility lies in the hands of the SUT not to exploit Half-Life's resources in an unreasonable manner. It was not required that network connectivity be modelled as part of the simulator, however investigative studies have begun with a view to including this feature in the future.

5.4 Discussion: A Comparison of TATUS and UbiWise

UbiWise is a device-centric simulator, it focuses on user manipulation of devices and the interactions between devices e.g. protocol support in the discovery phase. The simulator for this project on the other hand targets the intelligent technology controlling ubiquitous computing environments through the use of embedded sensors and actuators.

In both simulators, users of the systems can fill three roles, as defined in the paper UbiWise, A Ubiquitous Wireless Infrastructure Simulation Environment [2].

- User
- Researcher
- Developer

The following sections will discuss how the simulators differ with respect to each of these roles.

5.4.1 Role1: The User

A User is the person running the simulation. UbiWise offers two views to the user, the 3D physical view (game environment) and the 2-D device view (Java Swing Interface). UbiWise's developers considered immersion in the 3D physical view less important than contemplation of the overall results. A user in such an instance does all manipulation of the devices through the device view only. The 3D view is used solely to see the effects of device manipulation on the 3D environment.

The UbiWise example scenario illustrates two people standing in a ubiquitous computing environment, each carrying their own wireless device, namely a digital camera for character1 and a PDA for character2. Each character/player also has access to a shared storage space

which is represented by a picture frame on the wall. The data transferred between device and storage is an image and so each player can monitor when the information in the picture frame has changed.

Each player works from a separate client machine. Each client displays a 2D device view specific to the player as well as a view of the shared 3D space. In UbiWise's test-case there are two characters:

- Character1 carries a digital camera connected to the wireless network. User1's 3D view shows the character carrying a digital camera in the same manner as any character would carry a weapon in a first-person-shooter game. Within the 3D view the user can also see the physical environment around him including the picture frame on the wall and the other character. This user's 2D view displays all devices that the user has access to, namely the digital camera and the picture frame on the wall. Each device is contained in a sub-frame of the main window for this software.

- Character 2 carries a PDA in the place of the digital camera so views are as for client1 but replacing the camera with a PDA.

UbiWise is using the game engine to avoid attempting to generate a 3D view through Java's Swing package, a whole project in itself as they quite rightly point out. Users monitor changes to the physical world environment via the 3D view, but interact with the scenario through their 2D view. When Character1 transfers an image from the camera into storage the action essentially changes the texture of the wall within the space of the picture frame. The game engine looks after updating Character2's view of the shared storage space. In this case the player for Character2 is aware of the new data through changes that happen in the 3D view. Ideally the PDA would be aware of the change and automatically notify its user about the new content online.

By comparison, the test-case SUT for TATUS specifically targets predicting user intentions based on notifications from sensors and actuators embedded in the environment. For this it is not necessary to simulate devices or device characteristics. The general assumption when working with TATUS is that the virtual environment is adequately equipped and configured to support the experiments it runs.

Although both simulators have a 3D engine at their core which is supplemented by the use of Java Swing Interfaces, the TATUS does not intend that any of its Java interfaces are used at run-time. In contrast to UbiWise, TATUS is very much interested in allowing the user to become immersed in the 3D physical world for the duration of the experiment.

5.4.2 Role 2: The Researcher

Researchers for UbiWise and TATUS share a common work methodology:

- Both groups program SUT through Java.
- Both groups set up scenarios using a map editor.

However, this is where the similarities end. UbiWise's SUT is a 2D graphical representation of a device, mapping screen regions to Java handlers and manipulated using a mouse. SUT designs for UbiWise must be programmed directly into the WISE environment.

TATUS on the other hand allows researchers to develop Java software irrespective of any intention to connect it to the simulator. When the SUT is ready for testing, the researcher implements the SUTLock interface and includes an instance of the Proxy's Sender object so providing minimal disruption to the original code.

5.4.3 Role 3: The Developer and the Half-Life Experience

The role of the developer is consistent in both simulators. A developer is responsible for adapting the chosen game engine to improve the quality of the virtual ubiquitous computing environment. Following the experiences encountered while working with the HL SDK, it has become apparent that its complexity requires the presence of an experienced developer.

In order to tackle the SDK, a developer must at least have a good working knowledge of C/C++. The game engine exploits many of C's low-level features to maximise performance. Most KDEG researchers share a strong Java background. The initial transition away from the JVM which cushions programmers from low-level tasks such as memory management and garbage collection is frustrating and a source of aggravation.

The major issues complicating project development for TATUS stemmed mainly from the HL engine. The game is finely tuned in order to meet the high-performance demands

placed on it, ranging from screen refreshes at rates of 30 times per second combined with a network communication protocol to coordinate the 3D physical environment view on all client machines. This frequently led to clashing header files used for imported code, most notably when attempting to use the CSocket class.

Half-Life was released in 1998 with the first SDK following shortly afterwards. No version of the SDK has been officially documented to date. The release version 2.3 comes equipped with a set of ReadMe files offering a brief introduction for mod developers. In this respect the age of the SDK, works both to the advantage and disadvantage of newcomers. In the early days when the code was largely undocumented, a programmer going by the name Botman [21] spent a great deal of time working out the interactions between classes and dlls and writing down his findings. This opened up the world of Half-Life to many more game developers who were perhaps less skilled at using C/C++.

However, Botman's findings also had a negative effect on the Half-Life gaming community. As the number of mod developers increased so did the number of web pages publishing tutorials, opinions and general advice. It is common to find websites publishing inaccurate, misleading and badly-written material. Newcomers to the Half-Life scene must now trawl through both the good and bad websites to locate useful and valid information.

Further, most developers have a specific idea for a mod and hence their work is localised to a subset of the SDK classes. This means there is no single source of information available as a look-up for facts and solutions. Two useful resources are Planet Half-Life [22] and Handy Vandals Almanac [23] which index some of the best and most reliable sites available for SDK and mapping tutorials. A second hugely important resource is The Wavelength [24], which offers invaluable tutorials for building and debugging the SDK. In addition The Wavelength provides the most active HL coder's forum, where regularly an experienced programmer will be found logged on and willing to help.

5.5 Conclusion

In conclusion, the results presented in this chapter, supported by Tony O'Donnell's use of TATUS, show that a usable and flexible tool has been developed. The project objectives as laid out in chapter 1 have been satisfied. Finally the comparison between UbiWise and

TATUS confirms that this simulator contributes another dimension to 3D ubiquitous computing simulators.

Chapter 6

Conclusions & Further Work

This chapter presents ideas for future development of TATUS followed by conclusions about the design and success of the project.

