Systems and Services for Wearable Computers

Candidate: Neill James Newman

Thesis Submitted for the Degree of Doctor of Philosophy

Department of Electronic Systems Engineering,

University of Essex, Wivenhoe Park, Colchester CO4 3SQ

1st January 2002

Abstract

The use of both portable computing and mobile communication has increased dramatically in the last few years. Mobile devices combining computing and communications are now being explored and there is competition between manufacturers to provide more features and push the technology.

Integrating an increasing number of features into a small package creates additional problems to those of mobile operation. Contextual considerations such as the location and activity of the user become relevant to the interaction between the human and computer. Therefore a mobile computer should be able to perceive the environment and adjust the presentation of information automatically.

The aims of this thesis are to analyse the capabilities of some mobile interaction devices; to design a user interface system which takes into account these capabilities; and to integrate this user interface with a software framework which enables the machine to perceive the environment and react accordingly.

The thesis starts by specifying and detailing the construction of a mobile platform for the remainder of this work. This wearable computer consists of a small PC with a head-mounted display and a commercial portable keyboard called a Twiddler. A study investigates these interaction devices and contrasts the Twiddler and various head-mounted displays with a standard keyboard and mouse. The results show that it is possible to design a user interface which can increase the speed and accuracy of use of the Twiddler and head-mounted display devices, but generally they perform poorly in comparison to the normal desktop

devices. There are also indications of increased fatigue and user frustration when using these mobile interaction devices.

The observations of the results from the study show that the current desktop user interfaces are not as efficient in a mobile situation as they are in a desktop situation. This has prompted the author to investigate alternative user interface systems for mobile computing. A prototype software architecture called Sulawesi is presented which attempts to address the lack of an alternative user interface research platform. Sulawesi has been designed to encompasses contextual awareness, agent-based systems, and multi-modal user interfaces into a single development framework.

The software architecture allows physical and hybrid sensors and rendering mechanisms to be abstracted from applications, and a management layer allows communication between these subsystems. The user can command the system via constrained natural language statements. Speech recognition or textual input allow the user to command the machine, and a dedicated user interface, tailored to the head-mounted display, or speech rendition are used for output.

Also, a system which allows dedicated agents to process information from a sensor, and to affect the rendition of information to the user, have been incorporated into Sulawesi. A prototype agent which uses contextual information to affect how the information is displayed to the user is also described in this work, along with several novel mobile applications that make use of contextual information.

Acknowledgements

First and foremost, I wish to dedicate this thesis to Tracy for putting up with the keyboard noise late at night, to my father, Sidney Newman, and to my mother, Marilyn Newman for their support, and to my sister, Angie[1] Newman, for keeping my feet on the ground ;)

Many thanks to go to my supervisor, Dr. Adrian Clark for offering me the chance to embark on the Ph.D. for keeping me on track and heading in the right direction, for calming my fears and nurturing my interests.

Thanks to all the people in the VASE lab during my stay, Firstly, thanks to all the people of the past who have made life considerably more fun here at the lab, and some of whom I had the additional pleasure of working with on a project or two.

Thanks to Mike Lincoln for also being a very good friend over the years, for helping with some tricky coding aspects, for being able to bash around ideas with, and for introducing me to curry.

Thanks to Eddie Moxey for keeping me company in the lab, and for generally lowering the tone of conversation whenever possible ;)

Thanks to David Johnston for providing hours of entertainment with his witty and flowery use of the English language.

[1] She would be very annoyed if I called her Angela!

Thanks to Panagiotis Ritsos for pursuing the wearables interest in the lab.

Thanks to John Carson for providing a dry and satirical outlook on life.

Thanks to Graham Sweeney for providing some very raw Glaswegian humour!

Thanks to Andrew Chandler for being a very good friend, and for allowing me to prove my coding abilities in the Sumatra project.

Thanks to all the VASE lab Quake crew for allowing me to frag them.

Thanks to Adrian and the university sysadmins for allowing Mike and I to have complete control of the lab infrastructure, and for putting up with us when things went wrong ;)

Thanks also go to Chris Reeve, Garry Howes, Gary Johnson, Richard Wright, Robert Lee, Sumudu, and all of the Posse crew for being very good friends, and a special thank you to the little cat over the road for keeping me company while writing the thesis, a piece of chicken is on its way.

Lastly, I would like to thank all who know me. Fortunately my visual memory is very good, and I can remember the faces of all my friends, unfortunately my ability to remember names is terrible! I know I have forgotten to say thank you to some people here, so this one is for you.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Introduction

During the late 1970's a few individuals succeeded in adapting electronic systems so they could provide a useful task while on the move[1]. These systems were often crude and provided limited functionality as they were designed for a particular task. Since then several research groups[2] have been involved with adapting personal computers for use in a mobile environment. Due to the reduction in size of the hardware over the years, these wearable computing systems are now becoming more of an interest to a wider audience. Researchers from a growing number of disciplines are experimenting with providing computing power and information access to mobile users, from architects to archaeologists.

Most general personal computing devices in use today have some kind of desktop user interface and, as the vast majority of people are used to the desktop metaphor, it would seem a logical idea to put a desktop user interface onto a wearable computer to reduce the learning curve of the device. While the idea of having a computer present all the time may sound appealing, the use of the desktop metaphor on a wearable device may not be appropriate as it assumes that the user will be using a standard monitor, keyboard and pointing device.

[1] `http://www.eecg.toronto.edu/~mann/index.html`

[2] MIT, Georgia Tech, Carnegie Mellon, IBM, Sony and Boeing to name a few.

Although the metaphor is fairly intuitive on a personal computer, problems arise when trying to use the desktop interface with some of the alternative input and output devices available on a wearable computer.

This thesis investigates the characteristics of some of the current wearable user interface devices and proposes some guidelines for systems based around these devices. Also proposed is a contextual wearable user interface system which has been designed to aid in the implementation of non-traditional applications and user interfaces which encompass context awareness. The architecture is called Sulawesi and attempts to provide an implementation framework to allow researchers to test and develop wearable user interface systems and applications.

1.2 User interfaces in mobile environments

Until recently wearable computers were dedicated systems constructed by and for a single person. The machine was customized to suit the owner's personal preferences using non-standard input/output devices to achieve different interaction techniques. As can be seen in table 1.1, most of the user interfaces employed on these machines have been an amalgamation of existing desktop systems and novel input/output devices.

To date there has been no formal definition of how the interface between the user and the wearable computer should be constructed. Information about the various interaction mechanisms available for a wearable User Interface (UI) system provide an insight into what should *not* be included in the interface (see section 4.10). In comparison to the desktop Graphical User Interface (GUI) there has been little research into the interaction between a person and a computing device which is to be used while on the move. This lack of knowledge can cause problems when trying to design a wearable user interface.

Name	User Interface components
Steve Mann	NTSC screen, text based + Twiddler http://wearables.blu.org/showcase/3.html
Ken Williams	M1 head HMD, text based + Twiddler http://wearables.blu.org/showcase/4.html
Greg Priest-Dorman	M1 HMD, speech synthisis + BAT keyboard http://www.cs.vassar.edu/~priestdo/wearable.html
R. Paul McCarty	M1 HMD, X desktop + keyglove http://wearables.blu.org/showcase/7.html
Timothy Gray	M1 HMD, X desktop + Twiddler http://wearables.blu.org/showcase/8.html
Thad Starner	M1 HMD, text based + Twiddler http://wearables.www.media.mit.edu/projects/wearables/lizzy/
Mika Iltanen	M1 HMD, wrist keyboard, finger mouse http://wearables.blu.org/showcase/10.html
Jeff Hartman	M1 HMD, X desktop + Twiddler http://www.digiman.org/
Wearable EPSS	Virtual Vision HMD, Windows + trackpad/speech input[44]
Hitachi Wearable	Custom HMD,Windows CE,Touch pad http://www.hitachi.co.jp/Prod/vims/wia/eng/main.html
Gerd Kortuem	Virtual-IO HMD, Windows + trackpad/speech Input http://www.hitachi.co.jp/Prod/vims/wia/eng/main.html
Brian Rudy	M1 HMD, Windows + Twiddler http://www.praecogito.com/~brudy/techwear.html
Steve Feiner	Virtual-IO, Windows + trackpad[62]

Table 1.1: Some current wearable user interfaces.

1.3 Personal digital assistants, portable and wearable computers

The differences between a portable computer, a PDA and a wearable computer can be summarized in several ways, here comparisons are made between the types of interaction devices available and the task for which they are to be used.

Portable computers

These are commonly termed laptop devices which run industry standard software such as Windows, with a screen, keyboard and pointing device being used for data entry and object manipulation. While these devices are mobile in the sense that they can be moved to different locations, it is almost impossible to

use them while on the move (e.g. while walking). The user is expected to stop their current task and manipulate the desktop user interface and waiting until the machine has produced the desired results. For a desktop machine where a user solely uses the computer for a given task this may be acceptable, but for a machine in a mobile environment it would be unrealistic to expect the user to halt their primary task (such as driving) to assist the machine.

Personal digital assistants

These are usually smaller than the laptop devices, providing a lower power consumption and lower-than-laptop processing power. The palm pilot for instance is a good example of a PDA with a small physical footprint. It is still fairly usable for its size with a small touch screen and stylus combined with a graphical user interface. It can be argued that a PDA is more useful in mobile environments than a portable desktop machine because it has the ability to communicate information to the user without requiring the users intervention. For example; most PDAs posses the ability to turn themselves on at a predefined time and alert the user of an appointment (usually through a beep or an alarm). It has been observed that these simple alarms can remind users of an event without having to manipulate the device.

These two systems fall either side of a loose definition of a wearable computing system. Steve Mann [65] has stated that *"The wearable computer provides significant processing power, runs continuously, and is always ready to interact with the user. Unlike a hand held device, laptop computer, or PDA, it does not need to be opened up and turned on prior to use"*. The ability to proactive detect the users environment and assist the user is also considered an important feature for a wearable device.

In ubiquitous computing the sensors and processing power are placed within a fixed environment. An ubiquitous computing environment tries to understand

and react as changes occur within the environment. Conversely, a wearable computer places the sensors and processing power on a person, and the environment is not fixed. This makes many of the research problems involved in understanding and reacting to events considerably harder as the environment is not longer under control.

1.4 Wearable computers

With the reduction of processor size and power requirements over the last few years there has been an increase in mobile telecommunications and wireless data access. There are already mobile phones with built in diary functions, and PDA devices with mobile phone adapters, so it is not hard to imagine that eventually these devices will merge into a single personal communications device that will be accessible and operable while on the move.

Observations of the equipment and the environments in which wearable computers can be put to use in soon reveals a very large problem space in relation to a desktop computing device. A wearable computer poses a different set of issues and the possible interaction techniques manifest themselves in many forms.

The current generations of wearable computers have many incarnations. Some have been placed inside shoes [72] while others have been placed inside jackets and coats [63, 35]. As seen in table 1.1 the most popular commercial input device used by wearable users is a single-handed chording keyboard called a Twiddler which has an integrated tilt sensor that can be used to emulate a mouse. The wearable researchers in the past have used the Twiddler because of it's low cost and immediate availability, also a lot of research is focused towards novel applications of wearable computers rather than how to control the machine. A head-mounted display (HMD) provides a portable graphical display. Some HMDs are occlusive and project the display into one or both eyes, while other models provide a see-through display which augments the graphical

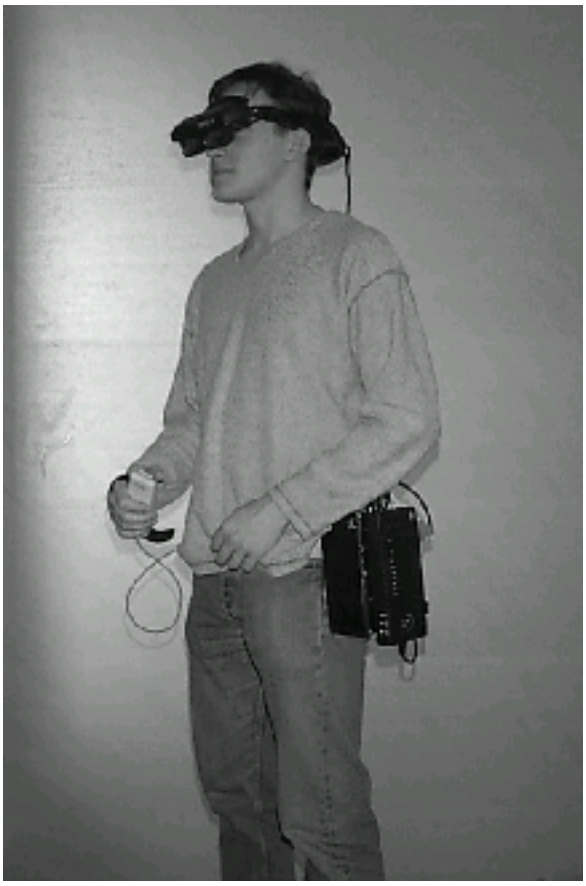


Figure 1.1: The author with his wearable computer.

display with the real world. The author's prototype wearable computer can be seen in figure 1.1 and a description of the construction of the machine is found in chapter 3.

The author's wearable computer uses a head-mounted display and Twiddler to replace the monitor and keyboard of a desktop machine and the initial interaction methods are based around a current desktop user interface. This use of a desktop interface system assisted in the rapid prototyping of the wearable computer, but it becomes obvious after only a few minutes of using the machine that in the long term the user interface is not appropriate. The desktop scenario usually involves a person sitting at a desk with a screen transferring information from the machine to the user. Manipulation of an input device such as a keyboard or a pointing device transfers information from the user to the

machine. The position of these devices depends upon the user: some people are able to type on a keyboard without looking at it, while others place the keyboard directly beneath the screen so they can glance at it to obtain a reference point for the location of the keys. On the other hand, a pointing device such as a mouse does not need to be within the user's peripheral view as current graphical environments provide sufficient feedback via a mouse pointer for the user to locate quickly and accurately the pointer on the screen, and generally there are only two or three buttons on the mouse so the user does not need to locate the position of the buttons visually.

1.4.1 Wearable user interfaces

Wearable user interface can be classified into two groups. Each takes into account the environments in which the wearable will be put to use, the kind of interaction methods that will be available and the tasks the machine will be expected to perform.

Primary task interface

If a person was using a wearable computer to assist with a task, or to pro-actively provide information, then it makes sense to try and integrate the information with their task as seamlessly as possible. This is analogous to a person driving a car within a speed restricted area. The user's primary task is driving while the speedometer is used as an indicator of the car's speed. The user can glance at the speedo and gain some knowledge of the car's speed, they can then focus their attention back to their driving task and adjust the car's speed up or down accordingly.

The author has termed this type of wearable interface a *Primary Task Interface* (PTI), in that it can provide small amounts of information with a minimum of distraction. The information could be very terse and rudimentary but enough to be useful for the user's task.

An example of a wearable PTI would be a system which assists when meeting people by recognizing them from images captured with a small video camera. An overlay of the person's name above their head would allow the user to concentrate on speaking to that person without having to concentrate on the user interface for any length of time.

Secondary task interface

If the wearable computer is not being used in the user's primary task then there may be occasions when the computer will be working in the background to offer potentially relevant information to the user. This type of interface has been called a *Secondary Task Interface (STI)* in that it can provide more verbose information to the user.

An example of a STI would be a tour guide system which offers information to the user. The user's primary task in these circumstances might be to avoid traffic or to admire an ancient building, but if needed the wearable system could be consulted to provide more detailed information on a particular route or building. Here the information would be more verbose and require the user to stop their primary task and focus their attention to the wearable device. It may also possible for the interface to be *primary* in some situations and *secondary* in others.

1.5 Environmental issues

In order to look at advancing the current wearable user interfaces we need to observe the types of interaction mechanisms used and the environments in which they will be used.

1.5.1 Carriage of devices

The carriage of the wearable computer will vary depending on the environment in which the device will be used. Some people have placed the main computer parts inside a small bag slung over the shoulder as this allows the user to don and doff the machine easily. This approach was used by several MIT researchers with the Tin Lizzy[3] machine. Unfortunately, the carriage of the bag means that the machine is under constant vibrations and knocks, and this may cause long-term problems with the hardware. It has also been noted that carrying a delicate machine in a bag is not ideal as people think it is just a bag and take little care when walking past the person carrying the machine.

Some people have placed the main processing part of the wearable computer on the waist via a belt arrangement[4]. This is a more comfortable position than the shoulder bag as it does not pull on the shoulders, but the arrangement means that the machine still suffers from being knocked. Other researchers have placed the machine on the back (either via a rucksack [62], or on a specially designed jacket [28]). This has the advantage of not being knocked easily by other people, but the location makes it difficult to control any switches of buttons which may be on the physical device.

Also, some people [39] have looked into the carriage of wearable computer devices, and have suggested that unless careful mounting of the device is considered, the possibility of back pain and skeletal disorders may arise from incorrect positioning of the devices.

1.5.2 Location of displays

When designing a wearable user interface, one of the main considerations should be the type of display and how it will affect the desired application. At the moment there are three types of display which can be realistically used in a

[3] `http://wearables.www.media.mit.edu/projects/wearables/lizzy/lizzy/index.html`
[4] `http://www.cs.vassar.edu/~priestdo/wearable.pics.html`

mobile environment.

The first type utilizes a normal graphical display which can be worn on the body, such as PDA screens and arm-mounted displays. These displays are good for a *Secondary Task Interface* in that they can work in the background providing information to the user, and the user can consult the device when information is needed. The display would not be very suitable for a *Primary Task Interface* as it requires the user to focus attention on the device rather than on their task.

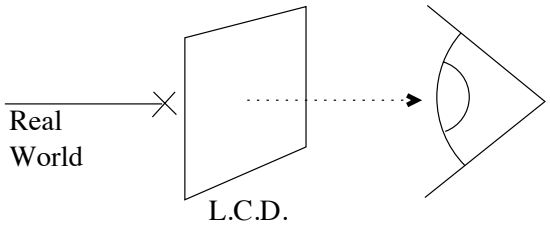


Figure 1.2: Immersive display.

The second type of display utilizes small liquid crystal displays (approximately 1cm×1cm) which are placed in the user's field of view. These displays, such as the M1[5], usually provide a resolution of around 320 × 240 pixels and are used in a monocular configuration. This type of display is often referred to as an immersive display because it is viewed directly and obscures the user's field of view (see figure 1.2). Again this type of display has been classified as a *Secondary Task Interface.* The main reason is, because it obscures the user's field of view, the user cannot easily use the display and focus on a task simultaneously.

The third type of display is similar to the second type (described above), with small liquid crystal displays (LCD) being used. Instead of the physical LCD screens being positioned in front of the users eyes, they are placed outside of the field of view of the user and mirrors are used to bounce the light from the LCD screens into the eyes of the user (see figure 1.3). These type of displays nearly always have half-silvered mirrors placed in the user's field of view: this allows

[5] `http://www.liquidimage.ca/`

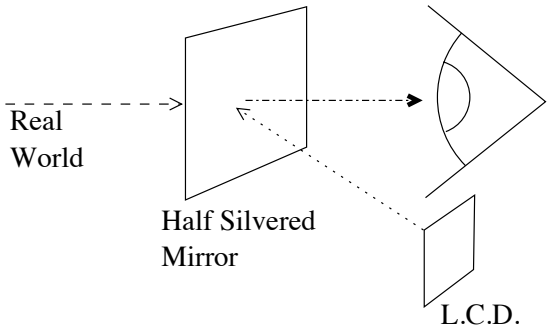


Figure 1.3: Augmented display.

the image from the LCD screens to be "augmented" with the image from the real world. This type of augmentation is used by the MicroOptical corporation [50] and by I-O Display Systems LLC[6] in their Virtual IO glasses. This type of display can be used to implement a *Primary Task Interface* and overlay information onto the real world. If the information is provided at a low enough rate it is possible for the user to proceed with their *primary* task while still receiving information from the machine. This type display system is currently used by fighter pilots around the world: in helmet-mounted head-up displays the computer overlays targeting information upon the real world to assist the pilot. At the same time it is possible for the pilot to fly the plane without having to glance at a computer screen.

1.5.3 Lighting conditions and display visibility

A wearable computer provides a mobile computing platform. This inherently means that the device could be used indoors and outdoors in different environments. One of the main consideration for a graphical user interface on a mobile computing device is *can the user see the user interface clearly enough to understand it?*.

There are certain physical limitations on the display devices available, such as the amount of light they emit, and these limitations cannot easily be changed.

[6] `http://www.i-glasses.com/`

Conversely, a user interface for such a device should be usable in many types of environment. If the machine is to be used outdoors all through the year then the interface should work just as well on a bright summer day as it would on a dark winter day. The same should be true if the machine is to be used during transitional lighting environments such as continuous operation from indoors to outdoors or from working during the afternoon into the night. Similar observations of illumination have been recorded by [16] where a field survey of aircraft engineers was performed. The observations showed that in bright lighting conditions the users would move their head away from the sun and try to cast a shadow over the display.

The desktop user has the ability to move away from the interface when desired. This gives the user a sense of freedom to be able to disassociate themselves completely from the machine when needed. On the other hand a user interface which is present in the user's field of view all of the time can become overpowering even when it is not being used actively.

A person has a relatively low amount of bandwidth available with which to send information from the eye to the brain [2][7]. When focusing on a object a persons eyes do not stay fixed on a single position for long, the eyes tend to wander around an object rather than staring at a fixed position. A *Primary Task Interface* that dominates the user's field of view may provide a low rate of information to the user but, unlike the desktop user interface, the HMD user interface is in permanent view. There may be situations where it would be required to turn off the user interface, perhaps automatically when certain conditions arise.

1.6 Current wearable user interfaces

The standard terminals, such as the console of a Unix or DOS system, allows commands to be entered while in different places. While a text entry system

[7] chapter 2, page 23

does not necessitate a textual output, most of the text only modes of interaction are based around the command line metaphor which produce a textual output. It can be argued that these console based systems can limit the ability to control the machine.

The author's initial experiences with graphical user interfaces provided by desktop environments, such as Windows95 and X windows, suggests that they are difficult to control with the Twiddler device. A simple task such as opening a text editor and entering a paragraph of text was a lot harder than with a desktop machine. The following observations highlight some of these problems:

- It was not easy to control the graphical cursor with the Twiddler's pointing device. In a brief comparison with a mouse, a trackpad, and a trackball the Twiddler was perceived as the worst performer. In a mobile environment a mouse would not be the best device to use as it (obviously) requires a flat surface to operate, so a trackpad or a trackball might be more suitable than the Twiddler or mouse. The Twiddler was chosen as it integrated a textual entry and pointing device into a single package.

- The graphical cursor was difficult to observe while on the move as the user had two focal points to try and concentrate on: the mouse pointer in the near field of view and the real world, which was in a further field of view. This lead to confusion in trying to control the machine and trying to navigate along a corridor.

- Typing sentences or commands with the Twiddler proved difficult as it required a lot of practice to use at any reasonable speed. It was noted that a novice user constantly observes the Twiddler to find which keys they are pressing. Also, due to the low resolution of the head-mounted display, it was difficult to read the text, compounding the problem.

- The graphical user interfaces used on the machine (X and Windows95) meant that the user had to stop what they were doing and focus on com-

manding the machine. In some situations there was so much information being presented to the user they had to stop even the simplest tasks (such as speaking!) in order to concentrate on the machine. The GUIs were almost impossible to use while moving about due to the use of point and click operations. Vibrations caused the pointing device in the Twiddler to stray away from where the user was trying to position the cursor. Also the selection boxes and radio buttons in most GUI dialogs were difficult to select.

- When using the augmented head-mounted display, the variations in lighting conditions made it very difficult to see the interface, and sometimes the real world! The immersive head-mounted display was not as badly affected by light variations. With both displays it was possible to improve the legibility of the interface by changing the colour schemes to include high contrasting colours such as black on white.

These annoyances were at first attributed to the faults of the interaction devices being used. Further observations revealed that, although in some situations the devices could never perform as well as a desktop device, they were not the main cause. The metaphor that underlies the desktop graphical user interface does not fit in with the concept of a wearable computer, but which component of a desktop interface is at fault needs to be explored before a remedy can be proposed.

1.7 Alternative user interfaces for wearable computers

What would be the ideal user interface for a wearable computer? Due to the sparse number of wearable user interface systems we need to look further afield than the wearable computing arena to other areas such as multimodal interfaces,

intelligent agent interfaces and contextual awareness. These fields provide an insight to a potential wearable user interface, an example of which would be a system which listens for its user, understands what the user has asked it to do using speech recognition, gestures, machine vision and other channels of information, carries out the users request automatically, and presents the results back to the user when it is most appropriate and in a suitable format. The machine could also observe the user's environment and suggest relevant information when it is convenient to do so.

Investigations into multi-modal systems 2.2 which gather information from multiple channels have been around since the early eighties [59]. Since then several multi-modal systems have been constructed [19], but examination of these systems reveals that they are specific applications which provide multi-modal interaction techniques. Although they are good at achieving their desired task, trying to use the interaction techniques for anything other than the applications they were written for may be difficult and in some cases impossible. Another difficulty is that development tools for creating and researching into alternative user interface systems are scarce. Kortuem et al. writes [27]:

> "Sophisticated user-interface management systems are available to facilitate the development of window-based direct manipulation user interfaces. Yet, there is almost no support for building non-traditional user interfaces that make use of voice technology and body tracking or a combination thereof. In addition, the limited experience in designing user interfaces for hands-free or one-handed operations poses a serious problem."

Also, the type of information that should be provided, and how it is presented should depend on the users circumstances. The use of sensors to gather information and some higher level autonomous processing of that information to adjust or manipulate existing data has been termed contextual awareness [56] by many.

Brown etal[8] have also defined six types of generic context-aware applications, but they emphasize that:

> "it is not our aim to produce a taxonomy: there are plenty of applications that do not fit any of the generic applications we discuss"

The author believes that the use of proactive autonomous applications that observe the context of their surroundings and present timely information in a suitable format will be very important in a mobile environment.

Because of the diverse nature of tasks that wearable computers will be used for, it is almost impossible to design a single application to suit all needs in all environments. The aim of the work in this thesis is to provide a system that is flexible enough to encompass a wide range of interaction techniques and paradigms, that can be adapted through well-defined programming interfaces, and can be tailored for a specific purpose. An explanation as to what some of the terms mean and how the author attempts to achieve them are below.

A multimodal, multimedia system

As defined in the literature, this is the ability to control a machine using various input modalities, and the machine being able to generate outputs via different rendering interfaces. The utilization of speech control may be very desirable in some circumstances, but in other situations there needs to be alternative ways of controlling the user interface. Providing an extendible system allows others to develop input and output mechanisms tailored to their circumstances.

An agent system

Ideally a wearable system should pro-actively assist its user, so it seems obvious to allow agents to perform the raw processing and automation of

[8] Context-awareness: some compelling applications
http://www.dcs.ex.ac.uk/~pjbrown/papers/acm.html

tasks. The system allows agents (in this work they are interchangeably called *services* or *applications*) to be written by third parties. The prerequisite of this is a well-defined software interface to allow applications to be integrated automatically into the system. The applications are not to be functionally limited in any way; and with the use of a small Java wrapper (see chapter 6 for an example) it is possible to interface with native code easily.

A proactive system

The system should be able to monitor the environment of its user and make decisions based on how the system perceives the environment. The work provides mechanisms to allow agents in the system to view the environment through the input channels. This means that an agent can act on the user's behalf when certain combinations of conditions arise.

1.8 Research aims

1.8.1 User interface issues

In comparison to the desktop user interface, there has been very little research which attempts to address the interaction problem between a person and a computing device which is to be used at the same time as being mobile. Smith et al [11], Bass et al [43] and Baber et al [15] provide some information on various aspects of wearable-human interaction, but there are many commercial wearable interaction devices for which there is no information available.

While information exists about the performance of virtual keyboards, forearm keyboards and chordic keypads [12], currently there is no information on the performance of the Twiddler or various head-mounted displays that are being deployed in wearable computing systems. In order to rectify this lack of information the following needs to be addressed:

- Determine the operational parameters of the Twiddler, the M1 and the Virtual-IO head-mounted displays through user trials.
- A set of guidelines produced from this information for the construction of specialist graphical user interfaces.

It is hoped that this information will be of use to designers of wearable user interface systems based on the Twiddler and head-mounted displays.

1.8.2 Sulawesi

It has been speculated that the ideal human-computer interface for use in a mobile/ubiquitous environment would be one which listens for the user; understands what the user has asked it to do; carries out the users request automatically; and presents the results to the user when it is most appropriate and in a suitable format.

For the author the most interesting question is: *What type of software framework is needed to integrate contextual information with a mobile user interface?* In order to investigate this type of user interface, an architecture called Sulawesi is proposed. The experimental work uses the author's wearable computer[9] as the target platform.

1.9 Outline of the remainder of this thesis

Chapter 2 provides a critique of existing material, focusing on human-computer interaction with wearable computing, multimodal systems, intelligent user interfaces, contextually-aware systems. Also presented is an overview of current wearable user interface systems with the pros and cons of each system and a summary of the features being highlighted.

Chapter 3 describes the construction of the wearable computer used for this research.

[9] Described in chapter 3

Chapter 4 presents the design, implementation and results from a user test to determine the limits of some of the current popular wearable interaction devices. The results from this work are then used to propose a set of guidelines for designing a user interface system based around these devices.

Chapter 5 presents the design and implementation of the Sulawesi system. Consideration is given to the design of proactive agents, the combination of multiple sensor data, and the transformation of sensor data into higher-level abstractions.

Chapter 6 describes the creation of context-aware applications within the Sulawesi framework using multiple sources of sensor information. A set of implemented examples is described which aim to justify the Sulawesi design. Also described is the implementation of a wearable graphical user interface based around the guidelines in chapter 4. Consideration is made within the GUI to allow the intelligent agents to control the presentation of information to the user.

Finally, chapter 7 presents a summary of this research and indicates areas of possible future work.

Chapter 2

A Review of Existing Work in the Area

2.1 Introduction

At the moment there are only a few people who use wearable computers and their individual preferences for the user interface varies. This discussion aims to identify key areas of published work which may be relevant in providing user interfaces for alternative computing systems. This review highlights the point that there are indeed types of user interfaces which may be useful for wearable computers, but there is little information describing how these interfaces are manifested or the interaction issues involved.

Here the author presents a critique of key material in the areas of Multimodal Systems, Wearable Computing Systems, Intelligent User Interfaces and Contextually Aware Systems. The last part of this chapter looks at what user interfaces are currently being used by wearable researchers and the user interfaces being deployed by wearable manufacturers.

2.2 Multimodal architectures

Investigations into computing systems which monitor and react to data from multiple streams have been around for almost as long as the windows, icons, menus, pointer (WIMP) paradigm.

In 1979 the first well-known piece of work which simultaneously gathered data from various streams, inferred a meaning, and produced a response was constructed by Bolt [59]. By using a magnetic tracker strapped to the hand, a speech recognition package and a large 3D projection room, it was possible to gesture that an object should be moved by pointing at the object and moving the hand from the object to an empty space while uttering "put that there".

The benefits of a system such as this in an environment where the use of input devices are restricted, such as a wearable computer, are immediately obvious. It is argued by Van Dam [6] that this type of user interface paradigm, using tactile interfaces, natural language understanding, speech recognition and 3D visualisation techniques will be combined in the future and he has called this amalgamation of alternative interaction mechanisms a "post-wimp" interface.

While there have been several multimodal systems developed over the last few years, each provided a different functionality based around a similar design methodology. Hartung et al. [40] describe a system called HARP which was designed for the acquisition, integration and representation of multimodal information. The architecture highlights this design methodology by dividing the system into four distinct groups which process information at different levels. The *input mapping* stage receives signals from sensors and maps the information into a perceptual space. The *cognitive processing* stage observes this perceptual space and attempts to resolve symbolic references. A *symbolic database* is used to resolve the symbolic references in the perceptual space. Once the system has identified which perceptual space is being used, the *output mapping* stage can then produce a suitable high-level output in response to results from the cog-

nitive processing stage. The system relies on being able to gather information from an environment and being able to understand this information, to be able to process the information and produce a correct response. These concepts and ideas have had a strong influence on the design of the Sulawesi system which forms the core contribution of this thesis.

Other multimodal architectures encountered implicitly use this model as well. The architecture by Wilson et al. [51] uses several expert systems which gather various pieces of information from the environment. A central controller decides which expert systems need to communicate with each other in order to resolve potential ambiguities. Once this has been achieved the controller then sends the result to a final expert system which displays the information.

Similarly, Moran et al. [20] use the Open Agent Architecture with dedicated agents to gather information from speech and handwriting subsystems. A *modality coordination agent* is responsible for combining the information from these different subsystems and to resolve ambiguous references. From this a single understanding can be determined and a *facilitating agent* is used to delegate requests to other agents.

In all of these systems there may be many different expert systems or agents working simultaneously. The fundamental concept here is that the various parts of each system can be categorised into one of the *data gatherer, data processor* or *information provider* groups. This approach requires significant knowledge in various domains, such as speech and handwriting recognition, and while the techniques are well documented there are still problems in trying to resolve ambiguities and conflicting information. These problems can be seen when a handwriting system has to decide whether the letter *I* is an *i*, an *l* or a *1*.

Also, the systems described above are all stand-alone prototypes in that they are designed for a single purpose. In order to expand, adapt or reuse these

systems to perform a different task would be no minor feat, and in some cases it would be easier to write the new application from scratch. In some cases it may not be feasible to provide a single architecture for all purposes, but the author contends that a simple modular architecture, where modules could be added and tailored for particular applications, could provide enough functionality for most of the multimodal systems in the field today.

2.3 Multimodal human-computer interaction issues

Some of the multimodal systems encountered by the author try to make the interface between the machine and the human more natural by using speech recognition and some form of natural language processing. But how should this natural language be presented by the user to the machine? And how should the machine respond if it does not understand?

Some of the problems which arise in multimodal communications, such as uncertainties, ambiguity and feasibility are discussed by McGee et al. [22], and McGlashan [66] describes some of the modal management techniques developed in a speech-only dialogue system. From this work it is clear that two methods of confirmation for reducing uncertainty and ambiguity have been employed. The first is an *early confirmation* technique which attempts to recognise each modality stream when it is received. When a high scoring match is interpreted a response is generated so the system can provide immediate feedback. The second type of feedback is *late confirmation* which takes an overview of all modal inputs and attempts to infer what the user has asked for; only when a high scoring match has been obtained is feedback generated.

While neither of these methods alone can cope with all situations, the combination of these two techniques can produce a hybrid feedback system which

can reduce the deficiency in either of the systems on their own. This idea is especially important in a multimodal system where several modes of information will need to be combined to resolve ambiguities.

2.4 Multimodal applications

Nearly all of the multimodal applications in the literature use some form of agents or expert systems to process information. Faure and Julia [17] investigate the use of agents to analyse multiple modes of interaction. The system constructed utilises speech recognition, pen input and keyboard interaction to manipulate a graphical application with dedicated areas in which pen and speech can manipulate sketches and drawings. To process multiple channels of information and respond accordingly, two different agent types were defined: those which reacted to their environment ("reactive agents"), and those which viewed their environment and interpreted a higher level of understanding about it. The prototype system used only reactive agents to respond to events in the user interface. The use of agents which interpret a higher level of understanding were not used due to the lack of processing power on the target hand-held devices and the latency involved in providing feedback to the user interface. This may be a bad sign when trying to develop a multimodal system for mobile devices. The author believes that the complexity involved with agents which view their environment and interpret a higher level of understanding will determine the processing overheads of the mobile device. Also the concept of using two classes of agents, one reactive and the other making decisions, is invaluable in designing and building a system which processes multiple streams of information.

Over the last few years, much multimodal work has been in the hand-held devices area, specifically in software for manipulating maps. Oviatt [68] describes an interface for dynamic interactive maps: it combined speech and a pen in-

terface for the navigation and manipulation of a map on a Personal Digital Assistant. The results from user trials showed that the vast majority preferred to interact with the map using both speech and pen modalities, with only a few preferring to use the pen-only interface. Interestingly enough there were no users who preferred to use the speech-only interface. Similarly Cheyer and Julia [3] describe their multimodal map work using handwriting, gestures and speech recognition. This system is slightly different in that it was developed using the Open Agent Architecture which had the advantage of being able to communicate with commercial applications such as mail systems, calendar programs and databases. The down side of using the Open Agent Architecture was the complexity of the software: the processing requirements for the Personal Digital Assistant device were increased drastically.

Another area where multimodal interfaces are being actively developed is the area of Virtual Reality. Commenting on the difficulties with controlling a virtual environment with a WIMP interface, Wyard and Churcher [57] describe a prototype multimodal system called MUESLI. The system is able to control the look and feel of a virtual room by the texture of surfaces such as curtains, walls and carpets. This is achieved through the combination of speech recognition, natural language processing and gestures. The benefit of this system is that a relatively inexperienced user can control an advanced computing environment with very little training, but this only works for certain situations where the problem can be easily defined and manipulated.

A similar system, described by Roy and Pentland [19], attempts to resolve ambiguous references through an adaptive learning algorithm. The system uses real-time gestures with two colour video cameras and speech recognition. The interface is manifested on a large back projected screen with a virtual graphical toucan called Toco. Placed around the toucan are coloured objects and when the user moves their hand and points to an object, the toucan watches the se-

lected item. The user can teach the interface by uttering a phrase like "this is a red cube" and can query the interface by uttering a phrase similar to "what colour is this". This system provides the user with both the forms of feedback defined by McGee et al.[22] earlier in this chapter: early confirmation is realised by the toucan moving its head to watch selected objects, and late confirmation is provided by resolving what the user has asked for and which object is selected.

These combined methods of confirmation work well together. The author has personally manipulated this system and concludes that the use of both early and late confirmation in a multimodal interface results in a perceivable improvement in resolving ambiguities and reducing the amount of confusion experienced by the user.

2.5 Wearable systems

The person who popularized the idea of wearable computers, Steve Mann, defines [64] a wearable computer as

> "a new form of human-computer interaction comprising a small body-worn computer that is always on and always ready and accessible. In this regard, the new computational framework differs from that of hand held devices, laptop computers and personal digital assistants. The "always ready" capability leads to a new form of synergy between human and computer".

Mann also discusses the operational modes of a wearable computer and predicts that a fundamental paradigm shift will occur when these systems provide capabilities which

> "empower the user with useful information".

Other early wearable systems are described by Smailagic and Siewiorek [5] where three generations of wearable computing hardware are detailed. The system comprises of head-mounted displays, speech recognition, global positioning systems, and the relative performance of each system is analysed. It is concluded that the newer systems provided a lighter, more power efficient system that were able to run for longer periods of time.

While the hardware requirements for this type of system are a well-understood mechanical and electronic engineering problem, the way in which a user controls the machine and the software architectures which support this type of seamless mobile interaction are less well-understood.

Kortuem [27] provides a detailed view of the software engineering involved with wearable computers. The most challenging issues such as limited resources; limited mobile infrastructure; lack of support for developing non-traditional interfaces; and the integration of the complex systems are discussed with the conclusion that there is a serious problem with the lack of understanding in these areas.

In order to extend the knowledge base Kortuem also provides a detailed description of software engineering. These findings are then applied to a wearable software architecture which attempts to address some of the issues stated beforehand. While there is no evidence of the architecture actually being implemented, the details of the system seem to address the issues of resource management and integration of complex systems. Whether this system would address the problems of developing non-traditional interfaces is not known, but the author believes it contributes an architecture which could be used as the foundation for a wearable user interface system.

