Context Information as Augmented Reality

Eric Patrick Chubb

A dissertation submitted to the University of Dublin,

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

2008

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.

Signed _____

Eric Patrick Chubb

September 2008

Permission to lend and/or copy

I agree that Trinity College Library may lend and/or copy

this dissertation upon request.

Signed _____

Eric Patrick Chubb

September 2008

Acknowledgements

Firstly, I would like to thank my supervisor Dr. Stefan Weber for his invaluable assistance and guidance during the course of this project, particularly in keeping me focused on what was important and suggesting possible directions for the dissertation to follow. Many thanks must also go to Eleanor O'Neill of KDEG for her co-operation and hard work in modifying the TATUS simulator for use with my dissertation, particularly at short notice. I would also like to thank Dr. Ciaran McGoldrick and Ricardo Carbajo for assistance with the queries I had during development.

I would like to thank my family for their support and encouragement during the past year and also during the rest of my time in university.

Finally, I would like to thank my classmates in Trinity for a hugely enjoyable and memorable year, and for their support during the course of the dissertation.

Abstract

The vision of Ubiquitous Computing [41] by Mark Weiser describes a new generation of computers as being part of the environment and invisible to the user. Such an environment would consist of systems embedded in everyday items. These systems would allow users to interact with them in a casual manner and would communicate with each other. Many of these systems would have sensors that measure environmental attributes, or "context", such as temperature, ambient light levels, humidity or wireless Internet signals in a dynamic fashion. This information would be used to facilitate decision processes within the devices, but also may be useful to a user in understanding their environment.

This dissertation investigates the architecture for the collection and display of dynamic context information as part of an augmented reality system. A novel interactive environment called the Context-Aware Augmented Reality (CAAR) system is presented, which visualises environmental context such as temperature and luminance. This environment moves the focus of interaction between user and computer from a discrete site such as the desktop, to the user's whole environment. The system is then evaluated in terms of its facilitation of a greater awareness of a user's environment, and also as an enhancement to current human-computer interaction technologies.

Contents

1 Introduction **1**
1.1 Shortcomings of human-computer interaction 1
1.2 Augmented Reality 2
1.3 What can be visualised? 4
1.4 Research Questions 7
1.5 Objectives 8
1.5.1 Work Undertaken 8
1.5.2 Roadmap 9

2 State Of The Art **10**
2.1 Technology 10
2.2 Literature Review 15
2.2.1 Existing Augmented Reality Systems & Applications 15
2.2.2 Context-aware systems, designs and approaches 18
2.2.3 Problems & Challenges of Augmented Reality and Context-Aware Computing 21
2.2.4 Summary 23

3 Design **24**
3.1 Project Requirements 24
3.1.1 High-Level System Design 25
3.2 Design 1: Acquiring Context 28
3.3 Design 2: Visualising Context 32
3.3.1 Summary 35

4 Implementation **36**
4.1 Supporting Technologies - Software . . . 36
4.1.1 TinyOS . . . 37
4.1.2 TATUS / Half-Life 2 . . . 38
4.2 Supporting Technologies - Hardware . . . 39
4.2.1 Sensor Technology . . . 39
4.2.2 Display Technology . . . 42
4.3 Implemented Technologies . . . 43
4.4 Contextifier & newTOSBase . . . 43
4.4.1 Getting Context . . . 44
4.4.2 Sending Context . . . 52
4.4.3 Routing Context . . . 56
4.5 Processing Context . . . 60
4.6 Visualising Context . . . 65
4.7 Challenges & Implementation difficulties . . . 68
4.8 Summary . . . 69

5 Evaluation **70**
5.1 Comparison between CAAR and other augmented reality systems . . . 70
5.1.1 Mobile Augmented Reality System (MARS) . . . 70
5.1.2 ARQuake . . . 72
5.2 Project Objectives versus Project Implementation . . . 72
5.3 Informal User Study . . . 73
5.3.1 Viewing screenshots . . . 73
5.3.2 Viewing the running system . . . 77
5.3.3 Results . . . 78
5.4 Summary . . . 79

6 Conclusions and Future Work **80**
6.1 Future Work . . . 80
6.1.1 Improved Wireless Sensor Network Implementation . . . 80
6.1.2 Support for optical head-mounted displays . . . 81
6.1.3 Location awareness of sensors and users . . . 81

6.1.4 Support for additional types of context 81
6.1.5 Improved visualisation techniques 82
6.2 Conclusions 82

List of Figures

2.1 Concept shot of a see-through augmented reality device 11
2.2 Closed-view HMD - Taken from Inition's website 11
2.3 A MicaZ mote without sensorboard 14
2.4 A SunSPOT sensor 15

3.1 High Level Architecture of CAAR 26
3.2 Initial Sink Activation. 30
3.3 "Hello" Message Propagation. 31
3.4 Completed route discovery process. 32
3.5 TATUS simulation of a lecture room in the Lloyd Institute in Trinity College 34
3.6 Revised High Level Architecture of CAAR 35

4.1 A MicaZ sensor node with attached sensorboard 40
4.2 Sink node on programming board 41
4.3 An MTS420 sensorboard 42
4.4 A Vuzix iWear VR920 headset 43
4.5 Light measurement gradients 63
4.6 Humidity measurement gradients 63
4.7 Temperature measurement gradients 64
4.8 Initial visualisation implementation 66
4.9 Final visualisation implementation 67

5.1 CAAR system without sensory input 74
5.2 CAAR system with one sensory input 75
5.3 CAAR system with both sensory inputs 76
5.4 CAAR system with one sensor artificially heated 77

5.5 Compiled results for user study 78
5.6 Chart of response time results in milliseconds 79

Chapter 1

Introduction

What delights us in visible beauty is the invisible.
-Marie von Ebner-Eschenbach

1.1 Shortcomings of human-computer interaction

Conventionally, human beings interact with computing technology in an extremely generalised manner, namely the Windows, Icon, Mouse, Pointing (WIMP) system or traditional "desktop" model, where interaction takes place via a graphical user interface displayed on a monitor and input devices such as mice and keyboards. It involves a user consciously and intentionally engaging with a computing system in order to achieve a specific task. While it is acceptable for many applications, it does not usually exploit or satisfy particular characteristics or requirements of others, such as some videogames, multimedia entertainment, aviation, medical or automotive computing applications. Additionally, this model often imposes unnecessary and artificial limitations on how a user may interact with the system, such as an overly complex set of input commands, monoscopic visual output and a cluttered user interface.

This problem becomes more evident and more disabling as the adoption of computing technology within all sections of twenty-first century society continues to proceed with the stunning expediency we have seen so far. Smaller, increasingly powerful and more network-capable computers are constantly being developed, with the result that whereas computers were once confined to the home study or office, they are now to be found

in satellite navigation devices in cars, mobile phones, portable audio players, home entertainment and digital television set-top boxes, so-called "mission-critical" applications such as nuclear reactors and flight-control systems, and even home appliances such as refrigerators. With this rapid expansion of the computing application space, the traditional desktop model's major limitations become all the more apparent, and it becomes clear that such novel application domains will demand models of interaction far beyond the sophistication of conventional desktop-based interfaces that are currently in use. Indeed, a report by an analyst at Gartner Research has predicted the demise of the mouse as the input device of choice within the next 3-5 years [37]. Some progress has been made in overcoming the shortcomings of the now positively antiquated desktop model, with sensors as well as gestural and facial recognition technology becoming commoditised in home electronics devices. Apple's iPhone, iPod Touch and the latest generation of their MacBook series of laptops have finger gesture recognition built in, while Nintendo's "Wiimote" controller for its "Wii" videogame console has a built-in accelerometer which can detect any motion the controller experiences and display the results according to the game being played. However, all of the above maintain the notion of explicit, conscious interaction between people and computers.

1.2 Augmented Reality

A field of research which aims to overcome the limitations of the current point-and-click standard interface, and profoundly transform the manner in which humans and computers interact is that of Augmented Reality. This research discipline investigates the fusion of real-world sensory data from sources such as human eyesight or hearing with computer-generated data, with the aim of "augmenting" the real with the virtual. Typically such augmentation takes place through the use of a head-mounted display which uses transparent or semi-transparent glasses over which computer generated data is superimposed. However, augmented reality technologies are not restricted to visualisation, as auditory and tactile feedback devices are also capable of enhancing one's perception of the world around them. Although there are several definitions as to what actually constitutes an AR system, Ronald Azuma, in his paper "A Survey of Augmented Reality" [6], identifies three main characteristics that define an AR system. An Augmented Reality system, he states

- "Combines real and virtual"
- is "Interactive in real time"
- is "Registered in 3-D"

Within this definition, the term "registered" refers to the projection of computer-generated images over a user's view of the physical world. The "Tinmith" website provides a richer definition of Augmented Reality, stating that it is "the registration of projected computer-generated images over a user's view of the physical world. With this extra information presented to the user, the physical world can be enhanced or augmented beyond the user's normal experience" [29]. Jim Vallino of the Department of Software Engineering at Rochester Institute of Technology in New York states that "augmented reality presented to the user enhances that person's performance in and perception of the world" and that the ultimate goal of Augmented Reality "is to create a system such that the user can not tell the difference between the real world and the virtual augmentation of it" [40]. At present, the technology has several applications. showing promise in military simulation applications, videogames and even surgical procedures and medical scanning, with several working implementations of these applications already in use.

There is an obvious connection between Augmented Reality and the field of Ubiquitous Computing. The latter outlines a "post-desktop" view of the computing world where computers are seamlessly integrated into everyday activities and technologies, and where a user's engagement with a computing system need not be conscious or intentional, or even require the user to know they are using a computer system at all. Augmented reality offers a similar view of the computing landscape, moving the focus of interaction from a single, explicit, well defined point, into the user's entire surrounding environment. Mark Weiser, the-once Chief Technologist of the Xerox Palo Alto Research Centre (PARC) and originator of the term "ubiquitous computing," wrote some of the pioneering literature on the subject. In his paper "The Computer for the 21st Century", he presented the term "embodied virtuality" which refers to the process "of drawing computers out of their electronic shells." [41]. He also stated that it is a definition which is diametrically opposite to that of "virtual reality," which he describes as trying "to make a world inside the computer."Indeed, Azuma states that a user, immersed in a virtual reality "cannot see the real world around him" whereas "AR allows the user to see the real world, with virtual

objects superimposed upon or composited with the real world" and thus "supplements reality, rather than completely replacing it" [6].

As is the case with most technologies, there are some inherent challenges in AR which need to be overcome for it to be fully accepted by would-be users. Presenting a visual environment that seems realistic to the user in a timely fashion has proved challenging mostly due to the so-called "registration" problem [6], which is the problem of keeping the user's view of the real world and their view of computer-generated data geometrically aligned, and a sizeable body of literature has been written with techniques to address it, often by using head-tracking techniques. Nonetheless, AR continues to prove an interesting alternative to more conventional methods of interacting with a computer.

1.3 What can be visualised?

Although augmented reality presents us with the potential for novel and exciting ways to use computing devices, some questions remain as to what data we wish to present, how we propose to acquire that data, and how we propose to present it in an intuitive and meaningful way to someone who has never used an augmented reality system before. Indeed, in the augmented reality literature reviewed for this dissertation, very little research detailed how data would be collected, if at all. The field of "Context-aware" computing offers a means to perform such data acquisition. It is a research discipline which focuses on computing devices can sense changes in their "context" and react to them. Context can be the physical environment in which the computing system exists, such as the air temperature or humidity. Context can also be the computing resources available to the system, such as power and networking, or it can be the social situation in which the system is being used, whether at home, in an office, with a co-worker or without. It can also be more standard information such as the time of day or of the year.

Historically, context has been difficult to define in the literature but some researchers have attempted to do so nonetheless. Schilit et al [34] define context in a simple taxonomy:

- *Physical context* such as magnetism, pressure, temperature etc.
- *Computing Context* such as network connectivity and bandwidth
- *User Context* such as a user's location, profile or even their social situation

Specifically they state that context is "where you are, who you are with, and what resources are nearby... Context encompasses more than just the user's location, because other things of interest are also mobile and changing. Context includes lighting, noise level, network connectivity, communication costs, communication bandwidth, and even the social situation; e.g., whether you are with your manager or with a co-worker." Chen and Kotz [11] extend this definition to include *time context* which could be the current time of day or month of the year. Dey [12] defines context as "any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." Finally, Schmidt et al. [36] define it as "knowledge about the user's and IT device's state, including surroundings, situation, and, to a lesser extent, location." Context-awareness, then, is the ability of a computing device to sense context in the terms defined above, abstract and evaluate the context, and behave in a way which maximises its usefulness within that context. The challenges, therefore lies in acquiring context and extracting meaningful information from it, and matching specific context to particular application behaviour semantics.

Context-aware devices are typically (but not necessarily) small embedded devices which support networking, usually wirelessly, and are highly mobile and portable, taking the form of sensors or other devices. Portability is very important if the devices are to exploit user context and be distributed around a user's environment. Such devices are constantly becoming more affordable, while increasing in computational power and accuracy, so it is not difficult to imagine how useful and ubiquitous they may become in the near future. Like Augmented Reality, context-aware computing has clear links to Ubiquitous/Pervasive computing or "UbiCom." Moran and Dourish [22] refer to a definition of UbiCom they found in an (apparently unreferenced) call for a conference on Pervasive Computing. It defines UbiCom as comprising of

- Numerous, casually accessible, often invisible computing devices
- Frequently mobile or embedded in the environment
- Connected to an increasingly ubiquitous network infrastructure composed of a wired core and wireless edges

This suits the definition of Context-aware computing devices rather neatly, as they too

can be characterised by their mobility and small size, and their potentially simultaneous ubiquity and invisibility.

Now that computing context information and context-awareness has been defined and discussed, the question arises of why we would concern ourselves with context, and more specifically, why we would want to visualise it in an AR system. The fact remains that computing devices, generally speaking, have little or no concept of the environment of which they operate, beyond simple configuration settings such as region, locale or language for one's operating system, the name of the person using the device in question or the current date and time. They have no awareness of the user's current context or the social environment in which they are being used. Furthermore, even if context is used, it generally is used in a very limited fashion, within specific application scenarios such as the Jimminy note taking system [31] or the Xerox ParcTab [33] personal communication device. In addition, context, when it is employed at all, is largely unnoticed or unusable by the user directly. This can restrict the user's perceived utility of the computing device itself.

With that said, a different class of device has emerged in recent years which acts as a "consumer" of context data, rather than a producer. These are typically consumer electronics devices, good examples of which are Apple's "iPhone" or Nintendo's"Wii" console. The Wii's wireless controller contains an accelerometer which is used to control the action on screen. The iPhone contains an accelerometer for detecting the device's geometric orientation, a light sensor to dim screen brightness depending on one's surroundings to save battery life, and a proximity sensor which disables the device's touch screen when the iPhone is held up to a user's ear to prevent spurious inputs. While the context mentioned above is used to modify device behaviour, the data is not gathered for the purposes of transmitting to another device for example. Also, the context has a very specific technical purpose, such as saving battery life, or eliminating the need for wired controllers, but does not provide any new services or functionality to the user, and the device still has no concept of the social or user situation in which it is being used.

Dey [12] makes a strong case for the use of contextual information in computing applications, arguing that within "traditional interactive computing, users have an impoverished mechanism for providing input to computers. Consequently, computers are not currently enabled to take full advantage of the context of the human-computer dialogue." He also suggests that the definitions of context in the literature are insufficient,

criticising them as merely "synonyms for context; for example, referring to context as the environment or situation," and that previous "definitions that define context by example are difficult to apply," which he states has led to model of context which are too specific to particular application scenarios to be generally useful. By using contextual information in computational terms, we can create a richer, more user-centric model of interaction between humans and computers, and facilitate the provision of services and computational utility that adapts and moulds itself to the situation in which it is being applied. Rather than being designed on a static, immutable set of assumptions, context-aware devices can evaluate the user's current social or environmental context and adjust its behaviour accordingly.

In order to use context in a more user-centric and useful way, we need to develop a way to present it to a human being in an intuitive manner. Visualisation makes sense as a means to do this, as humans are primarily visual creatures due to the effect of "visual capture," or the phenomenon of sight overriding all other senses such that they conform to what is seen [42]. Therefore, Augmented Reality provides us with the ability to present context in a natural, but totally immersive manner, exploiting the spatial arrangement of such context rather than displaying it on a computer monitor in a fixed location. When dealing with a synergy of Augmented Reality and Context-Aware Computing, it is the physical context that we are most interested in, as it opens up a novel applications, like visualising previously invisible environmental context such as air pressure, radiation levels, ambient humidity, temperature, and so forth on the head-mounted display unit or visor.

1.4 Research Questions

The research questions this dissertation aims to address are:

- How should dynamic context information be visualised in an augmented reality system
- Does such a visualisation facilitate a greater understanding of the environment

1.5 Objectives

With answering of the research questions above in mind, the central goals of this dissertation are threefold:

- To explore means of acquiring data which can be visualised in an Augmented Reality system.
- To develop a method of presenting computing context in a novel yet intuitive and accessible manner.
- To explore alternative modalities of human-computer interaction.