6.1 Further Work

Appendix F presents proposed future development for TATUS under the following headings:

- Extension to Proxy
- Profiles for Message Definition Tool
- Network connectivity modelling
- XML file logging for Message Definition Tool
- Logging and recording experiments
- Sensor/Actuator library
- Extension to Outbound Message DTD
- Extension to Inbound Message DTD

However within this section only the first three items from the list will be discussed.

6.1.1 Extension to Proxy

The major potential development for TATUS is an extension or overlay to the Java Proxy's API, so improving data selection by incorporating a run-time filtering system. The idea

is inspired by TinyDB [13], a query processing system specifying a simple SQL-style interface for data collection. Some TinyDB features relevant to this project include metadata management, high level queries and multiple queries.

Metadata management involves cataloguing and describing the kind of data available from the game engine along with defining the semantics associated with each piece of data. Currently researchers must manually refer to documentation for this information. Using declarative style high-level queries, the researcher will have a more sophisticated method of data selection including conditions for data collection. Finally, running multiple queries in parallel will allow collection of data at variable sample rates from a subset of available sources.

6.1.2 Experiment profiles for Message Definition Tool

Part of the feedback from Tony O'Donnell's use of TATUS suggested that a reduction or categorisation of the attributes presented at this interface would provide for a more user-friendly tool. In response to this request two ideas are put forward to improve the Java window.

1. Implementation of user or experiment profiles would allow a researcher to define a particular subset of attributes to be of interest to their particular research. Researchers could choose to display one of these profiles or the full attribute set when creating/altering messages.

2. Categorisation of attributes through highlighting is another approach to breaking up the attribute set. Highlights might distinguish between string, integer or vector data. The highlighting might also denote the attribute names that are fully documented in terms of their semantic meaning.

6.1.3 Network connectivity modelling

In connection with the discussions held during the CIT research meeting, there is keen interest to incorporate some network connectivity modelling in TATUS. This will either utilise a second simulator that already models this problem. Alternatively, it may make use of some of the existing Half-Life physics such as sound patterns or blast effects. Ideally, implementing

this idea will provide developers with models for latency, disconnection and lost messages so further improving the plausibility of the experiments and results that they produce.

6.2 Conclusions

This project has realised a usable and flexible ubiquitous computing simulator for use as a test-bed for SUT. The toughest work occurred in the earlier stages of design when investigating the capabilities and facilities provided by the engine. This discovery period drove much of the design's development, for example the dual method of state extraction using triggers and polling bodies. Similarly, the DTDs are very much inline with the implementation of the engine i.e. Touch, Use, Think commands.

TATUS fulfils the objectives presented in chapter 1 as follows:

1. The Java Proxy hosts a network connection and protocol interpretation scheme in order to connect the SUT to the simulator.

2. Researchers can subscribe to event notification by using a TcdTriggerMessage entity.

3. The Inbound Messaging DTD supplies an environment instruction protocol.

4. Adaptable message content, combined with both event-driven and polled state extraction provides a flexible access route to Half-Life's data. TATUS also has built-in provisions as part of its socket control to accept more than one SUT as a client.

5. Usability has been substantially addressed through implementation of the Java Proxy, extension to the Map Editor and finally development of a Message Definition Tool.

6. In accordance with objective four, TATUS does not impose a veto on any data leaving the simulator, for this reason some responsibility lies in the hands of the SUT not to exploit Half-Life's resources in an unreasonable manner.

When working with the HL SDK, its sheer size and complexity will hinder the most experienced C/C++ programmers. It is highly recommended that a novice learns at least the basic language feature set especially where C++ deviates from other object-oriented languages such as Java. There is no quick and easy route to modifying the HL SDK but it

can be done intelligently. Identification of useful and reliable resources is essential at an early stage.

The final deliverable for this project is more than a simulator. It is a system that takes the researcher right through the stages of developing a scenario, defining the message content supplied to the SUT and finally running a test. The design behind TATUS offers huge opportunity for extending the simulator for example through an improved TinyDB style Proxy. In the long term it is believed that such simulators will play a crucial role in realising the sophisticated and visionary ideas imagined by Mark Weiser over a decade ago.

Appendix A

Glossary & Abbreviations

Term	Definition
Activator	An entity which uses another entity possibly by touching a trigger.
AI	Artificial Intelligence.
API	An Application Programming Interface provides access to a lower-level module, e.g. a virtual machine, through a set of software functions.
Brush	Solid object in a map.
BSP	Binary Space Partitioning improves screen rendering times by using a tree structure to represent geographical proximity among map entities and brushes.
BSP file	File generated by Hammer, see BSP.
Caller	An entity which invokes another's *Use* function. The caller often refers to an activated trigger.
CBaseEntity	The base class for all Half-Life entities.
Classname	The generic name for an entity in the map editor e.g. light or tcd_polling_body.
Dll	A Dynamic Linked Library is a file storing compiled and linked Windows functions aimed at a specific task.
DOM	The Document Object Model is a W3C standard specifying a common way for programs to represent XML documents as objects in a computers memory.
DOMDocument	Xerces' implementation of the W3C DOM specification.
DTD	Document Definition Type is a schema specification for XML documents defined by W3C.
Entity	Environmental effect in a map e.g. lighting.

Table 6.1 Glossary & Abbreviations

Term	Definition
Entity-brush	Combination of an entity and brush to create an object that can be activated.
FGD	Text file that supplies Hammer with a list of entities for Half-Life or a HL Mod.
FPS	First-person-shooter. A game in which the player interacts with the world through the eyes of a character.
gpGlobals	HL data structure storing the game's state.
Half-Life	FPS game released by Valve Corporation.
Hammer	Map editor for HL.
HL	see Half-Life.
HL SDK	Software development kit for Half-Life.
KDEG	Knowledge and Data Engineering Group, Trinity College Dublin.
KeyValue	HL function which loads map properties into C/C++ objects when a new level is loaded.
Macro	Set of program instructions stored in executable form.
MAP	Text file generated by Hammer representing a map in terms of classnames, targetnames and coordinates for size and location.
Mod	Game that results from reprogramming (modifying) the HL SDK.
Pev	HL data structure specifically used to store entity data.
Polling body	TATUS entity that gathers state information on a timed cycle.
Proxy	Intermediary program that interfaces a client and server especially used for protocol interpretation.
Q3A	Quake III Arena, FPS game released by id Software.
QC	Quake's variant on the C-language.
RFID	Radio Frequency Identification, analog-to-digital technology used to locate or track an item.
Scenario	Situation or experiment drawn up to test SUT.
SDK	Software Development Kit provides programmers with a software environment to develop further applications.
Spawn	HL function that creates an instance of an entity in memory.
Sprite	2D graphic that appears three-dimensional when viewed from any angle.
Storyline	see Scenario.
SUT	Software-Under-Test. Software still in development stages that will be tested using TATUS.
Targetname	String value supplied through Hammer to identify an entity.