An architecture designed by Fickas et al. [29] called Proem seems to address the problem of limited mobile infrastructure initially defined by Kortuem. The

system identifies possible collaboration of wearable users in a changing environment. Once a collaboration has been identified a session is set up between the users and collaboration can begin. If the session becomes disconnected, as can be quite common with mobile communications, the system provides mechanisms for handling this disconnection gracefully, and also transparently re-establishes the collaborative session when possible. In order to provide an expandable system the architecture abstracts the physical devices from the applications; this also contributes to the goal of handling disconnected sessions. The authors of this work comment that Proem only provides a limited subset of a fully-realised collaborative system; despite this, the work provides an insight into the complexities involved in designing a system to cope with limited infrastructure and disconnected sessions.

2.6 Wearable user interfaces

The user interfaces that are in common use on today's desktop computers, PDAs, virtual reality systems and voice-controlled systems make the computer the centre of attention. In his article, Weiser [48] argues that

> "the world is not a desktop"

He proposes that future computing systems such as wearables and ubiquitous environments should be focused toward a more invisible user interface, where the computer gathers information about the environment and responds accordingly. The use of a single modality is condemned in [48], but the possibility of using mixed modes of interaction depending upon the situation appears to be consistent with Weiser's ideas. While it is impossible to conform completely with the idea of making a computer interface invisible, the notion of making a user interface that is less obtrusive and more proactive that the current desktop metaphor is certainly feasible, and is the aim of this work.

On a similar note, Rhodes [14] and Lanuday and Kaufmann [38] both argue that the WIMP (windows, icons, menus, pointers) interface is unsuitable for use on a mobile computer as it makes some fundamental assumptions about the user's situation. The WIMP metaphor assumes that the user has fine motor control with which to position a cursor, the amount of screen real estate is fairly large and that the user's primary task is commanding the machine. As with [48] above, [14] and [38] conclude that the user interface should provide a progressive amount of interaction depending on the user's circumstances and the task they are trying to perform. The ideal system would be able to cope with all eventualities, but in practice it is very difficult to predict all possible outcomes. Also, the author has been unable to find any existing wearable computing system which demonstrates even the simplest implementation of this concept.

In defining a set of elements which are fundamental to wearable computing user interfaces, Segall and Curry [75] stress the importance of a hands-free interface system which is usable while being mobile. Also, the idea that the computer can perceive the user's environment and use this information to adapt to the user's requirements is highlighted. While Segall and Curry only provide a brief outline of the architecture involved to accomplish these concepts, their work provides evidence that people in field of wearable computers are not satisfied with the desktop environment and are searching for a new type of user interface for their systems.

This statement is reinforced by Heiber et al. [46] who comment that "the desktop concept has proven very successful for desktop computers, but the metaphor seems inappropriate for wearable computers with limited screen space and restricted input devices". An alternative user interface system for a wearable interactive mapping application is discussed where the desktop work space is replaced by an application manager. This allows the user to view one application at a time and toggle between several running applications easily. This

prototype is similar to PDA user interfaces, such as the palm pilot, but it still assumes that the only input/output mechanism available is the graphical device.

2.7 Wearable HCI issues

The author believes that the key to the widespread adoption of wearable computers is by enabling seamless interaction with the machine through various input and output mechanisms. The most obvious input mechanisms use the hands to control some kind of input device, or the voice to control a speech driven interface. There are some other mechanisms such as retina-based tracking and foot/leg control systems, but the author considers these to be impractical for a widespread ubiquitous wearable control device.

2.7.1 Hand control

Calhoun et al. [32] have provided an overview of several hands-free alternatives that were considered as candidate input devices for a wearable system. Speech, eye tracking, gesture recognition, electromyographic (EMG) and electroencephalographic (EEG) systems are discussed in great detail. The authors of this work comment that the amount of control achievable with these devices is rudimentary, and these devices are difficult to use with current interface systems because most existing dialogues are tailored for a desktop GUI. It is suggested by the authors that that some form of multimodal user interface system which uses several of the devices described above would provide a more natural user interface system for a wearable computer.

While some of the devices highlighted in this work might be satisfactory for wearable deployment, the use of EEG based systems is unlikely to be appropriate due to the amount of sensors that would have to be placed on the body. Also, because EEG sensors detect small electrical signals in the brain they are very sensitive to electrical interference and therefore are not well suited for use

where electrical noise might interfere.

The use of EMG sensors is explored by Goldstein et al. [47] where they describe an alternative wearable input device. The system works by picking up the muscle movements of the fingers using sensors placed on the forearm. A keyboard is printed on a sheet of paper and placed on a flat surface and the user presses the characters on the sheet. The EMG sensors detect the signals and the system can detect which key was pressed by the combination of the muscle movements used in order to press that key. Their findings show that the speed achievable with this system is only 16% less than that of the QWERTY keyboard. The reason for the reduction is not explained by [47], but the author speculates that the decrease in speed is attributed to the lack of feedback available to the user. The authors note that the system does not address the question of how information should be fed back to the user when they have pressed a key. They also comment about the amount of effort required to use this system. While their measurements showed a continuous trend toward lower muscular strain, the subjects in the test perceived the task as more strenuous.

It is speculated that the alternative interface system which provided no feedback was significantly different to typing on a normal QWERTY keyboard, therefore the learning curve associated with the task came into play and increased the mental load on the subject. Even so, the author feels that as there were no sensor devices around the hand area, this type of system could be useful in a mobile environment as long as a flat surface was available. The system may be usable by typing in the air, but the author speculates that the sensors detect the difference in muscle movements when the fingers hit a flat surface, and typing in the air might not produce the desired resistance to produce these signals.

A similar system to that of Goldstein was devised by Fukumoto and Yoshi-

nobu [26], but instead of using EMG sensors to detect muscle movements, wireless accelerometers were placed inside rings on each finger. Again the user typed on a flat surface and the data from the accelerometers were used to determine which key was pressed. A chording arrangement was used to enter sequences of codes. This makes direct comparison with a QWERTY keyboard almost impossible, but the authors conclusions show that an untrained user could remember and type approximately 27 sequences per hand, at a speed of approximately 130 codes per minute.

The only downfall the author can see with this system is that the physical sensors need to be placed on each hand and, while the rings are very small, wearing one on each hand may be uncomfortable and in some cases impractical.

2.7.2 Speech control

In theory the use of speech in a mobile application is very attractive as it allows relatively intuitive hands free interaction if the interface is well designed. Unfortunately there are problems in deploying a speech-controlled application in an environment with high ambient and variable noise levels. The noise introduces errors into the speech recognition system and, while most of the noise can be filtered out with directional microphones and software filters, the ability of these systems to cope with changing noise in the environment is still being explored.

Guidelines for using speech control in a wearable application have been provided by Najjar et al. [45]. The authors comment that "speech recognition technology presents new challenges to wearable computer user interface designers, especially in the areas of recognition accuracy and ease of use". The guidelines include using speech when the user's hands or eyes are not available for controlling a user interface, such as when the user is moving or focusing on a particular task.

In order to increase the recognition accuracy and reduce the end user's workload an emphasis is placed on using a short vocabulary and commands that

are fairly distinctive from each other. While the contribution provides a good overview of how to enable an application to use spoken commands, there is very little detail as to how a speech enabled application should be constructed, how well it could be expected to perform, or how feedback should be presented back to the user.

Some of these problems have been addressed by Roy et al. [61] in their survey of wearable audio computing interaction techniques. The work describes how a prototype conversational interface called *nomadic radio* was constructed. The design is based around a software agent architecture which can be completely controlled via a speech recognition module. This is different from commercial speech recognition packages where an existing application would be controlled via a spoken dialogue. The agents in the system are specifically designed for a speech-only interface and provide a basic form of feedback to the user's spoken commands. The system also attempts to determine the user's context by listening to the environment through the microphone. If the system decides that the user is talking then a short beep is generated rather than the results of a request being spoken. When the user is ready for the information they can ask for the message to be played back.

While a speech-only interface may be desirable in some circumstances there will be environments and situations where a speech interface is not appropriate. In these cases a system needs to be controllable via a number of alternative methods and, while the *nomadic radio* system provides a good example of how an audio user interface could be designed, the architecture does not provide an easy method for adapting or controlling the agents via alternative input devices.

There is also no evidence of how well the nomadic radio system works in terms of performance such as how often commands are mis-interpreted, so trying to compare this system with others may be difficult.

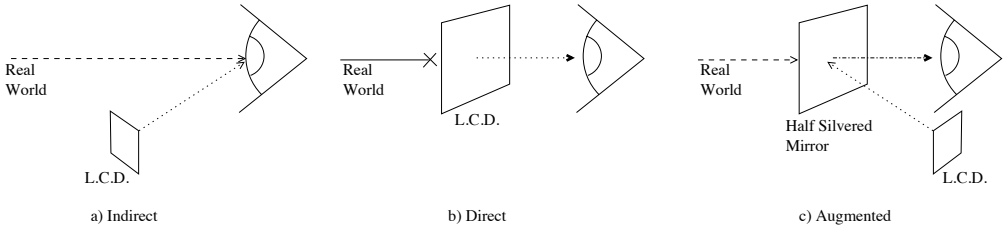


Figure 2.1: Indirect, direct and augmented head-mounted displays.

2.7.3 Head-mounted displays

Most of the wearable computers in existence today use some form of head-mounted display (HMD) system to project images into the eye. Some HMD devices use an indirect approach by placing the LCD out of the visual field of view, while others use a direct approach by placing the liquid crystal projection unit in front of the eye. A third type of HMD uses half-silvered mirrors to mix the signals from the display unit with the real world, resulting in an augmentation of real world and computer generated images (see figure 2.1).

While these HMD units have been around since the mid-1960s [34], it has only been in the last decade that the use of a portable HMD has been achievable at a reasonable cost. Because of this, the use of these displays has been concentrated to proof-of-concept prototypes in research labs, so little is known about what effects the displays would have with everyday use in the real world, or how environmental conditions will affect the performance of the HMD.

One of the few detailed studies on HMD interaction with a mobile environment is provided by Revels et al. [9]. Their glasses mounted display (GMD) system was developed for use by aircraft engineers and the focus of the work is on varying lighting conditions, the performance in respect to desktop computing accuracy and the effects of obscured vision with continuous wear. The results show that the position of the sun affected the direction the subjects faced when reading the display. Most subjects would stand between 90 and 180 degrees from the sun, casting a shadow over the display to try and increase the contrast

of the device.

In respect to a desktop user interface, the subjects found it very difficult to locate icons and pointers on the screen due to the lack of contrast. Also, the amount of control achievable with the pointing device was considered to be too crude for positioning the cursor, but the author could not determine whether this was due to the pointing device (a thumb tracker ball) or to the display device. The methods used in this work and the results from these tests are interesting. The other studies involving HMD devices are biased toward virtual reality applications where the user is in a controlled environment. While the tests provide an in-depth view of how sunlight and other environmental issues affect HMD, there is no information about how effectively the user was able to control the user interface using these devices.

Two studies were undertaken by Baber et al. [15] which investigate some of the human factors of wearable computers. Their first experiment compared a monocular HMD with a normal cathode ray tube (CRT) screen. The task was to locate and select a target on the display device. The results showed that a significant difference in reaction times was observed, with the HMD user exhibiting speeds that were 1.5 times slower in comparison to the user of the CRT screen. The authors note that there was no significant difference in the numbers of errors observed during these tests. They also comment that the difference in reaction times might be due to the HMD being perceived as uncomfortable.

Also of interest is the observation of *binocular rivalry* where information from one eye competes for attention with information with the other. The authors of [15] speculate that this may have also attributed to the slower reaction times for the HMD.

The second study compared the time taken for a user to perform a task using a wearable-based manual compared to a paper-based manual. The results show

that while the wearable computer produced faster task completion times, there was a noticeable trend for more errors to be made with this device.

The work suggests that although there may be potential benefits to be had from a wearable computing device, there is a significant difference in efficiency between using a normal computer screen and an HMD with a graphical target of a fixed size. What is not investigated in this or any other relevant work is whether a variation in the size or shape of the target has any effect on the reaction times of the user. This may provide valuable information as to how the elements in a graphical user interface could be designed to maximise the efficiency of the displays.

2.8 Intelligent user interfaces

The field of intelligent user interfaces covers computing systems which can vary their structure, functionality and purpose so that they appear intelligent to the user of the system. By using predefined ideas about how the interaction between a human and a computer should be constructed, it is possible to create a user interface system which can adapt to the users requirements and aid in the usability of the system.

A discussion of how a computing system can accommodate individual differences by using an adaptive user interface is provided by Benyon [23]. The work discusses some of the important issues about how a person's cognitive ability, personality and motor skills contribute to differences in the interaction methods employed when using a computer. A simple database retrieval task highlighted that people use a program in different ways. By including a piece of monitoring code in the database program it was possible for the system to monitor the user, decide which category the user fell into and adapt the user interface to match the category. The conclusions from this work showed that there was a

noticeable increase the efficiency of the task.

While the work only provides an overview of what is possible with an intelligent user interface, there are some important contributions such as how to monitor a complex operation by splitting it into a subset of smaller operations. These smaller operations may be easier to understand and an insight as to how to adapt the user interface may be easier to find.

A system which adapts to the user is explored by Moran et al. [20], where cooperative agents are used to make decisions about how the interface should be changed to suit the user. In this work a mapping application gathers information from multiple sources (pen,speech) and adapts the user interface to provide the right information in the most appropriate format at a convenient time. The system provides access to agents running on other machines, so speech recognition and natural language can be processed on larger workstations and communicate with the user interface on a PC or a PDA. While there are no quantitative results from this work the authors suggest that the development of the architecture allows the system to be split into several smaller domains with each domain being processed independently, on separate workstations if necessary.

A similar *anticipatory* architecture is described by Davidsson et al. [53] in which an entity has access to various sensors which communicate information to the internal processing modules. Each module would process this information, produce a result and communicate this with the other modules in the system. Once a module has received sufficient information to complete a desired goal it then produces an output through an effector, resulting in the user interface being modified. This process allowed the system to encapsulate some basic learning abilities and predictions about what is likely to happen next in the interface.

An intelligent interface application which has undergone significant testing

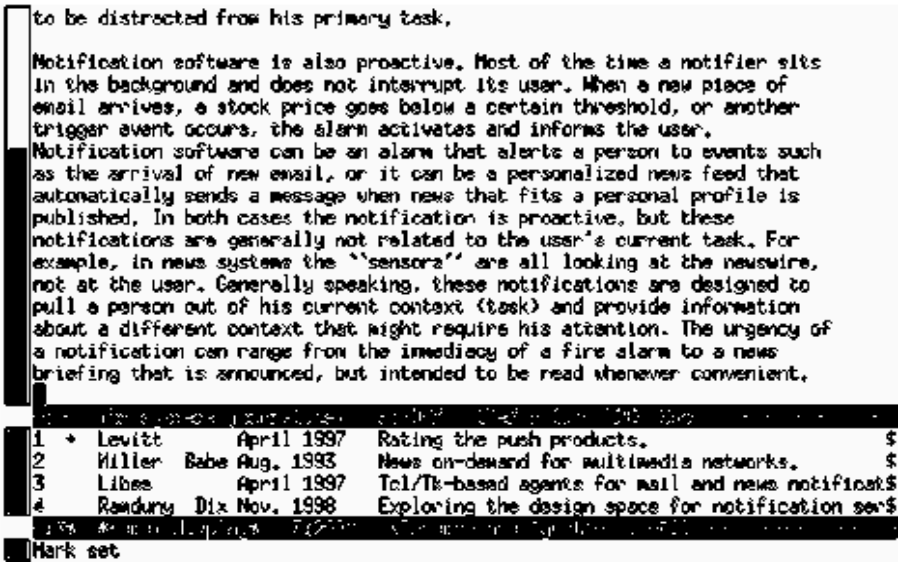


Figure 2.2: The Remembrance Agent.
(from `http://www.research.ibm.com/journal/sj/393/part2/rhodes.html`)

and development is the Remembrance Agent by Bradley Rhodes [13]. The system allows the machine to volunteer potentially relevant information by observing what the user is typing. The system keeps an index of the user's email, documents and other text files, and the index contains statistics about the frequency of words in the files. The Remembrance Agent integrates with Emacs or Microsoft Word and monitors what the user is reading or writing in the editor. The information in the editor is analysed and an index of word frequencies is calculated. This is then compared with the main index and a closest match is determined. These closest matches are then displayed in a section of the editor as the most relevant information (see figure 2.2).

Although the Remembrance Agent is text only, it seems plausible to adapt it to provide an audible interface, proactively suggesting relevant information when the user requires it. Also, the system provides one of the few genuinely useful intelligent interface systems in that it assists the user and allows them to focus on their primary task.

It is envisaged that these types of applications, which can adapt to the user

and provide relevant information at the correct time, will become very useful on mobile computing devices with limited interaction abilities. While there are a few ideas and prototype architectures in the field of intelligent user interfaces, there is little quantitative information about the performance of these interfaces in comparison to "non-intelligent" systems.

2.9 Contextual awareness

The term "context awareness" is a relatively new expression in the computer science arena. There have certainly been examples of contextual awareness in user interfaces in the early 1980s, as with Bolt's "put that there" [59] work described earlier. While Bolt's work does detect the context of where the user is pointing, the user still has to control the machine explicitly. It is unclear from the literature whether a system that is given contextual information by a user is indeed contextually aware or whether a contextually aware system is one which gathers the information for itself and applies some basic rules to determine the context. If the former statement is true then Bolt's work is indeed a form of contextual awareness, and the same can be said of any computing device which gathers information from a keyboard or a mouse! While in a strict sense this may be true, the author believes than a better definition for contextual awareness is a system which gathers data from the environment as unobtrusively as possible, automatically makes decisions and reacts according to a set of rules.

The contextually-aware type of systems usually require some type of sensing devices to monitor the environment. The sensors could be as simple as a pressure switch to tell where a user is standing [24], or as advanced as a video camera with an image-processing system to detect where a user is looking [25].

In the early 1990s Weiser [48] experimented with a simple form of location-

aware applications based around a hand-held device. Although this work was originally aimed at the field of ubiquitous computing, the same goals were in mind. Weiser is quoted as saying "The challenge is to create a new kind of relationship of people to computers in which the computer would have to take the lead in becoming vastly better at getting out of the way so people can just go about their daily lives". Contextual awareness is just one part of this goal, but it is an important one nevertheless. The only way a computing system can achieve "invisibility" is by helping them to understand their users in detail.

2.9.1 Gathering contexts

The gathering of simple contexts such as location has been explored in detail by Brown [54]. Using sensors such as IR transceivers and GPS, coupled with portable computing devices, it is relatively easy to locate a user in a space and present location-specific information to them. The concept of using contextual information as a trigger for an event was also introduced by Brown [55]. The author feels that as long as the sensors, hybrid or otherwise, can provide reliable and accurate data then the system will be useful in reminding people to do certain tasks at certain places. The concept of triggers is explored further by Pascoe [36] where an application running on a PDA uses GPS data to provide location-specific information to the user. The developed system was used to provide contextual information to assist an ecologist with the task of tracking giraffes in Africa [37]. The PDA was used to collect observations and general data in the field. Instead of the user having to input the location of a certain piece of information manually, the PDA gathered this information automatically from the GPS receiver when a piece of information was entered. Also, when the user returned to a place where information had been saved, the location of the user triggered the retrieval and display of information saved nearby. The study shows that the system was perceived as providing useful information on location about the giraffes movements, which before had only been obtainable

by returning to a base camp and plotting the location later that evening. While this example may not be familiar to most readers, the concept of providing information *at the right time and place* is easily recognisable as being of an advantage.

It is clear that some sensors will not function as expected in some environments. These problems have been discussed by Beadle et al. [33] and they have proposed that hybrid sensors are used to compensate for deficiencies in a single sensor. By using several sensors, each tailored for a particular stimulus, and a central processing stage with some basic intelligence, it is possible to produce a "virtual" sensor which can cope with a wider range of eventualities. There are some problems with this system, namely the ability to resolve conflicting sensor data. While some sensors will exhibit mutually exclusive types of data, such as a microphone and a light sensor, there may be instances where the range of two sensors overlap. It is clear here that the processing required to resolve this ambiguity may be complex and in some cases impossible.

More complex systems of gathering contextual information are explored by Starner et al. [69] where the use of a video camera to observe the environment fits in nicely with Weiser's ideas about helping computers become invisible. The system tracks the user's physical location by identifying targets in the environment.

The advantages of using such a system is that to cover a large area using infra-red or radio tags would be very expensive, the cost of the paper targets being very cheap in comparison. The disadvantage of this system is again to do with the targets: in order to cover a large area you would need lots of targets for the camera to recognise. While the practise of deploying strange looking targets is quite common in research labs, it would not be feasible to ask the local shop manager if you could place targets around a shop! The technical problems involved with target identification has largely been solved in [30] and

[21], with a fairly high accuracy being achieved, but the amount of processing power required today makes a mobile target identification system expensive.

The most complete contextually aware system the author can find is the Ubiquitous Talker by Nagao and Rekimoto [41]. The system uses a hand-held computer and a camera to recognise bar codes on objects. Once an object has been recognised, contextual information is displayed on the computer screen about that object. What is interesting about the work is the complete integration of natural language understanding, speech recognition and rendition. If a user points the camera at an object and asks a question about it, the system responds to the question with the relevant information. The author believes that this type of seamless interaction with the computer, contextual information and the real world is very important to the field of wearable user interfaces.

2.9.2 Contextual architectures

In order to explore contextual awareness researchers need a platform to base their experiments on. Most are developed in an ad hoc way, as the primary goal is to investigate contextual awareness rather than the systems which support the gathering of information for a contextual system. The current contextual applications are usually difficult to expand or adapt to embrace a new sensor or device. To date, the author can find only one example of a generalised architecture which attempts to provide a standard platform for contextual research and development. The Context Toolkit detailed by Dey et al. [4] attempts to provide an architecture which abstracts physical sensors from the higher level information processing via an interpretation layer which allows sensor data to be converted into contextual information. A standard API is applied to the system so that an application can request a particular type of contextual information.

The author of this work hopes that these types of systems will allow researchers to develop an understanding of what a context is, and how it can

benefit a user. The main problem is that, because there are no predefined contexts, it is very difficult to categorise anything but the simplest pieces of contextual information. It is therefore difficult to design an adequate software framework to encompass every eventuality. The only way in which this type of work will push back the boundaries is through an iterative design process, building on past successes.

2.10 Current wearable user interface systems

At the moment there are only a few people who use wearable computers and this section looks at the user interfaces employed on these machines.

2.10.1 Terminal and console systems

The traditional terminal and console systems have been used on wearable systems where the display resolution has been limited. The original terminals were 80 columns by 24 lines of text and the data rate was limited between 200 and 19200 baud. The terminal is ideally suited to low data rate displays, and the most common LED and LCD dot matrix displays connect via the serial port. There are a considerable number of applications which run from a terminal, and many wearable researchers prefer to use it as part of their user interface system.

2.10.2 The X11 system

Researchers who build their own wearable machines often install the Linux operating system. The various distributions usually come with the X11 window system which does not impose a specific ideology on the user interface. This allows individual users flexibility to configure the look and feel of their system by using various window managers, fonts and colours.

The way in which the X11 system has been designed allows it to run on non-standard displays such as small PDA screens and head-mounted displays.

Some wearable researchers use the X11 environment at a custom resolution to overcome HMD deficiencies and increase the legibility of the standard terminal on these devices.

2.10.3 The Microsoft Windows system

The Microsoft Windows operating system is the most commonly-found user interface on desktop machines, and due to the lack of specific mobile applications it is not surprising that this user interface is deployed on wearable computers in the short term. At the moment all of the wearable manufacturers use the Windows environment as the standard user interface for the machine.

Although Windows provides a common platform for development, it is sometimes impossible to configure the system to use the uncommon hardware or screen resolutions which are found on wearable platforms.

2.10.4 The Emacs system

Quite a few enthusiasts use the Emacs environment as the user interface for their wearable computers. At first the author thought it was a little strange to use a text editor as the user interface, but after a closer look it soon becomes apparent that Emacs is not just a text editor. The terminal-based version of Emacs provides an environment complete with an application programming interface and applications such as a news reader, an email client, a calendar, a diary, an integrated development environment for C++ and Java, and a web browser already integrated.

Although the graphical capabilities of Emacs are limited by the console, the functionality of the system provides the user with a consistent interface and shares some similar concepts with other user interface systems.

A few wearable enthusiasts[1] use the Emacs environment in combination

[1] Most notably Greg Priest-Dormer with his Herbert wearable computer. `http://www.cs.vassar.edu/ priestdo/herbert1.html`

with the Emacs-Speak software[2]. The commands are entered in the same way as with the terminal-based version but the output from the Emacs application is converted to a verbal rendition. This provides a lightweight environment which can render text easily as it does not require a screen, but there are obvious limitations with the graphics capabilities of this system!

2.11 Wearable software manufacturers

As the wearable field is still very young and these companies are still in their early stages, there is very little public information about these systems to date. The information included here has been found by the author in press releases and news articles over the last few years. It is obvious that a lot of information is missing but the manufacturers are secretive when queried about the details of their systems. The information provided in this section is principally for completeness, and while the author has tried hard to provide the correct information it is possible that some of it may have changed by the time of publication.

2.11.1 Wearix

Wearix[3] are currently developing two technologies. The ToolWear system is a development package with reusable software components included. This provides an environment for developing context-sensitive user interfaces, the automatic recognition of I/O devices and the use of multimedia streaming technologies. The second system, called UseWear, provides the wearable user with a complete operating environment with a media-independent user interface that adapts to the user and application requirements.

[2] `http://emacspeak.sourceforge.net/index.html`
[3] `http://www.wearix.com/`

2.11.2 WearableTech

Edeus[4] is currently under development at the WearableTech Corporation. They are developing using Microsoft Windows and have a user interface which encompasses the whole screen, providing a consistent interface to all kinds of media and documents via voice control. The system has the ability to generate documents through text, voice dictation and video capture.

2.11.3 Tangis

Tangis[5] is developing software kits that enable third party developers to produce user interfaces for wearables that work with general-purpose operating systems. The focus of their work is based on Microsoft Windows and they provide application programming interfaces for a variety of speech recognition engines, Visual Basic, Visual C++, ActiveX and the Microsoft mail and telephony APIs.

2.11.4 Charmed

Charmed[6] is developing a user system called Nanix. At the moment it is very difficult to determine exactly whether it is a user interface system, an operating system or both. There are a reports that it is a highly customised version of a Linux distribution, which places it firmly in the operating system arena, but people within Charmed refer to it as the Nanix UI. Whether this is marketing hype, or whether there is a dedicated user interface system, only time will tell!

2.11.5 Xybernaut

Xybernaught[7] is one of the leaders in providing wearable systems. Their products use the Microsoft Windows operating system, IBM's ViaVoice speech recog-

[4] http://www.wearabletech.com/
[5] http://www.tangis.com/
[6] http://www.charmed.com/
[7] http://www.xybernaut.com/

nition software and their own LinkAssist software. LinkAssist is a speech-enabled media toolkit for the creation of a hyper-linked electronic library.

2.11.6 Via

Via[8] is another of the leaders in supplying wearable systems. Their products include the Microsoft Windows operating system and IBM's ViaVoice speech recognition. Also available is a pen tablet which combines a touch pad with a hand-held screen.

2.11.7 IBM

IBM have made various press releases about their plans to sell a wearable computer, with television adverts being broadcast on both sides of the Atlantic. The machine seems to run the Windows operating system and uses their ViaVoice speech recognition to control the machine.

2.12 Chapter summary

From the literature and available commercial offerings, it is clear that there is not enough evidence or research material to substantiate the definition of a wearable user interface. Even so, one thing is obvious from the literature: the classic WIMP paradigm relies on the user having a fine degree of motor control with a pointing device but this may be impossible to achieve in a mobile environment. The problem seems to manifest around the lack of a suitable pointing/selection device in a mobile environment, as the use of menus, icons and windows (perhaps not tiled) have been pursued successfully on mobile computers such as the Palm Pilot and Psion devices in the past. The selection task has been solved on these devices by using styli to achieve direct selection, but with a HMD this would be tricky to achieve. Combined with the problem of inadequate selection

[8] `http://www.via-pc.com/`

devices, the use of HMD devices introduces unique environmental issues due to outdoor lighting conditions, and further understanding of the interaction between varying natural light and the HMD devices is needed before an attempt to address this problem is made.

Other factors which will affect the machine are the limitations and characteristics of the input devices. The use of speech to control the wearable computer is seen by many as one way in which the lack of sufficient input devices can be solved. The author believes that, while the use of speech recognition and natural language understanding may improve the interface between the wearable computer and the user, there will be situations where speech control is not desirable, and alternative forms of input will still be required.

The author believes that the following high level statements provide a good start for a future mobile user interface system:-

- As the user will be performing other tasks, a mobile graphical interface which is to be present in the users field of view, for any length of time, should be as uncluttered as possible to reduce the amount of concentration needed to operate the user interface.

- The use of pointing devices seems to be a particular problem in a mobile environment. A mobile interface should make extensive use of menuing systems with shortcuts manipulated by as few a operations as possible (perhaps a single button for each menu item).

- The use of high contrasting colours should make the use of the interface easier in varying lighting conditions through a HMD.

- A dedicated application area where one application at a time may occupy the space is paramount in reducing the amount of time the user spends concentrating on the user interface.

- The user interface, menu selection, application selection and manipula-

tion should be controllable via speech as well as a point/click and direct selection.

Alternative user interfaces have been explored in the multimodal, contextual awareness fields where systems gather information from the environment through obtrusive and unobtrusive means. Most of the multimodal systems that are trying to address deficiencies in the user interfaces are usually confined to where the applications or interaction devices are not standard. The data from the devices are processed and an appropriate form of response is generated.

Most of the intelligent user interfaces in the literature use some form of agent to process information and two fundamental types of agents highlighted in the literature. The first type reacts to a user's commands and returns the relevant information, while the second type interprets information over a period of time and makes decisions to provide information based on certain criteria.

In the literature, the concepts of autonomy, awareness, intelligence, and the ability to understand the user through various interaction mechanisms are considered important features of a future wearable user interface. But the biggest hurdle for any wearable user interface designer is the lack of a well-defined software framework to facilitate research and development. This means that any exploration into alternative user interface paradigms that use contextual awareness or multimodal systems is difficult due to the lack of infrastructure and the low amount of software reuse available.

2.13 Proposed research

The aim of this research is to explore context-aware user interface issues and to determine an operational framework to pursue these investigations. The author attempts to combine traditional user interface techniques with sensor and agent technologies to form a single development platform for contextual user interface research for deployment on a wearable computer.

The first piece of work provides the reader with an empirical study of the some common wearable interaction devices. This study is used to determine the operational parameters of the devices and construct a set of guidelines with which a traditional graphical user interface systems can be constructed. The next piece of work involves the design and construction of a contextual agent based system, which will allow dedicated agents to process information from various sensors and react accordingly. Finally these two pieces of work are amalgamated by incorporating a a graphical user interface, designed in accordance with the proposed guidelines in chapter 4, with the contextual agent system.

The Sulawesi contextual agent system uses several concepts from the previously mentioned literature review.

Sensor systems

Applications rely on receiving information from a variety of sensor systems such as speech recognition, accelerometers, global positioning systems, keyboards and pointing devices; and are constructed using a range of hardware and software. The use of sensors to gather information about the environment is fundamental in many multimodal applications [19, 59] and should be incorporated into the framework. A standard approach should be applied to the construction of interfaces to these sensor systems, to address problems highlighted by [27], and a single paradigm of data delivery used[9]. This will simplify the task of application development and make the system expandable, allowing the simple integration of future sensor technologies.

Natural language

When a person is mobile a common form of communication with another person is speech. In order to make the user interface more accessible while on the move the use of speech recognition and natural language parsing go hand-in-

[9] This is described in section 5.2.1.

hand[10]. A system which can understand fairly natural sentences and interprets commands from them has already been explored by [61], and the incorporation of a speech recognition/sentence processing stage within the Sulawesi architecture would provide the translation from human voice into text. The use of a rule based language parser interface would allow speech and other natural forms of expression like handwriting to be transformed into a command which can be executed. The architecture should also be adaptable enough for an third party application and the related command/sentence processing to be incorporated into the system with ease.

Contextual rendering

A mobile application should provide information when it is most appropriate and in a suitable format. The use of the multiple sensors to gather information about the environment can be used to determine the users situation. The information could determine the users location, physical position, whether the user is sleeping or whether they are speaking to somebody. This contextual information should determine how the machine communicates with the user. A separate rendering layer would allow contextual rendering decisions to be made[11]. The system should be able to render information in various formats depending on the current context of the mobile user. This rendering layer should normally be hidden from applications while providing a standard interface for the applications to request that information be rendered.

Agent based applications

The construction of an application should be as simple as possible. Sulawesi should be able to hide the complexities of the multiple sensor systems, the natural language parsing and the contextual rendering systems from the application by providing a standard interface into the management system. This allows de-

[10] This is described in section 5.2.2.
[11] This is described in section 5.2.3.

velopers to focus on the purpose of their application rather than having to worry about the complex sensor systems that lie beneath. As with the systems developed by [17] there would two different types of agents within the system, type A which reacts to events in the environment, and those which interpret a higher level of understanding of the environment an react when certain criteria are met.

System management

The combination of multiple sensor systems, contextual rendering systems and agent-based applications implies that some form of management is needed to provide a standard interface for all the components to communicate by. Sulawesi has been designed and implemented to tackle what has been considered to be important challenges in a wearable user interface, namely the ability to accept input from any number of modalities, and perform (if necessary) a translation to any number of modal outputs. The system would consist of a management agent, similar to [52], which combines the sensor and contextual rendering systems and would also control the flow of messages and construction of agents within the framework.

Chapter 3

The Construction of a Wearable Computer

3.1 The wearable computer, "Rome"

The wearable computer that has been developed is similar in design of the Tin Lizzy[1] prototype by Thad Starner. While the architecture uses PC104 cards in a similar way to the Tin Lizzy, the enclosure contains the whole machine including the power supply, the head-mounted display controller and a GPS receiver. The most obvious visible difference from the Tin Lizzy is the use of a cast aluminium enclosure with a belt attachment, this allows the heat generated by the machine to dissipate into free air. This design also allows the whole machine to be placed on one side of the hips, with a battery pack on the other side to counter balance the weight of the machine. A modified M1 head-mounted display has also been constructed which combines a boom microphone and speaker. The construction and use of the machine has enabled the author to get a better understanding of how a user interacts with a wearable computer, and some of the issues surrounding the interaction devices.

[1] `http://wearables.www.media.mit.edu/projects/wearables/lizzy/index.html`

3.2 Wearable computer technical specifications

Hardware

PC 104 Cyrix 133MHz 586 microprocessor, IDE, floppy, keyboard, serial, 20 MB RAM *(AMP[2])*

PC 104 VGA card, trident chipset 1/2 MB RAM (AMP)

PC 104 Single PCMCIA type II controller + slot (expandable for two cards) (AMP)

PC104 Four port serial card (AMP)

1.0 GB 2.5" Seagate IDE hard disk.

HCI devices

Twiddler keyboard

V1, a modified M1 head mounted display.

I/O devices

PCMCIA D-LINK DE-650 Ethernet card

CMC Allstar 12 channel GPS receiver

Custom IR transceiver

ADXL-202 accelerometer

Power supply

12V DC input, 5V @ 20W, 9V @ 9W output.

Batteries, 2x Duracell DR11 (6V @ 3.6 Ah) NiMH

Software

Linux (Redhat 5.2)

Sulawesi

[2] Advanced Micro Peripherals, The Chancel, St John's, Little Ouse, Cambridgeshire, CB7 4TG

3.3 Construction

The enclosure for the Wearable Computer has been constructed using an off-the-shelf 82mm × 188mm × 120mm cast aluminium box. Aluminium was preferred over a plastic or ABS case because of the combined heat dissipation properties, the robustness and the weight of the material. The case (see figure 3.2) has two slots cut in the back into which a leather belt is inserted. The slots are positioned at the top of the case to allow the machine to hang on the waist and the upper part of the leg. This gives the machine some support when the wearer is moving about. A laptop hard disk is used as they have been designed

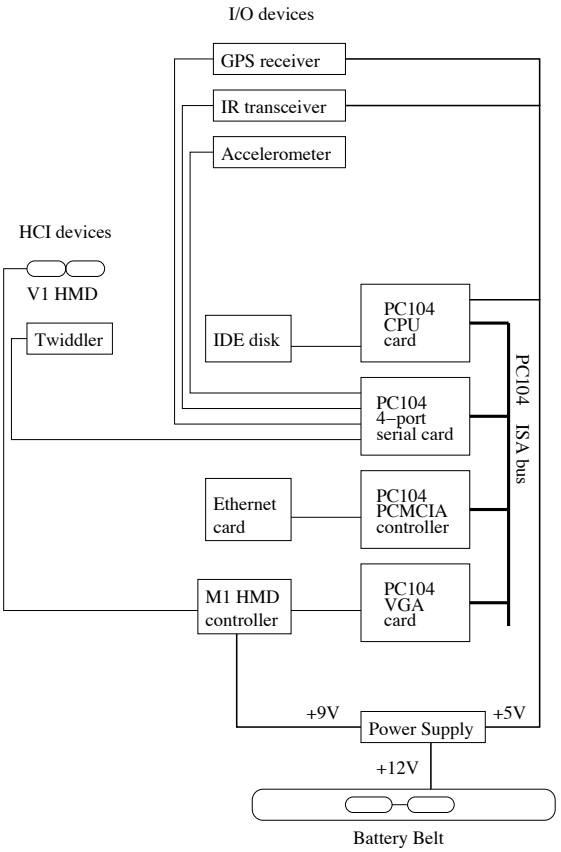


Figure 3.1: The wearable computer system architecture.

to operate under higher G forces and withstand vibrations better than a normal hard disk. The hard disk has been positioned at the bottom of the case, which means that it is more susceptible to adverse vibration. The main reason for not

placing the hard disk higher up in the case, and thus reducing vibration, was due to the heat dissipation of the system, the accessibility of external connections and the physical location of the PC104 hardware interfaces.

The two original connectors for the Twiddler were replaced with a single 4-pin mini DIN connector and a 4-pin mini socket was added to the case, this reduced the space requirements of the external Twiddler interface. Other connectors, such as the power in/out, the serial ports and the GPS antenna have been placed at the top of the machine.

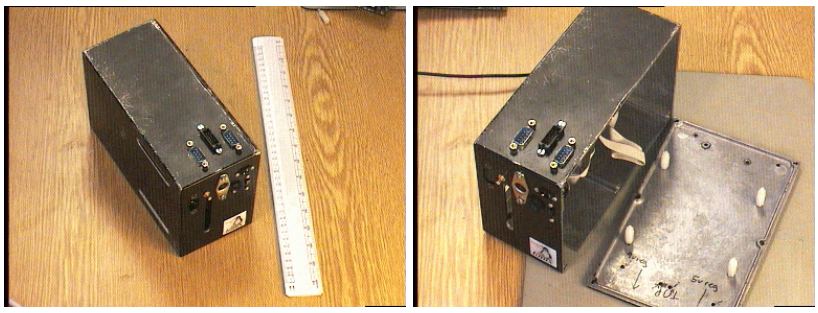


Figure 3.2: The main case when opened and closed, the external connections and the two belt slots can be seen on the side of the case (the white ruler is 30cm in length and has been included only as a rough indicator of the scale).

The core of the system is based around PC 104[3] cards. These are placed in the centre of the the case. The hard disk, the PC 104 cards and the power supply are all attached to the lid of the case. The PC 104 bus has been placed parallel to the sides of the lid: this means that when cards are placed in the PC 104 stack the connectors point toward the top and bottom of the case and therefore make the cabling a little easier.

Figure 3.3 shows the machine before construction. The left hand image shows (from top left to bottom right) the VGA converter, the GPS antenna, the main case, the Twiddler, the head-mounted Display, the PC 104 components, various circuits and cables, a battery pack, various screws, mounting brackets, and a leather belt. The right hand image shows the PC 104 components (from top left to bottom right): a four-port serial card, a PCMCIA docking card, the

[3] `http://www.controlled.com/pc104faq/`

hard disk, the GPS receiver, the motherboard (with DC-DC converter attached), and the VGA card.