1.5.1 Work Undertaken

As a means to achieve the abovementioned objectives, an interactive environment called the Context-Aware Augmented Reality (CAAR) system is presented, in which data is collected and aggregated by a context-aware system, in the form of a wireless sensor network. This sensor network consists of several nodes or "motes" with "sensorboards" attached which can sense and measure environmental properties such as light levels (luminance), temperature and relative humidity. Each mote is programmed with purpose-written software designed to record and forward measurements. A basic routing algorithm has been implemented which differentiates between sensor nodes and base station nodes. Base station nodes or "sinks," are motes connected to an interface board which, in turn is connected to a laptop computer. They simply send out "hello" messages, alerting sensor nodes to their existence and receiving sensor readings from those sensor nodes. Base station nodes are capable of forming TCP connections between each other and forwarding readings of individual sensors in their vicinity, in effect forming a superpeer-based peer-to-peer network. Each sensor node attempts to forward both its own sensor measurements and those of other motes in radio range to the base station, where the data is processed and converted into usable data. Each sensor node will attempt to transmit readings to the base station directly to prevent other sensor nodes' batteries being drained from forwarding other nodes' messages, but can choose to forward them through intermediate sensor nodes by detecting that they are out of range of the base station. Once the readings arrive at the base station, they are processed and converted into a

usable form such as Celsius, before being submitted to another component to be visualised. This component makes use of the Half-Life 2 graphics engine, and in particular, the TATUS [25] simulator developed by Eleanor O'Neill.

1.5.2 Roadmap

The structure of the remaining part of the dissertation is as follows:

- Chapter two discusses various technologies used and implementations developed within the fields of Context-Aware computing and Augmented Reality, as well as the challenges and common problems associated with each.
- Chapter three explains the design and implementation of the CAAR system, including technologies used, approaches implemented and problems and challenges encountered.
- Chapter four explores the evaluation of the CAAR system in terms of a user's ability to understand the visualisations being presented, as well as the performance of the wireless sensor network.
- Chapter five discusses what conclusions can be reached from the work performed in the dissertation, as well as possible future work to enhance and expand upon it.

Chapter 2

State Of The Art

This section examines the different technology platforms that exist within the domains of Context-aware Computing and Augmented Reality. It is useful to examine these as it provides necessary background to discussing the implementation presented further on as well as introducing the reader to the terminology and vernacular associated with the two topics mentioned above. In addition, an investigation and comparison of current context-aware and context-visualisation systems is presented, which looks at previous designs and approaches within these areas.

2.1 Technology

Augmented Reality

An Augmented Reality system can take several forms. The vast majority of systems incorporate some form of visualisation, achieved by the user wearing a head-mounted display and often an accompanying connected wearable computer which receives the data to be visualised, processes it and performs the augmented reality itself. The key differences between AR systems lie in how the virtual data is visualised and synthesised with the real world data. This, according to Azuma [6] forms a "basic design decision." He defines typical data visualisation techniques through use of a "see-through" or optical Head-Mounted display(HMD) and "closed-view" or video-based HMD. Takemura et al. [21] present a similar definition of AR system composition. A see-through HMD uses optical "combiners" which allow some light from the real world to pass through them

while also reflecting light from head mounted monitors to present the virtual data.

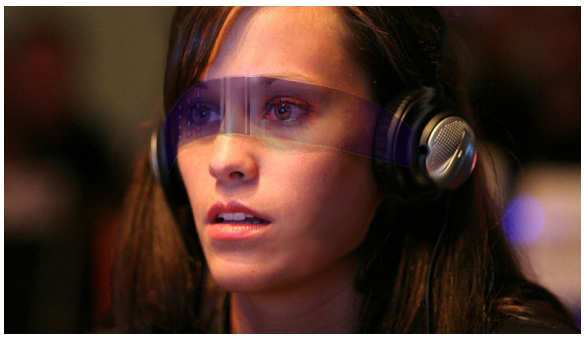

Figure 2.1: Concept shot of a see-through augmented reality device

By contrast, a closed-view Head-Mounted Display works by completely sealing off a user's field of vision from the outside world, instead using two small LCD monitors placed directly in front of the eyes to provide optical data. These monitors are usually fed live digitised video data from cameras mounted on the HMD.

Figure 2.2: Closed-view HMD - Taken from Inition's website

Each approach has specific advantages and disadvantages. Azuma [6] identifies the advantages of optical approaches, citing their lower cost and simpler implementation details. He also states that, unlike video approaches, the resolution of a user's view of

the real world is maintained, and that view of the real world cannot be cut off if the system suffers a power failure, which would be a safety concern in some applications. However, he also points out that optical approaches suffer from a lack of flexibility in the way synthetic and real data can be merged or "composited". Additionally, the field of view provided by optical HMD's is limited due to the additional distortion that widening the field of view introduces. Such distortion is difficult, but not impossible, to correct optically.

Video-based approaches typically allow for the mitigation of temporal mismatches seen in optical-based HMD's, as the user sees the real world data instantly, but the virtual data may suffer delays. The delay between the two data-streams, real and virtual, can be matched in a video HMD to provide a more acceptable experience to the user[10]. Additionally, complex registration strategies can be employed in video based augmented reality systems to reduce the effect of the registration problem, strategies which are impossible in optically based systems.

Video HMD's suffer from some drawbacks however. They are typically more expensive and more difficult to implement, as the closed-view provided requires the digitisation of real-world imagery into a video stream, which adds additional delays to the user's view of the world, unlike optical systems which require no such digitisation. Video cameras - the chief source of real-world data for a video HMD, introduce a visual offset between the user's eyes and the location of the cameras. Optical systems have no such problem. Finally, video-based HMD's suffer from an inherent resolution limit. This limit is typically far lower than the resolving power of the human eye, so implementers and manufacturers typically must deal with avoiding display issues such as pixellation.

Context-aware Computing

The range of platforms and devices which could be labelled as being "context-aware" is enormous, depending on the granularity and sophistication of the particular model of "context" being used. Schmidt, Beigl & Gellersen [36] identify a class of "ultra-portable" devices suitable for context-aware computing, namely mobile phones, PDAs and wearable computers. They focus on physical context, arguing in favour of "awareness of the physical conditions in a given environment, based on the assumption that the more an ultra-mobile device knows about its usage context, the better it can support their

user." For the purposes of this dissertation, systems which specialise in physical-context were examined, as it is physical context which the implementation mentioned later on aims to collect and visualise in an augmented-reality system.

One context-aware platform which is receiving a lot of commercial and academic attention is Wireless Sensor Networks (WSN)'s. These are typically composed of small, matchbox-sized devices, known as "motes," which support a range of environmental sensing capabilities. The fact that such sensor nodes can be distributed over a wide area and networked wirelessly facilitates sensing on a scale which is just not practical using other methods. The variety of sensing capabilities offered by these devices also provides an opportunity to gain an unprecedented level of information about a target area, be it a room, building or campus. Wireless sensors have a diverse range of possible applications, but frequently are used in monitoring or tracking scenarios, as detailed in the "Literature Review" section of this dissertation.

Wireless sensor nodes typically comprise a battery-based power source like AA batteries, a RISC microcontroller for data processing, EEPROM memory for storing application or user data and some form of radio transceiver for communication with other sensor nodes. The devices typically communicate via some standardised WPAN protocol such as ZigBee, which supports data rates of 250 kbps, 40 kbps, and 20 kbps and operates in the 2.4 gHz, 915mHz and 868mHz RF bands with CSMA-CA medium access control and optimisations for lower power consumption [4]. Some mote types are completely self-contained, with sensing capabilities built-in, but others require the use of an additional "sensor board" containing the sensors themselves. Typically, the sensors are essentially analogue-to-digital converters which are sensitive to different environmental phenomena, and simply return a raw 10 or 12-bit value to the software running on the mote, requiring some further processing or conversion to be usable.

There are many different commercially available sensor platforms, each of which support different sensing capabilities and use different hardware components and run different application types. One such example is the MicaZ mote product line, manufactured by Crossbow Technologies [5], which is amongst the best known. MicaZ's do not come with any sensing functionality built-in, but rely on the use of an external sensorboard which connects to motes via their JTAG interface.

Figure 2.3: A MicaZ mote without sensorboard

Wireless sensors typically have very limited computational resources, in terms of memory and CPU as well as a very limited power source, so developers are wise to implement software for motes with the intent of maximising their useful lifetime, by writing applications which make intelligent and efficient use of CPU and radio resources. The devices themselves are typically programmed using some sort of operating environment or framework which simplifies the development of application for wireless sensor nodes by providing compilers and loaders for these specialised platforms as well as development frameworks for quickly building applications. One such environment is TinyOS [3], an open-source, event-driven operating system which provides a programming toolchain based around the GNU compiler collection as well as pre-written software components to facilitate rapid application development. Application code is statically linked with TinyOS code and compiled into a small binary file. TinyOS will be discussed in more detail in the "Implementation" section.

Another type of wireless sensor is the Sun SPOT(Sun Small Programmable Object Technology) developed by Sun Microsystems. The SunSPOT is a considerably more powerful sensor platform than the MicaZ and others of its' type, utilising a 180MHz 32-bit ARM920T core processor with 512K RAM and 4MB of Flash memory [2]. What makes the SunSPOT particularly interesting is its support for a small version of the Java Virtual Machine (JVM) called Squawk [1] which facilitates application portability, as well as usage of Java's support for threads, objects and so forth. However, Squawk is mostly

written in Java itself, as opposed to C/C++ or assembly language as is the case with practically all other JVMs. SunSPOTs, like MicaZs use ZigBee to communicate with each other, but unlike MicaZs, come with a variety of sensing capabilities such as light sensors, accelerometers and thermistors built-in, as well as LED's for output. Unlike other mote types, SunSPOTs come with automatic power management to maximise the lifetime of their 750mAh rechargeable battery.

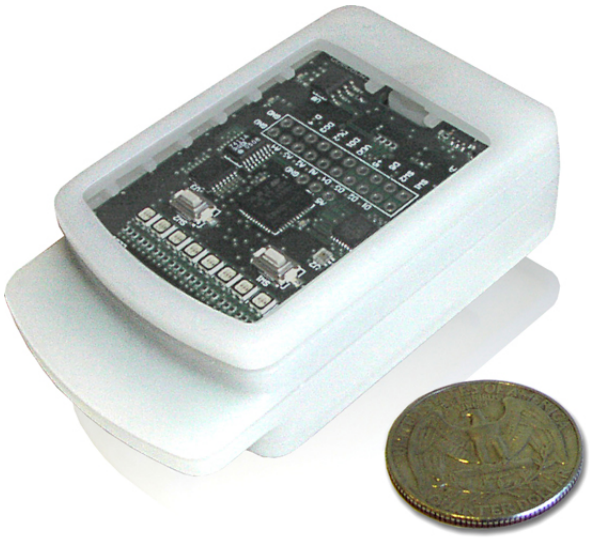

Figure 2.4: A SunSPOT sensor

2.2 Literature Review

2.2.1 Existing Augmented Reality Systems & Applications

Although largely overshadowed by the hyperbole surrounding virtual reality, augmented reality has found its way into a diverse and somewhat unexpected range of application domains. Azuma [6] identifies a range of applications where AR has successfully integrated. One such area which shows promise is in the medical field, specifically surgery. Grimson et al [15] define a technique for combining real video footage of a surgical patient with a three-dimensional MRI or CT model of intracranial or extracranial structures, which allows neurosurgeons to plan operations in great detail and view the surgical subject areas with minimum invasiveness. They detail a method which uses laser scanning and

MRI techniques along with a specially-calibrated video camera, which can be used to guide surgeons in great detail in delicate procedures such as biopsies or neurosurgery. State et al[14] developed an Augmented Reality system which synthesises ultrasound data with stereoscopic images acquired by a camera on a Head-Mounted-Display (HMD) which presents a combined view of the data to a surgeon using the HMD. Using this system, they managed to assist a surgeon in guiding a needle into a mock tumour inside a "phantom" breast used for training purposes.

Another area where AR is applicable is multiplayer gaming. Reijers et all [30] present a game, "Quazoom", based on FLARE, a framework developed by the authors for building AR applications, which allows users to move around a real-world environment in a style similar to "Quasar" and interact with real and virtual players, with the system keeping track of player locations via differential GPS technology. The game itself runs on wearable computers and communication takes place over a wireless ad-hoc network. This paper however focuses mostly on the problems of distributed computing such as maintaining consistency and ordering that present themselves in the system rather that the visualisation or data capture technologies which were used.

Piekarski and Thomas [28] developed ARQuake, a modification of the desktop PC game Quake designed to run in Augmented Reality scenarios. The player wears a HMD which uses a half-silvered mirror to combine data from an LCD screen with the user's real view of the world. The system runs on a wearable computer and uses Garmin GPS technology for position determination and TCM2 digital compasses for orientation. With this setup, a player can move around a real world and combat real and virtual players as seen through their headset. This system shows a lot of promise but was first published in January 2002, so it would be interesting to see what the massive improvements in consumer video cards and computing power that have taken place over the past six years can add to the experience.

Augmented Reality systems are suitable for application in the battlefield simulation field. These would be extremely useful for training purposes as the expectation is that many future military deployments will in urban settings which make for far more demanding and dangerous combat environments than open spaces. In Julier et al [18], the authors discuss the Battlefield Augmented Reality System (BARS) which comprises a wearable computer, wireless network system and see-through Head Mounted display. The system views the battlefield as a set of objects, ranging from physical objects such

as trees or buildings to the location of friendly and enemy forces, and is able to superimpose this information via the HMD onto the user's field of vision. This paper by its own admission does not look at the data link or data collection issues but discusses what data will be used in this virtual battlefield scenario.

"Studierstube" which was implemented at Vienna University of Technology by Fuhrmann et al [35] illustrates AR's potential as a collaborative and educational tool, being designed as an interactive collaboration application to allow multiple individuals to study three-dimensional scientific visualisations. Each user has his/her own head-tracked HMD which allows full stereoscopic views of a visualisation. The visualisation can also be manipulated by using a three-dimensional control panel called the "Personal Interaction Panel," which is superimposed over a user's POV. The authors argue that this is a more cost-efficient and space-efficient AR implementation than past efforts.

Another AR system which uses machine vision as a primary source of real-world data is the "Touring Machine", developed by Feiner et al [13] at Columbia University in New York. The system allows users to walk around Columbia University's campus and annotates buildings and landmarks with supplementary information. It comprises a wearable computer, differential GPS unit to keep track of a user's location and a spread-spectrum wireless network to communicate with additional infrastructure. Using this design, the Touring machine system grants the user true mobility, allowing them to walk around without worrying about being connected to a base computing device. However, the system was first described in 1997, and in the interval since mobile computing has gotten much smaller and much more powerful, so the potential for additional miniaturisation and optimisation exists, as the system described is quite cumbersome, requiring a user to carry a large computer in a backpack along with a large GPS antenna.

As effective as the abovementioned implementations are, they do not fully exploit the potential of AR in the sense that they are either annotating what is already visible, facilitating modelling of what is visible but perhaps not easily accessible(as is the case with the abovementioned surgical AR applications) or presenting visualisations which exploits a user's location only, but nothing significant about his or her environment. There remains a rich set of contextual data which is yet to be fully mined. One way of tapping into this potential is using context-aware computing devices such as PDA's, laptops and distributed wireless sensors. These would offer the potential to collect a vast range of environmental and contextual data over a large surface area such as a office

building or even a university campus, in effect providing an environmental weathermap of a user's locality. For instance, it is possible that a wireless sensor could collect data regarding magnetic fields, atmospheric pressure, atmospheric composition, surface acidity or alkalinity, electromagnetic or even radioactivity metrics. So it is not difficult to see the vast range of applications that could be realised using this synergy of AR and Context-Aware computing. However, this seems to be an emerging area as most literature on the subject does not go into much detail regarding the collection of context or the problems presented therein.

2.2.2 Context-aware systems, designs and approaches

As mentioned in the introductory section, context-aware devices detect and react to changes in their "context" or immediate environment. A question which immediately comes to mind is how to detect and monitor context. Chen et al [11] go into detail regarding methods of detecting context and context changes. They define three main levels of context: high-level, low-level and location. For location sensing outdoors, the authors state that GPS is an obvious choice, which it is. It's a widely available and accurate technology which would fill the requirements neatly. However, they also point out that indoor location detection is more difficult and relies on novel technologies like "smart floors" which identify a person by their footstep force profile. This however only works for humans. Other methods include ultrasonic technologies.

The authors define high-level context as the Context-aware system's user's current activity, and remark that this is difficult to determine, suggesting the use of Artificial Intelligence (AI) techniques, combinations of ambient light-sensors and accelerometers or possibly using calendar consultation. However, it is stated that all three methods fall victim to boundary conditions and undefined model problems. Low-level context is defined as network bandwidth, time or orientation, and it is stated that these can be determined with varying degrees of difficulty.

