Technical Report

UCAM-CL-TR-634
ISSN 1476-2986

Number 634

Visualisation, interpretation and use of location-aware interfaces

Kasim Rehman

May 2005

This technical report is based on a dissertation submitted November 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St Catharine's College.

ISSN 1476-2986

Abstract

Ubiquitous Computing (Ubicomp), a term coined by Mark Weiser in the early 1990's, is about transparently equipping the physical environment and everyday objects in it with computational, sensing and networking abilities. In contrast with traditional desktop computing the "computer" moves into the background, unobtrusively supporting users in their everyday life.

One of the instantiations of Ubicomp is location-aware computing. Using location sensors, the "computer" *reacts* to changes in location of users and everyday objects. Location changes are used to infer user intent in order to give the user the most appropriate support for the task she is performing. Such support can consist of automatically providing information or configuring devices and applications deemed adequate for the inferred user task.

Experience with these applications has uncovered a number of usability problems that stem from the fact that the "computer" in this paradigm has become unidentifiable for the user. More specifically, these arise from lack of feedback from, loss of user control over, and the inability to provide a conceptual model of the "computer".

Starting from the proven premise that feedback is indispensable for smooth human-machine interaction, a system that uses Augmented Reality in order to visually provide information about the state of a location-aware environment and devices in it, is designed and implemented.

Augmented Reality (AR) as it is understood for the purpose of this research uses a *see-through* head-mounted display, trackers and 3-dimensional (3D) graphics in order to give users the illusion that 3-dimensional graphical objects specified and generated on a computer are actually located in the real world.

The system described in this thesis can be called a Graphical User Interface (GUI) for a physical environment. Properties of GUIs for desktop environments are used as a valuable resource in designing a software architecture that supports interactivity in a location-aware environment, understanding how users might conceptualise the "computer" and extracting design principles for visualisation in a Ubicomp environment.

Most importantly this research offers a solution to fundamental interaction problems in Ubicomp environments. In doing so this research presents the next step from *reactive* environments to *interactive* environments.

Acknowledgements

I am grateful to Frank Stajano for supervising me in my PhD and steering it towards the interesting issues. His high standards have always been a challenge for me.

George Coulouris has always taken so much interest in my work. His experience has been invaluable in understanding what research is about. He has really gone out of his way to help me.

Without Andy Hopper I would not be here. He entrusted me with this project when he took me on as a student.

Furthermore, I thank everyone at the LCE and AT&T Labs Cambridge for providing an atmosphere that brings the best out of engineers. Many thanks to Duncan Grisby.

In addition to my supervisors and advisors the following people have in one way or another influenced my work through inspiring lectures, discussions or courses: Donald Norman, William Newman, Robert Spence, John Barton, Anind Dey, Hans Gellersen, Bashar Nuseibeh and Michael Jackson.

I would also like to acknowledge the support I have received from my sister, brother, sister-in-law, nephews and especially my parents.

Furthermore, my friends deserve thanks for their support, encouragement and entertainment in the seven years at university that have led to this thesis: Uli, Osman, Khaze, Mansoor, Wajahat, Zahed, Zubair, Hussein and everyone at the LCE especially Rip, Hani, Leo, Kam, Eli and Chong.

I would also like to thank my high school teachers, specially C.-P. Holste-von-Mutius for helping me develop my critical thinking and writing skills and also W. Bartsch for making me believe I had it in me.

The following people or entities have supported me and my work financially: My father, Cambridge University Board of Graduate Studies, AT&T Research Labs Cambridge, Cambridge University Engineering Department, Cambridge European Trust and St. Catharine's College.

For my parents.

Contents

List of Figures

Chapter 1

Disappearing Computer – A Two-Edged Sword

In the midst of a recent monologue on the coming era, delivered one-on-one in his Aspen office, Bill Joy offers to print out a paper that illustrates a salient point. He reaches for his laptop, which is equipped with the sort of wireless high-speed Internet connection that, one day, may be a routine adornment in all our cameras, palmtops, game machines, medical sensors and, yes, dishwashers. According to the theory, these will all be linked together, of course, in an infrastructure that will virtually eliminate crashes and glitches. He keyboards the command to print the document in the adjoining room. And nothing happens. “You know what?” he finally says. “I think this did get printed – on the printer back in my house across town.”
By Steven Levy; Newsweek; May 31, 1999

In an attempt to describe the lifestyle of the future, Oxford University neuroscientist Lady Susan Greenfield in her most recent book [1] sees a world in which humans will have access to instant connectivity and computers will sense the humans’ wishes and adapt to them. Computers will automatically do the “right” thing at the “right” time as humans wander around in a world with new and exciting entertainment, communication, shopping, work, household and other computer-supported facilities at their service.

The vision described has its roots in what was called “Ubiquitous Computing” by Mark Weiser at Xerox PARC Laboratories in the early 1990’s. Weiser tried to envision which way computing was heading after the Desktop had penetrated most offices and homes. By extrapolating technology trends he arrived at the conclusion that it would soon be possible to cheaply equip physical environments and everyday objects with computational and networking facilities. Hence, the number of “computers” a human would interact with on a continuous basis could increase from one to hundreds. In order to reap benefits from this potential a new computing paradigm would be required.

Influenced by the philosophy of Postmodernism [2], in particular the critique of strong objectivity, Weiser saw human empowerment at the centre of this new paradigm. Numerous computers would *transparently* support human beings in their everyday tasks who would only peripherally be aware of the computers’ involvement. In this sense computers would not only physically be hidden away in the environment, but also disappear in the human user’s focus of attention, quite the opposite of desktop computing.

The motivation for this kind of computing that has also been described as "Disappearing Computing" [3] is the observation that the most useful tools to human beings do not draw attention to themselves, a prototype of such a tool being a pencil [4].

As researchers have started working on overcoming the technical challenges in order to put this vision into practice the formidable design challenge associated with the statement of objective regarding this new interaction paradigm continues to be underestimated.

The main question is whether needing to interact with "hundreds" [5] of different computers will result in chaos with each one of them screaming for our attention as we try to remember what to do next, or will we be able to integrate them in a harmonious orchestra of devices and applications.

With most Ubicomp developers having dealt with at most one truly ubiquitous application at a time this problem has gone unnoticed. Using the two dimensions introduced by Lyytinen and Yoo [6], mobility and embeddedness, ubiquitous computing can be regarded as offering both high mobility and high embeddedness. Characteristic for Ubicomp systems is an element of context-awareness. Through embedded sensors Ubicomp systems perceive the user's current context, especially her location, in order to best meet her needs in a particular situation. This is called *context-awareness*, or location-awareness if only the user's location is used by the application to adapt to her needs. More common these days are "Pervasive Computing" and "Mobile Computing", each offering only high embeddedness or mobility respectively.

Nevertheless, at research centres at which a limited number of Ubicomp applications been deployed for everyday use the gravity of the interaction design problem is slowly unfolding.

In the following a number of these problems will be shown using examples. The analysis will show again and again that the problems arising are *not accidental* but have their very root in the idea of "Disappearing Computing". Designing tools that passively stay out of the way and yet pro-actively adapt to the user's needs is not an easy task.

Most future scenarios are content, explaining what happens if everything works according to the users wishes. What happens if a "computer" that has so skillfully stayed out of the human's way does not perform appropriately? Just an isolated case? Well, from our experience with desktop computing we know that system crashes occur regularly.

It is time to take a step back from tackling engineering challenges Ubicomp poses and instead dwell on traditional design principles in order to help Ubicomp systems to make that vital step from being prototypes to being reliable and well-crafted products of engineering design.

1.1 Problems Encountered in Ubicomp

The author will base his problem analysis on Norman's design principles [7] and look at typical violations of these in Ubicomp.

1.1.1 Lack of a Good Conceptual Model

Ideally such a model should be provided by showing affordances, constraints and mappings. Affordances are what a user can do with a device, constraints are what she cannot. Mappings bind user action and its expected outcomes. A door handle, for example shows that the door affords opening and constrains the side to which it opens. Pushing down the handle maps to an internal part of the lock moving to one side.

Many of the new interaction styles we encounter in Ubicomp, however, do not permit the provision of a good conceptual model. Gesture recognition interfaces, for example, draw their naturalness from the fact that they do not need to present choices (what they "afford") the user would need to select. Tangible Interfaces [8] usually suffer from the fact that it is impossible to achieve a perfect mapping between the constraints of the physical object being manipulated and constraints in the virtual world.

Sometimes the reason for missing affordances is partly of an aesthetic nature. One of the advantages of Radio Frequency (RF)-id tags [9] is that they make it possible to augment everyday objects completely unobtrusively, triggering some actions in the virtual world when brought close to a RF reader. The flipside is that we cannot really tell what will happen if we "use" the object or even which objects we can "use".

In other cases the nature of the affordance is such that it is hardly possible to visualise it. How could we visualise the existence of a wireless LAN connectivity in a particular region of a room?

1.1.2 Loss of Control

An important class of Ubicomp applications are proactive triggering applications, such as a museum guide that automatically brings up the correct artifact description, or headphones that automatically associate themselves with a running CD player when brought close. The problem is that most of these applications do not provide an "override" facility. This is mostly due to feasibility (should headphones have an interface?) or the desire to hide the computer (most of the attraction of the electronic museum guide is due to the fact that pages come up by "magic").

As far as automation is concerned, one has to realise that whenever context-aware systems infer something, we should be able to intervene if the inference is wrong. To find examples for "user frustrations" that result from wrong inference we do not even have to look at pathological cases. Aoki et al. [10] describe scenarios where visitors in a museum can get annoyed with what their electronic guide is displaying, if it is inferring using location only. The solution proposed includes an element of selection, i.e. present context-sensitive choices to the user and let her take the final decision.

Even if wrong inference is not the problem, we would naturally like to exercise some control on the actions that are done for us. Just consider the number of parameters associated with a simple action such as printing a document.

1.1.3 Lack of Feedback

One of the applications surveyed, for example, automatically puts your job at the front of the queue as you approach a printer [11]. The question that remains to be asked is, what happens if your job does not come out right away? The user has to validate different hypotheses as to why the system did not work as expected, since the standard printer user interface does not cater for these kinds of messages. Partly the problem is, that tasks in Ubicomp are often unstructured unlike traditional computing. Some applications, for example, allow you to enable call forwarding to the nearest phone [11]. Assume the user switches this facility on, for how long shall it stay on? Until the end of the day? Or until she leaves the room? Considering that the user may well be able to enable or disable dozens of such facilities it is useful for her to have continuous feedback as to what is enabled where, even if she is doing another task. The following example illustrates that these personal settings can become quite complicated and hard to remember [12]:

> Automatic Logon to other workstations is required as soon as he starts using one. For telephones, the script controls the "Busy" function. The user has asked that Busy is displayed on location screens when he is with more than 3 people, at a specified time (9.00-10.00 Mondays), at his present location if the badge button is pressed, and always when in the boss's office.

Also, since Ubicomp involves a lot of information dissemination it would be desirable to know what is being captured or passed on. Want and Hopper [11] describe this concern:

> A related problem is that with cameras proliferating all over a building it would be very desirable to know when you were 'on camera' and when you were not.

The most important lesson to be learnt here is that we are facing considerable asymmetry regarding the importance of feedback in Ubicomp environments versus the ability to provide it. It is not always clear *where* and *when* to provide feedback.

1.1.4 Breakdown of Traditional Mental Models

Ideally, interfaces should not only convey information about how to use them, but in their function as a surface representation [13] also convey an image of the underlying system. We use the term mental model in order to distinguish the model a user builds of a system from the model the user builds about its usage (described earlier). The problem here is that in a world of distributed, invisibly interconnected computers, the "surface" no longer exists. Various researchers (e.g. [14]) have reported that users of location-aware tourist guides got confused because the places being described, for example, were at least ten minutes walk away. The reason was the inaccuracy of the GPS sensors, which had resulted in the wrong page coming up. A more graphic description of the problem we are dealing with in this section was provided by Beverly Harrison [15]:

> [...]while I was at University of Toronto, we did some early work on what later became known as the "reactive room". Initially versions of the conference room were known as the "possessed room". Lights turned on and off when you moved or ceased moving. Equipment "spoke to you" through an electronic voice and told you that you had 5 seconds to "take action" or it would power down. Video signals switched to channels and "forgot" how to switch back.

We have seen earlier that we can eliminate automation problems by offering choices to the user. The problem of the "possessed room" is a bit deeper. It is that we do not understand how it works in the first place. Partly, this is due to the networked nature of Ubicomp systems. When we press a button on a normal machine we know its effect will become visible there and then. Norman's interaction model [7] for user interface (UI) design is based on the ability of the user to perceive causality. For a vending machine this is no problem, but what about cause and effect of the document printed on a printer across town.

Norman's Seven Stages [7] describe seven steps a user goes through when interacting with a machine. They always start with the user perceiving the state of the machine and end with her evaluating the outcome of the interaction. For Ubiquitous Computing one could add an extra stage: Find the machine. The difficulty of finding an information appliance or service does not only arise from the fact that they can be easily lost but also the fact that they may be invisible or unobvious, e.g. a tangible interface [8].

1.2 Thesis Outline

The rest of the thesis is organised as follows. The next chapter will concretise the problem to be solved and follow that up with a research statement. The implementation of an Augmented Reality system will be proposed to solve the problems mentioned.

Chapter 3 will deal with the basics of implementing an Augmented Reality system. The following chapter will deal with how to integrate such a system into a location-aware environment. Chapter 5 will discuss implications of adding interactivity to location-aware applications in terms of software architecture.

The chapter after that is about implementing a set of interaction prototypes needed to introduce control into our environment. Chapter 7 will then put everything together in a design and implementation of an interactive location-aware application. Our claims will be evaluated with users in the following chapter.

Chapter 9 is about how users interpret smart environments. Insights gained from this chapter and the experience in building interactive prototypes for our environment will result in a chapter that outlines a design approach. The aim of the design approach introduced in Chapter 10 is to present some steps, which will result in well-designed interactive location-aware application.

In the conclusion we will look at some possible future work and summarise our findings.

1.3 About This Thesis

1.3.1 Assumptions

Location-aware computing is a wide area. When we are looking at how humans can interact in location-aware or even ubiquitous computing we will see a wealth of very different interaction paradigms and styles, such as Tangible Interfaces (e.g. [8]) or speech interfaces etc. This implies that we need to limit the scope of this thesis somewhat in order to be able abstract findings that are not too generic to be useful.

Being experimental, this thesis is bound by the implementation of the system used to argue it. Specifically, the system employed has its roots in the idea of Sentient Computing [16], a tradition that started with the Active Badge [17] system. Location-aware applications that have arisen from this tradition have focussed on how computer systems can assist users in performing everyday tasks. Being conceived as mostly productivity-enhancing office applications they have carried forward some ideas known from traditional desktop office applications. The idea, for example, that there is a boundary between the machine and the user across which users have to communicate has its root in traditional engineering but also forms the basis of most everyday GUI applications. This is not the only way to look at Human-Computer Interaction (see, for example, [18]).

Other concepts "inherited" from the Sentient Computing project include the idea of defining location-awareness through regions in space or the use of a personal device to affect system actions. The following chapters will discuss the existing system (alongside the extensions made) in great detail. For now, it suffices to note that this thesis work is linked with one particular implementation.

1.3.2 Main Contributions

This thesis makes its main contributions in three areas.

1. **Visualisation of Location-Aware Applications**. In an attempt to increase intelligibility of location-aware applications, a system based on Augmented Reality has been implemented to allow applications to provide feedback to the user about their state. In doing so the thesis tackles a problem noticed by researchers and practicians in the field (c.f. Bellotti et al. [19]); however, few have shown how to implement solutions. One of the reasons for that is that location-aware applications have lacked the ability to provide feedback or present their working to the user. The use of visualisation offers new

opportunities to interact with location-aware applications. This thesis investigates a new interaction paradigm for location-aware applications based on visual interaction.

2. **Software Architectures for Context-Aware Computing**. The thesis introduces an extended Model-View-Controller [20] design pattern in order to deliver an architecture that fosters separation of concerns and interactivity. The increase in sophistication at which *responsive* application behaviour is modelled, is met by an increase in application interaction states (as application responses get finer grained). The software architecture presented here helps cope with the higher load of interaction state transition analysis and management for context-aware applications. It is one of this system's distinctive features to make use of architectural composition in order to cope with this complexity.

3. **Design for Ubiquitous Computing**. Examples in this chapter have shown that there is a real need to find and apply design principles to Ubicomp applications, which ensure that designers can predict implications their design decisions will have for the use of their applications. Using intelligibility/user understandability as one measure of good design, a number of approaches to achieving good design for Ubicomp are examined. In particular the applicability of Norman's principles [7] to Ubicomp is reflected upon.

1.4 Conclusion

This chapter was thought as an introduction to the field of Ubiquitous Computing. The philosophy behind it was presented and its potential to revolutionise computing in general was acknowledged. At the same time we recognised that the vision of hundreds of computers working together for the human being poses a significant design challenge that has been underestimated due to the fact that most Ubicomp systems implemented so far have had prototypical character.

A number of Ubicomp systems were analysed using well-established principles of general machine design. It was shown that all three of Norman's basic principles (provision of a good conceptual model, provision of control, provision of feedback) have been violated in current Ubicomp systems. The violation of these principles was recognised not to be accidental in character but a result of the fact that Ubicomp presents a breakaway from traditional computing.

Having elaborated on interaction problems with current Ubicomp systems the proposed solution shall now be outlined.

Chapter 2

Visually Augmenting Ubicomp Environments

If we wanted to crystallise the problem in one sentence we may say the following: Ubicomp systems try to stay out of the user's sight, but more importantly out of her mind, whereas a well-designed interactive system does make itself noticeable at some points. By now it should be clear why the author believes that Ubicomp systems should be *interactive*, i.e. they should be able to react to the user's actions and provide feedback in order for the user to decide what to do next. The question is how we can adapt traditional interactive system design to Ubicomp systems, considering that task or machine may not be structured or well defined respectively. In other words, how can we design a user interface for a Ubicomp environment that (c.f. Sections 1.1.1 through 1.1.4)

- maps affordances and constraints between the real world and the virtual one as naturally as possible
- can offer choices to the user
- can provide feedback anywhere at any time
- represents a uniform "surface" to a system that may consist of hundreds of interconnected devices

In our attempt to solve this design problems we can consult yet another one of Norman's principles: the principle of visibility. The author has come to the conclusion that an Augmented Reality system, that can dynamically place virtual information in the real world would best meet the requirements set out. Before examining what such a proposition will involve, let's have a look at some work that has previously been done on visualisation for Ubicomp.

2.1 Existing Visualisation in Ubicomp

It is helpful to introduce a taxonomy for such a survey, also because it will help us explore the design space. Since Ubicomp is supporting tasks we shall classify the visualisation according to the extent to which they support a particular task. We can distinguish four types of information the visualisation provides: task-central, task-peripheral, task-external and task-meta. Other classifications may be possible; not everyone will agree that the concept of a "task" is central to Interactive Systems Design, but this is a classical HCI assumption.

2.1.1 Task-Central Information

For some tasks the retrieval of information at some point is an established component of the task. The how, when and what of the retrieval in these tasks is pretty much fixed.

Examples in the ubiquitous field include navigation systems that give directions such as the Hybrid Indoor Navigation System [21], guides (Cyberguide [22], HyperAudio [23], GUIDE [24]) and visual reminders (Remembrance Agent [25], Memory Glasses [26]) that use a head-mounted display. The augmented reality "tagging" guide developed by Hoellerer et al. [27] as well as expert systems described by Siewiorek et al. [28] or Feiner et al. [29] show that the information presentation need not take place on small screen, but can be superimposed on reality.

Tasks which result in a large amount of information being retrieved can make use of visualisations that do more than just make visible, i.e. help forming a mental model. The mobile voice notes recorder developed by Degen et al. [30], for example, displays the continuous volume level of a tape segment, so that the user can see how long various notes are. Abowd et al.'s Digital Library uses a similar visualisation [31]. Dey et al.'s Conference Assistant displays a colour-encoded conference schedule [32]. C-MAP [33], another guide uses a network visualisation to show people of related interest. Ryan's FieldNote employs a map [34].

2.1.2 Task-Peripheral Information

When performing tasks humans (sometimes subconsciously) use a large amount of stimuli to fine-tune their task. While drinking tea, for example, we become more careful when we receive visual and haptic stimuli signalling "hot".

With new sensor capabilities we can now augment the human by giving him extra senses. Wouldn't it be helpful to "see" the range of a cordless phone through an augmented reality system while talking? Furthermore, by employing a network we can *extend* the range of our senses.

Examples of this include the Chameleon Mug that changes its colour depending on its temperature [35], Cutlery that warns against bacteria [36], Augmented Reality systems that just tag the environment, GroupCast [37] that displays common interests of two people who are meeting on a screen, IPADs that signal proximity of colleagues [38].

Again, visualisation can go further and create a mental image inside the human. Schilit's ActiveMap [39] is more sophisticated than IPADs. "Things that blink" [40] do not explicitly display any information, but use coloured LEDs to show how much two people who are meeting have in common. TouchCounters [41] use LEDs to visualise usage statistics of tagged containers.

The difficulty in visualising information that is not targeted at a specific task lies in the fact that its "efficiency" is very low. A lot of information needs to be displayed in anticipation that it may be useful to someone.

2.1.3 Task-External Information

In this section we will look at information that has not got anything to do with task we are performing at a given moment.

Awareness applications are an example of visualisation in this field. These make use of the fact that some particular information is often needed with low detail. The often cited Dangling String [42] shows the network load in a room by mechanical movement. Ambient Media [43] encodes quantitative information, showing how many hits a particular web page has accumulated by using lights of different colours. AROMA [44] shows colleague activity by an avatar. Similarly, OwnTime [45] encodes attributes of visitors waiting to see one. Further examples include WebAware [46] and a Weather Forecast Window [47]. AnchoredDisplays [48]

Information Type	Description	Examples
task-meta	information on how a task is performed	"this is an `active' printer"
task-central	task cannot be performed without it	specifying parameters for printing, select document
task-peripheral	information to fine-tune task	region where printing will be initiated, progress

Figure 2.1: Interaction-related information involved in using a location-aware printer

show that awareness applications do not necessarily need to visualise by forming a mental image of the environment in the user's mind, but can merely display news.

2.1.4 Task Meta-Information

This type of information concerns how a task is performed and whether it has been performed correctly. This information becomes especially interesting when the task involves the computer, as the information then becomes dynamic. The design of such an interactive ubiquitous system is discussed separately in the next section.

HyperPalette [49], an example for such an application uses 2-D projections in order to show the user of what he can do with his PDA.

2.2 Using Augmented Reality for Visualisation in Location-Aware Environments

The survey has shown that visualisation has been used in Ubicomp mainly in order to assist humans in a real-world task.

We can also see that researchers have so far steered away from visualising anything that may be considered as a feedback from a "computer". Reasons for this were given in the last chapter.

Figure 2.1 shows examples of the information the author has in mind. Take, for example, the task of instantly printing a document by walking up to a printer. A similar location-aware application is described in [11].

2.2.1 Location-Aware Applications

The previous section ended with the description of a location-aware application. Location-Aware computing is less of a subfield of Ubicomp, but rather one flavour of Ubicomp. An accurate definition of location-aware applications can be given as following:

A location-aware application is an application that reacts to changes of location of real-world objects or people. An application is a program that performs a service for the user and may involve multiple devices and services available in the environment.

The first location-aware application was a people-finding application that was used in order to locate people in an office. Users were made to carry small card-like infrared (IR) transmitters called Active Badges [17]. IR are sensors deployed throughout the office in order to pick up IR transmissions from the badges and forward them to a server. Locations of people could be used to forward phone calls, find out where meetings were taking place etc. The Active Badges also allowed to open doors using one of the two buttons on the badge.

As the field developed, richer applications were developed that made use of a person's location, such as tourist guides to display relevant information after sensing the user's position through GPS.

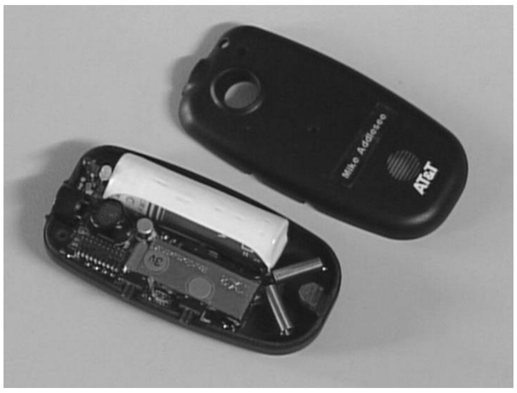


Figure 2.2: The Bat is a small device that is about 85 mm long. Two buttons are located on the left side (from [50]).

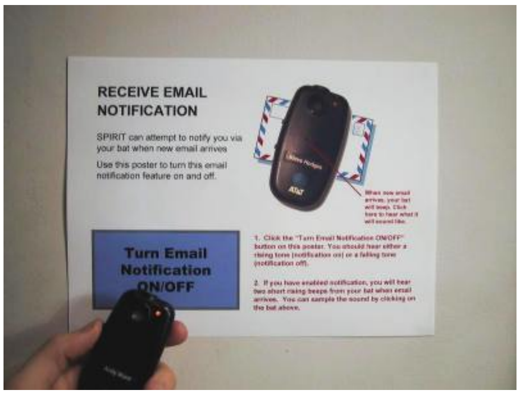

RECEIVE EMAIL NOTIFICATION
SPIRIT can attempt to notify you via your bat when new email arrives
Use this poster to turn this email notification feature on and off.
When new email arrives, your bat will beep. Click here to hear what it will sound like.
Turn Email Notification ON/OFF
1. Click the "Turn Email Notification ON/OFF" button on this poster. You should hear either a rising tone (notification on) or a falling tone (notification off).
2. If you have enabled notification, you will hear two short rising beeps from your bat when email arrives. You can sample the sound by clicking on the bat above.

Figure 2.3: Posters such as this one are located on walls. Bringing the Bat close to the corresponding "button" (thick-bordered rectangle) and "clicking" will send a command to the environment.

The next generation of indoor location systems such as the Active Bat [50] or Cricket [51] delivered more fine-grained location estimates, allowing these systems to be used for richer interaction. Figure 2.2 shows an Active Bat. By sensing such a locatable sensor/transmitter the system can infer the user's intent, e.g. standing close to a printer would mean the user wants her document transferred to the front of the queue.

Using fine-grained location technology more interesting applications could be developed. One particular way to use the Active Bat is by associating actions with locations. By bringing the Active Bat close to posters put up on walls in a place that has the Active Bat system deployed one can initiate computer actions. Figure 2.3 shows such a poster. Actions include turning on/off phone forwarding, scanning a document, requesting alerts etc.

Experience with these location-aware applications has shown that they suffer from all the problems mentioned in the previous chapter (the "nothing happened" syndrome is typical). In fact, they suffer even more so since, interaction with space is more difficult to visualise than interaction with devices.

Given the challenge location-aware applications pose the work described in this shall deal with making these applications interactive. Interactivity can be seen as the main umbrella encompassing solutions to all the problems identified in the last chapter.

2.2.2 Research Statement

Thesis: Off-Desktop Location-Aware Applications running in a networked environment can be made interactive in order to render them more understandable.

Interactivity implies that the user can at any time perceive the application state and, as may be the case, change it into a desired one by performing an action.

In order to support the thesis a system that can visually provide feedback from an application to the user will be designed and implemented. The system will be deployed in the target environment. It will be used as a platform to develop location-aware applications on. Ways to reap benefits from the new system in terms of understandability from the user's perspective will be examined and conclusions will be drawn.

2.2.3 Method

During the last sections it has become evident that the system to be proposed for solving the above-mentioned interaction problems will be similar to a GUI. However, desktop graphical user interfaces benefit from the fact they have access to a sophisticated visualisation system. One revolutionary idea of the GUI was control over every single pixel on the computer's display. If applications run in the real world rather than the computer an equivalent would be a three-dimensional monitor covering the entire space. Even though this is not possible, there is an alternative that comes close. Augmented Reality with head-up displays and trackers can provide the illusion that the whole world is a graphics container.

2.3 Augmented Reality

In general the term Augmented Reality (AR) is used for any system that links the real world with the virtual world. A simple example is that of RF-id tags that bring up web pages when sensed.

When the expression is used in its more specific sense it means a graphical overlay of virtual objects onto the real world. There are a few possibilities how this can be achieved, apart from using a head-mounted display (HMD). Firstly, one can use projectors in order to project graphics against a surface with which the user can interact. When this is combined with a computer vision system this gives a powerful I/O device. Secondly, one could use a PDA in order to capture the real world with its camera, overlay graphics on it and display it on its screen.

These are interesting approaches. They do not suffer from the fact that the user has to wear bulky hardware. Nevertheless, due to its universality and flexibility the HMD approach is proposed and shall be looked at in more detail.

The first and still one of the most useful HMD-based AR Applications is KARMA [29]. A see-through HMD is used in order to help a repair technician to open up a photocopier. The technician gets real-world instructions, a diagram overlaid directly onto the photocopier as her or she is performing the task.

HMD-based AR has found its way into some parts of manufacturing industry and has some limited use in operation theaters where overlaying patient scans onto the patient while operating has proven useful.

2.3.1 Enabling Technologies

The following will show what kind of equipment is available and required in order to build an HMD-based AR system.

Head-Mounted Displays

For Augmented Reality there are two options of HMDs to be used. See-through and non-see-through. Non-see-through HMDs are used for what is called Video See-through (VST) Augmented Reality. For VST AR the world is recorded through two cameras on the user's head, the recorded picture is augmented digitally and the result displayed to the user on mini-monitors in her helmet.

See-Through HMDs allow the user to see the real world as well as an image from a mini-display. This is achieved by using semi-permissive optical combiners. In this case the user can actually see the real world as well. This kind of AR is called Optical See-Through (OST). Optical HMDs give a better user experience. Their main drawback is their limited availability for high resolutions and darkened view due to the fact that they only let through half the light. Feiner et al. [52] present an application that allows a mobile user to view annotations referring to sights on campus over her see-through HMD.

Trackers

In Augmented Reality applications the user's view of the real world needs to be overlaid with graphically rendered objects. This can only occur if the world view is known at all times. The view seen by the user depends on his orientation and his position. In order for the application to estimate this view the user's position and orientation needs to be tracked. Both position (x, y, z) and orientation (pitch, yaw, roll) allow three degrees of freedom, i.e. the user's view is specified by six values. Any object that is part of the application needs to be tracked. In the case of static objects their position and form can be "hard-coded".

Augmented Reality applications allow for very little tracker inaccuracy, since virtual objects usually have to be aligned with real visible objects. Misalignments that occur are called registration errors. Inaccurate orientation values have a greater impact on the user's perception, since the perceived absolute misplacement of virtual objects increases dramatically with the virtual distance from the user. The required orientation accuracy for Augmented Reality has been estimated to a fraction of a degree. When commenting on required accuracy for perceived overlay one has to take into account the axis of rotation around which the error angle occurs. If virtual objects are rotated around an axis in eye viewing direction of the user, due to inaccurate orientation readings, larger rotation errors are tolerable. The virtual object appears at the same place just facing a different direction. Equation 4.1 and its discussion will show the relevance of this circumstance mathematically.

There exist some methods of correcting inaccurate values of position by performing post-measurement calculations. This can be done because some tracker inaccuracies are due to static environmental interference.

Another type of error that is perceivable is the dynamic registration error. This occurs when the tracker lags behind the user's head movement and cannot send updated position and orientation data fast enough. The result is a temporary misalignment of virtual objects.

There are four types of tracking technologies: Optical, ultrasonic, inertial and electromagnetic. All but one technologies use an emitter and a receiver. The receiver uses a property of the signal, such as time of flight (ultrasonic) or field strength (electromagnetic), in order to sense its position and orientation relative to the transmitter. Inertial trackers are known as sourceless, since they can estimate their position and orientation by taking into account all past head movements (mathematically done by integration).

Optical tracking can be based on sensing orientation changes with respect to a number of fixed LED beacons. The other option is to use a normal CCD camera and sense passive patterns. Computer vision techniques can then be applied to calculate the cameras position and orientation with respect some recognisable patterns. The ARToolkit [53] is an example of a toolkit that allows developers to track markers with a cheap webcam. Passive optical trackers are not very robust, since they depend on lighting conditions. On the other hand, their advantage is that the "transmitters" are cheap since they can just be patterns on a piece of paper. Accurate and fast *active* (using LEDs) optical trackers do exist, but are usually very expensive.

Electromagnetic trackers are the most commonly used trackers. Their problem is the interference with ferromagnetic and conductive materials as can be found in computer monitors, power lines and building frames. There are two types of EM trackers: DC and AC. DC avoids ferromagnetic interference but still suffers from conductive one. Their greatest disadvantage is that they are tethered, i.e. the user can only move within a radius of a couple of meters at best.

Inertial sensors have the advantage that they are sourceless, i.e. the working range is virtually unlimited, unlike electromagnetic trackers that only work within a few feet. This also means that they are tetherless. However, inertial trackers show a continuous drift in their readings, since errors accumulate through integration.

Ultrasonic trackers show inaccuracies as a result of reflections and speed of sound variations due to environmental conditions. As with optical trackers, ultrasonic tracking requires a line of sight. The Active Bat system [50] is an ultrasound-based *location* system. It is not strictly a tracking system. It does not provide accurate orientation values and has a very low update rate, just a couple of Hz, depending on the Quality of Service. Its main advantage is that it covers an area one could call "wide-area" in terms of AR: a whole office building. This is done by covering the entire ceiling with sensors.

2.3.2 Issues in Augmented Reality Systems

There are several limitations with AR. First of all, it is an immature field. Ready-made AR systems do not exist, nor is there a consensus on how best to build one. Custom-made systems always suffer from the fact that a lot of time is spent debugging and even then the systems are not in a robust state.

Secondly, achieving accurate registration, especially static, is notoriously difficult. One has to be content with virtual objects being offset by a few centimetres. A lot of this error depends on how good the system is calibrated. More will be said about this issue in the next chapter.

Thirdly, the maximum available resolution for see-through HMDs is 800×600. From experience, this is good enough not to pose an annoyance, but higher resolution is always better.

Fourthly, even though one can get by with a monoscopic HMD, a stereo HMD is highly recommended, especially when one is not visualising on top of flat surfaces.

Finally, lags in displaying graphics occur because the graphics can not be updated as fast as the user moves her head. While this used to be a major factor, it has now become less of problem with the advent of fast computers and graphics cards.

2.4 Critical Proposition Evaluation

After acquiring some of the necessary background knowledge we are now in the position to evaluate the proposition critically.

The first point that can be brought forward against the argument of the previous chapter that concluded that traditional design principles have been violated in Ubicomp, is that Norman's design principles are not applicable to Ubicomp design. After all, Weiser had envisioned a tool such as a pencil to be the model for computers of the next generation.

Examined more closely, the comparison with a pencil is inappropriate. Tools do not need to be told what to do, they only do one thing. Tools provide feedback in physical form, something the human instinctively understands. The reason why a discipline of interactive system design has evolved is precisely for that reason. Humans cannot sense the state of machines and therefore need to be informed explicitly. When talking about the tool “metaphor”, or tools in this specific context only, the author means archetypical tools such as pencils or hammers. In its wider sense, even a printer could be regarded as a tool.

Another thing that can be said with regards to the previous analysis is that the violation of design principles was accidental and these problems could be alleviated by making use of the facilities available. Can we improve existing applications by tweaking them here and there? The proposition that traditional mental models break down under Ubicomp together with its analysis (see previous Chapter) implies that users need a bigger picture. Ubicomp is more than a number of interesting applications running in space isolated from each other. They interact, they show similarities that make them easier to use etc., much like the GUI for desktop computing.

Another question is why is it necessary to use AR. Alternatives include using PDAs in the way Cooltown makes use of them [54], or using information appliances [55]. After all, AR involves bulky hardware. Augmenting location-aware applications with a PDA would require that the user carries a PDA around as she moves through the environment and keeps one eye on the PDA. AR supports a much more natural interaction style. The user can virtually states of devices, objects and even interaction regions as she moves through the environment. Given that location-aware applications run in space what is more natural than to visualise them there. In fact, using the attribute of embeddedness (discussed earlier), the AR solution is actually more to the idea of Ubicomp. The AR solution will allow each everyday object to have its “display”, no matter how simple it is.

It is true that AR equipment is bulky, but HMDs that are indistinguishable from normal eyeglasses have already appeared on the market. Apart from that, various “visionaries” of computer science have predicted that the HMDs will eventually be used by a large part of computer users (Dertouzos in [56], Feiner in [57]). Even sceptics like Donald Norman [58] believe that the “augmented human being” is inevitable (in fact he even goes further). Tracking poses a greater problem. Nevertheless, a tetherless tracker that uses a small head-mounted camera was implemented for this thesis.

Examining the requirements set out in the beginning of this chapter will show that the system chosen does indeed best meet them. A system similar to the one described was already successfully implemented once [59]. It was, however, more targeted towards visualising sentient information and was more of an exploratory nature.

The last question that can be asked is whether this is a topic worth researching. Unfortunately, many of the problems mentioned do not become apparent unless the applications have more than a prototype character. Other issues have not yet become problems because there are hardly environments that have many Ubicomp/location-aware applications concurrently running in them. Usability problems are likely to increase with cognitive load.

In addition to that, not much work has been done on the interaction design of Ubicomp applications. The thrust of Ubicomp research has been directed towards software infrastructures. The work presented also opens the door for research in how users will think about Ubicomp environments.

2.5 Conclusion

After having identified the problems this chapter started off with a proposal to use Visualisation in order to solve interaction problems. Previous work on visualisation in Ubicomp was surveyed and the proposal to solve the interaction problem in Ubicomp environments was gradually refined.

The idea of “Invisible Computing” was examined. The final proposition is to use AR in order to make location-aware computing interactive.

The work improves existing applications. Unlike a lot of other recent developments in Ubicomp it goes beyond building a new software infrastructure or reporting on yet another Ubicomp application.

It takes existing applications that are being used, tries to improve them and leaves conclusions behind so that this kind of improvement can be applied to other Ubicomp applications.

Chapter 3

Generating Interactive Virtual Worlds

Advancing knowledge scientifically is about having hypotheses, designing experiments, performing them and drawing conclusions. We are now coming to the experimental part of this thesis. In order to support the thesis that visual augmentation of location-aware applications in a Ubicomp environment is beneficial one needs to show that this is feasible at all.

This chapter describes an engineering task whose aim it was to deliver a system that uses a tracker, a head-mounted display and a desktop computer running some software in order to generate images that give the user the illusion that the computer-generated images are 3-dimensional objects that are located in the real world.

Building an Augmented Reality system from scratch involves making a number of system choices regarding the hardware and software used. Each of the AR systems briefly mentioned in the previous chapter has been engineered to its own requirements, each having its own strengths and weaknesses. Building your own AR system enables you to

- evaluate how feasible the proposed idea is
- tailor the system to your own requirements
- show how the proposed system might be built
- advance the immature science of building AR systems

The system presented in this chapter is a basic AR system. The following chapters describe its evolution as we take into account more and more properties of our target environment. This chapter's aim is to describe the experimental setup facilitating a better assessment and possible replication of the results presented subsequently.

3.1 System Description

The first choice to be made is what kind of tracker to use. Strengths and weaknesses of each technology were presented in the previous chapter. An AC electromagnetic tracker, a Polhemus 3SPACE "TRACKER" [60], was chosen. Its disadvantages are a small operating radius (around 70 cm for good tracking), strong interference due to metallic objects (minimum distance depends on size of object) and the fact that it is tethered. Furthermore the number of sensors is limited to 4, with an acceptable update rate (30 Hz) achievable if only 2 sensors are connected (sensors share bandwidth). Its position and orientation accuracy with in the operating radius is acceptable (0.25 cm, 0.5 deg).

In spite of the mentioned disadvantages an electromagnetic tracker was seen as an ideal solution for initial prototyping. The minimum number of sensors needed to demonstrate an overlay of real and virtual is 2. Furthermore, the used tracker has an acceptable update rate and is highly reliable, i.e. it delivers sensible readings all of the time. Finally, the sensor that tracks the head is quite unobtrusive. Especially, the last two points are important during development at early stages with excessive testing and debugging.

But most importantly, there are tetherless trackers that exhibit similar properties to the tracker used in terms of the software interface and the values they deliver. In order to exploit the facilities of our target environment we have to allow the user to move around inside the building. However, if the system is engineered (architecturally) in a generic enough way, the tracker can just be replaced by one suitable for tracking a person inside a building. Later sections will show that this requirement was indeed met. As a proof of concept, the same system will be made to use a tetherless tracker. Hence, for the conclusions we expect to draw from this exercise the proposed tracker is sufficient.