The first PC 104 card placed in the stack is the CPU and motherboard card (see figure 3.4). This has been designed to allow the CPU heat-sink to be pressed against the case lid to help with the heat dissipation. The next card on the stack is the four-port serial card which slots onto the motherboard card.

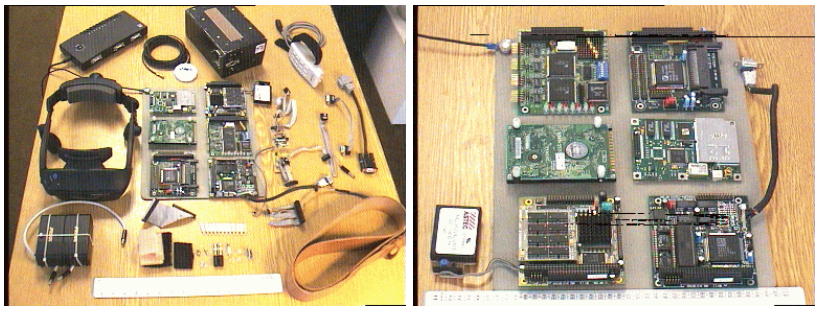


Figure 3.3: The complete system in its component parts.

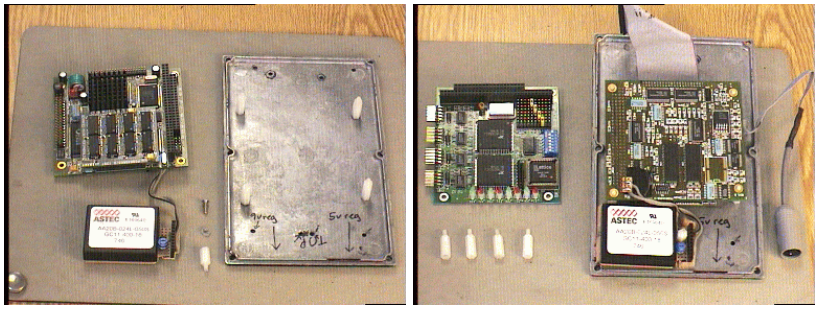


Figure 3.4: The lid with the motherboard, serial port card and DC-DC converter.

Two 6V Duracell rechargeable camcorder batteries are used in serial to provide 12V at 3.6 amps into the Wearable Computer. The power supply for the PC 104 cards requires a 5V rail capably of supplying 2.2 Amps (peak) and a 9V rail is needed for the VGA to NTSC scan converter. The 9V supply is generated by a simple regulator. The 5V power rail is generated by an off-the-shelf 20 Watt DC-DC converter. This converts 12V to 5V with a specified efficiency of 80 – 90%, which means that at most only 20% of the input power is lost as heat. Because the DC-DC converter is fairly efficient a heat sink is not required

as the device functions at full load in free air without any undesirable effects.

The input to the wearable has been chosen to be 12V so that the machine could be connected to a power supply or a car cigarette lighter. After the initial construction of the system the battery lifetime was determined by running the wearable computer until batteries were unable to supply power and the 9V regulator shutdown. At this point the HMD powered down and the running time could be determined. The system was left running idle and this test was performed four times, each time the batteries were able to supply power for 3 hours (±5 minutes). The peak power consumption has been measured at 15 Watts when the machine is booting, when the CPU is idle the power consumption is reduced to just over 6 Watts, and the machine dissipates 8-10 Watts when running the X Window system and being used.

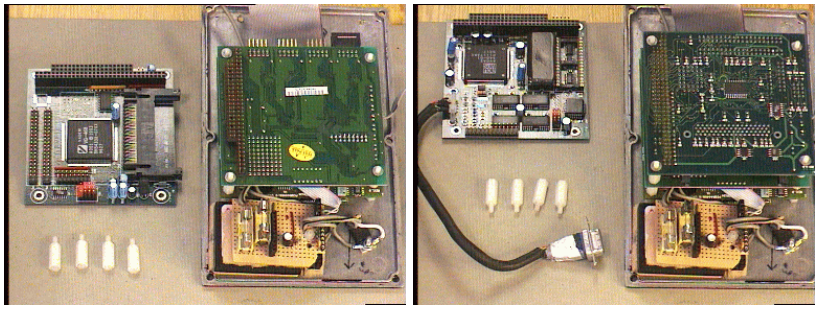


Figure 3.5: The PCMCIA and VGA card ready to be fitted.

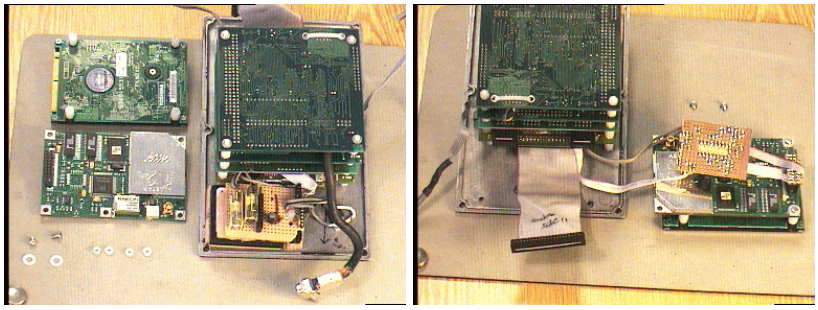


Figure 3.6: The hard disk and the GPS receiver card ready to be fitted.

The next card in the stack is the PCMCIA controller. A slot has been cut in the top of the case to allow insertion and removal of PCMCIA cards. The last card in the stack is the graphics card. The VGA connector has been removed

from the PCB and an extension cable made which bolts to the inside of the case (see figure 3.5).

The hard disk for this machine has been taken from a Toshiba Libretto. The GPS receiver is roughly the same size as the disk, so they have been bolted together and placed next to the PC 104 stack (see figure 3.6). Once all the com-

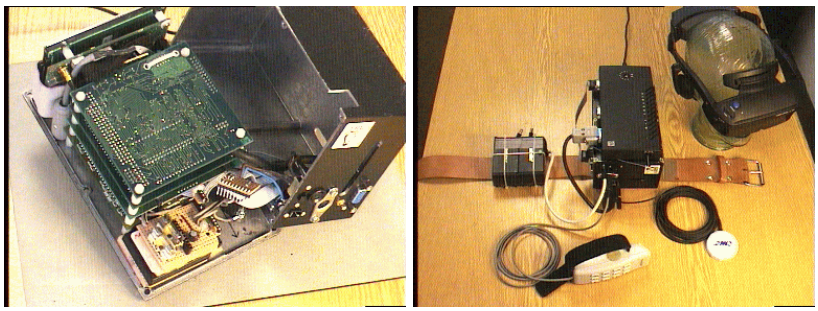


Figure 3.7: The complete wearable system.

ponents are assembled, the case is closed and the peripheral items are attached. Figure 3.7 shows the completed system with Twiddler, head-mounted display system and GPS antenna. The VGA scan converter fits on top of the case and the batteries are attached to the belt.

3.3.1 Power supply problems

After the power supply had been tested with light loads, it was considered to be safe to plug into the PC 104 cards. At first only the PC 104 motherboard, the VGA card and the hard disk were used to test the system. The system was booted and the machine powered up as expected. The machine was dismantled and the other PC 104 cards were fitted, but this time when the power was turned on nothing happened. The monitor plugged into the machine did not wake up from the suspend mode, there was no initial BIOS screen and no initialisation of the graphics card.

The system was reverted back to the minimal set of PC 104 cards. This time the system powered up and the monitor woke up from being suspended, but there was no BIOS prompt and the machine appeared to have hung. At this

stage it was noticed that the hard disk was not spinning (as it usually does when powered up). The hard disk was tested in a laptop machine and appeared to be working correctly. After several conversations with the manufacturer, the fault was found to be caused by the DC-DC converter. This functioned correctly when a low load was being pulled from the supply rails, but when a higher load was connected, the converter would respond to the initial start-up current by overshooting the desired voltage. This caused a very short, large spike to appear on the 5V rail, and this was affecting the motherboard by corrupting the BIOS and stopping the machine from booting. Although the DC-DC that was

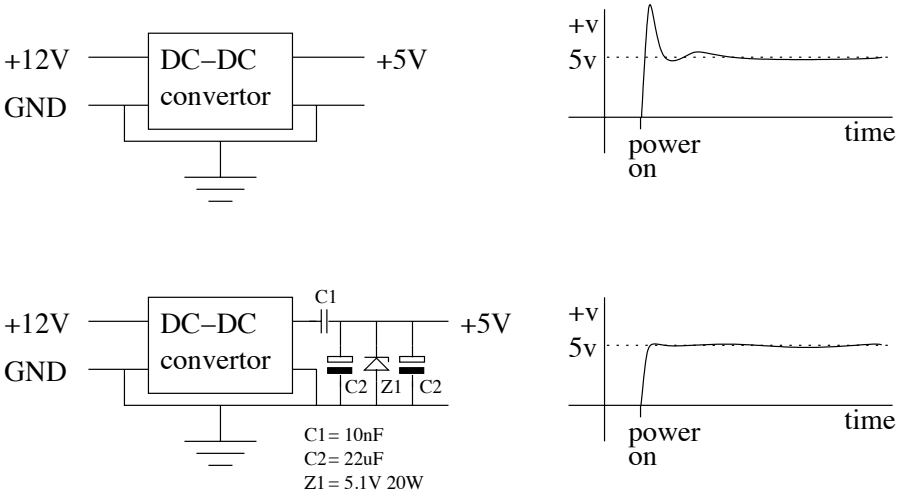


Figure 3.8: Output supply rail clamp.

ordered included a built in regulator, the device received was a slightly different model which did not regulate the output supply. Once the problem had been identified a few smoothing capacitors and a zener diode clamp were added to the output supply rail to stop the spikes (see Figure 3.8). This made certain that the supply rail could not rise above 5.1V and corrupt the BIOS.

3.4 Software and operating system configuration

At the time of development, the operating system chosen for the wearable computer was RedHat version 5.2, upgraded with the Linux 2.2.9 kernel. The main

reason for using a Unix-based operating system was the ability to control the machine without a graphical user interface, and to easily integrate a prototype user interface with the current functionality of the operating system. Since the author's research was primarily concerned with mobile user interfaces the use of a windowing oriented operating system would have placed constraints on the way the machine presented information to the user (this is explained further in section 5.2.3). Operating systems such as Microsoft Windows 9x and MacOS require the user to control the machine via the manipulation of graphical objects, control via any other means can be difficult without developing custom software to achieve a desired task. On the other hand, operating systems such as Linux and BSD provide adequate control of the machine via command line user interfaces. Built in shell scripting languages enables existing software to be manipulated and plugged together via simple commands and controlled by alternative interfaces. The choice of a command based operating system over a graphical based one was solely due to the ability to develop alternative user interfaces and control existing software via scripting languages easily.

The original head-mounted display was a set of I-glasses by i-O Display Systems[4] with a resolution of 320 × 240. The XFree86[5] Setup tool was used to configure X to use 320 × 240.

The Twiddler driver software was used to detect the Twiddler keyboard on the correct serial port. The Twiddler driver was configured and added to the `/etc/rc.d/rc.local` script, which starts the driver when the machine is first booted. Initially the console mode of the Twiddler driver functioned correctly but when X windows was started the Twiddler would not work. The Twiddler driver was using a program called a2x to control the X session, and it was complaining that it could not load the "libXaw.so" library even though it was clearly in the library load path. The author copied the correct library into the

[4]The company has recently changed its name from Virtual I/O, see `http://www.i-glasses.com/index.html`

[5]`http://www.xfree86.org/`

same directory as the a2x executable, after which the program loaded correctly.

After upgrading to the 2.2.9 Linux kernel it was noticed that the machine would hang for about a minute when first booted. This happened between the "uncompressing kernel" message and the next stage of the boot loader. After approximately one minute the kernel would boot and everything functioned correctly. The author could not remember the 2.0 kernel exhibiting this behaviour. It was observed that with the keyboard disabled in the BIOS and no QWERTY keyboard present, the 2.0 kernel did not exhibit this behaviour. In contrast, the 2.2 kernel exhibited a significant time delay between uncompressing the kernel and booting. The author speculates that the 2.2 kernel is trying to ascertain whether a keyboard is present and is waiting until the device detection code times out.

The boot process has been configured to insert the relevant PCMCIA modules into the kernel. The card services program detects when a PCMCIA card is inserted and has been configured to run a script depending on what type of card is inserted. A script has been written to allow the automatic configuration of the network when a PCMCIA network card is inserted, and automatic network shutdown when the card is removed. The network script works by bringing up the new network interface with the correct IP address; the default route is then configured to point to the interface and the correct DNS entries are added to the `/etc/resolv.conf` file. When the network card is removed, the script brings down the interface and removes the default route and the DNS entries for the device.

The default Linux kernel automatically detects 2 serial ports. The addition of a four-port serial card means that an initialisation script had to be written to instruct the kernel of these additional devices. The device entries have been made in `/dev/` using the `mknod` program (with the correct major and minor numbers of the serial ports). Once the device entries existed, a script initialises the serial ports at boot time. The script uses the `setserial` program to configure

the device node, IRQ and memory address range of the serial ports.

3.4.1 Software configuration problems

The first problem was noticed when the machine booted. The serial port initialisation script only detected four serial ports with an error message of "*/dev/ttyS4-5 not a valid device*". This was found to be a limitation of the default kernel which only supports four serial port devices. The *rc.serial* script was changed to only initialise four of the six serial ports (as only four were needed) and the machine was re-booted. Although the four serial ports were being configured the first serial port was very intermittent. The `setserial` program was used to report the correct IRQ, memory address range, and uart for the device. The serial line status program, `statserial`, was used to check the status of the serial lines. The `statserial` program would not initialise the first serial port and would produce a segmentation fault when asked to do so. The other three devices were tested with `statserial` and all worked as expected. This fault was noticed intermittently on both the 2.0 kernel and the 2.2 kernel. Eventually the fault was identified; The serial port initialisation script was setting up the serial ports using `setserial` but, until a program accessed the device, the IRQ for the serial port was not reported in the `/proc/` interrupts table. This table was being used by the card services program to allocate dynamically an IRQ to a PCMCIA card. When the PCMCIA network card was inserted, the card services program was allocating the IRQ for first serial port to the network card. This resulted in the `statserial` program producing a segmentation fault when the network card was inserted. The problem has been resolved by configuring the card services program to reserve IRQs for the serial ports and to not allocate them to the PCMCIA devices when they are inserted.

3.5 Construction of the V1 head-mounted display system

The original Tekgear M1 HMD has been modified and has been nicknamed the Visor Mk 1 (V1). The original Tekgear M1 came with a "light duty" headband[6] which is very similar to the ones used on the older style Walkman headphones. The M1 could be placed over the eye but the headband did not feel safe enough on the author's head to walk down the road with. The M1 was opened, the

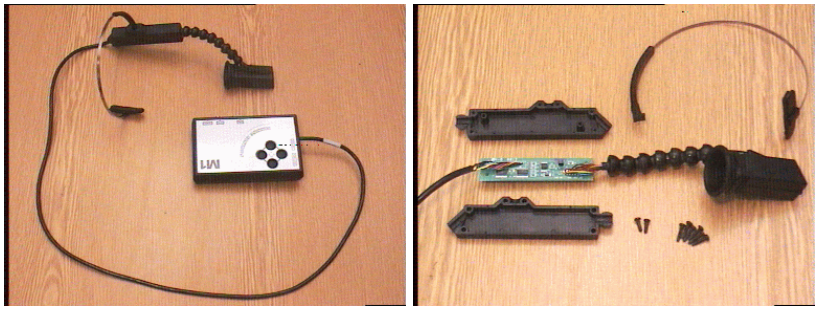


Figure 3.9: The M1 HMD.

head fastener contained a small circuit board with a few components on (see figure 3.9) where twelve wires entered the board from the main unit, and sixteen wires went to the eyepiece. Consultation with Dr. Jerry Bowskill, previously at BT research labs, revealed that the circuit could safely be moved from the head fastener to the main body of the M1 controller. It was decided at this point that the M1 eyepiece should be detached from the head fastener and placed on a more suitable head mounting.

The author has borrowed Thad Starner's use of safety glasses to mount the display, with a few modifications (see figure 3.10). The main problem with moving the M1 eyepiece was due to the number of wires entering the unit.

A standard VGA cable has fifteen wires and therefore could not be used so a separate wiring loom for the headmount was constructed. As the display device was going to be placed on the head the loom needed to be as small and

[6] Newer versions of the M1 come with a much more sturdy "heavy duty" headband.

Figure 3.10: The V1 HMD.

flexible as possible. If the cable was too large the weight of the cable would pull on the headmount and may becoming uncomfortable to the user, if the cable was too stiff this would have restricted the movement of the head. A section of small flexible VGA cable was connected to each arm of the safety glasses, providing thirty cables to the HMD unit. It was decided to add a speaker and boom mic to the headmount. A small bracket has been made and is used to attach the speaker/mic to the glasses. The four cables from the speaker/mic unit are connected to some of the spare wires in the loom.

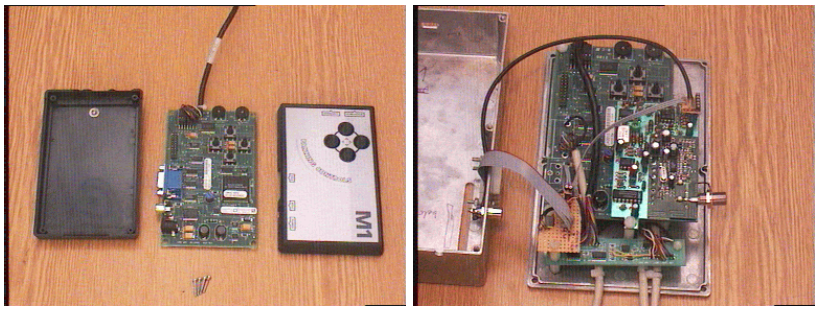


Figure 3.11: The M1 controller dismantled and placed inside a box.

A GND, 5V, 9V and 12V supply have been connected to the loom and an LED was fitted to the headmount as a power indicator. To date, there are still eight wires spare in the loom, and it is envisaged that in the future a small camera could be fitted which would use four or five of these spare wires. The other end of the loom have been run into a small aluminium box where the M1 controller is located (see figure 3.11). A pass through power/audio/video cable has been made to connect to the main Wearable Computer.

3.6 Conclusions

The resulting wearable computer has enabled the author to carry out most of the research presented in this thesis. While the construction of the machine has been engineered to withstand the rigours of being in a mobile situation, the size and weight of the machine remains problematic. The processing power available from the machine is not adequate enough to allow integration of the Sulawesi system and the speech recognition software, nevertheless, the experienced gained in the construction and use of the system has proved useful in the remainder of the research presented in this thesis.

Chapter 4

A Study of some Wearable Interaction Devices

4.1 Introduction

The current generation of wearable computers use various devices for controlling the user interface. There have been several novel input devices such as the wear-clam [73] and the wristcam [7], but these are research prototypes. Unless wearable researchers are specifically designing mobile interaction devices, the most popular commercially available input device at the moment is a single-handed keyboard called the Twiddler[1].

The Twiddler is a portable chording keyboard with an internal movement sensor which can be used to control a cursor in a graphical user interface. There have been claims[2] that the typing speeds achieved on the Twiddler are similar to that of a standard QWERTY keyboard, but no proof of these claims have been produced.

Many researchers use some form of head-mounted display on their wearable computer. The suitability of these displays for virtual reality user interfaces has been studied in laboratories [60], but the uncontrolled real-world environment in

[1] `http://www.handykey.com/`
[2] `http://www.handykey.com/site/testimonials.html`

which a wearable display will be used prompts some interesting questions. Some of these questions, such as *can the display be seen in direct sunlight?* can be answered and designed around using traditional engineering techniques. On the other hand, questions such as *is an augmented interface usable when walking?* are not so easy to answer and require studies to provide an insight into the interaction between the devices and the user interface.

Figure 4.1: The Twiddler single-handed keyboard.

The aim of these experiments is to provide some quantitative information about the current interaction mechanisms used by wearable researchers. The interaction devices play a critical part in determining how usable the wearable user interface is. It is intended that there will be sufficient information from these experiments to provide guidelines for designing wearable user interfaces using these devices.

4.2 Experimental design

The following experiments were designed to test some general aspects of the various interaction devices. A within-subjects [2][3] design was used, and a balancing operation was employed to reduce the effects of learning and biasing the results. This balancing operation [2] makes sure that the first test a subject undertakes is alternated between different subjects.

Some six subjects were used for the tests. Two subjects were under 21, two were between 22 and 31, and two between 31 and 40. Of the six subjects only one was female. All the subjects were computer literate and experienced in using a QWERTY keyboard. Of the subjects, half were aware of the Twiddler but had never used it and the other half had never even heard of the Twiddler. Six subjects would normally be regarded as a small sample for a trial; however, as we shall see, statistically-significant results can be inferred even from this sample size.

An anechoic chamber was used during the experiment to minimise external influences, but it is noted that the controlled environment was not realistic for either a desktop or a wearable computer. A computer equipped with Windows95 and a 17-inch monitor with a screen resolution set to 640×480, a QWERTY keyboard, the Twiddler and two different types of head-mounted displays were connected to the machine.

The experiments were performed so that the author could gain an insight into interaction mechanisms. The speed and accuracy of the text entry capabilities, and the cursor manipulation achievable with the Twiddler are compared to a standard QWERTY keyboard and a mouse. The study also compares the speed and accuracy of different head-mounted displays to determine what part they play in affecting the results obtained from Twiddler.

Without this information it would be impossible to design a graphical user

[3] Chapter 12, page 348.

interface to take advantage of the Twiddler's characteristics.

4.3 Text entry experiment

The first experiment was designed to compare the speed and accuracy of a text entry task using a standard QWERTY keyboard and the Twiddler. The goal was to type in text into a full screen Java applet. To reduce the effects of learning two different paragraphs of text were used and they were approximately 550 characters in length (see Appendix A) The first paragraph was used to familarise the user with the equipment and the method, while the second was used to gather data for analysis. The paragraphs took 2–3 minutes to type on a QWERTY keyboard and were considered to be of sufficient length to avoid any short-term learning effects. The paragraphs contained only letters of the alphabet: the full stops, punctuation, and special characters were removed from the text during the tests. The paragraphs were chosen for their clarity and conciseness and, while they had no particular statistical characteristics, it was felt that they provided a sample of what is expected to be typed with the devices.

The applet recorded each keystroke and the time at which the key was pressed, and the data were analysed to determine the speed and error rates achievable with the devices.

4.4 Direct manipulation experiment

The second experiment was designed to test the speed and accuracy of positioning a graphical cursor on the cathode ray tube (CRT) screen using a standard mouse and the Twiddler's pointing device. The goal was to position the cursor over a target and select it with the pointing device.

A full screen Java applet, seen in figure 4.2, generated twenty graphical targets, one at a time in a random position on the screen. The targets consisted of three circles: the outer one was black with a radius of 50 pixels, the middle

circle was white with a radius of 40 pixels and the inner circle was red with a radius of 30 pixels. When the target was selected it would disappear and a new target would appear at a random position. The applet recorded all cursor movement and cursor events, which allowed the data to be analysed off-line. The subjects were positioned approximately 60cm away from the CRT screen during this test

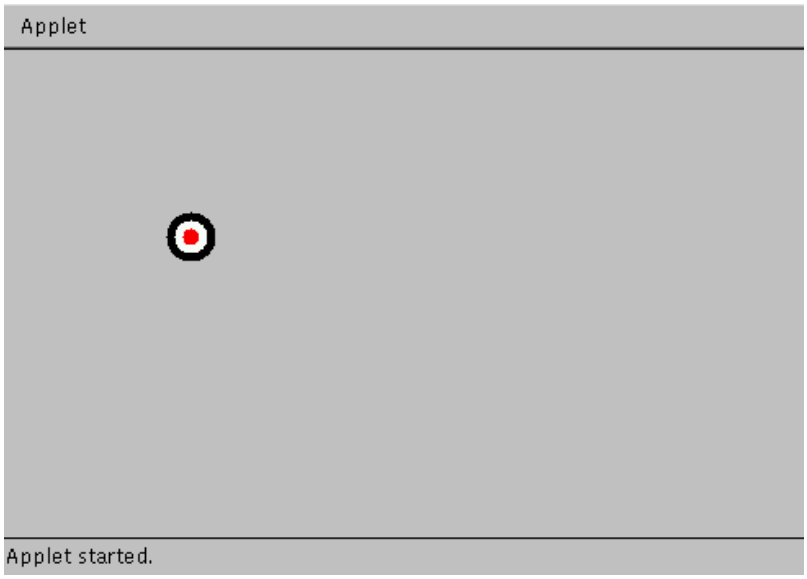


Figure 4.2: Applet displaying a graphical target.

4.5 Direct manipulation vs target size experiment

In experiment three, the user was asked to select a graphical target which was randomly placed on the CRT screen. The difference between this and the second experiment was that the applet displayed 15 targets of one size; then the size of the targets was reduced and another 15 were displayed; and so on. The target size was calculated using the central disc to determine the sizes of the two outer annulae. The size of the middle circle was determined by adding 10 pixels to the size of the centre circle, and the overall target size was determined by adding 10 pixels to size of the middle circle. At the start of the test the centre circle radius was set to 50 pixels, and decreased by 10 pixels every 15 selected targets. This

resulted in an experiment using 5 sets of targets with the centre radii set to 50, 40, 30, 20 and 10 pixels respectively. The object of this experiment was to see how small a graphical target could be reliably selected using the Twiddler.

4.6 Display technologies experiment

The fourth experiment was designed to compare a normal CRT screen, an augmented head-mounted display system and an immersive head-mounted display, the aim being to find out whether the head-mounted displays have an effect on the pointing devices. The experiment was similar in design to the third experiment with all combinations of input devices (Twiddler, mouse) and output devices (screen, augmented HMD, immersive HMD) being tested.

Figure 4.3: The Virtual-IO glasses.

The HMD display system used during this experiment was a set of Virtual-IO glasses (see figure 4.3). This HMD uses two 320×240, 8-bit colour LCD screens and two half-silvered mirrors to reflect the images into the user's eyes. The Virtual-IO glasses work by splitting the 640×480 VGA signal into two, sending the even scan lines to one eye and the odd scan lines to the other. The HMD also has a clip-on visor which enable them to swap between augmented (see-through) or immersive modes (occluded) easily. The subjects were again placed approximately 60cm away from the CRT screen, while the HMD displays were positioned approximately 3cm away from the subjects eyes.

4.7 Monocular displays experiment

The last experiment was designed to compare and contrast an augmented monocular HMD and an immersive monocular HMD system. The design was similar to the fourth experiment with all combinations of input devices (mouse, Twiddler) and output devices (augmented HMD and immersive HMD) being tested.

The Virtual-IO glasses, with one of the eye pieces removed to make them monocular, was used as the augmented HMD system. In this configuration, the Virtual-IO glasses only displayed the even lines in a 640×480 image, producing a visible resolution of 320×240 with 8 bit colour and the HMD units were placed approximately 3cm away from the subjects eyes.

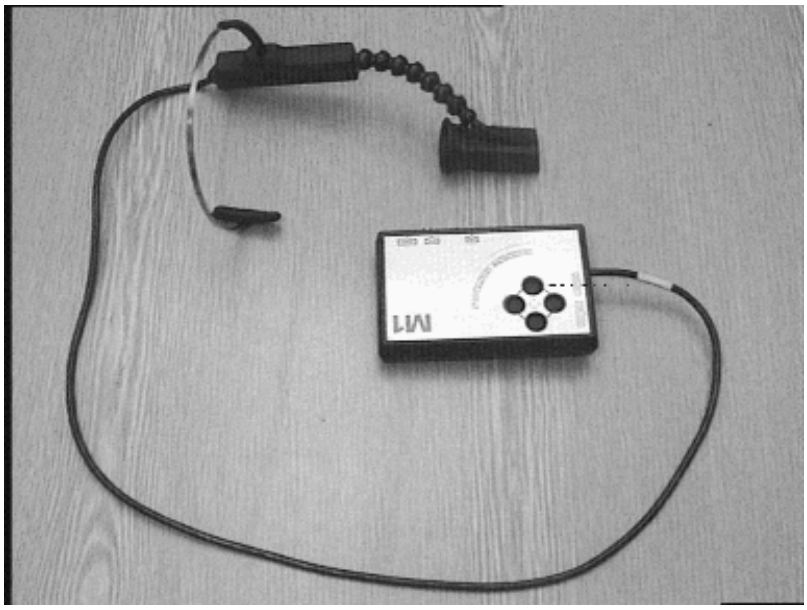


Figure 4.4: The M1 head-mounted display.

The Tekgear M1 (see figure 4.4) was used as the immersive HMD system. This provides a resolution of 320×240 grey scale, though Tekgear comment that

> "The M1 utilizes an additive sampling technique that compresses the complete VGA frame into the 320×240 format with minimal apparent loss in quality" [4].

[4] quoted from the M1 FAQ on `http://www.tekgear.ca/displays/m1faq.html`

Even though the devices have a similar display resolution on paper, the M1 appeared to be clearer. This test was designed to find out whether the appearance of a higher quality HMD affected the manipulation devices.

4.8 Observations during the experiments

All subjects were observed from outside the sound-proofed room during the tests. It was noted that during experiment one a few people decided to use the Twiddler with two hands, one hand to hold it with and the other to press the keys.

After all experiments involving the Twiddler, the volunteers complained of fatigue in the hand and wrist, and one subject stopped half way through experiment one and refused to continue due to severe cramp in their index finger. This subject completed the other tests successfully.

4.9 Results and discussion

4.9.1 Experiment one: text entry speed

The text speed entry times were measured by calculating the mean time difference between each key-press. There are cases where two key-presses resulted in one character being entered, for example an upper case letter at the start of a sentence required the shift key and then the character key. The briefing sheet given to each user stated that upper case letters were to be entered as lower case, but some users still entered upper case letters during the tests. The shift keys were not used in calculating the time difference between the characters.

	Twiddler	QWERTY
Paragraph 1	2415 $\pm$ 603.9	302 $\pm$ 80.6
Paragraph 2	1828 $\pm$ 373.8	365 $\pm$ 91.2
Experienced user	1651(mean)	420(mean)

Table 4.1: Mean and standard deviation of character entry times (milliseconds).

The analysis of the test data can be seen in table 4.1 with the null and alternative hypothesis defined as follows, H_0: $\mu = \mu_0$, $H_1 : \mu < \mu_0$ (where μ and μ_0 are the respective means of the QWERTY and Twiddler). The results were analysed using a one-tailed t-test[5] where t was calculated using the following formula:

$$t = \frac{\overline{x} - \mu_0}{s/\sqrt{n}}$$

where $\overline{x}$ is the mean value of the test samples

μ_0 is the hypothesized population mean

s is the sample standard deviation

n is the number of test samples

The analysis shows that the results were significant at the 1% level ($t_{<0.01,5} = 8.75$). It can be seen in table 4.1 that the first test produced a result of 2.4 seconds per character for the Twiddler compared with 0.3 seconds for the QWERTY keyboard. The results of learning for the Twiddler device can be seen when the first and the second paragraph results are compared. The character entry time has reduced by approximately 0.6 seconds and the standard deviation has dropped to 2/3 of the initial value.

A person who had been using the Twiddler for over a year was also asked to complete the second paragraph to provide some form of direct comparison. Although only one experienced subject was used, the character entry times for the second paragraph were within the mean and standard deviation of the other users.

The results in table 4.1 suggest that the Twiddler is approximately 4.5 times slower than the QWERTY keyboard for entering text. Using the results obtained, the average person in these tests could type approximately 180 characters a minute on a QWERTY keyboard, whereas with the Twiddler they could only achieve approximately 33 characters a minute. The speed was also compared

[5]See [2] pages 399-401.

to a relatively experienced user of the Twiddler. The times achieved by the test sample were comparable to the experienced user, but even the experienced user was still in the order of 4 times slower when compared to the QWERTY keyboard. From these results the author concludes that the Twiddler can be used without much training as the inexperienced subjects were approaching the same speeds as the experienced subject.

After the experiment, all of the subjects complained of fatigue in the hand and wrist. It is speculated that this fatigue was partly due to the length of time spent using the Twiddler and partly due to mental frustration at how slow they found the device.

4.9.2 Experiment one: text entry accuracy

The second test was used to determine the accuracy of the Twiddler device when compared to a QWERTY keyboard. The subjects were asked to input the text exactly as it appeared on the test sheet, and the input accuracy was observed by counting the number of backspace keys that were pressed. This provided information that a user had spotted a mistake during the test and had corrected it.

	Twiddler	QWERTY
Paragraph 2	34.6 ± 16.3	17.4 ± 2.5
Experienced user	11	8

Table 4.2: Number of corrections.

The results from the test data were analysed using a t-test, and were found to have a 96.2% probability of being significant ($t_{<0.05,5} = 2.36$). By dividing the number of errors by the number of characters (550), it can be seen that the percentage of errors for the experiment was approximately 6.3% and 3.16% for the Twiddler and QWERTY keyboard, respectively.

The results indicate that the Twiddler user produces approximately twice the number of errors when compared to the QWERTY keyboard.

4.9.3 Experiment two: cursor speed

The analysis of the data was used to determine the raw speed of the person using the device. The user was asked to select any region of the targets as quickly as possible. In this test the pixel movement times were calculated by recording the route traveled from the last target to the current target and dividing that distance by the time taken to reach the next target. This produced a timing value in pixels per millisecond. The data were analysed using a t-test and were found to be significant at the 1% level ($t_{<0.01,6} = 62.7$). Analysis of the data can be seen in table 4.3.

	Twiddler	Mouse
Cursor Speed	0.196 ± 0.01	0.506 ± 0.06

Table 4.3: Cursor speed (pixels per millisecond).

The observations of the device during the tests seem to imply that there is a non-linear relationship between the cursor movement on the screen and the amount of physical movement that is needed when using the Twiddler. This is different to the mouse which (in unaccelerated mode) has a linear relationship between the mouse movement and the cursor moving on the screen. The author concludes that this is due to the non-linear properties of liquid movement sensor within the Twiddler, further tests would need to be performed to determine the exact relationship between physical and cursor movement.

The results show that the Twiddler is approximately 2.5 times slower in manipulating a cursor around a 640×480 graphical display and selecting a target with a radius of 50 pixels. This result can have a significant impact on the performance of a standard desktop application with a complex user interface.

4.9.4 Experiment two: cursor accuracy

This analysis was used to determine the accuracy of the pointing device. The user was required to select the red innermost region of the target. The number

of incorrect selections of the target was calculated by comparing the number of times the pointing device's selection button was pressed with the number of times the target was selected. The data were analysed using a t-test and were found to be 97% significant ($t_{<0.05,6} = 2.18$), and the result is summarised in table 4.4.

	Twiddler	Mouse
Cursor Accuracy	8.16 ± 5.98	2.83 ± 2.23

Table 4.4: Cursor accuracy (targets hit).

The results show that when selecting a target with a radius of 50 pixels, the Twiddler produces approximately 3 times the number of errors when compared to the mouse. The standard deviation of the results from the twiddler are high enough to cross over with the results obtained from the mouse and this is reflected in the t-test which produces a 97% significant result.

From this significance figure it is possible that some users will be able to obtain comparable results with both devices. It is therefore recommended that these results are carefully interpreted. The result does not mean that the Twiddler is less accurate when compared with the mouse but the results do *suggest* that there is a difference between these devices. In order to find out which device performs better more testing would be needed to reduce the amount of error obtained in the test.

4.9.5 Experiment two: cursor overrun

Upon completion of the data analysis of the second experiment, it was observed that an extra piece of important information, not initially considered in the experiments, existed in the data. It was possible to determine how far the user had moved the input device in order to select a certain target, and therefore it was possible to gain an insight into how effective the devices were.

The ideal case for any device would result in a user moving the cursor in a perfectly straight line between the two targets. In the real world the inaccu-

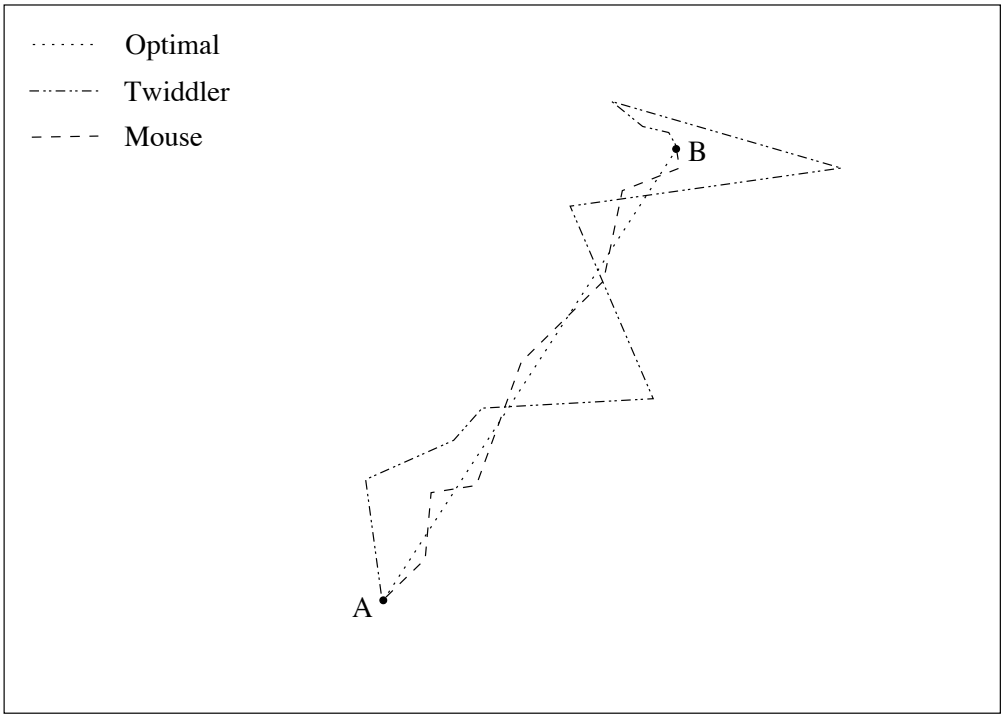


Figure 4.5: Traveling patterns between two targets.

racies of the pointing devices and the individual person's motor control result in deviation of the cursor from the optimum path. With this knowledge it is possible to determine the optimum distance between two targets and subtract the actual distance traveled. This can be seen graphically in figure 4.5.

This results in an *overrun* value which can be used to determine how accurate the device is. The overrun value is the combination of the amount of error in the user's hand/eye coordination and the sensitivity and accuracy of the pointing device. The data were analysed using a t-test and were found to be significant at the 1% level ($t_{<0.005,6} = 4.12$). The results can be seen in table 4.5.

	Twiddler	Mouse
Cursor Overrun	852.7 ± 448.4	97.7 ± 34.2

Table 4.5: Cursor overrun (pixels).

The results were not originally envisaged as part of the experiment, but the analysis of the data revealed that the information was just as important as the

accuracy and speed results. The amount of overrun seems to indicate how much effort was needed to control the pointing device. The results show that there is a significant difference in the amount of overrun needed in order to select a target. In order to locate the cursor over a target and select it, the subjects in this experiment required nearly 9 times the amount of distance to be traveled with the Twiddler when compared to the mouse.

The author suspects that the overrun contributes to the fatigue and annoyance experienced with the Twiddler device. It is also speculated that the overrun values are directly linked to the speed and accuracy of the Twiddler's pointing sensor, and a decrease in the overrun values would increase the speed and accuracy of the device.

4.9.6 Experiment three: cursor speed

The pixel movement times were calculated the same way as in experiment two. The data were analysed using a t-test and the results were found to be significant at the 1% level (see table 4.6).