Sensing changes in context and storing contextual information are other issues which must be considered. The authors discuss methods such as publish-subscribe systems which poll the current context at rates determined by human experts and publish it to clients interested in it. Storing contextual information requires a location model, which is

usually a *symbolic* or *geometric* model, with the former representing location as abstract symbols and the latter representing location as co-ordinates. Object-oriented or key-value pair-based models can be used to store other types of contextual data also. Ontological models are a promising method of storing contextual data, as ontologies describe concepts and the relationships between them and are a "very promising instrument for modelling contextual information due to their high and formal expressiveness and the possibilities for applying ontology reasoning techniques." [7]

One final (but very important) consideration of Context-Aware system design is the system architecture, and whether to use a centralised or distributed model [7]. Each has their place, with the centralised model, using a "context server" to provide applications with context, being the easiest to implement but the least scalable and fault-tolerant. The distributed model allows context to be held at several places to avoid potential bottlenecks, but difficulties arise in sharing context in a timely manner.

The previous work remained relatively platform-agnostic in terms of the methods of context gathering. However, Fitzpatrick et al [14] define the notion of a "sentient object," that is, a software component which can be deployed in a mobile environment (specifically on wireless sensors) and react to changes in its environment. They also are one of the few papers seen by this author which discuss the hardware platform on which Context-Aware applications would be deployed. Sentient objects communicate via event-based inter-process communication, and events are defined in terms of software and hardware. The objects can also produce real-world or software events. In this instance, a sentient object uses ubiquitous sensors which respond to some physical stimuli such as thermal energy, GPS co-ordinates, pollution sensors etc., to detect and gather context and react to it using so-called "actuators" which effect a change in the environment of the sentient object in response to the consumption of this real-world event. The authors then define a sentient object as "an entity which can both consume and produce software events, and lies in some control path between at least one sensor and actuator." The authors state they are investigating the use of XML as a context representation format, which would be ideal given the tools available to manipulate and transform XML into other types of information. Finally, the authors state that sentient objects make use of an "inference engine," which gives the object some decision making capability, and they may also use "stigmergy," which is co-ordination of distributed entities, not through

explicit communication between each other, but via modifications they make to their environment, first observed in social insect hierarchies.

Schilit et al [34] have written some of the seminal literature on Context-Aware computing, particularly with their paper "Context-aware Computing Applications." In this work they describe context as per above, and prototype Context-aware applications on a device called the ParcTab. One such application notifies users of location changes. "Proximate Selection" is also defined whereby objects located nearby to a Context-aware system are emphasised over distant ones, thereby determining the "context" of the device. It can be used from finding computing resources such as printers to finding nightclubs or ATM machines. This clearly becomes a crucial part of the typical Context-Aware system in determining what environmental information to report and what is to be disregarded.

One good general example of a context-aware computing system is Jimminy [31]. This system is presented as a "wearable note-taking and note-archival application" which displays notes in a head-mounted-display, based on aspects of a user's physical context, such as his/her current location, as well as user context, namely the people in the user's vicinity, as well as the subject and contents of any notes being written. Notes can be entered using a special one-handed keyboard connected to a wearable computer,and are tagged with context information mentioned above, while being displayed on a "Private Eye" HMD. Context is gathered by using "radio location beacons and infra-red active name badges," whose output is continuously monitored by an information-retrieval component within the system.

As mentioned in the "Technology" section, wireless sensor networks are a popular choice of context-aware platform, and are used in a diverse range of applications. Mainwaring et al [20] present a habitat-monitoring system comprising a wireless sensor network which used to evaluate the manner in which certain types of bird use nesting burrows and how their nesting environment changes over time, with respect to anthropogenic interference. They used acyclically-enclosed Mica motes to perform these measurements which were networked together and co-ordinated with a base station. Taylor et al [38] implement a wireless sensor network used to measure the electrocardiograph (ECG) readings of patients in hospital using tMote Sky sensor nodes connected to small patches worn by the user. This system has the advantage of using small, inexpensive components which can provide a constant stream of data to anybody who needs it in a hospital environment, as opposed to using traditionally expensive, bulky equipment useful only to specialists.

2.2.3 Problems & Challenges of Augmented Reality and Context-Aware Computing

Although Context-Aware Computing offer a novel human-computer interaction paradigm, they present a challenging environment for implementers. The sudden availability of huge numbers of different input types that Context-Aware systems present pose a challenge in refining this information into a form that is useful to users of a Context-Aware system. Li. et al. [43] define "Modelling the new input vocabulary of Contexts", and "Modelling Implicit and Continuous Interactions" as general challenges for developing Context-Aware Applications. In particular, the authors assert that "The use of contextual input results in a new input vocabulary that has not been deeply studied...compared to mouse and keyboard events." This implies the need for pre or post-processing on contextual data which is nontrivial.

Modelling of human behaviour within a context-aware system is difficult, due to the blatant unpredictability of human behaviour and the numerous complex motivators which drive such behaviour. Jameson et al. [17] argue that there is a danger that "the focus of attention in design may switch too completely from its traditional object, the user, to the context surrounding the user." As context-aware systems are based on inference of correct system or application behaviour from tangible contextual data, they struggle to deal with nondeterministic, abstract issues such as trust, interpersonal relationships, personal preferences etc., and this threatens to turn them into annoyances rather than assistants, if the user is not allowed participate in the decision making process. The addition of "Intelligibility" to context-aware systems has been proposed, defined as context-aware systems representing " to their users what they know, how they know it, and what they are doing about it." "Accountability" is also proposed, which suggests that context-aware systems "must enforce user accountability when, based on their inferences about the social context, they seek to mediate user actions that impact others." [8]

Sensor fusion is the process of combining data from distributed or otherwise disparate sensors with a view to improving the usefulness of the combined data over data from individual sources. Although this is a problem which largely affects wireless sensor networks, it has obvious implications for Context-Aware systems, large numbers of which, it is reasonable to assume, will use such sensors. Numerous algorithms have been proposed which attempt to fuse sensor data in a timely manner. Such algorithms have to cope

with computation, bandwidth or power constraints. One such algorithm is presented by Moura et al. [23], and uses graph modelling along with probabilistic techniques to merge data from different sensory sources. The results of experiments detailed in the paper show that for 150 sensors and 200 targets, the algorithm successfully merges the data from those sensors which is impressive to say the least.

Another challenge related to context-aware applications in wireless ad-hoc environments such as wireless sensor networks is that of meeting hard real-time deadlines. Such deadlines may be imposed in Augmented Reality scenarios where latency or lag may have disastrous consequences or at least severely depreciate the quality of the data being received. Boulkenafed et al. [9] state that "Mobile context-aware applications in the ubiquitous computing domain are challenged by the dynamism and flexibility of today's wireless networks," referring to the dynamic nature of wireless network topologies and their inherent constraints. It builds upon the "Sentient Object" model defined in [14] and approaches the problem on the application, event-based middleware and routing layers by "reducing the area of the environment where real-time communication is guaranteed to within the defined proximity bounds only."

One final issue which context-aware devices face is efficient resource management. Typically, the devices have highly constrained computation capabilities, memory and power sources, so the onus is on designers/manufacturers and context-aware network implementers to miminise the costs associated with all three. Some research has however been performed to determine ways of minimising power drain and using the resources available in context-aware devices more efficiently. Murphy [24] examined, through experimentation, the power-consumption patterns of wireless sensors under different scenarios, such as constant radio transmission and constant CPU activity. He then presented a "self-preservation module" which runs on sensors and allows them to monitor their power consumption and react to any changes therein.

Augmented Reality systems also pose a number of challenging design issues. Azuma [6]identified some of the key sources of errors in AR. By far the biggest problem is the so called "Registration Problem," that is, keeping the real and virtual data aligned relative to each other from the point of view of a user. If registration is not constant, the illusion of synthesis of real and computer data is lost. It is often caused by the fact that see-through HMD displays are incapable of displaying data as accurately as the fovea (the central part of the retina) can perceive it. The author also states that registration errors

are difficult to manage due to numerous sources of static and dynamic error causing it. He defined optical distortion, tracking & sensing errors and mechanical misalignments between the combiners, optics, and monitors in a see-through HMD as the most culpable types of static error contributing to registration problems.

The chief source of dynamic errors is lag or "the time difference between the moment that the tracking system measures the position and orientation of the viewpoint to the moment when the generated images corresponding to that position and orientation appear in the displays." Four techniques in particular for reducing the effect of lag are highlighted: reducing the lag itself completely, which is very difficult if not impossible; reducing the apparent lag, by using image deflection, where an image generator renders an image larger than required, and at scanout, uses orientation values to determine which part of the image is required; matching temporal streams can be useful, by synchronising streams from the video camera and digitisation hardware, and finally using prediction to try and reduce dynamic errors,although this appears to work better with short system delays than longer ones.

Additionally, vision-based techniques such as the use of "fiducials" or special markers have been shown to have some success with assisting with registration by allowing the Augmented Reality system to "determine the relative projective relationship between the objects in the environment and the video camera."

2.2.4 Summary

In this section, an overview of research relevant to this dissertation was presented, including work performed on augmented reality systems, context-aware computing applications and technologies. Current challenges and topics requiring future work in both areas were also presented, as well as a rationale for integrating technologies and approaches from both disciplines.

Chapter 3

Design

The previous chapters have defined and discussed context-aware computing and augmented reality, in terms of the technologies used, existing implementations and problems and challenges relating to designing systems of both types. We have also examined the current limitations of methods of human-computer interaction. This chapter discusses the design of the CAAR context-aware augmented reality system, as well as technical requirements and the software architecture of the system.

3.1 Project Requirements

From a high-level perspective, the the high level goals of CAAR are to present context information, in the form of wireless sensor network data visually, by using a closed-view head mounted display, with the aim of providing an immersive, ease-to-use and visually pleasing human-computer interaction paradigm. Chapter 1 illustrated some of the key objectives of the project, but was not very specific in design and implementation details. Requirements were ascertained by evaluating previous implementations of wireless sensor networks and context-aware computing systems, in conjunction with consultation with academic staff at Trinity College. Given that a limited timespan was available to develop the project, a particular subset of all the features and properties available to both wireless sensor networks and context-aware computing systems was chosen. After ascertaining the feature set of the system to be built, requirements were subsequently decomposed into four distinct functions that the system should provide, which are detailed as follows:

- An implementation of a wireless-sensor network which can route sensor data from an arbitrary node in the network to a particular network endpoint as long as the node is in range of another.
- Development of software to receive and process transmitted sensor messages, converting it from the raw binary values the sensors provide into a form suitable for human consumption.
- Development of an interface between the wireless sensor network components and an augmented-reality component, facilitating the visualisation of sensor data in a relatively easy manner.
- The actual visualisation of sensor data, using readily available software and hardware components, to present an immersive and intuitive interactive environment.

In addition to the actual specification of what the CAAR system should do, a series of non-functional requirements were considered:

- Accessibility and ease of use, requiring little or no instruction in advance. The user should be able to put on the head-mounted display and begin using the system almost immediately.
- Extensibility was a priority as the system offers a basis for performing much expansion in the future using different visualisation hardware and sensor technology.

The following sections of this chapter discuss the system design developed to satisfy the requirements state above

3.1.1 High-Level System Design

A vague and very high-level design of the CAAR system was formalised early on to make development and development decisions easier and more informed. The system needed a method of collecting context, processing it and then supplying it to a visualisation component. This overview is presented diagrammatically in Figure 3.1

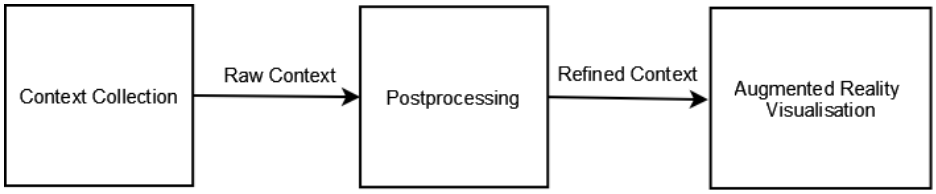

Figure 3.1: High Level Architecture of CAAR

This diagram introduces three main features of the CAAR system - context gathering, context processing and the augmented reality visualisation component. Each will be discussed below in terms of how they satisfy the project requirements listed above,

Gathering Context

Context acquisition is achieved through an implementation of a wireless sensor network. Each wireless sensor is running the same copy of specially-written software which performs the environmental sensing and radio transmission of sensor messages. A special sensor node, referred to from here on as a "sink", acts as the endpoint for all sensor messages sent, by listening for messages sent by all other sensors and interfacing with and uploading them to a processing node (in this case a laptop). The sink also periodically sends out "hello" messages alerting sensor nodes to the sink's existence. The implemented routing algorithm uses hello messages to help sensor nodes to determine who their neighbours are and which ones to use when forwarding their sensor readings. Each sensor's basic purpose is to take a sensor reading, and forward it on to the sink, while forwarding on messages from sensors that are out of radio range of the sink. Once they have arrived at the sink, they are transmitted via serial connections to the processing node, where the readings are converted into usable engineering units such as Celsius and Lux, before being submitted to the visualisation module of the Augmented-Reality component. For the purposes of this dissertation, only the thermometry, photometry and hygrometry functions of the sensors were used, as these environmental parameters are generally stable, tending to change reasonably slowly over time naturally. In addition, it was important that the chosen environmental parameters were easy to artificially manipulate for the purposes of demonstrating the project i.e. heating a room with radiators or turning off a light switch.

Each sensor node runs a copy of specially written software which takes a sensor reading, transmits it via radio to the sink node or another sensor node, and also forwards

on other nodes' sensor readings or "hello" messages from the sink. The sink node runs software to issue "hello" messages every two seconds, while listening for radio messages from sensor nodes and sending them to the processing node over a serial interface connection. The exact implementation details of the software running on the sensors, as well as the messaging format used by the sensors to communicate will be discussed in the next chapter.

Postprocessing

Postprocessing refers to the conversion of sensor data from the raw binary values that are received by the sink into usable engineering units such as Celsius. As postprocessing is a relatively expensive operation in computational terms, and does not depend on any specific sensor hardware, it was decided to implement it for a standard PC laptop running Microsoft Windows. Essentially the process involves parsing the radio frames received by the sink sensor node, extracting the sensor readings and performing complex binary arithmetic on the values to convert them into usable data.

Data is received from the sink over a USB-to-serial port converter in the form of 24 byte binary messages or packets, one for each batch of sensor readings. The messages contain frame delimiters, networking metadata and finally sensor reading data itself. The Windows API supports functionality for integrating serial port support into one's software, so a program was written to asynchronously listen to the serial port connection on which the sink resides, parse any sensor messages received and extract meaningful sensor readings from them.

In addition to receiving and processing sensor messages, the postprocessing component "wraps" sensor readings inside a simple text based messaging format for the purposes of communicating with the augmented-reality software interface, which will be discussed further on.

Visualisation

The visualisation component makes use of the TATUS [25] simulator developed by Eleanor O'Neill of Trinity College, Dublin. TATUS is a "ubiquitous computing simulator" which is intended to act as a test environment for the development of ubiquitous computing applications aimed at overcoming logistical and financial barriers when testing such ap-

plications. TATUS uses a 3D simulation of the Lloyd building in Trinity College to model a "smart environment" setting, which is ideal for the purposes of this dissertation as it provides a ready-made visualisation platform for sensor data used in the CAAR system. The simulation of the Lloyd building employs the Half-Life 2 engine developed by Valve Software, as this engine has advanced support for realistic object and fluid physics as well as providing a generally high-quality and realistic visual experience which has won it much acclaim within gaming development circles. In addition, the simulator is ideal for application within the headset-based environment which will be used in this dissertation, as the headset used in this dissertation supports Half-Life 2 directly by providing special preset configurations for hardware-based depth perception and head-tracking features, which avoids the need for implementing these features from scratch. The hardware used in the visualisation will be discussed in the implementation section.

The two "heavyweight" components within the CAAR system are the context-gathering and visualisation modules, and the design of these will now be discussed in detail.

3.2 Design 1: Acquiring Context

The wireless sensor network used in CAAR is reasonably simplistic in its design and function. A sizeable amount of research has been dedicated to address perceived design issues in WSNs. such as the potential application domains for WSNs [32]. Other research has aimed at classifying different implementations of WSN middleware according to what functionality they provide, and how they meet specific classifications of quality of service (QoS) requirements in WSN's [10]. They define QoS in terms of *application-specific* and *network* QoS features, identifying message delay, jitter, and loss, network bandwidth, throughput, and latency as network quality of service characteristics, while defining data accuracy, aggregation delay, coverage, and system lifetime as characteristics of application-specific QoS.

The sensor network implemented in CAAR does not attempt to address the above issues as they are beyond the scope of this dissertation. Instead, it simply focuses on getting sensor readings from sensor nodes to the sink node. If we consider the WSN as a left-to-right arrangement of sensors, with the sink at the extreme right and the outermost sensor at the extreme left, the algorithm's job is to get sensor readings from left to right. The flow of important data in the WSN is, ideally, unidirectional, unlike AODV.