The head-mounted display (HMD) used was a Sony Glasstron LDI-D100BE. Its resolution is limited to 800 × 600, but is the best available see-through headset. Some of the reasons for choosing Optical See-Through (OST) Augmented Reality, rather than Video See-Through were given in the previous chapter. The system we envisage will be an always-on wearable system. Given that the deployment of such a system will lie in the future it is safe to assume that resolution will improve on see-through HMDs. Also the fact that OST Augmented Reality does not require you to see the world through a camera all of the time makes it a likely candidate for future computing.

The graphics card used was a NVIDIA Quadro2 EX. After some testing on achievable frame rates it was found to be fast enough not to present a bottleneck. Both the HMD and the graphics card are stereo-capable. It will be seen that this is highly desirable for a good user experience, especially when virtual objects are being "placed" on anything else but flat surfaces.

The computer used was a Pentium 800 MHz running Microsoft Windows 2000. All programming was done in Visual C++.

Figure 3.1 shows the setup. The source generates a magnetic field. Sensors inside the field measure it and report it to the Systems Electronics Unit, which derives three position and three orientation values for each sensor relative to the source. This data is then fed into the computer, which uses its graphics card to generate the images. These images are displayed not on its screen, but on the see-through HMD. The relationship between the sensor tracking the head and the sensor tracking the object is used to determine the position and orientation of the virtual object relative to the viewpoint.

3.2 Digression: An Introduction to 3D Computer Graphics

What will be presented in this section can be found in most textbooks on 3D Computer Graphics, such as [61]. Only essential information for understanding the following argument will be presented.

A number of graphics standards have evolved over the past thirty years. The aim of graphics standards is to provide a conceptual framework in order to specify graphical scenes and their transformations in a simple, efficient and uniform way. The two most widely used standards are OpenGL [62] and Direct3D. They provide the developer with an API native to any standard graphics card. Programmers often use higher-level APIs in order to simplify scene construction and management. The use of a scene graph is an example for such a simplification. Scene graphs store all virtual objects and their properties in a hierarchical manner, making full use of the fact that the entire scene can be captured compactly in one tree structure.

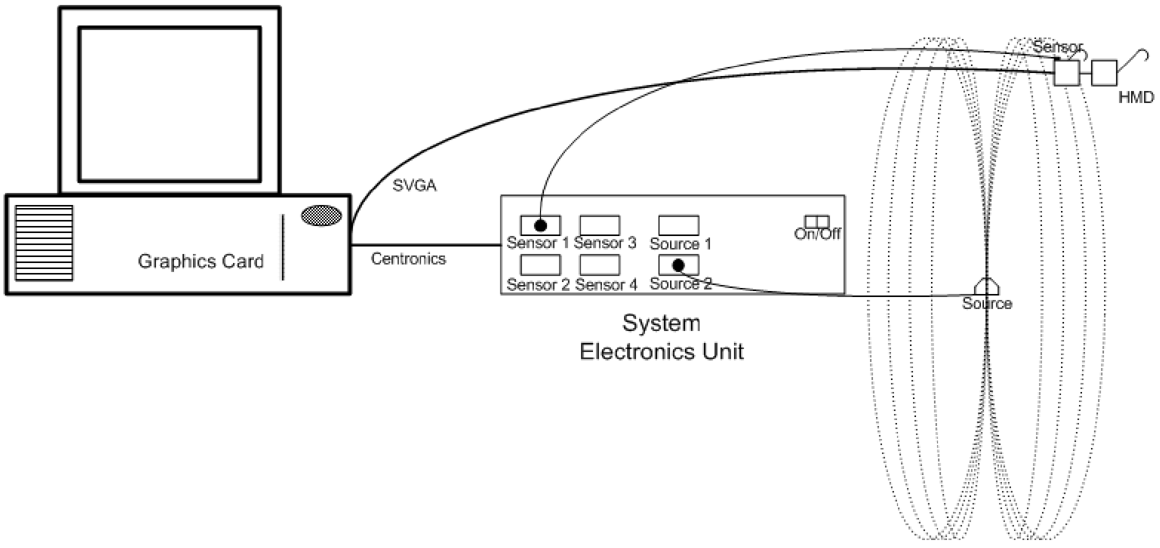


Figure 3.1: Diagrammatic depiction of the system setup

We shall now look at the concepts required to understand 3D Graphics Programming.

3.2.1 Coordinate Systems and Scene Construction

The language of the conceptual framework is Linear Algebra. The body of any scene can be described as a chain of linear transformations of primitive graphical objects. Linear Algebra has been found to be the ideal language to specify and manipulate the scene as well as for calculating the pixel map by the graphics processor.

The three important transforms used in scene construction are rotation, translation and scaling. They form part of the group of affine transforms, i.e. transforms that preserve collinearity and similarity [61]. In order to place a virtual object in a scene you could specify its transform relative to the origin. The convention when specifying these transforms, however, is to use coordinate transformations. You start with a coordinate system and transform the entire coordinate system to some other position, scale and orientation and draw the object using the original coordinates specified by its designer. The coordinate transformation method involves inverse transformations as compared to the "object transform" method.

Transformations are specified as homogenous matrices. Each describes how to convert coordinate values in one coordinate system to another. Multiplying the coordinates of the vertices of an object in one coordinates system by the matrix will result in their coordinates in the destination coordinate system.

A homogenous matrix for 3D graphics is composed of a 4x4 matrix describing the rotation and translation. The forth column describes the translation. Coordinates are left-multiplied with these matrices for transformation. Coordinates are made to carry a fourth coordinate that has been set to 1. In this way one matrix multiplication will result in the point coordinates being multiplied with the rotational part of the matrix and a consequent translation of the resulting point by the translational part of the homogenous matrix.

Figure 3.2 shows a homogenous transformation with the matrix showing the different parts.

Using this notation one matrix can express a rotation and subsequent translation (other transforms are not considered here for reasons of simplicity). The power of this notation unfolds when one is dealing with a number of transformations performed in sequence as may be required when constructing a scene gradually. These sequences can be specified mathematically as multi-

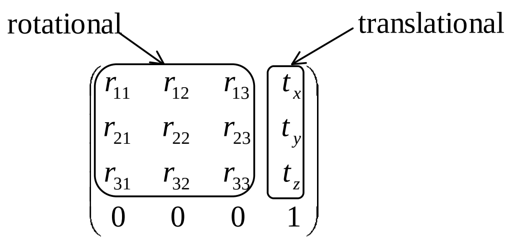

rotational
translational
r_{11} r_{12} r_{13} t_x
r_{21} r_{22} r_{23} t_y
r_{31} r_{32} r_{33} t_z
0 0 0 1

Figure 3.2: Homogenous transformation with the matrix showing the different parts

plications. Any number of transformations can be concatenated in this manner. This is because the rightmost multiplication (a matrix times a point) in a concatenation always results in a point which in turn is left-multiplied by another matrix in order to yield another transformed point etc.

The convention is to call the transformation that converts coordinate values specified in coordinate system A ("A-coordinates" for short) to coordinate values in coordinate system B, T_{BA}. A coordinate transformation from coordinate system B to coordinate system C will be T_{CB}, with $T_{CA} = T_{CB} * T_{BA}$. In another interpretation T_{CA} is the object transform that moves the coordinate axes of C onto coordinate axes of A (note the reverse direction).

In addition to the Modelview matrix OpenGL uses a Projection Matrix. The Projection Matrix is used for the perspective transformation.

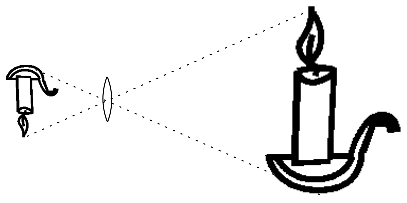
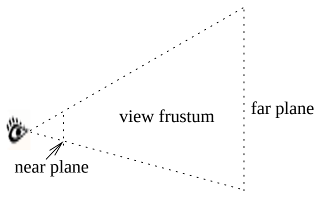
3.2.2 Basic Viewing

The second type of transformation required when presenting a scene to the user is the perspective transformation. A perspective transformation takes a world point and transforms it to a screen coordinate. A number of parameters specify how this mapping is performed and ultimately where the screen coordinate for each world point ends up. Choosing these parameters is equivalent to choosing a lens through which to view the scene. Figure 3.3(a) shows the classical projection through a lens. Multiple world points map to one screen coordinate.

When it comes to computer graphics the user needs to get the illusion that there is a virtual world behind the screen. This is achieved by using a perspective transformation, the difference being that virtual world coordinates are mapped to the screen, rather than real world coordinates. The (human) field of view in 3 dimensions can be regarded as a frustum (for the purposes of projection on a screen). Most graphics APIs allow the specification of this frustum with the graphics package calculating the corresponding matrix automatically. Figure 3.3(b) shows the classical computer graphics projection.

3.2.3 Stereo Viewing

By using stereo the user can be given the illusion of depth. In real life depth perception is guaranteed by the fact that the human's eyes see the same scene from two different viewpoints. In order to achieve a similar depth perception of the virtual world the two frustums need to be aligned properly in order to prevent double vision. This means that when rendering the left eye has to receive a picture rendered using one projective transformation, whereas the right eye needs to receive a picture rendered using another perspective transformation. Figure 3.4 shows why the two frustums for both eyes are different.

(a) Physical projection model (b) Frustum and screen as in OpenGL

Figure 3.3: Projection models

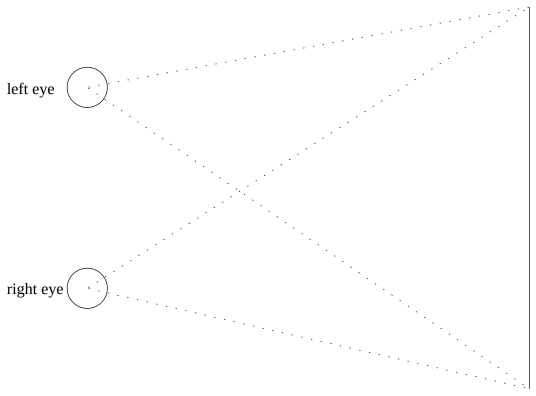


Figure 3.4: Stereo fields of view from above. Right and left frustum have different shapes.

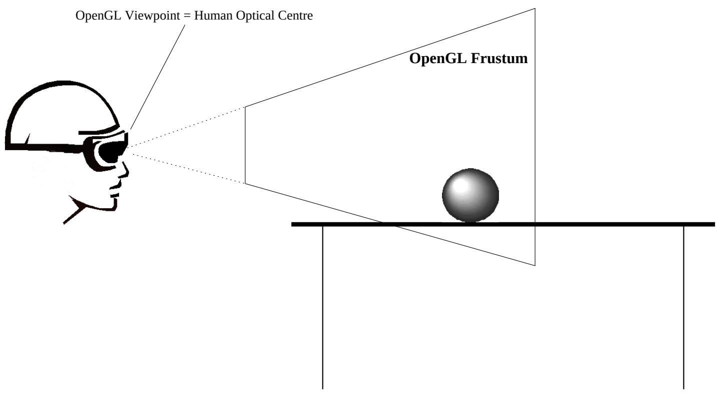


Figure 3.5: A virtual sphere on a real table. The sphere is rendered within the OpenGL frustum which coincides with the user's field of view through the glasses.

3.2.4 Rotation Mathematics

Trackers return values of certain trackable items with respect to a reference point, or coordinate system. A six degree of freedom (6 DOF) tracker will return 6 values: 3 translational and 3 rotational. These six values fully describe the position and orientation of any object with respect to a coordinate system. The translational values in a Cartesian coordinate system include x, y and z. There are many conventions to describe the rotational values. The rotational values we will use are angles called azimuth, elevation and roll.

Let's say the six values have been specified for, say, a sensor of an electromagnetic tracker. Remember that electromagnetic trackers return values of sensors relative to sources. We shall assume both span a coordinate system. Then the meaning of the six values is the following. In order to align the source coordinate system with the sensor coordinate system translate the source by $(x, y, z)^T$, then rotate it around its z-axis by the azimuth angle, then rotate it around its y axis by the elevation angle and finally rotate it around its x-axis by the roll angle.

This kind of rotation specification is called Euler Angles. Another way to specify a rotation by using a matrix. Using the three rotational angles a 3x3 matrix with nine elements can be constructed by using the following formula:

$$R = \begin{pmatrix} \cos A \times \cos E & \cos A \times \sin E \times \sin R - \sin A \times \cos R & \cos A \times \sin E \times \cos R + \sin A \times \sin R \\ \sin A \times \cos E & \cos A \times \cos R + \sin A \sin E \times \sin R & \sin A \times \sin E \times \cos R + \cos A \times \sin R \\ -\sin E & \cos E \times \sin R & \cos E \times \cos R \end{pmatrix} \tag{3.1}$$

3.3 Basic Calibration

3.3.1 Theory

In principle Augmented Reality will involve rotating and translating the scene according to the viewpoint of the user as she moves her head. Her field of view of the real world has to match her field of view of the virtual world in order to achieve an overlay (see Figure 3.5. In order to achieve this match a number of transformation need to be estimated in a process called calibration.

Figure 3.6 shows the transforms involved in order to achieve a match of real and virtual world. Our aim is to place a virtual cube on the source. The perspective transformation P_{SE} converts eye coordinates into screen coordinates. Eye coordinates have their origin in the optical

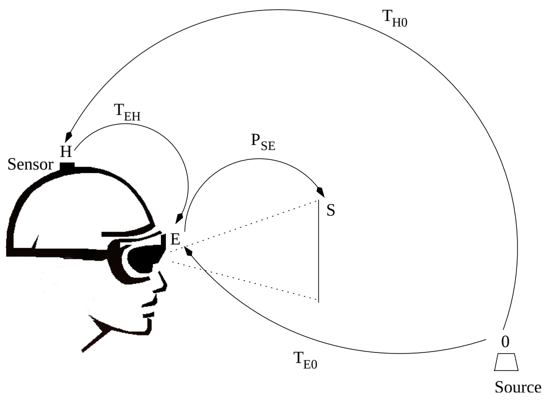


Figure 3.6: Transforms involved in order to achieve a match of real and virtual world

centre of the "viewing system" of the human, somewhere between the two eyes. P_{SE} needs to match the human field of view, or view frustum to be more accurate. The origin and orientation of the eye coordinate system are not readily determinable, but what can be tracked is a point on the head using a sensor. The transform T_{0H} is read from the tracker and used to update the orientation and position of the scene in the virtual world. A simple matrix inversion yields T_{H0}.

One can see that there is a need to determine two transforms: P_{SE} and T_{EH}. Once T_{EH} is determined, we can determine T_{E0} by calculating a transform concatenation by multiplication as seen above: $T_{E0} = T_{EH} * T_{H0}$.

T_{E0} is the transform that is required to move the virtual cube from the origin of the virtual world to the our source, keeping in mind the origin of the eye coordinate system is identical to the origin of the virtual world for overlay. By left-multiplying P_{SE} with T_{E0} we get the transform T_{S0}, the transformation of the virtual cube origin to screen coordinates. In order to render the cube, this is the transform the graphics processing unit has to use when transforming each vertex of a virtual cube. Due to the fact that it contains a dynamic component T_{H0}, the entire transform will be changing as the user moves her head.

We now have the problem of estimating the two transforms P_{SE} and T_{EH}, one describing a projection, essentially the human view frustum (remember we are projecting the field of view on a screen, hence frustum), and the other T_{EH} describing the transform required to transform head coordinates into eye coordinates, head coordinates having their origin on the sensor. This remains constant throughout the operation, given that the spatial relationship between the sensor and the user's head remains constant throughout.

In order to estimate the transforms we shall use a method well known from the field of Computer Vision: camera calibration. Camera calibration is used to estimate the intrinsic parameters of a camera. Figure 3.7 shows the pinhole camera model used for such a task. The projection screen has a coordinate system with axes u and v. Each real world point $(x, y, z)^T$ is mapped to a point (u, v) with the mapping being a projection matrix. The projection matrix is characteristic for each camera since each camera has its own specific field of view. The aim of calibration is to determine this. The projection depends on a number of parameters, called the intrinsic parameters.

The intrinsic parameters consist of the focal length f (in mm), the screen coordinates (u_0, v_0) of the image point of the optical centre (in pixel), constants k_u and k_v (in pixel/mm). In the convention used here f is always a positive number. It is important to know that the non-ideal camera's coordinate system (u, v) may be slanted.

Another way of specifying the projection is by a projection matrix using homogenous coor-

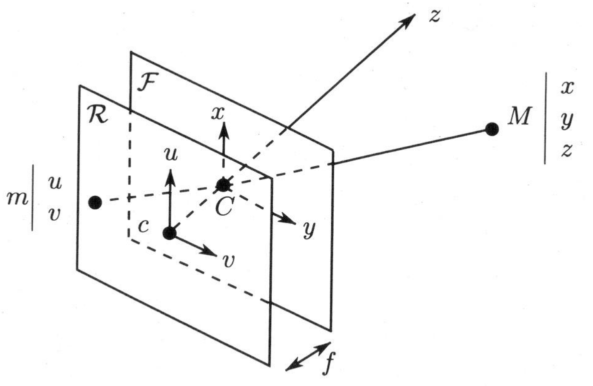


Figure 3.7: The classical pinhole camera model. R is the screen inside the camera, F is the focal plane containing the origin of "camera coordinates", also called the optical centre C (from [63]).

dinates. Equations 3.2 show the how a real world point $(x, y, z)^T$ in homogenous coordinates is mapped to a point on the projection screen $(U, V, S)^T$ in homogenous coordinates. As a reminder, homogenous coordinates carry around an additional constant as a fourth coordinate. Using homogenous coordinates for projection is described in [63]. For our purposes, just expand the matrix equations and you will get the simultaneous equations you might expect from a transformation that involves perspective foreshortening. In order to recover the actual coordinates from homogenous ones you divide by the constant.

$$\begin{pmatrix} U \\ V \\ S \end{pmatrix} = \begin{pmatrix} -fk_u & 0 & u_0 & 0 \\ 0 & -fk_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.2a}$$

$$\begin{aligned} u = \tfrac{U}{S} &= -fk_u \tfrac{x}{z} + u_0 \\ v = \tfrac{V}{S} &= -fk_v \tfrac{y}{z} + v_0 \end{aligned} \tag{3.2b}$$

Equations 3.2 have not taken the slant into account. It can be shown [63] that taking the slant θ into account, the projection matrix converting camera coordinates to screen coordinates can be re-written as Equation 3.3, making the total number of intrinsic parameters 6.

$$P_{SC} = \begin{pmatrix} -fk_u & fk_u \cot\theta & u_0 & 0 \\ 0 & \frac{-fk_v}{\sin\theta} & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.3}$$

The coordinates (x, y, z) are measured in a coordinate system that has its origin in the optical centre of the camera. The x and y axes are aligned with the u and v axes respectively. Calibration will essentially involve solving the Equations 3.2 (not forgetting the slant). In order to achieve this one chooses different real world points and finds out to which screen coordinates (u, v) they map. If this is done for a sufficient number of points all intrinsic parameters can be determined. In order to achieve a higher accuracy one can choose more than the minimum number of required points (2 equations for each point, 6 unknowns, so 3 points required) and solve using the numerical least-squares method.

There is however a problem. Screen coordinates in pixel can be determined by looking at the camera image, but the real world coordinates (x, y, z) are relative to the optical centre of the camera. Since neither the position of this point is known nor the orientation of that coordinate

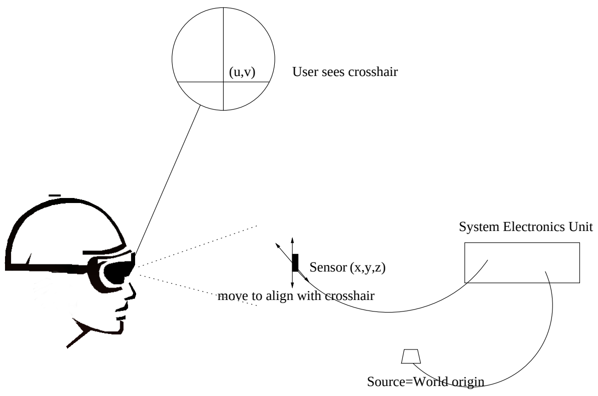


Figure 3.8: The calibration setup

system, we are dealing with more than 6 unknowns. The camera coordinate system's rotation and position introduce an additional 6 unknowns (3 coordinates for the position, 3 angles for rotation). These are called extrinsic parameters and just represent a transformation T_{CW} from the world origin to the camera origin as discussed before.

Therefore, when calibrating the camera we have to choose more points and solve for the 6 extrinsic unknowns as well, keeping in mind that the relationship between the extrinsic and intrinsic parameters with regards to the total transform is $T_{SW} = P_{SC} * T_{CW}$.

Using this knowledge, we recognise that the camera calibration problem is equivalent to determining matrices P_{SE} and T_{EH}, P_{SE} being equivalent to the projection matrix with the intrinsic parameters and T_{EH} being equivalent to the roto-translational matrix containing the extrinsic parameters.

As hinted earlier the method of determining these matrices is by sampling points relative to a fixed world coordinate system and recording the resulting image coordinates. Now the roto-translational matrix made out of the extrinsic parameters will describe the coordinate transformation between the world coordinate system and the camera coordinate system. Each point will result in 2 equations, meaning that we need at least six points. In fact the rank of the matrix M is 11, not 12 and there is an unlikely case in which 6 points are not sufficient. More details regarding this can be found in [63]. We shall call this method Faugeras' method from now on.

In general you get more accurate results the more points you choose. Having an over-specified system of linear equations to solve for the extrinsic and intrinsic parameters will have the advantage that one can use the least-squares method to get a more accurate result.

3.3.2 Implementation

We shall now come to the implementation part. A number of Optical See-Through calibration procedures have been implemented ([64], [53], [65]). The setup shown in Figure 3.8 can be used in order to run the calibration procedure:

The user sees crosses on his HMD crossing at a specified screen coordinate. The user moves the sensor using her hand until its origin's image falls on the specified coordinate. Its origin is the point whose position readings are forwarded to the computer through the tracker.

After the user has matched a number of real world points with given screen points, the projection matrix for each eye is determined. The same procedure has to be performed for the other eye as well. Ten points were seen as enough to calibrate one eye.

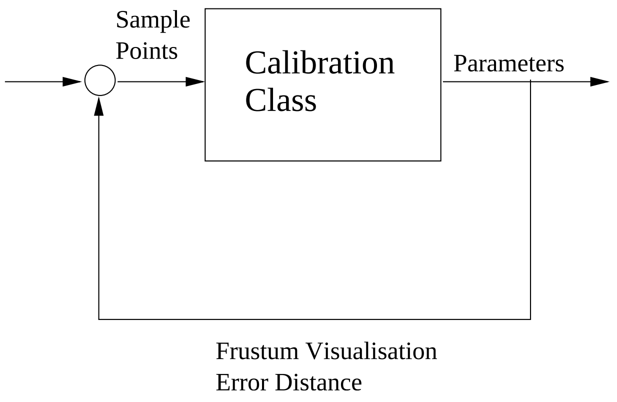


Figure 3.9: Opportunities to improve calibration by using feedback to the user

3.4 Interactive Calibration Procedures

When it comes to putting theory into practice we are inevitably faced with imperfections of the real world. They result from assumptions made about the physics of equipment and measurement.

Figure 3.11 shows a picture of the HMD. It can be seen that the HMD, which in principle is not much bigger than a pair of large sunglasses, had to be removed from its original enclosure and fitted into a plastic helmet. The reason for that is that even the small amount of metal contained in the original covering was enough to interfere with the tracker's readings leading to an inconsistent calibration, which in turn led to a highly distorted overlay.

It was also seen that calibrating both eyes for stereo has the disadvantage that the calibration needs to be even better, since the user's eyes need a very good match of right and left view in order to have depth perception.

In each case, it can be seen that the calibration process is the crunch point as far as errors are concerned. Inaccuracies can occur throughout the application, but in no other part do they have the chance to accumulate and introduce a bias in such a manner. Even tiny errors during calibration can result in large offsets and distortions in the final image. This lies in the nature of the calibration method chosen. A small number of points is chosen in order to calculate the view of the human eye (actually it is the view frustum of the combined HMD-eye viewing system since we are not using projection points on on the retina). A misalignment in the magnitude of pixels on the screen can be equivalent to errors in the magnitude of centimetres in terms of 3D coordinates, depending the sensor's distance from the HMD screen. As a comparison: The tracker used has an accuracy of 0.25 cm.

Sampling more points will lead to more accurate results after the least-squares error fitting process. However, there is a limit to how much effort can be expected from the user. So, there is a tradeoff of usability and achievable accuracy.

One the other hand, we can try to convert the apparent weakness of having a user in the loop into a strength. If we can provide feedback to the user about the quality of the calibration we can let her gradually optimise the calibration. Figure 3.9 shows the general idea. So far, measures for the quality of Optical See-Through calibration have only been introduced in off-line evaluation. One notable exception is by McGarrity et al. [66]. They devise a method in order to calculate the distance between a reference point and its virtual counterpart (output point) as provided to the user as an overlay by the system. This is taken as a measure for the quality of the calibration.

The system implemented here uses a different measure in order to provide feedback. The aim

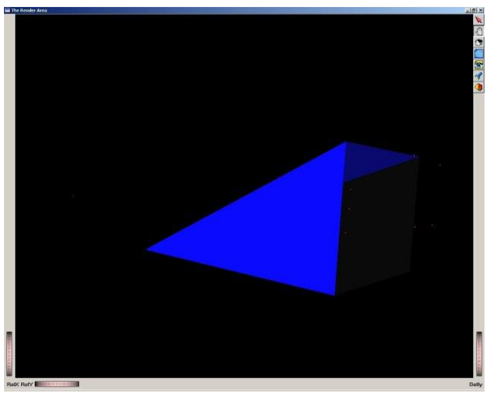


Figure 3.10: Frustum visualisation tool

was not to put more load on the user by asking her to go through another evaluation procedure, selecting various points, but to provide a single instantaneously available quantity that will offer enough information about how good the calibration is.

The idea is to use the projection matrix calculated and feed back the sample points into the matrix in order to calculate the total error. Effectively, this is similar to McGarrity et al.'s method. The difference is that the euclidian distance between reference and output is evaluated on the projection screen in pixel rather than on a evaluation board. And, the user does not need to perform an extra calibration evaluation procedure since we are using the same points the user sampled the first time around. Also, the user is not made to rely on her perception again during evaluation. Aligning points can be strenuous and inaccurate. The feedback the user gets is a single number showing the average euclidian distance in pixel between reference points and output points. If the distance is less than 5 pixels the calibration is good. A distance above 10 pixels needs recalibration.

The calibration of both eyes should be performed independently, giving the user the chance to recalibrate just one eye.

Another method to adjust calibration errors explored was to give the user the ability to manually adjust the estimated matrix. The user was given the chance to adjust its 12 parameters by key strokes and then compare how the overlay changed. It was, however, found that this method is too unpredictable and the user lacks understanding as to what effect a change in a particular parameter will achieve.

The general nature of calibration (=sampling points in two coordinate systems and calculating a conversion matrix) and its evaluation lends itself to further abstraction. It proved useful to separate all calibration code from the application code and compact it into a calibration class that provides various other facilities such as saving, restoring, analysis etc. as well.

One example of what kind of analysis can be done on calibration data is shown in Figure 3.10: a frustum visualisation tool.

This tool takes the data retrieved from the calibration process and automatically calculates the resulting frustum. It visualises this together with the transformation T_{EH} from Figure 3.6. The sampled real world points are visualised as dots and all need to lie within the frustum. By rotating the frustum so that one looks directly though it one can see whether the real world points actually project to the pre-specified ones. The visualisation also shows the slant of the frustum.

The tool is more suited for developers. It gives them a more holistic figure of the effects of imperfections. It also helps them "debug" a "blank screen" effect, since it shows them where overlays are placed with respect to the head.

3.5 Rendering

In order to render the scene in a way that the virtual world the user sees through her HMD overlaps with the real world, we need to make use of two transforms mentioned earlier: T_{E0} and P_{SE}. Please refer to Figure 3.6 and its explanation.

T_{E0} can be obtained from T_{H0} by using the relationship: $T_{E0} = T_{EH} * T_{H0}$. Since T_{H0} is readily available as tracker readings (after a simple matrix inversion) the problem we are left with now is determining T_{EH} and P_{SE}. T_{SH} is the total transformation from head coordinates (= sensor coordinates, measured in the coordinate system spanned by the sensor on the users head) to pixel screen coordinates: $T_{SH} = P_{SE} * T_{EH}$. By separating T_{SH} into the two components we can obtain both unknown transforms. T_{SH} can be obtained by camera calibration. It is the transform between head coordinates and screen coordinates. So, we sample a point in head coordinates and record its screen coordinates and numerically solve for the matrix.

In order to perform this separation we can use the method used in the ARToolkit [53] software. Projection matrices are known to be upper diagonal matrices [63], i.e. as you go through its rows from top to bottom, the number of leading zeros in the row increases by one, starting from no leading zeros in the first row. There is a matrix decomposition called Q-R decomposition that separates matrices into a product of an upper diagonal and another matrix. Since this decomposition is unique, we will be indeed left with the projection matrix P_{SE} and T_{EH}. With this we have obtained all necessary transforms.

We can however not use P_{SE} directly for rendering. OpenGL and all APIs based on OpenGL only handle normalised rectangular frustums with coordinates between -1 and 1. OpenGL provides facilities to generate these frustums from a specification of the six planes defining it, but since we are using a non-standard skewed frustum OpenGL cannot convert it to a normalised frustum. The reason for why our frustum is skewed is because our projection describes how to project a real world point to the HMD screen. The combined viewing system of HMD screen and the eyes looking at it is generally not specifiable by a straight projection.

Using the same equations OpenGL uses for its conversion [61] one can perform the normalisation manually before passing the matrix on to OpenGL. The ARToolkit software performs this in the following way. Given that matrix P with elements p_{ij} is the matrix obtained from the decomposition process, the normalised projection matrix can be obtained as in Equation 3.4.

$$P_{opengl} = \begin{pmatrix} \frac{2p_{11}}{wp_{33}} & \frac{2p_{12}}{wp_{33}} & \frac{2p_{13}}{wp_{33}} - 1 & 0 \\ 0 & \frac{2p_{22}}{hp_{33}} & \frac{2p_{23}}{hp_{33}} - 1 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.4}$$

The equation contains some parameters specific to OpenGL, such as f and n. These describe the far and near plane respectively [62]. In 3D graphics the programmer has to specify a view volume (a frustum for perspective projection). Everything in the virtual world outside this view volume is clipped, i.e. ignored when rendering. The near and far planes give the front and back limits of the view volume. For the purpose of this system these were set to 5 mm and 5 m. The variables h and w specify the resolution of the HMD in height and width. Lowercase p elements are from the projection matrix obtained from the decomposition process.

The Studierstube software makes use of the standard way to specify frustums in OpenGL. This means their frustums have to be straight. In order to obtain an accurate projection matrix in spite of freezing one of its elements to 0 (remember that element p_{12} in the projection matrix describes the slant of the image coordinate system) they use a different calibration procedure [64]. This has advantages since many higher-level APIs will only accept standard frustums.

Due to its more powerful scene management and construction facilities, it was decided to use a higher-level API called Open Inventor [67] for the rendering. Open Inventor (OIV) uses OpenGL but lets the programmer use a scene graph and also provides a language to specify 3D Models. With this facility any existing 3D Model of an object can be inserted to the scene. This is particularly useful since it opens the door to numerous 3D VRML (Virtual Reality Modeling Language) models already on the web and gives future developers the chance to design more sophisticated models for use in AR applications.

Integrating higher-level APIs into Augmented Reality systems is not always straightforward, since they usually assert control over the whole application in order to perform standard graphics tasks in the background transparent from the user. OIV lets the programmer specify a scene graph and performs the actual rendering of each frame in a loop automatically. In order to integrate OIV it was necessary to intervene in the rendering process.

One change that was necessary, was to perform a callback for each frame in order to read the tracker values and set the transform T_{E0}. Another task was to modify the library so that it supports stereo rendering using our skewed projection.

So far, we have just considered placing a virtual object on the source using the Transform T_{E0}. When constructing virtual worlds we need to place many more objects on many more places. The best way to construct a virtual world is to use such a point as the reference point of the entire scene. T_{E0} is constantly updated with each frame. Therefore all transforms that remain fixed with respect to the source do not need to be changed if we use the source as the reference point of the whole scene.

3.6 System Performance

The description of a basic AR system is now complete. Some further evaluation was performed in order to ensure the system was at least acceptable for the task.

Simple test applications were written in order to test how good the overlay was. Users were allowed to test the system at numerous occasions. Generally the overlay was acceptable with maximum offsets of about 1 cm. The main reason for these offsets is interference from metal in the Lab. This was concluded since the offsets were roughly the same whenever the experiments were performed. There are ways to compensate for these interferences systematically [68], for example through a lookup table of corrections at various points. This was however not deemed necessary, since the system only had prototypical character as laid out above. One of the requirements was that the system is independent from any particular tracker, so perfecting the use of one particular tracking technology was not an interesting area to explore.

Other sources of error are inaccuracies of the estimated matrices and the tracker's intrinsic inaccuracies.

The test applications overlaid a virtual object on the source and on a moving sensor. It was seen that the tracker measurements were very robust, making the overlays very stable. Real world objects static in the scene were labelled with text in order to demonstrate that virtual objects can be placed anywhere in the real world. For this the positions of real world objects had to be measured with respect to the source. Since the Lab has an Active Bat system (see Section 2.3.1) this task can be performed very easily.

It was seen that good calibration is especially necessary for objects that are a couple of metres away. With good calibration it was possible to overlay virtual objects on things that were at a room-scale distance.

Initially the frame rate was found to be very low. But by replacing the graphics card and

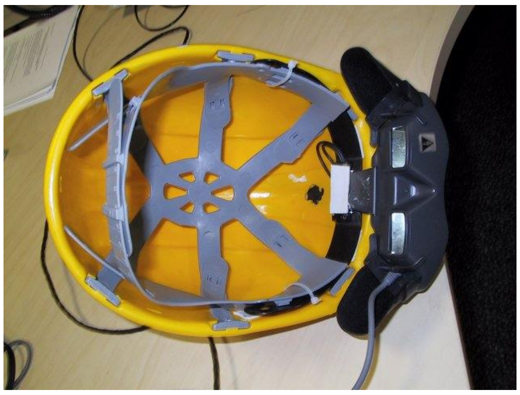


Figure 3.11: Close-up view of helmet

re-writing the tracker driver frame rates close to the tracker's theoretical limits were achieved. The important conclusion to be drawn from this is that 3D graphics, on which AR is based, is not the bottleneck of the system.

The mobility limitations of the system have already been discussed. Other factors influencing the user experience were bulkiness of the helmet (see Figure 3.11) and the relatively low resolution. Technology is improving steadily in both respects so that it is safe to assume that light-weight glasses and trackers will be available in the future.

Finally, the calibration procedure was found to be quite lengthy and tedious for the inexperienced user. The most important improvement here was to save a calibration once performed. Even though you get more accurate results by performing the calibration every time the system is used, it is quite acceptable to use a previous calibration and move the HMD around a bit in order to align real and virtual world. The reason for why you get more accurate results if you perform the calibration every time the helmet is put on or moved is that the transform T_{EH} (see Figure 3.6) changes with movement and should be estimated again if the sensor does not end up in the same place in relationship to the eyes of the user.

3.7 Conclusion

In this chapter an overview of the main problems frequently encountered while engineering an Augmented Reality system was given. The different ways researchers in this field have taken in order to tackle typical tasks were presented, some in greater detail, others were provided as references. Presenting these approaches gives the reader the opportunity to appreciate the system choices made.

The author has tried to provide a manual for others to build an Augmented Reality system, given that such "manuals" are not readily available. In addition to that the conceptual framework needed to understand the implementation of Augmented Reality was provided. The chapter provided in a very concentrated form the technical know-how needed to implement such a system.

The calibration problem was recognised as to be at the core of Augmented Reality. While other aspects in AR have improved over time, calibration still remains a cumbersome and seemingly ad-hoc task. Some new avenues to explore were presented in this respect.

In short this chapter has provided the reader with most of the knowledge needed to embark on the task of implementing a very basic AR system.

Chapter 4

Tracking Abstraction for Office-Scale AR

The last chapter dealt with building a basic AR system. Its main downside is that it is limited to a small operating radius. In order to deploy AR as an interface to an entire active environment as envisioned we need to work with a tetherless tracker. The claim that the implemented AR system can be extended to one that uses tetherless tracking shall be substantiated in the following.

We shall also consider other properties of our target environment and investigate how our AR system can be integrated in terms of making use of available tracking facilities. The field of integrating AR systems into environments with existing tracking facilities has only recently evolved.

The idea of regarding tracking as a service of an environment, only came about after cross-fertilisation of AR and Ubicomp. The future is going to see an increase of tracking facilities available in indoor environments. At the same time their variety in terms of technology used, sensor reading formats, degrees of freedom provided and software interfaces is going to increase likewise.

We shall see how we can manage tracking heterogeneity in our target environment and use sensor readings from multiple tracking systems.

This chapter can be regarded as a proof of concept chapter that shows how our system can be adapted to circumstances our target environment is likely to exhibit. The first part of this chapter will deal with the implementation of a tetherless tracker. We will then use our experience gained in order to identify valuable abstractions and come up with a generic software architecture for our system.

4.1 System Idea

The starting point of our development will be the ARToolkit software [53]. ARToolkit is being used for Augmented Reality in academia and research. Its main advantage is that it is cheap and yet achieves good registration. It is based on computer vision. A cheap webcam can be used in order to track markers. The software processes each frame from the camera and returns T_{CM} the transformation between the camera and the marker. Markers are square and have a very thick boundary in order for the software to reliably calculate its orientation and position with respect to the optical centre of the camera.

ARToolkit has the ability to recognise different patterns within the boundary. In this way it can determine which marker it is calculating the position/orientation of. It can also handle multiple markers in a frame. Figure 4.1 shows such a marker.

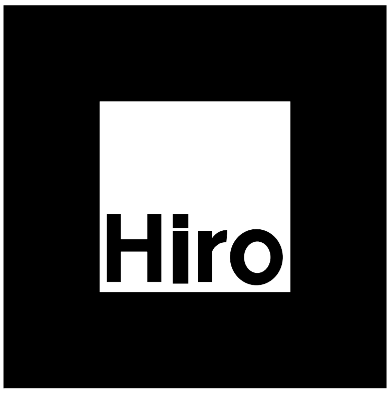


Figure 4.1: An ARToolkit marker

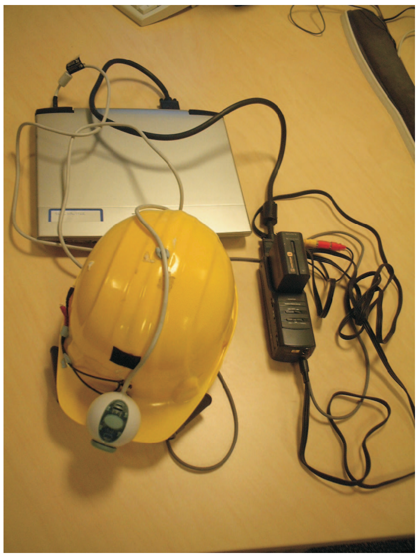


Figure 4.2: A photo of the equipment required

With a standard computer and graphics card, frame rates of up to 30 Hz can be achieved. Its position accuracy has been found to be 20 mm (for a distance of about 2 m). Its orientation accuracy has been analysed in [69].

ARToolkit uses pure OpenGL for rendering. The calibration procedure described in the previous chapter was based on ARToolkit. The difference is that there is no absolute source, just a camera and markers. The camera needs to be carried on the user's head so that it can recognise markers in the user's field of view, calculate transforms T_{CM} for each marker and use OpenGL in order to place virtual objects on them. This is the basic functionality of ARToolkit: placing virtual objects on pre-defined markers. Furthermore, ARToolkit provides functions to undistort the camera images for more accuracy and functions to calibrate the camera.

If the user can carry a battery-powered notebook in a backpack and a battery powered HMD with a USB camera the whole unit becomes portable, allowing the user to freely move indoors. Figure 4.2 shows a picture of the equipment needed to be carried around by the user.

We have seen that ARToolkit can be used in order to place virtual objects on markers. For many Augmented Reality applications this is sufficient. Markers represent something only users with the HMD can see, for example some models of objects a group of co-workers is manipulating in order to build a bigger model [70].

But for our purposes this is not sufficient. We want the user to make sense of an active "invisible" environment. In some cases we can tag an active device, say a printer, with a marker and place a visualisation relating to the printer's state on top of it. Or even better, we can tag an active device and try to overlay the virtual object on say its buttons. However, for this we would need to save the geometry of the device somewhere. The question is what would happen if we view the printer from the side that does not display the marker. We could have another marker on that side and save the printer geometry relative to this marker as well. Or, easier, we could save the printer geometry once and save the position of all its markers. But even then, for ARToolkit to work properly, we would require the user to be at most about a meter or two away from the printer at all times. Maybe it would be possible to infer from another marker at some completely different position in the room whether the printer buttons were in the user's field of view, and if so, where on her HMD to place them. The *general problem* is how to augment possibly imperfect information received from a tracker with information known about the setup of the environment.