Table 6.1 (Continued)

Term	Definition
tcd_message_distributor	TATUS entity that receives and distributes instructions from SUT.
tcd_message_sender	TATUS entity that sends state information to SUT.
tcd_message_trigger	TATUS entity that gathers state information on an event-driven basis.
tcd_polling_body	see Polling body.
Test-bed	Experimental platform to evaluate a tool's performance.
Texture	Pattern applied to a *brush*.
Think	HL function invoked on a timed cycle to give the impression of *thinking*.
Think interval	Period between invocations of an entity's Think function.
Touch	HL function invoked when two entities collide.
Trigger	HL entity used to activate events in a map.
Use	HL function invoked when an entity is used e.g. pushing a button.
W3C	World Wide Web Consortium.
Xerces	Apache project implementing the W3C *DOM* specification in Java, C++ and Perl environments.
XML	eXtensible Markup Language, a W3C specification for data and document structure.

Table 6.1 (Continued)

Appendix B

Pev & gpGlobals Data Structures

Source: `.\engine\progdefs.h`

Working Directory: Mod's source directory.

Data type	Variable Name	Variable Semantics
string_t	classname	Entity class
string_t	globalname	
vec3_t	origin	Entity's position
vec3_t	oldorigin	Entity's last position
vec3_t	velocity	Entity's velocity
vec3_t	basevelocity	
vec3_t	clbasevelocity	Velocity used for *client.dll* predictions
vec3_t	movedir	Direction of movement
vec3_t	angles	Direction the entity is facing
vec3_t	avelocity	Direction change (degrees per second)
vec3_t	punchangle	
vec3_t	v_angle	Player only, direction of view
vec3_t	endpos	
vec3_t	startpos	
float	impacttime	
float	starttime	
int	fixangle	Indicates adjustments required to *angles* value
float	idealpitch	Deviation from horizontal plane
float	pitch_speed	
float	ideal_yaw	Actual rotation around vertical axis
float	yaw_speed	Ideal rotation around vertical axis

Table 6.2 Pev variables & semantics.

Data type	Variable Name	Variable Semantics
int	modelindex	
string_t	model	
int	viewmodel	Personality of NPC/Player e.g. Scientist
int	weaponmodel	Player's weapon
vec3_t	absmin	
vec3_t	absmax	
vec3_t	mins	
vec3_t	maxs	
vec3_t	size	maxs - mins
float	ltime	
float	nextthink	Time think function will next be called
int	movetype	Movements such as walking, running, flying 0 = stationary, never moves 3 = walking 4 = monster movement with gravity 5 = monster movement without gravity i.e. fly 12 = follow entity pointed to by *aiment*
int	solid	
int	skin	
int	body	
int	effects	
float	gravity	
float	friction	
int	light_level	
int	sequence	Predefined animation sequence, NPCs are programmable.
int	gaitsequence	Player animation sequence, 0 = none
float	frame	Playback position in animation sequence
float	animtime	Time when frame was set.
float	framerate	Animation playback rate (-8x to 8x)
float	scale	Sprite rendering scale
int	rendermode	
float	renderamt	
vec3_t	rendercolor	
int	renderfx	
float	health	Health of an entity, 0 = dead
float	frags	
int	weapons	
float	takedamage	
int	deadflag	
vec3_t	view_ofs	Eye position

Table 6.2 (Continued)

Data type	Variable Name	Variable Semantics
int	button	
int	impulse	
int	spawnflags	
int	flags	
int	colormap	
int	team	
float	max_health	
float	teleport_time	
float	armortype	
float	armorvalue	
int	waterlevel	
int	watertype	
string_t	target	Entity to target e.g. for triggers
string_t	targetname	Name of this entity, as supplied in map editor
string_t	netname	Name of player, supplied in game config file
string_t	message	String supplied from map editor
float	dmg_take	
float	dmg_save	
float	dmg	
float	dmgtime	
string_t	noise	
string_t	noise1	
string_t	noise2	
string_t	noise3	
float	speed	
float	air_finished	
float	pain_finished	
float	radsuit_finished	
int	playerclass	
float	maxspeed	
float	fov	Field of View
int	weaponanim	
int	pushmsec	
int	bInDuck	
int	flTimeStepSound	
int	flSwimTime	
int	flDuckTime	
int	iStepLeft	
float	flFallVelocity	
int	gamestate	
int	oldbuttons	
int	groupinfo	

Table 6.2 (Continued)

Data type	Variable Name	Variable Semantics
float	time	Clock maintained by game engine
float	frametime	
float	force_retouch	
string_t	mapname	Name of current level
string_t	startspot	
float	deathmatch	
float	coop	
float	teamplay	
float	serverflags	
float	found_secrets	
vec3_t	v_forward	
vec3_t	v_up	
vec3_t	v_right	
float	trace_allsolid	
float	trace_startsolid	
float	trace_fraction	
vec3_t	trace_endpos	
vec3_t	trace_plane_normal	
float	trace_plane_dist	
float	trace_inopen	
float	trace_inwater	
int	trace_hitgroup	
int	trace_flags	
int	msg_entity	
int	cdAudioTrack	
int	maxClients	Max number of clients that can connect to Server
int	maxEntities	
const char	pStringBase	
void	pSaveData	
vec3_t	vecLandmarkOffset	

Table 6.3 gpGlobals variables & semantics.

Appendix C

Inbound, Outbound & Message Format DTDs

Inbound Message DTD

```
<!--This DTD defines the building blocks used to
construct a message inbound to the Simulator-->
<?xml version="1.0"?>
<!ELEMENT message (touch | use | think)*>
<!ELEMENT touch (name, activator)>
<!ELEMENT use (name, activator, caller, usetype)>
<!ELEMENT think (name)>

<!ELEMENT name (#PCDATA)>
<!ELEMENT activator (#PCDATA)>
<!ELEMENT caller (#PCDATA)>
<!ELEMENT usetype (on | off | toggle | set)>

<!ELEMENT on EMPTY>
<!ELEMENT off EMPTY>
<!ELEMENT toggle EMPTY>
<!ELEMENT set EMPTY>
```

Outbound Message DTD

```
<!--This DTD defines the building blocks used to construct
a message outbound from the Simulator-->
<?xml version="1.0"?>
<!ELEMENT message (time, (event | poll)) >
<!ELEMENT poll ( instance?, class_set?, targeting?, radius?, globals?)>
<!ELEMENT event (activator?, trigger?, target?, globals?)

<!ELEMENT instance ( name, pev )>
<!ELEMENT class_set ( name, pev, class )>
<!ELEMENT targeting ( name, pev)>
<!ELEMENT radius ( name, distance, pev )>

<!ELEMENT activator ( name, pev )>
<!ELEMENT trigger ( name, pev )>
<!ELEMENT target ( name, pev )>

<!ELEMENT globals (att)*>
<!ELEMENT pev (att)*>
<!ELEMENT class (att)*>

<!ELEMENT att (name, val)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT val (#PCDATA)>
<!ELEMENT time (#PCDATA)>
```