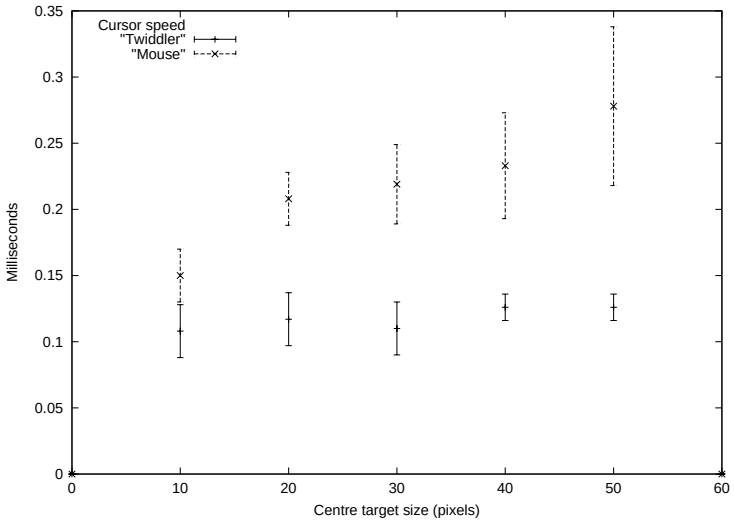


Figure 4.6: Speed vs target size using the Twiddler and mouse.

The results in figure 4.6 suggest that, with a mouse, as the target size decreases, the speed at which the user can control the mouse also decreases. This

observations is in agreement with Fitts law [58] which states that *"the time to acquire a target is a function of the distance to and size of the target"*. In contrast, the size of the target does not have an effect on the speed of the Twiddler as the speed remains almost constant regardless of the target size. These results have some profound implications for the use of the Twiddler.

Target size	50	40	30	20	10
$t_{<0.001,6}$	37.2	26.2	13.3	11.1	5.1

Table 4.6: The t values for speed vs target size.

The results for the second experiment show that the Twiddler is significantly slower to use than the mouse, probably because of the overrun discussed above. The author thought that the Twiddler would exhibit a reduction in speed as the target size was reduced, but as is shown this was not the case.

4.9.7 Experiment three: cursor accuracy

The accuracy of the third experiment was calculated in the same was as in experiment two. The data were analysed using a t-test and the analysis revealed that the results were statistically significant at the 5% level (see table 4.7), except for the data for the 40-pixel target which was 86% accurate. This means that the 40-pixel target results may not be as accurate as the other results, and further experiments would be required to reduce the amount of error obtained.

The results in figure 4.7 show that the Twiddler is consistently less accurate than the mouse when the target sizes are varied. The largest difference can be seen between the 30 and 20 pixel points. Here the mouse still has an accuracy in the 80 percent region while the Twiddler falls to the 30 percent mark. At the 10 pixel point the accuracy of the mouse falls to roughly the same accuracy as the Twiddler.

The data for this tests were studied and it was found that user number 1, for some reason, has a very low accuracy for this test but their speed and overrun results are similar to the other users. The user was consulted to check

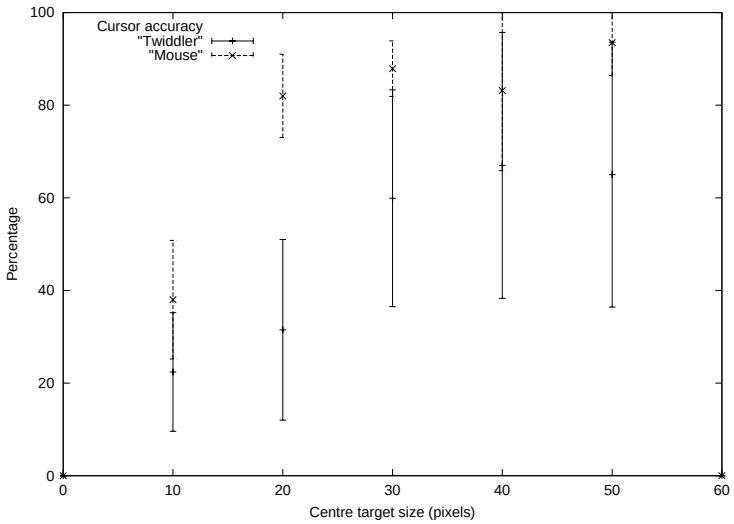


Figure 4.7: Accuracy vs target size using the Twiddler and mouse.

Target size	50	40	
value of t	$t_{<0.05,6} = 2.4$	$t_{<0.15,6} = 1.4$	
	30	20	10
	$t_{<0.025,6} = 2.9$	$t_{<0.01,6} = 6.3$	$t_{<0.025,6} = 3.0$

Table 4.7: The t values for accuracy vs target size.

that they understood the methods involved in the experiment, which they said they did. The data obtained were examined to ascertain whether the user had any trouble clicking on the target, but the data showed that the person rarely missed the target. The reason for the very low accuracy can be explained by the user not clicking on the red circle as required, but on the white or black parts of the target. This can only be achieved by the user having trouble locating the pointer over the desired target with the Twiddler, by not paying attention to the cursor location when pressing the Twiddler's selection button or by the user having some kind of hand/eye coordination problem. The author suspects that the first explanation is correct, but the errors obtained during these tests is larger than would have been expected. If we examine the data (shown in table 4.8) we find that there are marked individual differences in the results.

For example, the 30 pixel target has an average of 59.9%, but the span of this result is between 86% and 33%. It is important to recognise that these artifacts

highlight the wide range of individual differences, and that further tests would need to be undertaken with a greater sample of subjects to reduce the standard deviation. However, the author wishes to point out that even with these results, the difference between the Twiddler and the mouse are statistically significant in defining the operational parameters of the device.

User	50	40	30	20	10
1	11	13	33	18	5.2
2	75	93	76	40	40
3	73	80	37.5	11.7	11.7
4	93	80	80	53	31.25
5	60	60	46.6	13.3	26.6
6	80	80	86	53	20
Mean	65	67	59.9	31.5	22.4
S.D.	28.6	28.7	23.4	19.5	12.8

Table 4.8: Experiment three: cursor accuracy data.

The results show that between the 30 and 20 pixel point mark there is a significant decrease in target accuracy with both devices. The Twiddler is less accurate than the mouse at these levels and any target which is smaller than 30 pixels in radius will probably be harder to select. This will require more button clicks and will probably increase the amount of frustration experienced by a user.

4.9.8 Experiment three: cursor overrun

The overrun of the third experiment was calculated in the same was as in experiment two. The data were analysed using a t-test and all the results were found to be significant at the 1% level (see table 4.9).

Target size	50	40	30	20	10
$t_{<0.001,6}$	4.75	4.84	5.27	3.73	7.69

Table 4.9: The t values for overrun vs target size.

The results in figure 4.8 show that the Twiddler requires a significantly greater amount of movement in order to select a particular target. These results

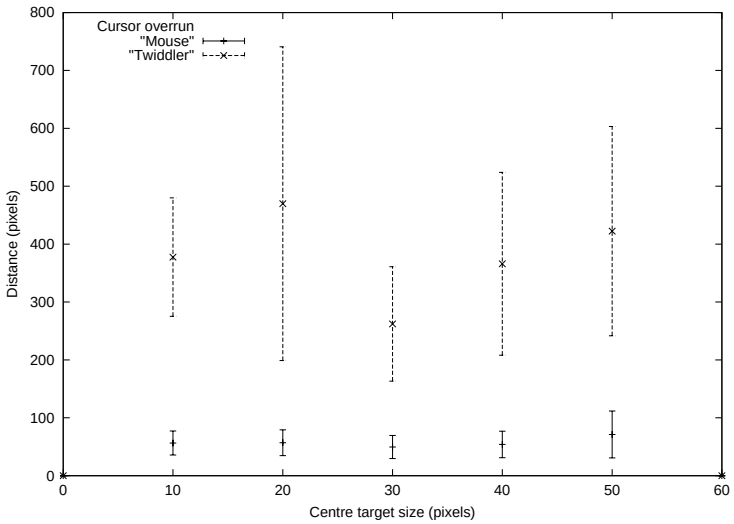


Figure 4.8: Overrun vs target size using Twiddler and mouse.

also imply that for the Twiddler there is an increase in effort required to select targets between 30 and 20 pixels in radii. This was not entirely unexpected as the test subjects all complained of fatigue and annoyance (see section 4.8) when using the Twiddler to select the smaller targets.

4.9.9 Experiment four: cursor speed

The data obtained from the tests were analysed and the results from the experiment can be seen in figures 4.9 and 4.10.

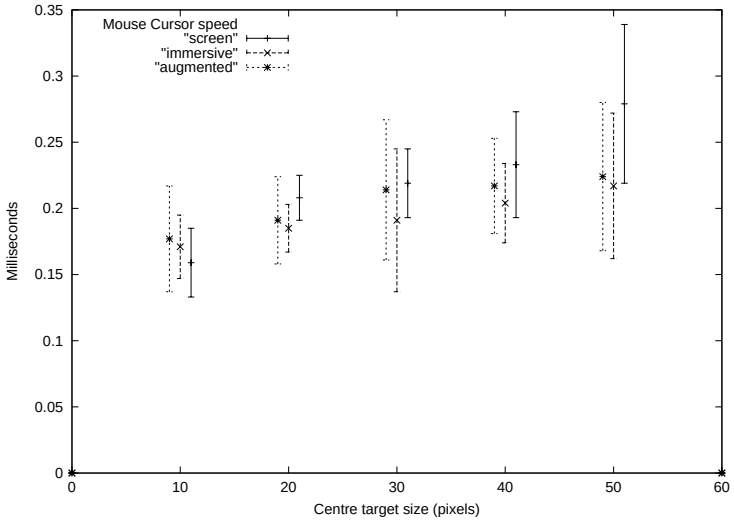


Figure 4.9: Speed vs target size using a mouse.

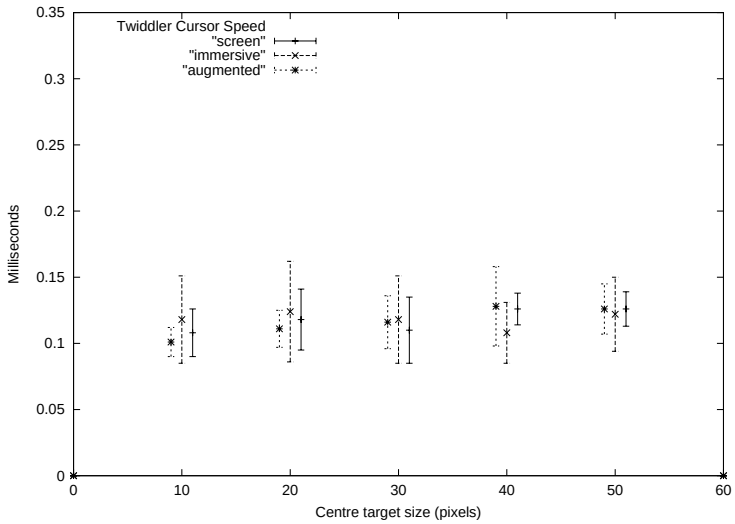


Figure 4.10: Speed vs target size using the Twiddler.

The results[6] show that the normal CRT screen, the immersive head-mounted display and the augmented head-mounted display have no little effect on the speed of use of of the pointing device. As the target size reduces, all three display systems show a small, comparable decrease in speed when the mouse is used. With the Twiddler there appears to be no difference in speed when comparing the target size against the various display systems.

4.9.10 Experiment four: cursor accuracy

The data obtained from the test were analysed and the results can be seen in figures 4.11 and 4.12. The results again show that again there is little difference in accuracy when comparing a normal CRT screen, an augmented or an immersive head-mounted display.

[6]The results are based on target sizes in increments of 10; the graph shows the results shifted either side of the 10 marks to make the data easier to read.

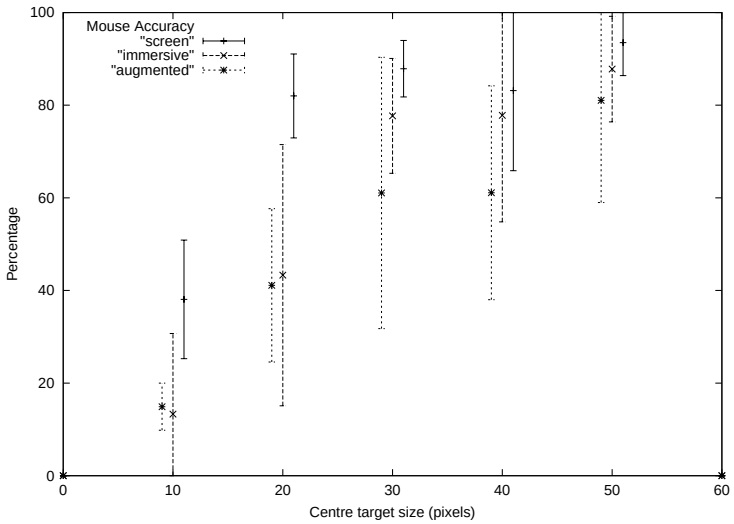


Figure 4.11: Cursor accuracy vs target size using a mouse.

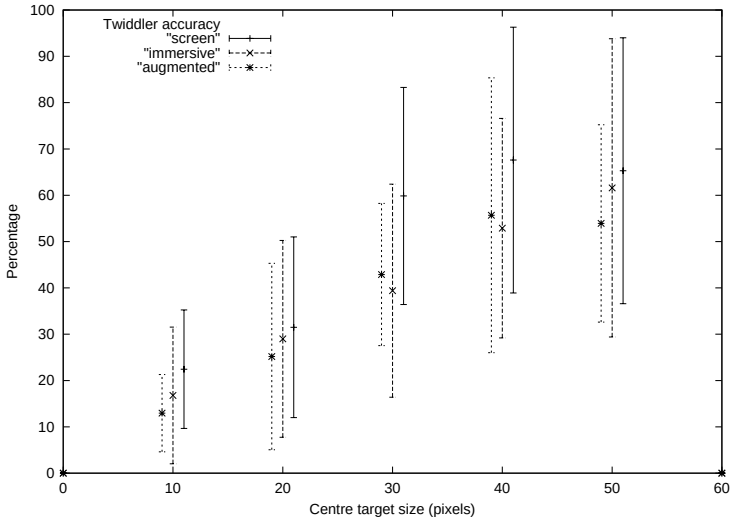


Figure 4.12: Cursor accuracy vs target size using a Twiddler.

However, the results show that with a mouse there is a difference in accuracy between the CRT screen and the immersive/augmented HMD when the target size is smaller than 30 pixels. It was originally envisaged that the combination of an immersive/augmented HMD would reduce the accuracy of the Twiddler but when the data were analysed there were few differences in accuracy between the three display types.

4.9.11 Experiment four: cursor overrun

The data obtained from the test were analysed and the results from the experiment can be seen in figures 4.13 and 4.14.

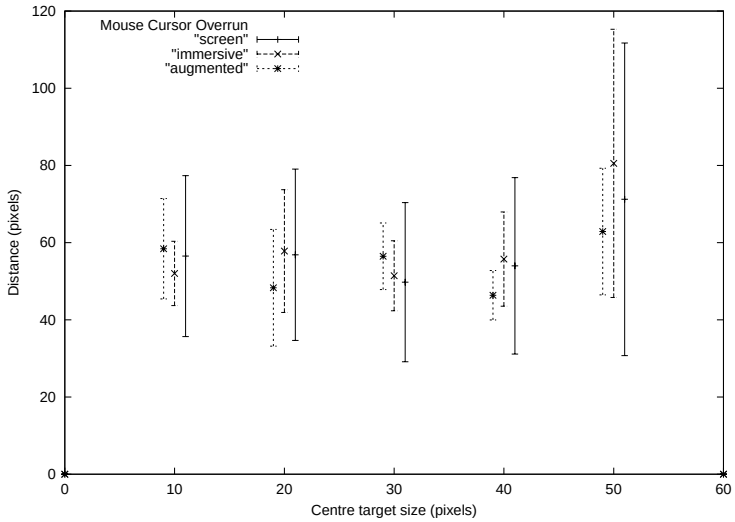


Figure 4.13: Cursor overrun vs target size using a mouse.

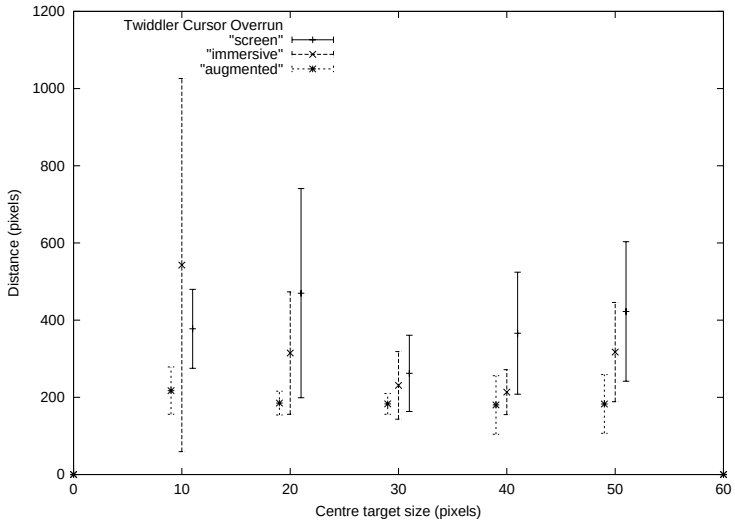


Figure 4.14: Cursor overrun vs target size using a Twiddler.

The results show that there is little difference in the amount of overrun when comparing the CRT screen, immersive and augmented HMD with the mouse. But when the results for the Twiddler are analysed, an unexpected result appears. While the CRT screen and the immersive HMD follow a similar pattern,

Target size	50	40	
value of t	$t_{<0.005,6} = 4.3$	$t_{<0.2,6} = 1.1$	
	30	20	10
	$t_{<0.005,6} = 4.4$	$t_{<0.005,6} = 10.3$	$t_{<0.005,6} = 12.9$

Table 4.10: The t values for the cursor overrun vs target size when comparing the immersive and augmented HMD with the Twiddler.

the augmented display exhibits a consistently greater amount of overrun for all the targets. In most cases the results are statistically significant at the 5% level. This suggests that using the augmented HMD may reduce the amount of overrun, and therefore the physical effort required to manipulate the Twiddler keyboard. These results were backed up by four out of the six subjects commenting that the augmented display seemed easier to use with the Twiddler when compared to the CRT screen or the immersive HMD.

4.9.12 Experiment five: cursor speed

The data obtained from the experiment were analysed and the results can be seen in figures 4.15 and 4.16.

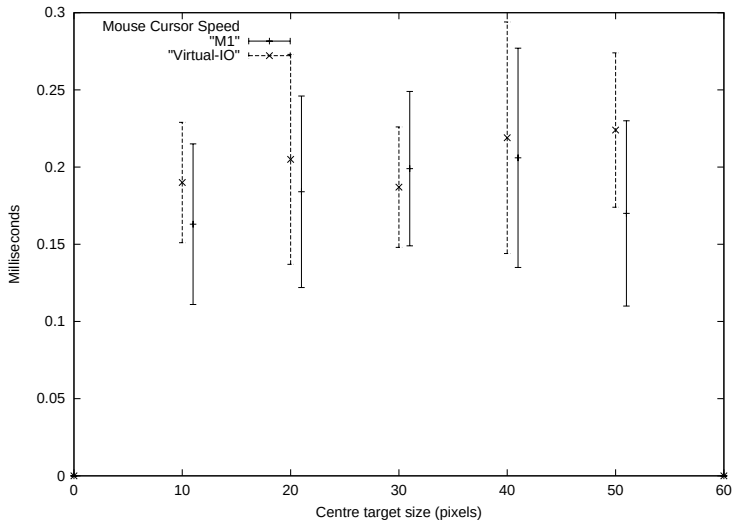


Figure 4.15: Cursor speed vs target size using a mouse.

The results show that there is no significant speed difference when using the monoscopic Virtual-IO or the M1 head-mounted displays. In this situation the

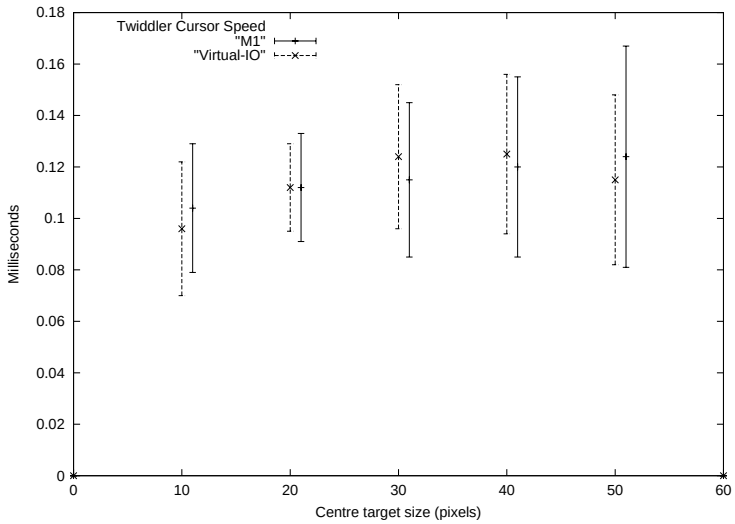


Figure 4.16: Cursor speed vs target size using a Twiddler.

cursor speed does not depend on the visible screen resolution in a monocular display system. The author was expecting the immersive display to exhibit an increase in speed due to the higher physical resolution of the immersive display device, but this was not the case.

4.9.13 Experiment five: cursor accuracy

The data obtained from the experiment were analysed and the results can be seen in figures 4.17 and 4.18.

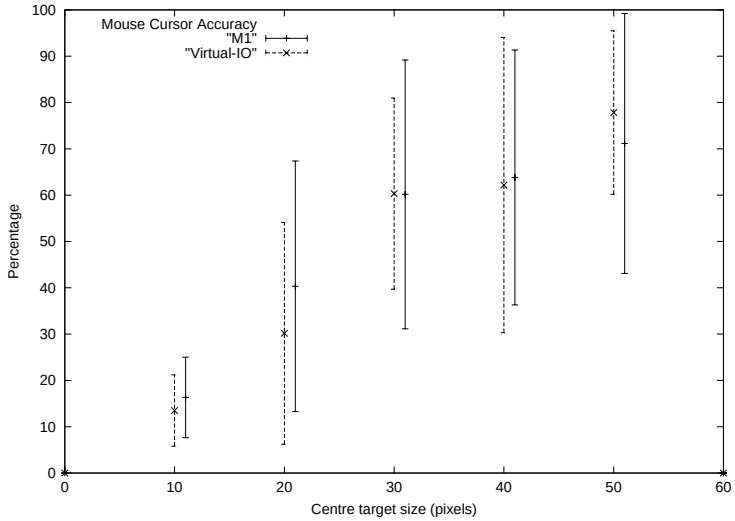


Figure 4.17: Cursor accuracy vs target size using a mouse.

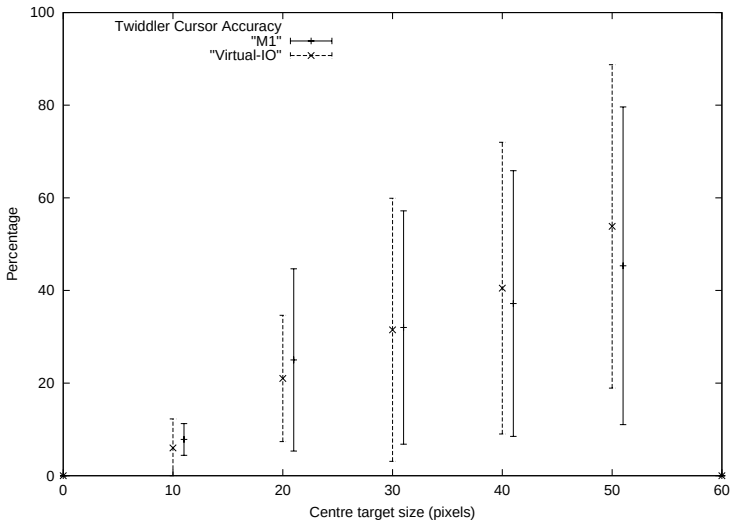


Figure 4.18: Cursor accuracy vs target size using a Twiddler.

The results show that there is no significant difference between the augmented and immersive HMDs in the accuracy tests. The author was expecting a noticeable difference in accuracy due to the M1's higher perceptible resolution, but both displays perform similarly.

4.9.14 Experiment five: cursor overrun

The data obtained from the test were analysed and the results from the experiment can be seen in figures 4.19 and 4.20.

The results show that there is no difference in the amount of overrun between the augmented and the immersive HMDs. In experiment four (page 89) the amount of overrun was significantly lower for the augmented HMD; this experiment shows that the advantage is lost when a monocular display is used. Although the higher perceptible resolution of the M1 provides a smaller mean and standard deviation when compared to the Virtual-IO glasses, the results are not significantly different.

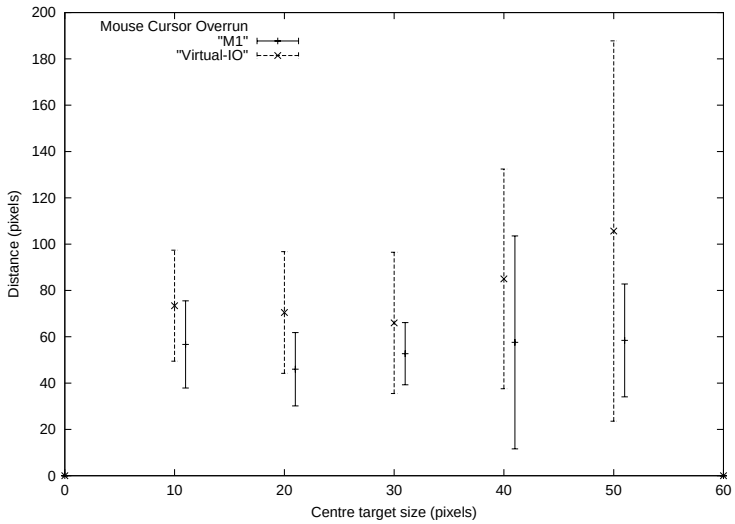


Figure 4.19: Cursor overrun vs target size using a mouse.

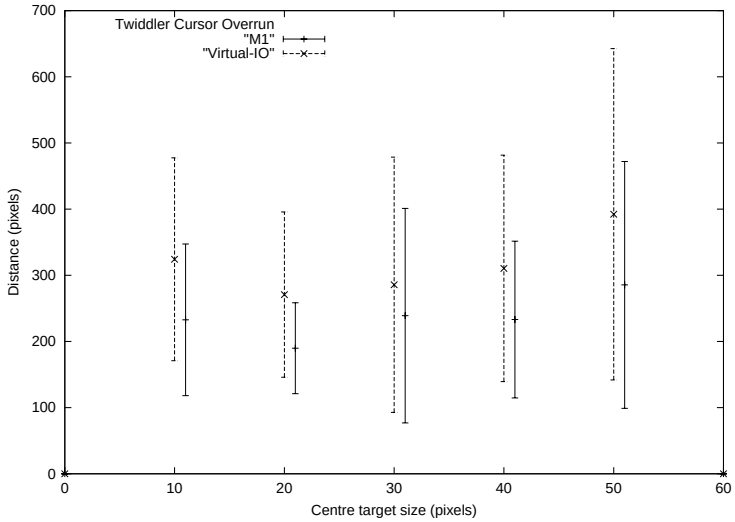


Figure 4.20: Cursor overrun vs target size using a Twiddler.

4.10 Chapter summary

The first few experiments provide the reader with an analysis of the average speeds and accuracy obtainable with the Twiddler input device for a particular group of people. This information is used by the author to design a graphical user interface for the Sulawesi architecture, using the Twiddler as an input device; see chapter 6. With a well-designed interface the Twiddler can be exploited

to provide an efficient way of controlling the user interface.

The results for experiment one suggest that the user of the Twiddler has a maximum speed and accuracy which is significantly lower than a QWERTY keyboard. Inexperienced subjects quickly approach the same speeds as that of an experienced user, suggesting that the device has a maximum typing rate of approximately 4 times slower than the QWERTY keyboard. The accuracy of the Twiddler is dependent on the experience of the user and, with training, the accuracy can be reduced to a comparable rate of a QWERTY keyboard.

The speed of text entry on the Twiddler will have an obvious impact on the types of application that can be used. For example, a word processor might be acceptable if a short note or letter were to be typed, but for a long, complex document it may be unsuitable. The test paragraphs are relatively small in comparison to this document, therefore it would be up to the designer to decide whether the wearable system would be used for the entry of large amounts of text. If so then another type of input mechanism, such as an arm mounted keyboard,or speech recognition may be more suitable.

The maximum speed achievable seems to be due to having to press two or more keys to obtain a single character. Chord keyboards such as the Microwriter [2][7] use chording to offer a significant speed increase by assigning a chord, a syllable or a whole word to a certain key or keys. The main difference between the Microwriter and the Twiddler is that the former was designed to be used with both hands and was fairly efficient. The Twiddler was designed to be used in one hand only, so any advantages that were gained by using two hands to enter chords have been lost.

This assignment of words or macros to the chords on the Twiddler has been applied by many, and claims of an input rate of 50+ words a minute is mentioned in [10]. On the evidence of this evaluation, this can only be achieved by assigning macros to each key. It is speculated that this assignment would indeed provide

[7] Also see `http://www.tifaq.com/keyboards/other-keyboards.html`

an increase in word speed, but it is also likely that an increase in the amount of time taken to learn the new chording pattern would be noticed.

The results for the second experiment suggest that the pointing device on the Twiddler is far from ideal for manipulating the smaller widgets, such as the radio buttons or check-boxes that proliferate in current desktop environments. The author speculates that a combination of the poor position sensing device used in the Twiddler and the non-linear behavior of that device contribute to the slow speeds, low accuracy and high overruns observed. Also, unless the amount of overrun is decreased, the amount of effort and frustration experienced will put many people off from using the Twiddler. Indeed this appears to have been noted by HandyKey who have replaced the tilt sensor for mouse control in the original Twiddler with a pointing stick (as found on IBM laptops) in the Twiddler 2.

The results from the third test show that as the target size gets smaller; the Twiddler's cursor manipulation speed slows down, the accuracy decreases and the amount of overrun required to select a target increases. Again, this reflects the coarseness of the sensor in the Twiddler. The results (seen in figure 4.7) also show that as the radius for the target size decreases to 30 pixels the amount of overrun for the Twiddler decreases, but between 30 and 20 pixels the amount of overrun rises sharply. This result implies that the optimal target size for the Twiddler would be somewhere between 30 and 20 pixels in radius.

The second and third experiments indicate that the pointing device on the Twiddler is adequate for a task that requires a cursor to be placed and clicked over an area approximately 30 pixels in radius. The results show that this is, in some sense, the optimal size for the targets: anything smaller requires a significant increase in effort and patience by the user, anything larger than 30 pixels will take up valuable screen real estate on a 640×480 display. The author speculates that the ratio between the target size and screen size will remain constant if the screen resolution is increased, as the main influence on performance is the

Twiddler device rather than screen resolution. These observations indicate the poor performance of the pointing device in the Twiddler, and it is expected that in a mobile environment the amount of control achievable with the pointing device will decrease rapidly. From these results the author suggests that the use of pointing tasks in a wearable user interface should only be undertaken when necessary. The use of non-graphical windowing, menu and icon systems should be explored before relying on traditional pointing tasks.

The results from the fourth experiment show that the CRT and immersive HMD do not have a large effect on the results obtained from the Twiddler. However, the augmented HMD may reduce the the amount of overrun experienced when using the Twiddler. As the author speculated earlier in this chapter, the speed and accuracy of the device is related to the amount of overrun experienced. This result may justify the use of the augmented HMD to reduce the amount of overrun experienced.

Another artifact from this display was noticed, which may or may not have an affect on future design issues. It was observed by several of the subjects that when the augmented display was used with either pointing device, a disorienting illusion occurred. If the person moved their head at the same speed and in the same direction as the cursor, they would think that the cursor was moving and therefore they must be moving the manipulation device. But if the head movement suddenly stopped, it was revealed that the cursor had not moved on the screen at all. This illusion seemed to occur when a static image was viewed on the augmented HMD and when the real world image, seen through the HMD, was moving in the same direction as the user wanted to move the cursor. The author cannot explain why this happens but suspects these events trick the brain into thinking that the hand is moving the pointing device, even though the hand is not moving.

The results from the fifth experiment show that there is no difference in the augmented and immersive monocular HMDs tested. Again, this was unex-

pected: in the fourth experiment the augmented display showed a decrease in the amount of overrun required to perform a similar task, but this advantage is lost when the display is over one eye only. The accuracies with the monocular devices are almost identical even though the augmented HMD has a lower perceivable resolution. This suggests that either the type of display does not affect the performance or that the augmented display performs better for a given resolution.

The fourth and fifth experiments indicate that the resolution of the head-mounted displays do not have a significant effect on the speed or accuracy of the system. The fourth experiment shows that there is an a reduction in the amount of overrun when using an augmented HMD. This reduction in overrun would allow a user to either operate a novel input device such as the Twiddler for a longer period of time, or to operate the device for the same period of time with a reduction in fatigue.

The slow performance and crude pointing device of the Twiddler, highlighted in this research, do not make it a good solution for a text/input device. Unfortunately due to the lack of any other commercially alternative mobile interface device the use of the Twiddler is persisted with.

4.10.1 Guidelines

From the work in this chapter the author has drawn up a set of guidelines for the various interaction devices.

- The first experiment indicates that if a small amount of text needs to be entered, then the Twiddler device will be adequate even though it is significantly slower than a QWERTY keyboard.

- The results from the second and third experiment indicate that the use of small check boxes and radio buttons should be discouraged as the precision obtained with the Twiddler is not sufficient to accurately select these types

of widgets. The results suggest that a target size of between 20-30 pixels provides a good trade off between screen real-estate and selection accuracy.

- The results from the fourth experiment show that the use of a stereo augmented HMD may reduce the amount of fatigue a user experiences with the Twiddler. Also an augmented HMD has the advantage of being able to overlay the images with the real world rather than obscuring the users vision totally.

- The difference between an augmented and immersive monoscopic HMD has little impact on performance. The results in experiment five show that the type of HMD does not significantly affect the ability to control the Twiddler or manipulate the widgets on the screen. Because of this the author suggests the use of an augmented monocular HMD as it will allow a person to focus on the real world when they are required to.

Chapter 5

Sulawesi, A Contextual User Interface Framework

5.1 Why was Sulawesi designed?

The goal of this research is to investigate user interfaces for wearable/mobile computing. As the previous chapter shows, some of the current interaction devices that have been tested and being used in the real world for user interface manipulation are not ideal for a mobile user interface. While the HMD systems tested appear not to significantly affect the speed and accuracy of graphical manipulations, it is clear that the speed and accuracy of the Twiddler is many times slower than a normal QWERTY keyboard and mouse. While current user interface systems can be modified and designed to allow for these differences in the interface devices, the ability to adapt these to include alternative forms of interaction is still a problem. Here the author has attempted to fuse together the knowledge gained from the user trails in chapter 4 with a system which can provide contextual and agent-based functionality.

The use of contextual and agent-based systems have been proposed by many people [31, 42] as a potential solution to some of the problems with interaction in a mobile environment, but there have been few attempts at tackling the user

interface issues involved with wearables, let alone designing and constructing a single framework which can encompass multimodal, agent-based and contextual systems. Due to the lack of a system to enable research into contextual wearable systems, the author has chosen to look at the general architectural and user interface issues involved in the design and implementation of a wearable software framework. The goal is to provide a platform which will allow others to research and develop contextual user interfaces for alternative situations such as ubiquitous, contextual and mobile environments. In these situations it is fairly obvious that the use of any input devices is fairly restricted, but the use of voice commands is appealing to explore.

There have been a number of different input/output modalities developed by different research groups all over the world. The idea behind the Sulawesi framework is to provide a "common" integration platform which will be flexible enough to encompass a wide variety of input devices, separating the application development from the input mechanism. The framework provides a set of communication primitives to provide diversity for a wide range of applications and devices. It also provides mechanisms to allow applications to communicate with each other via a broadcasting messaging system.

The system allows applications to query an input channel to determine the current interpretation of the environment. An example of this would be a location application which periodically asks a GPS channel "where am I?". The location application would not need to know what device it is asking for the information, so it could just as easily ask an infra-red beacon device or a vision based positioning system . This method of abstraction can be used to compare the accuracy of one device against another, or to gain an enhanced state of knowledge about the environment.

Another example is for a user to point at somebody while saying "tell me who that is". This is clearly a complex task which can be split into a few discrete parts. The first is to determine what the user has said at what point in

time (because the sentence would only be relevant for a certain period in time), finding out who the user is looking at that time (via a visual input device such as a camera), attempting to recognise the person in the image, and to generate a spoken output of the person's name.

5.2 The Sulawesi architecture and concepts

The Sulawesi architecture can be seen here in figure 5.1. In this chapter the system is documented using the UML [49] notation (except for figure 5.1).

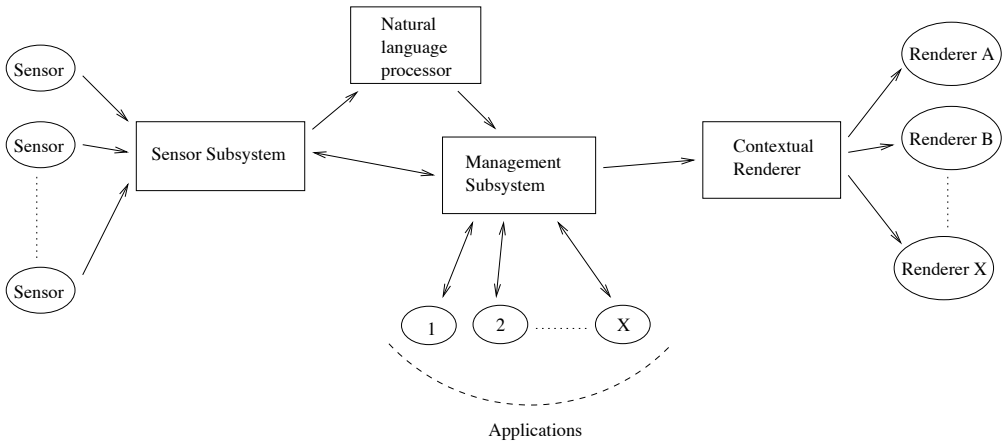


Figure 5.1: Sulawesi architecture.

Sulawesi has been designed to allow individual subsystems to be developed independently of each other. A well defined interface for each subsystem and its implementation is discussed in detail within this chapter.

As with any system under development, these interfaces were altered slightly as the design matured; even so the same underlying definition of what each subsystem should do and how it should communicate with other subsystems has been unchanged throughout the development.

Note: Although the Natural Language Processor was originally thought of as a separate subsystem, it has been incorporated into the Management Subsystem and is discussed in the following section.

5.2.1 Information abstraction layers

The separation of the input modalities from the services provides some problems in resource discovery and integration. If an input resource suddenly disappears, any services that are dependent on the resource may behave unpredictably. In order to try and solve this problem an intermediate service has been defined. This intermediate layer receives raw data from the sensor and translates it into an abstract form which an application can receive. Various tasks such as logging the position of the user or how often they walked about can be determined, but the raw data itself may be of limited use. A service can have access to the raw sensor data if needed, but an abstract form of the data may be more useful to the application depending upon the situation.

The use of the Information abstraction layer (I.A.L.) shown in figure 5.2 enables data sources to appear and disappear without an applications knowledge.

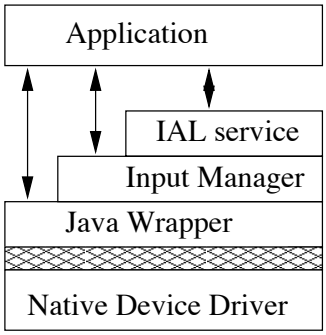


Figure 5.2: The information abstraction layer (I.A.L).