This is what the rest of the chapter is all about: dealing with frames of references, saving geometry, handling multiple trackers and how to bring all of this under a well-crafted software architecture. Note that all the problems mentioned above are typical for office-scale Augmented Reality. Nearly all conventional applications are limited to a small area such as a table and these problems do not apply.

4.2 A Cheap Optical Tracker to Track Head Orientation and Position

The first step is to design and implement a tracker using the ARToolkit software. ARToolkit is a complete Augmented Reality solution that performs calibration, tracking of markers and rendering all in one, but is not exactly a universal tracker. We would like a tracker to work much the same way as conventional trackers such as the electromagnetic one presented in the previous chapter. These trackers return a constant stream of position/orientation values for a number of known trackable objects with respect to a particular reference point.

By separating the marker tracking functionality from the ARToolkit software and repackaging it we can construct a stand-alone optical tracker. In order to achieve this we will need to cover the entire space with markers. The need to cover the whole room arises from the fact that one camera or one marker is not enough to deliver a constant stream of values for any position/orientation of the head in the room. Since we would be dealing with so many different reference points we would need to save positions of these references so that the tracker returns consistent values as it is "handed over" from one reference point to the other due to a considerable movement of the head.

So, the room will be covered with markers. The camera will pick these markers up as the user moves through the room. Since the positions of the markers will be saved, the tracker can return values, always absolute to a particular reference point, which simplifies using the tracker a great deal. Figure 4.3 shows the working of the system. In order to obtain values relative to our reference point R we need to determine T_{RC} $(= T_{RM} * T_{MC})$.

ARToolkit's tracking code returns the transform T_{CM}, i.e. the transform required to convert marker coordinates into camera coordinates, or the transform required to align the camera with the marker. T_{CM} is a standard matrix of the type we encountered in the last chapter. T_{MC} is the inverse of T_{CM}. It can be obtained by matrix inversion. A small test application was written in order to test if at least a camera's position can be tracked reliably. Four patterns

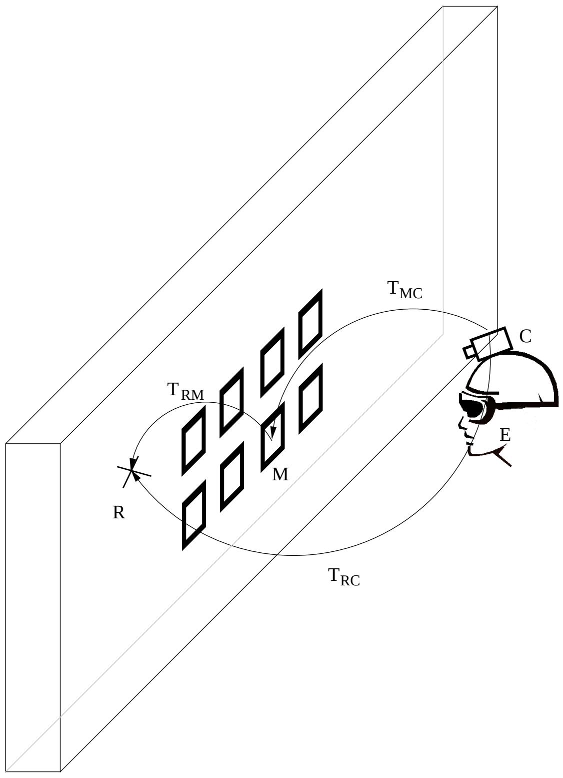


Figure 4.3: Tracking a user's head in a room by inferring its position from markers in the room

were aligned and placed close to each other. The camera was moved above the patterns, making sure all four were always in view. Internally, the four transforms, one to each marker, were used to calculate position estimates of the camera from a fixed point of reference arbitrarily chosen (from T_{RC}). Figure 4.4 shows the experimental setup.

This is not the best way to evaluate the accuracy of a tracker since we are not using any absolute value, but for a quick feasibility test this is sufficient. Each marker would "report" the camera position in the previously chosen point of reference. It was found that in some samples there were large discrepancies between the estimates (up to a few centimeters).

The discrepancies were surprising since the *observed* accuracy in the applications built with ARToolkit show extremely good registration.

It is generally difficult to estimate tracker properties from a number of samples. A "test drive" in the form of a small sample application can reveal much more. The idea was to create a visualisation on the screen using the sensor readings. While you would move the camera in its trackable region the visualisation would give you visual feedback as to how the tracker reacts to your movements. These kinds of visualisations can help AR developers a lot when it comes to understanding the properties of a particular tracker.

The discrepancies were confirmed but were not as bad as it first seemed. The reason for that was that occasional extreme outliers were not noticeable at a rate of 30 Hz. Nevertheless, it was still found to be inaccurate for our purposes with absolute inaccuracies of a couple centimeters.

The reason for this can be found in the nature of the matrix inversion. Equation 4.1 shows how to invert a matrix that contains a rotational and translational part. Normally inversion is a more complex process, but for these kinds of transformation (called affine orthogonal transformations) an inversion is easily accomplished.

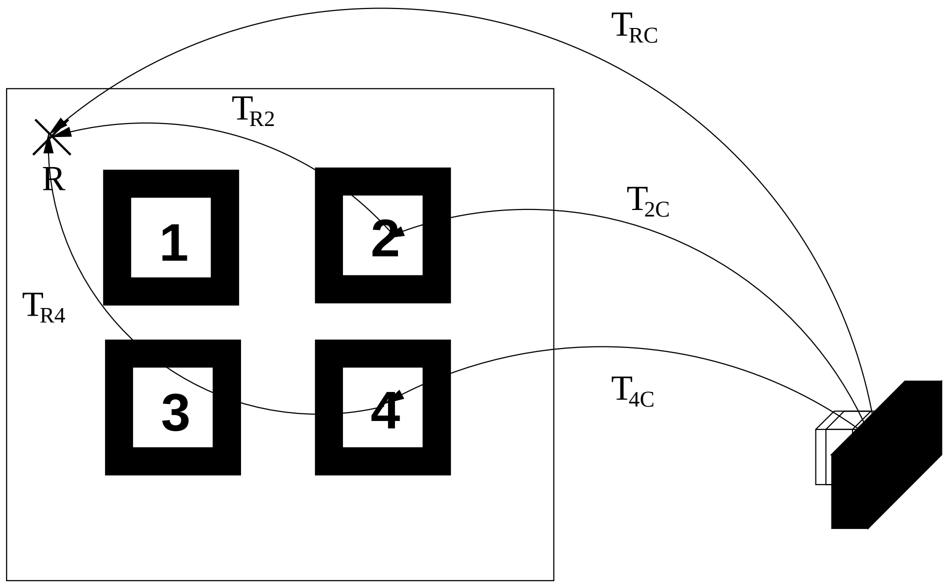


Figure 4.4: Evaluating expected positional accuracy of the proposed tracker. Calculations of T_{RC} were performed for all four markers and compared.

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} r_{11} & r_{21} & r_{31} & -r_{11} * t_x - r_{21} * t_y - r_{31} * t_z \\ r_{12} & r_{22} & r_{32} & -r_{12} * t_x - r_{22} * t_y - r_{32} * t_z \\ r_{13} & r_{23} & r_{33} & -r_{13} * t_x - r_{23} * t_y - r_{33} * t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

One can see that the translational part of the inverted matrix is a weighted multiplication of elements of the rotational part of the matrix T_{CM}. Please refer to Figure 3.2 in order to identify the two components (translational and rotation) of such a matrix. ARToolkit does in fact have orientational inaccuracies. These result in errors in the 9 rotational elements of the matrix T_{CM}. After matrix inversion these erroneous elements are multiplied and also become errors in position. When you are using ARToolkit in its original purpose, to overlay virtual objects on markers you hardly notice the orientation errors and you get the impression that the software has a remarkable accuracy. Inverting the matrix, however, brings out these errors in a magnified form!

By analysing the internal workings of ARToolkit it was possible to identify two points that could be used for targeted optimisation: the calibration procedure and the undistortion code that undoes camera lens distortion effects.

In an attempt to alleviate some of these difficulties it was decided to rewrite parts of the ARToolkit code using an industry-standard Computer Vision library: Intel's OpenCV library [71]. The idea was to replace the camera calibration code and undistortion code with OpenCV functions.

Camera calibration in OpenCV is performed in a simpler way. Numerous pictures of a calibration pattern are taken with the camera to be calibrated and the software automatically calculates the intrinsic and distortion parameters. For ARToolkit the user has to identify pixels that points on a calibration pattern captured by the camera are mapped to.

Using the same methods as above, the results using the modified tracker were much better.

One problem that did remain, however, was not enough robustness. The tracker was used in an Optical See-Through setup to put virtual objects into the environment and it was noted that the virtual objects seem to "shake" due to high frequency noise in the tracker values. A low-pass averaging filter was programmed that at any one time delivered a weighted average of the past ten raw tracker values. This resulted in some smoothness, but also in slowing down the update rate of the scene.

4.3 A Tetherless Indoor Tracking System

The next step was to deploy the tracker. Only part of a wall was covered with markers. The markers were surveyed. In order to make the new tracker integrate tightly into the existing infrastructure in the lab the benchmark application was to achieve an overlay on an Active Bat. The tracker was considered "good" enough if the overlay was deemed acceptable by test users.

An extra difficulty is involved in achieving the overlay. The readings from the Active Bat system need to be correlated to the readings obtained by the newly designed head tracker.

If we wanted to use our tracker in order to obtain position/orientation values of the camera in the Active Bat system frame of reference we would get T_{MC} from the tracker, i.e. the transformation from the head-mounted camera to the marker. Then there would be T_{0M}, the transformation from a reference point to the marker. In our case this reference point needs to be the Active Bat system origin. Only then will our tracker return values in the same reference frame as the Active Bat. Using both we can calculate $T_{0C} = T_{0M} * T_{MC}$, the position/orientation of the camera in the Active Bat system's frame of reference.

In order for our Augmented Reality application to work we need the transform T_{EC}, the transform between camera coordinates and eye coordinates and T_{SE}, the transform to screen coordinates. This we can find through calibration as described in the previous chapter. We need to sample points in camera coordinates and screen coordinates.

Figure 4.5 shows all transformations involved in implementing the benchmark application. For generality, we shall assume that we have arbitrarily chosen a reference point R in order to survey the markers.

The problem that needs to be solved for the benchmark application is to find T_{EB}, the transform between the user's eyes and the Active Bat. Using Figure 4.5 we find $T_{EB} = T_{EC} * T_{CM} * T_{MR} * T_{0R}^{-1} * T_{0B}$. Start reading the equation from the right and follow the path through on the diagram.

The next step is to extend this system even further. Adding new markers to the system was even further simplified. The user just had to enter positions of the markers she had obtained from Active Bat readings and the software would automatically name the markers, update its database and provide a marker. There was no need to train the system for each marker since a utility was used that provided the software with training data for dozens of pre-specified markers [72]. These improvements were vital in reducing the marginal effort needed to be put in when expanding the system to cover larger areas.

By these experiments we have shown that it is possible to implement a cheap tetherless indoor tracking system that can track head position and orientation to a degree sufficient for Augmented Reality applications. Furthermore, we have demonstrated how to interpret tracker readings from one system for another tracking system. So far, we have performed this task twice in an ad-hoc manner: once for combining the Active Bat system with the electromagnetic tracker (Chapter 3) and once with the tracker based on ARToolkit. We shall now see if we can find a more systematic way to categorise and implement the ideas presented in this chapter so far.

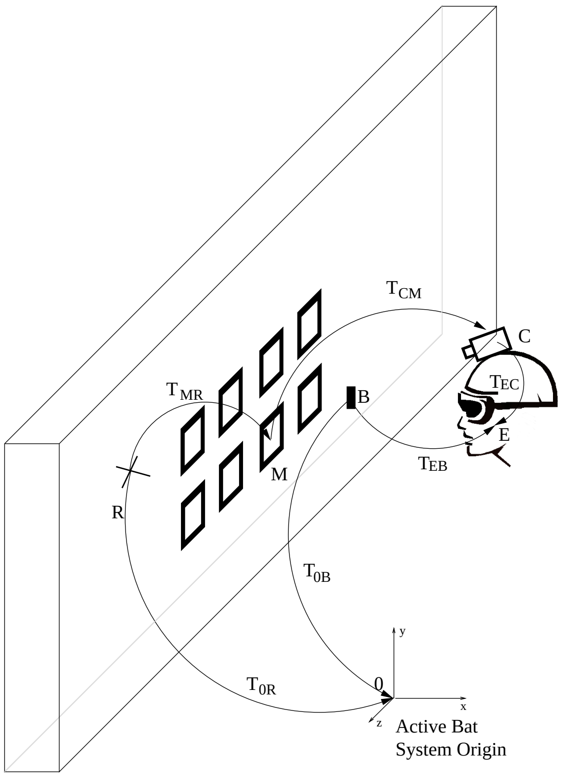


Figure 4.5: Transformations involved in implementing the benchmark application

4.4 A General Tracking Abstraction Architecture

In the past various researchers have developed location management frameworks in order to facilitate the use of location information for developers of location-aware applications. Acting as mediators between technology developers and application developers, they have successfully managed to create interfaces for, abstract from and integrate the underlying location technologies in systems such as SPIRIT [73] or QoSDream [74]. The *main benefit* of these frameworks is that they simplify the task of building location-aware applications.

These frameworks were put into place, mainly to manage location information at room granularity provided by the then current technology, the Active Badge [12]. The development of a number of fine-grained indoor location systems [75] in recent years is an indication of not only a natural advancement in technology, but also of a shift in focus of requirements. The main thrust of research is not geared towards finding people in buildings anymore, but towards using location technologies as enablers for rich interaction in computationally enhanced environments.

In the following it shall be shown how location management frameworks might be adapted in order to accommodate applications we envisage. The experience gained in using different tracking systems and generalising object positioning as presented in the previous sections, will be invaluable in accomplishing this task. Many of the ideas used have flowed into the design of a generalised tracking architecture. Before we think about how to extend existing location management frameworks, let's have a look at the state of the art.

4.4.1 Modelling Orientation and Position of "Things"

Work on managing sensor information from location technologies resulted in a number of interesting location models. Leonhardt's [76] model, for example, combines various features that are geared towards making low-level location sensor information more accessible to applications. It consists of four layers: Sensors, Reception, Abstraction and Fusion. At each layer the information is transformed into a higher-level representation. A hierarchical structure is employed in order to deal with multiple rooms. Various models also employ the idea of spatial regions [50] and containment as powerful abstractions in location-aware programming and some attempts have been made to incorporate topological regions [77].

In general, one can say that the later work on location management for course-grained location technologies is concerned with abstraction and integration of low-level sensor information. Integration generally involves fusion of information from sensors that differ in resolution, accuracy, representation of values and mode of operation (synchronous or asynchronous). In particular, a factor that leads to difficulties is the fact that architectures for acquisition come in two flavours, as Leonhardt has remarked [76]: the infrastructural location tracking system such as the Active Badge and the self-positioning (sensors report locations relative to a beacon they sense) system such as Cricket [51]. Combining both approaches "makes things complicated" indeed. This, by the way, is exactly the problem we were facing when integrating the Active Bat system (infrastructural) with our stand-alone tracker based on ARToolkit (self-positioning). See Figure 4.5 for how this problem was solved.

As we look at applications that use the new fine-grained positioning/orientation technologies, we realise that most of them still use purpose-built software architectures and are used for specific types of applications only, such as OpenTracker [78] for Augmented Reality . One of the reasons may be that there is a much greater variety of applications that can be implemented using orientation and fine-grained position; all of which have different requirements such as: talking assistants [79] (what is X looking at), interaction devices/mice [80] (what is the current position and orientation of X) or Augmented Reality (where is X in relation to Y). So, predicting how to abstract low-level sensor information for application developers becomes a much more difficult task.

On the technology side, we are again dealing with a variety of sensor characteristics and acquisition architectures. In particular, we can distinguish between the above-mentioned self-positioning system such as CRICKET [51] and infrastructural tracking system, such as the Active Bat.

As we have seen, the infrastructural approach has the advantage of delivering an absolute point of reference, making it easy to relate different sensor sightings to each other. The self-positioning case is not as straightforward. Since sensors only "see" parts of the tracking system(s), some calculation that involves incorporating real-world knowledge about the position of the "beacons" needs to take place. Our method to do this was by using coordinate transformations. However, small movements of beacons can make the estimates unusable. Even in the case of infrastructural approaches we face difficulties when reconciling values delivered from sensors using different coordinate systems. The use of different frames of reference does not necessarily imply use of different sensing technologies. When a number of cameras are distributed across a room the values they deliver are always relative to their respective positions. Without an absolute frame of reference there are $n \times n$ possible coordinate transformations for n cameras. Again, slight movements of cameras would affect the delivered estimates. Of course, the use of different sensing technologies using different representations of sensor information would make this problem even greater.

4.4.2 A Physical Model for Interaction in a Sentient Environment

After presenting some of the issues involved in integration and abstraction of fine-grained position and orientation estimates a new approach shall now be presented.

This approach is based on the idea of "Sentient Computing" [16], which for our purposes has been defined as [80]: "Using sensors [...] to maintain a model of the world which is shared between users and application."

The idea behind this is that spatial relationships of objects and people become the "language" both the system and user understand. This had two implications on the proposed architecture. Firstly, it was decided to maintain one single physical model of, say, an entire room, a model that reflects the real world. Secondly, it is believed that one needs to abstract from locatables. With locatables we mean trackable objects such as Active Bats or visual markers. In most spatial management frameworks these locatables have an id that is seen synonymous to a real world object. The approach taken in this model is that locatables are only a *vehicle* to perceive the real world and not objects of interest themselves.

In practice this means that we want to be able to "use" points in our space, e.g. to attach temperature values to them, independently of whether locatables happen to exist at them or not. Therefore, the application needs an absolute frame of reference, independent of locatables, in order to make a reference to points in the room.

In addition to structural abstraction, the notion of semantic abstraction is proposed. As was stated above, in most location frameworks, locatables are assigned meanings by way of their hardwired object id mapping. However, the semantics of a point should be entirely independent from the location model. The temperature value example above showed that, in the general case, we want to associate a location point not just with a person or object but some arbitrary information. In order to achieve this separation, it was decided to describe, for our model, only the intrinsics of the tracking system without specifying what various parts represent. Let us assume a printer has a locatable with id l-id attached to it. Then the event "locatable l-id sighted at (x,y,z)" does not result in a model update equivalent to "printer p is at (x,y,z)" using the implicit assumption that l-id=p. This is stored separately, possibly even with the printer's measurements.

In order to achieve this association, we use the well-known concept of references. All possible points of interest have unique ids that can be used by clients in order to associate information with any real world point. Figure 4.6 shows a room with a number of interesting points an application developer might want to make use of in his program. Note that some points are absolute and some points move. But for the programmer this shall be transparent. After defining the points, he can use their names in his program and the location management framework will update the position of the points transparently as it receives events from the sensors.

Let us say that in this particular example some information shall be overlaid on the telephone's keypad, denoted by P. The telephone is tracked by an Active Bat B at one side. We know that there is a fixed relationship between the Active Bat and the telephone, T_{BP} (let us assume the Active Bat's limited capabilities to report orientation are enough for our purposes). By inspection, we can work out the crucial transform T_{EP} required for overlaying information on P. We know that a marker is in sight of the camera, which has a fixed relationship to a point R, which in turn has been surveyed with an Active Bat. So that can be used in order to determine the camera's transform in the Active Bat system. We, furthermore, know that there is a fixed relationship between camera coordinates and eye coordinates (obtained during calibration). Using these two relationships we can calculate the relationship between the Active Bat and the eyes, T_{EB}. After that we use the fixed relationship between the Active Bat and P in order to find T_{EP}. By using only relationships that are either returned by a tracker or always known, we have been able to work out the transformation needed.

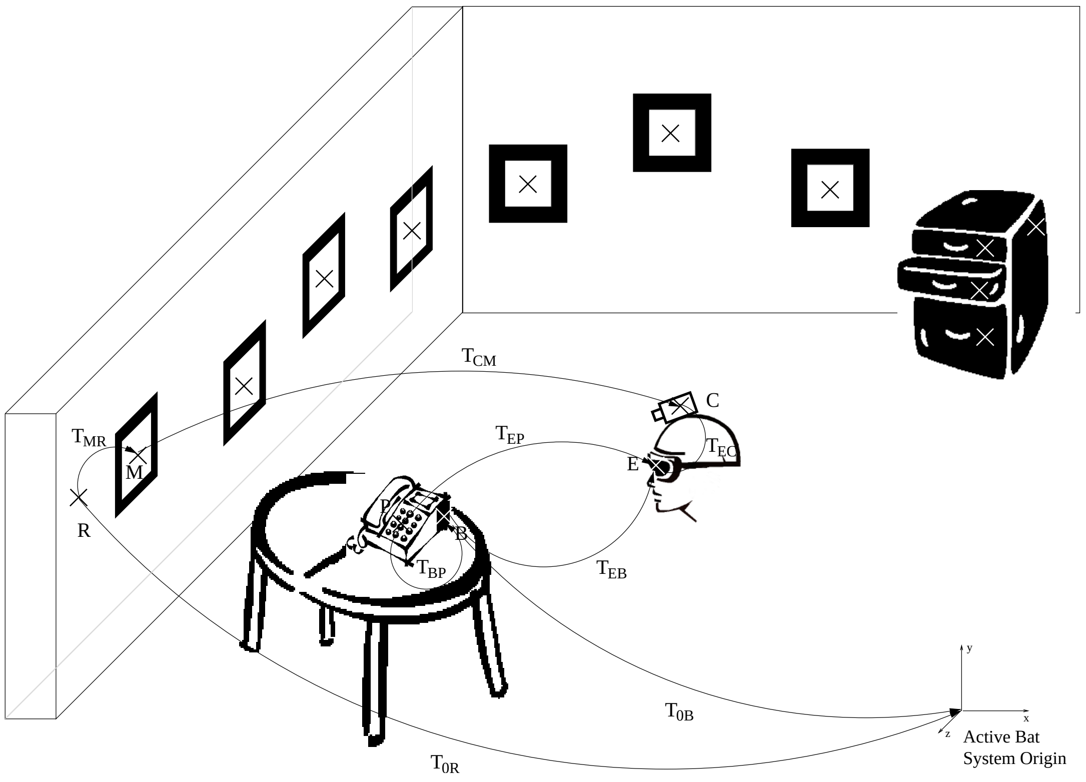


Figure 4.6: A room with a number of interesting points, marked with crosses

Ideally, what we want to do is, initially specify what points in the room we are interested in, what relationships they have to other points and then their transformations relative to a given frame of reference.

One could say that whereas most location models are concerned with attaching locatables to points, we are trying to attach information to points. In this way the model provides a good metaphor to "store" historical information, multimedia, interaction facilities etc. in the space. The physical model has the purpose of providing a coordinate system and its clients do not need to know about locatables.

For our AR purposes we could use the model to associate virtual objects with space. An additional "data model" could store virtual objects with references to points in the physical model described. The user's view would then be created by fusing information from the two. This method allows us to cater for the above-mentioned increase in possibilities developers have, using fine-grained position/orientation technologies, as opposed to location technologies such as the Active Badge.

Now that a great deal of the motivation was presented, we shall look at the physical model itself. The model consists of a network of points. Each node represents a point of interest. Points can be added to the network, provided they have some relationship to an existing point. This relationship, the edges in our network, can either be dynamic or static. If, when adding a point to the model, the edge to an existing point is specified as static, the model is signalled that the new point's position/orientation (strictly a point does not have an orientation, it just makes it easier to talk about points, rather than of origins in frames of reference) always has to remain fixed no matter how other points change. Figure 4.7 shows such a network. Each interesting point from Figure 4.6 maps to a node and each edge maps to an arrow between the nodes denoted by corresponding letters in that figure. Relationships that are fixed in Figure 4.6

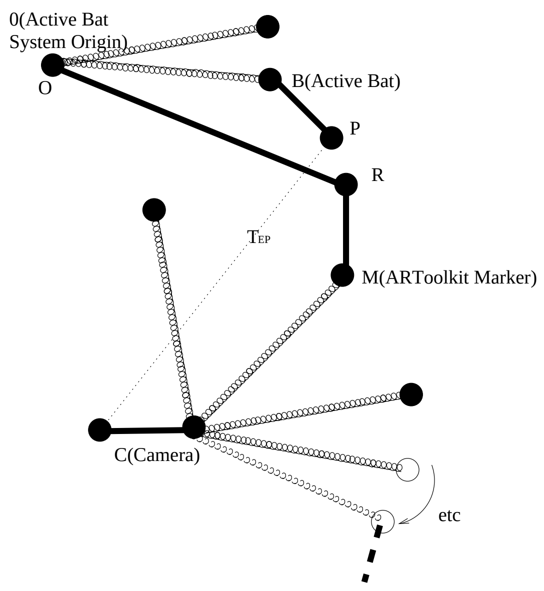


Figure 4.7: Conceptual view of the model we want to implement as a data structure. Dynamic links are shown as "springs", since they change constantly. This is the corresponding model to Figure 4.6. Only relevant parts are shown fully.

map to a solid edge, relationships that change map to "spring". The relationship of the camera to the many markers in the room changes as the user moves around, hence there is a whole set of "springs" coming out of node C, one of which represents our marker M.

For example, if you are specifying (how, will be shown in the next section) a tracking system that consists of visual markers deployed in the environment, you would specify transforms between the markers as static. Or, if you want to place a new point at a specific position/orientation from a point you have chosen as, say, a global origin you would specify a static relationship.

The fact that points can be added to a model in relationship to to others allows us to construct a dynamically updating physical model. Let us say our camera recognises a new visual marker B not in the model yet. It can, if it can determine its orientation/position in relation to a known marker A (if both are in view), add this to the model via a static relationship. The model stores this relationship and when it is asked to return the relationship between any two points, it can do so by walking through the network.

The specification of a dynamic relationship, on the other hand, lets the model know that the relationship is constantly updated through trackers. For example, the relationship between a camera and the visual marker B from above would be described as dynamic. If the camera were head-mounted and it recognised marker B, the model could return the user's head position in absolute terms for a client by walking through the network from A (see previous paragraph) to B, given that one client had chosen A as the global origin of its frame of reference. In this way one can specify more powerful constructs, e.g. a scene camera that can automatically compensate small movements by inferring its position/orientation from a visual marker that is always in view. As long as it can "see" any part of the whole system it will remain a useful member of the model.

Mathematically the relationships are modelled as homogenous transforms. As discussed in the previous chapter, these are 3×4 matrices that describe both a rotation and translation. The skeleton of the model is specified in terms of an XML file, in which dynamic relationships are described as relationships between sources and locatables. These are intrinsic to the underlying

]

tracking technologies. Since our model treats all points equally, this distinction is used to facilitate the specification only. In the case of the Active Bat system, the Bats can be seen as the locatables and the source is the origin the Bat system uses. In the case of a marker-based vision system, the locatables can be seen as the markers, whereas the camera can be seen as the source. Each source has a list of locatables it can recognise. This list is equivalent to the dynamic edges coming out of the node representing the source.

Previously, out of the two approaches described above (self-positioning and infrastructural) only the self-positioning one was seen as determining a relationship between points. After all, application developers using an infrastructural tracking system are not supposed to worry about frames of reference. It only becomes a problem when you are combining tracking systems. By modelling the origin of the infrastructural system, we can treat both approaches in a uniform way. The insight here is that any sensor value on its own is useless, unless one knows the reference point. By putting various reference points in a network, we can easily reconcile estimates by following paths through the network, multiplying transforms.

In this way we can use a self-positioning system that uses visual markers in the environment together with the infrastructural Active Bat system. However, we need to calibrate the two. Methods for calibrating tracking systems were discussed in great detail in Chapter 3. The resulting calibration transform would be mapped to a static relationship.

The model contains three types of nodes: sources, locatables and pivots. Pivots are general points of interest whose relative position/orientation clients want to have updated. Pivots can be oriented or simple, the latter being just a point without orientation. Say, you are using the the Model for an Augmented Reality application and want to overlay something on a file cabinet in your room. Your head is being tracked by an electromagnetic tracker. One of its sensors is placed on your head while the source is at a fixed position in the room. Now, you do not really need a sensor for the file cabinet. You can assume it will not move and define it as a Pivot with a static relationship to the source. Your AR application will automatically get the updated on the changing transform between your head and the Pivot as you move around.

Clients just see model points without knowledge about which of the points are actually locatables. In our Augmented Reality application the client could obtain a reference to an arbitrary point it can use as its global origin. It can then proceed to create the graphics scene, placing information at (x, y, z) coordinates with a specific orientation. Using references to the points any other information can be attached to them.

We mentioned the problem of reconciling values from n different sensors. The "universal language" in our model are the homogenous transformation matrices. Therefore, we require that the model receives values in the same units from the tracking modules.

4.4.3 Implementation

General Architecture

The framework is based on components. Every component extends a class that takes care of inter-component communication. The main components are Trackers and the Model. Client components can easily be added. One of the client components is a Renderer that takes care of creating a scene from the information the model provides. The inter-component communication takes place through event queues that were implemented. All trackers also implement the same interface in order to ensure uniformity from the model's point of view. With this architecture we can handle both synchronous and asynchronous sensors. The model could just update when a value becomes available. For this particular implementation, however, the approach taken was to slow down updates to the entire model in order to match the slowest tracker. This is to ensure that different update rates of trackers do not render the model incoherent at intermediate times.

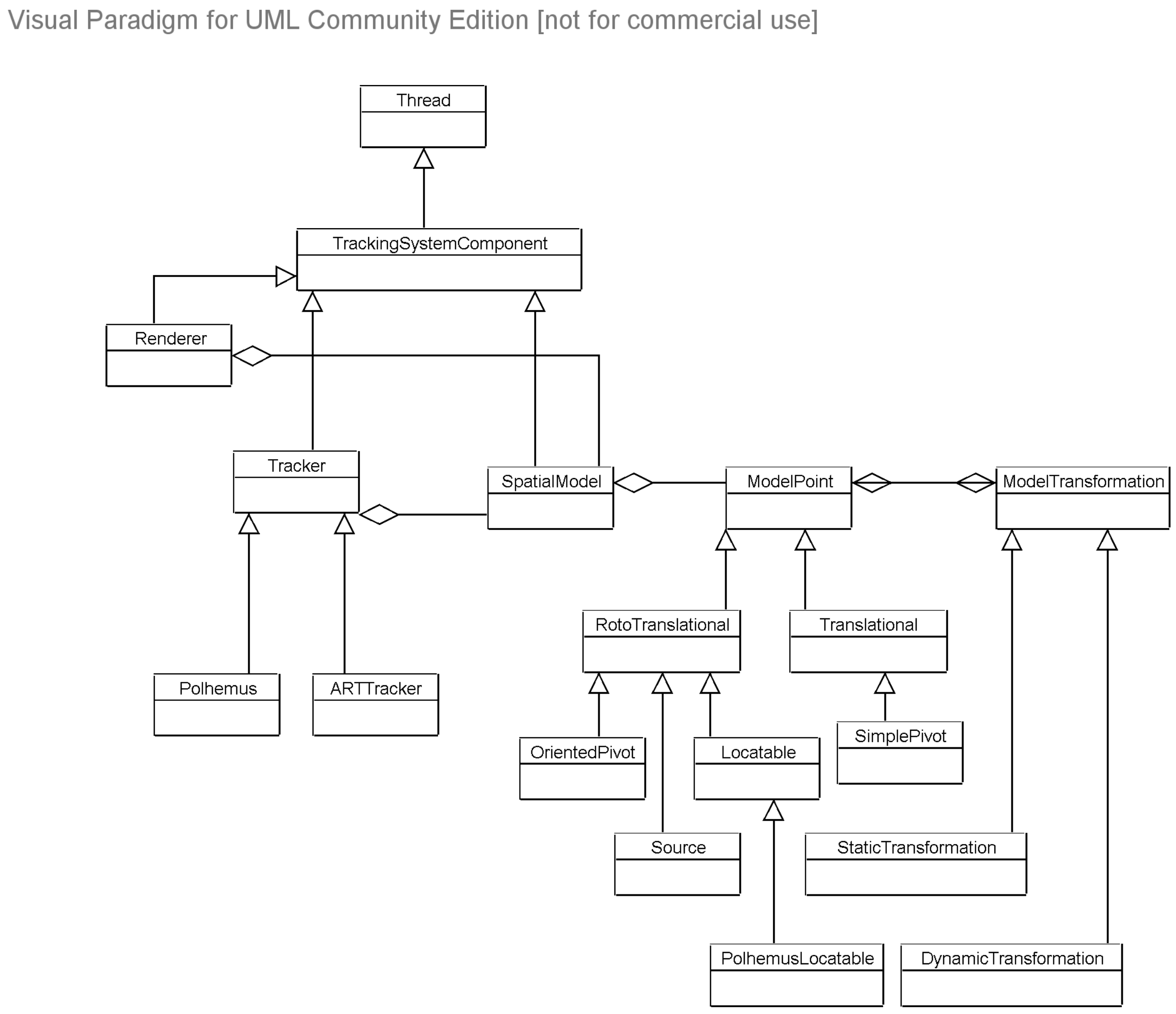


Figure 4.8: Class Diagram of the components Renderer, SpatialModel and Tracker. The SpatialModel consists of ModelPoints that are linked to each other with ModelTransformations in between. ModelPoints and ModelTranslations can be of different kinds.

In order to provide a unique referencing system for the points, the Tracker interface provides a mapping between ids recognised by the sources and ids recognised by the Model. These ids are stored in a hash table so that they can be looked up quickly. The hash table provides a reference to a point.

The data structure that holds the points consists of objects (see Figure 4.8) that are doubly linked so that the algorithm that finds the transform between two given points can walk through the network in both directions on its way from one point to the other. This is because trackers typically provide a directed transform for each locatable. In order to traverse an edge against this direction, the transform needs to be inverted. Two model points are linked by a model transformation. Each model point holds a reference to the model transformation and the model transformation holds references to both model points. Figure 4.8 also shows the different types of points and transformations we have discussed. The top left shows the component architecture.

Model Specification

In order to let the model abstract from underlying technologies developers need to specify the skeleton of the model. For this we use an XML document. Three different types of points can be seen: Sources, Locatables and Pivots. Pivots can be either simple or oriented, depending on whether orientation of a point matters to the "user" of the point. The first section (`<reference>`)is concerned with the intrinsics of the tracking system. It describes which source can pick up which locatable. Locatables and sources are given unique ids. relationships map to dynamic ones. The second, user defined section contains definitions of points of interest. These points map to nodes on static edges.

A code sample is given below:

```
<?xml version="1.0"?> <spatialModel
xmlns="http://www-lce.eng.cam.ac.uk/kr241/Spatial"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www-lce.eng.cam.ac.uk/kr241/Spatial
                   . -model-schema.xsd">
    <reference>
        <tracker trackerId="arToolkit1" xsi:type="artTracker">
            <source pointId="logitechCamera"
                xsi:type="artSourceType" parameterFileName="camera para.dat">
                <locatable pointId="hiro" size="80.0"
                    fileName="Data/patt.hiro" xsi:type="artLocatableType"/>
            </source>
        </tracker>
        <tracker trackerId="polhemus1" xsi:type="polhemusTracker" >
            <source pointId="polhemusSource1" xsi:type="polhemusSourceType">
                <locatable pointId="sensor1" xsi:type="polhemusLocatableType"/>
            </source>
        </tracker>
    </reference>
    <userDefined>
        <simplePivot pointId="aPosition" relatedTo="hiro">
            <position> 23.2 33.4 45.2</position>
        </simplePivot>
        <bridge from="logitechCamera" to="hiro">
            <transform>
                <row1>0.2 0.6 232.21 2.34</row1>
                <row2>32.43 121.43 23.54 43.3</row2>
                <row3>23.43 543.3 23.43 434.3</row3>
            </transform>
        </bridge>
    </userDefined>
</spatialModel>
```

When specifying a static relationship either a transform or function name has to be provided. The function name specifies a calibration function that is called in order to obtain the transform. Special cases of static transforms are bridges. These specify the transform between parts of two different tracking systems.

This specification is parsed and the model is automatically created for use by its clients.

The rules for this document have been specified in a separate XML Schema document. XML Schema was chosen as opposed to a XML DTD. This was because a schema is a lot more powerful. By using sub-classing it was possible to achieve an exact mapping between elements used in the schema and classes used by the model. Also, by using a Schema we could offload some constraint checking to the parser. Unique ids in specific ranges could be specified enabling unique references for each of the parts of the model.

A code sample is given below:

```
<complexType name="modelPointType" abstract="true">
        <attribute name="pointId" type="string"/>
    </complexType>

    <complexType name="sourceType" abstract="true">
        <complexContent>
            <extension base="sm:modelPointType">
            </extension>
        </complexContent>
    </complexType>

    <complexType name="artSourceType">
        <complexContent>
            <extension base="sm:sourceType">
                <sequence>
                    <element name="locatable" type="sm:artLocatableType"
                     maxOccurs="unbounded"/>
                </sequence>
                <attributeGroup ref="sm:artSAttExtension"/>
            </extension>
        </complexContent>
    </complexType>
```

Going through the definition of each type in our grammar is beyond the scope of this thesis and not required for the rest of the argument. The important thing to note is that it is possible to define a grammar with its own types and rules in order to describe our model fairly accurate.

4.5 Conclusion

We started off by recognising that our proposal to make sentient applications interactive implies a tetherless tracking system, since sentient computing intrinsically involves user mobility at least within a building. In order to argue our thesis it was necessary to show that a tetherless tracking system that can deliver position and orientation information within a building is feasible and viable.

By drawing on an existing Augmented Reality toolkit it was possible construct a sufficiently accurate stand-alone tracker by isolating the tracking code and repackaging it. The tracker was tweaked by identifying its main weakness and re-implementing part of it with higher quality algorithms (using OpenCV).

Deploying the tracking system in a part of a room showed that it is in principle possible to cover a whole room with an accuracy acceptable for a toy Augmented Reality application, especially in view of the fact that it has been designed to be extensible in every aspect and most tasks have been automated in order to decrease the marginal effort to extend it. It has its limits when it comes to delivering reliable overlays for more demanding applications (e.g. applications that involve interaction with virtual overlays) and is more difficult to use than a tracker that just delivers sets of the required six values as raw data.

Nevertheless, the "stand-alone" tracker did in fact need some infrastructural base in the form of a storage of marker positions. This together with the realisation that

- more topological information (i.e. their location) needs to be stored along with the markers,
- the tracker needs to be integrated more in the existing sentient environment (so that location events from the environment and tracker values can be related more easily) and
- each application setup requires bespoke mathematical calculations to be performed on the tracker output

led to the design of a tracking system architecture that solves all three problems elegantly by abstraction.

The architecture presented is mainly a proof-of concept. There are many issues the simple implementation provided does not deal with, such as conflicting sensor readings, different temporal resolutions, quality of sensor readings or how inaccuracies accumulate as you walk the graph.

The main focus of the work on the architecture was a conceptual one. In this sense the main novelties presented are to be found on the conceptual side, namely

- the idea to abstract from locatables
- the idea to represent origins of infrastructural systems explicitly and in this way combine the two types of approaches to tracking by using matrices
- the idea to implement a tracking system on the basis of a description of its intrinsics (what sensor is recognised by what source) using a bespoke grammar defined in XML

A system that uses some of the concepts introduced here is currently being built by the Interactive Media Systems Group at the Technical University in Vienna (personal communication). We shall now move up another layer and look at how we would implement interactive sentient applications.

Chapter 5

Building Interactive Location-Aware Applications

We shall start with a recap of what is being proposed and why. The thesis is that off-desktop location-aware applications running in a networked environment can be made interactive in order to render them more understandable.

Let's have a look at a typical networked environment. The main elements of such an environment shall be people, devices, computationally enhanced everyday objects (*active objects*), services, applications and regions (see Figure 5.1). Services are usually remotely accessible programs that receive requests and perform a specific task for users in the environment. In our architecture location-aware applications use regions in order to evaluate statements such as "Is X standing in front of the computer".

Now, the element that makes such an environment *reactive* is *events*. Typically, these are detected by the computer through sensors. The problem users have in such an environment is that even though these events are perceivable by both the user and the computer, each may regard a different course of action as appropriate. This can be due to either different interpretations of the event or different "initial conditions".

The solution proposed is to extend this model in order to make the environment *interactive*. Each of the elements mentioned shall have a way to provide feedback so that the user is notified about automatic actions by the computer and, perhaps, can even exert some control over them.