Message Format DTD

```
<!--This DTD defines the building blocks used to construct a message format-->
<?xml version="1.0"?>
<!ELEMENT format (event | poll) >
<!ELEMENT poll ( instance?, class_set?, targeting?, radius?, globals?)>
<!ELEMENT event (activator?, trigger?, target?, globals?)>

<!ELEMENT instance (name, pev) >
<!ELEMENT class_set (name, class, pev)*>
<!ELEMENT targeting (name, pev)*>
<!ELEMENT radius (name, distance, pev)*>

<!ELEMENT activator (Att)*>
<!ELEMENT trigger (Att)* >
<!ELEMENT target (Att)* >

<!ELEMENT globals (att)*>
<!ELEMENT pev (att)*>
<!ELEMENT class (att)*>

<!ELEMENT att (#PCDATA)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT distance(#PCDATA)>
```

Appendix D

KDEG Workshop Review

This document reviews the KDEG workshop that took place in January 2004.

Introduction

A high level aim of the simulator will focus on topics/experiments that are unrealistic to implement in the real world for either cost or logistical reasons. This document is the first draft of candidate features for the ubiquitous computing simulator. The document is divided into three sections. Section 1 focuses on input to the simulator from the real-world. Section 2 focus on extracting state from the virtual environment. Section 3 looks at useful features and possible adaptations for Half-Life to be incorporated as part of a realistic virtual ubiquitous computing environment.

Input to Simulator

This section lists the current set of inputs to the simulator, namely environment rules, user controls, real-world influence and events.

User Controls

User input is limited to keyboard and mouse commands used by a player to navigate around a map. From the findings published by the UbiWise [2] development team, it is known that modifying weapons in a FPS game to become useable ubiquitous computing devices is a difficult result to achieve. Here we propose that external real-world devices are interfaced to

the simulator to provide an extra set of controls in addition to the keyboard and mouse. This is also with a view to incorporating natural user reaction, and avoiding premeditated input to wireless handheld devices.

Real-World Influence

Real-world influence allow the output generated by a service operating in the real-world to influence the state of the virtual world. The simulator will need to understand the behaviour of such services in order to benefit from the input that it receives. An interface will be required to allow researchers to develop new services for test in the simulated environment. Commercial technologies/protocols such SOAP/WSDL, JXTA, Jini could be implemented as an option that a developer could turn on/off.

Events

The researcher should be able to trigger an event at any given moment e.g. a device failure or incoming message.

Extraction of Useful Information

State Extraction

The following information has been noted as interesting to current projects within the group:

- Location and velocity of devices.
- Location, velocity and stance of a character.
- State of fixed objects e.g. closed/open door
- Device occupancy

Logging

Maintaining a history of significant events will allow a particular scenario to be replayed under a particular set of conditions. A researcher may want to adjust these settings based on the outcome of previous tests i.e. change the state of the environment for future experiments in order to produce a more useful or interesting test result.

Provide stimulus to SUT

It should be possible to extract the state of the virtual world to provide input to software in development stages with a view to exploiting/testing features of the code. This is with a view to debugging and investigation of future potential for the SUT.

The Environment

The environment should focus on the interfaces between devices and software in the real and virtual worlds. Users and software in the real-world will be able to manipulate devices and objects in the virtual world to change the state of the virtual world. Software in the real-world can model its own view of the people and devices in the virtual world.

Character Interaction

The aim is to capture a character's intention according to his/her interaction with other characters and the general surroundings. Movement, velocity, gesture and gait are all indicative of a person's objective. Capturability is a major issue. Characters and event sequences should progress in as natural a manner as possible by removing as much preplanned movement as possible.

Ideas for capturability:

- **Gaze:** The orientation of a character indicates the direction that they are looking.
- **Gesture:** Use of existing character movement to imitate normal behaviour e.g. pointing by raising/lowering an invisible weapon.

Characters, other than that controlled by the researcher are needed not only for the researcher to interact with but also to occupy devices/services that are available in the world. The challenge is to make these NPCs behave in a normal/natural manner to produce a real-world effect.

Scenario: Calling a meeting should prompt all characters involved to go to the venue.

Issues:

- Natural movement patterns
- Programmable behavioural patterns
- Interaction between people/characters
- Interaction of people with environment/services/devices
- Multi-player vs NPCs

Wireless Media & Network Connectivity

The issues to be emulated:

- Intermittency and how well a particular service copes with varying levels.
- Packet loss models e.g. according to radio shadows.
- Parallel wireless links e.g. WLAN, Bluetooth, infrared running in the same room.
- Provide controls over the above three issues for the researcher.

It is expected that modification of game physics will be required to improve the quality of the simulated ubiquitous computing environment in this respect. There is potential for mapping network connectivity onto existing game physics such as sound or blast effects.

Appendix E

Test Scenarios

The following scenarios are supplied courtesy of Tony O'Donnell, Ph.D. student (KDEG), and are specifically tailored to his current research. In chapter 5 there are results presented based on Tony O'Donnell's use of TATUS.

Scenario 1: Presentation

Outline: This scenario covers an undergraduate project report presentation session.
Characters: Sheila (lecturer), group leaders for the 4 project groups, an audience of second year economics students
Equipment/Sensors: Multiple video/audio sensors, projector & screen

Policies

- The lecturer stands at the front of the theatre
- Presenters do so from the lectern
- Presenters have control of the projector subject to intervention by the lecturer
- When a presentation is underway, the lights dim
- Audience members may ask questions by raising their arms
- Presentations are relayed over the Internet
- Presenters should wait their turn by standing at the front of the class
- Presentations last 10 minutes

Presentation storyline

It is the end of term and the second year economics class have just completed their major term project. Each group must present their project to the class at a session where their classmates can challenge their report's findings. The class files into the lecture theatre, with the bulk taking their seats, while the presenters make their way down to the front. Sheila, the course lecturer has already decided the order of speakers and alerted the system to this order. As a result, the slides for the first group have been loaded when their presenter makes her way to the lectern. From time to time, members of the audience raise their hand to ask questions. The presenter signals that they will take the question by pointing at the student concerned. Once a question has been allowed, a directional microphone directs itself at the questioner, and their voice is replayed over the pa system, and the net.

When a speaker approaches the end of their allotted time, the system discretely alerts them via the lectern terminal. When the time has fully elapsed the lecturer can allow additional time with a suitable phrase, but failing that, the lectern microphone is disconnected and the student makes their way back to their seat.