The routing algorithm used to achieve this is quite simple, and has its limitations, but is effective and was developed solely for this dissertation. Fairly sophisticated routing algorithms already exist for WSNs and ad-hoc networks, but they were deemed unnecessarily complex to implement and so, an algorithm loosely based on the AODV [27] algorithm was chosen. AODV is a so-called *reactive* routing algorithm, in that it essentially works by waiting until a node requests a route to a destination node. Other nodes which hear this request log the node they heard it from, then forward and therefore propagate this request through the network. If a node which receives the request has a route to the destination, it sends a message stating as much back to the requesting node via an intermediate one. The original requesting node then calculates the shortest route to the destination node and uses it.

Routing Algorithm Design

Before discussing the detailed design of CAAR's routing algorithm, it is useful to explain some terminology first:

- *Reachability* is the property of a sensor node X being in radio range of another node Y.
- An *outgoing node* is a sensor node, A, selected by another node, B, to act as an intermediate node through which B's messages will be sent to, with the final destination being the sink.

The WSN algorithm used for this dissertation can be described as a *hybrid* routing algorithm, combining proactive and reactive, or on-demand elements. It has a minimal, but not non-existent level of fault tolerance. Sensors are viewed as largely independent, discrete entities within the sensor network, with a minimal dependency on their peers. The algorithm differentiates between sensor nodes based on whether they are in radio range of the sink or not. Each sensor in the network is assigned a unique numerical ID, except the sink, which always has an ID of 1.

Sensor nodes prioritise direct communications to the sink over forwarding through intermediate sensor nodes. If a sensor node, which did not have reachability of the sink, suddenly does, the sensor node will assign the sink as its outgoing node and begin direct communication. Conversely, if a sensor node which initially had reachability of the sink,

suddenly does not, it will wait for a fixed amount of seconds before selecting one of its peers (whichever one it receives a forwarded "hello" message from first) as its outgoing node. This allows sensor nodes to cope with changing network topologies. Initially, the sensor network is silent - all sensor nodes remain in a state of waiting. When a sink node is activated, it periodically sends out a "hello" or "ping" message, alerting sensor nodes within radio range to its existence, recording its ID as the originator of the message, within the message itself, as illustrated in Figure 3.2.

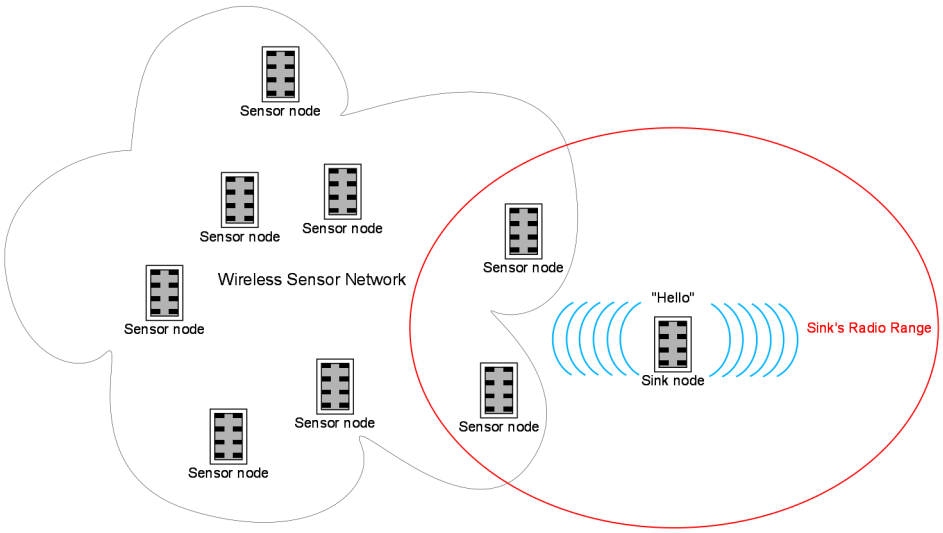

Figure 3.2: Initial Sink Activation.

This is similar to AODV's route discovery process, only it is a sink node, not a sensor node, performing the request. Once sensor nodes have received a "hello" message, they re-transmit it, inserting their unique numerical ID in the "last hop" field in the message. They then commence sensing and transmitting sensor messages, which simply contain environmental reading data. This behaviour constitutes the reactive component of the algorithm. Sensor nodes differentiate between forwarded "hello" messages and non-forwarded ones based on the value in a message's "last hop" field. If the field's value is 1, the "hello" message is not forwarded, and originated directly from the sink, and thus that sensor node has reachability of the sink, and will select the sink as its outgoing node. Otherwise, it was received from whichever node that owns the ID recorded in the "last hop" field, and thus elects that sensor node as its outgoing node, forwarding

its sensor readings along with sensor messages it received from other sensors, to that outgoing node. This is illustrated in Figure 3.3. This process occurs for *every* forwarded "hello" message ever received by a given sensor node. This means that if a sensor node without reachability of the sink has n reachable peers and n-1 fail, the sensor node will default to the remaining one as its outgoing node automatically. This does not of course apply to nodes with reachability of the sink, as they simply communicate directly to it.

The proactive element of the routing algorithm begins working once sensor nodes which received "hello" messages directly from the sink begin re-broadcasting them. With each re-broadcast, the "hello" messages will (hopefully) propagate outwards from the sink, allowing each sensor node to select an outgoing node and therefore gain a chained link, however long, to the sink. Therefore, if each node in the network has reachability of the sink, or has reachability of a node which has reachability of the sink, directly or indirectly, it will manage to get its sensor readings to the sink, not withstanding ubiquitous node failure or other unexpected network partitions.

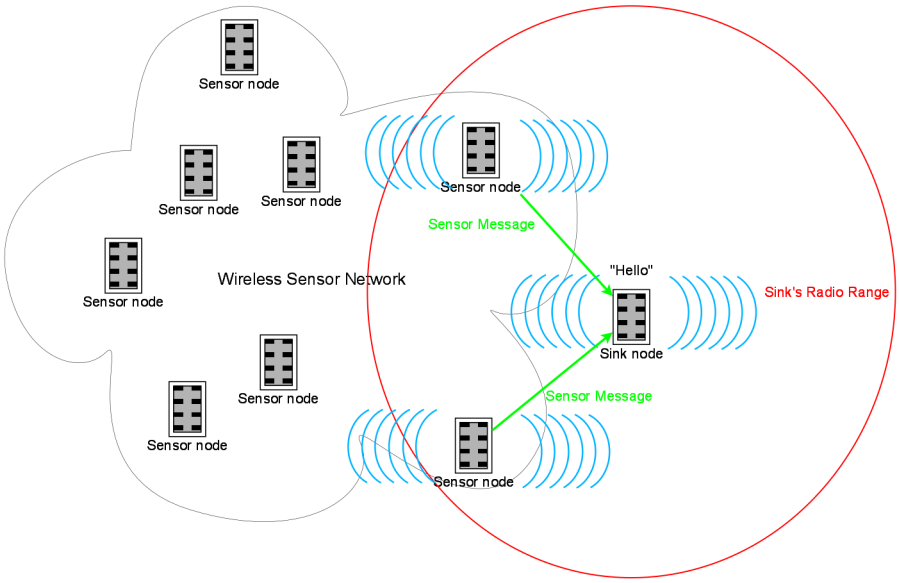

Figure 3.3: "Hello" Message Propagation.

In order to avoid the formation of routing loops, where two nodes with mutual reachability, X and Y, keep blindly firing the same message that X originally sent, back to and fro, X records its unique ID in a "pre-last hop" field, as well as the "last hop" field

in the message and sends it. When Y receives it, it inserts its ID in the "last hop" field in the message and forwards it. X will receive the forwarded message, and inspect the "pre-last hop" field. If it finds its ID in it, it will not re-transmit it, otherwise it will. Sensor nodes differentiate between forwarded sensor messages and non-forwarded ones based on a hop-count value inserted in the message, which is incremented every time a sensor message is re-broadcasted.

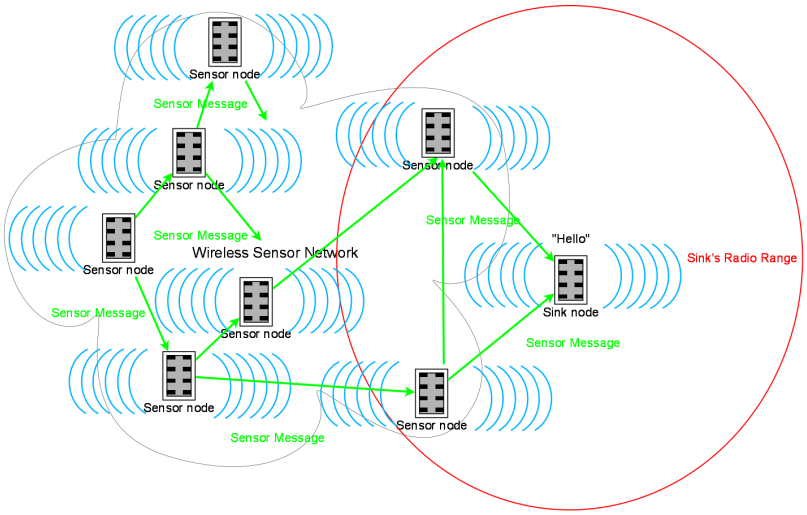

Figure 3.4: Completed route discovery process.

This algorithm has the notable advantage of requiring each node to only know about one of its neighbours, rather than maintaining a routing table which grows in direct proportion to the number of network participants. In addition, it allows nodes to move around freely *within* the network. Scenarios where the algorithm will fail are if the sink does not activate or fails during operation, or if a sensor node has not got reachability of either the sink or another peer which does, forming a network partition.

3.3 Design 2: Visualising Context

Once context is acquired and processed, the question remains of how to visualise it. This is arguably a much more difficult task, due to the myriad number of design and implementation requirements and issues which must be met and worked around, as mentioned in the "State Of The Art" section of this dissertation. If the real and virtual data cannot be

kept properly aligned or "registered," then then the illusion of the real/virtual synthesis is lost completely. Additionally, the development of intuitive representations of context is a complex process requiring research into human-computer interaction principles. In order to develop a system from scratch which met these requirements, it would require a timeframe much greater than that allowed for this dissertation, so an existing solution was chosen.

The TATUS project [25], developed initially by Eleanor O'Neill of the Knowledge and Data Engineering Group (KDEG) at Trinity College, is a "a ubiquitous computing simulator" [26] which was developed as a testing platform for ubiquitous adaptive computing applications which respond and adapt to context information. It allows implementers of adaptive systems to evaluate and test their software in an environment where context is "accurately controllable, recordable and replicable" [26] without being bogged down by the complex logistical issues and high costs associated with real-world testing.

TATUS performs simulation by using a three-dimensional virtual representation of the Lloyd Institute at Trinity College. This is visualised using the 3D graphics engine used by Valve Software in their popular videogame title "Half-Life 2." This is ideal as Half-Life 2 supports highly-advanced physics simulation, and visually realistic environments, in tandem with a freely available SDK which employs C/C++. In addition, the headset technology used for this dissertation, which will be mentioned in the "Supporting Technologies" section in the "Implementation" chapter directly supports Half-Life 2 and enhances a player's view via the use of 3D Stereoscopy and head-motion tracking. A screenshot of the simulator in action can be seen in 3.5

Figure 3.5: TATUS simulation of a lecture room in the Lloyd Institute in Trinity College

The simulator interacts with the outside world, namely the Software Under Test (SUT), via a Java-based "proxy." This proxy is initialised before the SUT is, opening TCP sockets for receiving and sending in the process, and receives software state messages from the SUT which can be used to control the ongoing simulation, while also transmitting simulator state messages to the SUT itself. Any SUT can communicate with the proxy via a simple, TCP socket-based network connection which allows the simulator and SUT to be run on different computers. Although O'Neill et al. [26] state that the expected SUT implementation platform is language, as will be mentioned in this dissertation's "Implementation" chapter, the software modules written for the CAAR system were developed in C or nesC, and managed to communicate with the Java proxy without difficulty. For the purposes of this dissertation, the simulator will be run on the same computer as the one receiving and processing sensor data.

Using this simulator to perform visualisation of computing context in the CAAR system brings a number of significant benefits. By using a simulated visual environment, as opposed to the real-world, as is the case with most augmented reality solutions, we completely bypass the problems of maintaining registration between the real and virtual data. Additionally, the tight integration between the headset technology used in this dissertation and Half-Life 2 allow head-tracking to be immediately added to the visualisation's

feature set.

Architecturally, TATUS is a "black box" as far as the CAAR system is concerned. As previously mentioned, the only point of interaction between the simulator and the outside world takes place through the Java proxy, which enormously simplifies the implementation of visualisation functionality. All the CAAR system is required to do, is "package" the sensor context received, into a format suitable for interpretation by the Java Proxy. This proxy than performs the necessary state extraction from the message, and passes it to the simulator to be visualised. The messaging format used will be discussed in the "Implementation" chapter. The simulator was modified by its original author, Eleanor O'Neill, to accept sensor reading messages from the postprocessing component in the CAAR system and perform visualisation.

Having researched and then determined the necessary feature list the system should provide, as well as what tools and technologies should be used to implement it, a final high-level design was developed as illustrated by Figure 3.6.

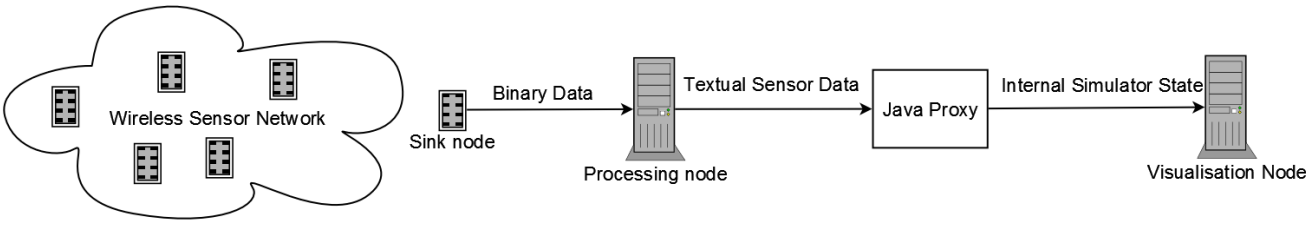

Figure 3.6: Revised High Level Architecture of CAAR

3.3.1 Summary

In this section, a review of the requirements for the CAAR system was presented, along with a textual and diagrammatical description of the how the implemented system meets those requirements. A detailed description of the core components was discussed, including the roles each component had to play and how components interacted. Additionally, a description of the iterative refinement of the design was described, as it moved from a vague high-level requirements list into a more concrete set of functionalities that CAAR should provide.

Chapter 4

Implementation

This chapter describes the implementation of the various components presented in Chapter 3. It is divided into three main sections. The first section discusses what supporting software and hardware technologies were used to build the various components. The second section provides a high-level overview of the software components in the system and the mutual relationships and interactions that exist between them. Finally section discusses the actual implementation itself, along with what challenges were encountered and their implemented workarounds.

4.1 Supporting Technologies - Software

As part of the implementation, various software technologies were used to, in turn, implement other software necessary to fulfil the requirements laid out in the design section, namely the software running on the sensor nodes, as well as the visualisation software used to display the information gathered by the sensors. This section provides a detailed exposition of these technologies individually, and details how they mutually depend on and interact with each other.

4.1.1 TinyOS

Component-Based Programming

In order to program the wireless sensor nodes used in this dissertation, an open-source, event-driven and component-based operating environment and development platform known as "TinyOS" [3] was used. Rather than dealing with individual sensor components on a hardware level, TinyOS programs use software "components", which essentially provide specific sets of functionality without bogging down the programmer in hardware-specific implementation details, instead providing a clean abstraction layer above them. For example, there are sets of components for different sensor models and product lines, and within each set, there is a component to flash a sensor's LED lights, another to transmit a radio message, yet another to receive a radio message, and so forth. As a result, a TinyOS application ends up as a single binary image composed of multiple components, which has complete control of the mote hardware.

nesC

TinyOS components are written in "nesC" (Network Embedded Systems C), a derivative of C aimed specifically towards embedded applications such as sensor networks. nesC contains the usual C syntactic and semantic constructions, but adds a concurrency model based around new execution constructs, specifically "events" and "tasks". *Events* are the calling of a hardware interrupt, which occur when a sensor node has completed sensing a particular environment parameter such as temperature or when a radio message is received, for instance. Event-handlers are therefore call-back routines which execute in response to events. Event-handlers can pre-empt one another. *Tasks* are functions whose execution is deferred or queued, and are suited to longer computational jobs. Tasks do not pre-empt each other, running to completion. They can however be pre-empted by event handlers. nesC also adds new keywords to the C language, namely `async` and `atomic`, which facilitate asynchronous code execution of events, and non-preemptibility in code execution respectively.

Components and Interfaces

TinyOS applications have two types of components. The first is a configuration file, which assembles components together, and links interfaces used by one component to interfaces provides by others. The second component is a module which contains the actual application implementation. Components are described by *interfaces*, which specify a set of *commands*, which are functions to be implemented by the interface's provider, and a set of events, which are functions to be implemented by the interface's user. If a component uses an interface, it must implement handlers for the interface's events. A top-level configuration file is required to "wire" components together, or making explicit the relationships between different components and interfaces. Although it can seem tedious at first, this component and interface-based approach has some real advantages. It's possible to construct entire applications without writing a line of code yourself, but by simply wiring pre-written components together. Additionally, the separation of concerns offered by components facilitates a more modular and design-oriented approach to building TinyOS applications.