The I.A.L. can also be used to provided a mechanism whereby data from the users environment can be fed back, altering Sulawesi's decisions and reactions. For example, if a user is driving it would not be advisable to display information on a head-mounted display or a portable screen as this would distract the user. Using movement sensor, video camera, GPS receivers, *etc*, it is possible for a *movement I.A.L.* to directly effect Sulawesi by requesting that all visual outputs are to be redirected to a speech rendition where possible. When the user has stopped travelling, the *movement I.A.L.* can cancel the visual→aural

redirection. Another example is of a *speech I.A.L.* which listens to an audio input stream from a microphone; the detection of somebody speaking triggers the *speech I.A.L.* which then requests that the *renderer subsystem* should pause all speech renditions until the has user stopped speaking.

5.2.2 Semi-natural language decoding

When humans recognise speech they do not understand every word in a sentence, sometimes words are mis-heard or a distraction prevents the whole sentence from being detected. Even so, a human can infer what has been said from the other words in a sentence. While this is not always successful, in most cases it is satisfactory for the understanding of a conversation. This type of sentence decoding has been termed *semi-natural* language processing and has been implemented using a rule-based system.

The core of the user interface is based around a string matching algorithm, this converts a understandable sentence into a command stream from which two pieces of information are extracted: the service to invoke; and how the output should be rendered.

The example below explains how the Sulawesi system converts human understandable sentences into commands:

COULD YOU SHOW ME WHAT THE TIME IS

I WOULD LIKE YOU TO TELL ME THE TIME

It can be argued that in practice these sentences result in similar information being relayed to a user. The request is for the interpretation of the time to be sent to an appropriate output channel, the result is the user receiving the knowledge of what the time is through that channel. Closer inspection reveals that almost all the data in the sentences can be thrown away and the request can still be inferred from the following verb and object.

SHOW TIME

TELL TIME

In the example above there has been a reduction in the number of words in the sentences. The author argues that very little of the information content has been lost from the sentence, and that it is still possible to infer the meaning of the sentence. If no verb is specified for the display of information, such as "what is the time", the system falls back to a default rendition of the time.

Sulawesi has been designed to allow sentences to be processed and interpreted in this way. The semi-natural language processing is achieved through an autonomously generated look-up table of service names and a language transformation table. The unique service names provide a simple mechanism with which to look-up a service such as *"time"* within a sentence.

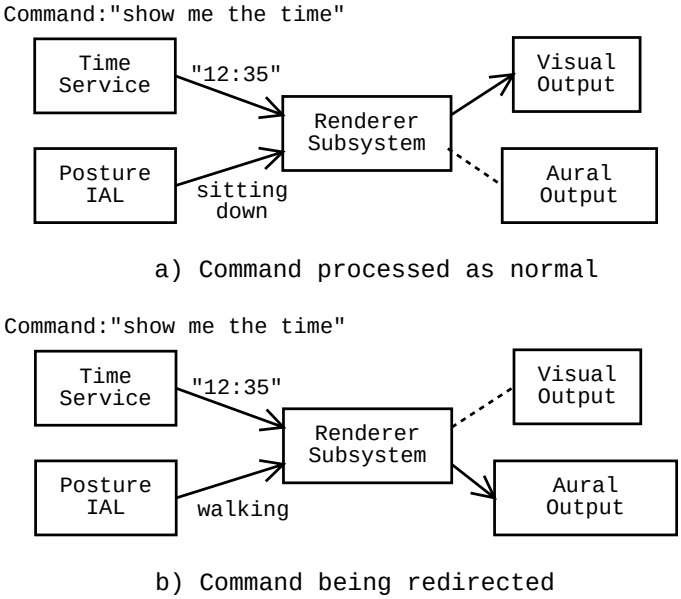


Figure 5.3: Commands being processed.

5.2.3 Renderer redirection

For any system to produce a response there must be some form of output which can be understood by a user. As mentioned in the previous section there are certain situations when it may be desirable to redirect one type of renderer to a different type. The *Renderer Subsystem* is invoked by a service requesting the rendition of a piece of information via a certain output channel. When the

Renderer Subsystem is initialised it opens the renderer look-up table (a file on the disk) and loads it into an internal hash table. This hash table can then be queried and manipulated by an I.A.L to redirect a rendition.

For example: if the user has asked to be shown a piece of information, implying a visual output, Sulawesi can use a *posture I.A.L.* (described in section 6.6) to determine whether the requested rendition is suitable for the current context of the user.

As can be seen in figure 5.3, if the *posture I.A.L.* decides that the requested renderer is suitable then the request is passed through to the requested renderer. But if the *posture I.A.L* indicates that the requested rendition type is inappropriate for the context of the user, then it asks the renderer subsystem to redirect the output to an alternative rendition type.

5.3 The architecture of the Sulawesi framework

In order to create a system where sensors, applications (services) and renderers can be implemented separately there needs to be a uniform interface between each of the components. A top down view of how these subsystems function and communicate with the rest of the system are broken down in this section. This provides a detailed description of the design using the UML [49] notation, and allows an expandable interface to be determined and documented.

5.3.1 The sensor subsystem

The foundation of any contextual system lies in the use of sensors to gather information from an environment. In a multi-sensor architecture it would be impractical to hard code all possible sensors into an application.

From a software engineering point of view, a generic sensor broker is used to communicate information from the sensors to the rest of the system and this has been called the Sensor Subsystem (seen in figure 5.1). The sensors commu-

nicate information through well defined interfaces, and the *Sensor Subsystem* is responsible for providing a common point for an application to establish uni- or bi-directional communication with multiple sensors.

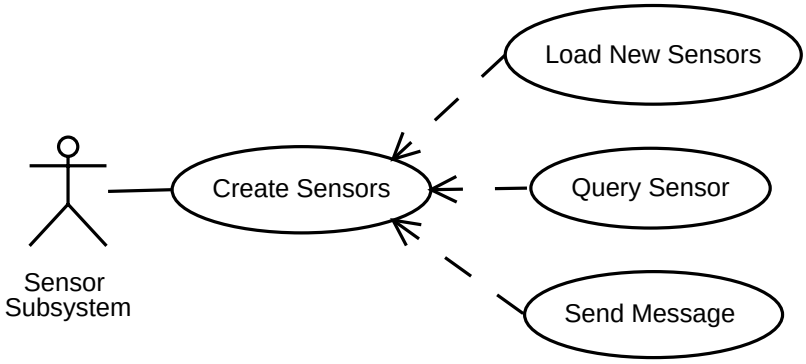


Figure 5.4: The sensor subsystem.

The *Sensor Subsystem* is responsible for initialising the sensors when the system is first started. When the sensors are initialised they register themselves with the *Sensor Subsystem*. This provides the subsystem with knowledge of which sensors are available. Also, provisions have been made to dynamically load new sensors into Sulawesi as and when they become available.

The types of sensors that can be realised are generally classified into two types. The first gathers asynchronous information by a triggering event in the environment, such as a door switch. The second type of sensor gathers periodic information at certain time intervals, such as a temperature sensor gathering data every second. The architecture allows both types of sensors to transmit their information to other parts of the system via a message-passing system.

Provisions have also been made to allow a sensor to be queried for its current perception by an application, or the framework itself. This allows asynchronous sensors to be polled by an application if it is necessary.

From this information we can create a UML Use Case Diagram for the *Sensor Subsystem* which can be seen in figure 5.4. As can be seen there are four Use Cases with the UML Actor in this figure representing the Sensor Subsystem.

The modular nature of the system means that, at startup, Sulawesi has no

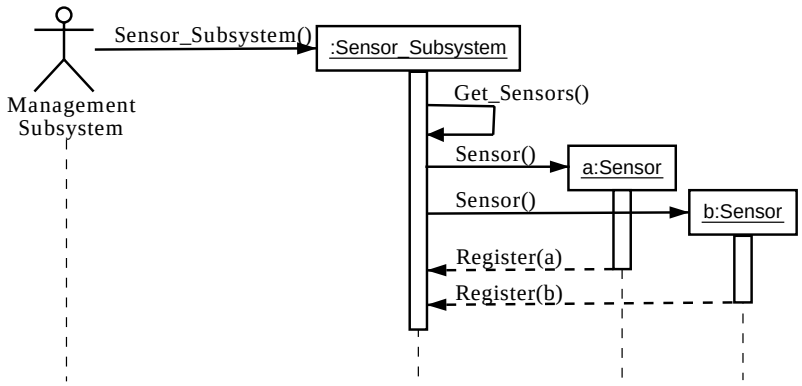


Figure 5.5: The sensor subsystem at start-up.

knowledge of any sensors, it has no knowledge of how many there are, or what they are called. The only way to dynamically load these sensors is to know where they are (which directory), to walk this directory and to load each sensor in turn.

Create sensors

When the system is first started the *Sensor Subsystem* is constructed by the *Management Subsystem* (see section 5.3.3). It then identifies which sensors are available by calling the `Get_Sensors()` method and constructs the sensors. On construction the sensors acknowledge that they are available by registering their presence with the *Sensor Subsystem*. The UML sequence diagram for the creation of the sensors can be seen in figure 5.5.

Load new sensors

As mentioned in the introduction of this section, in certain situations it is desirable to load dynamically new sensors when they become available. This is controlled by the *Management Subsystem* which sends a message to the *Sensor Subsystem*. The `Get_Sensors()` method kicks off the sensor detection stage (seen in figure 5.6). The re-detection of sensors works by comparing the list of currently registered sensors with the ones that are available on the file-system.

If a new sensor has been detected it is constructed and it then registers with the *Sensor Subsystem*.

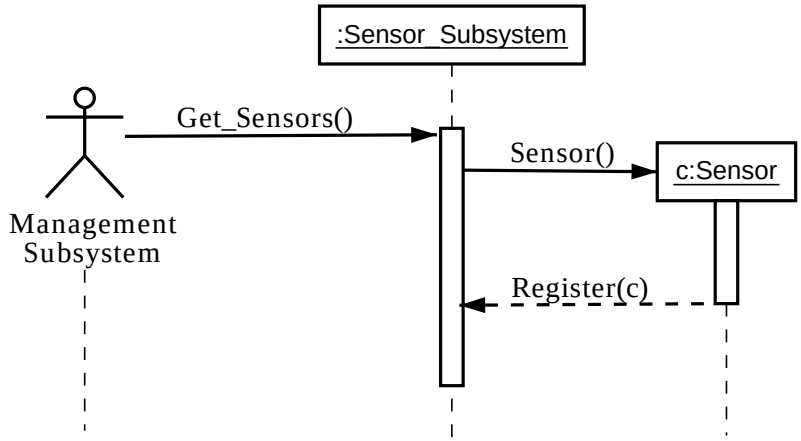


Figure 5.6: The sensor subsystem when re-detecting sensors.

Query sensor

The architecture allows an application to communicate directly with a sensor if so desired. This is achieved by asking for a connection to a particular sensor, the *Sensor Subsystem* then returns an object reference to the sensor. The application then communicates directly with the sensor through the *Management Subsystem*. The sequence diagram in figure 5.7 illustrates this concept.

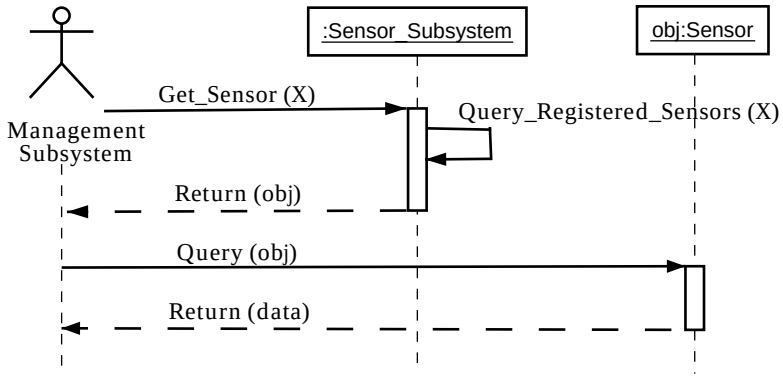


Figure 5.7: Querying a sensor.

Send message

In order to allow the sensors to send data to the *Sensor Subsystem* (and on to the *Management Subsystem*) a mechanism for passing data was needed. The nature of the system meant that almost all message needed to be copied to several objects, rather than to a single object. A traditional WIMP event-loop could have been used, and each object that wished to receive messages could subscribe to an event stream. Unfortunately, the WIMP event-loop has several drawbacks. If an object or piece of code hangs, it could be possible to receive all the messages without pushing them back onto the message stream. This has the effect of hanging the entire system as no messages are passed to any other objects. Also, the throughput achievable with the WIMP event-loop is not as high as a system based around a shared memory messaging pool, but this high throughput system is generally only needed where bottlenecks need to be reduced and performance needs to be increased.

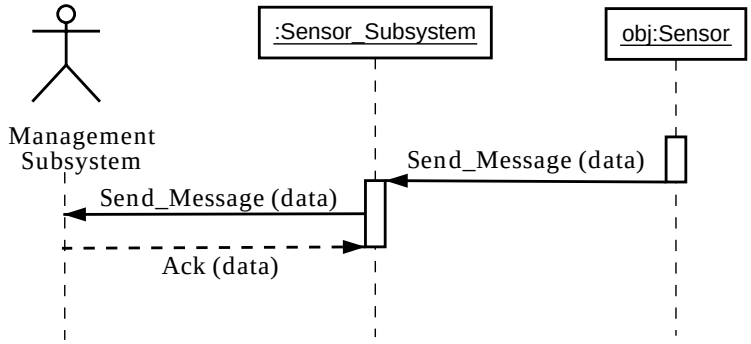


Figure 5.8: Sensor sending a message.

Because of the potential complexity of the system and the relatively small number of objects that may be interested in the sensor data, a simple delegation message-passing interface was used to broadcast messages to the relevant objects. This makes sure that a reasonable throughput of messages is achieved without the possibility of system hangs due to a mis-configured or mis-coded sensor.

When a sensor needs to send some data, it contacts the *Sensor Subsystem* and asks that a message is sent (see figure 5.8). The *Sensor Subsystem* is then responsible for transmitting this message to the *Management Subsystem,* which in turn distributes the message to the other objects in the architecture.

5.3.2 The contextual rendering subsystem

Alternative forms of output such as speech may be more appropriate depending on the situation or context of the user, and this has already been discussed in section 5.3. With this in mind the *Contextual Rendering Subsystem* has been designed. It is responsible for the creation and management of output renderers within the architecture.

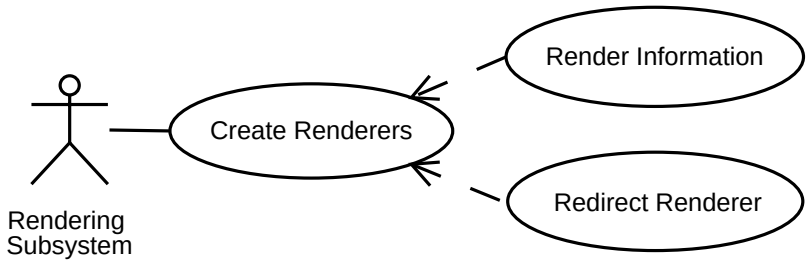


Figure 5.9: The rendering subsystem.

An application can request, via the *Management Subsystem*, that the *Contextual Renderer* should output information in a particular way. The *Contextual Renderer* makes decisions based on the state of the render redirection hash tables (discussed in section 5.4.2). Based on this information the correct renderer is chosen and the request from the application is passed to the renderer.

Again, the use of a standard software interface allows each renderer to communicate with the *Contextual Rendering Subsystem.* Also, the renderers are loaded at run time and register their presence with the *Contextual Rendering Subsystem.* The Use Case for this can be seen in figure 5.9.

Create renderers

When the system is first started the *Contextual Rendering Subsystem* is constructed by the *Management Subsystem*. It then identifies which renderers are available and constructs them by instantiating the objects. On construction, the renderers acknowledge that they are available by registering with the *Contextual Rendering Subsystem*. The sequence diagram for this case can be seen in figure 5.10 and shows the initialisation of two renderers. In a running system there may be any number of renderers and once a renderer has been constructed it sets up any internal configurations that are needed; this is discussed in more detail in section 5.4.7.

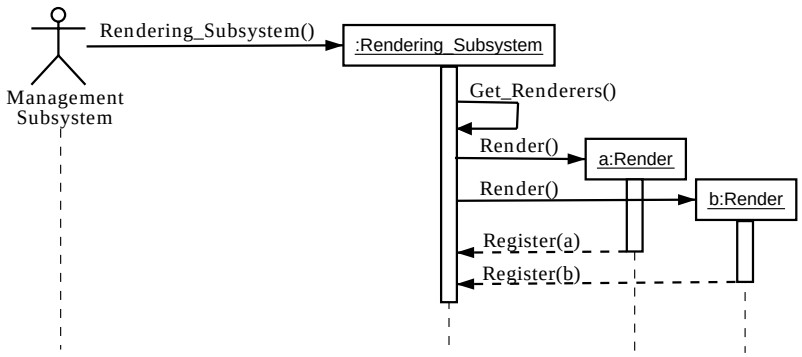


Figure 5.10: Initialisation of the rendering subsystem.

Redirect renderers

The redirection of renderers is manipulated by dedicated I.A.L services within the system. These services observe the environment through the *Sensor Subsystem* and if certain criteria are met they request that a renderer is redirected, the sequence of which can be seen in figure 5.11. The actual re-directions are performed by the *Contextual Rendering Subsystem* which keeps track of which re-directions are enabled.

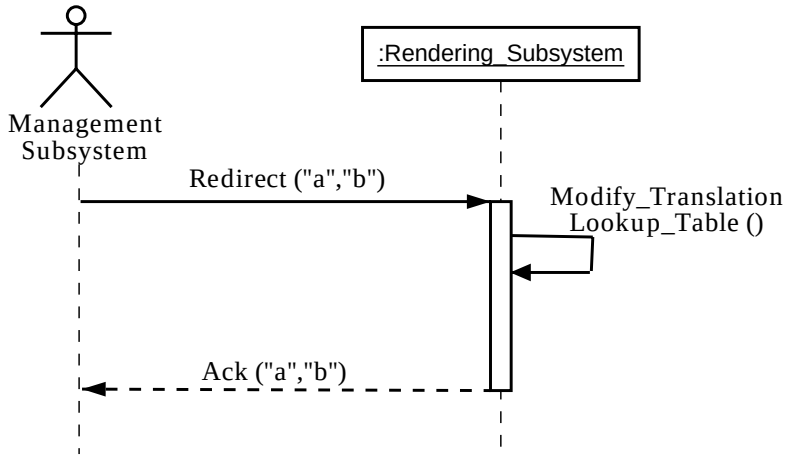


Figure 5.11: A render redirection message.

Render information

When an application needs to convey some information to a user, it sends a message to the *Contextual Rendering Subsystem* requesting the rendition of the information. For example, if the application asks to produce a visual response, it checks to see if any rendering re-directions have been configured. If there are no re-directions enabled then the request from the application is passed through to the relevant renderer; this can be seen in figure 5.12. If a rendering redirection has been enabled then the request from the application is passed to the renderer specified in the rendering redirection hash tables; this situation can be seen in figure 5.13.

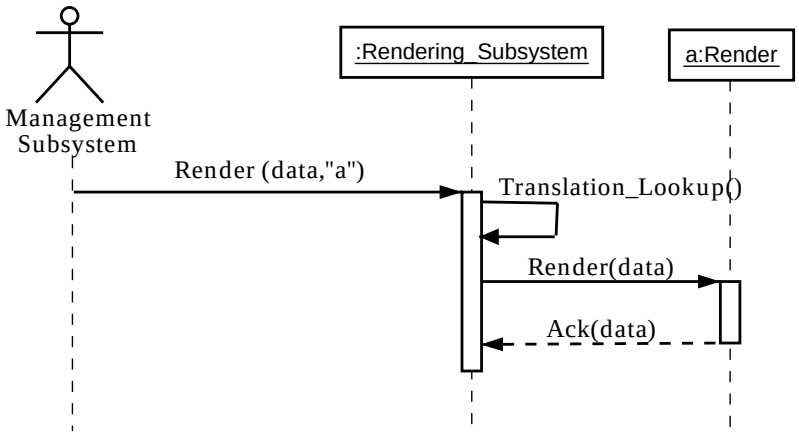


Figure 5.12: A render request with no re-directions.

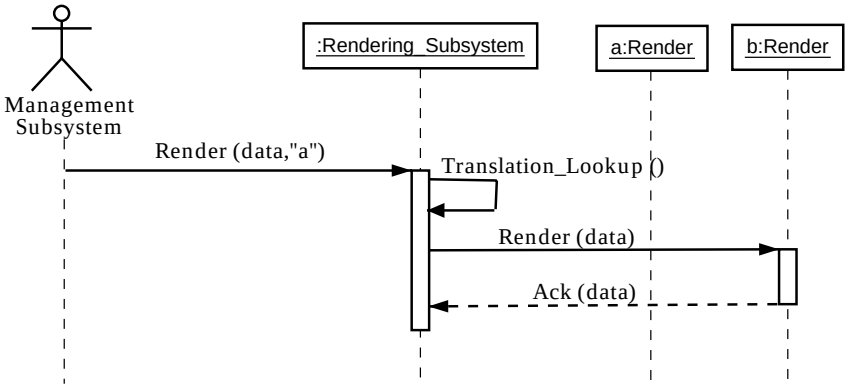


Figure 5.13: A render request with redirection from “a” to “b”.

5.3.3 The management subsystem

The heart of the system allows the *Sensor Subsystem*, the *Contextual Rendering Subsystem* and the applications to communicate with each other. The subsystem which handles this is called the *Management Subsystem* and is used as a central point for all communications within the system. The *Management subsystem* has three main responsibilities, which can be seen in figure 5.14. The first is to construct applications at the appropriate time, the second is to provide a generic interface to allow the other subsystems and applications to communicate, and the third is to receive commands from a user, decode them and pass the

command on to the relevant application.

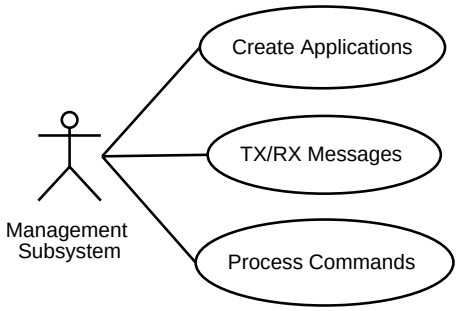


Figure 5.14: The Management Subsystem.

Create Applications

The two classes of application defined by [17] in section 2.4, namely the *Reactionary* and *Decisionary* applications, have been accommodated within the system. A *Reactionary* application simply receives a command and executes it. The *Management Subsystem* does not need to load all of the *Reactionary* applications when the system is initialised as they are instantiated only when they are needed.

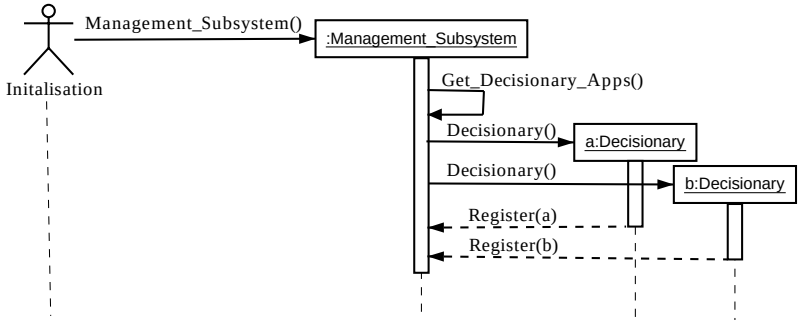


Figure 5.15: Construction of a Decisionary application.

The *Decisionary* applications make decisions based on the current perception of the environment and are analogous to the agents in [36]. These applications constantly monitor the environment through the *Sensor Subsystem*, therefore they need to be instantiated when the system is first started (initialised) and remain active until the system is shut down. These two classes of application

contribute to the different application creation sequence diagrams seen in figures 5.15 and 5.16.

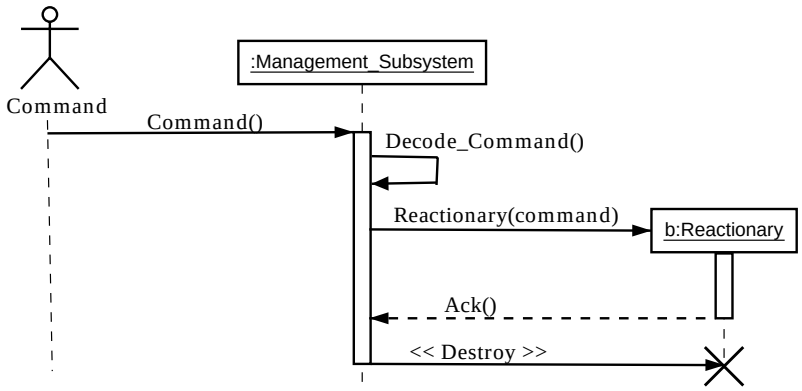


Figure 5.16: Construction of a Reactionary application.

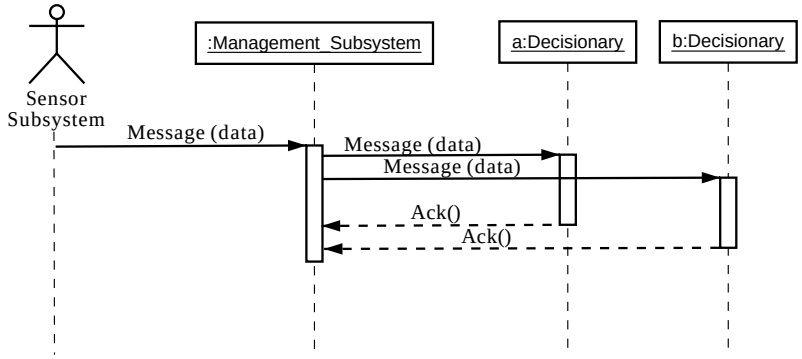


Figure 5.17: Message-passing from the Sensor Subsystem to the Decisionary applications.

Transmit/receive messages

The *Management Subsystem* is responsible for all the message-passing within the Sulawesi architecture. This is handled via a broadcasting mechanism which enables all *Decisionary* applications to receive information from the *Sensor Subsystem* and other applications. The sequence diagram in figure 5.17 shows the *Sensor Subsystem* as a UML actor, but this could be substituted by an application/service sending a message. The *Reactionary* applications do not receive any messages as it is not known when the application will be constructed, therefore

the messages cannot be guaranteed to reach this type of application.

Process commands

It is interesting to note that figure 5.16 also shows the sequence diagram for the *Reactionary* application command processing: the command is decoded and the corresponding application is constructed. The *Reactionary* application is passed the command on construction and then proceeds to process it. The application signals back to the *Management Subsystem* when the command has been completed, at which point the application is destroyed. A *Decisionary* application, on the other hand, will be sent a message containing the command for it to process. When a *Decisionary* application has finished it signals back to the *Management Subsystem* that it has completed the command. figure 5.18 shows the sequence diagram for this, note that the *Decisionary* application is not destroyed in this case.

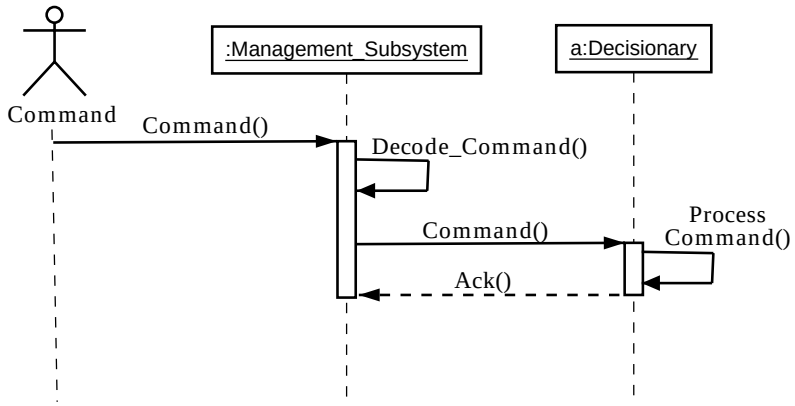


Figure 5.18: A Decisionary application processing a command.

5.4 Sulawesi implementation

The realisation of the Sulawesi system and the class structure of the implemented system can be seen in figure 5.19. It is written using the Java 1.1.x programming language, and this section provides a detailed description of the internal subsys-

tems. It can be clearly seen that the two types of applications, *Reactionary* and *Decisionary,* have their own separate interfaces, and in some cases are treated differently depending on the circumstances. This chapter concentrates on the main Sulawesi components, while the next chapter details the sensors, renderers and applications that have been constructed using this architecture.

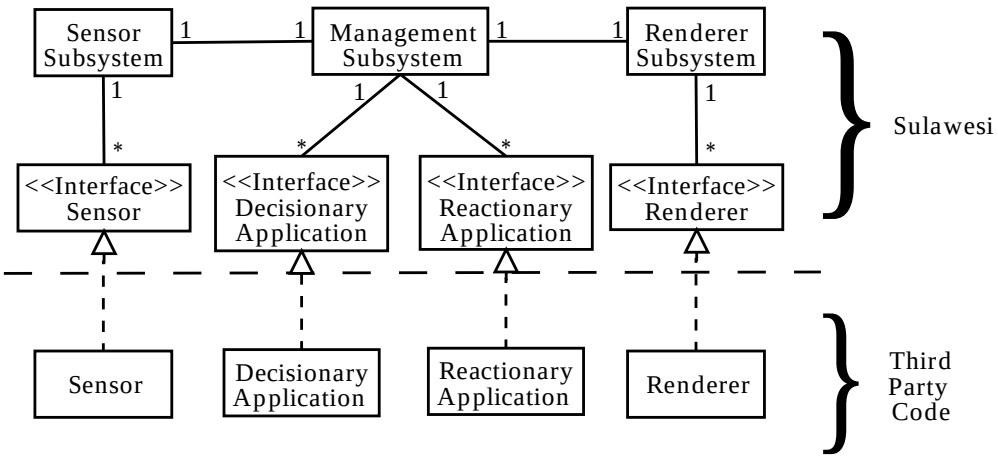


Figure 5.19: Class structure for the Sulawesi system.

5.4.1 Sentence structure

The use of look-up tables for natural language processing (described in section 5.2.2) inherently restricts the kind of sentences that can be used, but in a mobile environment where the user is concentrating on another task it is unlikely that the machine will be required to enter any lengthy dialogues with the user. It has been speculated by the author that in the mobile environment the types of functions that will be performed will require a terse set of commands. This philosophy has been used to design sentences that can be used within the Sulawesi architecture. In order to create a sentence which can be understood by Sulawesi the following rule needs to be adhered to:

[RENDER TYPE] [SERVICE NAME] [SERVICE ARGUMENTS]

In practice the [SERVICE ARGUMENTS] are all the words that follow the [SERVICE NAME]. If no render type is specified then a default render type is

used.

For example: assuming an email service is available and a user needed to send an email to Adrian, the following sentence would not work because the arguments to the email service are be *"about tomorrows meeting on agents":*

SEND ADRIAN AN EMAIL ABOUT TOMORROWS MEETING ON AGENTS

The email service will fail because it is not be able to determine who to send the email to! With only a few minor modifications the sentence can be rearranged so Sulawesi can interpret it:

SEND AN EMAIL TO ADRIAN ABOUT TOMORROWS MEETING ON AGENTS

In this case the arguments to the email service are *"to adrian about tomorrows meeting on agents"*. The adaptation of the sentence is still understandable to a human, but it now allows the email service to determine to whom the email should be sent, and the subject of the email.

The point which needs to be emphasised here is the ability to infer a meaning from a relatively natural sentence rather than the user having to adapt to the machine and remember complex commands.

5.4.2 Renderer look-up table

It is not practical to hard code all possible language transformations into a piece of software. Apart from taking a long time, such a system would not be easily adaptable to alternative situations.

The use of look-up tables provides an efficient way to enable a user to configure the system with their own personal sentence preferences without having to re-program or re-compile the sentence understanding code. Sulawesi knows what the words "*show*" and "*tell*" mean by referring to the renderer look-up table. The table is similar to the list shown below and is used to determine which output channel a result should be sent to.

|1|SAY|SPEAK|

|2|TELL|SPEAK|

|3|READ|SPEAK|

|4|SHOW|TEXT|

|5|DISPLAY|TEXT|

The first entry in the look-up table specifies that the first time the word "*say*" is encountered in a sentence, the results of the service should be sent to the "*speak*" output renderer.

5.4.3 Command buffer

Once a sentence has been decoded, the relevant information is entered into a command buffer. This buffer, referred to as a "*Batch*" in the Sulawesi source code, is constructed by the *Management Subsystem* and is passed between the services and the renderers during the lifetime of the command. The command buffer consists of a five element string array which contains the command, the contents of which can be seen below.

Batch[0] contains the process ID, assigned by the servicemanager.

Batch[1] contains the service to invoke.

Batch[2] contains the arguments to the service.

Batch[3] contains the type of renderer to use.

Batch[4] contains the renderer arguments.

When a command enters the system, the command buffer is constructed and passed to the relevant service. The service processes the command, on completion the buffer is passed with the service results, via the *Management Subsystem*, to the *Rendering Subsystem*. At this point the command buffer is used to determine how to render the results from the service.

5.4.4 Command execution

In order to see the how a command is passed through the Sulawesi system, figures 5.20 and 5.21 provide flow charts showing the various stages in the lifetime of a command. Observations of the chain of events provide an insight into the

system as the commands *"could you tell me the news, the sports pages please"* and *"could you show me my current location"* are processed.

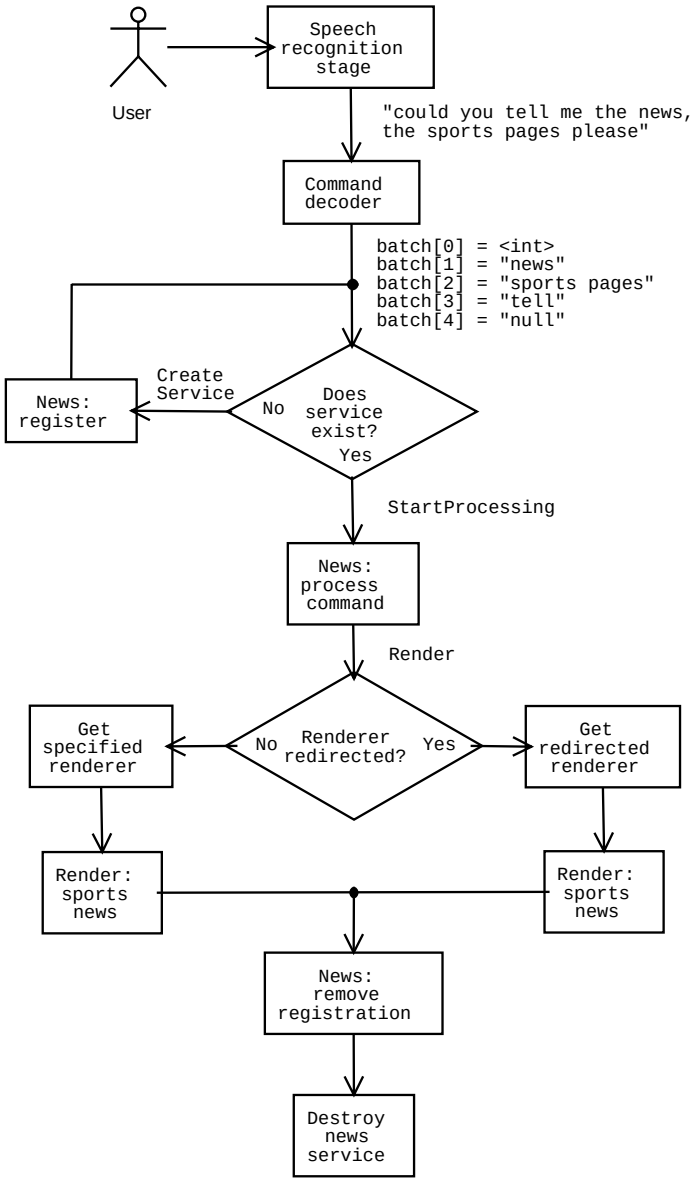


Figure 5.20: A *Reactionary* command being processed.

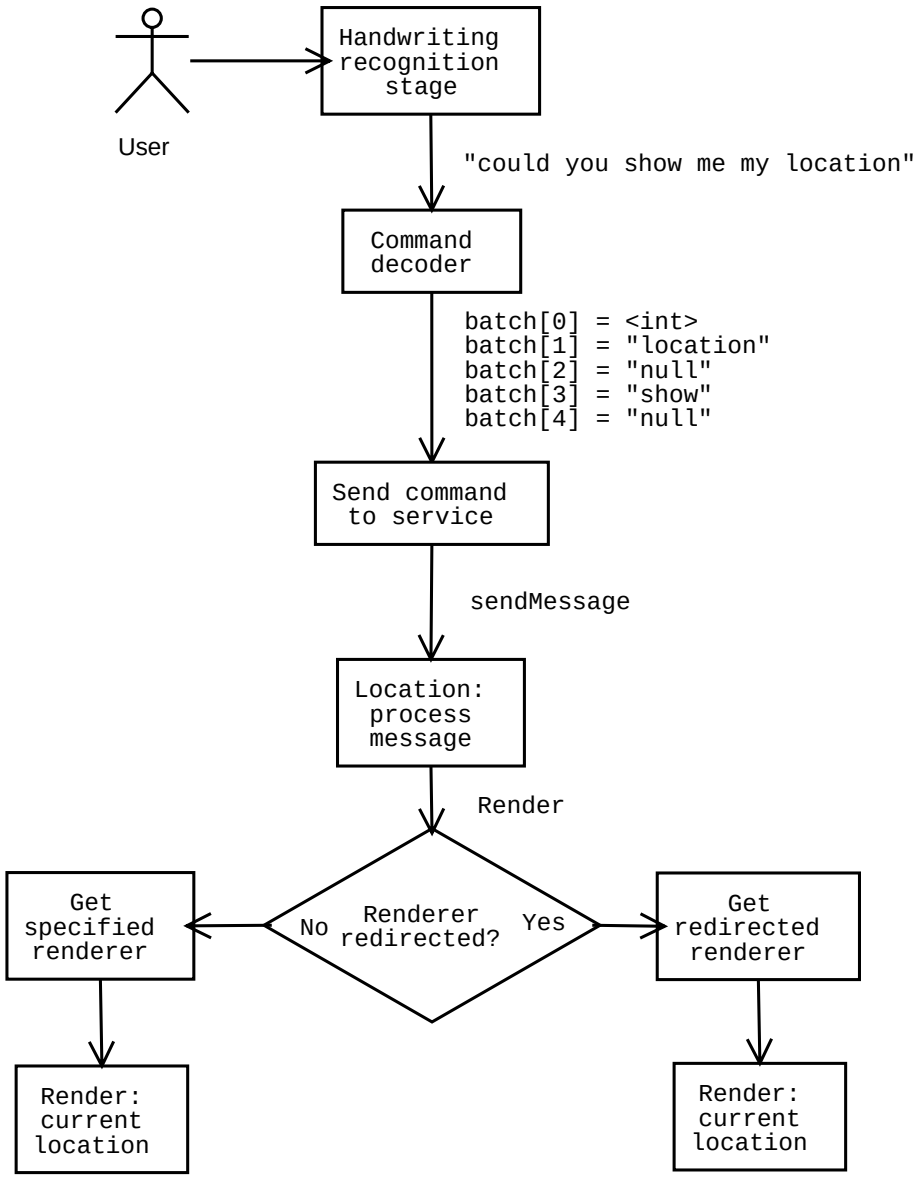


Figure 5.21: A *Decisionary* command being processed.