At the end of the presentations, they are placed online and the class are forwarded a link to their location as well as that of the recording of the session.

Scenario 2: Meeting

Outline: This scenario covers a research project meeting. The meeting takes place within a ubicomp-friendly meeting room.
Characters: John (Research Director), Anne, Peter, Frank, Jim, Sandra
Equipment/Sensors: Multiple video/audio sensors, projector & screen, electronic whiteboard, terminals

Policies

- The RD always sits at the top of the table
- Only one person may control the projector at a time
- Control of the projector is claimed by standing near the lectern

- Multiple users may interact with the whiteboard simultaneously subject to restrictions enforced by the RD
- The controller of the projector has temporary control of the meeting and can decide who can speak, with the exception of the RD
- When a presentation is underway, the lights dim

Meeting storyLine

It is Tuesday afternoon and the Systems Research Group arrives into the seminar room for their weekly meeting. John, the group's leader, has already logged into the room management system in order to configure it for today's meeting. This included equipping the room's phone system with international dialling capabilities for this afternoon's conference call, ensuring that a live stream be delivered to Jim as he will be attending remotely, advising the room's management system of the group members expected to attend in order to load profiles and mount their personal filestorage. He also requests that the meeting be recorded.

The group members enter the room and John makes his way to his usual seat at the top of the table. The other members take seats around the meeting table. Once everyone is seated the meeting gets underway with a review of the previous weeks action plan. The system displays a copy of the previous week's plan on each terminal, and as points on it are dealt with, it updates the plan in order to forward an update to the group's inboxes after the meeting.

The main parts of today's meeting are a presentation from Sandra and a discussion of revisions of the current prototype the group are working on. John calls on Sandra and she makes her way up to the lectern. The system loads Sandra's presentation from her filestorage and when she indicates she's ready to begin, the lights dim, the presentation's first slide appears and the recording system focuses on Sandra and stops recording interventions from other attendees.

Sandra begins her presentation, however half-way through the fourth slide Peter raises his hand. Sandra's terminal alerts her to the question and she pauses to allow the question. The system, recognising that she is allowing the question, begins to record Peter. After dealing with the question, Sandra proceeds with her presentation, answering some further questions. When she completes the final slide the lights come back on and she asks for further questions.

About ten minutes later, John realises that time is passing and interrupts to bring Sandra's presentation to an end. The system acknowledges John's role and allows this interruption. The meeting now moves onto a discussion of the group's current prototype.

Jim is the lead designer of the prototype, and he asks for an image of the current design to be displayed on the whiteboard. The same image is relayed to Jim's terminal. He gives a brief report of the prototype's progress.

After he finishes, Anne walks up to the whiteboard and suggests some design changes. Sandra points out flaws in two of them but the others seem like possibilities. A discussion of these then ensues and after some more minor alterations, an updated design is saved from the whiteboard.

The last main piece of business is a conference call with the group's partner in Amsterdam. Having previously equipped the room with international dialling capabilities John puts the call through and the system monitors the group for contributions. At the end of the call it is agreed that a further call will be necessary. John asks the system to suggest some suitable times, which it does by examining the members' diaries. Once a suitable time is agreed the call ends, as does the meeting.

The system places a copy of the recording of the meeting online and emails the attendees with a link to the recording.

Appendix F: Further Work

Appendix F presents proposed ideas for future development in the following areas:

- Extension to Proxy
- Profiles for Message Definition Tool
- XML file logging for Message Definition Tool
- Network connectivity modelling
- Logging & recording experiments
- Sensor/Actuator library
- Extension to Outbound Message DTD
- Extension to Inbound Message DTD

Extension to Proxy

The major potential development for TATUS is an extension or overlay to the Java Proxy's API so improving data selection data by incorporating a run-time filtering system. The idea is inspired by TinyDB [13], a query processing system specifying a simple SQL-style interface for data collection. Some TinyDB features relevant to this project include metadata management, high level queries and multiple queries.

Metadata management involves cataloguing and describing the kind of data available from the game engine along with defining the semantics associated with each piece of data. Currently researchers must manually refer to documentation for this information. Using declarative style high-level queries, the researcher will have a more sophisticated method of data selection to include conditions for data collection. Finally, running multiple queries in parallel allows collection of data at variable sample rates from a subset of the available sources.

Profiles for Message Definition Tool

Part of the feedback from Tony O'Donnell's use of TATUS suggested that a reduction or categorisation of the attributes presented at this interface would provide for a more user-friendly tool. In response to this request two ideas are put forward to improve the Java window.

1. Implementation of user or experiment profiles would allow a researcher to define a particular subset of the attributes to be of interest to their particular research. Researchers could choose to display one of these profiles or the full attribute set when creating/altering messages.

2. Categorisation of attributes through highlighting is another approach to breaking up the attribute set. Highlights might distinguish between string, integer or vector data. It might also denote the attribute names that are documented in terms of their semantics.

Network connectivity modelling

In connection with the discussions held during the CIT research meeting, there is keen interest to incorporate some network connectivity modelling in TATUS. This may utilise a second simulator that already supports modelling this problem. Alternatively, it may make use of some of the existing Half-Life physics such as sound patterns or blast effects. Ideally, implementing this idea will provide developers with models for latency, disconnection and lost messages, so further improving the quality of the experiments and results produced by researchers.

XML file logging for Message Definition Tool

As was previously mentioned, all usable XML files must be saved to Half-Life's working directory under the path `".\tcdUbiSim\xmlFiles\"`. This logging system would save a filename, file and file description to remind the user about the original intentions when creating the format file. A text field added to the Message Definition Tool would allow the researcher the opportunity to supply some comment by way of a file description.

Logging & Recording Experiments

Although game recording is provided in the standard game of Half-Life the feature has not yet been exploited for the simulator. However, recording and logging experiments is very important when reviewing results, applying changes to the software or designing future experiments. Ideally, a researcher will have access to these results possibly with some sort of cataloguing system for easy reference.

Sensor/Actuator Library

Not part of the project requirements, ubiquitous computing devices have not been added to the simulator. This extra resource, in the form of a sensor/actuator library, will further tailor the simulator towards becoming a true representation of a ubiquitous computing environment. Its purpose will be to provide researchers with devices they can embed in the environment and alter through parameters for use in experiments.

Extension to outbound DTD

The current format for attribute elements leaving the simulator is a name-value pair. There is no type information to allow the SUT to automatically interpret arriving data, it must know in advance the type of data to expect. This extension will introduce a data-type field as part of the basic attribute element.

Extension to Inbound DTD

This extension will introduce two new instruction types to the inbound messaging DTD. A poll instruction will allow the SUT to request specific data about a named entity on demand. The set instruction will allow the SUT to change variable values for a named entity.