In this chapter we will use our knowledge of Augmented Reality in order to see how we can make this environment interactive, giving each and every object the facility to display its state in situ.

5.1 System and Applications in a Ubicomp Environment

One of the elements mentioned needs to be elaborated on: the application. The question is whether an interactive environment needs to have the concept of applications. In traditional desktop computing most definitions will in one way or another define "application" as complementary to the operating system. In fact the separation of operating system tasks and application tasks has been the basis for the flexibility and effectiveness of traditional desktop computing.

Porting this idea to a Ubicomp environment may prove difficult because a uniform operating system does not exist due to its dynamic nature. Nevertheless, the attempt to separate "system" responsibilities from applications seems promising even in this field.

The system used for this thesis is called the SPIRIT system [50]. The SPIRIT system is a CORBA-based [81] system that models the real world as a database. The database is updated

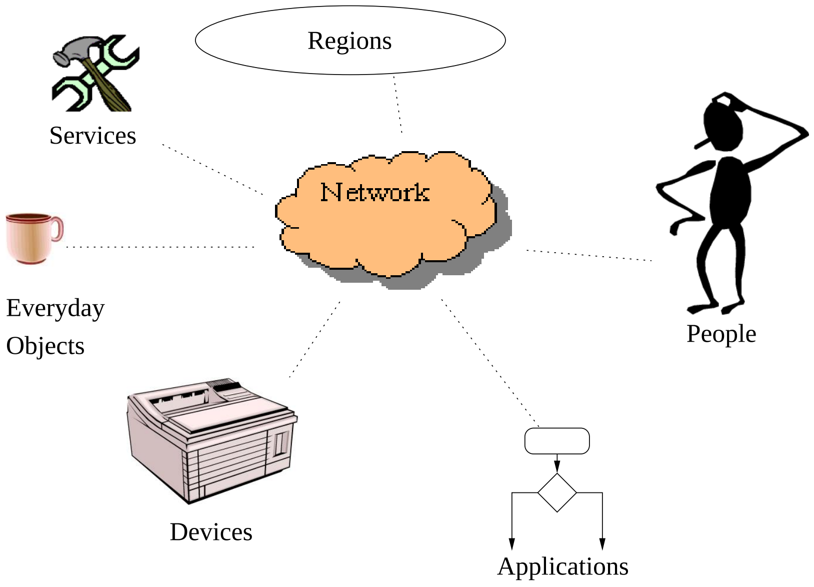


Figure 5.1: A networked environment

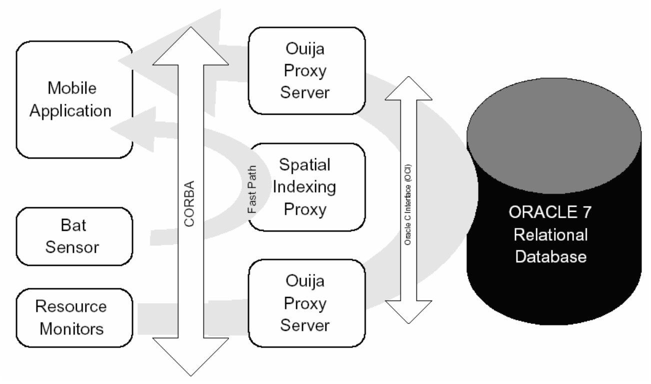


Figure 5.2: The SPIRIT architecture

as the state, especially location, of real world objects and devices is perceived by the computer. Every real world object has a counterpart in the system. These counterparts are CORBA objects called proxy objects. Each proxy object has a CORBA interface accessible from any computer on the network. In this way commands can be sent and information retrieved from the actual devices or other everyday objects. See Figure 5.2. The proxy objects are to be found in the middle part of the figure. They interface with the database on one side and clients on the other. The Spatial Indexing Proxy is responsible for maintaining the spatial model of the environment.

Developers now have the ability to write *applications* that can make use of any recorded device or everyday object much in the same way as traditional application developers make use of “objects” (in the computer science sense). The merits of separating applications from system now become clear. It is much easier for developers to develop applications, concentrating on a well-defined task and abstracting from low-level networking and sensor polling.

Given that we have decided to extend this existing application model we are now facing the problem of how to integrate an Augmented Reality application with a Ubicomp application.

5.2 Augmented Reality Application Models

We shall start by looking at the application model of an Augmented Reality application. A typical Augmented Reality application has the following structure:

```
normal main loop{
    get tracker readings
    for each virtual object
        calculate object position and orientation
        render
    switch to calibration main loop if mouse input=="calibrate"
}

calibration main loop{
    render crosshair
    check mouse input
    if mouse input=="terminate",
        switch to normal main loop
    if mouse input=="sample",
        get tracker readings
        save tracker readings
        shift crosshair
        if last sample,
            calculate and switch to normal main loop
}
```

This is in fact a simplified view of an ARToolkit [53] application. There are two modes: One for optical see-through calibration and one for normal operation. In normal operation the tracker values need to be read continuously. Every tracker value will have an id attached to it. This id will be used to look up the virtual object for that particular sensor, if there is a virtual object attached to a sensor at all. The value will then be used during scene construction as the offset of the virtual object from the origin. This needs to be done for all virtual objects. The user can switch into calibration mode at any time in order to (re-)perform the optical see-through calibration.

In calibration mode a crosshair will be presented to the user and she will use mouse input in order to tell the software when to sample a tracker value. Remember, for calibration it is required that a screen coordinate is sampled together with the corresponding 3D point. The user can terminate the calibration procedure by an appropriate mouse input at any time. If she has aligned the target with the screen coordinate given by the crosshair she will press a mouse button and sample the point. This will shift the crosshair for the next sample or if all points have been sampled, the application will return to normal operation.

There are other application models to be found in Augmented Reality. Studierstube [82], for example, is entirely based on an Open Inventor [67] scene graph. In fact the whole application is distributed over a number of nodes in the scene graph. This is possible because Open Inventor nodes can control the flow of program execution. They can be more than just descriptions of graphical objects. The problem with this approach is that application logic is lumped together with presentation.

Another application model is used by DWARF [83]. In this component-based model the application is placed in a separate module. The application can then access different service modules in order to perform functions. Especially, when it comes to tracking this makes a lot of sense. Tracking, even though it is required by all applications, is not the main purpose of any application. Hence, a general-purpose interface suitable for all applications can be provided. To build an application from ready-made components has its advantages when it comes to prototyping a set of predictable applications, but in terms of flexibility such a system would be a step back if applied to a feature-rich Ubicomp environment. As a comparison, "programming with space" [16] takes the approach of integrating the notion of space into conventional programs. Integrating Ubicomp features as data structures in conventional programs allows for more flexibility at the cost of less abstraction.

Nevertheless, the idea of reuse can still be appreciated. Providing a framework for application developers is vital in order to prove that it is possible to build not just one but any application in the proposed manner. The difficulty is in finding the right balance between flexibility and reusability. We do not want to constrain application developers in building their desired applications in any way, but at the same time we want to rid them from tedious tasks that need to be performed in a standard way. In separating the standard from the creative we can provide an application base that can be used as a skeleton for all applications.

More specifically, we are looking to design a framework that

1. allows developers to augment location-aware applications visually,
2. separates standard (common to all) application parts from the core application logic,
3. is not limited in what kind of visualisations it can present and
4. provides a strong link between the core application logic and visualisation.

The first requirement stems directly from the thesis. The second requirement follows from the discussion above. The third and fourth requirements follow from the idea that we want to be able to visualise the state of anything, anywhere at any time during the execution of a location-aware application.

5.3 Integrating AR and Ubicomp

Our task is complicated by the fact that we have to integrate a Ubicomp backend. The code below shows a simple spatial application in pseudo code. As soon as the user's Active Bat enters a zone, e.g. a region in front of a computer, she is logged in automatically.

```
Class AutomaticLogin::EventConsumer{
    AutomaticLogin(){
        register with SPIRIT for Active Bat events forwarding zone and user
    }
    onZoneEntered(user id){//called by SPIRIT
        LoginServiceInterface lsi;
        getLoginServiceInterface(lsi) //dummy variable lsi
        lsi.login(user id)
    }

    onZoneLeft(user id){//called by SPIRIT
        if user logged in
            LoginServiceInterface lsi;
            getLoginServiceInterface(lsi) //dummy variable lsi
            lsi.logout(user id)
    }


main(){
    create AutomaticLogin
    register with SPIRIT
    start AutomaticLogin
}
```

The actual code for such an application would be 3 to 4 pages long and various checks and registrations need to be performed. *AutomaticLogin* consumes SPIRIT events and is a *thread*. SPIRIT uses a spatial indexing algorithm [50] in order to determine when to notify its event clients about events involving regions. The SPIRIT database has relevant regions saved, such as a region around a phone, around a computer, a room, around a Bat etc. The system updates its database as trackable objects move in space. During evaluation it will determine overlaps of regions, such as an overlap of an Active Bat with a room, meaning the person has entered the room. Whenever such an overlap is determined, event clients that have registered for these events are notified. In this application *AutomaticLogin* has registered for two events: entering the computer zone and leaving it.

Now, let's say the task is to convert this reactive application into an interactive application, i.e. we want to give the user the ability to "see" that something has or, as may be the case, unexpectedly has not happened when her Active Bat enters the region in front of the computer. It is also useful to "see" the region the computer understands as "in front of the computer".

For this purpose we will try to visualise the region itself and the position the computer senses the Active Bat to be. As soon as the computer has detected the Active Bat in the region the visualisation of at least one of the virtual overlays (Active Bat or region) needs to change as feedback. They need to be changed back as soon as the Active Bat leaves the region.

The resulting code will look as following:

```
normal main loop{
    get tracker readings
    for each active object
        /*object id needed now in order to synchronise view and
        state*/
        use object id to retrieve object position from global variables
        use object id to retrieve state from global variable
        /*using global variables for inter-thread communication*/
        use object id to find virtual representation
        render
    switch to calibration main loop if mouse input=="calibrate"
}


Class ARAutomaticLogin::EventConsumer{
    AutomaticLogin(){
        register with SPIRIT for Active Bat events for a zone
        /*need to monitor all Active Bat movements now*/
        register for Active Bat movement events
    }
    onBatMoved{user id) //called by SPIRIT
        /*every movement needs to be recorded now for visualising the
        overlay*/

        BatInterface bi
        getBatInterface(bi) //dummy variable bi
        save bi.getBatPosition(user id) globally
    }

    onZoneEntered(user id){//called by SPIRIT
        /*need to save state for later visualisation*/
        change global state variable
        LoginInterface lsi
        getLoginServiceInterface(lsi)//dummy variable lsi
        lsi.login(user id)
    }

    onZoneLeft(user id){
        /*need to save state for later visualisation*/
        change global state variable
        LoginServiceInterface lsi
        if user logged in
            getLoginServiceInterface(lsi)//dummy variable lsi
            lsi.logout(user id)
    }


main(){
    /*Ubicomp part*/
    create ARAutomaticLogin
    register with SPIRIT
    start ARAutomaticLogin

    /*AR part*/
    run normal main loop
}
```

The calibration main loop is the same as before. This, again, is a very simplified version of a real application. The actual application would span many pages and therefore has been abstracted here. We see that we have two threads running at the same time: *ARAutomaticLogin* and the main loop. *ARAutomaticLogin* now needs to record every Active Bat movement and position. This will be used by the rendering part in order to shift the overlay accordingly. We have, furthermore, introduced a state variable in order to let the rendering code know how to render the region and Bat overlay. If the system senses the Active Bat as inside the computer zone, let's say it will render both in a different colour. The states need to be changed as soon as the Bat leaves the zone.

It is, of course, possible to give more feedback and give the user more interaction facilities and control, but for simplicity we will not extend the user-friendliness of the application.

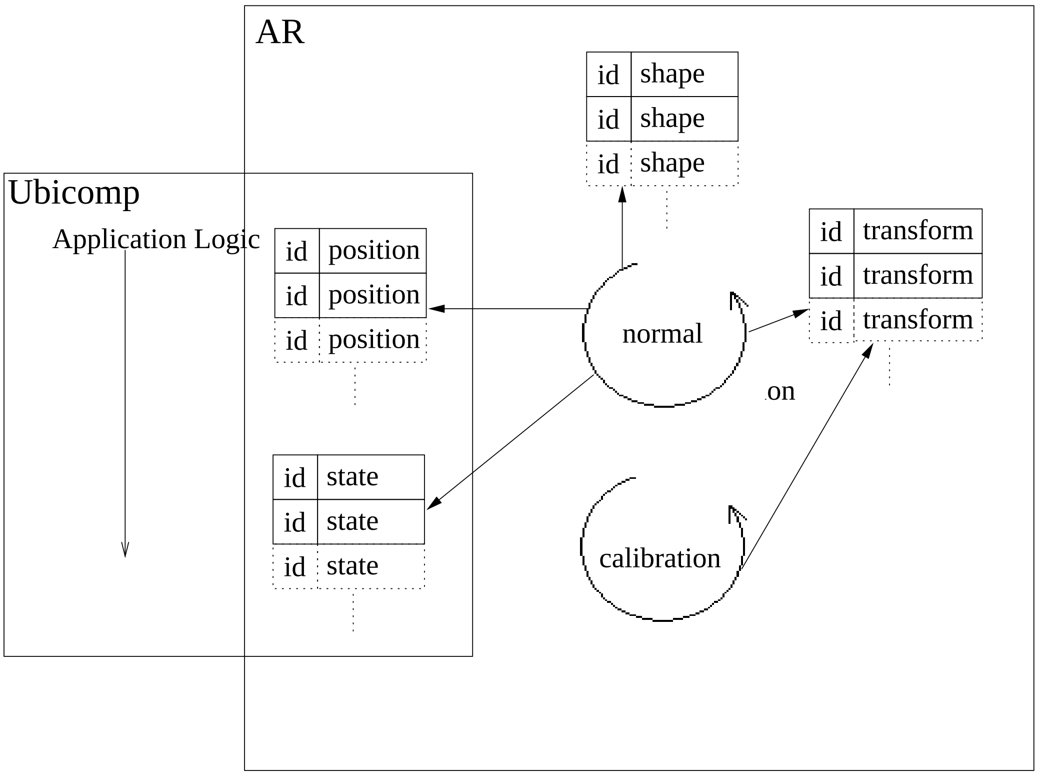


Figure 5.3: Schematic representation of data flow during rendering

By looking at the code we observe that the two parts appear incoherent. Partly, because the AR part is procedural in its nature and the Ubicomp part is object-oriented. [1] In fact, we are actually running two applications in two threads that are *sharing* information rather clumsily by global variables and id numbers. This situation is presented schematically in Figure 5.3 with special emphasis on data flow during rendering.

The two blocks represent the Ubicomp part and AR respectively. The Ubicomp thread shares variables for position (needed for the overlay) and state of each active object with the AR part.

The AR part consists of two loops, depending on the state of the AR application: calibration mode or normal mode. In calibration mode, the main loop needs to read in transforms from the tracker using sensor ids. In normal mode the main loop reads in these as well, but needs to perform calculations on them in order to render virtual objects associated with these transforms with the user's eyes as a reference point. In order to find the visual representations of objects such as the zone or the Active Bat, the process needs to look up the ids received either from the Ubicomp part via the Active Bat system or from the tracker sensors. This is, of course, assuming the physical objects we want to overlay on, are being tracked by either.

The architecture presented above is clearly sub-optimal for large applications. Too many global variables are involved. There are too many parts all over the code developers need to adapt in order to fit their application into this architecture. In fact, each arrow shown in Figure 5.3 implies a dependency of the code.

In the ideal case a developer would just provide the Ubicomp part of the application and the connection to the AR part would be virtually automatic. In other words a developer should not need to worry about the presentation process when designing the application logic.

[1]This arises from our particular implementation, but is perhaps supported through the fact that AR requires developers to perform a standard set of tasks for each frame and Ubicomp applications are mostly idle.

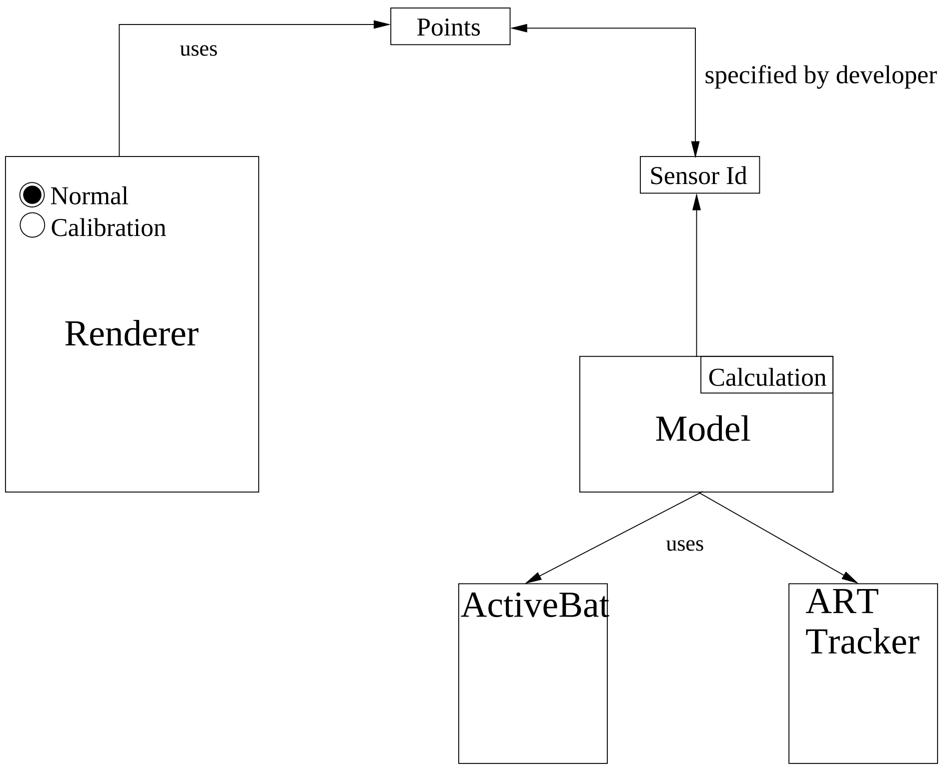


Figure 5.4: Integrating the tracking system (not implemented)

We shall try to approximate this goal by refactoring and employing established architectural design patterns. The resulting architecture shall be more flexible, reusable and enforce systematic development.

5.4 First Steps Towards a Reusable Application Base

The first step was to replace the OpenGL rendering loops with a scene graph-based API (Open Inventor, [67]). We shall concentrate on a component we shall call “Renderer”. In our case the Renderer can have two modes: Calibration or Normal. In a scene graph-based toolkit this can be achieved very easily. Two different scene graphs, that each both encapsulate input event handling as well as a rendering loop, are used. When a mode change occurs they are simply switched and everything else (rendering, input event polling) takes place in the background anyway.

The second important improvement would be to integrate the tracking system described in the previous chapter. Figure 5.3 shows that developers need to read in tracker values (from both the Active Bat system and head tracker) in both possible main loops. The readings need to be mapped to some meaningful points via id lookup tables. These points have virtual objects, that need to be looked up by id, associated with them. Some matrix calculation needs to be performed in the normal main loop in order to set correct rendering transforms relative to the user’s eyes.

This, however, is exactly what the tracking system described earlier provides. Figure 5.4 shows how this integration can be performed.

The reference point calculation is performed in the Model. In our hypothetical application, we have two types of trackers updating the Model. The concept on which Renderer and Model "synchronise" are *Model Points.* These are named by the developer in the shape of an XML file. Trackable (Active Bats or Sensors) ids are assigned to Model Points so that all components can uniquely map a Trackable to its virtual object. The Renderer has two modes, one scene graph for each.

We have now shown how to separate code that is common to all applications and package it into a number of components that communicate with each other. The question now is, how integrate the part that is different to all applications, i.e. the left side of Figure 5.3, with our application base.

5.5 Model-View Controller Architectures for Visual Interaction in Context-Aware Environments

5.5.1 Overview

Reviewing the requirements mentioned at the end of Section 5.2, we see that the framework put forward so far satisfies the first three requirements. The requirement that the application logic needs to be linked to the visualisation could have been extracted from the hypothetical login application. Global variables are used in order to connect the state of an object with its visualisation. The main reason for that was that we were dealing with an application that exhibits some kind of *interactivity.* The user moving her Active Bat in and out of a region and receiving feedback represents interactivity. The question now is how to integrate the application with the application base in a way that supports interactivity.

The following sections will deal with the core architecture of the system implemented. The architectural problem to solve is to integrate an Augmented Reality interface with a Ubicomp backend, taking care of event flows from trackers, sensors and interaction devices.

The architecture presented makes contributions in addressing following issues associated with visual interaction in context-aware environments:

- **Separation of concerns**. Applications running in context-aware environments typically need to read in values from a variety of sensor sources, compare these against conditions (which in turn depend on context, profiles etc.) and affect changes accordingly. Add to this the data flows associated with visual augmentation as described above, and it becomes clear that a breakdown of the "work load" by identifying appropriate components and specifying communication protocols between them is a desirable architectural feature.

- **Flexibility**. Context-aware applications need to cater for a surplus of flexibility simply due to the fact that what is part of *context* in a particular setting, cannot be determined a priori. Active objects can appear and disappear, events can suddenly gain importance for certain active objects. The approach this architecture takes in order to address this issue is to make use of indirect event communication, i.e. senders do not need to know about receivers or the relevance of the message they are sending.

- **Interactivity**. Ultimately, our aim is to show the user the state of the Ubicomp environment at any instant in order for her to act upon it. For desktop GUIs a number of interaction architectures exist that allow applications to respond visibly to any user action affecting the common interaction state. This architecture introduces the Model-View-Controller (MVC) paradigm to Ubicomp in order to provide meaningful feedback about users' actions at all times.

- **Modelling context-aware interaction in architecture**. Good architecture exhibits a close relationship of form and function. While there are many ways for applications to interpret a user's actions with respect to her context, a solution that models this in architecture has a number of benefits. Design patterns [84] such as the extended MVC pattern introduced here, generally allow programmers to reduce the complexity of algorithms by making use of a particular architectural structure.

5.5.2 The Classical Model-View-Controller Design Pattern

The most successful software architecture paradigm for building interactive systems is the Model-View-Controller (MVC) paradigm [20].

The original MVC paradigm models the domain as a collection of objects, as usual in object-oriented design. Each object is then partitioned in three parts, a Model, a View and a Controller. The Model contains all intrinsics of the domain object. The View's responsibility is to render the domain object. A Model can have many Views, e.g. you could present the same data as a graph or a table. In the original MVC paradigm each View has one Controller that interprets input device movements taking place in the View (window) and performs an corresponding change to the Model.

One of the advantages of this paradigm is that intrinsic domain object properties are kept separate from the intricacies of their presentation, which are device-dependent. The fact that many Views can be hooked up to one Model makes this design flexible and extensible, especially considering the fact that the Model does not need to refer to any intrinsics of View. In fact, a View only needs to implement an interface with one function called *update()*. When the Model changes it calls this function on all dependent Views, not "knowing" how exactly the change will affect a particular View.

Note that in the MVC paradigm the entire visualisation is built from Views of Models in the application. Let us examine the benefits of this approach in an example. In order to make it more interesting, the example is from our application domain, rather than from a desktop GUI environment. A virtual menu appears next to the Active Bat as an overlay and is controlled by the Bat buttons.

Figure 5.5 shows the corresponding diagram. The Active Bat has a Model, View and Controller. Its View needs to change whenever an input event (button press) occurs. The menu is supposed to be part of the Active Bat. Hence, its View appears as a sub-View of the Active Bat's View. The other sub-View is the overlay of the Active Bat ("Bat Body"), i.e. mainly the labels on the buttons. When a button is pressed we want both Views to change simultaneously. The event is received by the Bat Controller and forwarded down the hierarchy to the Controllers of both sub-Views.

In a window-based GUI environment this hierarchical structure is used in order to handle multiple, possibly overlapping windows. The problem there is to find out which window should react to the user's action. Input device movements are dispatched to the entire hierarchy and each Controller decides whether to act upon it (i.e. change the Model it is responsible for) or pass it on to the Controllers of its View's sub-Views. This results in an automatic organisation of the control flow.

We notice that the View hierarchy fulfills two functions. Firstly, it ensures easy composability by allowing Views to be attached to each other. The second, more subtle function, is to make sure that an interaction event received at the top of the hierarchy filters down to the right Controller. Remember that if a Controller decides that the event is not in its scope it will not forward it to its View's children.

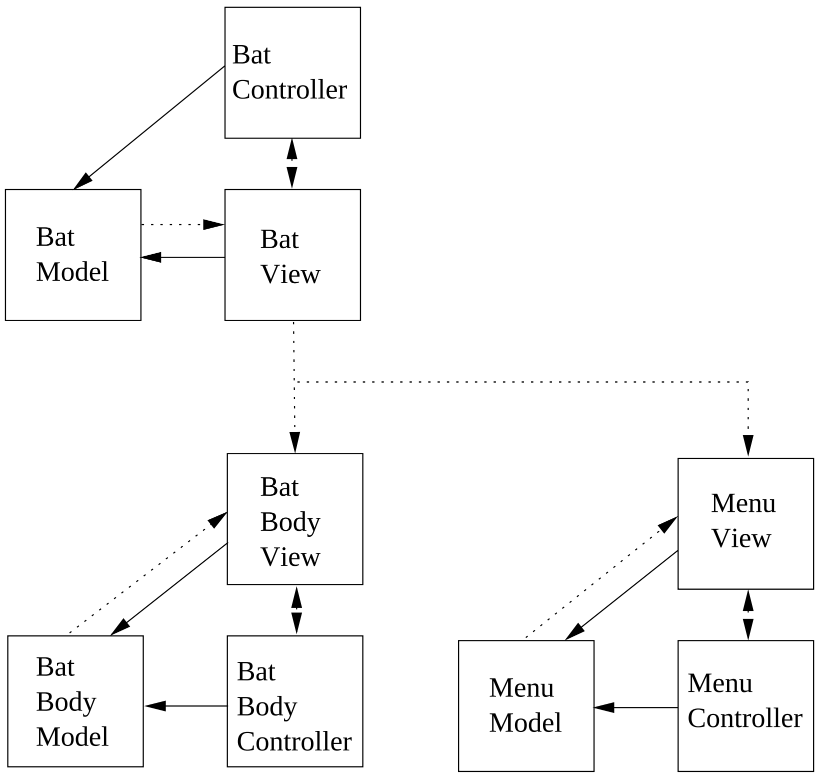


Figure 5.5: A diagram for a virtual Bat menu. Arrows show references. Base references are shown as dotted arrows.

5.5.3 The Extended Model-View-Controller Design Pattern

Let us imagine we applied the classical MVC approach without modification to our applications. Sentient events would be sent from the backend to the top-level Controller which would forward them down the hierarchy. The problem that would arise is that Models would need to be context-aware. Assuming that each active object adapts to the current user context, each Model would need to store the user context as a state variable that modifies the behaviour exposed to the rest of the world. The result is that all Models would be "contaminated" with context code.

Furthermore, the evaluation of whether an event signifies a change in the user's context would need to be performed in each Controller for each event. It would be better if one single entity could notify *all* Controllers about when a user has entered or left a context, performing the evaluation only once, but messaging all significant Controllers.

One thing we should keep in mind is that in GUIs, Views set the *context* of an event. An interaction event such as a mouse click can only be interpreted with knowledge of the View it was performed in. That is why some MVC implementations lump Views and Controllers together. In Ubicomp, however, the interpretation of an event has got nothing to do with the Views in our application. So let us try to separate the two classical functions of Views, i.e. presenting the Model and *setting the context for the Controllers.*

The idea is to introduce a new set of components whose responsibility is to set the context in which Controllers receive a particular interaction event. Analogous to classical Views these components need to receive events and ensure that only Controllers within their context are forwarded these events. The number of components of this type therefore needs to be equivalent to the number of contexts the application recognises. We shall call these components "Contexts".

Figure 5.6 shows the architecture of an extended MVC design pattern. At the top we see a chain of the Context components. The application will use one of these components for each *user* context. For location-aware computing we will assume a context to represent a *region*, i.e. the user's context is the (predefined) region she is currently in [2].

Context components encapsulate properties of the context they represent. The first Context component will receive all events from the backend. If a *context event* is received, the Context components will evaluate it one after the other: If it evaluates to false for a particular Context it means that the user has not moved into the context represented by this component, i.e. this is not the current user context. The event will be forwarded to the next Context. If it evaluates to true, the corresponding Context component will become *active*, activate its Controllers and initiate an action in order to deactivate all other Contexts (by calling a function *deactivateAll()*).

As an aside, the design pattern used to work out which Context is active is called "Chain of Responsibility" [84]. It is normally used when various components need to work together in order to determine which one is responsible for handling a particular event. One question is why to use a chain of responsibility in our case, rather than to broadcast the events to all Contexts. The reason is that we can make use of the exclusive nature of user contexts, i.e. the user can only be in one context [3]. The set of contexts the evaluation function of each Context components operates on, only includes contexts that appear in and after it in the chain of responsibility. Each Context can ignore all Contexts that appear before it in the chain, making the evaluation more and more efficient as we move down the chain. Broadcasting would imply that all components perform an evaluation without knowledge of results from other components.

In any case, all future *interaction events* coming into the chain will now be forwarded through the chain until they reach the active Context, which then forwards it down a level to its Con-

[2]Figure 6.3 in Chapter 6 will present an example for Contexts being non-region-related, but our focus in this thesis is on location-aware applications.

[3]Theoretically, users could be in multiple hierarchically organised contexts ("in the office" and "in the meeting room") but this was not explored further and not applicable to our existing location-aware applications.

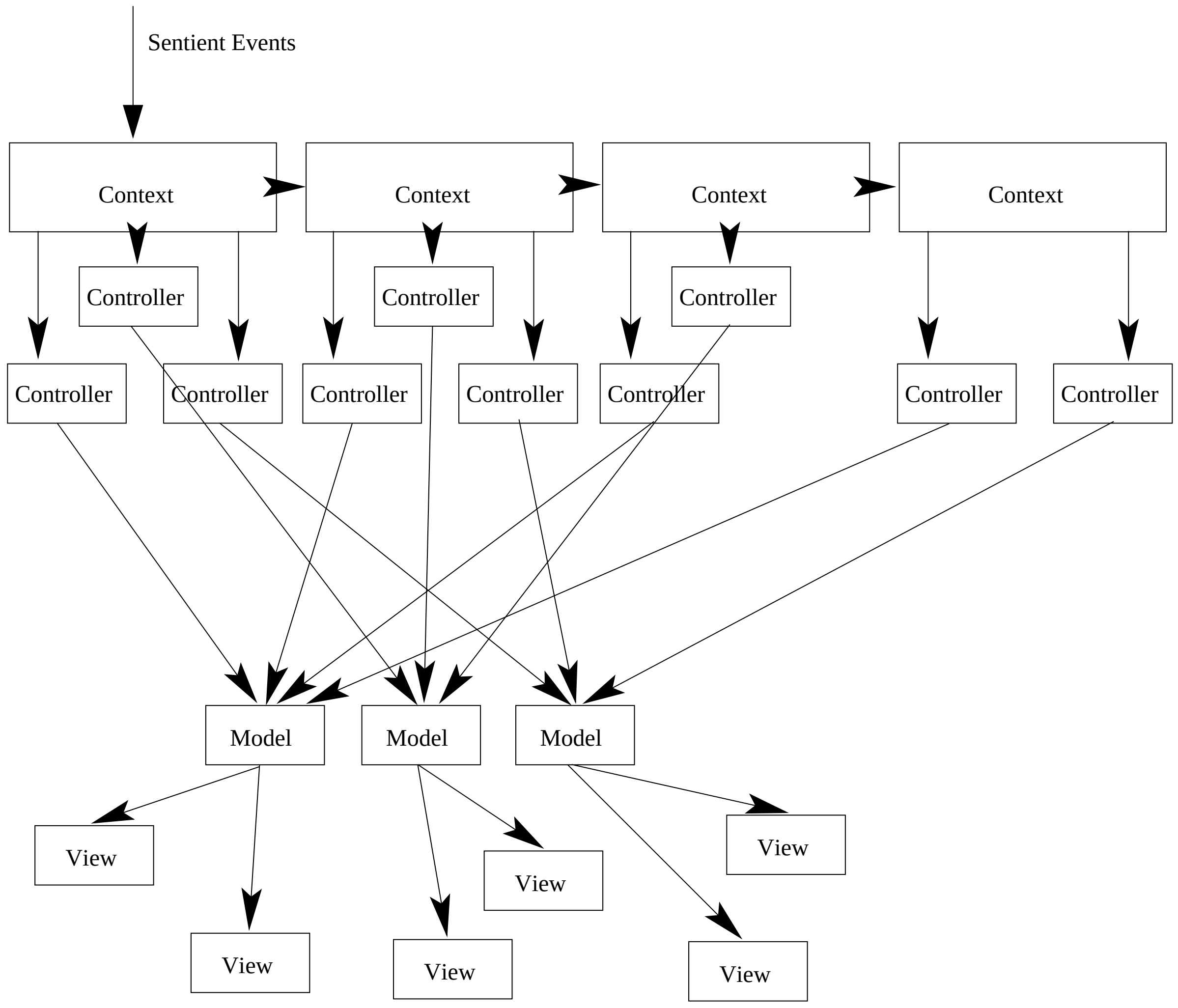


Figure 5.6: The extended MVC architecture. Arrows indicate event flows.

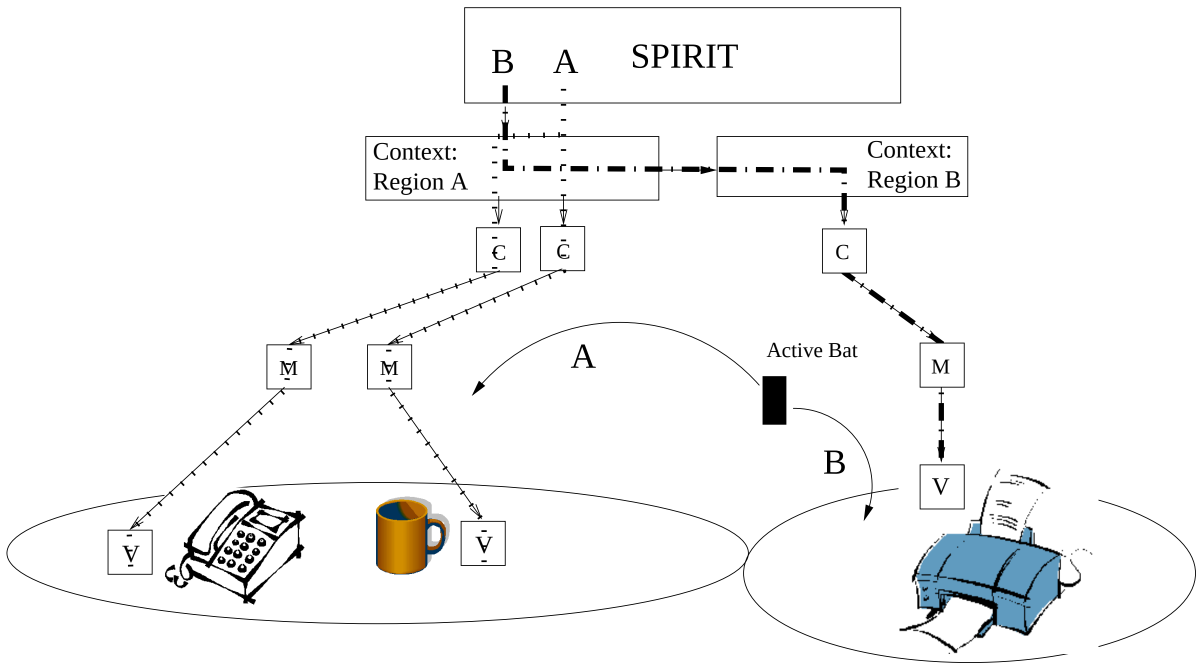


Figure 5.7: Control flow changes according to Active Bat location

trollers, i.e. only Controllers attached to the active Context get to receive and *interpret* interaction events, as all deactivated Contexts *ignore* and forward *interaction events*. A context can be activated again if it receives a *context event* that makes its *evaluate()* function evaluate to true. The result: context-aware interaction modelled in architecture; interaction is always interpreted according to its context.

It is important to know that we distinguish between *context events*, i.e. events that imply a potential change of user context, and *interaction events*, which are events that signify an interaction, such as a button press or, say, a particular gesture. The backend needs to tag events accordingly as potentially context-changing or interaction-related. This distinction maps to what others have called implicit and explicit input in Ubicomp [85]. However, our model does not see context events as "input" and in fact treats them very differently to input (interaction) events. Context and interaction are modelled orthogonally: Each interaction can be performed in each context, but is interpreted differently accordingly.

In terms of location-awareness, once a *region* has been activated, all events of potential interest to active objects will be forwarded to active objects located in that particular region only, i.e. only active objects *near* the user will start reacting to user actions via the Controllers and their visualisations will change each in their own way.

Figure 5.7 shows how the control flow takes different paths through the software architecture depending on whether the Active Bat enters Region A or Region B. Region A has two active objects in it and hence two Controllers are affected. All Controllers affect Models which in turn affect the Views. This results is an update of the visualisations the user sees on her HMD, giving her an interactive experience.

Figure 5.6 implies we write one Controller per Context per Model. This gets rid of context evaluation code in the Controller and enables a clearer Controller design (at the cost of needing to write multiple Controllers per Model). If an active object never reacts when the user is in a particular context no Controller is needed for that Model in that Context.

Let us now look under the hood of the components involved in this design pattern. We

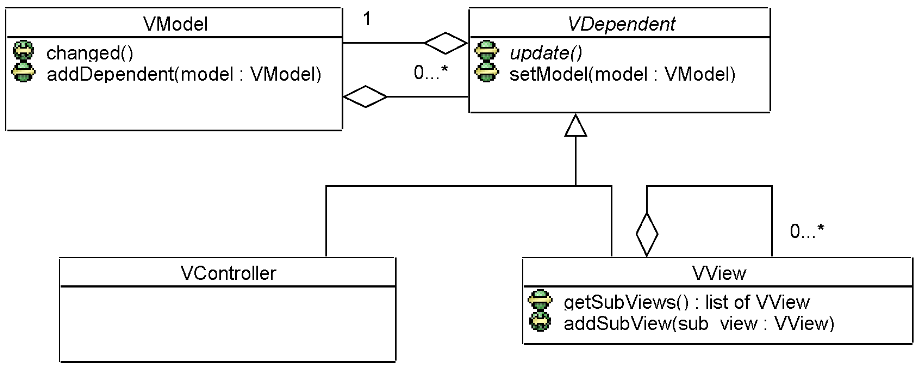

VModel
changed()
addDependent(model : VModel)
1
0...*
VDependent
update()
setModel(model : VModel)
0...*
VController
VView
getSubViews() : list of VView
addSubView(sub view : VView)

Figure 5.8: The basic MVC architecture

will start with the basic MVC pattern. This is shown in Figure 5.8 as a class diagram. Our class names are prefixed with a 'V'. It shows that the Model can inform Dependents about updates that are performed on it by calling its own *changed()* method. These Dependents can either be Views or Controllers. Both can access their Model. Views can have sub-Views. *The reference-holding entity has a diamond at the start point of its link.*

Figure 5.9 shows the new type of component called *VContext*. A number of functions/attributes to handle their (de)activation can be identified. The two functions *contextChanged()* and *interaction()* handle the two types of events we have talked about. An evaluation function is characteristic for each context and returns true if the context event received activates this Context component. Controllers can be added to Contexts as children and other Contexts need to be added as siblings in order for the chain of responsibility to work.

Controllers handle interaction events. In the SPIRIT case, we can identify three interaction events: pressing one of the two Active Bat buttons and moving the Active Bat. Spirit Controllers will be able to handle these events and interpret them according to the Context they are attached to.

The figure also shows that the components provided can be extended in order to suit a particular context-aware environment. What this means is that the communication protocol between the components, which is a bit complex is taken care of automatically. It has been done reliably once and will pervade the whole architecture.

5.5.4 Integration

We shall now show how the architecture we have been describing since the start of the chapter can be complemented with the (extended) MVC paradigm in order to provide the missing link between the application logic and the Renderer.

Most steps will be equivalent to the use of MVC in GUI environments. For any location-aware application developers will need to determine domain objects. The way this framework has been used so far is that Models are linked to the Ubicomp backend by calling remote procedures.