5.4.5 Management Subsystem

The *Management Subsystem* is responsible for the communication between the sensor subsystem, the rendering subsystem, and the construction of the services within the Sulawesi. Section 5.3.3 describes two types of application that can be implemented within the framework. These two application types contribute toward the different software interfaces defined within the architecture, namely

the *Decisionary* and the *Reactionary* interfaces.

The Reactionary interface

The *Reactionary interface* is very simple and provides only one method which needs to be implemented, the *startProcessing* method. The instantiation and registration of a *Reactionary* service results in the `startProcessing` method being called by the *Management Subsystem*. The `startProcessing` method provides a way for the *Management Subsystem* to signal to an application that it can start processing a command. When the request has been completed the application's results are returned to the *Management Subsystem* which passes the results on to the *Renderer Subsystem*.

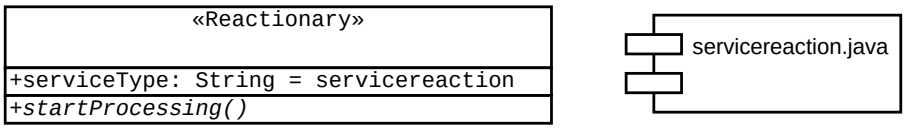


Figure 5.22: The Reactionary interface.

An application which implements the *Reactionary interface* needs to include some additional code within the constructor. As can be seen in the listing in appendix B.1, the constructor needs to accept an object reference and an integer priority level (line 13). Registration is then carried out and the name of this application, the object reference, the type of service and an empty command buffer are passed back to the *Management Subsystem* (line 18). The command buffer is used by the *Management Subsystem* to pass a command to a application upon registration. The final stage in the constructor is to set the application's thread priority (line 21).

After registration, the command buffer is filled out by the *Management Subsystem,* which then calls the application's `startProcessing` method, enabling the application to process the command.

On completion of the command, the application needs to ask the *Management Subsystem* to render the results (line 13). Finally, the application requests

that its registration is removed from the system (line 47), and the *Management Subsystem* then destroys the application.

The Decisionary interface

The construction and registration process in the *Decisionary* interface is similar to the *Reactionary* interface and this can be seen in the listing in appendix B.2, lines 13-19. The *Decisionary* interface also defines some other methods to allow asynchronous communication between the application and Sulawesi.

When a *Decisionary* application is constructed (line 13) it registers itself (line 19). If Sulawesi needs to send the application a command, there needs to be a way of allowing it to be sent to the service. The `getBatch` method provides a software call-back which allows the *Management Subsystem* to retrieve the command buffer and place a request in it.

The `recieveMessage` method (line 33) allows the *Management Subsystem* to pass messages generated from the *Sensor* and *Renderer Subsystems* into the *Decisionary* application. If need be, the service processes these messages and makes decisions based on the information received. The last method which needs to be implemented is the `className` method (line 28) which simply returns the textual name of the application. This is needed by the *Management Subsystem* when trying locate and communicate with a specific application.

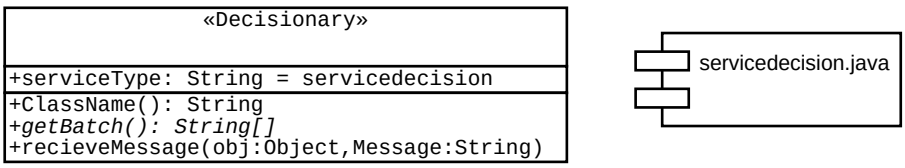


Figure 5.23: The Decisionary interface.

Apart from the various methods which need to be implemented, a *Decisionary* service needs to include a method called `writeObject`(line 45). This is called by the Java Virtual Machine when the *Management Subsystem* serialises the application when Sulawesi is closed down.

The ThreadManager class

The *Threadmanager* is responsible for maintaining a list of which applications are present in the system, the application names, the object references and the applications thread priority. Methods for adding and removing threads to and from the hash-tables are provided via the `addThread` and `removeThread` methods. The `addThread` method queries the application to find out its priority level, and then the application is placed in the application hash-table. The `removeThread` method, in contrast, removes a particular application from the hash-table. In order for the *Management Subsystem* to find and send messages to particular applications, the `findObjects` method is used to retrieve object references from the internal hash-tables.

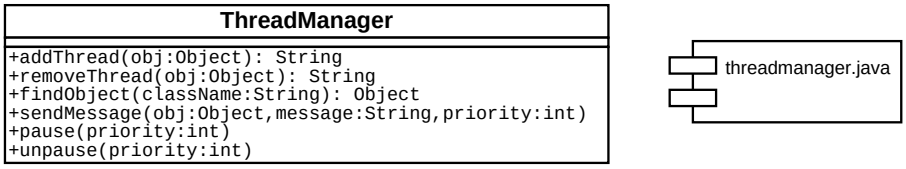


Figure 5.24: The ThreadManager class.

In order to call the pause method in every *Decisionary* application, the `pause` method is used to pause all applications in a certain priority group. To complement this function the `unpause` method is used to resume previously paused *Decisionary* applications. The `sendMessage` method is used to send a message to every service in a certain priority level. The object reference of the message sender is included in each message to allow the recipient to determine where the message has come from.

If the *Management Subsystem* needs to shut down any running applications, the `serializeObjects` method provides a mechanism to serialize all running decisionary services to files on the disk. This is used to store any persistent data that might have been present within the *Decisionary* applications.

The ServiceManager class

The *ServiceManager* is the central point of communication for the *Sensor* and *Renderer Subsystems*. The *ServiceManager* decides which messages are to be sent and to where, it provides mechanisms for decoding the natural language input streams and it is also responsible for the initialisation and destruction of the applications within the architecture.

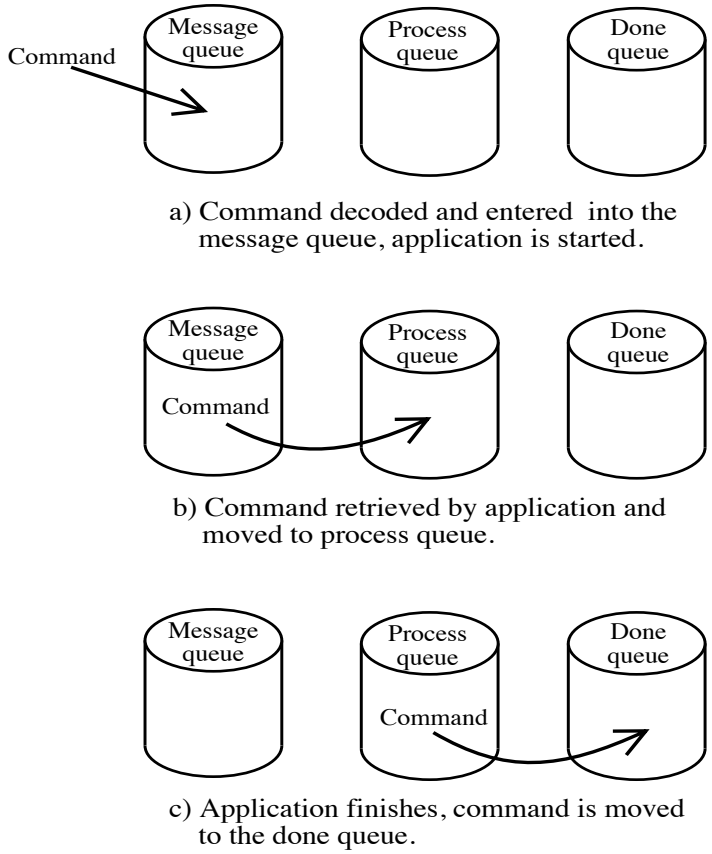


Figure 5.25: The message queue transitions.

The *ServiceManager* is initially constructed by the *Sensor Subsystem,* and after construction all the applications found are written to an internal configuration table. The *ServiceManager's* constructor argument is the object reference of the *Sensor Subsystem* which is stored to allow bi-directional communications.

In order to pass natural language commands to the system, the *ServiceManager* contains a `DataIn` method. The *ServiceManager* decodes the command

(as described in section 5.2.2), decides which application is being requested and finally puts the command into the message queue file (see figure 5.25).

If the requested is for a *Reactionary* application then it is constructed and passed the command. If, on the other hand, the request is for a *Decisionary* application then a message containing the command is sent to that application. At this point the *Decisionary* application retrieves the command from the message queue. A *Reactionary* application will be given the command when it calls the `register` method, passing its object reference to the *ThreadManager*. A *Decisionary* application calls the `getProcessBatch` method to retrieve the command.

In both of these situations after the command has been sent to the application, the *ServiceManager* moves the data from the message queue to the process queue file (see figure 5.25). When the application has finished processing the command and the appropriate response has been created, the service calls the `remove` method which initiates the moving of the command from the process to the done queue file (see figure 5.25c). If the application is a *Reactionary* one, the application's object reference is removed from the *ThreadManager's* hash tables and the application is destroyed.

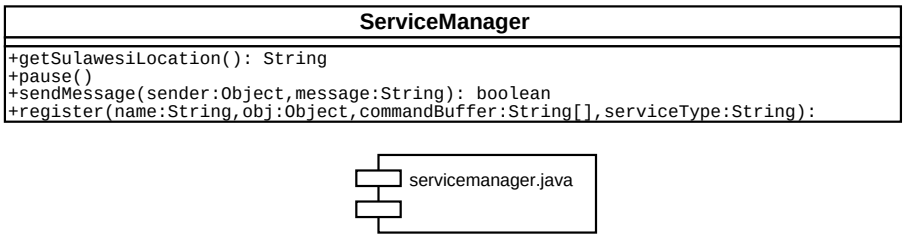


Figure 5.26: The ServiceManager class.

The use of message queues on the disk means that if the system suddenly loses power, the state of the system can be determined by looking at the messages queues. A basic form of recovery can then be performed when the system is restarted by analysing the message queues and performing simple string searches. As messages only traverse the queues in one direction it is possible to

determine whether the message is corrupt and whether to restart a particular service. (although this functionality has not been implemented).

The `pause` method is used as a central point to pause and unpause all applications within the architecture. It calls the `pause` and `unpause` methods in the *ThreadManager* for each priority level.

The `getSulawesiLocation` method is used by the *Sensor* and *Renderer Subsystems* to determine where the framework resides in the file-system, and is also used to determine the location of the input and output directories for native device drivers.

In order to close down the system nicely the `shutdown` method is used to save the state of the services. When called, the *Decisionary* applications are paused and serialised to disk. When the system is next started, any services which had been previously stopped are un-serialized and allowed to resume their processing. This mechanism provides a basic form of persistence processing between power cycles.

The last, and probably the most crucial, method in the whole framework is the `sendMessage` method. When any sensor or application needs to send a broadcast message, they call this method supplying their object reference and a string containing the message. The *ServiceManager* then sends the message to all applications within the framework, with the exception of the sender of the message.

5.4.6 Sensor Subsystem

The *Sensor Subsystem* is implemented as a two separate components. The first component is the SENSORBASE interface (or INPUTBASE in the source code). This provides a generic interface for third party sensors to implement and allows enhancements to be made to the system without having to rewrite any of the architecture. The interface provides a generic way for a sensor to be loaded and controlled by Sulawesi. The interface, as seen in figure 5.27, defines a single

method called `Query` which allows the *Sensor Subsystem* to request data from the sensor.

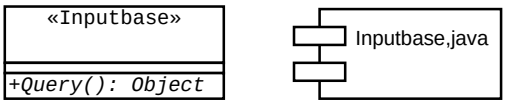


Figure 5.27: The SENSORBASE interface.

Although only one method is defined in the SENSORBASE interface, in practice a sensor which implements the interface needs to provide an additional function. The following refers to the example sensor code in appendix B.3. The constructor of each sensor has to accept a single object as an argument (see line 9). This argument allows the *Sensor Subsystem* to pass in its object reference. The reference needs to be stored (line 11) as it allows the sensor to register (line 12) and to communicate asynchronously with the *Sensor Subsystem*. Without this code the sensor will not be automatically integrated into the Sulawesi system.

The *Sensor Subsystem* (or Input Subsystem in the source code) provides a centralised point for constructing all the sensors. On initialisation, the *Sensor Subsystem* is constructed by the *Management Subsystem* which passes in it's object reference via the `passInTheManagerPointer` method. This allows the two subsystems to communicate with each other.

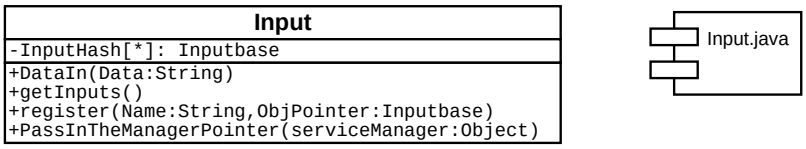


Figure 5.28: The Sensor Subsystem.

When the *Sensor Subsystem* is initialised, the *Management Subsystem* requests that the sensors should be loaded via the `getInputs` method. A list of available sensors is then kept by the *Sensor Subsystem* in the `SensorHash` table.

After being loaded the sensors register themselves via the `register` method and their object references are placed in the `SensorHash` table for later use.

The *Sensor Subsystem* contains a `DataIn` method which allows a sensor to send data or commands into the system. The *Sensor Subsystem* does not process the data in any way, and all commands or data that are received by this method are passed straight through to the *Management Subsystem*.

5.4.7 Renderer Subsystem

The *Renderer Subsystem* is implemented as two separate components similar to the *Sensor Subsystem*. The RENDERBASE interface (seen in figure 5.29) is called OUTPUTBASE in the source code, and provides a generic interface which allows third party renderers to implement and enhance the system. The RENDERBASE interface provides methods which allow a renderer to be loaded and controlled by Sulawesi, and three methods exist which a renderer must implement.

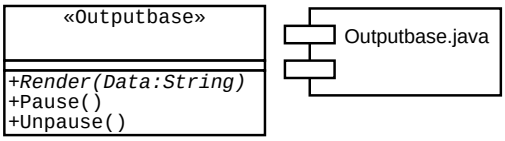


Figure 5.29: The RENDERBASE interface.

The `Render` method is used to request that a certain piece of information be output in some form. The `pause` method is used to pause the rendition of a certain piece of information and the `unpause` method is used to resume a previously paused rendition. All of these methods are called directly by the *Renderer Subsystem* to control the rendition of information and not by the applications directly.

As with the SENSORBASE interface, each renderer allows the constructor to accept a single object as an argument (as can be seen in appendix B.4 line 9). The object passed in is the reference to the *Renderer Subsystem*. The renderer stores this reference (line 11) and then registers itself with the *Renderer Subsystem* (line 12).

The main part of the *Renderer Subsystem*, seen in figure 5.30, provides the mechanisms for constructing the renderers. The *Renderer Subsystem* is also

responsible for the management of the available renderers. When a renderer registers (via the `Register` method) the *Renderer Subsystem* stores the registering objects reference in the `RenderHash` table. This hash table is used to determine which renderer is responsible for rendering which type of information. The `renderoutput` method is used by the *Management Subsystem* to request the rendition of a piece of information. When this happens the *Renderer Subsystem* checks the `RenderHash` table to find the requested renderer, and then the information is sent to that renderer to be rendered.

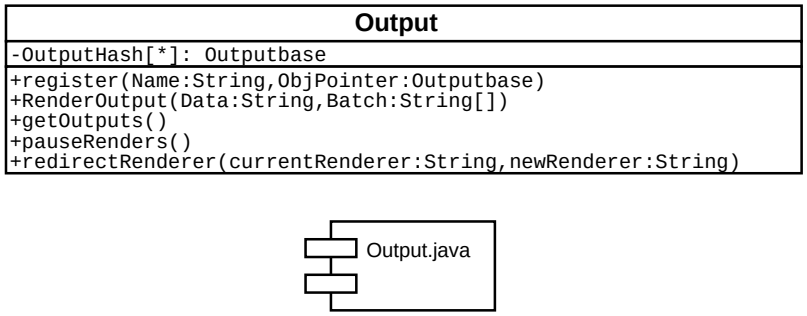


Figure 5.30: The Renderer Subsystem.

When the *Management Subsystem* needs to pause a rendition it calls the `pauserenders` method. This results in calling the *pause* method of each renderer in the `RenderHash` table. A subsequent call to the `pauserenders` method results in the `unpause` method of each renderer in the `RenderHash` table being called. This enables applications to be able to pause the rendition of information until a more appropriate time.

The *Renderer Subsystem* also provides a mechanism to allow applications to redirect a renderer via the `redirectRenderer` method. This enables renderer redirections, explained in section 5.2.3, to be performed.

5.5 Chapter summary

This chapter describes in detail the software architecture involved in implementing what the author believes to be key functional requirements for a contextually-aware mobile user interface system.

The Sulawesi architecture has been designed to explore the design and implementation of a software framework for a contextually-aware user interface system. To focus on individual components of Sulawesi, the design is split into individual subsystems which can be developed independently of other components, and by defining the format of data passed between these subsystems, the message-passing system provides a consistent communication interface which allows other subsystems and agents to receive and process information easily. These systems have been split into three distinct groups, namely the *Sensor*, *Management* and *Contextual Rendition* subsystems.

The *Sensor* subsystem is responsible for the initial registration and communication conduit for the external sensing devices. The *Management* subsystem is the central conduit for message-passing between the *Sensor* and *Contextual Rendition* subsystems. It is also responsible for the management of the agents within the framework. Two types of agents paradigms have been accommodated within the framework, namely the *Reactionary* and *Decisionary* agents [17]. This allows one type to react to commands when instructed, while the other makes decisions based on information from the *Sensors* and other agents within the system, producing results when triggering criteria are met.

The presentation of information, or rending, is controlled by the *Contextual Rendition* subsystem. This allows *Decisionary* agents to influence output rendition type depending upon the agents perception of the environment.

Because this system has been has been targeted at mobile/wearable systems, the author feels that the use of speech recognition and natural language is important. The system allows rule-based sentences to command the agents

within the system. The sentences contain information about which agent should be controlled, commands for that agent, and how the results from that agents should be presented to the user. The author believes that these sentences are easier to remember and dictate when in a mobile situation than the manipulation of a complex user interface.

Chapter 6

Applications built on Sulawesi

6.1 Introduction

The Sulawesi framework allows applications to be constructed which explore alternative user interfaces for mobile and contextually aware applications. To test the viability and functionality of Sulawesi, two prototype applications have been constructed with various sensors and rendition mechanisms. This has enabled the author to start exploring alternative interaction mechanisms and applications for mobile users.

The Sulawesi system described in the previous chapter only provides a framework in which to develop alternative applications and interaction mechanisms. This chapter details the applications, sensors and renderers that have been developed to demonstrate the capabilities and features of the Sulawesi framework.

6.2 Sensors

The gathering of some contextual information can be relatively simple, while other contexts can be much more difficult to obtain. In order to provide the system with information about the persons environment, three sensors have been integrated with Sulawesi. These sensors provide raw data from which other contexts can be determined. The simplest which has been explored by many

other people is the context of location, and two sensors which provide location information are used. The third sensor is an accelerometer which provides the system with acceleration information about the user. These sensors were chosen because they were already available off-the-shelf, and this reduced the development time.

6.2.1 Global positioning system

One of the location devices that has been used is a GPS receiver. The device (a 12 channel Canadian Marconi Company Allstar receiver) provides an outdoor accuracy of approximately ten metres and communicates via a standard RS232 serial port. A dedicated device driver[1] is responsible for the low level initialisation of the board, and the data received are converted into Longitude, Latitude and Altitude (LLA) coordinates. The LLA data are generated by the driver once every second and a GPS sensor component broadcasts the data into Sulawesi (see figure 6.1).

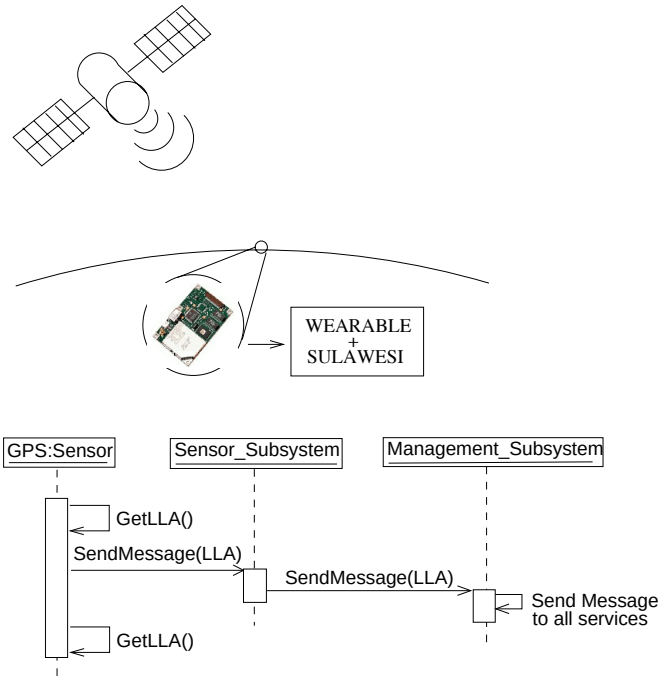


Figure 6.1: GPS generating LLA signal and being broadcast into Sulawesi.

[1]The Linux driver was written by David J. Johnson of the University of Essex.

This sensor allows the system to know where a person is on the surface of the planet (with a few exceptions like the polar regions!). Unfortunately, GPS only works outdoors, so when the receiver enters a building of any substantial size the GPS signal is severely attenuated by the walls. In this situation the positional information is lost.

The GPS receiver will still produce location messages based on its last known position, but the positional errors introduced are roughly proportional to the size of the building. For example; if a person walks in one end of a building and out the other, the GPS error at the far end (before the user leaves the structure) will be the roughly equal to the length of the building.

6.2.2 Infra-red

In order to overcome the indoor deficiencies of the GPS receiver, a second location device provides location information inside a building with a small amount of infrastructure [8, 70]. The Infra-red (IR) transceivers[2] are based on the MIT locust with modifications to allow bi-directional data transmission.

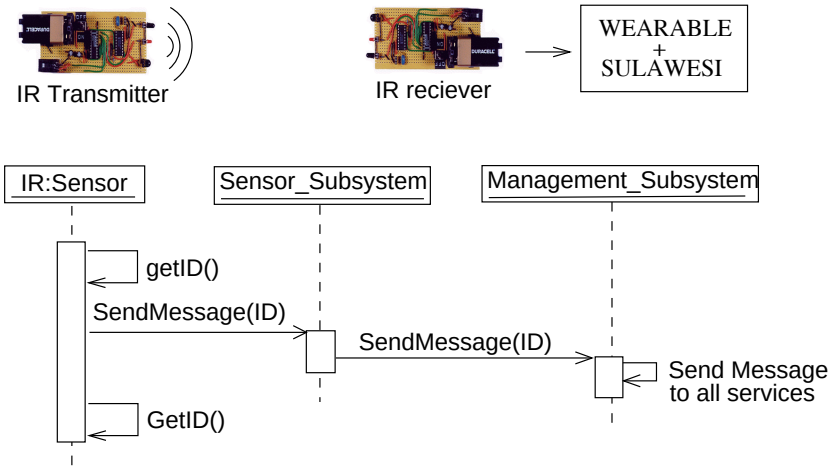


Figure 6.2: Infra-red transceiver broadcasting an ID.

The device consists of a PIC microcontroller which encodes and decodes the raw IR signal, and a MAX232 transceiver which allows the device to commu-

[2]Built by Panagiotis Ritsos of the University of Essex.

nicate with an external processor via an RS232 serial port. Each of the IR transmitters are programmed to broadcast a unique identification number every few seconds, and a physical location can be *tagged* (as defined by [55]) by placing a transmitter at that location.

The transmitters are placed inside various rooms around the university. The identification number is used as a form of location information, and with a simple translation table it is possible to determine the rough location of the user from the IR signal.

The IR receiver is connected to the wearable computer and transfers any signal from the device to Sulawesi via a dedicated device driver. The identification number is received from the IR device by a dedicated driver, and a Sulawesi sensor broadcasts this information into the Sulawesi system.

6.2.3 Accelerometer

The use of a simple 2-axis accelerometer enables the system to gather information about the user's posture and is similar to the work of [18]. The physical device is an Analog Devices adxl202. The author has placed this sensor on the upper part of a persons leg, seen in figure 6.3, and when the user moves about the sensor detects the acceleration values of the leg. The device connects to a RS232 serial port and a device driver gathers the X/Y acceleration values once a second and transmits this information into the Sulawesi framework. By positioning the accelerometer at the top of the leg it is possible to determine simple contexts such as whether a person is standing up, sitting down or walking. This is discussed further in section 6.6.

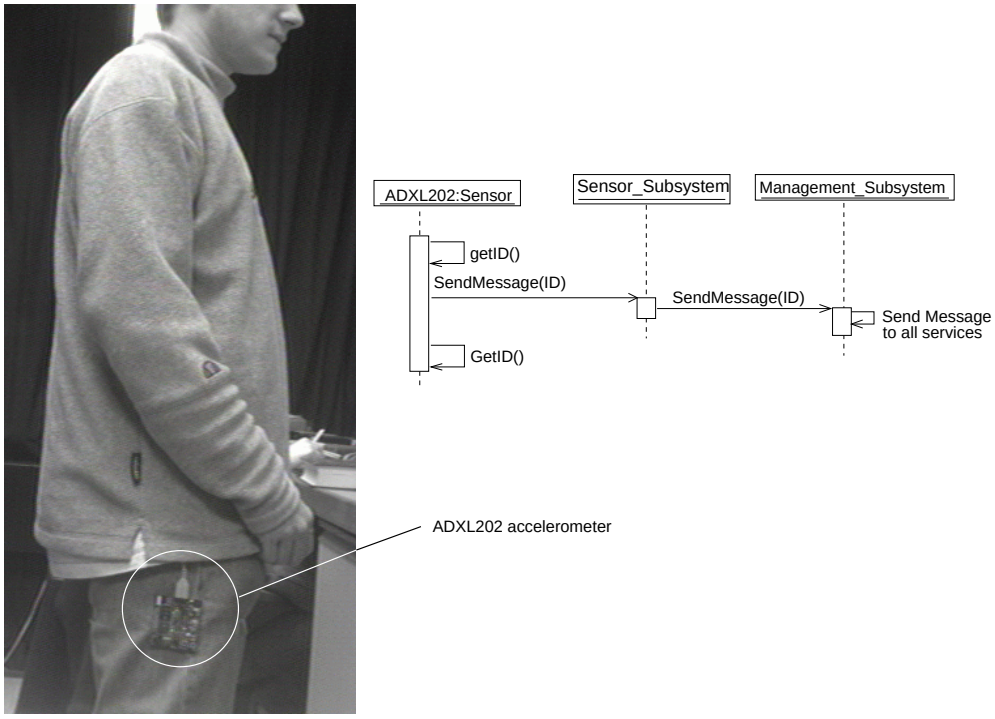


Figure 6.3: The accelerometer worn on the leg.

6.3 Renderers

The renderers that have been developed convert the output from a service into a format suitable for presentation to the user. The current services produce textual responses which can be easily converted into a visual or audible rendition. The author has focused on the simple visual and audible translations in this section.

6.3.1 Gili: A prototype wearable user interface

A prototype monocular *Primary Task Interface* has been constructed using the observations of the head-mounted displays mentioned in chapter 4. A Virtual IO[3] augmented HMD is used to project the user interface over the user's physical environment and one of the display panels on the head mount was removed to

[3]I-O Display Systems LLC `http://www.i-glasses.com/`

make the system monocular.

As can be seen in figure 6.4 the user interface is coloured black and white. It is analogous to the console-based systems in use by many wearable users and the high contrast between black and white aids the visibility of the interface in various lighting conditions. This enables a user to *see through* the interface with as little of their vision obscured as possible. There are some questions about whether using white text on a black background is the best way to proceed. I believe that in a mobile situation the best option is to obscure they users eye with as little light as possible, this means to provide a black background. However, I do accept that this assumption has no scientific proof in a mobile situation, and this question will have to be answered by others in due course.

Figure 6.4: Gili user interface (Linux + Java).

While Gili shares some features with the console-based systems, the interface is designed to allow multiple applications to share a work-space and to select them easily using the current input devices. Although devised and developed independently, Gili shares some characteristics with the GUI on EPOC devices such as the Psion 5 with an independent application area and menu driven commands.

The user interface has been designed for a *lower than desktop* graphical resolution and colour depth. The interface uses 320 × 240 pixels (1/4 VGA)

and allows grey scale displays such as the M1 to provide good contrast between the black background and the white features.

Within the Gili interface the main graphical controls have been placed in the users peripheral field of view; this enables a user to see through the main panel of the display while performing other tasks.

A command entry box is positioned at the bottom left of the user interface and a row of interactive menu selection buttons are positioned on the right hand side of the interface. The four graphical buttons can be selected via dedicated single key presses on the Twiddler. They have also been made as large as possible to try and overcome the deficiencies in using the Twiddler to select small targets (as described in section 4.10) should this be required. The menu buttons also provide early confirmation (as termed by McGlasham et al. in section 2.3) via visual feedback, flashing in "reverse video" for a short period of time when they are selected.

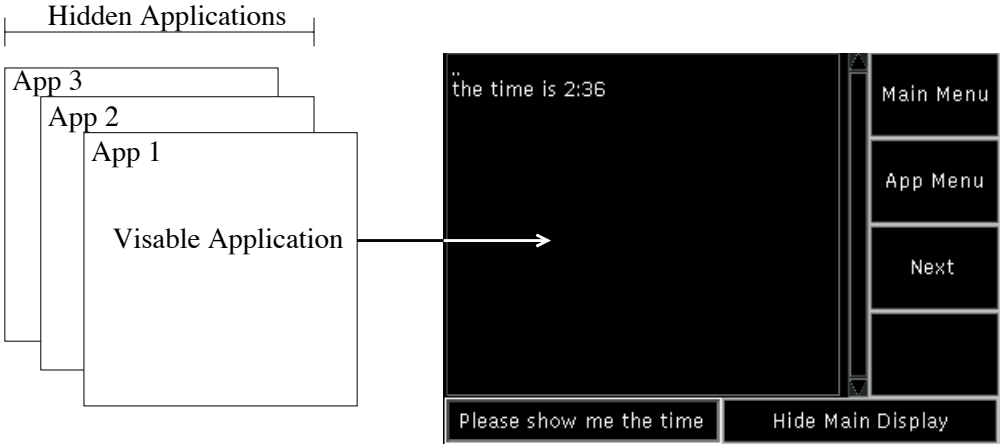


Figure 6.5: Stacked applications.

The main panel allows multiple applications to inhabit the user's foveal field of view. This mechanism allows applications to be *stacked* with only the application at the top of the stack being visible (see figure 6.5). The menu buttons provide a mechanism to raise and lower applications on the stack. The menu in figure 6.5 shows a **Next** button which is used to rotate the applications

on the stack. This results in a previously hidden application becoming visible. The menu also shows a **Main Menu** button which does not change. Pressing this button will return to this initial menu.

The menu selection mechanism is designed to be out of the user's foveal view and is accomplished by placing the menu to one side of the Gili interface. When an application is at the top of the stack it can override the bottom three menu buttons. The overridden buttons are activated by pressing the **App Menu** button and results in the application menu becoming visible (see figure 6.6).

Figure 6.6: Notes application menu (Windows + Java).

The last part of the graphical display is perhaps the most valuable for a mobile user interface. The **Hide Main Display** button enables the main panel to be turned on and off. This provides a mechanism which allows a user to free up the main panel of the interface and it also provides software hooks which allow a Sulawesi application to blank the main panel if it decides that the situation calls for it. This is achieved by calling the pause method in the Gili Interface code.

The Gili user interface provides an API to allow applications to integrate with the buttons and main panel of the Gili interface. Figure 6.7 shows the software interface. The `setup` function is called when the application is first instantiated. The application then contacts the *ServiceManager*, retrieves the Gili object reference and registers with Gili. Once this has been achieved, Gili can then interface with the application. When the application is in focus (i.e.,

at the top of the stack) Gili invokes the `focused` method of the application. Here the application can override the names of the buttons by defining them in the `userButtonXname` methods (where X=1,2,3). When a button is selected, Gili passes the event onto the `userButtonXpushed` method and the application processes this event.

Others have commented that the use of textual buttons may not be the easiest solution in a mobile environment as the time taken to read and identify which button to press may be longer than the identification and selection of a graphical icon. Again, the author recognises that the use of graphical icons may indeed be a better solution, but has no scientific proof to suggest this would be the case in a mobile situation. Further work would need to be performed to confirm this assumption.

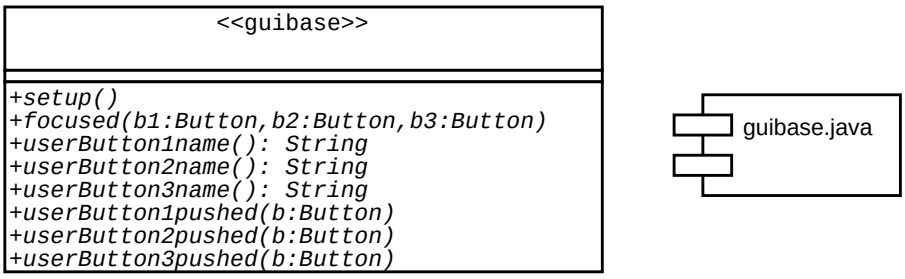


Figure 6.7: Gili application interface.

This mechanism allows several application to inhabit the same physical area, but only the application in focus can be controlled by the user through Gili. It is possible to control applications by other means such as speech and this is discussed further in section 6.7.3.

The Gili graphical interface, while simple in it's construction fulfils some of the outlined requirements for a mobile graphical interface defined in chapter 2. Namely, the simplicity of the interface to reduce clutter, the use of a simple menuing system to overcome pointing and selection tasks, the use of high contrasting colours (although black text on a white background may be easier to read, a dedicated application area, and a user interface which is controllable

via point & click operations as well as speech and direct manipulation through dedicated keys. Unfortunately, due to time constraints, the interface has not undergone an empirical study to confirm the acceptability of this layout. However a simple task involving four people was undertaken. Each participant was asked to navigate along a series of corridors with the monocular immersive HMD first displaying the a graphical desktop interface (standard Windows 98 desktop with no applications open), and then the Gili graphical interface. At the end of these simple trials each person said it was far easier to navigate and perform their walking task with the Gili interface. While no actual manipulations of either interfaces were performed, the indication is that the Gili interface is a easier to view when performing other tasks.

6.3.2 Text renderer

The simplest output renderer which has been created is one that displays basic ASCII text. As all of the services that have been developed within the Sulawesi framework produced a textual result this seemed appropriate. A visual text renderer has been developed and integrated within Gili (see figure 6.8), and allows any Sulawesi service to display a textual response via the Gili user interface. The figure shows the user asking to be shown the time, and the response being displayed in the text renderer inside the Gili interface.

6.3.3 Speech generation

The textual results produced by the Sulawesi services can also converted into speech using various text-to-speech methods and off the shelf software has been used to speed up the development time.

The simplest method for speech synthesis simply passes the text string to the Rsynth[4] program and, although fairly crude in its rendition, Rsynth produces a quick and understandable response.

[4] `http://wwwtios.cs.utwente.nl/say/index.html`

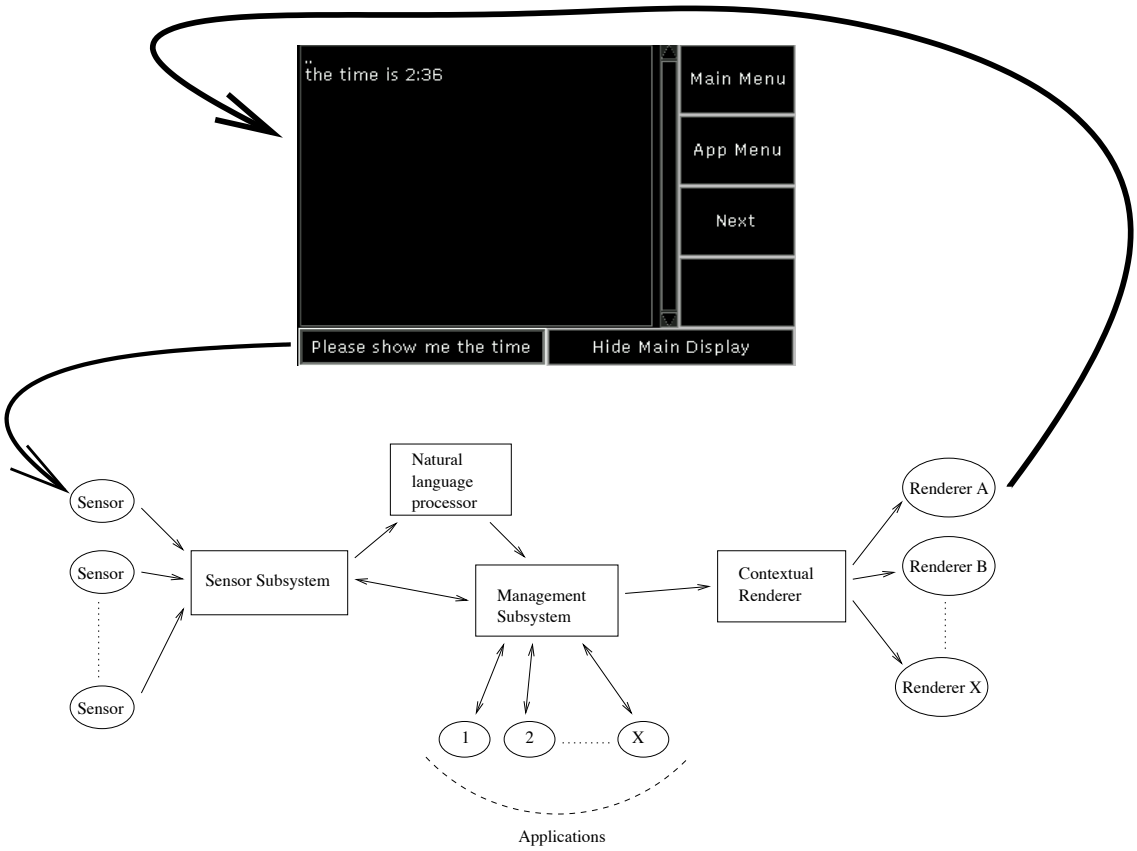


Figure 6.8: Gili and Sulawesi.

The next method uses IBM's ViaVoice[5] software and the Java Speech API[6] to produce a spoken output. Although the rendition is clearer than the Rsynth program, the ViaVoice implementation available for Linux (the authors development platform) was unstable at the time of development.

The last speech renderer uses the Festival [1] software from the University of Edinburgh. While this method produces the best sounding speech in comparison to Rsynth and ViaVoice, the software is more processor intensive than the other two systems and this resulted in a much slower response on the (rather slow) wearable computer used for this work.

[5] `http://www-4.ibm.com/software/speech/index.html`

[6] `http://www.alphaworks.ibm.com/tech/speech`
`http://java.sun.com/products/java-media/speech/index.html`

6.4 Commanding the machine

As described in section 5.2.2, Sulawesi can accept commands in the form of well crafted sentences. There are currently two ways of entering these sentences. The first is via the Gili interface. Using some form of keyboard it is possible to enter a command via the GUI, but in practise the keyboard input is mainly used for demonstration and testing purposes because it is much slower trying to type a sentence with the Twiddler in the real world.

The second way to enter a command is via a speech recognition system. A Sulawesi speech recognition sensor has been implemented and allows sentences to be transferred into the system. The speech recognition sensor uses IBM's ViaVoice software and the Java Speech API to translate the spoken words and sentences into commands that Sulawesi can understand.