Appendix G: TATUS Screenshots

Figures 6.1 - 6.5 follow the sequence of images corresponding to the scenario discussed in chapter 5 in relation to demonstrating both TATUS v1 and v2. Following these images are screenshots for both Hammer, the map editor, and the Message Definition Tool.

Figure 6.1 Lecturer waits outside the conference room.

Figure 6.2 Audience wait for lecture to commence.

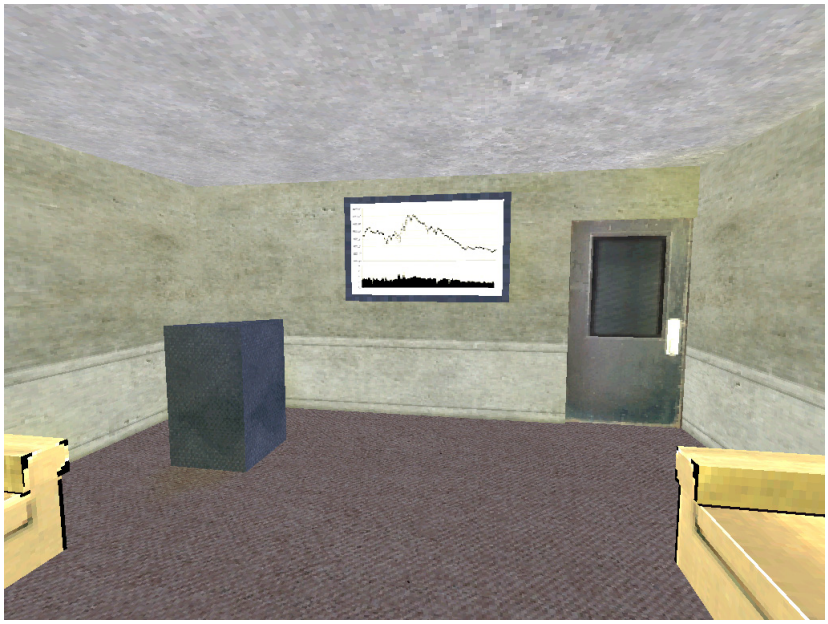

Figure 6.3 View from audience.

Figure 6.4 Lights dim for presentation to commence.

Figure 6.5 Audience member stands to ask a question.

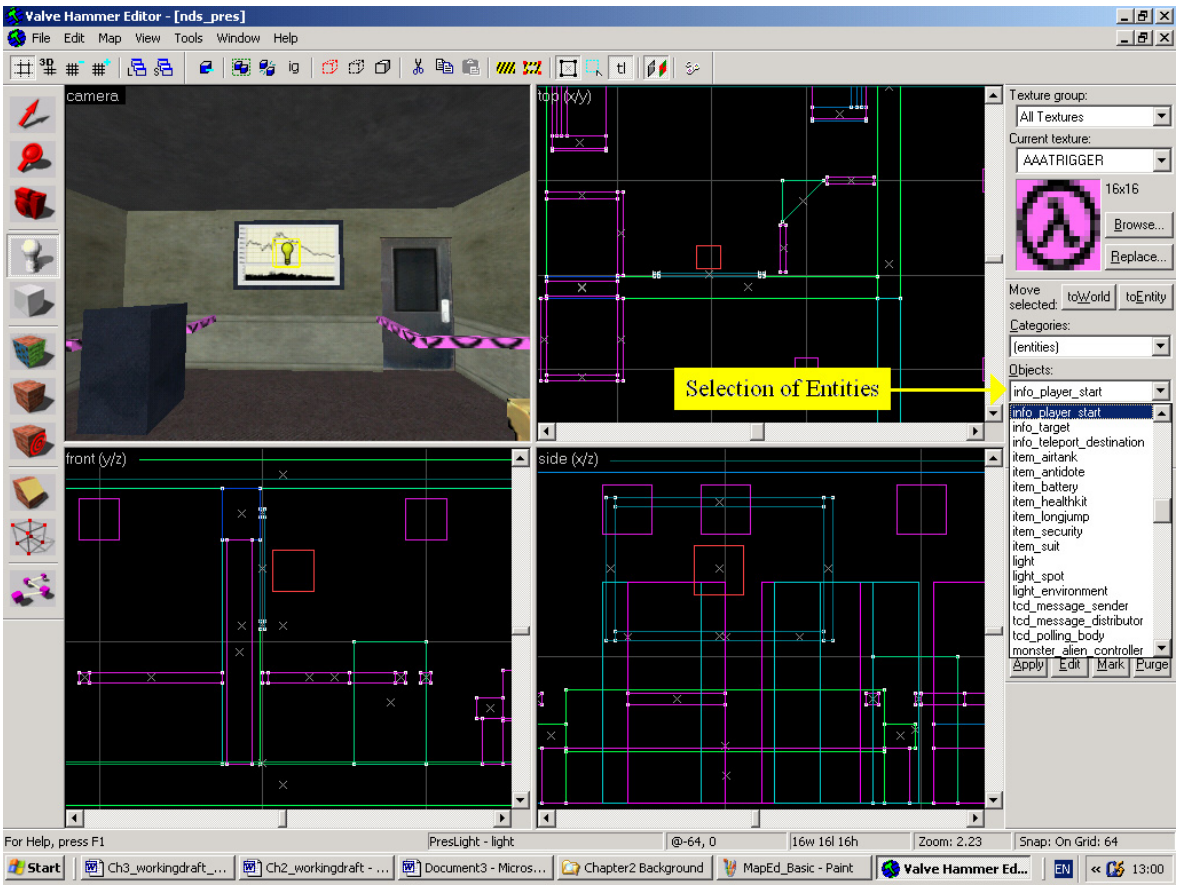

Figure 6.6 Standard Hammer Interface.