In this context it is helpful to think about benefits the SPIRIT model has for Ubicomp infrastructures. The main benefit of this backend is that it provides *uniform access* to *active objects* in the environment. Proxy CORBA objects (in the computing sense) of actual active objects, devices and services are kept in a single world model. In the design presented in this

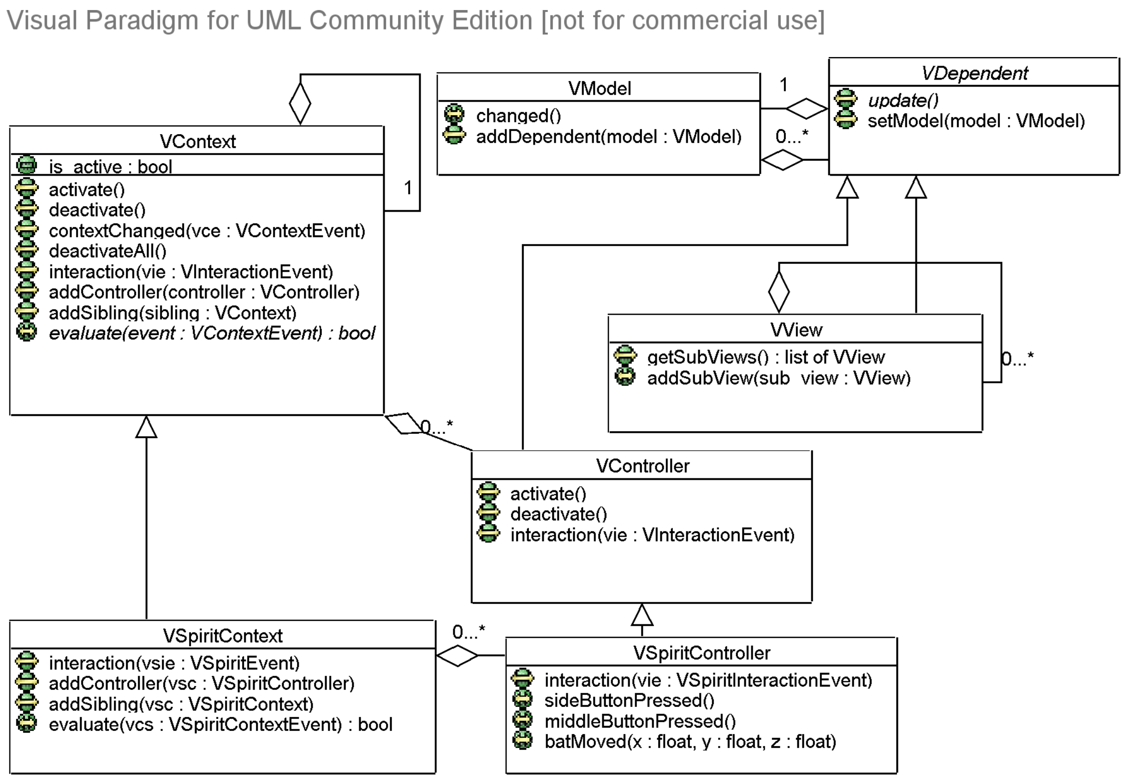

Visual Paradigm for UML Community Edition [not for commercial use]
VContext
is_active : bool
activate()
deactivate()
contextChanged(vce : VContextEvent)
deactivateAll()
interaction(vie : VInteractionEvent)
addController(controller : VController)
addSibling(sibling : VContext)
evaluate(event : VContextEvent) : bool
1
VModel
changed()
addDependent(model : VModel)
1
0...*
VDependent
update()
setModel(model : VModel)
VView
getSubViews() : list of VView
addSubView(sub_view : VView)
0...*
0...*
VController
activate()
deactivate()
interaction(vie : VInteractionEvent)
VSpiritContext
interaction(vsie : VSpiritEvent)
addController(vsc : VSpiritController)
addSibling(vsc : VSpiritContext)
evaluate(vcs : VSpiritContextEvent) : bool
0...*
VSpiritController
interaction(vie : VSpiritInteractionEvent)
sideButtonPressed()
middleButtonPressed()
batMoved(x : float, y : float, z : float)

Figure 5.9: Modelling Context-Aware Interaction in Architecture

chapter the Models are images of these objects, containing only the attributes that are important for the application at hand. This, of course, only refers to Models that map to active objects, devices or services. A typical location-aware application will have other data structures that map to Models such as lists etc. A detailed design example will be given in the Chapter 7. Figure 5.10 shows our framework. Note the link between the backend and the application on one side and the link between the application and the Renderer. The application uses MVC triples and Context components, the Views being tightly coupled with the scene graph.

We discussed earlier that the Renderer contains a scene graph whose transforms need to be updated as the user moves in the environment. In the system implemented for this thesis the Renderer reads head tracker values in directly and updates the user's view transform in the scene graph. However, transforms relating to active object positions, as opposed to the user's *view* transform, are updated through the path through the extended MVC architecture. There is a possibility to unify the way the architecture handles head tracking values and active object location changes using the tracking system described in the previous chapter. Please refer to Figure 5.4 for a proposal.

Apart from transforms the scene graph also contains the virtual objects, i.e. the visualisations. In the MVC paradigm the entire visualisation the user sees of a system is made up of Views that are hierarchically organised. So, in our case the Views need to map to part of the scene graph, i.e. they each contain part of the scene graph. That particular part describes the presentation of their Model.

The View also needs to be able to change this part of the scene graph according to the current state of the Model. The current state of the Model is checked whenever the View's *update()* function is called. How is the View *hierarchy* mapped to the scene graph hierarchy? Well, each View comes with a small scene graph describing its look. When a View is rendered

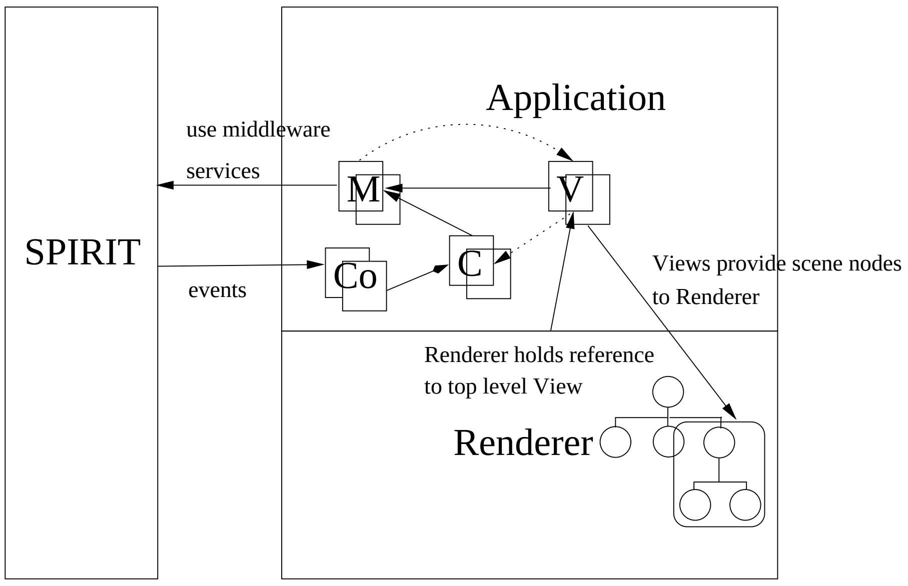


Figure 5.10: Architecture of the framework. ‘Co” signifies Context components.

it makes sure the scene graph it builds contains the sub-View nodes. This is done at each level.

Figure 5.11 shows how the MVC architecture interfaces with the Open Inventor based Renderer. First of all a View is extended to contain Open Inventor specific information (hence the postfix *OI*), such as a mini scene graph whose top node is called *view* and a viewer, which is the canvas the graphics API renders to. The function *attachSubViewSceneNodes()* is used by a View to add the sub-View scene graph to its own. The Renderer gets a reference to the top level application view and renders the complete scene graph held in the top application view.

The real power of hierarchy comes from the fact that Views can be attached to any other virtual object of the application. Typically, the application will contain a number of overlays on active objects or devices. From a software architecture point of view, it is trivial to attach a menu to any of them, because the process of rendering has been standardised in a way that if you can manage to get one View to display on your HMD everything else will display at the the right place.

Given that the scene graph lends itself so elegantly to the organisation of hierarchical views, it becomes clear why the extra effort of integrating a scene graph-based API as described in Chapter 3 pays off.

5.6 Comparison of MVC Design Patterns

In this chapter we introduced an extended MVC design pattern. This pattern has certain limitations that may not always be valid, such as the exclusive nature of user contexts or the ability to model user context only. Also, it has only been tested with location-aware applications. There are many ways to extend the MVC pattern and in this section we shall look at some properties of the MVC pattern in order to give readers some background knowledge so that they can adapt this design.

There are more than one ways to implement/adapt MVC. We briefly talked about an implementation based on the classical MVC paradigm. While responsibilities of Models, Views and Controllers are obvious, the way the three components interact allows for variability.

The first thing we can agree on is that a View hierarchy is needed. The view the user sees

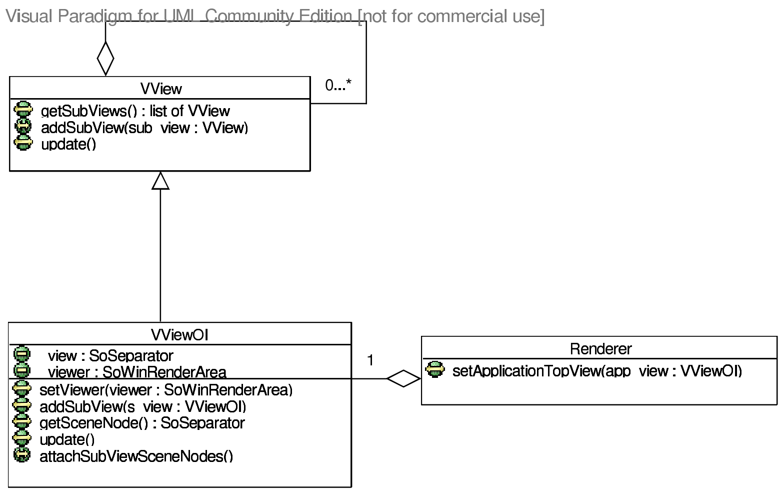

Visual Paradigm for UML Community Edition [not for commercial use]
VView
getSubViews() : list of VView
addSubView(sub_view : VView)
update()
0...*
VViewOI
view : SoSeparator
viewer : SoWinRenderArea
setViewer(viewer : SoWinRenderArea)
addSubView(s_view : VViewOI)
getSceneNode() : SoSeparator
update()
attachSubViewSceneNodes()
1
Renderer
setApplicationTopView(app_view : VViewOI)

Figure 5.11: Integration with the Renderer

is logically grouped into components that belong together and need to move together.

The variability starts with whether the interface to the middleware shall send events directly to each Controller or whether it should inject it somewhere in the architecture, knowing only the right Controller will pick it up. First of all, we need to know whether the middleware can provide such specific events. If events require some evaluation before they are dispatched to the right Controller some evaluation component such as our *VContext* is needed.

Secondly, we should keep in mind why the original implementors of the GUI MVC paradigm injected events into the hierarchy, rather than send them to a specific controller. In fact, they used the View hierarchy in order to let an event trickle down the hierarchy. The problem they were facing, however, was that they needed this process to find out which window was supposed to receive the, say, mouse event. It had to be the one "on top". Our middleware might not need such a process to determine the right receiver.

If this is the case, we can construct a simpler architecture. Sometimes the event the middleware spits out is useless to all but one Controller. The cost of not letting the Controllers decide who shall receive the event is higher coupling. Whenever we add a new object the interface to the middleware has to be changed. In the original MVC implementation the sender does not need to know the receiver.

Our extended MVC design pattern is one way to adapt MVC. What needs to be remembered is that its main aim is to model context-aware interaction by assuming that context and explicit interaction are orthogonal, e.g. it has the ability to elegantly handle a button press differently according to the user's context. This is because once the context is changed a whole line of Controllers providing interpretation of events is switched. So far, only the user's context has been modelled and not object contexts. These end up as attributes in the object's Model. Also, we have not experimented with introducing sub-contexts. This design pattern's benefits are again lower coupling, the fact that existing protocols implemented in base classes can be reused and a cleaner design. Its drawback is that it could complicate the architecture unnecessarily.

With this background knowledge the designer should hopefully be able to make an informed decision as to how to solve a particular interaction problem.

5.7 Rapid Prototyping and Simulation for AR in Ubicomp

Another aspect we can see is that this architecture supports composability. Identically to the way MVC supports composability in GUI environments complex widgets can be constructed from more basic ones using sub-Views. This is desirable from a developer's perspective when it comes to extending this framework with more and more widgets that can be plugged in to solve interaction challenges.

Furthermore, it can be observed is that the framework fosters rapid prototyping. Due to the fact MVC separates application logic from presentation developers are put into the position to develop these separately. This was, in fact found to be a highly desirable feature.

Location-aware applications can be developed in the following manner with this architecture: You start with a small part of your location-aware application. In order to get a working prototype you design and implement the Controllers and Models. You now have a working location-aware application. You then add the Views to the application and it becomes interactive. The application is then easily extended by adding more Models and more Controllers. Once the extended functionality is tested you can add the Views again and so on. This kind of incremental development is possible due to the low coupling between *MVC-triples*.

Another benefit of being able to plug in Views at any time during the development is that we can prototype interaction in a non-AR desktop environment and just need to plug in the scene graph into our application base in order to turn the application into a real AR application. This kind of prototyping, even though it has not been used in AR so far can accelerate the implementation process dramatically. Setup times (setting up the helmet, calibrating etc.) for AR experiments can be a couple of minutes. Programmers performing usual non-AR debugging easily perform dozens of test runs within an hour. It proved very useful that most of the programming work could be done off-line.

In order to take this further we can even simulate interaction properties by providing a new Controller. In order to simulate events that would normally be sensed by the Ubicomp backend the prototype controllers attach themselves to the same Model and View, but take keyboard or mouse events instead.

This idea can be taken even further. A Simulation Environment for Ubicomp was constructed on the same basis. A whole room was modelled in 3D and used as an environment to place a number of virtual interaction objects into. First, the interaction widget was placed into the virtual simulation environment. Then its properties were tested by simulating Active Bat movements, essentially simulating a person who is using her Active Bat. When the Active Bat was brought into specific regions the visualisation of interaction objects would change correspondingly, providing feedback. The developer is able to simulate a person moving through this environment and interacting with it.

The architecture is so flexible that developers can easily decide what to simulate and what not to. For example, in order to replace the Active Bat simulation with actual Active Bat events, one only needs to change a Controller.

The most useful part is that it is a trivial task to adapt the entire application for the Augmented Reality application base, turning it into an application that reacts in the same way and has interaction objects in the same physical location as the simulation. Figure 5.12 shows a view of the simulation environment. Using this environment we can simulate a walk-through and interaction in the actual Ubicomp environment. One can see a number of red polygons that signify particular regions. When the user puts on her HMD and looks at the actual room she will see exactly these polygons overlaid on reality.

Related to this issue is the benefit that programmers who extend the framework do not need to know a great deal about how Augmented Reality works. The application base works in a

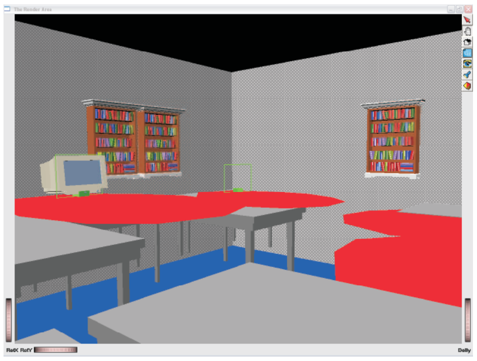


Figure 5.12: A view of the Simulation Environment. Using keystrokes the developer can simulate moving around in the actual environment.

way that most of the mundane tasks are taken care of automatically, allowing developers to concentrate on using their creativity to solve interaction challenges.

An interesting extension to this would be to let developers place widgets into the simulation environment by a Direct Manipulation Interface and specify regions and what is supposed to happen if events take place in those regions. The result would be a visual programming environment for location-aware computing. If end users are ever to write their own applications this is a promising approach. In any case the simulation properties of this environment will come handy for end users. Very often the conditions end users want to specify for actions that occur when something else happens (e.g. play music when I enter the office) can become complex. In such cases it is beneficial for the user to be able to check whether the actual location-aware application will perform as desired.

5.8 Related Work

In this section this framework shall be compared to a few reference frameworks in the field. The closest framework in the field of Ubiquitous Computing is the Context-Toolkit by Dey et al. [86]. Its purpose has been to make building context-aware applications easier and in doing so it represented the first important step to engineering context-aware applications in a repeatable manner. It uses a a set of components called Context Widgets (Interpreters and Aggregators) in order to acquire context in a simple way, abstract it and fuse it in a way so that applications can use it.

First of all the notion of an application that is separate from a system is present in Dey et al.'s framework, too. The framework itself, however is targeted at what we have understood as the backend, i.e. at context acquisition, interpretation and abstraction. Its purpose is not enforce good interaction design but rather to enforce good software engineering practices. Similar to Controllers in our framework their Interpreters and Aggregators interpret an action. One difference is that they do not necessarily cause an action, i.e. a change in the application. Rather do Aggregators and Interpreters provide context for potential use. This results in a weaker causality than the framework presented here. Even though "context servers" are popular in Ubicomp infrastructure design (also see [87] and [88]) there are problems with this approach.

On one hand it is appealing to offer generic context for many application to use. On the other context can not really be separated from a specific activity as Dourish [89] has argued. Rather it may be more desirable to embed context-awareness fully into an application. The approach taken in this framework does exactly that by interpreting context within the application.

One has to remark, however, that our framework has only been used for location-aware applications and Controllers are simpler and *less* ambitious components compared to Interpreters. They result in a instant change of a Model. This Model can be a data structure or a change in the backend. So, another difference is that when a context-aware change occurs a View is “in the loop”. After all, each Model has one or more Views attached to it.

All in all one can say that our framework supports the design of interactive applications that use context, rather than focussing on providing context for any application. One of the important implications of this is that this architecture specifies and, in fact, provides inter-component communication protocols. Developers only need to subclass existing classes in order to develop their own applications. In this way they can concentrate on functionality their specific applications provide.

Gaia [90] uses a MVC-like paradigm in order to manage input and output to and from applications. The whole application’s configuration maps to just one Model. One of the aims of the original MVC was to program understandable user interfaces as described by Shneiderman [91] by enforcing a visualisation of an application’s internal structures. In Gaia, however, the MVC-like architecture has a purely technical aim of managing I/O to applications.

Similarities and differences to two main Augmented Reality frameworks were discussed in Section 5.2.

Recently there has been one effort to combine AR and Ubicomp by Newman et al. [59]. Using the Active Bat system and a inertial tracker they were able to deploy their system in the entire building. Their architecture that is SPIRIT-based as well is not so much concerned with introducing interactivity but with visualising basic sentient information.

As far as the simulation environment is concerned, there has been some work on simulation in Ubicomp as well. Ubiwise [92] simulates a 3D environment for developers to test scenarios with simulated devices. In contrast to the work presented here that is a full-fledged simulator with many functions. Unlike the work here it is targeted towards simulating devices rather than location-awareness. One important difference is that our simulation environment runs the actual code of an augmented location-aware application, where as Ubiwise is more suited for scenario analysis.

5.9 Many Applications - Many Users

Now that we have discussed the idea of an application, we can start to think of what implications it has for the architecture if we are in an environment where there is not just one application but many running simultaneously with many users in the environment.

In our mobile experiments the user had the application running on her laptop. This is not essential if the application can communicate with the Renderer remotely. Only the Renderer would need to run on the user’s laptop. The Renderer would access the Views remotely and build up one scene of the environment.

The first question is whether users shall share the same set of applications or each should have her own set. Since location-aware applications are always tailored to a particular space it makes sense for all users to share the same applications in the same space. In order to provide personalisation, services that take parameters can be run, such as reminder services.

There are two issues here. First of all we need to have a mechanism of enabling users to register/unregister for applications. Models need to be kept for each user. Technically, this might not be difficult but it poses new challenges to the user interface we shall discuss in the next chapter. In order to construct such an architecture we can draw from CVE (Collaborative Virtual Environments) architectures. CVE [93] faces similar problems in the sense that many users share an application and access to data needs to be controlled through an appropriate interface.

Secondly, in an environment running lots of such applications there are inevitably going to be clashes. In principle every application can be associated with one or more regions. The problem occurs when applications have overlapping operating regions. An event can have different meanings according to which application is active. In GUI environment this is handled by the fact that only one application's Controller is interpreting events at any one time. I.e. only one application is in focus, the one on top. So what is required, is a mechanism to keep track of which regions are being used by which application and either prevent overlap or allow a hierarchy and application switching. Note the similarity to a GUI environment that deals with this problem in an exemplary manner.

In the architecture presented the Context component is associated with exactly one region. Having only one Context component that is remotely accessible for all applications is the most obvious way to implement this using our architecture. As far as the interpretation of events is concerned, this architecture is very flexible. As soon as a context, in our case region, is entered the interpretation of whatever takes place while in that region is performed by Controllers attached to that particular Context. Each Controller affects one active object. More than one Controller can be interpreting user action simultaneously. On the other hand this framework provides the facility of activating and deactivating Controllers. So, exclusive control can indeed be presented, i.e. only one active object will react to a user action.

5.10 Conclusion

We started the chapter off by recognising the need for a framework that would integrate a Ubicomp backend with Augmented Reality and add Interactivity on top of it all. In order to accomplish this ambitious task we needed to separate responsibilities of the framework as opposed to the ones of an application. We constructed an application base that can be used as a skeleton to build location-aware Augmented Reality applications on.

This mainly involved structuring callbacks, polling and graph-building mechanisms in a manner that they stay out of the developers way. A new architecture was proposed that would make developing and prototyping applications easier.

In addition to that an architecture was designed that inherently supports building flexible and extensible interactive location-aware applications. It allows the implementation to communicate itself to the user to great detail. A new design pattern based on MVC was introduced in order to elegantly handle the interplay of explicit and implicit interaction so often encountered in location-aware environments. In this framework the emphasis is on good application design, location-awareness just being one of many functionalities embedded within the application.

Chapter 6

Interaction Prototypes

The previous chapter was concerned with what one might call user interface software architecture. It provided the building blocks for adding interactivity to a location-aware environment. In this chapter we shall show how to solve a variety of interaction challenges typically encountered in Ubicomp environments. Some of these have already been discussed in the introductory chapter of this thesis.

In order to show the broad spectrum of problems now solvable we shall start with designing prototypes that solve typical interaction problems in Ubicomp. Gradually increasing their complexity, we will then move on to show how these prototypes can be used in a full-fledged application. Chapter 7 will go further and show a design approach that can be employed in order to build a general interactive spatial application. Chapter 10 will then extract principles from the experience gained from implementing these prototypes for the benefit of all Ubicomp application designers.

The device we want to concentrate our interaction design around is the Active Bat (Figure 6.1). The Active Bat is a small device about the size of two fingers. It mainly functions as a sensor, transmitting ultrasonic signals which are used by the infrastructure to calculate its position to up 3 cm. In order for this to work the entire building's ceiling needs to be covered with ceiling ultrasound sensors.

In addition to its function as a sensor the Active Bat has two buttons. Pressing these buttons can be sensed by the sensor system and can cause an action in the middleware, i.e. the user can interact with the environment with this two button interface. One of its drawbacks when using it as an interaction device is that the Active Bat does not have any display. Nevertheless, it can provide some feedback by beeping. We need to keep in mind that its designers wanted Active Bats to be attachable to most everyday objects and also be wearable on the users' chest. Power consumption needs to be minimal if batteries are to last for months. Its deployment has justified its simple design: The Active Bat is the only device most users in our lab are ready to carry around at most times. This requirement is the most vital for an interaction device. Already, designers have come up with new suggestions to use these pervasive devices in our lab such as the use of the Active Bat to control a mouse cursor from a distance, its use as a reminder etc.

Nevertheless, these devices are the cause of much frustration, mainly because they do not fulfil the expectations users have from an interaction device these days. During user trials (see Chapter 8) one user described them as the "most unergonomic devices ever made". It is not surprising to see why. The device hardly offers any feedback and the information the user can send to the system is comparable to the information you would get from 2 LEDs (this is just an information-theoretic comparison), i.e. 4 bits. Compare this with your interaction facilities your desktop offers you.

We shall try to use 4 bits to tame the Ubicomp environment nevertheless. One of the

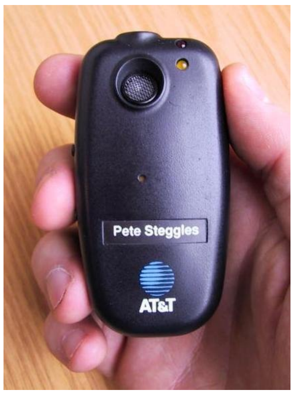


Figure 6.1: The Bat is a small device that is about 85 mm long. Two buttons are located on the left side.

issues we will be concentrating on is how to introduce *explicit* interaction into the Ubicomp environment.

The interplay of implicit and explicit interaction in location-aware environments has hardly been examined. Some of the reasons why designers steer away from introducing explicit interaction into a Ubicomp environment were discussed in the introductory chapter. At the same time it is indispensable for a controllable environment in which humans set the rhythm rather than the computer.

But before we look at explicit interaction, we shall examine how much we can get closer to our goal of making location-aware computing understandable just by visualising what is going on and what is possible.

6.1 Affordances and Feedback in Space

6.1.1 Intent

To let the location-aware applications reveal information objects with spatial aspects to the user.

6.1.2 Motivation

The lack of affordances and feedback in many Ubicomp applications was described in the introductory chapter. In location-aware applications these problems usually arise from the fact that it is not always feasible or possible to provide feedback or show what is available at the user's current location.

The introductory chapter describes an application that prints your job as you approach a printer [11]. What is needed in order to make this application understandable and predictable for anyone? And what happens if you think you are standing next to the printer but the printer does not "think" so? Regular users of location-aware applications find the "nothing happens" situation the most unintelligible situation (see Chapter 8 for some user comments), even though in many cases a tiny piece of information (e.g. "move slightly right") could resolve the situation.

So, the paradoxical situation we are dealing with here is that by adding location-awareness to an application we have *under some circumstances* increased the cognitive effort required to interact with the application. The user has difficulty performing even the most basic action if she is left guessing as to *where* her action needs to be performed. Were location-aware applications known to be highly reliable this might have been excusable because the "nothing happens" situation would only occur once in a while. This way, however, the first thing users need to do when it does occur is to test out different locations to make sure their Active Bat is actually located in the appropriate active region. As Chapter 8 will show, even experienced users often only have a vague idea of how these regions look like.

The example of the application also shows that only showing what is available is not enough when we are dealing with mutable conditions in space. The printer will only react *if* the Active Bat is within a particular region. In order to make the application intelligible we need to go beyond showing how things are supposed to be to what is actually happening. What is needed is a way for location-aware applications to provide feedback.

6.1.3 Design Rationale

The concept of affordances was introduced by Gibson [94] in order to describe a purely biological link between (directly perceived) information in the environment and human action. It was used to explain why animals and humans a priori avoid certain actions with certain objects, e.g. why they don't try to walk through walls: "[...] a glass wall affords seeing through but not walking through, whereas a cloth curtain affords going through but not seeing through. Architects and designers know such facts, but they lack a theory of affordances to encompass them in a system".

This concept was developed further by Norman [7] who proposed that by putting affordances into technology one could achieve the same effect. One important point is that he allows the process of recognising "affordances" to require knowledge of cultural conventions as opposed to Gibson.

Reading some of Weiser's examples for invisible interfaces [95] we recognise a surprising similarity to the theory of affordances when he describes a Ubicomp interface for pilots: "You'll no more run into another airplane than you would try to walk through a wall". The similarity stems from the ecological approach both take, i.e. the idea that because humans dwell in their environment human activity cannot be separated from its environment or context.

So, what we see here is that affordances are potential enablers for Invisible Computing. *Perceiving* what a particular smart space "affords" in terms of interaction facilities is vital in reducing the cognitive load in finding what a user can or cannot do in a space. Extending Norman's work, Hollan et al. [96], point out that people "off-load cognitive effort to the environment" and "establish and coordinate different types of structure in their environment". The way objects in the environment are presented to us can reduce cognitive effort. This does not only apply to affordances. We usually organise our desks, in fact the entire environment, in a way that increases our productivity.

So, humans use their environment to "become smarter" [13]. We can make use of this insight when visualising spatial profiles. One example of a spatial profile are so-called "quiet zones" used in the SPIRIT/Active Bat system. In such a system users in an office building are tracked very accurately by Active Bats. This has many beneficial applications but for reasons of privacy users can nevertheless opt to remain untraceable for the system. They accomplish this by declaring a small location as their Active Bat's "quiet zone". Within the quiet zone the Active Bat does not transmit its position. One of the problems user are known to encounter is that they cannot remember the locations or don't know where exactly it is. Generally, spatial profiles contain preferences that are only valid in a limited region of the physical space, set by users for use by applications.

Coming back to our printer example, we see that the main reason for unintelligibility is the lack of feedback. We realise there are two kinds of feedback involved in this case. The first type is the one well known from traditional computing, informing the user that something has happened. The second type of feedback needed is about what the computer "understands" as "near the printer". This is due to the fact that, even though the computer tries to imitate the human by using space as a common language [80], the concepts of nearness do not quite coincide. Therefore it is important for the user to know what the system understands as "near" and where the system thinks the user is.

6.1.4 Implementation

The approach adopted was to use very simple cues in the form of 3d graphical objects. In order to understand what a computer or printer understands as "near" the region the specific location-aware application uses *internally*, was visualised as a polygon in space. By using internal sensor readings in order to overlay an outline on the Active Bat the user can reliably tell where the system sees the user's sensor. When the user brings her Active Bat into a region the overlay changes colour. These are very simple visualisations. Chapter 7 will show more sophisticated examples. But for the time being we can see that even these simple cues if applied to all location-aware applications can eliminate a large number of "breakdowns".

Quiet zones were visualised as cubes in the environment. User don't need to rely on their memory or pieces of paper anymore in order to find what the system regards as quiet zone.

As an extension, one could similarly visualise the signal strength of wireless LAN. After a survey, signal strength information could be visualised encoded in colour, shape or size of spheres, for example.

The main requirement for these visualisations is a link to a world model that describes properties and an architecture that supports their display and update in space. Basically, exactly what was presented in the previous chapter.

6.1.5 Other Envisioned Uses

One of the opportunities location-awareness gives us, is to offer localised services, i.e. something is offered to the user, in a limited region of the environment only. The user typically interacts with such a service through a wireless link. For example, users may only be able to send or receive messages in particular parts of a building. The reasons for having such services vary from security concerns to services that are only locally relevant. In some cases physical limitations of the wireless connection impose spatial constraints. Consider, e.g. wireless LAN. It is offered in many offices and public places but one problem is how users can find out where exactly they can receive it and where it is strong enough. As airwaves get crowded more and more so does the amount of information required to *choose* a service.

6.2 1-bit Information Feed-forward

6.2.1 Intent

To provide for timely user intervention into an automatic inference process at minimal cost.

6.2.2 Motivation

There is a location-aware application [97] that uses people's "active" mugs in order to infer if there is a meeting going on in a room. If two or more hot mugs (they are fitted with temperature sensors) are sensed in a meeting room the doorplate automatically changes to "meeting". It is not difficult to think of cases where this simple inference might not be according to the user's wishes, e.g. the users could be having lunch and in fact appreciate other people joining in.

Other applications of this class include applications that play a particular song when the user enters a room or lock/unlock workstations automatically as the user leaves or approaches.

As the user walks through the location-aware environment the system is constantly making inferences on her behalf. What happens if a user does not agree with such an inference. In the mug example users could use a different mug. In the other cases users could manipulate their location sensor whose reading is involved in making the inference. In order to perform such tricks the user needs to be informed about internal inference processes. The possibilities discussed in the previous section can come in handy for visualising state transitions of internal objects.

A better way is to provide an override facility in the user interface. One question is where shall this kind of option appear. The other is how does the user tell the system which inference she wants to override. A speech interface that can understand which inference to override is difficult to implement, that is, given the user knows that some inference has taken place at all.

If a universal usable mechanism to turn an inference off could be devised the value of the location-aware services could increase. Horvitz [98] has performed research into what he calls mixed-initiative interfaces. His findings are mainly relevant for traditional agents but some of his principles relate exactly to the problem we are facing in Ubicomp. One of his conclusions is as following:

> The value of agents providing automated services can be enhanced by providing efficient means by which users can directly invoke or terminate the automated services.

6.2.3 Design Rationale

Given that we are working with an HMD we can now think of how we can make use of it in order to introduce user initiative into an environment dominated by system inferences.

The first thing we should ask ourselves is what communication modus shall we use in order to inform the system. Given that specifying what to override is complicated and unpredictable the best way is for the system to *show* the user an option she can select.

The next question is how this user intervention shall take place. The main constraint we are facing is user attention. The option needs to be displayed in a way that it can be ignored. At the same time it should be able to inform the user about its existence. After all, the user cannot know a priori that an inference has taken place. As unpredictable the situation may be for the user, our solution needs to have an element of predictability in it: users should be able to find it when they need it and instantly interact with it. This interaction should not take much more than a simple hand movement. After all, we just need to send a very simple command to the system.

One of the problems a solution needs to overcome is that there cannot be a fixed location where this interaction can take place, such as a touch screen. Rather does this interaction need to take place in the user's current context, i.e. location. If a change of location were required it would defeat the purpose of the interaction.

Another question that needs to be addressed is appropriate timing. The system cannot know whether there is a need to prompt the user on its own. We can make use of one important fact here. Even though the situation that demands intervention is unpredictable to the user, it is predictable to the application developers. So the override mechanism needs to be pluggable into their programs. By providing such a standard facility we can furthermore guarantee that users will not need to learn a new modality for each override.

Finally, there is the issue of understandability. The option needs to be distinctively recognised as an override option. Then, of course we have to deal with traditional design principles, such as providing feedback etc.

6.2.4 Implementation

The Active Bat was described in the introduction of this chapter. We saw that it has two buttons at its side. If used economically these buttons have the potential of solving many of the interaction problems commonly encountered in location-aware environments.

In order to solve the interaction challenge described above the button metaphor was employed. Button metaphors are used in desktop computing in order to perform tasks such as confirming an action. In order to port such a button into the augmented world an interaction modality needs to be decided upon. Since we are using the Active Bat as our main interaction device we will regard it as equivalent to a mouse in desktop GUIs.

The user will see a virtual AR button appearing in the environment at arm's length. The user can then bring her Active Bat close to it. The virtual button (we shall call these *virtual* buttons in order to avoid confusion with the Active Bat buttons) will then signal that the Active Bat is in range and change its appearance. The user will press one of the Active Bat buttons and affect the action. Similar "buttons" have been successfully used in location-aware environments on posters on walls that allow users to perform actions [50]. In the following we will call the poster buttons "SPIRIT Buttons" in order to avoid confusion with the *virtual* AR buttons and Active Bat buttons. See Figure 2.3 for a SPIRIT Button.

The improvements made by its *visual* augmentation are so considerable that we can talk of an entirely different modus operandi. First of all, we are able to make this button pop up dynamically and always near the user. And secondly, we are using a three state interaction model. The button has three interaction states: idle, indicated and selected. The function of the *idle* state is to provide feedback about the state and nature of the object the button controls. In order to be *selected* a button needs to be visibly *indicated* first. The SPIRIT Buttons could not have interaction states because they could not provide visual feedback.

The virtual button is a simple rectangle that can face any direction. It can have any text or image in it. In order to use it, the user brings her Active Bat close to the virtual button, which makes the virtual button change its visual appearance (indication). A subsequent pressing of the Active Bat button (selection) will provide feedback. When the Active Bat button is pressed an event is sent to the backend which ultimately ends up in the virtual button's Controller and Model as described in the previous chapter. The user has the illusion that her "clicking" the virtual button has resulted in an action.

Figure 6.2 shows how such a virtual button can be used in a situation described above. An icon prompting the user for a potential manual override is magnified. It shows a hand signalling 'stop' in front of a computer. We are assuming the virtual override button appears *on the user's mug.*

Where to place the virtual button depends on the application, but in the applications described it is always possible to identify an object or device that is vital to the interaction or another fixpoint. Overrides that concern actions that happen upon entering a room can be placed on door frames. The use of well-designed icons allows to increase the number of override options presented. A few heuristics for icon design will be given in Chapter 10.

One thing that is vital in this context is that we have given the developer the control to decide when and where to place the virtual button. The benefits of an interactive architecture might have become clearer. Let us for a moment design the interaction for the above example.

The diagram in Figure 6.3 shows two user contexts: "Meeting" and "No Meeting". In each context active object events, i.e. location and temperature changes relating to the two mugs, are forwarded to Mug Controllers. Each of the mugs have a view that can be updated according to temperature etc. Object linkages are shown through lines.

In each context the corresponding Door Plate Controller receives context updates, making sure the Door Plate View is up to date. The evaluation function for the contexts is a function

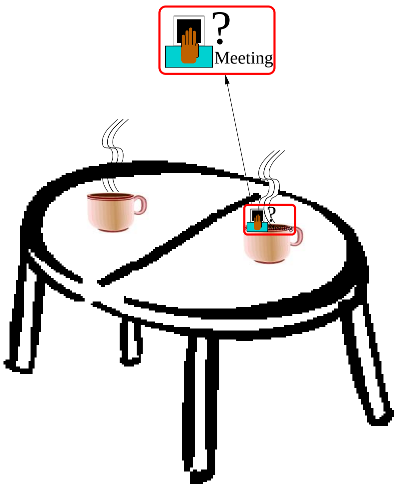


Figure 6.2: When two cups filled with hot coffee or tea are in the meeting room a virtual button pops up allowing the user to override the inference made by the system that a meeting is going on.

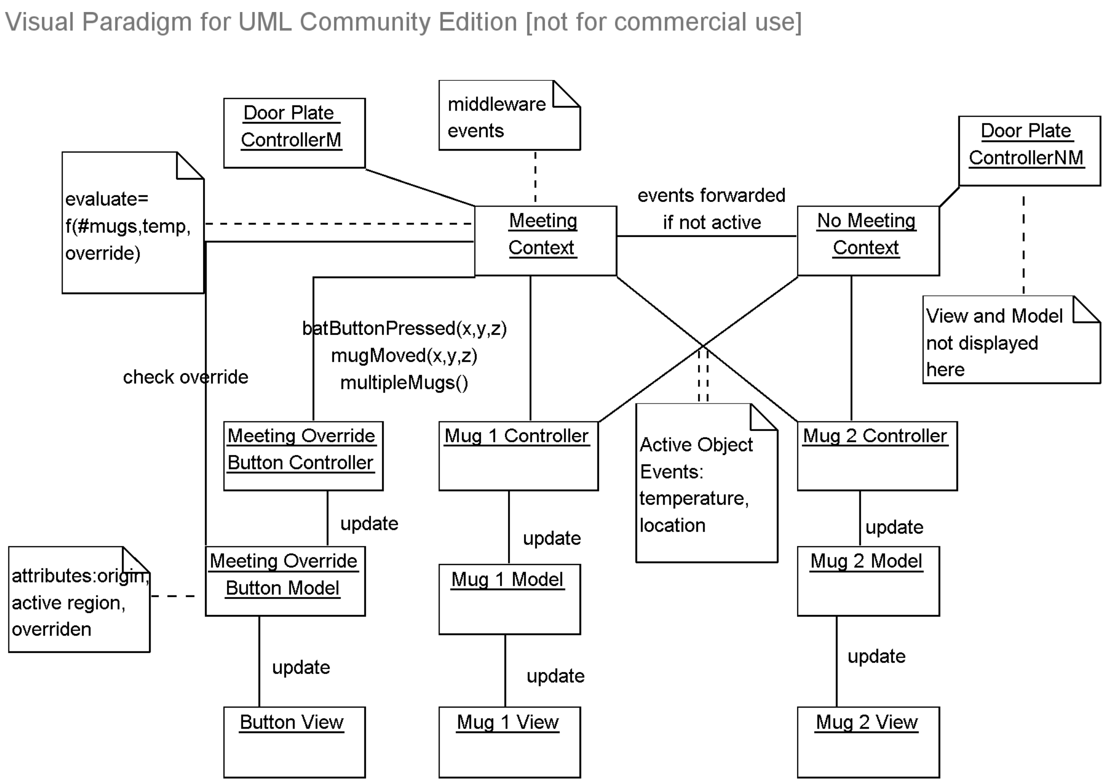

Visual Paradigm for UML Community Edition [not for commercial use]
Door Plate ControllerM
middleware events
Door Plate ControllerNM
evaluate= f(#mugs,temp, override)
Meeting Context
events forwarded if not active
No Meeting Context
View and Model not displayed here
batButtonPressed(x,y,z)
mugMoved(x,y,z)
multipleMugs()
check override
Meeting Override Button Controller
Mug 1 Controller
Active Object Events: temperature, location
Mug 2 Controller
update
update
update
attributes:origin, active region, overriden
Meeting Override Button Model
Mug 1 Model
Mug 2 Model
update
update
update
Button View
Mug 1 View
Mug 2 View

Figure 6.3: Interaction between objects for a virtual button

of the number of mugs, their temperature and the manual override. Since we want the user to intervene into an internal inference process we need to provide a link from the user interface right down to the internals of the location-aware application. This function is where this link ends.