TinyOS Toolchain

In addition to a component-based development paradigm, TinyOS provides a powerful toolchain for compiling applications and subsequently programming sensor nodes with them, based around the GNU Compiler Collection. The nesC compiler, *ncc*, is capable of performing quite advanced static analysis on program source code, including the ability to detect data race conditions, and hazardous non-atomic memory accesses. Furthermore, programs for linking, assembling and loading binaries onto sensor nodes are provided. TinyOS also provides TOSSIM, which allows developers to simulate and debug the execution of their code on sensors without actually having to upload them on to the devices themselves, with the ability to simulate a network of thousands of sensors all running the program being simulated.

4.1.2 TATUS / Half-Life 2

Please refer to the "Design 2: Visualising Context" heading in the "Design" chapter for an overview of the TATUS simulator and its use within the CAAR system.

4.2 Supporting Technologies - Hardware

As part of the implementation of this dissertation, two particular types of hardware were used - a video head-mounted display, and wireless sensors. This section will discuss both in terms of their hardware composition, and their role within the CAAR system

4.2.1 Sensor Technology

Sensor node

The sensor technology used for this dissertation is the MicaZ sensor (See Figure 4.1), manufactured by Crossbow Technology [5]. The physical dimensions of a MicaZ are 58mm x 32mm which makes them very portable. Each mote consists of a 7.3 MHz ATmega128L processor, 128KB of code memory, 4KB of data memory, and a Chipcon CC2420 radio which supports the 802.15.4/ZigBee [4] WPAN protocol, transmits at up to 250 kilobits per second and has an outdoor transmission range of approximately 30m. The motes are supported as a sensor platform by TinyOS 1.1.17 with a range of pre-existing software components already available for them. MicaZ's are powered by two AA batteries. The motes do not have any sensing capability "off the shelf," instead relying on an external (and separately sold) sensorboard. As output the sensors have three LEDs, a red, yellow and green one. These were used extensively throughout the debugging process. MicaZs also come equipped with a piezoelectric sounder, although this was never used. The motes are programmed using a special "programming board," which a mote slots into and is flashed using special software tools provided by TinyOS. The programming board connects to a laptop or desktop computer using a USB cable, although it is detected as a COM device, so special drivers, available on the Internet, are required. The sink node remains plugged into the programming board

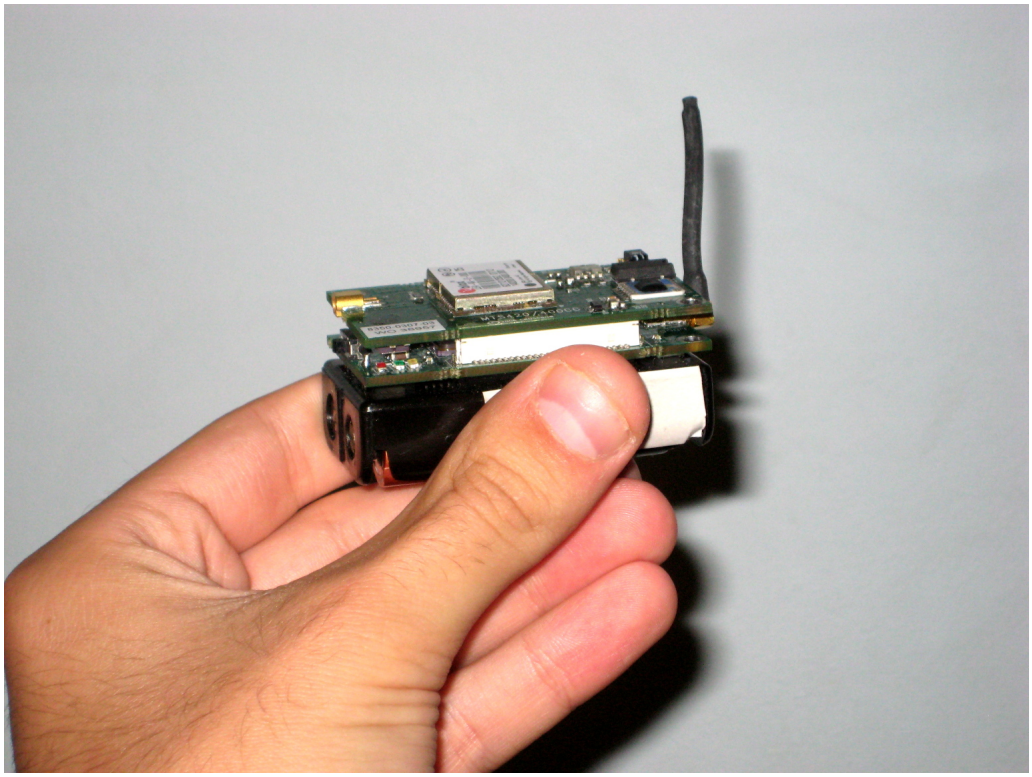

Figure 4.1: A MicaZ sensor node with attached sensorboard

Sink node

The sink node is identical to a sensor node in terms of the hardware used, only the software running on it is different. The sink node remains permanently connected to the programming board mentioned above, and can be seen below:

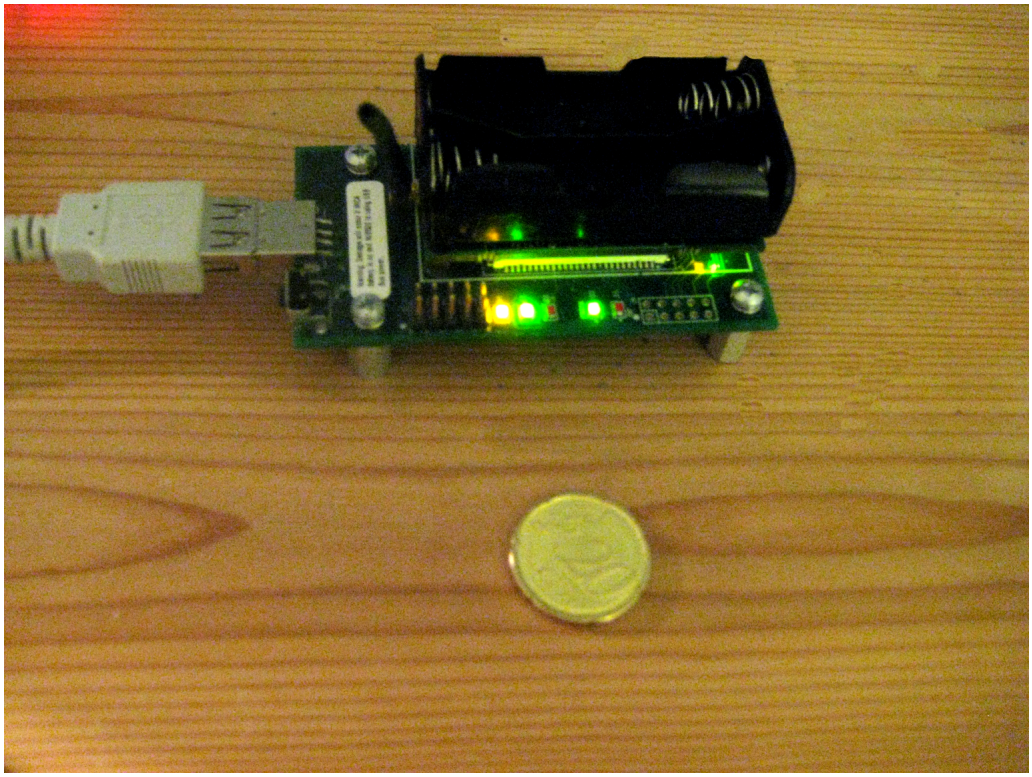

Figure 4.2: Sink node on programming board

Sensorboard

As mentioned above, MicaZ motes need an external sensorboard to sample environmental context. The sensorboard chosen for this dissertation was the Crossbow MTS420 board(). It has GPS, barometry (air pressure), hygrometry (relative humidity), photometry (ambient light), thermometry (temperature) and accerometry (acceleration force) sensing capabilities [39]. For this dissertation hygrometry, photometry and thermometry were chosen as environmental context to visualise, due to their relative non-changeability and the ease with which they can be artificially manipulated. The sensorboard connects to a MicaZ mote via a JTAG connector, and sits on top of the mote itself.

Figure 4.3: An MTS420 sensorboard

4.2.2 Display Technology

The HMD unit chosen for this dissertation is a Vuzix iWear VR920 headset. This unit is a closed video head-mounted display. It uses two progressive scan 640x480 pixel LCD displays to present imagery, and is equipped with tri-axial head-tracking for changing the user's view of the virtual world in accordance with their head-movements. The display is also equipped with 3D stereoscopy, which provides an illusion of depth in the visuals being presented by presenting a slight offset between the video on the left and right monitors, presenting video in the same fashion our eyes perceive the real world, unlike a computer monitor which provides only one point of view. The 3D stereoscopy feature directly integrates into many popular PC games, including Half-Life 2, and it adds an extra level of realism to the virtual world being created.

Figure 4.4: A Vuzix iWear VR920 headset

4.3 Implemented Technologies

Recall back to the design chapter that identified four specific features that the system should provide. These included gathering context, processing, interfacing the context with an augmented reality subsystem and finally visualising the context. In order to meet these requirements, a specific set of software components needed to be developed, essentially one for each requirement. Each component will now be described in detail, including code extracts where appropriate and particular implementation motivations.

4.4 Contextifier & newTOSBase

The first part of the implementation to be developed was the context-acquisition functionality, which involved developing two TinyOS components, "Contextifier" and "newTOSBase." Contextifier is the component which runs on sensor nodes, sensing and transmitting environmental data while forwarding readings from other sensor nodes also. newTOSBase is a modified version of a pre-existing TinyOS component, "TOSBase" which

originally received sensor readings from sensor nodes and transferred them to a host computer via serial port. The modifications made in newTOSBase relate to the addition of a function to periodically send out "hello" messages which activate an dormant initially dormant sensor network, and facilitate route discovery throughout the network. Please note that mote and sensor node are used interchangeably thoughout this section.

4.4.1 Getting Context

TinyOS provides components for using the humidity, light and temperature sensors on MicaZ sensor nodes: they are `TaosPhoto` for the light sensor, and then `SensirionHumidity` for humidity and temperature. TinyOS provides other components for MicaZ motes, but they were not used in this dissertation. In terms of external interfaces, Contextifier uses the interfaces illustrated in Code Extract 1.

Code Extract 1 Interface usage and provision declarations for Contextifier

```
Module Contextifier{
   provides interface StdControl;
}
   uses{
        interface ADC as Humidity;
        /* commands for starting /stopping humidity sensor */
        interface ADC as Temperature;
        /* commands for starting /stopping temperature sensor */
        interface ADCError as HumidityError;
        /* events to handle humidity sensor errors */
        interface ADCError as TemperatureError;
        /* events to handle temperature sensor errors */
        interface SendMsg as Send;
        /* commands for sending radio messages */
        interface ReceiveMsg as Receive;
        /* commands for receiving radio messages */
        interface StdControl as CommControl;
        /* wiring of communication component */
        interface Timer;
        /* provides timer functionality */
        interface Leds;
        /* provides commands for controlling LEDs */
        interface SplitControl as TaosControl;
        /* "on/off" functionality for sensors */
        interface ADC as TaosCh0;
        /* commands for using channel 1 of light sensor */
        interface ADC as TaosCh1;
        /* commands for using channel 2 of light sensor */
   }
}
```

In code extract 1, we see a declaration for a module, one of the two mandatory parts of a TinyOS component.`HumidityError` and `TemperatureError` describe events for when sensors perform erroneous behaviour, which must be handled by the using component. All the other interfaces just provide different functionalities through commands, which were described earlier in this chapter as functions to be provided by the interface's provider. The `as` keyword simply renames interfaces to a more desirable identifier, similar to the `typedef` keyword in C. Each node maintains two state variables, `next_hop_id`, which represents the current node's outgoing node, which is a peer assigned to receive all messages sent by the current node , and `timeout` which records the number of seconds which have passed since last receiving a "hello" message from the sink directly, without having been forwarded. Both of these variables are critical to routing decisions made by the mote, as will be shown later on.

The only interface which Contextifier provides is `StdControl`, which is TinyOS's standard control interface, used in any component which requires initialisation or the ability to "power down." `StdControl` describes three functions, called `init()`, `start()` and `stop()`, which must be implemented by any component which uses the interface. `init()` "initialises" a component and its subcomponents, allowing it to initialise variables and initialise other sensor components . `start()` is used to turn on and start a component and its subcomponents.. `stop()` is the obvious antithesis of `start()`, halting a component and its subcomponents. The implementations of these three functions can be seen in code extracts 2,3 and 4 respectively.

Code Extract 2 Implementation of `StdControl.init()`

```
command result_t StdControl.init() {
   /* use "atomic" keyword to make block memory accesses non-preemptible */
   atomic {
      msg_ptr = &msg_buf;
      /* a buffer to hold radio message data */
      fwdMsg_ptr = &fwdMsg;
      /* a pointer to a buffer holding forwarded sensor message data */
      fwdHelloPtr = &fwdHello;
      /* a pointer to a buffer holding forwarded "hello" message data */
      sensorMsg = (GenericMsg *)msg_ptr->data;
      /* pointer to buffer holding sensor data */
      timeout = 0;
      /*interval passed since last receipt of hello message directly from sink */
     next_hop_id = 0;
     /* id of node to send messages to */
   }
   call CommControl.init();
   /* initialise radio component */
   call TaosControl.init();
   /* initialise light sensor component */
   call TempHumControl.init();
   /* initialise Temperature/Humidity Sensor component */
   return SUCCESS;
}
```

Code Extract 3 Implementation of `StdControl.start()`

```
   command result_t StdControl.start() {
      call HumidityError.enable();           // catch humidity sensor hangs
      call TemperatureError.enable();        // catch temperature sensor hangs
      call Timer.start(TIMER_REPEAT, 5000);  // start timer (fires every 5 secs)
      call CommControl.start();              // start radio
      return SUCCESS;
}
```

Code Extract 4 Implementation of `StdControl.stop()`

```
   command result_t StdControl.stop() {
      call CommControl.stop();
      call TaosControl.stop();
      return SUCCESS;
   }
```

A timer is a useful way to regulate executions of particular code sections within TinyOS components. In Contextifer, a timer "fires" every five seconds, which begins a new iteration of sensor measurements and result transmissions. In order to have periodic code execution, one must handle the `Timer.fired()` event as illustrated in code extract 5.

Code Extract 5 Implementation of `fired()` event handler for Contextifier's Timer

```
  event result_t Timer.fired() {
     timeout++;    //increment interval since last "hello" message from the sink
     call TempHumControl.start(); //start humidity sensor
     return SUCCESS;
}
```

In the above code extract, every time the timer fires, a counter is incremented. This counter records how long it has been since a sensor received a "hello" message directly from the sink, not from an intermediate node. If more than 5 incrementations occur, the sensor starts addressing all its radio traffic to intermediate sensors, not the sink, with the underlying assumption being that the mote has moved out of radio range of the sink. This allows sensors to move in and out of range of the sink at will.

The code fragment above also demonstrates how sensor components are started, by calling their `start()` command. However, event handlers signifying the successful starting of components must be handled - in Contexifier, events are used to start the sensing process, as shown in code extract 6:

Code Extract 6 Implementation of `startDone()` event handler for humidity sensor

```
/* the temperature and humidity sensor has started correctly,
   so get humidity readings */

event result_t TempHumControl.startDone() {
    call Humidity.getData();
    return SUCCESS;
}
```

Once humidity sensor data has been captured successfully, we handle the `dataReady()` event for the humidity sensor and store the captured data in a buffer reserved for outgoing sensor messages. This event then invokes the capturing of temperature sensor data. The `async` keyword is used to allow events to asynchronously pre-empt tasks and other `async` events. It should only be used where such pre-emption is necessary and safe, as is the case with a sensor reading.