Once the sentence has been entered into the system, Sulawesi processes the command and an agent produces a response. If the results are sent to the text renderer, they are displayed in the main panel of the Gili interface. This can be seen in figure 6.8, where the sentence "please show me the time" produces "the time is 2:36" via the text renderer in the Gili main panel. The code in appendix C.1 describes the *reactionary* time agent. The basic code is similar to the example code give in section 5.4.5, with the construction and registration of the application defined in lines 19-28. The main processing part of the application is specified in lines 32-59 inclusively, here the application retrieves the time from the system (line 34), formats the time for the correct timezone (lines 36-41), strips off the trailing seconds (lines 47-50). Once this past this stage the representation of the time is in a sufficient format for rendering, here the application requests the rendition of the time (line 54) and finally removes the registration from the *ServiceManager* (line 58).

6.5 Location I.A.L

The task of producing the location context for an application is explored here. The raw longitude, latitude and altitude (LLA) coordinates from the GPS receiver can be stored by any application receiving messages from Sulawesi, such as a *note taking* application. In this example when a note is saved it stores the position of the user. Recalling the note would reveal the location that the note was first stored, this may be useful in triggering the users memory into remembering the situation when the note was saved. This is similar in concept to the work of [37].

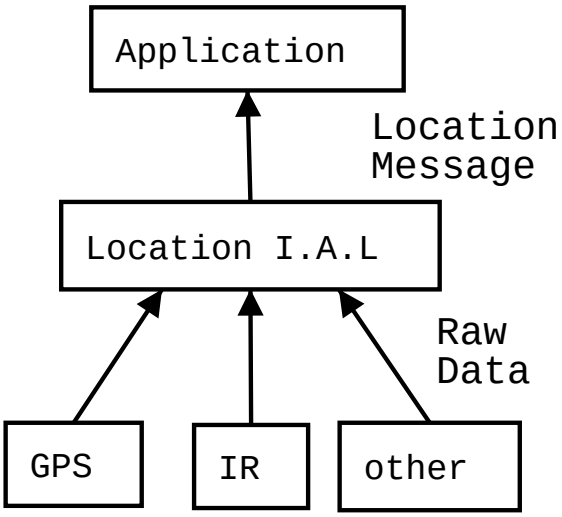


Figure 6.9: Location I.A.L.

It may be almost impossible for a user to remember where on the planet a particular LLA position corresponds to. If the LLA information is abstracted into a user definable place name it is easier for the user to understand and relate to. For example, if the note application revealed that a note was stored at *University of Essex* this may be a bit easier to understood rather than if the note was stored at *LLA point* $52.5^o, 0.34^o, 3.4^o$

The location I.A.L. is responsible for providing seamless location messages to applications. Inside a building the IR sensors are used to locate the user, and outdoors the GPS sensor is used for positional information. The location I.A.L.

can accommodate as many location sensors as are needed. This can increase the accuracy of the positional information available, but it will also increase the complexity of the internal logic used in determining where the user is.

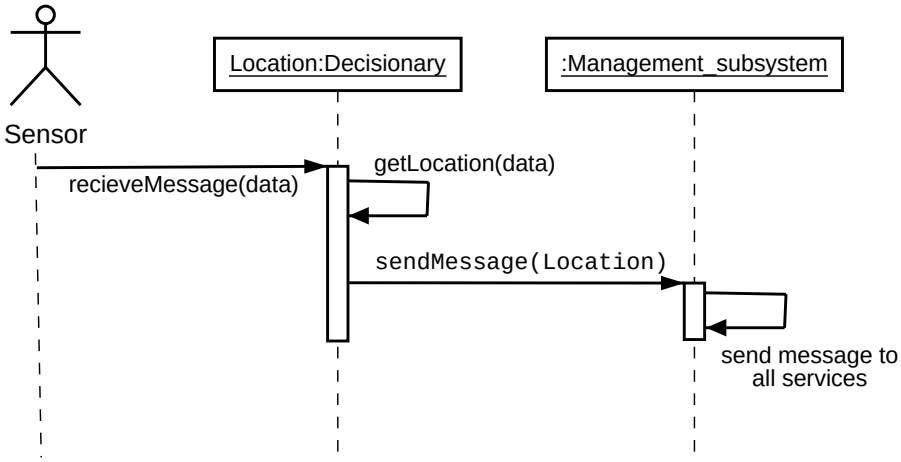


Figure 6.10: Location I.A.L. sequence diagram.

The location I.A.L. transparently decides which signal should be translated. If the user is inside a building, the GPS signal is lost and the sensor stops producing coordinate data. When this happens, and messages are received from the IR sensor, the location I.A.L. uses the IR data to generate the desired location messages. These messages are then broadcast to all the applications within Sulawesi. The converse is true when the IR signal disappears and the GPS signal is received: the location I.A.L. generates location messages based on the GPS signal.

Once the location I.A.L. has received data messages from a sensor, it transforms that data into an abstracted location name via the use of a look-up table. In this implementation each sensor has a separate look-up table and once the translations have occurred the Location I.A.L sends a location message to the Sulawesi system. The location messages contain a simple string. The name of the I.A.L is included to enable an application to identify where the message came from. A example message from the Location I.A.L would contain the

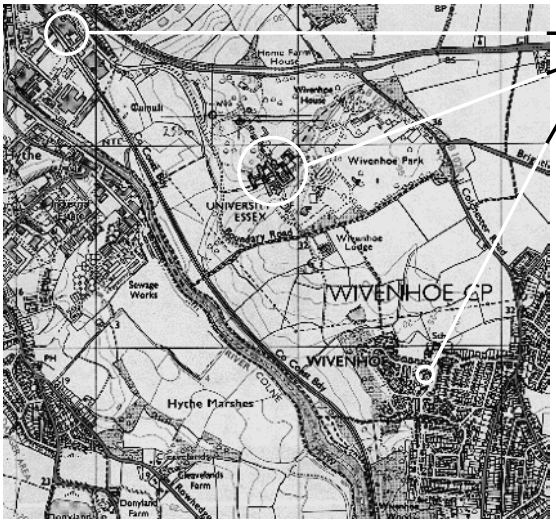
string "Location: University of Essex". Here, the colon is used to delimit the name of the application that sent the message from the physical location.

In this case it is clear that the decision logic to determine which sensor is producing correct location messages is fairly trivial as the two sensors are effectively mutually exclusive. More advanced sensors and complex decisions can be made if the errors from each sensor are available.

This mechanism of choosing the right sensor at the right time enables devices to appear and disappear without any disruption in the location messages as far as the applications are concerned. Also, the design and implementation of the I.A.L. allows other forms of positional information such as pico-cellular devices [67], RF id tags, mobile phone cells, or even a visual identification marking system such as fiducals [74] to be easily incorporated into the system.

6.5.1 GPS location translation

Each of the sensors send raw coordinate data to the location I.A.L. If the data are from the GPS receiver, a translation table is used to convert the LLA point to a more meaningful place name. The translation table contains the names of known locations which have been stored beforehand along with their LLA positions. The distances to the various places are calculated and if the location of the user is within the defined radius of a particular location then it is flagged as the location of the user. The name for the location is then used as the abstract place name. A graphical representation of these translations can be seen in figure 6.11. The translation table is stored in an easily configurable text file. An entry from the file consists of an identifying number, the LLA coordinates, the place name and the cell radius (kilometres) from the LLA point (degrees).



ID	LLA (degrees)	Name	Radius (km)
1	0.94 51.87 9.73	University of Essex	0.7
2	0.23 52.74 9.73	The Shops	0.5
3	1.53 49.83 9.73	Home	0.1

Figure 6.11: Map showing locations and cell radii.

6.5.2 Infra-Red Location Translation

When the user enters a location with a IR beacon, a handshake transfers the unique identification number to the receiver. This ID is then looked up in a similar table used for the GPS translations. As can be seen below, an entry in the file consists of an identifying number (for internal indexing), the unique location ID and the abstract place name.

```
|1|55|Vase Lab|
```

Once the location has been translated, the location I.A.L. broadcasts a location message containing the place name to the applications within the Sulawesi system. By using simple text files for the translation tables, the user can modify and allocate different place names to the locations.

6.6 Posture I.A.L

The posture I.A.L. works in a similar way to the location I.A.L. in that it abstracts the contexts of walking, sitting down and standing up from accelerometer

data. The contexts are determined by analysing and recognising various patterns in the accelerometer data. As mentioned in section 6.2.3, the sensor is placed on the upper part of the leg (see figure 6.3) and produces X and Y data streams with patterns that vary depending on the user's posture. The sequence diagram for the Posture I.A.L can be seen in figure 6.12.

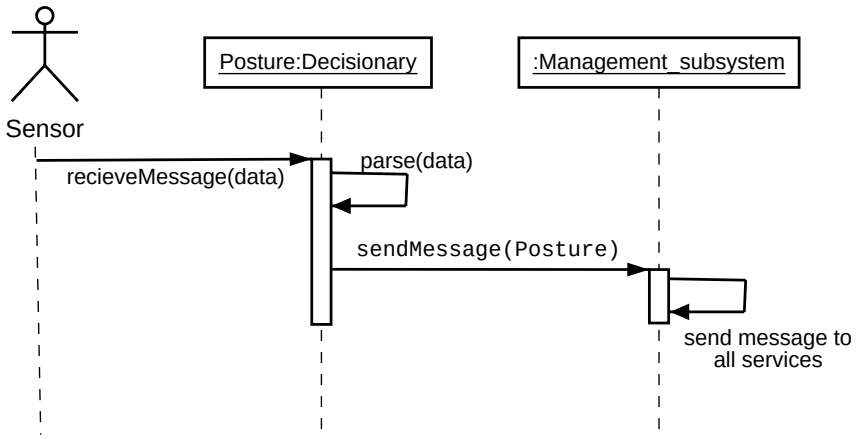


Figure 6.12: Posture I.A.L. sequence diagram.

As can be seen in figure 6.13, the X and Y traces in the top image are produced when the user is sitting down. When the user stands up the traces go through a transitional period and settle with the X and Y traces in the opposite position when compared to the *seated* trace. The orientation of the sensor is used to determine whether the user is standing up or sitting down, but one more important piece of information is inferred when the user is standing up.

The *standing up* context is further analysed by taking the Fourier transform of the data. If the user is standing up, the amplitude of the Fourier transform reveals a relatively small amount of power in the signal. But if the user starts to walk, the amount of power present in the signal increases. The X and Y traces in the bottom image of figure 6.13 show the variation in the signal as the user walks about.

The difference in power between the *standing up* signal and *walking* signal

is quite large and a simple threshold is used to determine whether the user is standing up or walking. This calculation is performed once a second and the I.A.L. sends posture messages containing the context to all services within the Sulawesi framework.

The posture messages contain a simple string. The name of the I.A.L is included to enable an application to identify where the message came from. A example message from the Posture I.A.L would contain the string "Posture: standing up". Here, the colon is used to delimit the name of the application that sent the message from the user's posture.

As can be seen from appendix C.2, the class definition on line 7 states that this is a *decisionary* application. The constructor (lines 24-32) handles the registration of the service with the *ServiceManager*. After this, the application waits until a message is received via the `recieveMessage` method (line 50). If the message starts with the string "ADXL" (meaning it is from the accelerometer code, see line 52) the message is parsed (line 56) and the correct posture is broadcast around the Sulawesi system (line 59). Also at this stage, if the posture message is a *standing up* message, then the posture I.A.L. diverts any textual responses to a spoken renderer (see line 63). If the recieveMessage message does not start with the "ADXL" string, then the current posture perception is output (see lines 70-89).

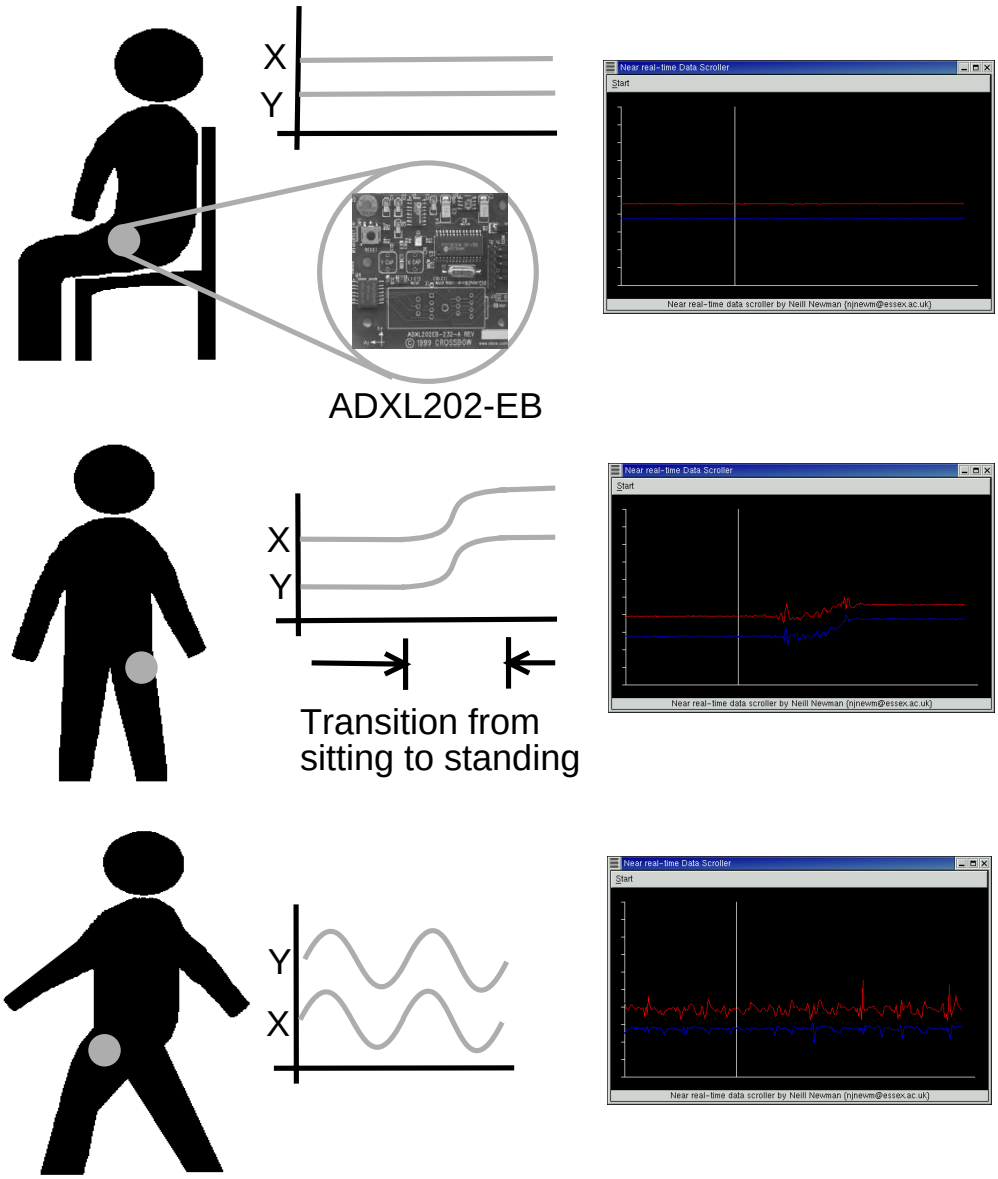


Figure 6.13: Diagram of accelerometer data when sitting, standing and walking.

6.6.1 Contextual rendering based on posture

The simple accelerometer that gathers information about the user's posture may be interesting from an academic point of view, but it is not very useful for the end user to be told whether they are standing up or sitting down! However, the information about the user's posture can be used by the Sulawesi system to determine how best to render information to the user. If the simple assumption

that *when a user is walking it would be a bad idea to produce any visual distractions* is taken into consideration, the system can be set up to divert any visual responses to an audible rendition automatically.

The system implemented means that, when the user is standing up or walking, the posture I.A.L. contacts the contextual renderer and requests that any visual responses are to be redirected to an audible rendition. It also contacts the Gili interface and requests that the main panel should be hidden.

In practice this results in the following functionality: if a user is sitting down and asks to be *shown the sports news*, the framework renders the information as text via the Gili interface as expected. Conversely, if the user is walking and asks to be *shown the sports news*, (see section 6.7.1) the contextual renderer will divert the visual response to an audible rendition. This way Sulawesi appears to try not to distract the user from their primary task.

6.7 Applications

6.7.1 News

The News service demonstrates Sulawesi's sentence processing and output redirection mechanisms. The service is a *reactionary* agent and only responds when it receives a command such as *"could you show me the news, the slashdot headlines please"*. The command is parsed by Sulawesi which extracts the rendition type "show" and the service to invoke "news". The rest of the sentence after the "news" keyword is then placed in the service arguments section of the command buffer. The news service retrieves it's arguments from the command buffer and, in this example, gathers the headlines from the slashdot (an technology based Internet portal) news feed. The headlines are then parsed and the results are sent to the contextual renderer.

The News service uses a configuration file called *news.cfg* to define the location of the news feed file. An example of an entry from *news.cfg* can be seen

below:

|slashdot|http://www.slashdot.org/slashdot.rdf|

The URL in the configuration file refers to an XML based RDF[7] file, and a sample file from the slashdot feed can be seen in appendix C.3.

As seen in the class definition on line 8 in appendix C.4, the News application is a *reactionary* service. The application is constructed and registers itself on lines 28-37. Once registered, the *ServiceManager* calls the `startProcessing` method on line 41. This method reads in the configuration file (line 44), this calls the `createNewsArray` (lines 111-154) method that creates an array containing the name/URL pairs found in the *news.cfg* file. Next the array is checked to see if the arguments to this service (held in `batch[2]` on line 56) match the name section of the array. If a match is found then the URL is stored as the particular news feed (line 57) to be downloaded. The site is then searched by calling the `SearchNewsSite` method on line 62. The retrieval and rendition request is handled in the `SearchNewsSite` method (lines 68-107), where the RDF file is retrieved (line 80) and parsed (line 90). The results are then sent to the *ServiceManager* to be rendered (line 106).

At the time of writing this type of file is used by thousands of Internet sites for publishing news headlines. When the news service is invoked, the relevant information is retrieved from the Internet. The information is parsed and the current headlines are sent to the contextual renderer. This results in the correct information being relayed to the user in an appropriate format depending on what the user is doing.

6.7.2 Spatial reminders

The chronological schedulers found in nearly all electronic diaries and PDAs provide reminders which are triggered by the date and time. This is particularly

[7] `http://www.w3.org/RDF/index.html`

useful in that the machine *volunteers* information when it detects a triggering event. In a similar way, the *Spatial Reminder* is designed to provide location-specific reminders which are triggered by the spatial position of the user. This requires that the application constantly monitors the user's location and makes decisions based on its perception of the user's position. For this reason the Spatial Reminder is implemented as a *decisionary* application to allow it to constantly monitor the user's location.

Figure 6.14: Spatial reminder after being triggered by a location.

The Spatial Reminders can be used for many different tasks, but the author originally wrote the software with an autonomous shopping list reminder in mind. A Spatial Reminder is set using a simple phrase such as *When I go to the shops could you remind me to get some bread and milk.* The location I.A.L. provides the abstract place name and when the user enters the area of *the shops* the Spatial Reminder sends the reminder to the contextual rendering stage, resulting in the information being relayed to the user (see figure 6.14).

The Spatial Reminder code, found in appendix C.5, is a decisionary application (as seen in the class definition on line 8). The application is constructed and registers itself on lines 32-44. On construction the application calls the `setupReminder` method (line 43) which initialises an array of abstract place names and absolute locations (lines 164-223). Once this has happened the

Spatial Reminder waits until a message is passed into the application via the `recieveMessage` method on lines 54-78. If the message is a command from the user, *such as remind me to get some bread from the shops*, the startProcessing method is called from line 76. This method stores the reminder and location name into a hash table (lines 88 - 115). The other message which the Spatial Reminder responds to are generated by the *Location I.A.L.* and when these messages are received, the reminders are checked by calling the `checkReminders` method on line 72. This checks the location specified by the Location I.A.L. messages with the hash tables, if a match is found (line 141) then a response is generated (line 144) and it is sent to the *ServiceManager* to be rendered (line 163).

6.7.3 Notes application

The notes application provides a user with the ability to compose, save and restore previous notes. The functionality required by the notes application is more complex than the other applications so far described, and has therefore been integrated with the Gili API. This has given the author an idea as to what application support is needed to control the application via alternative methods such as speech. The source code for the Notes application can be seen below.

The graphical control of the Notes application is manipulated through the Gili interface, and this is why the source code is considerably longer than the previous source code listings. The application code (found in appendix C.6) overrides three buttons when the APP MENU is selected on the Gili interface. These three buttons are defined on lines 220 – 231 inclusively. The application implements the `userButtonXpushed` interfaces starting on lines 236, 269 and 315. This allows the events from the interface to be passed through to the application. In the notes application these buttons refer to the SAVE, QUERY, and LIST buttons seen in figure 6.6. The SAVE method stores the currently entered note to a file on the disk, and the LIST method lists the stored notes.

The Notes application has also been integrated with the Remembrance Agent [13] and when a note is stored it is indexed by the RA, this can be seen on lines 254 and 338. The QUERY method (lines 266 – 308 and 376 – 405) allows the previously saved notes to be searched by querying the RA, with the results being sent to the relevant rendition mechanism (see lines 306 and 405).

The Notes application also provides a speech controlled interface. Through the Sulawesi system, the application can receive a message containing a command. This command is processed in the *recieveMessage*/*parseMessage* methods on lines 69 and 76. The keywords save, query and list are searched for, and the *save*, *query* and *list* methods on lines 330, 357 and 374 are invoked. When these methods have completed they return their results to the service manager for rendering, rather than directly to the Gili interface.

6.8 Chapter summary

The use of an adaptable software framework has assisted in reducing the development time associated with creating an alternative contextual user interface and applications, and it also ensures consistency, essential in a user interface system. While the original development of Sulawesi took many months, the applications and sensors discussed in this chapter each took less than a week to develop.

The use of multiple sensors and the I.A.L. allows a *hybrid* sensor to be developed independently of any other systems and forces the developer to focus their concentration on the single component, message passing and documentation of the sensor. The Location I.A.L. demonstrates a seamless indoors/outdoors hybrid sensor providing the context of location and, although crude in its implementation, the example highlights the benefits of a system which can make decisions based on several different physical sensors.

The use of an accelerometer allows the author to investigate the potential

uses of a mobile contextual user interface. Although evidence is not included here, initial observations suggest that giving the machine a basic understanding about what the user is doing can have an effect on how the user perceives the user interface. If the wearable computer is to become ubiquitous in everyday life it must be able to adapt to the users situations. For example, if a mobile phone had enough "intelligence" to know when a user is talking to somebody, it may be seen as "more intelligent" if it muted the annoying beeps and vibrated instead, diverted the call to an answer phone service, or even answered the call and said "Fred is talking to somebody, can you either leave a message or phone back in 5 minutes". This type of intelligent user interface can be developed easily using the Sulawesi architecture as in the previous example where the GUI is automatically muted, all visual requests are diverted to an audible response when the user starts walking. The benefit here is that when the user is walking, the display is "muted" so as to not distract them. The author believes that this is a crucial part of any mobile user interface: it should assume that the users primary task is *not* controlling the machine.

The Gili user interface has been designed around the limited cursor control available with the Twiddler, and the menu buttons have been made as large as possible without taking up too much screen real estate. The main panel of the Gili interface allows applications to take control of this area, but they are still under the control of Gili and, if an I.A.L decides the visual display should not be viewable, the main panel can be visually muted. This allows the user to see through the main panel and focus on their primary task.

The notes application provides an insight into how an application can be integrated with the Gili interface, and also how a third-party application, such as the Remembrance Agent, can be incorporated into the Sulawesi architecture. It also demonstrates how a simple application within Sulawesi can tie into other information sources, such as location, to gather addition information about where the note was stored.

The last application is the Spatial Reminder application which can alert the user when a certain location is entered or exited. The system also demonstrates the use of agents providing information to the user autonomously. While no quantitative claims are made, the feedback from people who have seen Sulawesi in action has been sufficient to convince the author to speculate that this approach has much promise.

This chapter also demonstrates how some of the applications are controlled by the careful construction of seemingly natural language sentences. It is speculated that this type of command may be easier for the user to remember. The author also feels that a coherent well-documented system will aid in the research and development of applications/user interfaces for alternative platforms.

Chapter 7

Conclusions & Further Work

7.1 A critical appraisal of Sulawesi

The aim of this research was to explore context-aware and multi-modal issues, because of this the individual components of the system are fairly simple in their implementation. The interfaces and applications provided by Sulawesi are by no means fully developed, and the author believes that a significant amount of work is still needed in the area of contextual interfaces for mobiles devices. Also, the author feels that, while the design of the Sulawesi architecture includes enough features to demonstrate simple prototypes and the internal architecture is fairly solid, it is clear that there are features of the system that had been overlooked in the implementation.

7.1.1 The overall architecture

The basic premise that messages within the architecture would propagate without delay and in the correct order may limit the ability to scale the system to a large number of sensors/renderers. This may be resolved by placing time-stamps within the messages, in a similar way to sequence numbers on Internet Protocol (IP) packets. This would increase the complexity of the *management subsystem* as the message buffer will need to store up message requests, sort them based

on time stamps, and periodically send them in the correct order.

Another problem with the message passing system is due to the use of strings as messages. This means that the system is not as efficient as it could be. The author acknowledges that this problem will also affect the ability to scale the system, and proposes that a more efficient form of message passing, perhaps using binary representations, should be explored in any further work.

The *semi-natural* language processing capabilities also leave a lot to be desired. If any speech processing or natural language experts read this document I am sure they will cringe at the way in which Sulawesi decodes the sentences. There are obvious limitations with the way applications can be controlled using these types of single-sentence commands. For example if a note was to be stored via the speech input mechanism, a single sentence might not be sufficient to distinguish between the note to save and the command to save it. The author proposes that there would need to be a specific word/command to switch between dictation and command modes in order to save the dictation.

These common speech recognition and dialogue processing questions are not currently addressed or implemented in this work, although these issues are currently being looked at by various research groups.

The method of passing commands in the *batch* array could be improved by providing a method interface to the data. It is also believed that the method interface would increase the legibility of the source code.

7.1.2 Input

The Speech input component of Sulawesi is less than satisfactory. At the time of writing the speech input code, the IBM speech recognition software available for Linux was a little difficult to get working reliably. The work in getting reliable speech recognition code was postponed due to too much time being taken in trying to get speech recognition working reliably. In the two or so years since the code was written, the processing power available and the maturity of the

speech recognition software leads the author to believe that there are now more robust systems and APIs available for the speech recognition development, and the speech input component of Sulawesi may be out of date and clumsy in its implementation.

While the author speculates that in most circumstances the use of speech is preferable, there are some situations where using the Twiddler may be advantageous such as typing a command when in a library.

7.1.3 Renderers

Context switching depending on the user's circumstances opens up a whole plethora of questions. Which contexts would be suitable for a particular rendition type is open to debate: the author believes that different people will use different renditions in different circumstances, and therefore a predefined collection of context types and situations may not be achievable.

However, the author considers that there are certain situations where a particular type of rendition is not acceptable. The thesis commonly refers to a visual rendition being unacceptable when a person is driving a car. Perhaps instead of defining which rendition types are *suitable* for situations, the converse should be explored.

There will be certain application rendition types which will be impossible to redirect to an alternative rendition type, such as a visual map being redirected to a speech rendition.

The author is unsure how to handle some of these eventualities, should the map display be paused until a later time, or should a crude and possibly inaccurate conversion to an alternative rendition type be applied to give the user the information at the right time?

7.1.4 Applications

The design of applications currently uses one form of input modality at a time, i.e., if one starts to use a visual input stream to control the application, then one must continue to use that input stream until the application has completed the task it was asked to do. But the core of the Sulawesi architecture uses several streams of information to determine how the information should be presented to the user, such as the posture IAL (in section 6.6) altering the rendering subsystem.

Ideally the application should be controllable from several different streams and should be able to switch from a spoken input stream to a visual one on the fly. Although this sounds complex, the author believes that it is not a difficult task if the decision and tracking logic involved for the application are determined beforehand.

The spatial reminder currently reads the *location.gps* file to determine place names. This should really ask the Location I.A.L for these names, and an API for exporting the names should have been coded. At the moment it works, but it is not as extendible as it could be.

7.1.5 User interface

While the Gili interface provides some basic forms of feedback, such as providing visual feedback for a spoken input. The author believes that although this basic feedback works, it is by no means the best solution. A better solution may be to repeat the command, and ask for confirmation from the user to proceed with the command. The author feels that a lot more work, including user trials would be required to make this system usable.

7.2 Conclusions

Van Dam provides an overview of the generations of human computer interface design over the last fifty years [6], concluding that Human-Computer Interaction (HCI) research stands still for long periods of time and is interrupted by rapid changes. He also predicts that we are now approaching the "post-wimp" phase in which speech recognition, natural language processing, smart agents, machine vision and multi-modal systems will be the next major advance in the human computer interaction systems. This work has investigated some of these issues and has lead to the implementation of a multimodal framework systems of the Sulawesi system. The system attempts to address the lack of any predefined software framework and promotes the use of re-usable code through a well defined object hierarchy. While the system does not explicitly investigate natural language, machine vision or speech recognition, several examples are outlined which provide an insight into how proactive contextual aware information agents can be designed and used.

Unfortunately there are only a few people who are actively looking into alternative user interfaces for mobile systems, and this leads to a lack of any clearly defined development framework in which to experiment. The author has identified this as being a major deficiency in the field, and the Sulawesi system has focused on providing an architecture which may be used to investigate alternative application and user interface systems, from contextually aware mobile applications to intelligent agents that respond to a mobile user's requests.

The software has been released on the Internet for the past year or so, and already the author has received over 200 download requests, with people using the software to experiment with things that the author had never imagined. The most bizarre, yet at the same time very interesting, use was by a researcher in Florida who said he was going to use Sulawesi with a special sensor to detect and decode the clicks generated by a dolphin! A simple application would then be

controlled by the animal and a response generated through an underwater sonic rendition device. This single piece of work, while only described in an email, has lead the author to feel that the goal of Sulawesi, to provide an adaptable software framework for alternative applications, has succeeded.

At the start of the tests on the various wearable user interface devices, it was clear that these devices were slower and more difficult in controlling any kind of user interface. But until now there have been no formal evaluations of the Twiddler in combination with a head-mounted display device. In the analysis of the results the author has observed the possibility of binocular rivalry, which has also been observed by [15]. This will have an effect on how the future wearable/portable head-mounted displays will interact with future user interfaces.

The author has also determined that the amount of control available with the Twiddler device is not adequate for controlling a desktop user interface. The design and integration of the trackpoint [71] in the Twiddler2 suggests that HandyKey has identified that the liquid sensor within the device was inadequate. The research by IBM[1] leads the author to speculate that the trackpoint will increase the usability of this device.

7.3 Future work

Investigations into how a user interface should interact with the user in a mobile environment needs much more exploration. The author has applied his own set of guidelines and instincts as to how he would like the user interface to perform and has applied these to the Sulawesi framework, but these may not be the same issues striven for by others.

The author also believes that the wearable computer will have a transitory existence: they will soon evolve into something different, such as a next-generation mobile phone, or a super-PDA. However, the physical constraints

[1] `http://www.almaden.ibm.com/cs/user/tp/tp.html`

and modes of use of these devices mean that the conventional desktop user interfaces are inappropriate due to the amount of control needed to manipulate a graphical selection device and manipulate graphical objects. Multimodal user interfaces may be the best long term solution and by adapting the interface through knowledge of the environment and the users context may give further improvements. The issues involving speech recognition and natural language processing are being actively researched today. But the unresolved issues surrounding the understanding and grouping of contexts to certain situations, the development of applications that can be manipulated by several different interface devices and switched transparently, and the testing of such systems, involves a huge amount of work which is not currently being looked into.

Bibliography

[1] A. Black and P. Taylor. The architecture of the Festival speech synthesis system. In *The Third ESCA Workshop in Speech Synthesis*, pages 147–151, 1998.

[2] A. C. Downton, G. Leedham, P. Barnard, P. Johnson, H. Johnson, P. Jones, S. Jones, B. Anderson, P. Boucherat, and G. Ashworth. *Engineering the human-computer interface.* McGraw-Hill, 1992.

[3] A. Cheyer and L. Julia. Multimodal maps: an agent based approach. In *International conference on cooperative multimodal communication*, May 1995.

[4] A. Dey and G. Abowd. The Context Toolkit: aiding the development of context-aware applications. In *22nd International Conference on Software Engineering*, 6 June 2000.

[5] A. Smailagic and D.P. Siewiorek. The CMU mobile computers: a new generation of computing systems. In *International computer conference*, pages 467–473. IEEE, 4 March 1994.

[6] A. Van Dam. Post-wimp user interfaces: the human connection. In *Communications of the ACM*, volume 40,2, pages 63–67. ACM, February 1997.

[7] A. Vardy, J. Robinson, and L-T. Cheng. The wristcam as input device. In *International symposium on wearable computers.* IEEE, October 2000.

[8] Albrecht Schmidt and Hans-W. Gellersen. Enabling implicit human computer interaction: a wearable RFID-tag reader. In *4th international symposium on wearable computers*. IEEE, October 2000.

[9] A.R. Revels, L.L. Quil, D.E. Kancler, and B.L. Masquelier. Human interaction with wearable computer systems: a look at glasses-mounted displays. In *Conference on cockpit displays V: Displays for defense applications*. SPIE, April 1998.

[10] B. Crabtree and B. Rhodes. Wearable computing and the remembrance agent. *BT Technology Journal Vol: 16 No:3*, July 1998.

[11] B. Smith, L. Bass, and J. Siegel. On site maintenance using a wearable computer system. *ACM, CHI*, pages 119–120, 1997.

[12] B. Thomas, S. Tyerman, and K. Grimmer. Evaluation of three input mechanisms for wearable computers. *Proceedings of the First International Symposium on Wearable Computers*, 1997.

[13] B.J Rhodes. The wearable remembrance agent: a system for augumented memory. In *Personal technologies journal special issue on wearable computing*, volume 1, page 218:224. Personal Technologies, 1997.

[14] B.J. Rhodes. Wimp interface considered fatal. In *Virtual reality annual international symposium*. IEEE, 15 March 1998.

[15] C. Baber, D. Haniff, L. Cooper, J. Knight, and B. Mellor. Preliminary investigations into the human-factors of wearable computers. In Hillary Johnson, Laurence Nigay, and Chris Roast, editors, *People and computers XIII: Proceedings of HCI'98*. Springer-Verlag, 1998.

[16] C. Esposito. Wearable Computers: Field-Test Results and System Design Guidelines. In *Interact*, July 1997.

[17] C. Faure and L. Julia. An agent-based architecture for a multimodal interface. In *Proceedings of the twelfth national conference on artificial intelligence*. American association for artifical inteligence, August 1994.

[18] Cliff Randell and Henk Muller. Context awareness by analyzing accelerometer data. In *4th international symposium on wearable computers*. IEEE, October 2000.

[19] D. Roy and A. Pentland. Multimodal adaptive interfaces. In *1st workshop on perceptual user interfaces*. ACM, October 1997.

[20] D.B.Moran, A.J.Cheyer, L.E.Julia, D.L.Martin, and S.Park. Multimodal user interfaces in the open agent architecture. In *International conference on intelligent user interfaces*, pages 61–68. ACM, January 1997.

[21] D.J. Johnston. *Augmented Reality for Archaelogical Reconstruction*. PhD thesis, Vase lab, University of Essex, 2002.

[22] D.R. McGee, P.R. Cohen, and S. Oviatt. Confirmation in multimodal systems. In *International joint conference of the association for computational linguistics and the international committee on computational linguistics*. ACL, 1998.

[23] D.R.Benyon. Accommodating individual differences through an adaptive user interface. In M. Schneider-Hufschmidt, T. Kuhme, and U. Malinowski, editors, *Adaptive user interfaces: principles and practice*, pages 149–166. North-holland, Amsterdam, 1993.

[24] D.Salber, A.K.Dey, and G.D.Abowd. A context-based infrastructure for smart environments. In *1st International Workshop on Managing Interactions in Smart Environments*, 1 December 1999.

[25] E.C. Crowe and N.H.Narayanan. Comparing interfaces based on what users watch and do. In *Eye tracking research & applications symposium*, pages 29–36, November 2000.

[26] F. Masaaki and T. Yoshinobu. Body coupled fingering: Wireless wearable keyboard. In *Computer human interaction*. Association for computer machinery, SIGCHI, 26 March 1997.

[27] G. Kortuem. Software architecture and wearable computing. Technical report, University of Oregon, December 1996.

[28] G. Kortuem, M. Bauer, T. Heiber, and Z. Segall. Netman: The Design of a Collaborative Wearable Computer System. In *Journal on Mobile Networks and Applications*, volume 4. ACM/Baltzer, 1999.

[29] G. Kortuem, S. Fickas, and Z. Segall. Architectural issues in supporting ad-hoc collaboration with wearable computers. In *Workshop on software engineering for wearable & pervasive computing. The 22nd international conference on software engineering*, 6 June 2000.

[30] G.A. Thomas, J. Jin T. Niblett, and C. Urquhart. A versatile camera position measurement system for virtual reality TV production. In *International broadcasting convention*, pages 284–289, September 1997.

[31] G.D.Abowd, C.G.Atkeson, J.Hong, S.Long, R.Kooper, and M.Pinkerton. Cyberguide: A mobile context-aware tour guide. In *2nd ACM International conference on Mobile Computing*, 1996. `http://www.cc.gatech.edu/fce/cyberguide/index.html`.

[32] G.L. Calhoun and G.R. McMillan. Hands-free input devices for wearable computers. In *4th symposium on human interaction with complex systems*, 25 March 1998.

[33] H.W. Beadle, B. Harper, G.Q. Maquire Jr, and J. Judge. Location aware mobile computing. In *International conference on telecommunications*. IEEE/IEE, February 1997.

[34] I.E. Sutherland. A head-mounted three-dimensional display. In *American federation of information processing societies*, volume 33, pages 757–764, January 1968.

[35] J. Farringdon, V. Oni, C.M. Kan, and L. Poll. Co-modal browser: an interface for wearable computers. In *The 3rd International Symposium on Wearable Computers*, pages 45–51. IEEE computer society, October 1999.

[36] J. Pascoe. Adding generic contextual capabilities to wearable computers. In *The 2nd international symposium on wearable computers*, pages 92–99. IEEE computer society, October 1998.

[37] J. Pascoe, N.S. Ryan, and D.R. Morse. Human computer giraffe interaction: HCI in the field. In C.Johnson, editor, *Workshop on Human Computer Interaction with Mobile Devices*, May 1998.

[38] J.A. Landay and T.R. Kaufmann. User interface issues in mobile computing. In *4th workshop on workstation operating systems*, October 1993.

[39] J.F. Knight and C. Baber. Wearable computers and the possible development of musculoskeletal disorders. In *4th international symposium on wearable computers*, October 2000.

[40] K. Hartung, S. Munch, L. Schomaker, T. Guiard-Marigny, B.L. Goff, R. MacLaverty, J. Nijtmans, A. Camurri, I. Defee, and C. Benoit. Di3: development of a system architecture for the acquisition, integration and representation of multimodal information. Technical report, Nijmegen institute for cognition & information, March 1996.

[41] K. Nagao and J. Rekimoto. Ubiquitous talker: spoken language interaction with real world objects. In *14th international joint conference on artificial intelligence*, pages 1284–1290, 1995.

[42] Gerd Kortuem, Zary Segal, and Martin Bauer. Context-aware, adaptive wearable computers as remote interfaces to 'intelligent' environments. In *IEE, Second International Symposium on Wearable Computers*. University of Oregon, Oct 1998.

[43] L. Bass, C. Kasabach, R. Martin, D. Siewiork, A. Smailagic, and J. Stivoric. The design of a wearable computer. *ACM CHI*, pages 139–146, 1997.

[44] L. J. Najjar, J. J. Ockerman, and J. C. Thompson. Using a Wearable Computer for Mobile Training and Performance Support. In *Educational Multimedia/Hypermedia and Telecommunications*, page 1461, 1997.