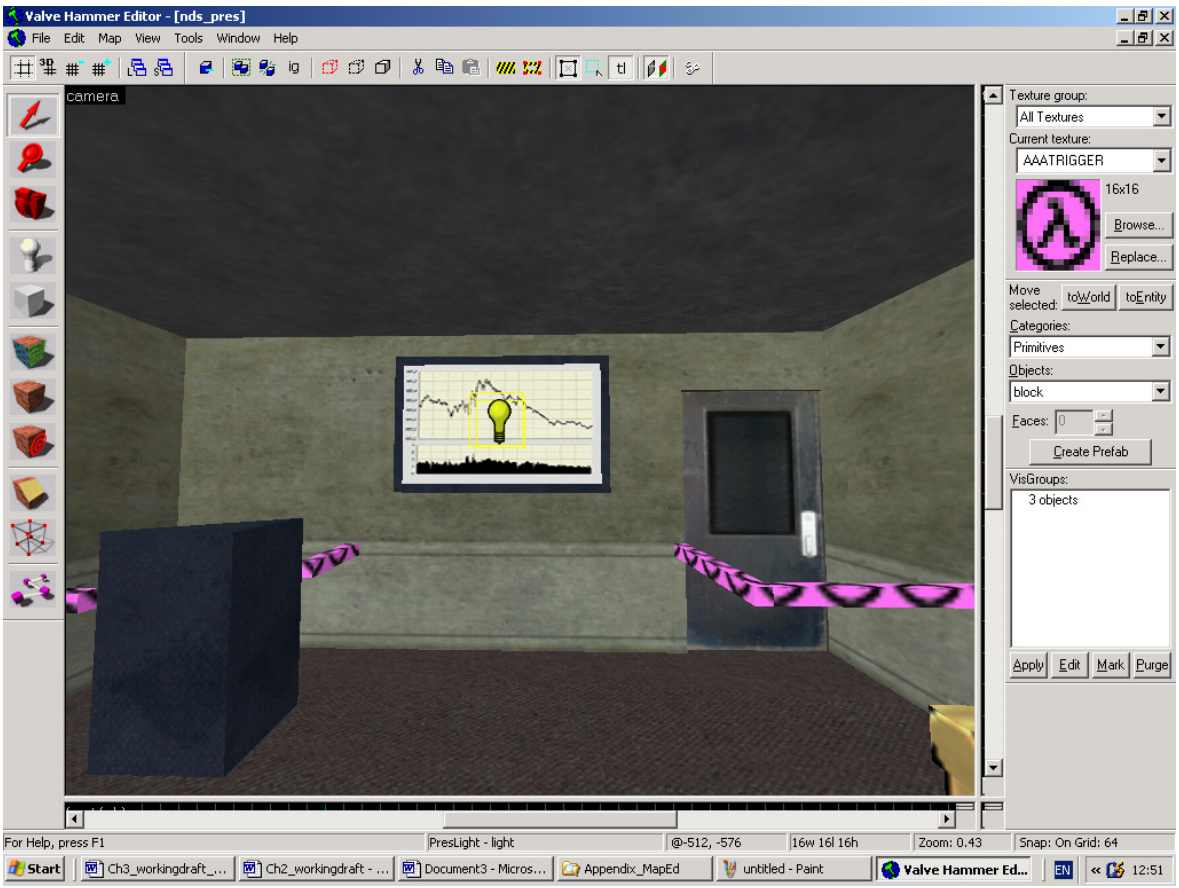

Figure 6.7 Map under development in Hammer.

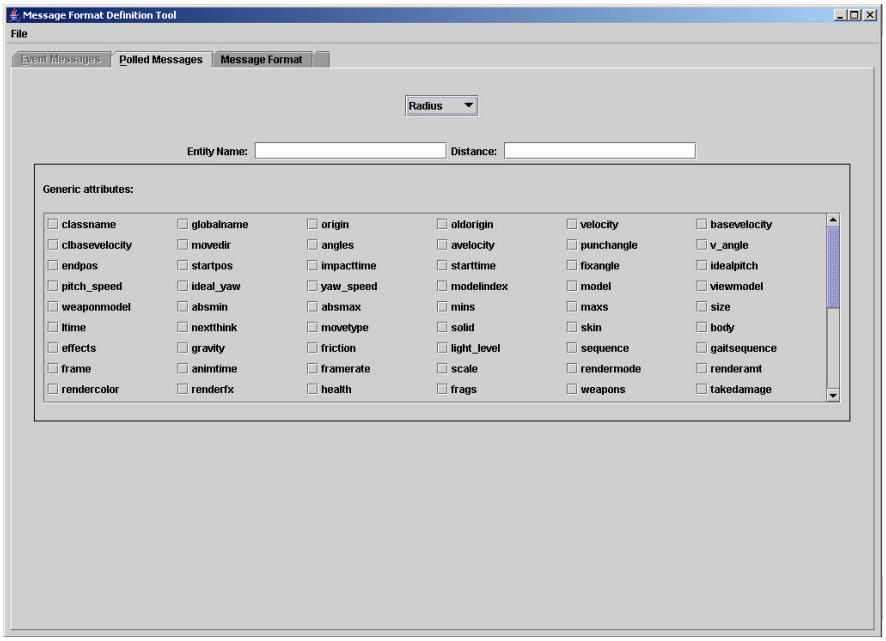

Figure 6.8 Poll message, radius attributes.

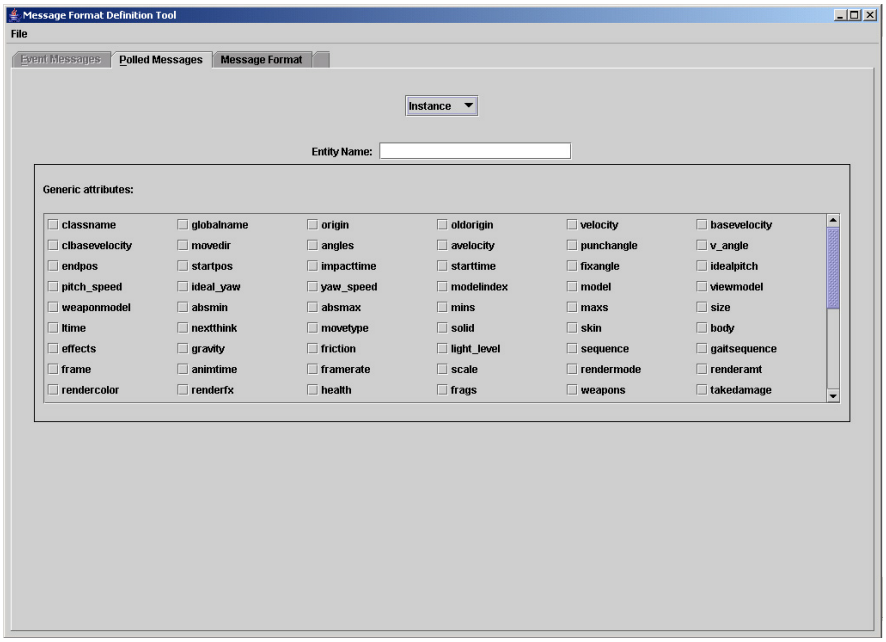

Figure 6.9 Poll message, instance attributes.