Code Extract 7 Implementation of `dataReady()` event handler for humidity sensor

```
async event result_t Humidity.dataReady(uint16_t data) {
   sensorMsg->humidity = data;        //store captured humidity data
   return call Temperature.getData(); //get temperature data
}
```

Similarly, once temperature data has been captured, the `dataReady()` event for the temperature sensor is called and the data is stored. For some undetermined reason, the light sensor would not function correctly if its `start()` command is called from `StdControl.start()`. It does work however if started after the temperature sensor has captured data and been stopped:

Code Extract 8 Implementation of `dataReady()` event handler for temperature sensor

```
async event result_t Temperature.dataReady(uint16_t data) {
   sensorMsg->temperature = data;
   call TempHumControl.stop();
   return SUCCESS;
}
event result_t TempHumControl.stopDone() {
   call TaosControl.start();
   return SUCCESS;
}
```

The MicaZ light sensor has two "channels" for receiving data, so there are separate `dataReady()` events for both:

Code Extract 9 Getting light readings

```
 async event result_t Temperature.dataReady(uint16_t data) {
    sensorMsg->temperature = data;
    call TempHumControl.stop(); // stop temp/humidity sensor
    return SUCCESS;
}

event result_t TempHumControl.stopDone() {
   call TaosControl.start(); // temp/humidity sensor stopped correctly -
                             // start light sensor - get readings
   return SUCCESS;
}

event result_t TaosControl.startDone(){
   call Leds.redOn();     //blink red LED to mark temp/humidity capture
   return call TaosCh0.getData(); // get data from first light sensor channel
}
async event result_t TaosCh0.dataReady(uint16_t data) {
    sensorMsg->lightch0 = data & 0x00ff;  // get and store low order byte - requi
    return call TaosCh1.getData();        // get second light sensor channel's re
}

async event result_t TaosCh1.dataReady(uint16_t data) {
   sensorMsg->lightch1 =  data & 0x00ff;
   post send_sensor_msg();      // we've gotten all required sensor data;
                                // send a sensor message
   call Leds.redOff();
   return SUCCESS;
}
```

The block of code above reads current light sensor values, stores them and then "posts" a sensor message transmission. The `post` keyword is synonymous with queueing

invocations of tasks. Recall that tasks are functions which are queued in a FIFO order and scheduled for execution by TinyOS.

4.4.2 Sending Context

Once context has been acquired, Contexifier will attempt to send it using a task, `send_sensor_msg()` designed for the purpose. Before we discuss the implementation of that task, it is useful to describe the messaging format that Contextifer uses. A purpose written message data structure is used for the purposes of storing data and facilitating the sensor node routing algorithm, and is defined below as a standard C structure. The enumeration defines constant values used to differentiate between sensor reading messages and "hello" messages, by inserting the appropriate value in the `msg_type` field.

Code Extract 10 GenericMsg - the messaging format used by Contextifier

```
typedef struct
{
   //uint8_t is a renamed unsigned byte
   //uint16_t is a renamed unsigned short

   uint8_t msg_type;    //am I a sensor or hello message?
   uint8_t bcast_id;    //id of sensor node sending message
   uint8_t dest_addr;   //id of destination sensor node
   uint8_t hop_count;   //how many times have I been forwarded?
   uint8_t last_hop_id; //who last forwarded me?
   uint8_t pre_last_hop_id; //who forwarded me before that? - prevents routing loop
   uint16_t humidity;   //humidity sensor data
   uint16_t temperature; //temperature sensor data
   uint16_t lightch0;   //first light sensor channel data
   uint16_t lightch1;   //second light sensor channel data
}GenericMsg;

enum{ //-> used by msg_type field above
   HELLO = 1, // it's a hello message
   SENSOR = 2, // it's a sensor message
};
```

The format described above constitutes the data "payload" for another, pre-defined messaging format called "TOS_Msg," that TinyOS uses to facilitate mote-to-mote communications. The exact structure of this format varies between different mote platforms, but for MicaZ motes it is as described in code extract 11:

Code Extract 11 TOS_Msg - prescribed messaging format used by TinyOS

```
typedef struct TOS_Msg {
    // The following fields are transmitted/received on the radio.
    uint8_t length;
    uint8_t fcfhi;
    uint8_t fcflo;
    uint8_t dsn;
    uint16_t destpan;
    uint16_t addr;
    uint8_t type;
    uint8_t group;
    int8_t data[TOSH_DATA_LENGTH]; //GenericMsgs go in here
                                   //Contextifier doesn't use any other TOS_Msg fie

    // The following fields are not actually transmitted or received
    // on the radio! They are used for internal accounting only.
    // The reason they are in this structure is that the AM interface
    // requires them to be part of the TOS_Msg that is passed to
    // send/receive operations.

    uint8_t strength;
    uint8_t lqi;
    bool crc;
    uint8_t ack;
    uint16_t time;
   } TOS_Msg;
```

GenericMsgs are inserted into TOS_Msgs by using casting arithmetic. This is demonstrated in code extract 2. As stated in the comments, Contextifier does not use any other TOS_Msg fields. We can see how it all fits together by looking at `send_sensor_msg()`:

Code Extract 12 Sending a sensor message with `send_sensor_msg()`

```
task void send_sensor_msg(){
   if(next_hop_id == 0)return; // if 0, sensor network has not been activated ye
   sensorMsg->msg_type = SENSOR; // set message type
   sensorMsg->bcast_id = TOS_LOCAL_ADDRESS; //a constant - this mote's numerical
   sensorMsg->dest_addr = next_hop_id; //message destination
   sensorMsg->hop_count = 0; //number of times forwarded

   // try to send msg - yellow LED goes off if successful

   call Leds.yellowOn(); // turn on yellow LED and send message

   if((call Send.send(TOS_BCAST_ADDR,sizeof(GenericMsg),msg_ptr)==SUCCESS)){
      call Leds.yellowOff();  //if message was sent, turn off yellow LED
   }
   return;
 }
```

In order to send a sensor message, the mote checks to see if it has determined who it should be sending it to - either the sink or another mote. Recall that the `next_hop_id` variable stores the id of the mote's *outgoing node*. if it's 0, the mote has not received a "hello" message from the sink, forwarded or non-forwarded and so remains dormant. Otherwise, the mote sets the sensor type to be a sensor message, inserts its numerical ID (assigned using the TinyOS compilation string for Contextifier), sets the identifier for the mote it wants to receive the message and sets the hop count to zero, as it is a new message. The mote activates it's yellow LED and tries to send the sensor message. If successful, the LED goes off, otherwise, it stays on and we know there has been an error. A `Send.sendDone()` event handles successful transmission of the message.

4.4.3 Routing Context

By far the most complex part of Contextifier is the routing component. It is necessarily complex due to the large number of different circumstances it has to cater for. The `Receive.receive()` event performs all message routing management, checking incoming message types, destinations and originators, while another task, `fwd_msg()` performs the actual forwarding of messages that didn't originate from the mote it is running on. As stated before, every sensor node has a unique numerical ID, apart from the sink, which has a node ID of 1.

The first scenario the routing algorithm covers is receipt of a "hello" message directly from the sink, without any forwarding. If this occurs, the mote sets the sink to be its *outgoing node* and addresses all messages to it. It checks the `last_hop_id` field of the TOS_Msg containing the "hello" message, and if it is set to 1, then it was sent by the sink directly, as the only mote with that numerical ID is the sink, and the only message the sink can send is a "hello" message. It also resets the `timeout` variable to signify we are currently in radio range of the sink. This behaviour means that a sensor node will *always* choose the sink as its outgoing node over a peer, if given the choice.

Code Extract 13 Routing Scenario 1: Node is in radio range of the sink

```
event TOS_MsgPtr Receive.receive(TOS_MsgPtr msg) {
//we got a message from the base station so we're in range
   if(((GenericMsg *)msg->data)->last_hop_id == 1){ //the sink was the last hop
      atomic next_hop_id = 1; //select the sink as ourgoing node
      atomic timeout = 0; //reset sink message interval timeout
```

The next situation the routing component covers is receipt of a message for the first time by a dormant sensor node. The sensor node in this scenario hasn't selected an outgoing node (the `next_hop_id` variable is set to 0), and the node which sent the message has a numerical id. The mote in receipt of the message therefore chooses the sender of the message as its outgoing node and can begin sensing and transmitting readings.

Code Extract 14 Routing Scenario 2: Dormant Node receives a message for the first time.

```
//we got a message from another mote so we can fwd through it
  else if (next_hop_id == 0 &&((GenericMsg *)msg->data)->last_hop_id!=0){
     atomic next_hop_id = ((GenericMsg *)msg->data)->last_hop_id;
  }
```

The third routing scenario is, having had the sink selected as the current node's outgoing node, the passing of more than five seconds since the current sensor node received a non-forwarded "hello" message from the sink, coupled with a subsequent receipt of a message from a peer. As a result, the current node simply selects its outgoing node as the sensor node identified in the message just received.

Code Extract 15 Routing Scenario 3: Node has not heard from the sink in over 5 seconds - so revert to using a peer

```
//we haven't heard from the base station in a while so go back to forwarding via pe
   else if (timeout > 5 && next_hop_id == 1 &&((GenericMsg *)msg->data)->last_hop_i
      atomic next_hop_id = ((GenericMsg *)msg->data)->last_hop_id;
   }
```

The first three scenarios manage a node's routing decisions based on who sent a recently received message. The next and final two scenarios base decisions around what type of message was received, not who sent it. The first of these two scenarios is the receipt of a sensor message from a mote who has elected the current one as its outgoing node. The receiving mote checks the `msg_type` field in the received message's `data` field to see that the message is addressed to itself and then posts a task, `fwd_msg()`, responsible for forwarding messages not originating from the current mote. Before posting the task, the mote turns on its red LED. If the message is forwarded correctly, the LED is deactivated.

```
switch(((GenericMsg *)msg->data)->msg_type){  //check the message type
   case SENSOR:
      memcpy(fwdMsg_ptr->data,(GenericMsg *)msg->data,sizeof(GenericMsg))
      // if sensor msg is specifically addressed to us, forward it on
      if(((GenericMsg *)fwdMsg_ptr->data)->dest_addr == TOS_LOCAL_ADDRESS){
         call Leds.redOn(); // turn on red LED
         post fwd_msg();
      }
   break;
```

It is useful at this point to see how `fwd_msg` works to put the implementation of the two final routing scenarios in context.

Code Extract 16 fwd_msg: forward a message received from another mote

```
task void fwd_msg() {
   ((GenericMsg *)fwdMsg_ptr->data)->hop_count+=1;
   ((GenericMsg *)fwdMsg_ptr->data)->dest_addr = next_hop_id;
   // this part prevents routing loops forming
   ((GenericMsg *)fwdMsg_ptr->data)->pre_last_hop_id = ((GenericMsg *)fwdMsg_ptr->d
   ((GenericMsg *)fwdMsg_ptr->data)->last_hop_id = TOS_LOCAL_ADDRESS;
   while(i < rand() % 799 +200)i++;
   i = 0;
   if((call Send.send(TOS_BCAST_ADDR,sizeof(GenericMsg),fwdMsg_ptr))==SUCCESS){
      call Leds.greenOff();
   }
}
```

`fwd_msg()` works by taking the received message and incrementing its internal hop counter. It fills in the id of the intended recipient, and stamps the message with its own numerical id in two fields, `pre_last_hop_id` and `last_hop_id`. This prevents routing loops, where two nodes mindlessly forward the same message back and forth indefinitely,

as upon receiving a message, a mote checks to see that it wasn't the message's last hop, or the hop before the last, and only retransmits if both evaluations return negative. This prevents the scenario of mote A sending a message to mote B, then B retransmitting it to A, and A retransmitting to B ad infinitum.

The fifth and final routing scenario is receipt of a "hello" message. The recipient mote first checks to see that the message received is not just a forwarded copy of one it already forwarded in the recent past, to avoid routing loops. This is achieved by examining the `pre_last_hop_id` and `last_hop_id` fields mentioned above. All being well, it then forwards the hello message to anyone in range using the `fwd_msg()` task.

Code Extract 17 Routing Scenario 5: Receipt of a hello message

```
   case HELLO://we received a hello message from base stn., possibly forwarded so f
      memcpy(fwdMsg_ptr->data,(GenericMsg *)msg->data,sizeof(GenericMsg));
      if(((GenericMsg *)fwdMsg_ptr->data)->last_hop_id!= TOS_LOCAL_ADDRESS &&
         ((GenericMsg *)fwdMsg_ptr->data)->pre_last_hop_id!= TOS_LOCAL_ADDRESS){
            call Leds.greenOn();
            post fwd_msg();
      }
   break;
}
return msg;
```

This marks the end of the discussion on the Contextifier module. As mentioned before, TinyOS components have two distinct parts, a module, and a top-level configuration file, but the latter is not relevant to the implementation itself or provide insight into the problems it addresses, so it will not be discussed. The other context-acquisition module used is newTOSBase, a trivial modification of a freely available TinyOS component, TOSBase, which provides "base station" functionality, allowing a mote to connect to a desktop or laptop computer via a serial port and upload all received messages from the sensor network to programs listening to the serial port. The addition of sending "hello" messages every second which take the form of GenericMsgs, is the only difference between newTOSBase and TOSBase.

4.5 Processing Context

Once the newTOSBase TinyOS component has uploaded a sensor message to the host computer's serial port, the sensor readings must be processed, as they are not in a form suitable for general consumption. This requires two stages - parsing radio messages and extracting the sensor data, and then converting that data into usable engineering units. This is achieved using a Microsoft Windows program called "BaseStation", written in C, for this purpose. The program uses the Windows API to access the serial port, and then uses special functions based on source from TinyOS to convert the sensor data.

Accessing the serial port and parsing the messages are reasonably trivial, as there are examples on how to do the former using Microsoft's documentation. The `ReadFile()` Windows API function allows a program to asychronously receive serial port data on a byte-by-byte basis, which is more efficient than constantly trying to read data from the port. Parsing sensor messages from the sink involves using flag bytes in the radio messages to determine when an entire message is received, and then parsing the contents of that message. A message is received as a 24 byte array delimited by the hexademical digit `7E`. A special function, `parseMessage`, processes this message after receipt and extracts meaningful sensor data from it, using binary shifting and arithmetic, some of which is shown below:

Code Extract 18 Converting raw sensor data into engineering units

```
struct tempsensor{
   uint16_t humidity;
   uint16_t temp;
   uint16_t lightch0;
   uint16_t lightch1;
};
void parseMessage(unsigned char * packet){
   struct tempsensor * tsensorPtr;
   /* cast data section of buffer to correct types */
   tsensorPtr = (struct tempsensor *)&packet[13];
   /* convert temp readings into Celsius  */
   float ftemp = (float)tsensorPtr->temp;
   float tempdata = -38.4 + 0.0098 * ftemp;

   /* convert humidity readings into relative percent */
   float HumData = (float)tsensorPtr->humidity;
   float fHumidity = -4.0 + 0.0405 * HumData - 0.0000028 * HumData * HumData;
   fHumidity = (tempdata - 25.0)*(0.01 + 0.00008 * HumData) + fHumidity;
}
```

The process of converting light readings is somewhat more elaborate, as there are two sensor "channels" to be processed as shown below:

Code Extract 19 Converting light sensor data into engineering units(Lux)

```
// convert readings from Taos Light Sensor on MTS420 board
// to usable engineering units - taken from TinyOS source
float mts400_convert_light(uint16_t taosch0, uint16_t taosch1)
{
   //two sensor "channels," taosch0 and taosch1
   //converted individually and combined at the end
   unsigned ChData,CV1,CH1,ST1;
   int ACNT0,ACNT1;
   float CNT1,R,Lux;

   ChData = taosch0 & 0x00ff;
   if (ChData == 0xff) return -1.0; // Taos Ch0 data: OVERFLOW
   ST1 = ChData & 0xf;
   CH1 = (ChData & 0x70) >> 4;
   CV1 = 1 << CH1;
   CNT1 = (int)(16.5*(CV1-1)) + ST1*CV1;
   ACNT0 = (int)CNT1;

   ChData = taosch1 & 0xff;
   if (ChData == 0xff) return -1.0; // Taos Ch1 data: OVERFLOW
   ST1 = ChData & 0xf;
   CH1 = (ChData & 0x70) >> 4;
   CV1 = 1 << CH1;
   CNT1 = (int)(16.5*(CV1-1)) + ST1*CV1;
   ACNT1 = (int)CNT1;

   R = ((float)ACNT1)/((float)ACNT0);
   Lux = (float)ACNT0*0.46/exp(3.13*R);
   return Lux;
}
```

Once the converted values have been determined, they are converted into a textual, gradient-based representation. The reason for this is that it was decided users did not need to know the precise numerical values of sensor readings, but merely that they fall into a defined range. The gradient ranges were chosen based on an observable set of values, rather than absolute minima and maxima. The gradient values used are listed in the tables below:

Light reading ranges (in Lux)	Gradient text
0-300	Lum_1
301-600	Lum_2
601-900	Lum_3
901-1200	Lum_4
1201-1500	Lum_5
1501-1800	Lum_6
1801-2100	Lum_7

Figure 4.5: Light measurement gradients

Humidity reading ranges (in percent)	Gradient text
0-20	Hum_1
21-40	Hum_2
41-60	Hum_3
61-80	Hum_4
81-100	Hum_5

Figure 4.6: Humidity measurement gradients

Temperature reading ranges (in degrees Celsius)	Gradient text
15-18	Tem_1
19-21	Tem_2
22-25	Tem_3
26-28	Tem_4
29-30	Tem_5

Figure 4.7: Temperature measurement gradients

Concurrent to converting sensor readings into a textual format, the "BaseStation" application sends the readings over TCP Windows sockets on port 5062 to the loopback address, `127.0.0.1`. This is because a Java "proxy", which will be discussed in the next section, listens on that port for incoming sensor data, and essentially passes it to the component responsible for visualising context. As most software which employs a TCP network connection follows a standard coding "template" for achieving network connectivity, which is widely available on the internet, the TCP code will not be reproduced here.