The Meeting Override objects need to store the override state, the origin and the region around the origin of the virtual button where its state can be changed to *indicated.* The origin has been chosen to be on the user's mug and is updated by mug movement events. When multiple mugs enter the room the virtual button is activated.

6.2.5 Other Envisioned Uses

Generally, there are many cases where users in a Ubicomp environment only want to exert a minimum amount of control, namely just want to send 1 bit of information back to the system. Hence, a large number of applications can witness a large increase of usability employing our virtual button.

Settings in the environment that can only take two states, such as ringers, alerts or webcams etc. are likely candidates for this kind of solution as well.

Strictly, speaking the button is more than a 1-bit interaction. After all, the id of the Active Bat is sent in the "Active Bat button pressed" event. This enables a further number of actions, that take the user's id as a parameter. One example could be a virtual button on a wall saying "click here in order to send a birthday e-card to person X".

6.3 Right Options at the Right Time

6.3.1 Intent

To provide shortcuts to the most probable user actions in a particular context.

6.3.2 Motivation

In the previous section we talked about the problem of where to place interaction facilities. The idea discussed was to find a reference point for the interaction and place the interaction facility there. In some cases, however, this is not possible or feasible.

Imagine your room has just opened the window because it sensed the temperature in the room was above your specified preference. You, however, have realised that there are car fumes coming in from outside. How can you undo the last action without needing to walk up to a certain place?

Another interesting class of applications are walk-up and use applications. A lot of the interaction with these applications is sparse. It has to be, otherwise the user would get overwhelmed with so many applications running concurrently in a Ubicomp environment. In fact, sometimes a 1-bit or 2-bit interaction is sufficient, if only the right "bit" were offered at the right time. An example of this could be a volume control.

6.3.3 Design Rationale

In order to design a solution we want to use two strategies in order to increase the information bandwidth from the user to the system. First of all, by offering options from which a user can choose necessarily increases the amount of meaningful information the user can send to the system. Secondly, we want to exploit context-awareness in order to offer options that do not remain constant, but change according to the context of the user. So, the main idea in designing

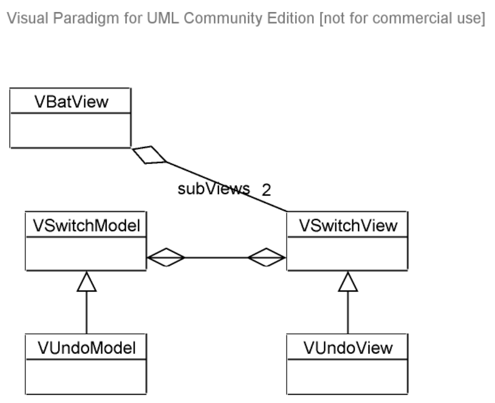

Visual Paradigm for UML Community Edition [not for commercial use]
VBatView
subViews 2
VSwitchModel
VSwitchView
VUndoModel
VUndoView

Figure 6.4: Architecture of hot buttons for the Active Bat

a solution to these problems is to offer not many options but the right options, or at least the most likely.

6.3.4 Implementation

In order to design a solution to this problem, again a well-known metaphor was employed. The "hot button" metaphor known from mobile phones. In mobile phones hot buttons change their function according to the context of the user within the mobile phone's menu navigation. The Active Bat offers two buttons. Each of these is labelled with a text overlay, making sure that these labels and the function of the buttons can change according to the user's context.

Implementing an "Undo" command now becomes easy, if a label is offered on one of the buttons as soon as the system has performed an automatic action.

The naive approach to implement hot buttons would be to store label descriptions for Active Bat buttons in an Active Bat Model and send an event via the Active Bat Controller every time the user enters a new context. Figure 6.4 shows an alternative way to implement a hot button for an "Undo" command. Here, we are actually associating a *functionality* with the Active Bat button. The links between the abstract classes `VSwitchView` and `VSwitchModel` are responsible for the link between the label and the functionality it represents. We are regarding the labels as a sub-Views (each Active Bat has two buttons, so two sub-Views) of the Active Bat View, i.e. they will generally only be a text label provided by the Model of the functionality that the View represents.

The hot button is an example where explicit (Active Bat button press) and implicit (location change) interaction need to be handled simultaneously and, as you have seen in the previous chapter, our MVC architecture handles this elegantly.

We can now think about how to implement the "Undo" command for the example of the closing window described in the previous subsection. Figure 6.5 shows one design. We are assuming that the window has closed and the user wants to undo the last action. In order for this to be possible we need to use command objects [99] that encapsulate all data needed to perform and undo a command. This object is created whenever an un-doable action takes place in the environment. The command object has a function that performs and undoes the action corresponding to the command it represents. After the command object has been created (2) it is executed (2-4) and added to a command list(5). When the user presses the Active Bat button labelled "Undo" (6) the command object is retrieved from the command list (7) and its undo function is executed (8-10). Object linkages are shown through lines. *Please follow through the linkages according to the number sequence.*

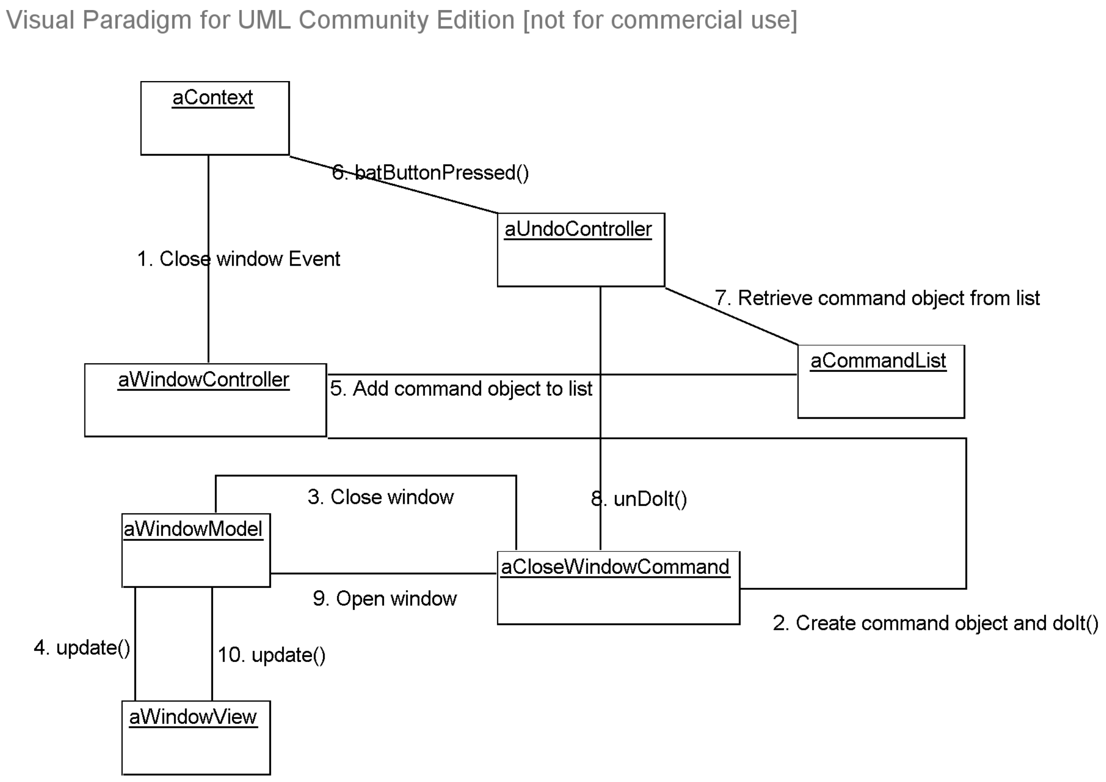


Figure 6.5: Undo command for a Ubicomp environment

6.3.5 Other Envisioned Uses

Similarly, controlling simple walk-up and use applications becomes easy if the system can guess which two options to present. For an MP3 player the volume control is the most obvious. But of course, all possible interactions need to be catered for somehow, even if it means resorting to a remote control. Our strategy is to try to cover as many cases as possible and solving at least a few common challenges.

Of course this modality of interaction can be combined with the ones presented above. Affordances can signal a context around a device that needs to be controlled. Bringing an Active Bat close would change its context, hence displaying other options. A “Properties” interaction could be implemented by bringing the Active Bat close to a virtual handle of an object or region and pressing the Active Bat button.

In other cases the interaction proposed above is too obtrusive. One of the reasons for its explicitness was the fact that override facilities are unpredictable. Hence, attention needs to be drawn to the interaction object.

In desktop computing the more predictable an interaction facility is the less it draws attention to itself. Hence, icons mainly show what happens on a double-click, rather than a right-click (being predictable: the same kind of context menu comes up for all icons). Similarly, implementing a “right-click” for Ubicomp, say a “Properties” interaction would be badly designed if it used the virtual button described in the previous section.

6.4 A Powerful Interaction Modality

6.4.1 Intent

To provide a facility for users to specify system commands.

System commands are the most explicit commands in Ubicomp environments. A user performing these is fully aware of interacting with a computer.

6.4.2 Motivation

In the last section we touched upon the difficult problem of where to offer an interaction facility for actions that are not necessarily associated with a particular device or a location. A simple example is a command like "Don't publish my location on the Internet now".

Another example is turning email notification on and off in a location-aware environment. Users can be allowed to be notified about an email their email account has just received. The solution used by the implementors of the Active Bat/SPIRIT system is to provide SPIRIT Buttons such as in shown in Figure 2.3. There are at designated places on the wall users visit to change their setting by pressing an Active Bat button when it is close to such a poster. The Active Bat then toggles the setting and makes one of two sounds in order to indicate which way the setting was toggled.

The first problem with this kind of solution is that due to the inability to provide an always-on representation of the setting's current state, the application developers have lumped together providing information about the current state and the actual state transition. In other words, you only know what state the setting was in after you have changed it.

There does not seem to be a location users would intuitively prefer to have such an interaction facility. The root problem is that such a poster is associated with a specific location, the concept of turning email-notification on or off is not. What happens very often in practice is that users make their way to such a poster and toggle the setting only to find out that the setting was right in the first place. So, effectively there are three problems that need to be solved: actions that are not associable with a specific location or device, the need to provide information about these settings and separating the provision of such information from the actual action.

Partial solutions to the problem of controlling actions that are not associated with specific locations or devices were offered previously, but they only work if the system can guess what the user wants to do. A semi-reliable solution to the email-notification problem would be to offer such an option on a Active Bat button, before the user enters a meeting. However, what is needed for interaction in such environments is at least one solution that is fully reliable, even if it involves more effort. You should be able to edit, change and view your settings anywhere.

In this spirit, let us look at a few other hard problems that have so far suffered from solutions that do not have a fall-back facility in order to cover rarer cases of user requests.

One of the most frequently used device in Ubicomp scenarios is the smart printer ([39], [11]) that automatically prints out your job when you are standing nearby. A typical scenario could include a lecturer printing out handouts at an unknown location. Hodes et al.'s scenario [100], for example, ends with:

> A minute later, you are notified the print job has completed, retrieve your printout, and return to finish the lecture[...]

Another scenario [101] is described as:

> For example, Esquirt enables a mobile phone to send (or"squirt") the reference (e.g. a URL) of a document to a nearby printer. When the printer receives the reference, it retrieves the document and prints it for the user.

In each of these scenarios the user is relieved of performing a series of tedious operations in order to get the desired result. From experience we know that the case in which a printout is right the first time round is not exactly the norm. Usually, tuning printer settings is required, at which point we have to resort to the desktop.

Most location-aware applications steer away from cases that are not straightforward since they involve interaction that is difficult to design or apparently breaks the Ubicomp paradigm of computers working in the background.

Nevertheless, it should be tried to provide a solution that can handle another chunk of "exceptional" cases. Printing is just one example of walk-up and use applications that require more information, but it has been used here because printing is so pervasive. The next chapter will deal with a more uncommon example.

6.4.3 Design Rationale

In order to solve this problem we need to decide on a reference point. The interaction examples discussed in the previous section have in common that they are the kinds of tasks that make the user ask herself: "where can I do that?". Having a reference point you can refer to when there is a problem is helpful as we know from our everyday life.

Partly, the kind of tasks remind us of a PDA menu. Designing a solution using a PDA would be on option, but we shall try to come up with the solution that is in the spirit of the others. Ideally we want to continue building up our interaction base by adding new interaction facilities and combining them with old ones. In any case, the amount of explicit interaction that will be required in a fully functional location-aware environment of the future will pose a problem for users who need to keep their eyes and mind on the PDA for every interaction. As discussed earlier, the theory of affordances [94] implies that visualisations need to be in situ in order to be perceived at an instinctive level.

The main requirement is that complex interaction needs to be supported, involving states and multiple settings. It is desirable that our solution is generic enough to be used in many different situations.

6.4.4 Implementation

The solution designed, again, uses metaphors from well-known domains. It was decided to make use of a virtual menu that is overlaid to one side of the Active Bat. Figure 6.6 shows a sketch of the virtual menu that appears next to the Active Bat. The two Active Bat buttons have been labelled. See Figure 7.4 for a menu that was used in an application.

With this decision we have given users in such an environment a point of reference: the Active Bat. It can be seen that this solution is on par with the others. In fact it borrows the "hot buttons". This means that this menu is context-aware. It can change according to the user's context, as can its contents. The art is to fill the contents of the menu with the most appropriate options for the context, i.e. region the user is located in.

The control of the menu was chosen to be cyclic, only one Active Bat button is available to scroll through the items. An important feature is that the menu can be of *any depth.* Using hierarchies we can structure the presumably large amount of options for the user. The items of the menu are little squares (refer to Figure 7.4 for their actual size). They can carry any text, picture or icon. At any one time only on depth can be displayed with the user being able to select/move down or return to the previous depth. All in all this kind of interaction is well-known from mobile phones.

One highly desirable property of this kind of menu is that it can be attached to any object such as a phone, for example. But even when using it with Active Bats only, we can find more

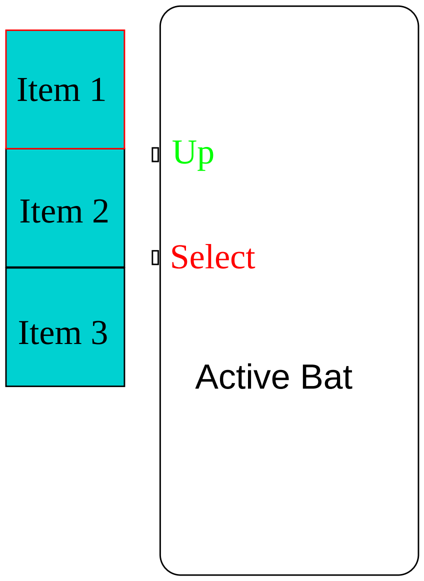


Figure 6.6: A sketch of an AR overlay menu on the Active Bat. The top ‘Item’ has a red outline.

powerful uses. Active Bats are cheap and can be affixed to items as simple as a mug. This again can offload some interaction from the user’s personal Active Bat. As applications develop so will the use of these menus and their interaction.

Of course, there is a limit to complexity that this kind of menu can handle. Large menu structures are difficult to manage for the human mind. But exploiting context-awareness can further increase its flexibility.

Also, let us remind ourselves that not all interaction needs to be solved in this way. Simpler solutions for whole classes of applications were proposed in previous sections. In any case such a menu is a far cry from having no interaction facilities at all. This menu was used and tested.

6.4.5 Other Envisioned Uses

One little researched aspect of Ubicomp is the use of modes. Modes describe states in the interaction. In a specific mode the user’s interaction can be interpreted differently than in others. One example of a mode is the use of different tools in paint programs. Modes have been criticised [102] because the user has to keep track of the interaction state. However, there are cases when modes are useful. The most familiar one is an “application” itself. Whether a mode is useful or not depends on each individual case and we shall look at this again in the following chapters.

Our framework so far builds on the idea of an “application”. We talked about the necessity of being able to manage applications that are running in your environment. If there are many application the user has to be able to address an application explicitly even if these cases are rare. The menu can be used to select a specific application. How to implement application switching in the software architecture is a difficult task that shall not be part of our discussion of interaction design problems.

Another use of modes for our system could be to pose general queries, such as “which devices are recording?”. The system could then switch into “paranoia” mode. All relevant devices could change their appearance to suit the current mode. In this way, you can avoid cluttering your view with information that is hardly ever needed.

6.5 Related Work

HP's Cooltown project [54] is one of many PDA-based solutions that help the user interface with a pervasive infrastructure envisaged in all ordinary buildings of the future. Unlike ours their middleware is based on the WWW. Using their PDAs users can walk around, find and interact with services.

Stanford University's istuff [103] uses physical devices such as buttons, sliders, knobs, mice, pens etc. to send commands to the middleware. Their input capabilities are much richer and perhaps more natural. This, however, also has the disadvantage that you need to learn new interaction styles for each device. Also, you cannot carry them around and they are not as flexible as virtual interaction widgets. Most importantly the feedback, if at all, will require interpretation. The architecture here enforces a kind of uniformness in this respect.

It is, however, possible to combine both interaction styles. At least the architecture proposed here supports it. In this way the user can tradeoff familiarity against naturalness and richness.

Recently, a laser pointer has been proposed for interacting with the Ubicomp environment [104]. The laser is much more limited in terms of interaction possibilities and amount of information the user needs to send to the system. But it shares with this work the idea that users should have a personal device for interaction.

There is some work rooted in 3D User interfaces that uses AR in order to create an amplified natural interface. Using ARToolkit [53] Poupyrev et al. [105] have created an interface in which inconspicuous square 10 cm x 10 cm cards have actions, information or design elements associated with them, that can only be seen through HMDs. The applications presented support industrial or engineering design processes. Our *focus* here is not a new interaction style but what kind of interaction is required in a Ubicomp environment.

6.6 Conclusion

The motivation for the work presented in this chapter arose from a survey of more than a hundred Ubicomp/location-aware applications (some are available at [106]). From these 20-30 were identified as posing difficult interaction challenges. These interaction challenges were then broken down to a few categories. For these interaction prototypes were designed and built.

We decided on using an Active Bat as our universal interaction device. The question was how much could we get out of this simple device using Augmented Reality.

A gradual approach was adopted, trying to solve easy problems first, then slowly combining means in order to tackle more difficult challenges. Starting with the idea of affordances and feedback in space, in order to increase intelligibility we moved on to providing explicit interaction. It was recognised that in many cases the user's ability to convey a minimal piece of information can considerably increase the control over an application, if the interaction is implemented in accordance with common design principles.

Then we moved on to exploring how we can exploit the user's location in order to increase the input flexibility of a simple 2-button device. The final interaction prototype delivered made use of a hierarchical menu metaphor, giving it much more informational bandwidth than the other interaction facilities. Its limits were recognised as was its potential as a cheap and pervasive interaction device.

One idea that was introduced was to regard an Active Bat as a personal point of reference and structure all interaction around it in a uniform way. Classifying kinds of interaction challenges in Ubicomp and systematically trying to tackle them is perhaps the greatest contribution of this part.

Chapter 7

First Interactive Application in Space

In the previous chapter we devised a number of prototypes. The next logical step is to put them into use by solving an actual problem. The aim of this chapter is to describe how to visually augment a location-aware application, the design decisions involved, the resulting architecture, the use of interaction prototypes to solve interaction problems and the overall visual interaction design process; in short everything you need to deal with when creating an interactive location-aware application.

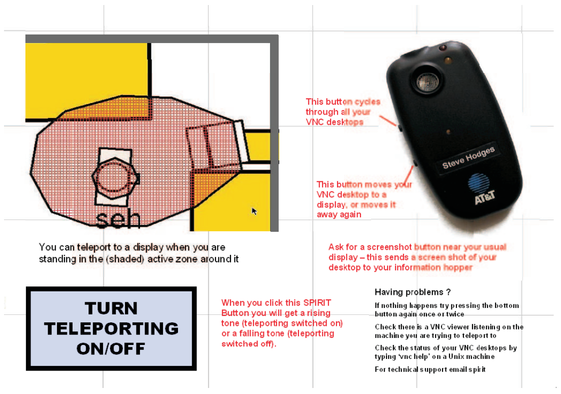
7.1 Problem Description

The application we will be looking at is a Desktop teleport application. This application already exists in the author's lab and we shall try to make it interactive by augmenting it visually. Until now, people using the teleport application perform actions with their Active Bats without receiving any feedback from the computer. We shall try as much as possible to keep the functionality of the application, but just allow the user to visualise what she has been actually doing when she was using the original version of the application.

Many GUI environments allow you to have different Desktops, each containing a particular set of applications, documents and settings. In our location-aware teleporting application, users can walk up to a computer, press an Active Bat button and have a Desktop that might be running on a different computer "teleported" onto the current computer. VNC [107] is used in order to achieve this. VNC stands for Virtual Network Computing and allows users to access their GUI "Desktop" remotely from any computer. The computer running the Desktop locally contains a VNC Client that is listening to "connect Desktop" events from the middleware. When it receives such an event it connects to a VNC server which then sends bitmapped images showing its current screen to the client. The server receives mouse and keyboard events in return: The server can be "remote-controlled" by another computer, in a way similar to that by which X-Server can be accessed in the X-Window System.

It is important to note that users can have multiple Desktops running simultaneously. One use for this application would be to walk up to a computer in the lab, press the Active Bat button and bring up your personal Desktop that contains your email inbox. After checking your email you can disconnect. All of this is done without having logged in or out.

As discussed in Chapter 5, one of the most important abstraction of the SPIRIT system is a physical region or zone. Proximity events are generated by evaluating the overlap of regions. The database contains interaction regions for all active objects. Whenever bats enter these


seh
Steve Hodges
AT&T
TURN TELEPORTING ON/OFF

You can teleport to a display when you are standing in the (shaded) active zone around it

This button cycles through all your VNC desktops

This button moves your VNC desktop to a display, or moves it away again

Ask for a screenshot button near your usual display – this sends a screen shot of your desktop to your information hopper

When you click this SPIRIT Button you will get a rising tone (teleporting switched on) or a falling tone (teleporting switched off).

Having problems ?

If nothing happens try pressing the bottom button again once or twice

Check there is a VNC viewer listening on the machine you are trying to teleport to

Check the status of your VNC desktops by typing 'vnc help' on a Unix machine

For technical support email spirit

Figure 7.1: Posters such as this one are located on walls. Bringing the Bat close to the corresponding "button" (thick-bordered rectangle) and "clicking" will send a command to the environment.

regions and/or their buttons are pressed an event is generated. Programmers can add regions to the database as their applications require.

In the teleport application, when users enter one of these zones which exist around computers in the lab and whose dimensions are stored in the world model the Active Bat buttons (see Figure 6.1) invisibly gain functionality. The upper Active Bat button is supposed to cycle through the Desktops of the user, since she can have more than one running. The user can see a different Desktop on the screen every time this Active Bat button is pressed. It is possible for the user's Active Bat to be in two teleport zones simultaneously. This could be the case if the user is, say, working on two computers that are next to each other and their teleport zones happen to overlap. The lower Active Bat button will cycle through available machines. It is also possible that the user does not want the buttons to control his Desktops at all, maybe because another application is using a zone that is defined to be inside, overlapping with or somewhere close to the teleport zone. Therefore the user needs to have the ability to switch teleporting off so that there is no interference with other applications. Turning teleporting on and off is accomplished by using a SPIRIT Button. Please remember that these are *not* actual buttons but specific locations in the physical space, typically marked by posters on walls (see Figure 7.1). Users can put their Active Bat on the appropriate position on the poster and press any bat button in order to execute an action, in this case turn teleporting on or off.

The description of this application will immediately reveal a number of potential usability problems. One problem is that the Active Bat can take on different functionalities according to where in the 3D space it is located. Note that location contexts can be defined arbitrarily in the shape of regions in the physical space. With many applications running simultaneously this can become a considerable problem. Another problem is the different concepts of a zone the computer and user have. For the SPIRIT system a zone is a collection of 2D coordinates together with an origin and a rotation. The user on the other hand does not (need to) use regions in order to understand the concept of "in front of a computer". The result is that the user and the computer will have slightly different ideas of what constitutes "in front of a computer". Many applications using many overlapping or proximate zones in the environment worsen the situation. The input capability of the Bat has been another source of problems. Not only is it limited but needs to be remembered for every application at every location, unless such a location is marked by a descriptive poster on the wall.

As applications are deployed and evolve more and more functionality is added to them. In the teleport application case the designers soon realised that users needed to switch teleporting on and off. Apparently, the Active Bat does not offer a practical way of adding this functionality. The solution chosen, by using posters on walls, even though successful for applications that are confined to a small area has been found to be somewhat awkward. This in fact is part of a bigger problem. Since a lot of Ubicomp is about personalisation specifying, updating, remembering, even understanding personal settings prove more and more difficult as the application base grows. In our case the personal setting that needs to be remembered only includes whether teleporting is active. In addition to these problems, we face the usual problems of applications without a feedback path. The "nothing happened" syndrome is notorious in the author's lab. Basically, error diagnosis by the user is impossible and the only advice given to users is to try again and email support.

7.2 Overview

A careful analysis of recent proceedings [108] of the main Ubicomp conference will reveal that the interaction design side of things is regularly neglected which, however, is not that surprising, given that Ubicomp is a new and fast-changing field. Another reason may be that the diversity of devices and technologies is so great that a common design approach has hardly been attempted. One of the advantages of having a single interface environment such as a GUI or our AR system is that programmers as well as users can make use of sets of abstractions common to all applications. Ideally there is a good match between the two sets (user's and developer's), so that the interface will reveal some of the underlying structure, eventually leading to a deeper user understanding of the application.

With this in mind we shall employ a design approach used in traditional GUI development that is geared towards creating a better model of the workings of the software in the user's mind. This is also with a view to the usability problems mentioned in the previous section.

On the software side the aim was to make use of the Model-View-Controller (MVC) architecture presented in Chapter 5. As discussed earlier, it originates from the GUI. The difference is that our events are not mouse events, but real world events (including Active Bat events) and our view is a 3D graphics view overlaid on objects and locations in the real world.

The first step is to think about typical user tasks and formulate them in plain language. This is done so that one can identify *information objects* [109]. The information objects, i.e. pieces of information required to do the task, will later end up in the Model part of the MVC architecture. Related information objects can then be accumulated to form a Model. Each Model will then have a View. This will ensure transparency and understandability by design. The advantage of such a methodological approach is that it is very exhaustive in terms of finding information that may be useful with a view to eventually implementing Shneiderman's principle of *Continuous representation of the objects and actions of interest* [91].

Shneiderman's thesis, as proposed for desktop systems, is that the user shall be able to inspect objects relevant to her at all times as this helps her in her interaction with the system.

7.3 Task Analysis

Our task analysis is based on Olsen [109] and consists of answering two questions:

- What do the users want to do?
- What information objects are required?

We will apply the task analysis to our problem with the ultimate aim of identifying Models, Views and Controllers required for our application.

7.3.1 What do users want to do?

Looking at the application description from Section 7.1 we can identify some scenarios describing how users want to use this system. It is always helpful to look at them from the user's perspective:

- I want to check if my teleporting setting is active.
- I want to check whether this machine can do the teleport.
- I want to turn teleporting off.
- I want to teleport the Desktop with my slides on it to this machine.
- I want to browse through my Desktops.
- I want to shift this Desktop to another machine.
- I want to check how many Desktops I have.
- I want to use this machine to check my email.

This list of scenarios is not exhaustive. Any number of scenarios can be constructed. How many scenarios to include is a design decision. More scenarios will result in more functionality. There are user interface that are perceived as overloaded with features and some are perceived as too elementary. Ultimately, putting the application into practice will help finding the right balance.

Looking through the scenarios we can see that some scenarios can be decomposed into basic *tasks*. For example, "I want to use this machine to check my email" can result in an action that can be decomposed into: find out if I can teleport to this machine, choose the right Desktop and do the teleport.

Figure 7.2 shows the result of our task analysis. The leftmost column indicates the tasks we have identified.

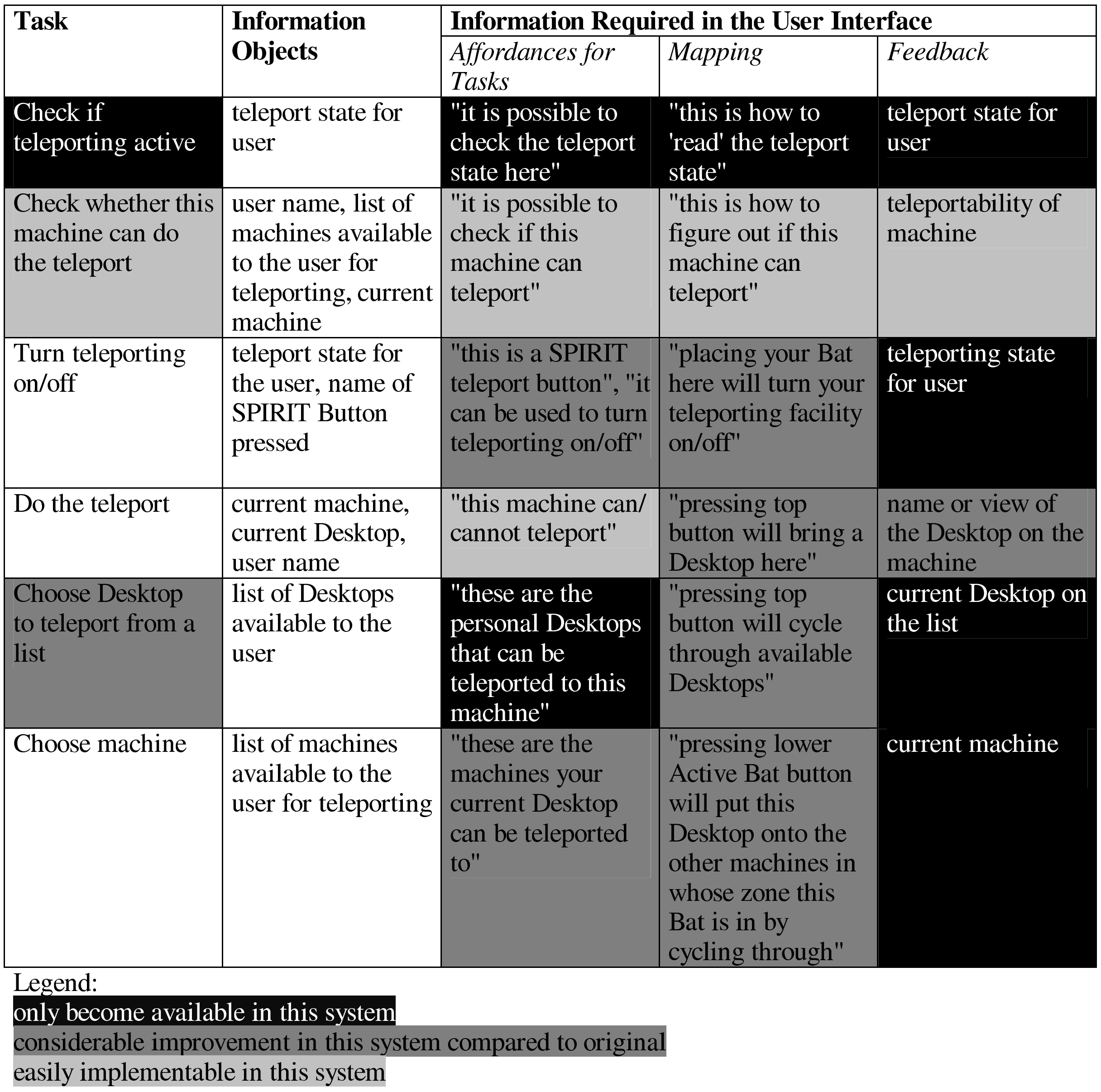
7.3.2 What information objects are required?

The next step is to identify the *information objects* required by the computer and the user. Remaining with Figure 7.2 we can see the information objects in the second column from the left.

For each task we see the information required by either the computer or user to perform the task. Let us take, for example, the first task: check whether teleporting is on. The information required by the *user* is the state of her setting. The second task in our table is as following: check whether this machine can do the teleport. The pieces of information required by the *computer* are the name of the user which machine she is standing in front of and the list of machines available to the user for teleporting.

This analysis continues in a similar fashion for all tasks, each time specifying the information objects required either by the user or computer.

Olsen's analysis [109] contains a further step which is concerned with identifying methods for classes. This is more an implementation task rather than part of gathering requirements for the user interface. Our step towards implementation consists of identifying Models for our MVC architecture.



Task	Information Objects	Information Required in the User Interface		
		Affordances for Tasks	*Mapping*	*Feedback*
■ Check if teleporting active	teleport state for user	■ "it is possible to check the teleport state here"	■ "this is how to 'read' the teleport state"	■ teleport state for user
░ Check whether this machine can do the teleport	user name, list of machines available to the user for teleporting, current machine	░ "it is possible to check if this machine can teleport"	░ "this is how to figure out if this machine can teleport"	░ teleportability of machine
Turn teleporting on/off	teleport state for the user, name of SPIRIT Button pressed	▩ "this is a SPIRIT teleport button", "it can be used to turn teleporting on/off"	▩ "placing your Bat here will turn your teleporting facility on/off"	■ teleporting state for user
Do the teleport	current machine, current Desktop, user name	░ "this machine can/ cannot teleport"	▩ "pressing top button will bring a Desktop here"	▩ name or view of the Desktop on the machine
▩ Choose Desktop to teleport from a list	list of Desktops available to the user	■ "these are the personal Desktops that can be teleported to this machine"	▩ "pressing top button will cycle through available Desktops"	■ current Desktop on the list
Choose machine	list of machines available to the user for teleporting	▩ "these are the machines your current Desktop can be teleported to"	▩ "pressing lower Active Bat button will put this Desktop onto the other machines in whose zone this Bat is in by cycling through"	■ current machine

Legend:
■ only become available in this system
▩ considerable improvement in this system compared to original
░ easily implementable in this system

Figure 7.2: The table shows the results of our initial task analysis. The final implementation did not use a SPIRIT Button to turn teleporting on/off.

7.4 Identifying Models

In order to identify the Models required for our MVC architecture we extract all the information objects from the previous analysis. The left column of Figure 7.3 shows all the information objects identified. We can try to aggregate similar information objects into one Model. Ultimately, it is a design decision how many Models to group the information objects in and how to name them.

Models typically map to both real and virtual objects. For example, the Bat as a real object has a Model, just as the list of available Desktops together with the current one represents a Model. Both will have a corresponding View in the virtual part of the world the user sees. Typically, in Ubicomp applications a lot of functionality is done in middleware. Therefore,

Information Objects	Model
current Desktop	Desktops Model
current machine	Machines Model
list of available Desktops to user	Desktops Model
list of available machines to user	Machines Model
user name	Bat Model
Region (its dimensions)	Teleport Zone
teleport state	Teleport "SPIRIT Button"
name of spirit button pressed	Teleport "SPIRIT Button"

Figure 7.3: Determining Models from information objects after task analysis.

the choice of what should be a Model or not, is influenced by the facilities provided by the middleware. Machines/computers are not explicitly modelled in our architecture, even though they seem to be likely candidates for a MVC triple. The reason for that is that the only relevant property with respect to our design, whether a machine has the ability to teleport or not cannot be retrieved through the middleware. It does not allow us to find out if there is a VNC server running on a particular machine or not, we can only issue a connect/ disconnect command to the middleware and wait for the result. Hence it makes no sense to have a separate Model for the machine that would store the property "has VNC server running".

7.5 Designing the Views

The next step is to take the information from the Models and try to accommodate it in a View so that it uniquely represents the state of the Model at all times. Then further visual cues are added to the Views by analysing the information required in the user interface.

Olsen's approach is good at identifying information required to perform a task. But in order to design good interfaces we need to provide extra information to the user, namely information about *how* to accomplish the tasks using the system.

In order to identify this kind of information we now *adapt* Norman's [7] conceptual model approach. It states that in order to give the user a good Conceptual Model we need to show the user what tasks can be performed (affordance) and how they can be performed (mappings).

We can make use of the tasks already identified and listed in Figure 7.2. There are many definitions of the term *affordance*. Chapter 9 will deal with this in great detail. In the absence of any other suitable term we shall call all information that signals to the user what the system affords (can do) as affordance. Mapping information is defined to be all information needed so that the user can decide what to do in order to get from the current state to a desired state. Feedback is all the information needed in order to determine whether the task is progressing or was completed successfully. In this table transient feedback that needs to be provided during the tasks (i.e. a button has been pressed) has been ignored, since it is too low-level and implementation-dependent. Ideally, there should be a large overlap between the implementation model characterised by the information objects, and the user conceptual model characterised by the information provided in the user interface.

What can be seen from Figure 7.2 is that some of the supporting information required relies on *conventions*. For example, our lab members here know what SPIRIT Buttons are and will instantly understand what a SPIRIT Button can do. These conventions can be either at a metaphorical level such as the SPIRIT Button or at a representation level, which can be general cues for "being selected", "can use with", general permission signs or just icons. These visualisations need to be designed so that they are easily learnable and recognisable. It is evident

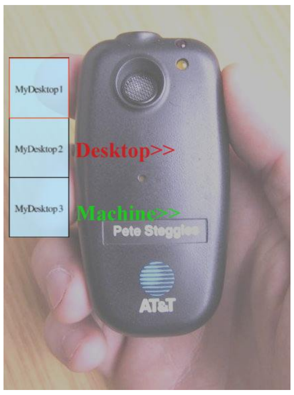


Figure 7.4: The augmented view of a Bat while inside a teleport zone. The menu of Desktops is controlled by the button marked by the overlay "Desktop>>" (simulated picture). The AR HMD is see-through, so the real-life scene is not present in the computer-generated image. Therefore, the composite must be simulated in order to show here.

that we will end up using a mix of traditional conventions (e.g. selectability, icons) and new Ubicomp-specific ones (e.g. SPIRIT Buttons, general permission signs). Implementing more and more applications will reveal which conventions are useful. The author's work comparing traditional GUIs with Ubicomp environments has led us to recognise a few more fundamental Ubicomp concepts that may be worth conveying to the user, such as the already mentioned "can use with" and "this is mine" or a "this automatic action depends on that event occurring".

7.6 Implementation

When the user walks into a zone the labels on her Active Bat buttons signifying the particular functionality appear. They disappear when the user leaves the zone. These are hot buttons as described in Section 6.3. Inside the zone the user has the ability to switch through her Desktops. As previously, this is accomplished by the user pressing the upper button on the Bat. In the augmented application a menu appears overlaid next to the Bat with each item indicating a Desktop by name. As the user presses the upper Active Bat button Desktops switch through on the computer as before, but now she sees a red outline on the menu jumping from item to item. The current Desktop on the computer and the current menu item always match. The augmented Bat with the menu is shown in Figure 7.4. It shows the Active Bat when it is inside a teleport zone. The menu of Desktops is controlled by the Active Bat button marked by the overlay "Desktop>>". The menu discussed in Section 6.4 was chosen for the visualisation for the Desktops available, giving their names and indicating the current one with a red outline.

Even though the initial application programmers had "cycling" in mind our interface suggests the more familiar "scrolling". What is meant by cycling is that, each time users pressed the Active Bat button another Desktop would appear on the screen. When the last of the users' Desktops was reached in this process, the next button press would run through the Desktops again. In the new application this procedure was visualised as scrolling down a menu. The selection jumps to the top once it scrolls beyond the last item.

Teleportable machines (computers) were designed to have a green outline overlaid around their actual monitor. Using the lower Active Bat button, labelled in green the user can cycle through the machines, a bright green outline jumping from monitor to monitor indicating the current machine (omitted in final implementation).

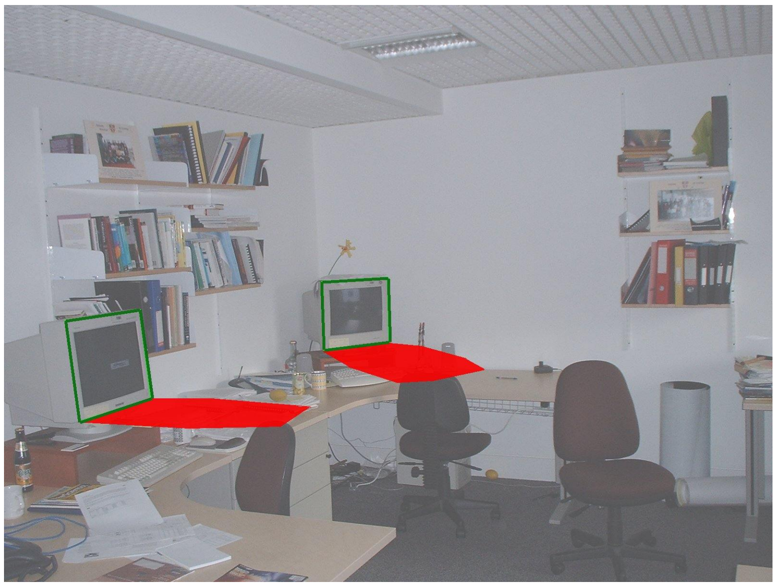


Figure 7.5: Users see the teleport zones through their glasses. Ideal depth perception needs stereo images (simulated picture). The AR HMD is see-through, so the real-life scene is not present in the computer-generated image. Therefore, the composite must be simulated in order to show here.