[45] L.J. Najjar, J.J. Ockerman, and J.C. Thompson. User interface design guidelines for wearable computer speech recognition applications. In *Virtual reality annual international symposium*. IEEE, 15 March 1998.

[46] M. Bauer, T. Heiber, G. Kortuem, and Z. Segall. A collaborative wearable system with remote sensing. In *2nd international symposium on wearable computers*. IEEE, October 1998.

[47] M. Goldstein, R. Brook, G. Alsio, and S. Tessa. Ubiquitous input for wearable computing: qwerty keyboard without a board. In Chris Johnson, editor, *1st workshop on human computer interaction with mobile devices*, May 1998.

[48] M. Weiser. Some computer science issues in ubiquitous computing. In *Communications of the ACM*, July 1993.

[49] Mark Priestley and Elizabeth Robinson, editors. *Practical object-oriented design with UML*. McGraw-Hill, 2000.

[50] M.B. Spitzer, N.M. Rensing, R. McClelland, and P. Aquilino. Eyeglass-based systems for wearable computing. In *1st international symposium on wearable computers*, pages 48–51. IEEE, October 1997.

[51] M.D.Wilson, D.Sedlock, J-L. Binot, and P. Falzon. An architecture for multimodal dialogue. In *2nd VENACO workshop on the structure of multimodal dialogue*, 1991.

[52] Douglas B. Moran and Adam J. Cheyer. Intelligent agent-based user interfaces. In *International workshop on Human Interface Technology*, `http://www.ai.sri.com/ moran`, 1995. paper copy.

[53] P. Davidsson, E. Astor, and B. Ekdahl. A framework for autonomous agents based on the concept of anticipatory systems. In R. Trappl, editor, *cybernetics and systems*, pages 1427–1434. world scientific, 1994.

[54] P.J. Brown. Some lessons for location-aware applications. In *Human computer interactions for mobile devices*, May 1998.

[55] P.J. Brown. Triggering information by context. In *Personal technologies*, volume 2. Springer-verlag, September 1998.

[56] P.J. Brown and G.J.F. Jones. Context-aware retrieval: exploring a new environment for information retrieval and information filtering. *Personal and Ubiquitous Computing*, 5(4):253–263, 2001.

[57] P.J. Wyard and G.E. Churcher. All channels open: multimodal human/computer interfaces. In *Technology Journal*, volume 18. British Telecommunications, January 2000.

[58] P.M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. In *Journal of experimental psychology*, volume 47, pages 381–391. American psychological association, 1954.

[59] R. Bolt. *The Human Interface: Where People and Computers Meet.* Lifetime learning publications, 1984.

[60] Randy Pausch, M. Anne Shackelford, and Dennis Proffitt. A user study comparing head-mounted and stationary displays. In *Symposium on research frontiers in virtual reality*, pages 41–45. IEEE, 1993.

[61] Deb Roy, Nitin Sawhney, Chris Schmandt, and Alex Pentland. Wearable audio computing: A survey of interaction techniques. Technical report, MIT Media Lab, 1997.

[62] S. Feiner, B. MacIntyre, T. Höllerer, and T. Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. In *First Int. Symp. on Wearable Computers*, 13 October 1997.

[63] S. Mann. 'Smart clothing': wearable multimedia computing and 'personal imaging' to restore the technological balance between people and their environments. In *Multimedia*. ACM, 1996.

[64] S. Mann. An historical account of the wearcomp project. In *1st international symposium on wearable computers*. IEEE, October 1997.

[65] S. Mann. Wearable computing as a means for personal empowerment. Technical report, University of Toronto, `http://www.wearcomp.org/wearcompdef.html`, 1998.

[66] S. McGlashan. Towards multimodal dialogue management. In *12th twente workshop on language technology*, September 1996.

[67] S. Narayanaswamy, S. Seshan, E. Brewer, R. Brodersen, F. Burghardt, A. Burstein, Y. Chang, A. Fox, J.M. Gilbert, R. Han, R.H. Katz, A.C. Long, D.G. Messerschmitt, and J. Rabaey. Application and network support for Infopad. *IEEE Personal Communications*, March 1996.

[68] S. Oviatt. Multimodal interfaces for dynamic interactive maps. In *Human factors in computing systems: CHI'96*, pages 95–102. ACM, 1996.

[69] T. Starner, B. Schiele, and A. Pentland. Visual contextual awareness in wearable computing. In *2nd international symposium of wearable computers*. IEEE, October 1998.

[70] T. Starner, D. Kirsh, and S. Assefa. The locust swarm: An environmentally-powered, networkless location and messaging system. In *1st international symposium on wearable computing*. IEEE, October 1997.

[71] Ted Selker and Joe Rutledge. Trackpoint II: the in-keyboard pointing device. *IBM personal systems technical solutions*, January 1993.

[72] Thomas A. Bass. *The eudaemonic pie*. Vintage books, 1985.

[73] W. Mayol, E. Rodriguez, L.A.F Hernández, and V.T. Rangel. The WearClam. textt-thttp://www.robots.ox.ac.uk/ wmayol/WearClam/index.html.

[74] W.A. Hoff, K. Nguyen, and T. Lyon. Computer vision-based registration techniques for augmented reality. In *Intelligent Robots and Computer Vision XV*, volume 2904, pages 538–548. SPIE, November 1996.

[75] Z. Segall and M. Curry. Wearable computing research summary. Technical report, University of Oregon, January 1996.

Appendix A

User Test Paragraphs

A.1 Paragraph One

The BBC will die if it does not get funding to expand in the digital age bosses have warned. Director of television Alan Yentob said the corporation must develop new services which already include continuous news Internet sites and an education channel to cater for digital viewers new ways of watching. Put the BBC in a box and that box will soon become a coffin and the BBC will wither and die he told the Royal Television Society conference in Cambridge. An independent panel led by economist Gavyn Davies has proposed that there should be an annual tax to fund the BBCs fledgling digital TV service.

A.2 Paragraph Two

Four Russian referees are facing the most severe measures after they were barred from officiating a Uefa Cup match because they were too drunk. Sergei Khusainov, Sergei Martynov, Pavel Ginzburg and Feizudin Erzimanov were allegedly drinking heavily before Thursday evenings encounter between Hapoel Haifa Israel and FC Brugge Belgium. According to the director of the Russian Soccer Union, Alexander Tukmanov local football officials told the referees they would not be allowed to take control of the match which the Israelis won.

Appendix B

Interface Code

B.1 Example reactionary interface code

Listing B.1: Example Reactionary Interface Code

```
1  package sulawesi.services;
2
3  public class time extends Thread implements servicereaction{
4
5    // a reference back to the service manager
6    private servicemanager Manager = null;
7    private Integer threadPriority = null;
8
9    // create an empty buffer for the service manager to fill
10   private String[] batch = new String[5];
11
12   // Constructor registers itself with the service manager
13   public time(Object creator, Integer priority){
14     Manager = (servicemanager)creator;
15     threadPriority = priority;
16
17     // Register with the service manager.
18     Manager.register("time",this, batch, serviceType);
19
```

```
20      // Set the Priority that the service manager gives us.
21      this.setPriority(priority.intValue());
22
23    }
24
25    // The startProcessing method is called by the
          Servicemanager.
26    public void startProcessing() {
27
28      // Create a date object and set the time zone to GMT.
29      Date date = new Date();
30      DateFormat dateFmt = DateFormat.getTimeInstance();
31      TimeZone zone = TimeZone.getDefault();
32      zone.setID("GMT");
33
34      dateFmt.setTimeZone(zone);
35      zone = dateFmt.getTimeZone();
36
37      // Gets the formatted time.
38      String out = dateFmt.format(date);
39
40      // Just send the time to the output renderer
41      // the command array is sent as well so the
42      // render type can be determined.
43      Manager.out.renderOutput("the time is " + out , batch);
44
45      // When processing has finished.. remove registration.
46      // Remove the class from the servicemanager, batch[0] = ID
47      Manager.remove(batch[0], this);
48    }
49  }
```

B.2 Example decisionary interface code

Listing B.2: Example Decisionary Interface Code

```
1  package sulawesi.services;
2
3  public class wibble extends Thread implements servicedecision
       , Serializable{
4
5    private servicemanager Manager = null;
6    private int threadPriority;
7
8    // create an empty buffer for the service manager to fill
9    private String[] batch = new String[5];
10
11   // Constructor registers itself with the service manager
12   public location(Object creator, Integer priority){
13     Manager = (servicemanager)creator;
14     threadPriority = priority.intValue();
15
16     // Set the thread priority and register
17     this.setPriority(priority.intValue());
18     Manager.register(ClassName(),this, batch, serviceType);
19   }
20
21   // returns the buffer for this service
22   public String[] getBatch(){
23     return batch;
24   }
25
26   // definition of className from servicedecision interface
27   public String ClassName(){
28     return "wibble";
29   }
30
31   // process messages here
```

```
32    public void recieveMessage(Object Obj, String Message){
33    }
34
35    // the startProcessing method is called by the
          Servicemanager
36    public void startProcessing() {
37    }
38
39    // when Sulawesi is shutdown you MUST include this code
40    // to de-reference the Manager object.
41    private void writeObject(ObjectOutputStream out) throws
          IOException {
42
43      // sets the servicemanager object pointer to null
44      Manager = null;
45
46      // calls the default write on this object
47      out.defaultWriteObject();
48    }
49  }
```

B.3 Example sensor code

Listing B.3: Example Sensor Code

```
1  package sulawesi.input;
2  import sulawesi.decision.input;
3
4  public class foo implements sulawesi.base.misc.inputbase{
5
6          private input ip;
7
8          // Constructor for the foo object.
9          public foo(Object creator){
10                 // register back with the input subsystem.
11                 ip = (input)creator;
12                 ip.register("foo",this);
13
14                 this.start();
15         }
16
17         // Implementation of the inputbase interface.
18         public Object Query(Object in){
19                 return anObject;
20         }
21
22
23         // Send data to the input subsystem
24         public void run(){
25                 ip.DataIn(ConformingCommand);
26         }
27
28 }
```

B.4 Example renderer code

Listing B.4: Example Renderer Code

```
1  package sulawesi.output;
2  import sulawesi.decision.output;
3
4  public class foo implements sulawesi.base.misc.outputbase{
5
6          private output op;
7
8          // Constructor for the foo object
9          public foo(Object creator){
10                 // register with the Renderer subsystem.
11                 op = (output)creator;
12                 op.register("foo",this);
13         }
14
15         // Implementation of the outputbase interface.
16         public void Pause(){
17                 // pause this renderer here
18         }
19
20         // Implementation of the outputbase interface.
21         public void unPause(){
22                 // unpause this renderer here
23         }
24
25         public void render(String in){
26                 // render the string here
27         }
28 }
```

Appendix C

Application Code

C.1 Time agent code

Listing C.1: Time Agent Code

```
1  package sulawesi.services;
2
3  import java.util.*;
4  import java.text.DateFormat;
5  import sulawesi.decision.servicemanager;
6
7  // This service will send the ascii string for the time
8  // to the ouptut rendering stage
9
10 public class time extends Thread implements servicereaction{
11
12   // a reference back to the service manager
13   private servicemanager Manager = null;
14   private Integer threadPriority = null;
15   // create an empty buffer for the service manager to fill
16   private String[] batch = new String[5];
17
18   // Constructor registers itself with the service manager
19   public time(Object creator, Integer priority){
```

```
20      Manager = (servicemanager)creator;
21      threadPriority = priority;
22
23      // register with the service manager
24      Manager.register("time",this, batch, this.serviceType);
25
26      // set the Priority that the service manager gives us
27      this.setPriority(priority.intValue());
28    }
29
30
31    // this method is called by the service manager
32    public void startProcessing() {
33      // create a date object and set the time zone to GMT...
34      Date date = new Date();
35
36      DateFormat dateFmt = DateFormat.getTimeInstance();
37      TimeZone zone = TimeZone.getDefault();
38      zone.setID("GMT");
39
40      dateFmt.setTimeZone(zone);
41      zone = dateFmt.getTimeZone();
42
43      // gets the formatted time
44      String out = dateFmt.format(date);
45
46      // remove the seconds
47      int index = 0;
48      index = out.indexOf(":");
49      index = out.indexOf(":", index+1);
50      out = out.substring(0, index);
51
52      // just output the time to the output object
53      // the batch array is sent to determine the renderer
54      Manager.out.renderOutput("the time is " + out , batch);
55
56      // when processing has finished, remove registration.
57      // batch[0] = ID
```

```
58        Manager.remove(batch[0], this, this.serviceType);
59      }
60    }
```

C.2 Posture I.A.L code

Listing C.2: Posture I.A.L. Code

```
1  package sulawesi.services;
2
3  import sulawesi.base.misc.*;
4  import sulawesi.decision.servicemanager;
5  import java.io.*;
6
7  public class posture extends Thread implements servicedecision
       , Serializable{
8
9    // a reference back to the service manager
10   private servicemanager Manager = null;
11   private int threadPriority;
12
13   // create a buffer for the servicemanager to fill
14   private String[] batch = new String[5];
15
16   // stores the status of the user
17   private String Status = null;
18
19   // wether to start processing the data
20   private boolean Go = false;
21
22
23   // Constructor registers itself with the service manager
24   public posture(Object creator, Integer priority){
25     Manager = (servicemanager)creator;
26     threadPriority = priority.intValue();
27
28     // set the thread priority
29     this.setPriority(priority.intValue());
30
31     // register with the service manager
```

```
32      Manager.register(ClassName(),this, batch,serviceType);
33    }
34
35
36    // used to return the batch for this object
37    public String[] getBatch(){
38      return batch;
39    }
40
41
42    // interface from servicedecision, does nothing here
43    public void startProcessing() {
44
45    }
46
47
48    // When the adxl input module recieves some data it
49    // broadcasts it, this object recieves the messages here.
50    public void recieveMessage(Object Obj, String Message){
51
52      if(Message.startsWith("ADXL:")){
53
54        // parse the message and
55        // figure out what to do!!!....
56        parse(Message);
57
58        // send the posture to other services
59        Manager.sendMessage(this, "Posture: " + Status);
60
61        // redirect output renders if standing up
62        if(Status.indexOf("standing up") > 0){
63          Manager.out.redirectRenderer("text","speakoutloud");
64        }
65        else{
66          Manager.out.redirectRenderer("text","text");
67        }
68
69      }
```

```
70      else if ((Message.indexOf("posture") > 0)
71              || (Message.indexOf("am i doing") > 0)
72              || (Message.indexOf("i am doing") > 0)){
73
74        // ok get an ID for this service request
75        String Batch = Manager.getProcessBatch(ClassName());
76
77        // ok we have a request, so create a batch
78        // array for this object
79        batch = Manager.search.getBatchFromNoID(batch, Batch);
80
81        // just output the posture to the output object
82        // the batch array is sent as well so the renderer
83        // can be determined
84        Manager.out.renderOutput("you are " + Status , batch);
85
86        // release the batch when done...
87        Manager.releaseBatch(Batch);
88      }
89    }
90
91
92    // the run method does some processing the processing
93    private void parse(String data) {
94
95      // parse the 'ADXL: sitting' into 'sitting'
96      int index = data.indexOf(":");
97
98      // set the status
99      Status = data.substring(index);
100   }
101
102
103   //definition of className from servicedecision interface
104   public String ClassName(){
105     return "posture";
106   }
107
```

```
108
109    //get rid of the servicemanager pointer when serialized
110    private void writeObject(ObjectOutputStream out) throws
           IOException {
111
112      // sets the servicemanager object pointer to null
113      Manager = null;
114
115      // calls the default write on this object
116      out.defaultWriteObject();
117    }
118  }
```

C.3 Slashdot RDF news feed

Listing C.3: Slashdot RDF news feed file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://my.netscape.com/rdf/simple/0.9/">

  <channel>
    <title>
      Slashdot: News for nerds, stuff that matters
    </title>
    <link>
      http://slashdot.org
    </link>
    <description>
      News for nerds, stuff that matters
    </description>
  </channel>

  <image>
    <title>
      Slashdot
    </title>
    <url>
      http://images.slashdot.org/topics/topicslashdot.gif
    </url>
    <link>
      http://slashdot.org
    </link>
  </image>

  <item>
    <title>
      Kick Your Input Device
```

```
33      </title>
34      <link>
35        http://slashdot.org/article.pl?sid=01/07/26/1740257
36      </link>
37    </item>
38
39    <item>
40      <title>
41        Mundie Speech @ OSCON - Blogged In Real Time
42      </title>
43      <link>
44        http://slashdot.org/article.pl?sid=01/07/26/1823233
45      </link>
46    </item>
47
48    <item>
49      <title>
50        Business Wants a New, Profitable Internet
51      </title>
52      <link>
53        http://slashdot.org/article.pl?sid=01/07/26/1553257
54      </link>
55    </item>
56
57  </rdf:RDF>
```

C.4 News agent code

Listing C.4: News Agent Code

```
1  package sulawesi.services;
2
3  import java.io.*;
4  import java.util.*;
5  import sulawesi.base.*;
6  import sulawesi.decision.*;
7
8  public class news extends Thread implements servicereaction{
9
10   // a reference back to the service manager
11   private servicemanager Manager = null;
12   private Integer threadPriority = null;
13
14   // create an empty buffer for the service manager to fill
15   private String[] batch = new String[5];
16   private String NewsConfigFile = "config/news.cfg";
17   private searcher find = new searcher();
18   private String delimiter = "|";
19   private String[] servicesArray = new String[100];
20   private int servicesArrayLength;
21   private Hashtable servicesHash = new Hashtable();
22
23   // priavte variable to hold the HTML page
24   private String RSSdata;
25
26
27   // Constructor registers itself with the service manager
28   public news(Object creator, Integer priority){
29     Manager = (servicemanager)creator;
30     threadPriority = priority;
31
32     // register with the service manager
```

```
33      Manager.register("news",this, batch, this.serviceType);
34
35      // set the thread priority
36      this.setPriority(priority.intValue());
37    }
38
39
40    // this is called by the servicemanager
41    public void startProcessing(){
42
43      // create the inital hash from the news.cfg file
44      boolean ok = createNewsArray(NewsConfigFile);
45
46      String Location = "none";
47
48      // find if any arguments to the service have been given,
49      // if so, try and find a matching URL
50      for(int i = 0; i < servicesArrayLength; i++){
51
52        String name = servicesArray[i];
53
54        // if the service arguments contain a word which is in
55        // the news.cfg file, retrieve RDF file from the URL
56        if(batch[2].indexOf(name) >= 0){
57          Location = servicesHash.get(servicesArray[i]);
58        }
59      }
60
61      // get the headlines from this site...
62      SearchNewsSite(defaultLocation);
63    }
64
65
66    // contacts the server, gets back the page, strips the
67    // HTML out and then sends the output text to the renderer
68    private void SearchNewsSite(String HostAndPage) {
69
70      // new class handle
```

```
71      http httpHandle = new http();
72
73      // results string declaration
74      StringBuffer results = null;
75      String links = new String();
76      String RDF = new String("error");
77      String PageName = new String();
78
79      // gets the whole HTML page
80      RDF = new String(httpHandle.getHTTP(HostAndPage ));
81
82      // work on a copy of the data
83      String RDFcopy = RDF.toUpperCase();
84
85      // if the server does not respond within the retry limit,
86      // then write an error.
87      if(!RDF.equals("error") || !RDF.equals("")){
88
89        // get the headlines from the rss feed...
90        results = new StringBuffer(getHeadlines(RDFcopy));
91
92        // add a nice little message ;)
93        results.append("This is the end of the news." );
94      }
95      else{
96          // could not retrieve the page for whatever reason
97          results.append("Headlines currently unavailable");
98      }
99
100     // just send the results to the output object the batch
101     // array is sent so the render type can be determined
102     Manager.out.renderOutput(results.toString(), batch);
103
104     // when processing has finished.. remove registration
105     // remove the class from servicemanager, batch[0] = ID
106     Manager.remove(batch[0], this, this.serviceType);
107   }
108
```

```
109
110   //create the array of url's from the news.cfg file
111   private boolean createNewsArray(String File){
112
113     // read in the services file and create an array.
114     String Entries = Manager.file.ReadFromFile(File);
115
116     if(Entries.length() > 0){
117
118       // ok seperate words in the Entries string.
119       int ArrayIndex = 0;
120
121       int start = 0;
122       int next = Entries.indexOf(delimiter, start+1);
123
124       String word = Entries.substring(start+1, next);
125       String nextWord;
126
127       // while the end of the file is not reached
128       while(!word.equals("EOF")){
129
130         // move the pointers
131         start = next+1;
132         next = Entries.indexOf(delimiter, start+1);
133         nextWord = Entries.substring(start, next);
134
135         // ok add entries into translation hash
136         servicesHash.put(word, nextWord);
137         servicesArray[ArrayIndex] = word;
138         ArrayIndex++;
139
140         start = next+1;
141         next = Entries.indexOf(delimiter, start+1);
142
143         word = Entries.substring(start+1, next);
144       }
145
146       // take a copy of the number of services ;-)
```

```
147         servicesArrayLength = ArrayIndex;
148
149         return true;
150       }
151
152       // the fileEntries was too short !!! ie didn't exist
153       return false;
154     }
155
156
157     // This method removes the tags out of the document.
158     public String getHeadlines(String data){
159
160       // local variables
161       int Index = 0;
162       int endIndex ;
163       StringBuffer returnBuffer = new StringBuffer();
164
165       // puts the HTML page into the HTMLdata string
166       RSSdata = data;
167
168       String itemString = "<ITEM>";
169       String titleString = "<TITLE>";
170       String titleEndString = "</TITLE>";
171       String linkString = "<LINK>";
172       String linkEndString = "</LINK>";
173
174         while(true){
175           // get the start of the item element
176           Index = RSSdata.indexOf(itemString, Index);
177
178           // end of the file
179           if(Index == -1){
180               break;
181           }
182
183           // get the start of the title element from the item
184           // element
```

```
185           Index = RSSdata.indexOf(titleString, Index);
186
187           // get the end of the element
188           endIndex = RSSdata.indexOf(titleEndString, Index);
189
190           // get the headline
191           returnBuffer.append(RSSdata.substring(Index +
                  titleString.length(), endIndex) + ".\n");
192
193           // set the begin pointer to the end of the current
194           // element
195           Index = endIndex;
196         }
197
198         // return the headlines
199         return returnBuffer.toString();
200     }
201 }
```

C.5 Spatial reminder agent code

Listing C.5: Spatial Reminder Agent Code

```
1  package sulawesi.services;
2
3  import sulawesi.decision.servicemanager;
4  import sulawesi.base.misc.*;
5  import java.util.*;
6  import java.io.*;
7
8  public class remind extends Thread implements servicedecision
       , Serializable{
9
10   // a reference back to the service manager
11   private servicemanager Man = null;
12   private Integer threadPriority = null;
13
14   // create a buffer for the service manager to fill
15   private String[] batch = new String[5];
16
17   // private variable to keep the messages in
18   private String newData = null;
19
20   // vector array to store location names in
21   private Vector locationVec;
22
23   // vector array to store reminder messages in..
24   private Vector reminderVec;
25
26   // current location
27   private String Location;
28
29
30   // Constructor registers itself with the
31   // ServiceManager
```

```
32    public remind(Object creator, Integer priority){
33      Man = (servicemanager)creator;
34      threadPriority = priority;
35
36      // set the thread priority
37      this.setPriority(priority.intValue());
38
39      // register with the service manager
40      Man.register("remind",this, batch, this.serviceType);
41
42      // setup the reminder..
43      this.setupReminder();
44    }
45
46
47    // used to return the batch for this object
48    public String[] getBatch(){
49      return batch;
50    }
51
52
53    // interface to recieve messages
54    public void recieveMessage(Object Obj, String theMessage){
55
56      // copy the messge
57      Dat = theMessage;
58
59      int number = 0;
60      int index = 0;
61      number = Dat.indexOf(this.ClassName());
62
63      if(Dat.startsWith("Location:")){
64
65        // get the location from the message...
66        index = Dat.indexOf(" ", 1);
67        String temp = Dat.substring(index+1, Dat.length());
68        Location = temp.toLowerCase();
69
```

```
70        // check to see if we want reminding about
71        // anything from this location ;-)
72        checkReminders();
73      }
74      else if(number > 0){
75          // ok so we got a command, start processing
76          startProcessing();
77      }
78    }
79
80
81    // the run method does some processing
82    public void startProcessing() {
83
84      String Rem;
85      int index = 0;
86      index = newData.indexOf("status");
87
88      if(index > 0){
89          // does the message contain a location
90          // which is in the locationVec array
91          Enumeration vEnum = reminderVec.elements();
92          int i = 0;
93          while(vEnum.hasMoreElements()){
94
95            // increment reminder number
96            i++;
97
98            // location name in vector
99            Rem = (String)vEnum.nextElement();
100
101           // create a nice message
102           String res = new String("Reminder" + i + ":" + Rem);
103
104           // ok get an ID for this service request
105           String Bat = Man.getProcessBatch(ClassName());
106
107           // ok we have a status request, so create
```

```
108             // the batch array for this object
109             // (no ID exists because this is a query)
110             batch = Man.search.getBatchFromNoID(batch, Bat);
111
112             // send a message to the output manager to
113             // render. The batch array is sent as well
114             // so the render type can be determined
115             Man.out.renderOutput(res, batch);
116         }
117     }
118     else{
119         // puts the reminder into the reminder vector
120         reminderVec.addElement(newData);
121     }
122   }
123
124
125   // when a location message is recieved check to see
126   // if the location is in any of the messages.. if so
127   // then print them out
128   public void checkReminders(){
129
130     String Rem;
131     int index = 0;
132
133     // does the message contain a location which is
134     // in the locationVec array
135     Enumeration vEnum = reminderVec.elements();
136     while(vEnum.hasMoreElements()){
137         // location name in vector
138         Rem = (String)vEnum.nextElement();
139         index = Rem.indexOf(Location);
140
141         if(index > 0){
142
143           // create a nice message
144           String result = new String("Reminder: " + Rem);
145
```

```
146             // ok get an ID for this service request
147             String Bat = Man.getProcessBatch(ClassName());
148
149             // create the batch array for this object
150             // (no ID exists because this is a reminder)
151             batch = Man.search.getBatchFromNoID(batch, Bat);
152
153             // just send the message to the output
154             // manager to render. the batch array is
155             // sent as well so the render type can
156             // be determined
157             Man.out.renderOutput(result, batch);
158           }
159       }
160     }
161
162
163     // set up the reminder object
164     public void setupReminder(){
165
166       // initalise the location vector object
167       locationVec =  new Vector();
168
169       // initalise the reminder vector object
170       reminderVec = new Vector();
171
172       // get the location from the config file
173       // read in location.gps file do a look up on the
174       // coordinates
175       String LocationsFile  = Man.SulawesiLocation + "config" +
              Man.f.separator + "location.gps";
176       String Lookup = Man.file.ReadFromFile(LocationsFile);
177
178       // parse the file and store data
179       // get location of first '|'
180       String seperator = "|";
181       // reset the start value
182       int start = 0;
```

```
183     int end = 0;
184     String LocationName = "";
185
186     // while not at the end of the file, read in data
187     while(!LocationName.equalsIgnoreCase("EOF")){
188       // get location of 1st and 2nd seperator string
189       start = Lookup.indexOf(seperator, end);
190       end =  Lookup.indexOf(seperator, start+1);
191
192       LocationName = Lookup.substring(start+1, end);
193       // if the index is the end of file marker,
194       // then break out of this while loop
195       if(LocationName.equalsIgnoreCase("EOF")){
196           break;
197       }
198
199       // location of 2nd and 3rd seperator
200       start = end;
201       end =  Lookup.indexOf(seperator, start+1);
202       LocationName = Lookup.substring(start+1, end);
203
204       // location of 3rd and 4th seperator
205       start = end;
206       end =  Lookup.indexOf(seperator, start+1);
207       LocationName = Lookup.substring(start+1, end);
208
209       // add the location name into the vector array
210       LocationName = LocationName.toLowerCase();
211       locationVec.addElement(LocationName);
212
213       // location of 4th and 5th seperator
214       start = end;
215       end =  Lookup.indexOf(seperator, start+1);
216       LocationName = Lookup.substring(start+1, end);
217
218       // location of 5th and 6th seperator
219       start = end;
220       end =  Lookup.indexOf(seperator, start+1);
```

```
221         LocationName = Lookup.substring(start+1, end);
222       }
223     }
224
225
226     // definition of className from servicedecision
227     // interface
228     public String ClassName(){
229       return "remind";
230     }
231
232
233     // get rid of the servicemanager pointer when
234     // serialized.
235     private void writeObject(ObjectOutputStream out) throws
          IOException {
236       // sets the servicemanager object pointer to null
237       Man = null;
238
239       // calls the default write on this object
240       out.defaultWriteObject();
241     }
242   }
```

C.6 Notes agent code

Listing C.6: Notes Agent Code

```
1  package sulawesi.services;
2
3  import java.util.*;
4  import java.text.*;
5  import java.awt.*;
6  import java.io.*;
7  import sulawesi.base.misc.*;
8  import sulawesi.decision.*;
9  import sulawesi.output.text;
10
11
12 public class note extends Thread implements servicedecision,
       guibase {
13
14   // a reference back to the service manager
15   private servicemanager Manager = null;
16   private Integer threadPriority = null;
17
18   // create an empty buffer for the service manager to fill
19   private String[] batch = new String[5];
20   private String FilesLocation;
21   private String RAdataLocation;
22   private File F;
23   private String RAindex;
24   private String RAretrieve;
25   public TextArea notePanel;
26   private String noteConfigFile = "note.cfg";
27
28   // a runtime object which is accessed by this class
29   private Runtime r;
30
31
```

```
32  // Constructor registers itself with the service manager
33  public note(Object creator, Integer priority){
34
35    Manager = (servicemanager)creator;
36    threadPriority = priority;
37
38    // register with the service manager
39    Manager.register(ClassName(),this, batch, serviceType);
40
41    // set the thread priority
42    this.setPriority(priority.intValue());
43
44    // setup the notes app
45    setup();
46  }
47
48
49  // used to return the batch for this object
50  public String[] getBatch(){
51    return batch;
52  }
53
54
55  // definition of className from servicedecision interface
56  public String ClassName(){
57    return "note";
58  }
59
60
61  // Interface to recieve messages
62  public void recieveMessage(Object Obj, String theMessage){
63    this.parseMessage(theMessage);
64  }
65
66
67  // parses the message to determine which it is, either
68  // a list, a query or a save note request
69  private void parseMessage(String theMessage){
```

```
70
71      int save = 0;
72      int query = 0;
73      int list = 0;
74
75      save = theMessage.indexOf("save");
76      query = theMessage.indexOf("query");
77      list = theMessage.indexOf("list");
78
79      /* test to see if the words exist */
80      if(save == -1 && query == -1 && list == -1){
81        // no show, words are not in the sentence so ignore
82      }
83      else if(save != -1 && query == -1 && list == -1){
84        // ok save is only present in the string
85        // call save function
86        this.save(theMessage);
87      }
88      else if(save == -1 && query != -1 && list == -1){
89        // ok query is only present in the string
90        // call the query function
91        this.query(theMessage);
92      }
93      else if(save == -1 && query == -1 && list != -1){
94        // ok list is only present in the string
95        // call the list function
96        this.list(theMessage);
97      }
98      else if(save == -1 && query != -1 && list != -1){
99        // ok list and query are present in the string,
100       // find which is first and call the relevant
101       // function
102       if(query < list ){
103         this.query(theMessage);
104       }
105       else{
106         this.list(theMessage);
107       }
```

```
108       }
109       else if(save != -1 && query == -1 && list != -1){
110         // ok save and list are present in the string
111         // find which is first and call correct function
112         if(save < list ){
113           this.save(theMessage);
114         }
115         else{
116           this.list(theMessage);
117         }
118       }
119       else if(save != -1 && query != -1 && list == -1){
120         // ok save and query are present in the string
121         // find which is first and call correct function
122         if(save < query ){
123           this.save(theMessage);
124         }
125         else{
126           this.query(theMessage);
127         }
128       }
129       else if(save != -1 && query != -1 && list != -1){
130         // ok all words are present in the string
131         // find which is first and call correct function
132         if((query < list) && (query < save) ){
133           this.query(theMessage);
134         }
135         else if((list < query) && (list < save)){
136           this.list(theMessage);
137         }
138         else if((save < query) && (save < list)){
139           this.save(theMessage);
140         }
141       }
142     }
143
144
145     // the run method does all the processing
```

```
146   public void startProcessing(){
147
148   }
149
150
151   // implementation of the guibase interface
152   public void setup(){
153     // get sulawesi location from the servicemanager
154     FilesLocation = Manager.getSulawesiLocation() + "data"
155               + F.separator + this.ClassName();
156
157     // create a new file object for this location
158     F = new File(FilesLocation);
159
160     // check that the <sulawesi>/data/note directory exists
161     // if not create if
162     File dataDirectory = new File(FilesLocation);
163     if(!dataDirectory.exists()){
164       dataDirectory.mkdir();
165     }
166
167     // get sulawesi location from the servicemanager
168     RAdataLocation = Manager.getSulawesiLocation() + "data"
169               + F.separator + "RA";
170
171     // check that the <sulawesi>/data/RA directory exists
172     // if not create it
173     File RAdirectory = new File(RAdataLocation);
174     if(!RAdirectory.exists()){
175       RAdirectory.mkdir();
176     }
177
178     // load the configuration file and get the values of
179     // RA-index and RA-retrieve
180     String config = Manager.getSulawesiLocation() + "config"
181               + F.separator + noteConfigFile;
182
183     RAindex =  new String(Manager.search.GetNextData(
```

```
184       Manager.file.ReadFromFile(config),"RA-index"));
185     RAretrieve = new String(Manager.search.GetNextData(
186       Manager.file.ReadFromFile(config),"RA-retrieve"));
187
188     // instantiate the runtime object
189     r = Runtime.getRuntime();
190
191     // register with gili, get the text object from the
192     // output renderer manager
193     text textOutput = (text)Manager.out.getOutputObject("text"
            );
194
195     /* create a note panel */
196     notePanel =  new TextArea("",8,60,
197               TextArea.SCROLLBARS_VERTICAL_ONLY);
198     notePanel.setEditable(true);
199     notePanel.setBackground(Color.black);
200     notePanel.setForeground(Color.white);
201
202     // register with the gili text renderer
203     textOutput.register((Component)notePanel, ClassName(),
            this);
204   }
205
206
207   // set button names and panel to the front
208   public void focused(Button b1, Button b2, Button b3){
209     b1.setLabel(this.userButton1name());
210     b2.setLabel(this.userButton2name());
211     b3.setLabel(this.userButton3name());
212   }
213
214
215   // The three buttons need names
216   public String userButton1name(){
217     return "Save";
218   }
219
```

```
220
221   public String userButton2name(){
222     return "Query(RA)";
223   }
224
225
226   public String userButton3name(){
227     return "List ";
228   }
229
230
231   // Implementation of the guibase interface
232   public void userButton1pushed(Button b){
233     // this is the SAVE button, so should save the note
234
235     // get the text from the edit buffer
236     String writeData = notePanel.getText();
237
238     // check that there is more than 1 character in
239     // the edit buffer before writing.
240
241     if(writeData.length() >= 1){
242       // construct the filename from the date & time
243       Date date = new Date();
244       String file = FilesLocation + F.separator
245                     + date.toString();
246
247       // write file
248       Manager.file.OverWriteFile(file, writeData );
249
250       // remove text from the display to show that
251       // it has been written ok
252       notePanel.setText("");
253
254       // now index the file with the remberance agent
255       try{
256         // need to add 'file' in order for the RA to index
257         // filenames with spaces in !!
```

```
258         Process p = r.exec(RAindex + " " + RAdataLocation
259                     + " " + FilesLocation);
260       }catch(Exception e){}
261     }
262   }
263
264
265   // implementation of the guibase interface
266   public void userButton2pushed(Button b){
267     // this is the QUERY button, query the current text in
268     // the buffer with files in the notes data directory
269
270     BufferedReader inStream = null;
271     PrintWriter outStream = null;
272     String RAdata;
273
274     // get the text from the edit buffer
275     String queryData = notePanel.getText();
276     try{
277       // call the  RA retrieve program with the data location
278       Process p = r.exec(RAretrieve + " " + RAdataLocation );
279
280       // connect up the streams to the process
281       inStream = new BufferedReader(
282         new InputStreamReader(p.getInputStream()));
283       outStream = new PrintWriter(p.getOutputStream());
284
285       BufferedReader errStream =  new BufferedReader(
286         new InputStreamReader(p.getErrorStream()));
287
288       // now send commands to RA and parse the response
289       // for now only get two matches
290       outStream.println("query 2\n");
291
292       // send retrieve command
293       outStream.println("retrieve\n");
294
295       // send query text + ctrl-D
```

```
296         outStream.println(queryData + "\n");
297         outStream.println("^D");
298         outStream.flush();
299
300         // read the results
301         while (!inStream.ready()){
302             RAdata = inStream.readLine();
303         }
304
305         // display the results
306         notePanel.setText(RAdata);
307       }catch(Exception e){}
308     }
309
310
311     // implementation of the guibase interface
312     public void userButton3pushed(Button b){
313       // this is the LIST button, so list the notes in the
314       // <sulawesi>/data/notes directory
315       int i = 0;
316       String[] dirList = F.list();
317
318       // print out the director listing
319       while(i < dirList.length ){
320         notePanel.append( i + ": " + dirList[i] + "\n");
321         i++;
322       }
323     }
324
325
326     // this handles saving the note
327     private void save(String theMessage){
328
329       if(theMessage.length() >= 1){
330         // construct the filename from the date & time
331         Date date = new Date();
332         String file = FilesLocation + F.separator
333                       + date.toString();
```

```
334
335         // write file
336         Manager.file.OverWriteFile(file, theMessage );
337
338         // now index the file with the remberance agent
339         try{
340           // need to add 'file' in order for the RA to index
341           // filenames with spaces in !!
342           Process p = r.exec(RAindex + " " + RAdataLocation
343                       + " " + FilesLocation);
344         }catch(Exception e){}
345       }
346
347       // just output the result to the output object
348       Manager.out.renderOutput("saved the note", batch);
349     }
350
351
352     // this handles lisiting the notes
353     private void list(String theMessage){
354       int i = 0;
355       String[] dirList = F.list();
356       StringBuffer temp;
357
358       // print out the director listing
359       while(i < dirList.length ){
360         temp.append( i + ": " + dirList[i] + "\n");
361         i++;
362       }
363
364       // just output the result to the output object
365       Manager.out.renderOutput(temp, batch);
366     }
367
368
369     // this handles quering the note
370     private void query(String theMessage){
371       BufferedReader inStream = null;
```

```
372     PrintWriter outStream = null;
373     String RAdata;
374
375     try{
376       // call the  RA retrieve program with the data location
377       Process p = r.exec(RAretrieve + " " + RAdataLocation);
378
379       // connect up the streams to the process
380       inStream = new BufferedReader(
381         new InputStreamReader(p.getInputStream()));
382       outStream = new PrintWriter(p.getOutputStream());
383
384       BufferedReader errStream =  new BufferedReader(
385         new InputStreamReader(p.getErrorStream()));
386
387       // now send commands to RA and parse the response
388       // for now only get two matches
389       outStream.println("query 2\n");
390
391       // send retrieve command
392       outStream.println("retrieve\n");
393
394       // send query text + ctrl-D
395       outStream.println(theMessage + "\n");
396       outStream.println("^D");
397       outStream.flush();
398
399       // read the results
400       while (!inStream.ready()){
401           RAdata = inStream.readLine();
402       }
403
404       // just output the result to the output object
405       Manager.out.renderOutput(RAdata, batch);
406     }catch(Exception e){}
407   }
408 }
```