In addition to sending sensor readings, the "BaseStation" application indirectly starts the visualisation simulator. Once a TCP connection has been established, the simulator checks to see that the "BaseStation" program can receive messages by sending the string "testmessage," which is then echoed back via TCP from "BaseStation." The simulator operates in terms of "experiments," which define a test scenario for a ubiquitous computing application. Experiment definitions are stored in XML files local to the simulator, including one for CAAR itself. Although in this instance the simulator runs on the same host as "BaseStation," this design allows software-under-test (SUT) such as CAAR to be on a different machine to the one running the simulator. The simulator sends a request for the name of an experiment definition file to use, to which "BaseStation" replies with the name of the CAAR-specific file. One the simulator receives the name of the experiment definition, it begins running. The next section will discuss the details of the visualisation implementation used for CAAR.

4.6 Visualising Context

The final part of the CAAR system's implementation is the visualisation component. It takes as input, textual sensor reading inputs, and produces a visual interpretation of those readings. As mentioned in the "Design" chapter, visualisation is achieved by using the TATUS simulator, developed by Eleanor O'Neill of the Knowledge and Data Engineering Group (KDEG) of Trinity College, Dublin, in tandem with specialised head-mounted display hardware. TATUS was devised to allow ubiquitous computing application implementors to test and evaluate their software in a controlled, simulated environment, without suffering the costs traditionally associated with doing so in the real world.

Software Visualisation

The simulator visualises a 3D model of the Lloyd institute in Trinity college using a bespoke map which is rendered using the Half-Life 2 Graphics engine. This map was modified by Eleanor O'Neill to accept input from the "BaseStation" application mentioned in the previous heading, and render it. The exact internal workings of the simulator are proprietary information, and therefore are unavailable for discussion here. However, in consultation with Eleanor, a messaging format was first devised to allow the two pieces of software to communicate; this ended up being what is now the textual gradient-based sensor format described at the end of the last section. The next step involved adding capabilities to the BaseStation program to allow it to form TCP connections with the simulator, to start the simulator itself, and then begin sending sensor messages to it.

In terms of the visualisation itself, and its appearance, several different options were considered. The visualisation is set in the model's representation of the "Lloyd Common Room." A simple text-based annotation was discussed were sensors would appear as black boxes in a room in the map, and above them would appear sensor readings in text form. However, this option was discounted due to implementation difficulties that would need to be overcome, as well as a generally unsatisfactory appearance. The next option that was considered was using lights to visualise readings, associating a light colour with a particular range of readings, for instance, yellow for cool, red for hot, light blue for low humidity, dark blue for high humidity. The sensors would appear as small boxes with three lights, one for each reading, underneath the box. An initial implementation of this proved promising, and is shown below in Figure 4.8.

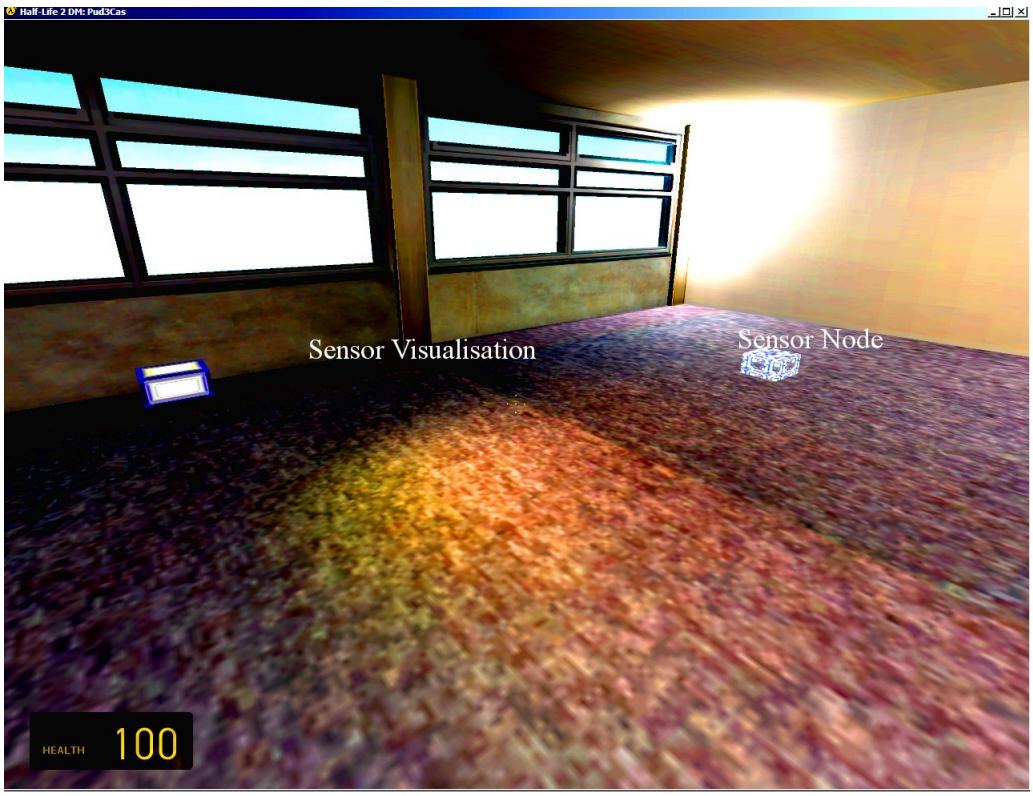

Figure 4.8: Initial visualisation implementation

Although this implementation showed promising, problems appeared in the way Half-Life 2 uses lights, such that if more than three of the lights being used for the sensors were turned on simultaneously, the lights would not appear, which was obviously an unacceptable limitation. Therefore, the map was modified again to use a different visualisation technique. The final version used blocks on a wall, with one column of blocks for each sensor reading, and the number of blocks in each column corresponding to the number of possible sensor gradient messages available for each type of reading. Each sensors' readings would therefore manifest as one batch of three columns of blocks. The blocks are coloured according to the value of the sensor reading achieved. If a sensor receives a humidity reading, a block in the right column turns blue. The higher the reading, the higher the block is in the column and bluer the colouring of the block. The colouring scheme used for the blocks was using the left column of each array of blocks for light readings, and colouring them light yellow to deep yellow. The middle column was used for temperature, going from pinkish-red to sanguine, and finally the right column was

used for humidity, going from light blue to deep blue. This is illustrated below:

Figure 4.9: Final visualisation implementation

In order to start the simulator, a set of Windows batch scripts are run, which initialise the Java proxy mentioned in the previous section which listens for incoming sensor messages, and set up an RMI server for remote procedure calls within the simulator. Once this is complete, a message displaying the simulators "ready" state is displayed in a console window. The "BaseStation" application is then invoked, with the sink sensor node plugged in to its programming board. If the sink is not plugged in, "BaseStation" will exit with an error stating that it can't find the sink node. "BaseStation" then establishes the TCP connections required to send sensor messages to the simulator's Java proxy, complaining if the ports necessary are closed, and exiting. Afterwards "BaseStation" sends configuration messages as mentioned in the previous section to configure the simulator for use and start it, and begins receiving, processing and forwarding sensor readings to the simulator. Once the simulator begins receiving these messages, it activates the relevant boxes and keeps displaying them as long as there are sensor messages to be received.

Hardware Visualisation

The head-mounted display used by the CAAR system integrates on the software level with the TATUS simulator. A special Windows process called the "iWear Monitor" allows the use of 3D stereoscopy and head-tracking in the simulator by placing a configuration file and replacing two dynamic link libraries within the Half-Life 2 program files subdirectory. The display is then activated by switching on external VGA output on the computer to which it is connected and starting the simulator using the process stated in the paragraph above. The 3D stereoscopy is perhaps under-utilised in the simulator, as there are no objects or furniture in the room that are simulated to facilitate a perception of depth, bar an open door. However, the head-tracking adds an extra degree of realism, by giving the user complete control of their environment through head movements alone.

4.7 Challenges & Implementation difficulties

The implementation of the CAAR system was not a straightforward process by any means. By far the hardest part was getting TinyOS set up and working. Although there are tutorials available online which introduce developers to the concept of components and event-based programming, most of the difficulty lies in getting the TinyOS toolchain properly installed configured. Multiple, seemingly mutually incompatible versions of TinyOS are available for download, with little or no information on the changes between them. TinyOS packages are apparently distributed with little or no testing, and as a result, very few of them work "out of the box." An installation under Linux was attempted, but the compilers had great difficulty configuring paths for including components and other necessary compilation apparatus. A virtualised operating system called "Xubuntos" is available, which is a customised version of the "Xubuntu" operating system with TinyOS tools pre-installed and configured. However, using this option requires a powerful machine capable of running the virtualised operating system.

Eventually, a Cygwin installation was partially successfully installed, so far as compiling programs and programming motes with them. Unfortunately, the principal debugging tool available to TinyOS users - the TOSSIM simulator - would not work, despite trawling through mailing lists, which, as it turns out are about the only form of support available to users. The inability to use TOSSIM effectively doubled the amount of time required

to develop the Contextifer component, as the only form of debugging available was to associate the flashing of a particular mote LED with the successful completion of a function call or setting of a variable, as well as manually inspecting dumps of received radio messages and checking values. This is problematic enough when dealing with one mote on its own, but when multiple motes are introduced, debugging becomes a process of trial and error whose complexity scales rapidly. It also meant that each and every minute change to Contextifier required re-programming the motes, introducing an unnecessary level of tedium to proceedings.

4.8 Summary

In this chapter, a detailed overview of the application components required for the CAAR system to meet the high-level requirements outlined in the previous chapter was presented. The software and hardware technologies which formed the technological foundation of the CAAR system were discussed, along with code extracts showing exactly how certain functionality was achieved were discussed and what messaging formats networked components used to communicate with each other , in addition to screenshots showing exactly how the finished system performs. Finally, a brief discussion relating to implementation difficulties was presented.

Chapter 5

Evaluation

This chapter presents an evaluation of the implementation of the CAAR system presented in the previous chapter. The evaluation is performed based on three central factors, namely a comparison between the CAAR system and other existing augmented reality systems, and what novel contribution CAAR provides. The second basis of evaluation is a comparison between initial project objectives and the final deliverables produced, and finally, through the results obtained through performing a small and informal user study on classmates, to determine how CAAR compares as a means to interact with a computing system with more traditional technologies.

5.1 Comparison between CAAR and other augmented reality systems

5.1.1 Mobile Augmented Reality System (MARS)

Existing augmented reality solutions generally depend on some form of fixed or static dataset in order to provide data to be visualised. The Mobile Augmented Reality System (MARS) [16] provides a "narrated multimedia documentary" which allows a user to move about the campus at the University of Columbia, New York, and through the use of a see-through head-mounted display, see the campus environment annotated with additional virtual data. MARS is based upon the Touring Machine [13] presented in the state of the art section. The system uses a wearable computer with an attached real-time

kinematic GPS position tracker, as well as a hand-held display device for presenting additional material. The advantages of this approach are that it provides more freedom of movement for a user, as they are not confined to one location as is currently the case with CAAR due to the use of a laptop. Additionally, the use of an optical or see-through head mounted display to annotate the real environment allows the MARS system to augment practically any environment the system can participates in. With the CAAR system, the traditional concept of augmented reality is inverted, as it annotates virtual data with real data, whereas most augmented reality systems perform the annotation of real data with virtual data. However, it was felt that this represents an interesting alternative to standard augmented reality interfaces.

However, the MARS system is limited, as it relies upon a fixed data set with which the environment can be visualised. As its developers state, " All location-based information is stored in a campus database on the backpack computer." This confines the usable location of the system to whatever locations the backpack computer has stored location based information about. Additionally, the data being presented is, in itself, unremarkable as it is information most likely available (and sourced) from other available sources such as books, maps etc. The CAAR system relies on a purely dynamic form of data - environmental context, which is not stored for later retrieval, but visualised immediately. The sensory sources of data used by the CAAR system, can be unobtrusively embedded in any environment, allowing CAAR to receive data in a much wider variety of settings than MARS. In the CAAR system, the data being presented is information which is not generally presented in a user-centric manner augmented reality systems or other similarly immersive or intuitive interactive systems. Environmental context is normally presented using thermometers or specialised meters which use monochrome LCD displays, which obviously do not facilitate a very extensive understanding of a users environment. The CAAR system presents context in a manner in which it is very directly usable to a human consumer, in order to help facilitate user understanding of his/her environment. Although MARS does use what could arguably be called environmental context, via the GPS tracking system, this information is not used for the user's consumption, but to act as a trigger for the system to provide relevant, location-based information.

5.1.2 ARQuake

ARQuake [28], developed by Wayne Piekarski and Bruce Thomas, is a modified version of the videogame Quake, intended for outdoor use. The user experiences a combination of the real world with the addition of computer generated monsters, and uses a wearable computer, head-mounted display and a gun styled as a "haptic feedback device" for providing tactile feedback. Quake was modified to accept movement instructions from a digital compass and GPS system instead of a traditional keyboard and mouse. A Quake map of the University of South Australia campus was created, and buildings modeled as solid black objects so they do not appear in the visualisation, but serve as a frame of reference for positioning monsters.

As with the MARS system, ARQuake relies on static data sources namely a Quake map and monsters to annotate the real world environment with, which again limits the environment in which the system is usable to wherever has been modelled in Quake, in this case the University of South Australia campus. However, like MARS, it allows freedom of user movement by relying on a wearable computer and GPS location tracking, rather than one computer in a fixed location, as is the case with CAAR.

5.2 Project Objectives versus Project Implementation

The initial proposal for this dissertation laid out a broad set of objectives the project could achieve, what technologies it would use and additional implementation details. Some of the research aims highlighted were "to present information regarding the signal strength of different sources such as wireless networks in an intuitive, natural manner via a head-up display" and that "data would be collected via various wireless sensors placed in opportune locations, which would report data back a portable computing device." Indeed, the original title for the dissertation was "Signal Strength as Augmented Reality." However, as the implementation proceeded it became clear that capturing wireless sensor data would be cumbersome, as most devices used for doing so, such as the Wi-Spy by MetaGeek need to be plugged in to a desktop computer, which goes against the spirit of unobtrusiveness highlighted in Marc Weiser's definition of Ubiquitous Computing, so environmental context was chosen as a data source, as this can be acquired using small,

wireless and portable sensors.

With regard to visualisation, it was intended originally to use a see-through optical head-mounted display. This would prove problematic however due to the registration problem, where the real and virtual data would be misaligned. Additionally, even if that difficulty was overcome, the display would need quite a high display resolution, to avoid display artifacts such as pixellation. Ultimately, a video head-mounted display displaying a virtual environment was chosen as this circumvents the registration problem, and would allow the implementation to be completed within the allocated timeframe.

5.3 Informal User Study

As part of the evaluation of the CAAR system, an informal user study involving four classmates was undertaken, to get some sense of how the system compares to the desktop computing paradigm as a means to interact with a computing device. The study involved a hypothetical scenario, where users of the system would be told that there was a fire in the room or building being visualised, and that using the system, they must determine which direction would be the safest to run in. Starting off, users were shown five screenshots of the system, fours displaying different readings, and one displaying no readings at all. Keystrokes were used as a means for participants to state their responses, so if they pressed the right key, they determined that moving right was the safest direction to go. The time intervals it took for participants to express their choices were timed using a program called "Recording User Input" or RUI [19], which can detect and timestamp all mouse movements and keypresses to millisecond resolution. The reaction times captured when actually using the system and seeing live visualisations would then be captured and compared to the timings obtained for static images of the system, to see if there was any noticeable difference. All participants were initially instructed as to the meanings of the different readings being presented to them.