Teleport zones from the database were visualised in space as described in Section 6.1. Figure 7.5 shows the shapes of these zones around computers. Users can now see where to place or where not to place their Active Bat in space in order to achieve what they want. It was found that depth perception needs to be supported when visualising zones in “thin air”. Therefore, stereoscopic see-through glasses are recommended for such a visualisation. Very simple cues such as an outline around a monitor were enough in order to relate zones and computers. When a Active Bat enters such a zone it affects changes in Models which in turn change Views.

The simulation environment described in Section 5.7 was used for prototyping. The visualisations were designed and view changes were simulated on a Desktop computer first. Key strokes were used to simulate bat movement in our case but the simulation environment can also receive events from the middleware directly.

7.7 Observations

As expected, two circumstances about the software architecture were found to make it very easy to port code from the simulation environment to the AR system. Firstly, the fact that a MVC architecture was used meant that all the Model code remained unchanged. The Controller as recipient of Bat events was virtually left unchanged as well. Secondly, since a scene graph based rendering API was used, the change from simulation environment to actual AR only meant attaching nodes in the right place in a scene graph.

This also shows that the framework is *not necessarily dependent on HMDs.* The architecture can support any type of View, be it HMD, a 3D birds-eye view in our simulation environment, or even visualisations on PDAs. In each case a Model is sending update events to a View that then updates itself accordingly.

When designing our UI we resorted to using states. The Active Bat can be in a finite number of states depending on where in the physical world it is located. As stated in Section 6.4.5 modes have been criticised [102] in traditional GUIs. For example, the Active Bat visualisation displays different options under different circumstances.

We can identify two different kinds of modes here. Firstly, context is intrinsically a mode. The assumption of a context-aware application is that the user does not want to access options that relate to another context. This has advantages and disadvantages. We would not be developing context-aware applications if we thought there were more disadvantages. The second type of mode is a user setting, in this case whether teleporting is on or off. Personalisation implies these kinds of modes. And again, being able to personalise your environment has advantages as well. It is, however, important that users can keep an overview over these modes. The Macintosh Human-Interface Guidelines give an overview of when modes are most acceptable [110]. Basically, you need to make sure that the user does not confuse one mode with another.

On the architectural side it was found that it is better to keep Models free from context states, i.e. they should not be aware of context. The reason for that is that the same contexts occur across all Models and hence can be abstracted to a higher level. A detailed description of this issue can be found in Section 5.5.3. On the other hand, the use of states in Models, makes rendering more efficient since only a small number of different View states can exist. In our case the View of the Active Bat can only have two states, one when it is inside a teleport zone and one when it is outside. When the Active Bat changes its state, the View just switches persistent nodes in the scene graph.

A fundamental difference to traditional GUIs is that whereas in a GUI environment we have a dedicated screen space for each application but a shared input device, the situation is reversed in our case. Input events that lead to changes of Models can take place anywhere. For example, it is possible to change the number of Desktops from any terminal in the lab. This will then be reflected in the visualisation. When there are many agents accessing the same Model some synchronisation or awareness may be necessary in order to avoid user confusion. Our situation is similar to Computer Supported Cooperative Work in this respect. The shared space for all applications implies we need to be able to distinguish which application is managing which zone, for example. So far, we have only implemented one application, but when many applications are running simultaneously in overlapping regions users need to be able to control which application is "in focus". Ideally the user shall be able to see how many and which applications are running and yet be able to focus on one. In addition to usual methods for visual distinction, such as colour encoding, the use of inter-application modes is helpful. The user could select fundamental Active Bat modes in a top-level menu on the Active Bat indicating which application shall be receiving input events (this was already suggested in Section 6.4.5). The fact that our system is modular and event-based facilitates rerouting events to specific applications. In addition to that, by changing modes the user can change the view of his world according to the application she is using or checking on. Each application has its own visualisations. In implementing this we face similar problems to traditional GUIs, such as changing the application in focus or getting out of a particular mode. See Section 5.9 for a discussion on the issue of multiple applications.

When talking about GUIs the question that inevitably arises is that of metaphors. Does Ubicomp need metaphors such as the "Desktop" in order to simplify interaction? The initial answer would be negative. A metaphor, i.e. a familiar concept from the real world that represents a computer-oriented one, should not be needed since the user is operating directly on the real world. Even though there is research direction that tries to break completely with the desktop metaphor, such as Norman's [55] there are examples where introducing metaphor has been successful in Ubicomp interaction. Examples are the SPIRIT Buttons (cf. Figure 7.1) or "Pick-and-Drop" facilities used in Cooltown [54]. The main disadvantage of a metaphor is that the user needs to have some previous knowledge. We touched on this topic when we discussed *conventions*. A glance at Figure 7.2 will show that the information that is required a priori will

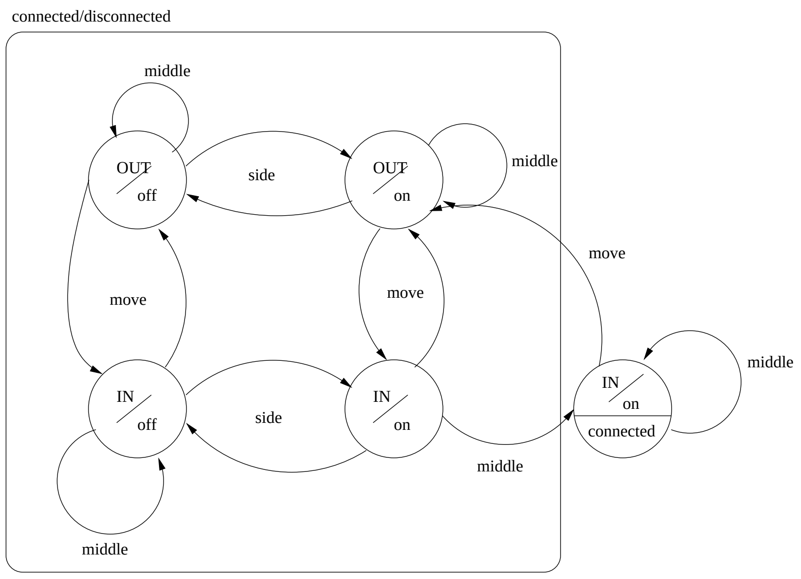


Figure 7.6: Interaction state diagram for our application.

grow if they are not used judiciously. Conceptual models, on the other hand, will show the user what to do at first glance.

As a last step the design was changed to use an actual Active Bat button to turn teleporting on and off rather than a SPIRIT Button. We chose the Bat button because unlike the original application designers we had the ability to enforce the conceptual model by using "hot" Bat buttons with dynamic labels, as known from mobile phones. Controlling teleporting with a SPIRIT Button was deemed unnecessary and its awkwardness for applications that are not restricted to a specific area has been mentioned before. In a nutshell: SPIRIT Buttons have a location, phone call forwarding or teleporting application settings don't.

7.8 Controller Design

Figure 7.6 shows an interaction state diagram for our teleporting application. States are characterised by

- whether teleporting is on or off,
- whether the user inside or outside of a teleporting zone, and
- whether your VNC Desktop is connected to a computer.

The teleporting state is shown in each state in a lowercase "on" or "off". The location context is shown as a capital "OUT" for outside the teleporting zone and a capital "IN" for inside the teleporting zone. Four states are grouped together under the heading "connected/disconnected". This was done for purposes of clarity. Strictly, we are dealing with 8 states. After all, each of the three state variables mentioned above can take two states independently, which gives $2 \times 2 \times 2$ states. What the grouping in the Figure means is following: the states that have been grouped together will have either one state or the other (connected or disconnected) *as a group.* This is determined by whether the rightmost state has been visited. This is the state where the state

	In Teleport Zone			**Outside Teleport Zone**		
Controller Function	*Bat Controller*	*Desktops Controller*	*Teleport Service Controller*	*Bat Controller*	*Desktops Controller*	*Teleport Service Controller*
middleBatButton Pressed()	-	if teleporting is active, connect next Desktop	-	-	-	-
sideBatButton Pressed()	-	-	toggle Teleporting state	-	-	toggle Teleporting state
batMoved(x,y,z)	update Active Bat position	-	-	update Active Bat position	-	-
activate()	-	If teleporting is not active, deactivate View; If active and connected, maximise View; If active and not connected, minimise View;	-	-	-	-
deactivate()	-	deactivate View	-	-	-	-

Figure 7.7: Controller design

variable describing the connection of Desktops to a computer changes. Upon leaving this state, the group of four will have this particular state variable set as "connected".

Other than that, we can see that there are *three state transitions* possible in each state: move, side and middle. Here "move" refers to moving in order to change your location context (inside or outside of teleporting zone). The two descriptions "side" and "middle" refer to pressing the upper Active Bat button (also called middle button) and the lower Active Bat button (also called side button). **Please note** that in this implementation the lower Active Bat button (side button) toggles the teleporting state.

What we see from this diagram is that its complexity poses a challenge for both users and developers. We can gain two insights:

1. Context, modes, and interaction are orthogonal. Each interaction can take place in any context under any mode.

2. In order for the user to understand how this interaction works we should try to provide a different view of the user interface for each state.

Translating this state diagram and the requirement to visualise each state into a group of Controllers is remarkably simple using our architecture. Imagine for a minute how you would translate this state diagram into procedural code without making use of our architecture.

Figure 7.7 shows how Controllers are designed. Our architecture presented in Section 5.5.3 allows us to specify one Controller per Model per Context. The table shows Controllers at the top, to the left Controllers for inside the teleport zone, to the right Controllers for outside. We can now go through the table and fill out what needs to be done for each state transition. Previously we identified three actions possible in each state: move, side and middle. These map to Controller functions `activate()`, `sideBatButtonPressed()` and `middleBatButtonPressed()`. The function `batMoved(x,y,z)` is not a state-changing function. It is merely used in order to update the position of the Active Bat for the AR overlay. Functions `activate()` and `deactivate()` are called whenever the context that holds the particular Controller is entered or left. They are inverse, meaning that activating one context deactivates the other.

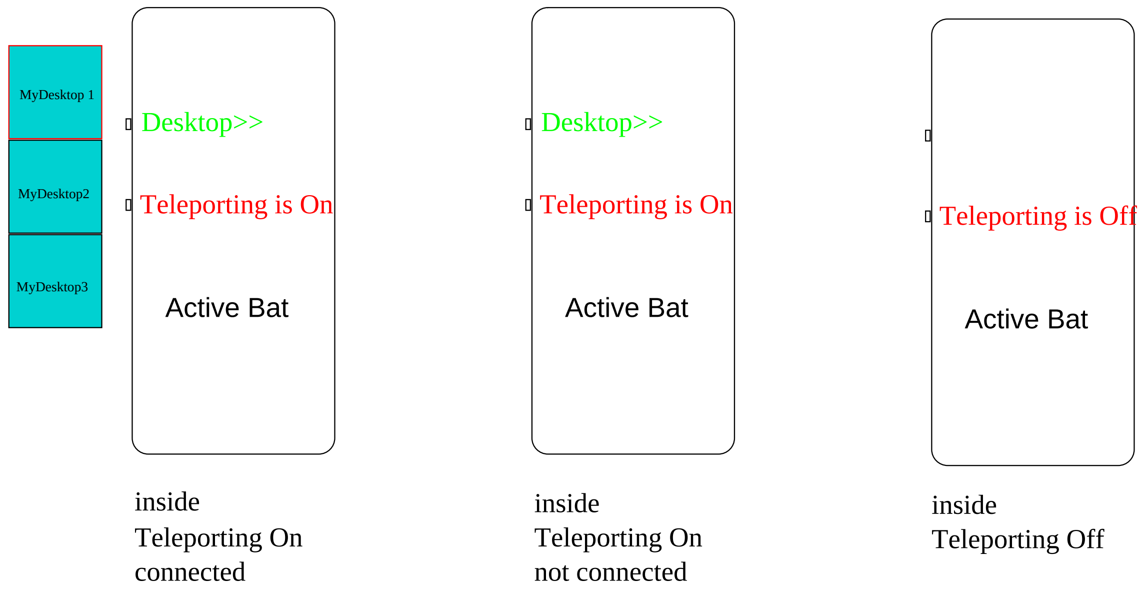


Figure 7.8: Some 'Desktops' Views. The View signals to the user the state of her interaction.

We see that in each context no more than one Controller can interpret a specific Active Bat button press. Furthermore, we see that Controllers also control Views. This is necessary in order to inform users about relevant state transitions. In a particular example (Desktops Controller in teleport zone) we have chosen to show the user whether teleporting is active and her VNC Desktop is connected by changing the View of the Desktops when the user enters the teleporting zone. Remember that that the View of the Desktops consists of a menu and a label named "Desktop>>". When this context is activated the View is changed according to the state variables. Minimising the View of the Desktops means we only show the label, maximising means we show both the label and the menu and deactivating means we do not display this particular View. Figure 7.8 shows Views for a few states.

7.9 Interaction Architecture

Figure 7.9 shows a collaboration diagram of our location-aware application. On the top left we see three Detectors. These are components that listen to particular events from the backend and forward it to the application. The object `aTeleportMovementDetector` receives events of whether the user is inside a teleporting zone. The other Detectors forward events whenever an Active Bat moves or an Active Bat button is pressed. Then, below them we see two Context objects. This application only knows two contexts: inside the teleporting zone or outside. Each context has Controllers attached to it. Every Controller has a link to a Model and a View. Only the interaction relating to Controllers *inside* the teleporting zone is fully shown for clarity.

We can see three Models. Each of these Models can be accessed by its corresponding Controller in each context, even though only the interaction inside the teleporting zone is shown. The Controllers will change their corresponding Model according to the events they receive, if they are active at all. After every change the Model will inform its View and request it to update itself. The Controllers also have a link to the Views. This is because Views might need to be changed according to a context. Remember that we tried to keep context out of the Models, so in such a case the Model could not ask the View to update.

The descriptions on the diagrams show what happens in five different scenarios A through E.

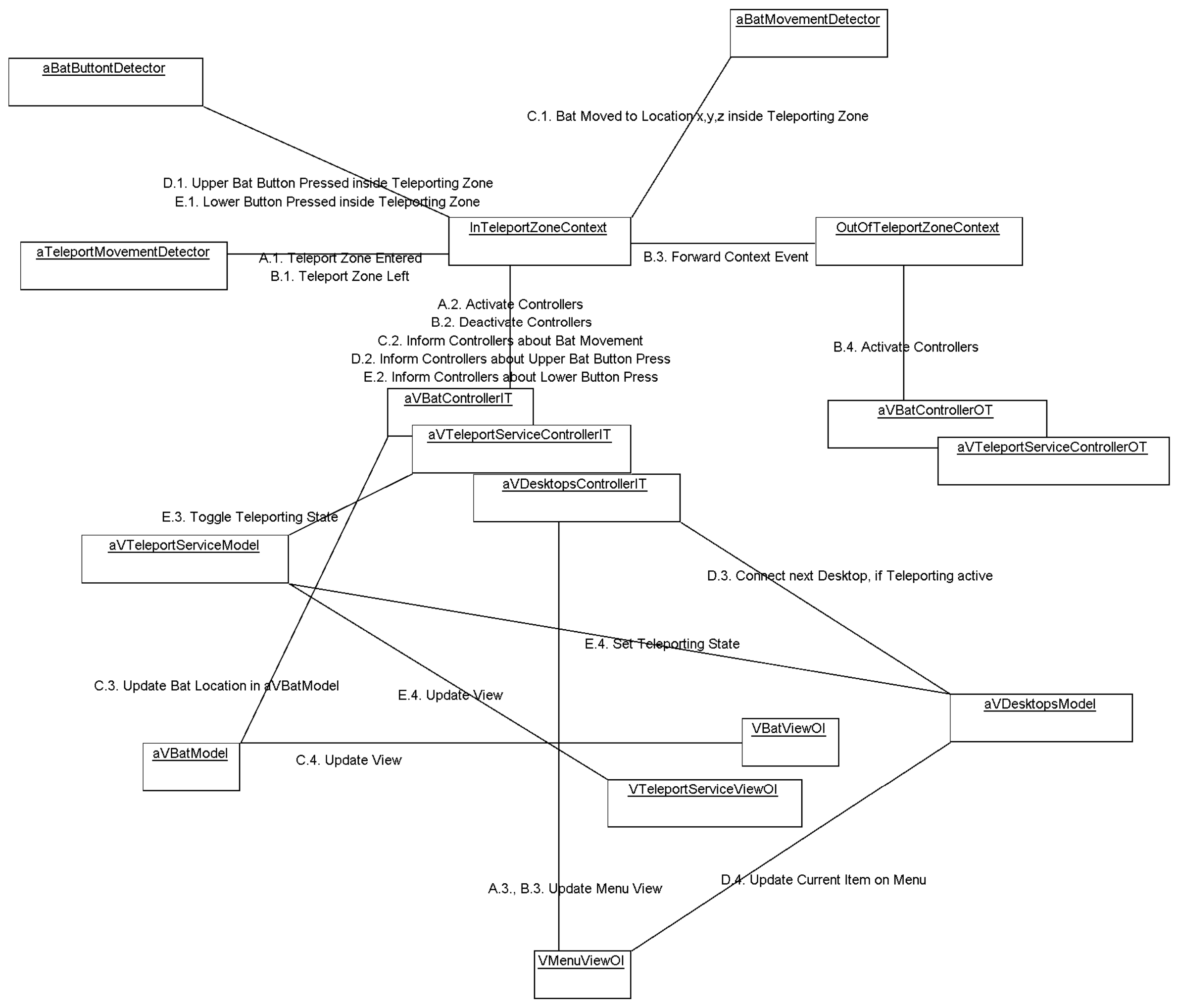

aBatMovementDetector
aBatButtontDetector
C.1. Bat Moved to Location x,y,z inside Teleporting Zone
D.1. Upper Bat Button Pressed inside Teleporting Zone
E.1. Lower Button Pressed inside Teleporting Zone
InTeleportZoneContext
OutOfTeleportZoneContext
aTeleportMovementDetector
A.1. Teleport Zone Entered
B.1. Teleport Zone Left
B.3. Forward Context Event
A.2. Activate Controllers
B.2. Deactivate Controllers
C.2. Inform Controllers about Bat Movement
D.2. Inform Controllers about Upper Bat Button Press
E.2. Inform Controllers about Lower Button Press
B.4. Activate Controllers
aVBatControllerIT
aVTeleportServiceControllerIT
aVBatControllerOT
aVTeleportServiceControllerOT
aVDesktopsControllerIT
E.3. Toggle Teleporting State
aVTeleportServiceModel
D.3. Connect next Desktop, if Teleporting active
E.4. Set Teleporting State
C.3. Update Bat Location in aVBatModel
E.4. Update View
aVDesktopsModel
VBatViewOI
aVBatModel
C.4. Update View
VTeleportServiceViewOI
A.3., B.3. Update Menu View
D.4. Update Current Item on Menu
VMenuViewOI

Figure 7.9: Interaction in a location-aware application.

Scenario A is the scenario of entering of a teleporting zone. The right Context and Controllers are activated and a View is changed accordingly. Scenario B describes what happens when a zone is left. One Context deactivates itself and its Controllers and forwards the context event to its sibling for evaluation, which then activates itself. Scenario C shows what interaction takes place if the Active Bat is moved inside the zone. Scenarios D and E relate to a button press inside the teleport zone. Each will result in only one Controller interpreting this event and affecting exactly one Model which in turn will result in an update of the corresponding View. Object linkages are shown through lines. *Please follow through the linkages according to the number sequence.*

What is left is to look at how the framework communicates with the backend. The SPIRIT backend consists of a Spatial Indexing Proxy that emits spatial events after registration. It also updates the world model held in the database as objects in the environment move. One part of the backend are the services. This is the pervasive infrastructure, a frontend to access any device or active object in the environment via an interface. This application needs to access the Teleport Service. This service communicates with VNC [107] clients running on desktop computers in order to switch their Desktops.

It is fair to say that the main drawback of this architecture was found to be an increase in effort to debug. As can be seen from the diagram even small applications can have complex architectures with many objects holding references to other objects and shifting responsibility. Thankfully, many developers are accustomed to developing using the MVC paradigm, so this becomes less of a problem.

7.10 Conclusion

Chapter 6 introduced a number of interaction prototypes. The aim of this chapter was to put them into practice by enhancing an existing location-aware application.

A methodological approach was adopted, starting the design by identifying basic information objects with a view to modelling all the information involved in our application. Another analysis revealed all the information required by the user in order to complete basic tasks. By fusing the results of both analyses and applying traditional visual design principles we were able to produce complete and efficient visualisations.

We saw that a typical interaction designer will need to deal with a number of contexts, modes, settings and other application modes. This can quickly lead to an explosion of the state space and poses a challenge for both the designer and user. However, by making proper use of the architecture presented in Chapter 6 we were able to manage this complexity in the user interface easily.

The resulting application gives users a better understanding of the invisible computer: it shows what it can do (activation areas), what it will do (function of buttons) and whether it is actually working.

This chapter concludes the experimental part. We shall now test our hypothesis that we can improve user understanding by augmenting location-aware applications with users.

Chapter 8

User Evaluation

In order to evaluate whether our goals had been achieved user trials were conducted. The test application used for this was the teleporting application described in Chapter 7.

Remember that such a location-aware teleporting application has already been deployed in the test environment for years, albeit without an AR interface. As described, the application allows users to "teleport" their Desktops to the machine they are standing close to. Users who want to use this application need to start up Desktops on some computer in the lab beforehand. This needs to be done using VNC (Virtual Network Computing) [107]. It is similar to starting a new X Server in Linux. Once this is done users can walkup to any computer on the network that has a VNC client installed and connect to any of their Desktops.

Location-awareness comes into play when users are enabled to use their Active Bat button in order to send the "connect" command to the middleware, which keeps track of the Desktops each user is running (Figure 6.1 shows how an Active Bat looks like). It then connects one of the user's Desktops to the machine that it determines of being in the user's field of view. The system uses a *zone* that has been defined around each computer and evaluates if a Active Bat button has been pressed inside this zone. Please refer to Figure 7.1 in order to see the shape of the zone around a computer. It is supposed to encompass possible positions a user looking at the screen could be at. Another requirement is that the machine is running a VNC client that is listening for "connect" commands from the middleware.

The test application is a derivation of this existing teleporting application. One feature not included in the test application was the ability to switch between machines that are in a visible zone. This feature is generally not used in the lab anyway. Instead you can now control the state of the teleporting service using the second Active Bat button. The AR visualisation shows the teleporting zones around the computer in which teleporting can be initiated using the Active Bat button labelled "Desktop>>" (an AR overlay!). The second button has an overlay label that reads "Teleporting is on" or "off". Pressing it will change the label from one state to the other.

The Active Bat buttons are used as "hot" buttons, i.e. the labels change according to context. In this case the label disappears if an Active Bat button function is not available to the user. Pressing the "Desktop>>" button inside the teleporting zone will display the menu as seen in Figure 7.4 with each item representing a Desktop denoted by its name. At the same time a Desktop will be teleported to the machine. The button label disappears as the user moves outside the zone, signifying that it is not possible to teleport outside the zone.

In order to compare the effect of the Augmented Reality interface the same application was tested on the same test subjects once with and once without the Augmented Reality interface.

More specifically the aims of the user trials were:

- to evaluate whether the users' understanding had improved with the Augmented Reality interface

- to find out how users' mental models about this location-aware application looked like
- to evaluate how the user experience changed with the Augmented Reality interface.

Mental Model theory [111] assumes that humans form internal representations of things and circumstances they encounter in everyday life in order to explain how they work. One important aspect is that these representations are "runnable" in the head, i.e. they can be used in order to predict the result of a particular interaction with the world. They are not always accurate, which is why humans can have misconceptions about the effects of their interaction. Nevertheless, a mental model can be updated to a more accurate one when a situation occurs where a misconception becomes obvious to the human.

8.1 Process

The number of test subjects was chosen to be ten.

Five of the test subjects can *roughly* be regarded as novices and five as experts depending on how often they use the existing or similar applications. All subjects were required to have at least some knowledge of location-aware applications. There are simple reasons for this. Not only do we expect our ultimate target users to fulfil this requirement but complete unfamiliarity would most probably distort the results. After all, when testing traditional desktop applications evaluators do not test users who have not used a GUI before. As a result of this all users have some technical background, since lay people do not use location-aware applications.

A particular challenge is posed by the "experts". Some of these are actually involved in developing location-aware applications. It would be interesting to find out if the Augmented Reality interface can increase their understanding of the application. In fact, some very interesting observations were made regarding these test subjects.

The experiment consisted of 5 parts. Each of the two experimental parts were preceded and followed by an interview part. The first experiment was a test involving the non-augmented teleporting application, the second one the augmented teleporting application. The guide questions used by the evaluator are shown in Appendix A.

The questions try to evaluate the cognitive load associated with both versions of the application, the mental model users have about it and how well users are able to predict what a particular actions will result in. The interview was conducted in a more flexible manner with the evaluator shortening or lengthening particular parts until he was satisfied to have an understanding for the test subject's abilities.

8.2 Results

Results have been subdivided into nine categories:

8.2.1 Overall Understanding

It was found that the overall understanding of the application was much better during and even after the use of the visualisation as compared to the non-visual case. From the answers it was concluded that the understanding of the application for eight test subjects had increased significantly. For example, the visualisation was found to make things "much clearer", "very easy to understand" or to "definitely" increase the understanding of the application.

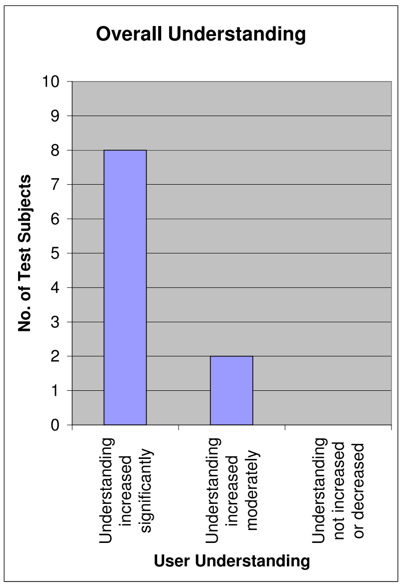


Figure 8.1: The visual interface increases overall understanding.

Two of the test subjects thought their understanding of the application had increased moderately. Both, however, were developers. Interestingly one of them had had a misconception about the shape of the zone around the computer in which teleporting is active. This shape is stored in a database containing the world model. Previous designers have defined this particular shape to be the zone in which a user is most likely to be if she is looking at the monitor. The visualisation showed this particular test subject how the zone actually looked like.

Two "tricky" bits can be identified in this application. Firstly, the fact that teleporting only works if the user is standing in a particular zone and, secondly, the fact that the teleporting state (on/off) influences the function of the first button. Teleporting will not work outside the zone but will only work inside it if teleporting is enabled. On the other hand, turning teleporting on or off will work independently of the location. This make sense since users want to turn teleporting on or off independently of their location.

As the questions show, users were asked to explain how the application works before and after using the visual interface. It was found that in general their second explanation was much deeper and more detailed than the first one, especially with respect to the two non-straightforward concepts.

The answers obtained in the interviews correspond to the observations made while the test subjects were using both versions of the application. Seven test subjects had problems with completing the tasks required when using the non-visual version, whereas nobody had problems with the visual version. The most dramatic increase in understandability was to be observed in novices. Some of them found the tasks too complicated to even attempt when using the non-visual version.

Interestingly, seemingly unrelated issues were understood through the visualisation as well. One of the test subjects (expert) said that she only understood after seeing the visualisation that there is actually a delay between the time the button is pressed and the time the middleware affects a change. This is due to the nature of the sensor system employed. You do not usually expect a delay when you press a button in your everyday life. So, apparently what some users of the Active Bat system assume when they do not see an effect immediately is that the button press did not work at all. If they are impatient (the delay is short for most users, less than

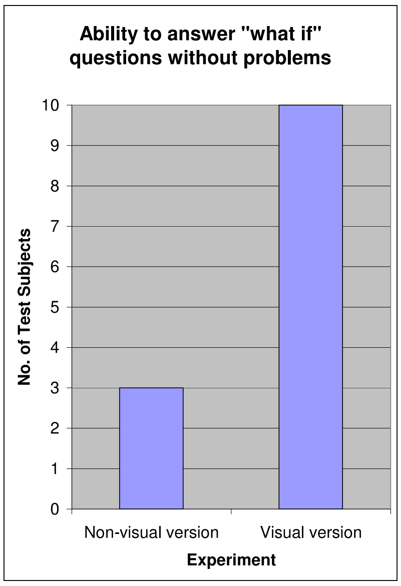


Figure 8.2: All test subjects using the visual version could complete all tasks.

half a second) they will press the button repeatedly. Eventually the initial button press gets through affecting the desired change, but this will be attributed to subsequent button presses. A difficulty arises if the button is toggling a state.

8.2.2 Feedback

One expert user encountered an interesting problem. After mixing up the function of the bottom button and the top button she could not get out of the cycle of pressing the wrong button and revising her mental model about the application and pressing the wrong button again. When trying to teleport she turned it off by pressing the wrong button, and since nothing happened she assumed she had pressed the wrong button. But then the other button would not work because teleporting was off etc. The evaluator had a personal feedback facility where he could monitor the test subjects button presses.

The feedback offered by the visual version prevented any such situation. The use of feedback is not to be underestimated. Especially experienced users appreciated the fact that the Active Bat could give feedback. The only feedback used currently in the lab is audio in the form of beeps of different pitches. One test subject explained that when she hears a beep she has “no idea of what is going on”.

The lack of feedback can be extremely frustrating. A novice user, for example, literally got frustrated and wanted to break off the non-visual experiment because he did not know what the application was doing.

What is interesting is that we are dealing with an application that does provide some kind of feedback on the screen, i.e. when teleporting was successful it will display a Desktop. So, it was tried to find out why users had so little use of this feedback. In fact, no test subject mentioned it as a desirable feature or otherwise. It turns out to be that, that this kind of feedback is only useful as a confirmation for someone who knows how to get to a particular state, but not useful for someone who is not sure about “their way” to that state. One test subject described this as “posterior” feedback, which brings us to the point of predictability.

8.2.3 Predictability

We have seen that being able to predict what is going to happen when interacting with the system is highly desirable. In fact most of the questions during the interviews are targeted at finding out how well the user can predict the effects of her interaction. *What if* questions directly probe the user's mental model since they are forced to "run" it. In this aspect the result of the user trials was overwhelming: The *what if* questions could be answered by all test subjects using the visual interface.

At one point during the user trials a situation occurred in which containment events (i.e. computer-generated events that tell the application that the user's visible field contains a machine) were not being forwarded to the application properly for some reason. The result was that the application was, for a short period of time, not able to detect a user in the teleporting zone. The test subject, even though a novice, correctly predicted that the application would not react to him.

The fact that the visual interface shows you *where* a particular action will have an effect and where not was mentioned again and again as a desirable feature. At any one time the visual interface would only show available functions at a particular location through its labels. Overall, judging from the user echo, this can be regarded as the most helpful feature during interaction.

One test subject said that the fact that she always knows the "current state" and "what you can do in this state" is the best feature of the system. Apparently, this person likes to regard interaction as a state machine, a virtual machine best known for its predictable behaviour.

At this point we should maybe take a look at what function the menu fulfills in this context.

8.2.4 Representations

The impression of always knowing "where" one is in the interaction arises from the fact that the menu has indicators of the current state of action, most importantly a red outline around the current menu item that gradually progresses through the menu as the user shifts through Desktops. Remember that the menu items represent Desktops with the current one having a red outline.

Looking at it this way, the menu has the function of a "map" for the interaction. It can be used to find where you are in the interaction process and where you can go to.

The interviews found that the menu had reduced the cognitive effort associated with interaction amongst 9 of the test subjects. One person said he was not sure. It was described as a "good illustration" reducing the "cognitive effort" required. Test subjects said that the menu made their understanding "definitely" easier or "significantly better". Specifically, test subjects liked the fact that it showed them the "context" of the interaction, i.e. how many desktops there are, "how many clicks" are required to bring up a specific Desktop or what will happen next.

Other representational aspects that were found to be helpful was the shape of the zones and the way the user's location context was represented on the Active Bat. The Active Bat's overlay would configure itself for the particular context (inside or outside teleporting zones) and offer corresponding choices. One test subjects who liked the representation of the zones a lot said the visualisation changed the way she had thought about the application.

8.2.5 Usefulness of Information Provided

Another aspect we could look at is the usefulness of the information provided. We have already elaborated on the interaction-related information, i.e. information that is required in order to show *how* to interact.

In addition we have information that can be described as *task-peripheral* in the classification of Figure 2.1.

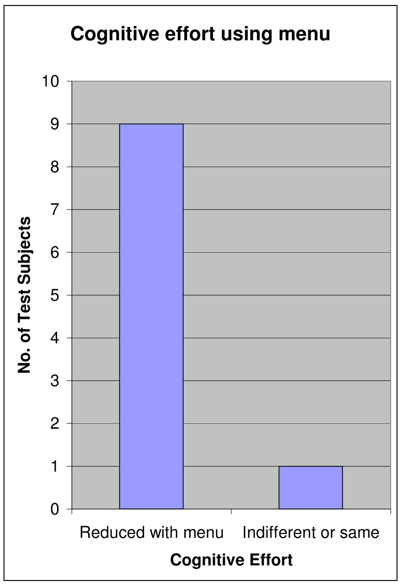


Figure 8.3: The menu reduces cognitive effort amongst test subjects.

Some of the insights useful in this context are as follows:

- Only expert users or developers could remember how many Desktops they have running at the beginning of the experiments. Many users in the lab have Desktops running for months because they forget about them. Since Ubicomp is supposed to support unstructured, often interrupted tasks, offloading memory requirements is desirable.
- Using no visual interface, only expert users can answer questions about where exactly teleporting is active and not even they are always sure.
- The fact that the interface shows when you are inside a teleporting zone seemed to be disproportionately helpful in answering the *what if* questions. It seems that thinking about whether you are at the "right" location "blocked out" thinking about whether teleporting was actually on or off, i.e. visualising "where" something will work, will "free" cognitive resources for other processing. Remember, that in order for the user to evaluate whether a teleport will be successful two conditions need to be fulfilled, the user needs to be in the teleporting zone *and* teleporting needs to be enabled. The fact that this error was so consistently performed struck us as odd.

  After consulting some research on *systematic errors* the most plausible explanation is that what we had witnessed was a working memory overload. According to one theory [112], systematic errors are performed when the working memory goes beyond a threshold, but are not performed at all when it is below that threshold. This is one of the more unexpected results of this user trial. Even though we have been using location-aware applications for years in our lab the load some of them pose on the working memory is not mentioned when users are asked about usability problems.

  Let us for a minute enumerate the items that need to be kept in the user's short term memory for our application: which Active Bat button to use for teleporting, where to stand, whether teleporting is enabled, how to enable it and whether the machine has a listening VNC client running on it; and all of this is just for one application. Looking at it from this perspective it becomes clear how a memory overload could occur.

8.2.6 User's Mental Model

General Observations

The basis for eliciting the mental model users built of the application are the *what if* questions asked, the two explanations of how the application worked and an additional task given to the test subjects. The additional task was described as following:

> Imagine you want to provide a manual of the application for other users. Instead of a description can you draw a diagram for this purpose. Try not to use text if you can.

Some of the insights gained are as follows.

First of all, we can say that the mental model theory is suitable to explain our findings. Users are able to answer questions about what the effects of particular actions are using their mind only. When building these models users make certain assumptions about how things should work. In one case, for example, the test subject thought you need to point the Active Bat at the monitor in order to teleport. This, even though, neither the Active Bat nor the monitor shows any clues that the signal used between the two is directional.

In another case a test subject thought the teleporting zone would exactly coincide with the extent of the desk on which our computers usually stand. In fact every test subject had some kind of idea of where teleporting would be active. Especially, the case of the person who associated the desk extent with the teleporting zone for no logical reason shows that users might need to have some visual idea of where this zone is. By trying to aim for invisibility we leave a gap in the user's mental model that is filled by self-initiative. This insight potentially has grave consequences for all location-aware applications. Basically, if we do not show particular regions or locations signifying different contexts, the gap we as designers leave will be filled eventually, not necessarily correctly.

Interestingly, even developers had wrong ideas about how the zone looked. A lot of the test subjects assumed a small zone around the monitor. The actual zone has the shape of a pear and has a long diameter of about 2 meters and a short one of about one meter.

Another observation that was made is that mental models about the application can vary a lot. For example, one of the test subjects, in his explanation employed no metaphors at all. The drawing produced by him even includes a reference to a variable and a lot of text. So, in general we can say that this is a very non-visual person.

As a contrast another person produced a drawing in which he visualises the on/off button as a light bulb. His drawing is more realistic than any other. Whereas most drawings produced are schematic, this person, for example, included more details in the depiction of the computer than would be necessary for a user in order to recognise it as one. The drawing is shown in Figure 8.4. This by the way was the most accurate "manual".

Another person seems to have a conditional model (see Figure 8.5). His "manual" includes a number of different cases that "work" or do "not work".

One person, a developer, had completely absorbed the idea of a "spatial application", i.e. the location-aware application is like a computer application taking place in the real world and operates humans and things rather than computing objects. Her explanations contained references to "active regions" and other computing concepts. In general, one had the impression as if she was describing the whole application from a bird's eye view, just like programmers do. At one point it became clear that she had incorrectly assumed that the world model used by the SPIRIT system gets updated instantly and automatically. In reality moving a monitor (these are not tracked by the Active Bat system as everyone knows in the lab), for example, does not get updated in the world model unless someone surveys the new position. Considering

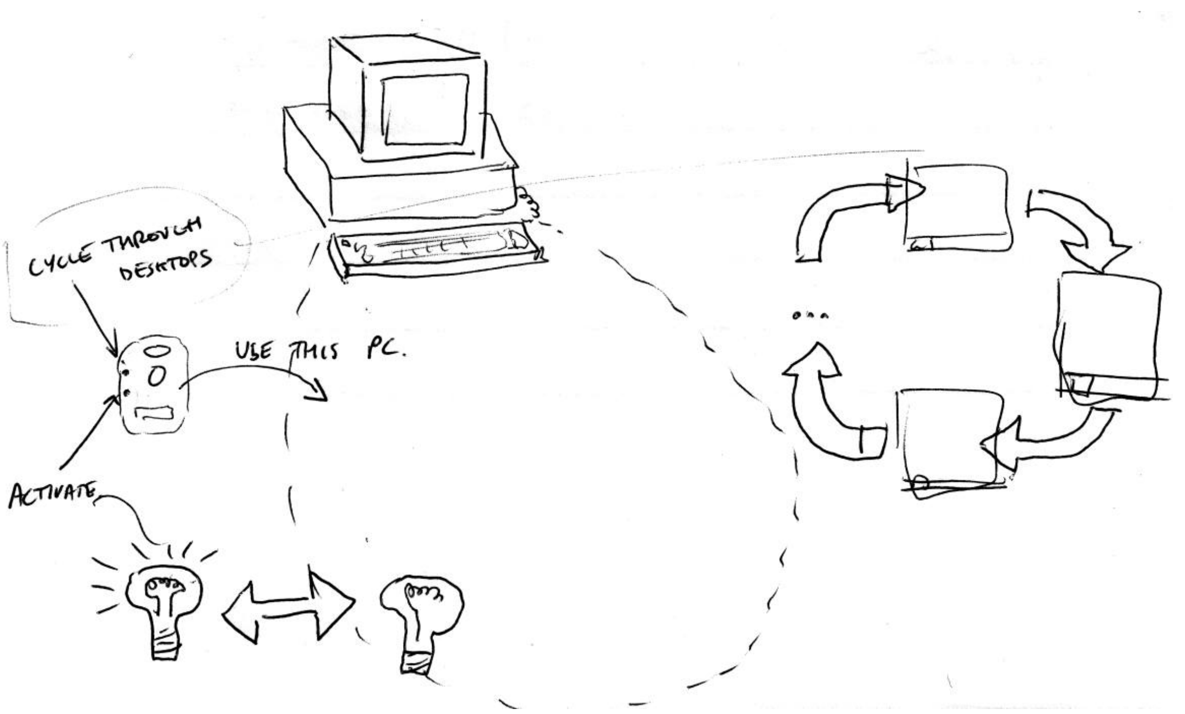


Figure 8.4: Full marks for this diagram. The test subject has understood the application fully. The diagram shows that the test subject might be a visual learner.

her mental model of the application, it becomes clear that she had failed to take into account an aspect that is not analogous to applications running in the computer: human intervention in order to update variables.