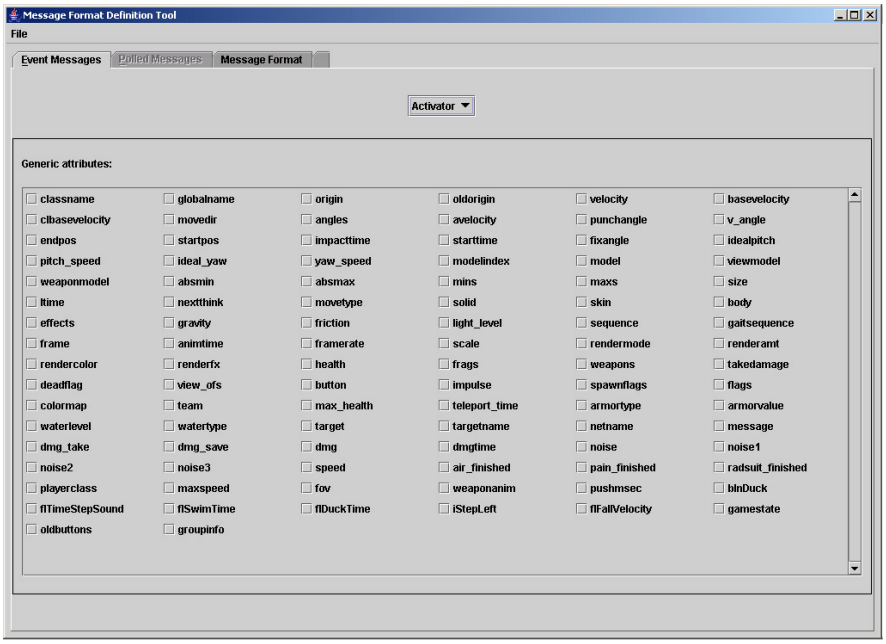

Figure 6.10 Event message, activator attributes.

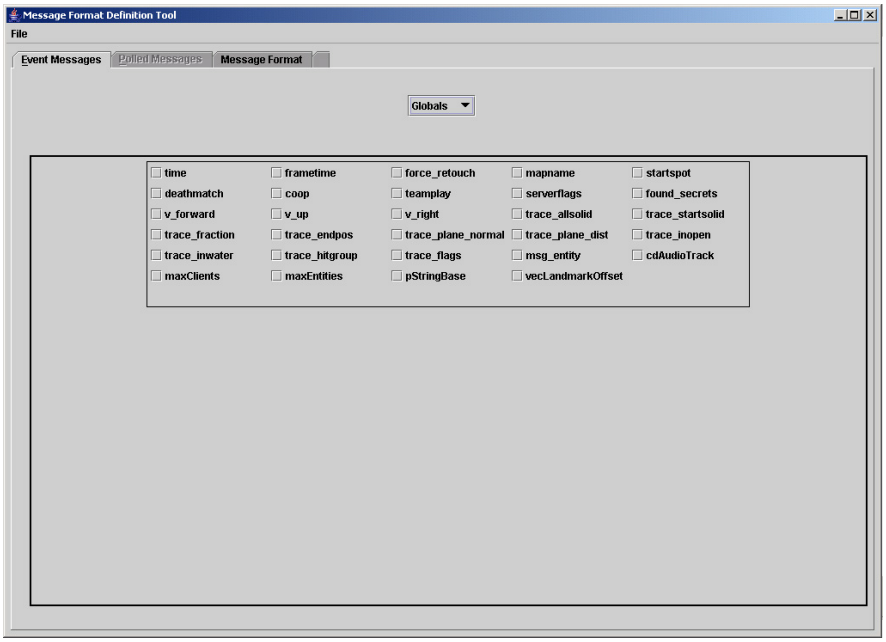

Figure 6.11 Event or Poll message, global attributes

Bibliography

[1] *The Computer for the Twenty-First Century*
Mark Weiser
Scientific American, 1991,Vol 265, No. 3, pp. 94-104

[2] *UbiWise, A Ubiquitous Wireless Infrastructure Simulation Environment*
John J.Barton, HP Labs
Vikram Vijayaraghavan, Stanford University
Copyright 2002, HP.

[3] *User Study Techniques in the Design and Evaluation of a Ubicomp Environment*
Sunny Consolvo, Intel Research Seattle
Larry Arnstein, Dept. of Computer Science & Engineering, University of Washington.

[4] *Sentient Computing Project*
Cambridge University Engineering Department
Andy Ward, Pete Steggles, Rupert Curwen, Paul Webster
http://www.uk.research.att.com/spirit

[5] *Evaluating a Location-Based Application: A Hybrid Test and Simulation Environment.*
Ricardo Morla, Nigel Davies, Lancaster University
Pervasive Computing, IEEE, 2004

[6] *CIT Meeting Review*
Declan O'Sullivan
Thursday 15th July 2004

[7] *M-Zones Deliverables D4.3*
Experimental Testbed Report, June 2004

[8] *XML: eXtensible Markup Language*, http://www.w3.org/XML

[9] *DTD: Document Type Defintion*, DTD Tutorial
http://www.w3schools.com/dtd

[10] *SVG: Scalable Vector Graphics*
http://www.w3.org/TR/SVG

[11] *DOM: Document Object Model*
http://www.w3.org/TR/DOM-Level-2-Core/glossary.html

[12] *Xerces: XML parsers in Java and C++*
http://xml.apache.org

[13] *TinyDB: In-Network Query Processing in TinyOS*
Sam Madden, Joe Hellerstein and Wei Hong
http://telegraph.cs.berkeley.edu/tinydb/documentation.htm

[14] *Java 2 Platform*
Standard Edition, v 1.4.2 (SDK)
http://java.sun.com/j2se/1.4.2/download.html

[15] *C++ How To Program*
Harvey M.Deitel, Paul J.Deitel
Prentice Hall Professional Technical Reference, Fourth Edition, 2002

[16] *The Microsoft Developer Network, Library*
Microsoft Visual C, http://msdn.microsoft.com/library

[17] *Half-Life*, Valve Corporation, http://www.valvesoftware.com

[18] *Half-Life SDK v2.3*, Valve Corporation, http://dev.valve-erc.com

[19] *Valve Hammer Editor v 3.4*, http://collective/valve-erc.com

[20] *Sierra Entertainment, Inc*, http://half-life.sierra.com

[21] *Botman's Bots*, HL SDK Tutorial, http://www.planethalflife.com/botman

[22] *Planet Half-Life*, http://www.planethalflife.com/fixxxer

[23] *Handy Vandal's Almanac*, http://karljones.com/halflife/almanac.htm

[24] *The Wavelength*, HL Mod development Resource
http://thewavelength.net

[25] *The World of Half-Life*, Mapping Tutorials & Forum
http://cariad.co.za/twhl

[26] *Quake III Arena*, id Software, http://idsoftware.com

[27] *Quake Downloads*, id Software
http://idsoftware.com/business/techdownloads

[28] *Quake III Rally*, Square Eight
http://www.squareeight.com/webprojects/development/quakerally.shtml

[29] *Unreal Tournament*, Epic Games, http://www.epicgames.com

[30] *Atari*, http://corporate.infogames.com