5.3.1 Viewing screenshots

For the first screenshot, the users viewed a picture of the system running, but with no sensor data:

Figure 5.1: CAAR system without sensory input

All participants stated that they did not have enough information to make a decision about which direction they would move in. The second screenshot involved using one sensor to provide data, illustrated below:

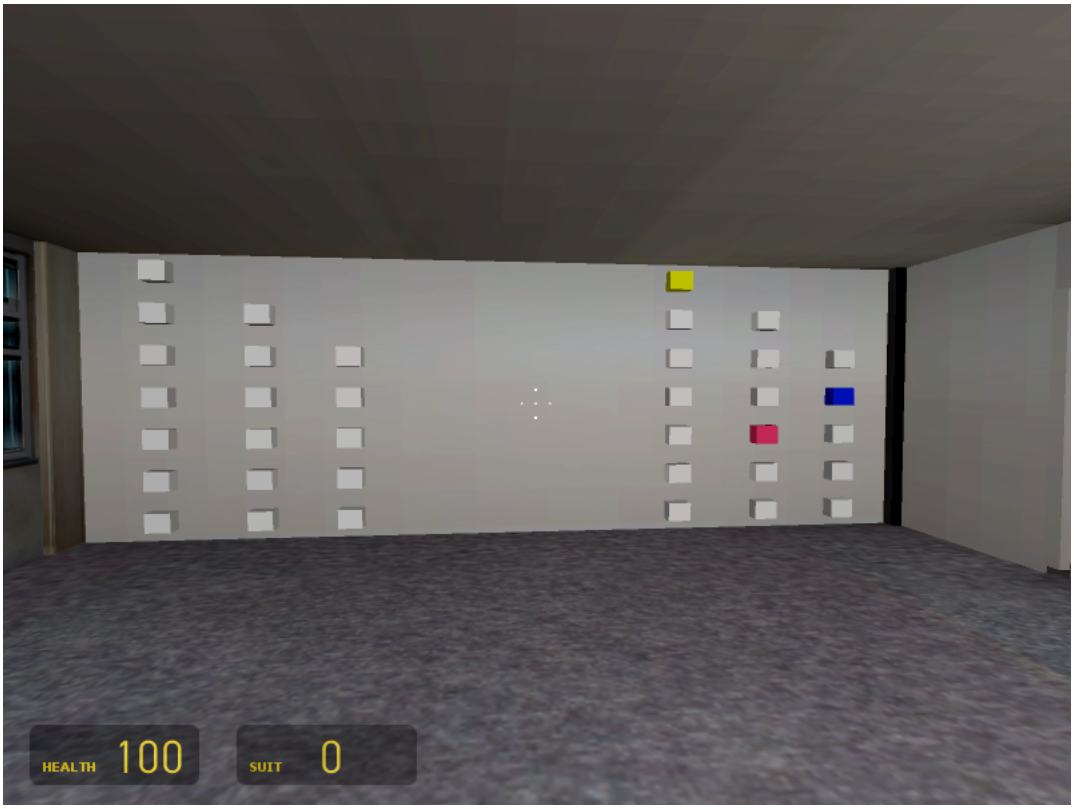

Figure 5.2: CAAR system with one sensory input

Three participants stated they still did not have enough information to make an informed decision, while one stated he would go in the direction of the area being sensed, as he felt it represented "normal room temperature" by pressing the right key. The next screenshot illustrated the system running with both sensors providing input.

Figure 5.3: CAAR system with both sensory inputs

As the right sensor showed a slightly higher temperature gradient than the left, participants unanimously chose to move left or away from the wall of coloured blocks. In the next screenshot, both sensors were providing input, but one sensor was placed over an electric radiator to manipulate its heat sensor readings, as shown below:

Figure 5.4: CAAR system with one sensor artificially heated

All participants unanimously agreed they would move left, as the higher temperature reading was on the right.

5.3.2 Viewing the running system

The next phase of the user study was getting the four participants to view the system running, and ideally trying to reproduce the conditions seen in the screenshots. The system was configured so users would start off facing the wall of coloured blocks containing the sensor readings. The system was viewed with no sensory input initially, and as was the case with the screenshots, participants could not decide on a direction to go, citing a lack of usable information. Then the system was viewed with one sensor activated, but all participants stated they lacked enough information still. Next, both sensors were activated, and the left sensor showed elevated temperature readings relative to the right sensor, so two participants stated they would move to the right, while two participants

opted to move away from the wall towards the back of the room. For the final display, the electric radiator was used again by placing the right sensor over it, and users quickly and unanimously decided to move left.

5.3.3 Results

Shown in Figure 5.5 is a table of compiled results for chosen user responses, and reaction times taken in stating those responses, from all participants, for the four screenshots, and then the four live sensor scenarios. In addition, a chart of the reaction times is provided in Figure 5.6. As some scenarios were not answered by any participants, there is no data to be charted for those scenarios. The X-axis denotes the different scenarios, while the Y-axis displays reaction times in milliseconds. The data shows that for the screenshots, all of the screenshots that elicited a response took one second or more to actually get that response, with the mean response time being 1135.11 milliseconds. For the live scenarios, the response time was lower, with an average of approximately 890 milliseconds.

Scenario	Participant 1	Participant 2	Participant 3	Participant 4
Screenshot 1	N/A	N/A	N/A	N/A
Screenshot 2	N/A	(right) 1202ms	N/A	N/A
Screenshot 3	(left) 1208ms	(down) 1003ms	(left) 2530ms	(down) 1741 ms
Screenshot 4	(left) 1020ms	(left) 1102ms	(left) 1390ms	(left) 980 ms
Live Scenario 1	N/A	N/A	N/A	N/A
Live Scenario 2	N/A	N/A	N/A	N/A
Live Scenario 3	(down) 801ms	(back) 1115ms	(back) 950ms	(down) 888 ms
Live Scenario 4	(left) 600ms	(left) 907ms	(left) 1056ms	(left) 904 ms

Figure 5.5: Compiled results for user study

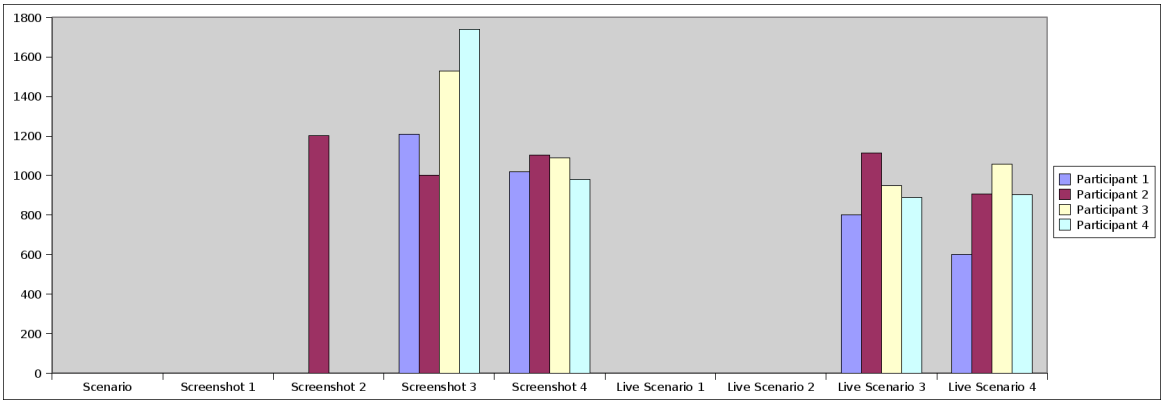

Figure 5.6: Chart of response time results in milliseconds

Two things can possibly be conferred from this data. The screenshot responses may have taken longer to generate a user response, as users were initially unfamiliar with the visualisation format and found it difficult to an extent to comprehend initially. As the experiments proceeded, increasing familiarity may have caused response times to drop, particularly in live scenarios. The second possibility is that the visualisation and presentation of context is inherently easier to understand when it is running, than being displayed on a cluttered desktop with additional distractions. More user studies would be required to acquire a more detailed and accurate picture of how the system performs compared to a desktop-based mode of interaction.

5.4 Summary

In this section, the CAAR system was evaluated along three separate fronts; how the system compares to other augmented reality systems, how the systems finished version compares to the initial statement of objectives in the projects first proposal, and how the system fares in terms of usability, through the undertaking of an informal user study. Additionally, results and a subsequent analysis of the user study was performed.

Chapter 6

Conclusions and Future Work

This chapter presents future work which could expand on and improve the CAAR system implemented as part of this dissertation. In addition, some conclusions which can be drawn from the design, implementation and evaluation of the project are presented.

6.1 Future Work

The CAAR simulator could be further developed upon in the following ways:

- Improved Wireless Sensor Network Implementation
- Support for optical head-mounted displays
- Location awareness of sensors and users
- Support for additional types of context
- Improved visualisation techniques

6.1.1 Improved Wireless Sensor Network Implementation

The wireless sensor network implemented in CAAR is relatively simplistic. The routing algorithm employed does not take into consideration issues such as power management, flow control or handling of duplicate messages. Additionally, the algorithm was only tested using two sensor nodes, so its true scalability is not known at this time.

The WSN could be improved by adding power management capabilities to prolong the useful lifetime of nodes, by reducing the frequency of sensing iterations. Checking for duplicate messages would also reduce the power consumption by requiring less message re-transmissions. Flow control would be necessary as the WSN scaled, as a node could be theoretically swamped with messages from peers which require forwarding, which would consume excessive CPU time and power. 68

6.1.2 Support for optical head-mounted displays

Currently the CAAR system uses a closed-view video head-mounted display which displays a virtual environment annotated with real-world data. Using an optical head-mounted display would be closer to the spirit of augmented realitys original definition, of annotating the real with the virtual. It would also allow the system to be used in any environment which supported embedded sensing or ubiquitous computing devices. However, it is a considerable implementation challenge, requiring a workaround or complete solution to the previously discussed registration problem. Additionally, the augmented reality would need to be location-aware, so it would only display data from sensing devices near the users physical location, for instance.

6.1.3 Location awareness of sensors and users

Location-awareness would markedly benefit the CAAR system, as it would allow the system to be fully aware of a users location within the environment being sensed. Sensor data would automatically be filtered according to a users location, so sensor nodes could be turned off if the user is not in their physical proximity, saving power. It would also benefit the routing algorithm, as it would nodes to route data more directly to the sink, by only using peers who are closer to the sink than they are, rather than taking convoluted routes around the network. Sensors do currently have GPS capability, but the resolution provided is to tens of metres, which is clearly inadequate in an augmented reality context.

6.1.4 Support for additional types of context

CAAR currently supports sensing for light levels, temperature and humidity. This could be expanded to detect other types of context such as ultraviolet and other forms of

radiation, magnetic fields, wireless internet signals, soil PH and so forth, allowing the user to acquire a rich set of context which would facilitate environmental understanding, and have a range of applications in adding visualisation to environmental monitoring.

6.1.5 Improved visualisation techniques

As already stated, the CAAR system uses a closed-view video head-mounted display. This could be developed upon to provide a more intuitive form of visualisation in the virtual environment presented in this dissertation. Rather than using coloured boxes, a 3D thermometer could be used to show humidity readings, while a lightbulb could show light levels and a water droplet would show humidity levels, as examples. These would likely require no a priori instruction, as they would use everyday metaphors which would be easy to understand.

6.2 Conclusions

This project has produced a feasible integration of augmented reality and context aware computing. By far the most challenging part of the dissertation was building a working implementation of a wireless sensor network, due to the almost non existent debugging capabilities available. However, a working WSN was developed which facilitated the systems acquisition of context. The project provides a firm foundation upon which to explore a completely novel way to interact with computing devices, and provide an interaction paradigm much more in tune with our natural faculties. Much room for expansion exists in visualising different types of environmental information, and facilitating a greater understanding of the environment in which the user may find themselves. As sensors become smaller, cheaper and more powerful, along with an improvement in the form factor, capabilities and resolution of head-mounted displays, it is a technology which shows a lot of promise in the new era of computing applications to come.

Bibliography

[1] Squawk homepage. `https://squawk.dev.java.net/`. Retrieved on 28/08/08.

[2] Sunspot homepage. `http://www.sunspotworld.com/`. Retrieved on 09/09/08.

[3] Tinyos homepage. `http://www.tinyos.net/`. Retrieved on 28/08/08.

[4] Zigbee/802.15.4 feature list. `http://www.ieee802.org/15/pub/TG4.html`. Retrieved on 28/08/08.

[5] Micaz 2.4ghz - product sheet. `http://www.xbow.com/Products/productdetails.aspx?sid=164`, 2008. Retrieved on 28/08/08.

[6] R.T. Azuma et al. A Survey of Augmented Reality. *PRESENCE-CAMBRIDGE MASSACHUSETTS-*, 6:355–385, 1997.

[7] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

[8] V. Bellotti and K. Edwards. Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Human-Computer Interaction*, 16(2, 3 & 4):193–212, 2001.

[9] M. Boulkenafed, B. Hughes, R. Meier, G. Biegel, and V. Cahill. Providing Hard Real-Time Guarantees in Context-Aware Applications: Challenges and Requirements. *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, pages 119–127, 2005.

[10] D. Chen and P.K. Varshney. QoS Support in Wireless Sensor Networks: A Survey. *Proc. Intl Conf. Wireless Networks*, pages 227–233.

[11] G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research, 2000.

[12] A.K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[13] S. Feiner, B. MacIntyre, T. Höllerer, and A. Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal and Ubiquitous Computing*, 1(4):208–217, 1997.

[14] A. Fitzpatrick, G. Biegel, S. Clarke, and V. Cahill. Towards a Sentient Object Model. *Workshop on Engineering Context-Aware Object Oriented Systems and Environments (ECOOSE)*.

[15] WEL Grimson, GJ Ettinger, SJ White, T. Lozano-Perez, WM Wells III, and R. Kikinis. An automatic registration method for frameless stereotaxy, imageguided surgery, and enhanced reality visualization. *Medical Imaging, IEEE Transactions on*, 15(2):129–140, 1996.

[16] T. Hollerer, S. Feiner, and J. Pavlik. Situated documentaries: embedding multimedia presentations in the real world. *Wearable Computers, 1999. Digest of Papers. The Third International Symposium on*, pages 79–86, 1999.

[17] A. Jameson. Modelling both the Context and the User. *Personal and Ubiquitous Computing*, 5(1):29–33, 2001.

[18] S. Julier, Y. Baillot, M. Lanzagorta, D. Brown, L. Rosenblum, and NAVAL RESEARCH LAB WASHINGTON DC ADVANCED INFORMATION TECHNOLOGY. *BARS: Battlefield Augmented Reality System.* Defense Technical Information Center, 2001.

[19] U. Kukreja, W.E. Stevenson, and F.E. Ritter. RUI - Recording User Input from Interfaces under Windows and Mac OS X. *Behavior Research Methods, Instruments, & Computers*.

[20] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, 2002.

[21] P. Milgram, H. Takemura, A. Utsumi, and F. Kishino. Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum. *Proceedings of Telemanipulator and Telepresence Technologies*, pages 282–292, 1994.

[22] T.P. Moran and P. Dourish. Introduction to This Special Issue on Context-Aware Computing. *Human-Computer Interaction*, 16(2, 3 & 4):87–95, 2001.

[23] JMF Moura, J. Lu, and M. Kleiner. Intelligent sensor fusion: a graphical model approach. *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*, 6, 2003.

[24] Frank Murphy. MoteLife. `https://www.cs.tcd.ie/publications/tech-reports/reports.08/TCD-CS-2008-17.pdf`, 2008.

[25] E. ONeill. TATUS A Ubiquitous Computing Simulator, 2004.

[26] E. ONeill, M. Klepal, D. Lewis, T. ODonnell, D. OSullivan, and D. Pesch. A Testbed for Evaluating Human Interaction with Ubiquitous Computing Environments. *Proc. of 1st Intl Conf. on Testbeds and Research Infrastructures for the Development of NeTworks and Communities*, pages 60–69, 2005.

[27] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[28] W. Piekarski and B. Thomas. ARQuake: the outdoor augmented reality gaming system. *Communications of the ACM*, 45(1):36–38, 2002.

[29] W. Piekarski and B.H. Thomas. The Tinmith system: demonstrating new techniques for mobile augmented reality modelling. *Proceedings of the Third Australasian conference on User interfaces-Volume 7*, pages 61–70, 2002.

[30] N. Reijers, R. Cunningham, R. Meier, B. Hughes, G. Gaertner, and V. Cahill. Using Group Communication to Support Mobile Augmented Reality Applications. *Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC 2002).*

[31] B. Rhodes, R. Innovations, and CA Menlo Park. Using physical context for just-in-time information retrieval. *Computers, IEEE Transactions on*, 52(8):1011–1014, 2003.

[32] K. Romer and F. Mattern. The design space of wireless sensor networks. *Wireless Communications, IEEE [see also IEEE Personal Communications]*, 11(6):54–61, 2004.

[33] B. Schilit, N. Adams, R. Gold, D. Goldberg, K. Petersen, J. Ellis, and M. Weiser. An Overview of the Parctab Ubiquitous Computing Experiment. *IEEE Personal Communications*, 2(6).

[34] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on*, pages 85–90, 1994.

[35] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavari, L.M. Encarnacao, M. Gervautz, and W. Purgathofer. The Studierstube Augmented Reality Project. *Presence: Teleoperators & Virtual Environments*, 11(1):33–54, 2002.

[36] A. Schmidt, M. Beigl, and H.W. Gellersen. There is more to context than location. *Computers & Graphics*, 23(6):893–901, 1999.

[37] Maggie Shiels. Say goodbye to the computer mouse. `http://news.bbc.co.uk/2/hi/technology/7508842.stm`, 2008.

[38] S.A. Taylor and H. Sharif. Wearable Patient Monitoring Application (ECG) using Wireless Sensor Networks. *Engineering in Medicine and Biology Society, 2006. EMBS'06. 28th Annual International Conference of the IEEE*, pages 5977–5980, 2006.

[39] Crossbow Technology. Mts420 datasheet. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MTS400-420_Datasheet.pdf`. Retrieved on 28/08/08.

[40] J. Vallino. Introduction to Augmented Reality. *Department of Software Engineering, Rochester Institute of Technology. internet: http://www. se. rit. edu/jrv/research/ar/introduction. html.(9-7-2002)*, 14:27–51, 2002.

[41] M. Weiser. The Computer for the 21st Century. *Scientific American*, 9:933–940, 1991.

[42] R.B. Welch. *Perceptual modification: adapting to altered sensory environments*. Academic Press New York, 1978.

[43] L. Yang and J.A. Landay. Rapid prototyping tools for context aware applications. *Proceedings of the CHI*, 2005.