Similarly, some had the idea of a state machine as discussed earlier. One test subject drew one for her "manual".

Another test subject had a mental model that was similar to the use of a remote control. He was the only person who regarded teleporting as a property of the Active Bat. When asked whether his teleporting setting was enabled he responded with "you mean on my Bat?" before seeing our visualisation. His diagram shows only the Active Bat with button functions labelled.

We can see that users' mental models show a great variability. This may seem to be a problem for designers who want to design interfaces according to users' mental models. On the other hand, mutable mental models also mean that we as designers can influence them by providing an appropriate user interface. For each mental model we have elicited we can come up with a user interface that fosters it.

A final question that might be of importance is what users associate teleporting with most. The test subjects were asked to decide what teleporting is best described as property of. The majority saw it as a property of space. One person, a middleware programmer, saw it as a property of the "System". Two test subjects saw it as a property of the Active Bat system. One of them is a developer who has written applications for the Active Bat system. The other knows about the Active Bat system but is not sure what it can do. So, for him, the "mythical" Active Bat system seems to be responsible for all these location-aware applications in the lab.

The visualisation as implemented tried to provide the user with a mental model that would make users see teleporting as a property of space. See Section 6.1 for the rationale behind this. We realise that, to what extend we can influence the user's mental model depends on her background.

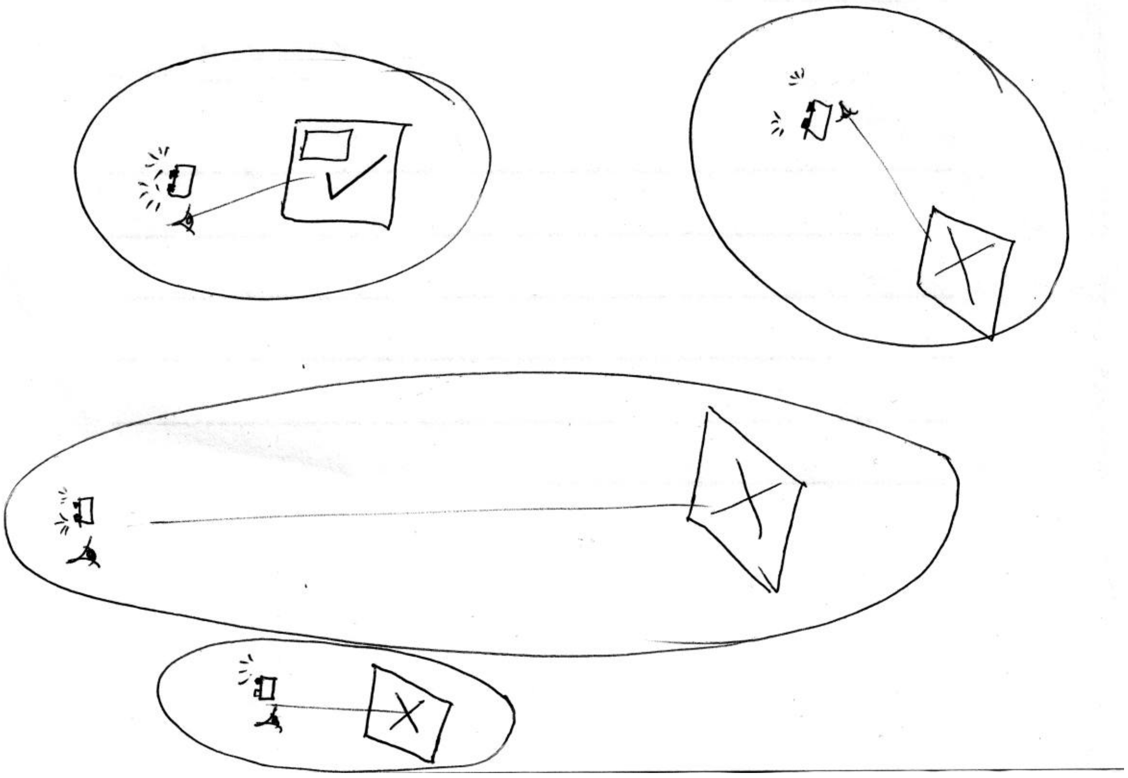


Figure 8.5: The test subject has sketched cases in which teleporting “works” and does “not work”.

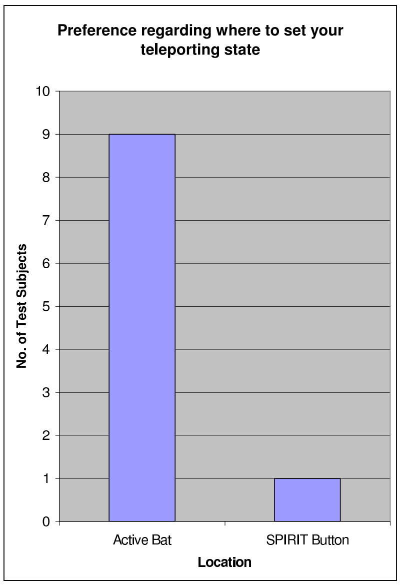


Figure 8.6: Users prefer having their settings available on their Active Bat.

Metaphors

It was observed that all test objects employed metaphors to explain how the application works. One test subject expressed that Desktops were “stored” somewhere in order to be accessed remotely. As the exact inverse another test object talked about “carrying” Desktops around.

One test subject talked about “sharing” Desktops with others, just as you share pictures. One subject employed a number of metaphors from desktop computing such as “clicking”, “moving” or “transfer”. Significantly, she was the one who had the misconception about the need to “point” the Active Bat towards the computer.

We shall elaborate on the point of users making analogies in order to explain new situations in the Chapter 9.

8.2.7 User Preferences

We have previously discussed the problem of setting preferences in location-aware environments (please refer to Section 6.4.2). In this context it was found that users do not like to be interrupted from their task in order to check or set a preference.

9 out of 10 users preferred to be able to set the teleporting preference on their Active Bat rather than using a SPIRIT poster deployed at a particular place in the lab, which would involve visiting that location every time you want to change the setting.

Another facility that has been implemented in our lab due to the limited interface of the Active Bat is a “beep log”. The only feedback Active Bats can give you without an AR interface are beeps. In order to provide different kinds of feedback, beep melodies are employed using different pitches. Most users, however, cannot remember the meaning of each. Therefore, a beep log is provided. It can be used to check what alerts mean a user has received. This log, however, is seldom used. One test subject tried to explain why “no one” used the beep log. “If you don’t know why [your Bat beeped]...who cares”.

8.2.8 User Feedback

At the end of the experiments test subjects were asked what was the most desirable and worst feature of the system. The following gives a list of the most desirable features mentioned:

- Feedback
- Predictability
- "Coolness"
- Explicit showing of location contexts
- Visualisation of the Desktops as a menu

Most of these points have already discussed in the previous sections. There is no clear cut definition for "coolness", but it is the adjective used by several test subjects. We will come back to this point when we look at the overall user experience in the next section.

The most undesirable features can be listed as:

- Calibration
- Bulkiness of hardware
- Slow update rate
- Limited field of view
- Some small modifications mentioned by individuals such as fonts, colours, descriptions and separation of indication from selection

Calibration relates to a short process (10 s) to be performed once by each user per experiment. In this experiment test subjects had adjust the HMD until they would see a virtual cube on a particular location.

The slow update rate is not a property of the head tracker but comes from the Active Bat system (around 2 to 3 Hz). Hence, only location updates of the Active Bat overlay suffered from this problem. The rest of the application was running at 30 Hz, the update rate obtained by the electromagnetic tracker.

The limited field of view is due to the HMD. Since it contains a mini-monitor it uses to generate the virtual images its field of view cannot wrap around the user's head.

One person mentioned he would prefer to be able to indicate a particular Desktop on the menu first and then select it, making the application bring up that particular Desktop right away. At the moment a the only way to get to a particular Desktop on the menu is to scroll through all in between, making them appear one after the other.

As far as the rest of the performance is concerned, zone boundaries are quite accurately identifiable, i.e. the virtual overlay can be related to the actual region in space accurately enough.

It was found that users were able to relate regions to corresponding computers. The simple cue of an outline around monitors proved highly successful for depth perception.

It has to be said that undesirable features were hardly ever mentioned in the interviews and the echo was overwhelmingly positive. It was only after prompting test subjects that these were mentioned. This does not mean that these features are negligible. In fact most of these will become bigger problems when users have to wear HMDs for a longer period than a few minutes. Nevertheless, these are mainly technical problems that can be solved by advances in technology.

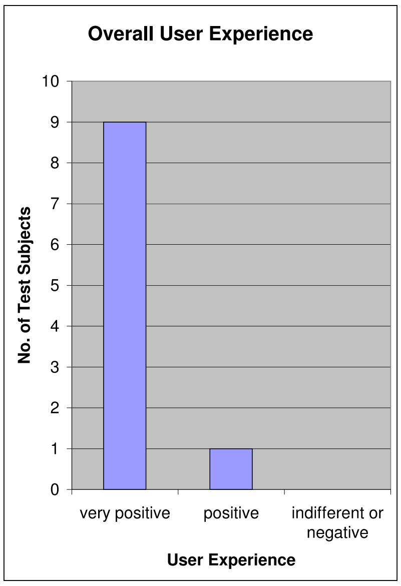


Figure 8.7: Users enjoyed the visual interface.

What would have been more worrying is, if we had received feedback such as “All these visualisations confused me”, or “I don’t see the point of the visualisation”. Or, if we had not observed that people who could not complete tasks in the non-visual experiment improved dramatically completing all tasks using the Augmented Reality interface.

8.2.9 Overall User Experience

Some of the insights gained from the user trials concern the user experience. This is not a full evaluation of how the user experience changes if you are experiencing all location-aware applications through an Augmented Reality interface. Much more experiments with a non-prototypical system would be required for that. Nevertheless, we can obtain hints as to how the user experience will change if AR interfaces become more widely used.

Nine out of ten test subjects made very positive statements in this respect. One test subject went for the more conservative “it increases usability a bit”. The other nine stated that they liked the application “a lot”, “very much” or used similar classifiers.

One of the test subjects said that the Augmented Reality interface lets you know that “the application is not broken”. She was an experienced user of location-aware applications and this seemed to be her biggest problem with location-aware applications in general. The remark actually says more about the user experience users currently have with “invisible” location-aware application than it applies to the visually enhanced version.

Another test subject said he would not rely on the teleporting application and would always have a backup if he planned to use it in order to teleport his Desktop containing presentation slides to the presentation room (a popular use of the application).

One peculiarity we talked about earlier was that a developer had thought the teleporting zone was only a small zone around the monitor. This developer has had programming experience with many location-aware applications and has been using the teleporting application. He did not realise that the teleporting zone was a lot bigger, simply because he only uses the small region in front of the monitor.

What these examples show is a particular attitude to location-aware applications. Apparently, users hardly explore them. They are conservative in the sense that they only use what they know works and even then they are in a constant state of uncertainty as to whether it is performing or not. This is, of course, not the attitude we as designers can allow the users to have. What needs to be done is to actively work on changing this user experience. Rather than adding new applications and functionality we need to spend time thinking how we can give users the feeling that they can rely on, even play with, the applications without breaking them. As described earlier, one test subject was so frustrated with the invisible version of the location-aware application that we broke off the non-visual part of the experiment.

In this context, what was mentioned again and again was a kind of "coolness" factor experienced by users using the Augmented Reality interface to the location-aware application. Possibly, by generally introducing more enjoyable features into location-aware applications we can influence the user experience.

8.3 Conclusion

In this chapter we have performed a user evaluation of an interactive location-aware application. The main insight has been that user understanding increases by adding interactivity to location-aware computing. When drawing conclusions we have made an effort to abstract from this particular application and generalise observations.

The other eight categories identified, do not form exclusive sets but show different aspects of user understanding. The reader will notice a lot of overlap across all evaluation categories and hopefully this will give him or her a good perspective on the whole issue.

It should be noted that our evaluation task was focused on determining the effects of having a visual interface on application understandability. How better user understanding translates into more effective use is an interesting question. This, however, was not part of our test suite as a quick glance on the questions in Appendix A will show.

One question that might be interesting in this context is to what extent are these insights applicable to the field of location-aware computing in general. After all, the application tested has a strong Active Bat/SPIRIT colouring to it. Also, the test subjects differ from ordinary users in that they are living with location-aware applications around them, some are even using them on a daily basis. Consequently, some test subjects have more expertise with location-aware applications than ordinary users.

One could argue that this distorts the results obtained. On the other hand, the environment in our lab is what a future office environment might look like. For user testing this offers unique opportunities.

The question remains whether we can apply our insights to, say, a location-aware electronic guide (such as [24]) application that is being used by an ordinary tourist. The author would like to think that our observations form a basis for user testing suite that specifically targets understandability in any location-aware application.

On this note, we should add that the teleporting application does have more complexity than an ordinary electronic guide application. At the same time these complexities are typical for the applications we have in our lab and possibly what we will encounter in the future. User preferences, location contexts at a smaller scale than rooms, integration of networked services or one-handed interaction are all characteristics we are likely to encounter if the number of location-aware applications in a particular space will increase.

It is anticipated that the full extent of the improvement made to the existing system will show once technology has progressed enough to deploy it widely on a permanent basis. By that time, the number of applications, settings and devices the normal human will be dealing with will have increased so much that requirements with respect to reliability, controllability and user memory will have become unmanageable without such a support system.

Chapter 9

Interpreting Location-Awareness

The previous chapters described a number of interaction prototypes that were implemented and tested in an application. We shall now try to distill the experience gained into some principles that shall advance our understanding of how users perceive smart environments (current chapter) and how to design applications (next chapter) for them.

The basic premise is as follows: Humans make a mental model [113] of their environment and objects in it. A mental model is a description about a “thing”, that exists in the head of the user and can be used to make (deductive) statements about the “thing” without needing to experience the “thing” itself.

9.1 Clues Obtained from Experiments

Before we start developing a model that will help us explain how users might perceive a smart space, let us try gathering a few clues that were obtained in the experiments.

1. **Interaction with the world vs. interaction with the computer**

   When designing the interaction for the teleport application in the Chapter 7 we saw that a typical location-aware application will encompass real and virtual (interaction) objects.

   In our design cases we can distinguish between two types of interaction. In one case the user is performing actions that have a meaning in the real world, e.g. walking into a room. We have also called this kind of action implicit.

   In the other case the user is performing actions that explicitly address a computer, e.g. interacting with a virtual menu. This kind of interaction can be called explicit.

   There are cases, especially in mixed reality environments where it is difficult to say whether a particular action has a meaning outside the computer system or not. Interacting with a virtual handle on a real object is such an example. Holding an Active Bat at a particular location in order to send a command (SPIRIT “Buttons”) to the system is another example.

   For our purposes the main concern is how *conscious* the user is about a computer or about an “interaction” at all. Therefore, one could argue that interaction with virtual objects in mixed reality environments is “interaction with the world”, if the virtual objects are *perceived* as being part of the real world.

2. **Location-aware applications can be provided with a conceptual model**

   Chapter 7 presented an application that originally did not offer a clue as to what it can do, how to use it or what action results in what effect.

Figure 9.1: Future model of a Retinal Display by Microvision (from [114])

After augmenting it visually, users could use it and understand its workings. The visualisation showed users what concepts are involved, such as active regions, Desktops and machines. Menus, labelled buttons and outlines indicating selection gave users the ability to understand the operation of the application.

3. **Augmented Reality can be used in order to implement "Invisible Computing"**

   Using Augmented Reality we have been able to resort to Weiser's original idea that computers should support users in their task actively and yet stay out of their way. Support can be provided anywhere and has made visible the applications previously operating in digital darkness. As the trend towards miniaturisation continues, circuits and optical components in head-mounted displays and trackers will decrease in size. Unobtrusive head-mounted displays that are indistinguishable from ordinary glasses are already appearing on the market with ultrasonic and other forms of tracking getting more and more accurate. In this light, the augmented glasses have the potential to become the ultimate invisible interface between human and computer.

9.2 Perceptual Approaches to Cognition

Interestingly, Weiser, in one of his presentations [95] contrasts interacting with the world with interacting with an agent, making clear the former is preferable. One of the points raised is that interaction with something other than just the real world would draw attention to the computer. This is indeed true. Interacting with virtual menus, for example, breaks the illusion of no computer being there. So there is a tradeoff between controllability and keeping this illusion. True invisibility might not be achieved, but by making an optimal tradeoff we might get close.

Norman's theory of affordances [7] originates from Gibson [94]. Gibson understands affordances as basic relationships between animals and parts of their environment. Norman tried to use this concept by proposing to build affordances into technology and appealing to the same basic senses in order to achieve usability. These affordances, since they are not universal (not every human can interpret a knob, for example) were renamed *perceived affordances* [55]. A *perceived affordance* is an advertisement for an affordance. A region visualised on the HMD could be called a perceived affordance indicating something will happen if the user enters it. It is not a real affordance since users can enter other areas as well. Real affordances of a region could be: the ability to have a party, a meeting, surf the web with a WLAN card etc.

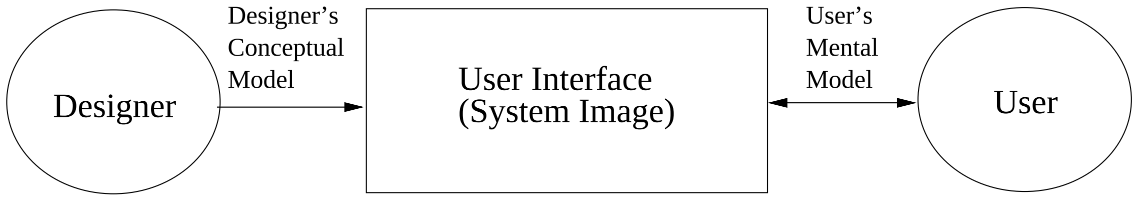


Figure 9.2: The model shows how users and designers relate to artifacts.

While it is possible to implement *perceived affordances* of a physical space using Augmented Reality, it is not possible to implement constraints. We would need a mechanism to prevent the user mechanically from performing certain movements. Likewise, providing good mappings for implicit interaction is impossible. As a reminder, a good mapping in a device ensures that the physical relationship (this theory was mainly conceived for physical artifacts) between user input and effect is intuitive. Please also see Section 1.1.1.

Let us say the region has a particular colour for a particular function (say, when entering a blue region your online status will change to "busy"), then the visualised region on the HMD neither affords this, nor does it advertise it. This is just called symbolic communication. Note that in this context, we are looking at the signal the blueness of the region is sending.

Norman's theory goes beyond affordances. It describes a model of how designers and users can relate to an artifact. The model [115] is shown in Figure 9.2.

It consists of following parts: the conceptual model provided by the designer on the surface of the artifact, the system image the artifact presents and finally the user whose mental model is created by interaction with the artifact.

The aim of well designed artifacts is to reduce the cognitive effort required to use them [13]. In fact, well-designed artifacts can benefit humans by reducing the cognitive load they would encounter without their use.

Weiser's original idea of invisibility can be formulated as following: Even though computers continuously support humans with their everyday tasks they are only noticed peripherally, similar to the way humans notice, say, physical obstacles (such as furniture) in the environment. People dwell with computers and traditional boundaries between user and computer start to *disappear.*

As humans move from task to task, computers need to follow. Location-awareness can be used in order to make these transitions unnoticeable for the human.

So far, the problem has been that location-aware applications that "inhabit" the same space as humans with the aforementioned aim, had become noticeable on a higher cognitive level (also called *reflective mode* [13]), because they lacked perceptual properties that would appeal to the less conscious *experiential* [13] mode.

The approach proposed in this work is to build affordances into these applications with the ultimate aim that users would *run into an undesired location-aware application no more than they would "try to walk through a wall"* (refer back to Section 6.1.3 for more on this idea).

Another observation we made concerning Augmented Reality's ability to create such an ecosystem of intertwined interaction and symbiosis was that we were able to let the user offload cognitive effort to the environment, specifically in the case of remembering where quiet zones are located. By allowing users to customise their space it might be possible to achieve the same kind of gain in productivity humans achieve when they organise their physical environment [116]. Since this relies on perception, Augmented Reality would need to be a vital part in order for this customisation to be effective. Norman calls cognition that is facilitated by the environment of a human "Distributed Cognition" [13]. The theory's basic premise is that humans use their environment as a whole, including seemingly unrelated items, in order to perform processing for a specific task. In this sense it is very similar to Weiser's vision.

We touched upon an important point in Section 5.9 regarding how many users will use the same environment. From a "Distributed Cognition" perspective users could gain additional benefit from the environment if every user would see location-aware applications at the same location in space, even though it is technically possible to let every user have her own applications. In this way users could learn from each other.

9.3 Further Observations

In addition to the ones described above a number of further observations were made.

1. **The "conceptual model" methodology reaches its limits in Ubicomp environments**

   We have learnt to distinguish between "interaction with the world" and "interaction with the computer". When it comes to interaction with purely virtual objects design methods that provide a good mental model are well-known from GUI applications. Designing the menu involved showing virtual affordances, using constraints (e.g. only one way to get from one menu item to another anywhere in the menu tree) and mappings (e.g. top button scrolls up, rather than down).

   When it comes to what we have called *interaction with the world* in Ubicomp we encounter difficulties in providing a conceptual model as defined by Norman [7]. The first difficulty is the "where". Due to the distributed and often invisible nature of Ubicomp it is not always clear where to show affordances. If the application is not confined to a small region with all the objects it influences in the user's sight the affordance approach is not feasible. Otherwise, the user would not know a priori where to find the affordances or that they exist at all. The previous chapters showed how we handled this problem: If we were not able to pin down the desired interaction to a particular object or location we resorted to providing it as an *interaction with the computer*, i.e. controlling the action via a menu.

   Another problem related to this is the *identifiability* of the "device", "artifact" or "system" in Ubicomp environments. The solution proposed in the design cases was to assume every user carries a "personal" Active Bat that can be used in order to address the system.

   The third difficulty experienced was the inability to make use of good mappings and constraints. We talked about this problem in the previous section.

2. **Recognition, recall and association can be cued in interaction**

   When designing the interaction in the last chapter we resorted to a number of concepts that users know from other domains, such as desktop computing.

   The virtual button in space that can be "clicked" (by bringing the Active Bat close and pressing a Bat button) is a very successful example of a known interaction procedure users were able to relate to. When a menu and "hot buttons" known from mobile phones were used, this was done with the assumption in mind that users would immediately understand their usage, which they did. Also, one of the reasons an Active Bat was chosen to be the "first point of contact" (please refer to Section 6.4.3) in order to change system settings was because it is easier to associate one point with the "system". Similarly, the concept of an "application" is easily understood because it is known to the user from desktop computing and recognised.

3. **Visualising concepts the computer uses can be beneficial**

   One of the useful visualisations was the interaction region of an Active Bat. Before users had to guess where to place their Active Bats in order to initiate interactions. Now this was immediately visible.

   The concept of a user standing in front of a machine was implemented within the application as an overlap of the user's Active Bat with a region defined to be "in front of" the machine. If this overlap occurred the application could assume the user was standing in front of the machine. The problem occurs when the user believes she is standing in front of the machine, but still is outside the region defined within the application.

   Only knowledge of how the computer evaluates this relationship can prevent such a situation. Hence, presenting this concept to the user is helpful.

9.4 Analysis

Let us now analyse these findings in the light of cognition theory introduced so far.

The "conceptual model methodology" (this is a design approach, not a theory about mental models) introduced by Norman [7] has been successful for designing physical artifacts. The systems we are considering, however, differ from physical artifacts. One difference is that they cannot be confined. Norman's model as depicted in 9.2 partly draws its power from the idea that the artifact is presented as a black box with strong input output correlations. In Ubicomp we cannot identify such a black box. Consider the following scenario [117]:

> Your alarm clock might ring later than usual if it logs on to find out that you dont have to get the kids ready for school - [it is] snow day!

Or a scenario mentioned in our introduction [15]:

> [...] while I was at University of Toronto, we did some early work on what later became known as the "reactive room". Initially versions of the conference room were known as the "possessed room". Lights turned on and off when you moved or ceased moving. Equipment "spoke to you" through an electronic voice and told you that you had 5 seconds to "take action" or it would power down. Video signals switched to channels and "forgot" how to switch back.

How would we give the devices involved in these actions affordances? We cannot create a black box around all of them, just their physical distribution would not permit this. The problem with Ubicomp environments is that they can neither be regarded as one black box nor as many independent black boxes. If we would perform *all* interaction virtually, say control everything via a desktop we would indeed have our black box which can make us understand dependencies etc. But in the general case we are inside the black box, looking at its internals.

There are nevertheless many cases where we can regard an application or a task confined to a small area. The teleport application was such a case. The problem we ran into there, however, was that we could not provide good mappings or constraints as discussed earlier. Not having good mappings means it becomes difficult to understand cause and effect. The inability to provide constraints implies a loss of control. We solved these problems by resorting to virtual interaction.

Norman, in his theory, distinguishes between *knowledge in the head* and *knowledge in the world* [7]. When it comes to that "magical" moment where the user meets the interface she will try to make sense of it using both. *Knowledge in the world* consists of clues to how to use the device, built in by its designers. *Knowledge in the head* can consist of experience, conventions or culture.

A lot of Norman's theory deals with how to provide *knowledge in the world* and, in fact, our Augmented Reality interface just does that so aptly. At the same time we greatly benefitted from what one might call use of "metaphors", i.e. porting something experienced in another domain and applying it to the one at hand. Some of them have already been mentioned above. For example, the spatial "application" is a metaphor in this context. By making the analogy to objects and events in traditional computing applications one can visualise the interaction of people and real world events in location-aware computing.

The design cases in the previous chapters showed how we made use of conventions and metaphors in our user interface design again and again. The theory of affordances has always had a sceptical stance towards these kind of elements in the user interface ([118], [119]). They are not seen as "natural", since they encompass cultural knowledge. In his examples Gibson [94] talks about "animals" making sense of their environment and so stresses the supposedly purely biological properties of affordances. The reason why this might be important is that the cognitive effort would be reduced if technology could appeal to our basic senses.

The result of all this is that *knowledge in the head* is underrated in perception-based theories of cognition, i.e. theories which regard perception as the driving factor of sense-making.

The human mind has great abilities of making associations and "re-cognising" things. Sense-making itself has been described as the enlargement of small cues [120], i.e. perception can be regarded as the initiator of a non-linear dynamic process in the head that results in recognition (cf. neural networks). This is why users can easily relate to SPIRIT Buttons, they are reminded of the desktop analogue.

We tried to make use of the associative powers of the mind when we proposed to give the user a reference point she can identify as the "system", i.e. a personal Active Bat. By objectifying abstract entities we can help the user build up a network of links. Everything usually associated with the "system" will automatically point towards the personal Bat (a node). If we look at the myriad settings, devices, interaction styles, data formats and inferences we will find that we need to provide some structure to the user in order for her to make sense of this chaos, building on previous concepts. In educational settings this approach is called *scaffolding* [121].

Originally, location-aware computing was proposed in order to make use of a *shared metaphor* or *concept* [80] between user and computer, the assumption being that it was possible for the computer to see at least parts of the world the way the user sees it. Space seems to be a likely candidate for a shared concept, since it is easily describable in Maths. But it was soon seen that the overlap between the computer's and the user's understanding could not be perfect. This was stated when we talked about how the computer stores the concept of a region as opposed to how users see it. Small incongruities can lead to usability problems. The user can, however, deal with these, but only if she receives a feedback of what exactly the computer's view of the world is. In our application users do not just rely on being in front of the computer in order to teleport but can see the region the computer uses as well.

On a more critical note, systematically presenting internals of an application to the user is in contradiction with Norman's black box model.

9.5 Other Approaches

Let us first recap the difficulties encountered in applying a purely perceptual approach to understanding sense-making in Ubicomp:

- Practical considerations in Ubicomp make it difficult to rely on only the "conceptual model" methodology.

- Under some circumstances the black box approach can inhibit sense-making in a Ubicomp environment.
- Knowledge in the head is not promoted as a means to bring structure into the perceived chaos of Ubicomp environments for the user. In this context, we specifically do not mean short term memory but rather experience, conventions or culture. The reliance on short term memory should be kept to a minimum. Please refer to Section 8.2.5 for a more detailed discussion.

Cognitive science is based on theories that are difficult to verify since nobody can know what really goes on in the head when people make sense of things. Each approach stresses some other aspect. Sometimes it can be helpful to look at other approaches.

One approach to explaining how people build mental models is the metaphorical one. We have encountered many metaphors in our design cases and most of them have been described above. We shall just have a look at a few core concepts of this theory that can prove useful in our context. The following is a summary of [115].

The metaphorical approach assumes that *all* knowledge is metaphorically based. A metaphor is porting a concept from one domain to the other. Metaphors can provide a scaffolding for learning, making knowledge (in the head) available for new situations. One can distinguish between global and auxiliary metaphors. Global metaphors, unlike auxiliary metaphors, encompass the entire interface, such as the desktop metaphor. As an aside, for a Ubicomp environment it is probably difficult to find a global metaphor, but auxiliary metaphors have been successful (SPIRIT Buttons, Pick'n'Drop in Cooltown [54] or Infostick [122]). Types of metaphors have been classified as operational (processes, actions, algorithms) and organisational (structures, classes objects, attributes).

Metaphors have properties that are similar in the input and output domains and some that are not. The latter are called mismatches. Metaphors have come under criticism [119] because of these. However, since the metaphorical approach explains everything as application of metaphors, it is not really about weighing out advantages and disadvantages of "using" metaphors. Neal and Carroll [115] provide a more detailed discussion of this.

The above shows that there are approaches in Cognitive Science that stress *knowledge in the head* to an extreme extent. The problems we face in Ubicomp require exploring these kinds of approaches.

An entirely different approach is activity theory. Activity theory is an ecological approach, just as Gibson's and Norman's approaches. It does not separate humans from their environment. Activity is seen at the centre of all understanding. Subjects (humans) and objects use activity as a means to influence each other. An introduction to activity theory, especially relevant to our context can be found in [118]. We shall only look at some important points Bærentsen and Trettvik make.

- **Activity can be decomposed with implications on the subject's consciousness**. One can decompose activity into actions that can again be decomposed into operations. Activities have motives, actions have goals and operations have conditions. Operations can be decomposed into consciousness operation and adaptive operation. Adaptive operations are instinctive, whereas consciousness operations are interiorised cultural-historical behaviour.

  Activities are motivated by personal sense, actions by objective concepts in the head and operations by the sensory fabric of the human.

  It is useful to point out that motives, goals and operational constituents are flexible and dynamic during an activity. One important point to remember is that there are many *levels* of activity that can be supported, not just adaptive (instinctive) operations.

- **Affordances only have a meaning within an activity**. Gibson's original idea was that affordances would send signals to the animal that would automatically perform operations (see previous point). For our environment a more realistic view would be that humans perform different kinds of activities and some of their operations are more or less supported by affordances. There is no one-to-one mapping between affordances and activity .

- **There is no sharp distinction between "natural" and "cultural" objects**. In the teleport application when we used the term affordance we recognised the need to define it first. This is because there is some ambiguity of what can be classified as an affordance and what cannot. The problem is how to distinguish an affordance from a convention. Norman argues that the outline of an icon is a perceived affordance whereas the graphical depiction inside it is symbolic communication [123]. Therefore a printer icon only affords clicking but not printing. It is arguable whether users perceive the icon in this way.

  By loosening this definition and regarding affordances in the context of their use, one can design for specific uses and learnability. The strict definition would assume that affordances are natural and instinctively perceived.

9.6 Applying Schema Theory to Ubicomp

The final approach to Mental Model Theory shall now be introduced: Schema Theory. The introduction to the theory is based on Rumelhart's essay [124].

Schema theory is a theory about how knowledge is organised in the head and how new situations are related to it. Schemata are units in which knowledge is packed. It is easiest explained using an example. When a human sees two people exchanging money against a good, the observer can immediately form a hypothesis of what is going on, what its implications are, what has happened and what might happen. A few clues are enough for the observer to classify what he sees as a buy-sell transaction, identifying purchaser, seller, merchandise.

Each of these are called schemata. They are validated against what is being seen and used to work out further implications etc. The process of recognising other schemata can work bottom-up or top-down. The example above shows a bottom up inference first (money, object, exchange means buy-sell transaction) and top-down afterwards (that means the people are purchaser and seller and the object is a merchandise). When humans learn new concepts they link them with existing ones.

Schemata are used in perceiving objects, understanding discourse, remembering, learning, social cognition and solving problems. In learning we can distinguish three types: Accreation (no new schemata are formed, the new situation is related to an older one and stored), tuning (existing schemata evolve) and restructuring (creation of new schemata by copying old ones and modifying them).

One can probably assume that the cognitive effort involved increases in that order. Schemata are constantly applied by the human. This process draws attention to itself if there is a breakdown in applying an existing schema.

Schemata have been critisised because they seem to model the human to closely to a computer, but in fact schemata are anything else but rigid. They leave much room for variability, one could say they are fuzzy.

There are different definitions of schemata even though all agree on the basic points (i.e. they are prototype structures). Bruning et al. [125] distinguish between schemata and concepts. In that terminology, concepts are used to classify objects (e.g. merchant) and schemata describe how they relate to each other in situations (e.g. process of buying). In the following we will use the term "concept" if we are only referring to "objects" and use "scripts" if we are referring to how they relate to each other. If we want to make statements that refer to both scripts and objects we will use "schemata" as before. The reader is reminded that Rumelhart [124] sees the process of recognising objects and recognising scripts as the same.

By looking at sense-making in Ubicomp from this point of view we might be able to grasp what goes on in the user's head and help her manage the overwhelming nature of Ubicomp.

In our own design cases we have used various schemata albeit without identifying them explicitly. In Chapter 6 we talked about presenting the personal Active Bat as "the first point of contact" for the user. Here we are making use of a schema already in the user's knowledge base: the helpdesk schema.

All metaphors used so far cue an existing schema. When we talked about representing nearness we talked about different *concepts* of user and computer. Location-awareness itself can be seen as a schema. When an informed user hears "location-aware" she knows there is a region, an effector and an application. She knows an effector can either be a person or an active object. The active object could be a device or an everyday object and so on. Using this knowledge she can diagnose breakdowns.

The reader will realise that the way a schema is stored in the user's head is not unique. Depending on the user's experience her schema might have other components involved than the ones mentioned above. The question is to what extent this needs to be considered when designing a user interface for schema recall. The approach we want to take is to make the user think in a way that will make it easier for her. The means we have for that is the user interface. In this sense the user will *learn* and *recall* schemata by interacting with the user interface. From educational settings we know that humans' mental models can evolve over time.

So, what schemata might we be interested in conveying or recalling? A *scenario analysis* can reveal schemata.

Let us see if we can analyse Ubicomp scenarios for this purpose. This one is taken from [126]:

> Tom is at home. He enters the living room sits down at a PC in the corner. He surfs through a selection of MP3's, and adds them to a playlist. He gets up and sits down on the couch. His session follows him to the large wall screen across from the couch. This screen is selected because it is available and in Tom's field of view. Tom picks up a remote control sitting on the coffee table and uses the trackball on it to request the room controls. They appear in a window on the wall screen, showing a small map of the room with the controllable lights. He uses this interface to dim the lights. Tom opens up his playlist and presses play. The music comes out of the room's large speaker system. Sally enters the living room from the sliding doors to the outside and walks over to the PC. She has to manually log in, since she hasn't previously identified herself. She brings up a Word document that is an invitation to a shindig she and Tom are hosting. Wanting Tom's input, she asks him if she can use the large room display. He uses the remote to release control of the wall screen, and she uses the room's controls on her PC to move her session to that display.

What schemata do the users in this scenario need to know in order to fully grasp what is happening?

- "Follow me" (...his session follows him...)
- "Output Device" (screen)

- "Inferred choice" (...selected because it is available and it is in Tom's field of view)
- "Controllable" (...room controls...)
- "One-time authentication"(...manually log in, since she hasn't previously identified herself...)
- "In Control" (...release control...)
- "Movable" (...move her session)

In this manner we can go through various scenarios and identify schemata we could present to the user in our UI. Here are just a few examples obtained from analysing other scenarios in the literature: "Automatic Association", "Resource Appropriation", "Override", "Trigger", "Compatibility", "Profile", "Role", "Personalised", "Proximate information exchange", "Trackable", "Walk-up and Use", "Active", "(Un-)Synchronised", "Inference", "Electronic Token", "Accessible by", "Dependent On" etc.

It is sometimes possible to find hierarchies, such as "Inference" can be decomposed into "Inference based on previous history of this interaction" and "Inference based on preference somewhere else". But it is probably not a good idea to overdo the hierarchy, since we do not want to restrict users too much in how they perceive the world and let them form their own structures from the visualisation we provide.

Icons in GUIs can (re)present schemata as well. Even though it would be possible to create deep hierarchies of Icons, examples where this has been done are few. The Macintosh UI, for example, has very broad categories, indicating whether the icon represents a control panel service by including a slider in the layout of the icon.

Having done the analysis and having identified the schemata used we could now go on to decide how much of this and how it should appear in the user interface. With our means we would want to visualise these schemata somehow. A proposal for the *how* will be given in the next chapter. The designer is still providing a conceptual model to the user, but one that is very different from the one mentioned above.

The approach of using Cognitive Science in order to engineer user interfaces is nevertheless common. Norman's idea was that building affordances into technology can increase their usability. Our approach is that making technology cue schemata will result in higher understandability.

One of the differences is that this is conception-based rather than perception-based. It has the potential to provide a deeper understanding of the system at the cost of needing to learn more conventions, i.e. increase *knowledge in the head.* It is not based on the user's sight but on her understanding of the situation.

9.7 Related Work

We talked about the problem of not being able to regard the Ubicomp environment as a black box, because we are in it all the time. Another approach is to have many independent devices that each can be treated as a black box and that do not have any effect on what is going on outside of them. Norman, the proponent of the black box approach actually proposes just that. He proposes to use information appliances [55], i.e. many "black boxes". This, however, is not really Ubicomp anymore (even though the underlying philosophy is the same), let alone location-aware computing.

When we look at understanding interaction in Ubicomp environments, Dourish [18] has written a whole book on this topic. However, he gives a philosophical account that seems somewhat detached from problems encountered in building Ubicomp applications. It is similar to Weiser [4] using Postmodernism [2] in order to justify the need of Ubicomp. The account given here takes an approach based in cognitive engineering.

Belotti and Edwards [19] have pointed out what needs to be done in order provide more intelligibility and accountability in context-aware systems. Their aim is not to analyse sense-making in Ubicomp per se, but to give guidelines in order to make systems more usable. Their principles are at the same level as Norman's principles [7]. In this chapter, however, we have looked at what leads to these principles, not the design principles themselves.

9.8 Conclusion

We have seen at least four different approaches to understanding what happens in the user's head when she tries to use a device or a system. None of them is incomplete, you can probably explain everything with any approach. Just the focus is different.

The author has chosen approaches that each stress one or more aspects he found important during the experiments. Some of the important conclusions we can draw from this discussion:

- There are different levels of cognition and actions.
- One can distinguish explicit from implicit interaction.
- One can distinguish *knowledge in the head* from *knowledge in the world.*

Different levels of cognition and actions were described most clearly in the activity level approach. What is important to remember is that conscious activity can be decomposed into more or less conscious parts, each being performed at a different cognitive level. The UI should be designed to appeal to each level in its own way and support the corresponding subpart of the activity. For example, it could provide affordances for low-level operations and provide symbolic cues for high-level actions.

Implicit and explicit interaction differ in whether one is consciously using a "computer" or not. It is important for Ubicomp to keep the cognitive effort as low as possible since we are dealing with so many interactions and devices. This is where the discussion on invisibility comes in.

We found that various approaches try to explain what tradeoff the user performs between *knowledge in the head* and *knowledge in the world.* Each approach gives us a different "recommendation" of what to target more.

Each designer needs to strike a balance and hopefully the background provided here will help designers to do so. The author has highlighted parts of every theory that seem to stress what he found important when implementing the interaction prototypes/applications.

Chapter 10

Visual Interaction - Design Principles and Beyond

Chapter 9 presented background theory needed in order to provide users with a conceptual model of a location-aware environment. In this chapter we shall try to make use of this in order to work out methods, heuristics, guidelines and steps that can be used in order to translate the background theory and the experience gained in implementing interactive location-aware applications, into a "good" user interface.

Design can be approached from two points. Either from a philosophical basis or experience. An analysis of either then leads to principles of what is "good". These can then be converted to implementation guidelines that tell you how to achieve what is "good" in a repeatable manner, and finally by putting these guidelines into a framework we can come up with a design discipline.

'Discussion' sections of most Ubicomp papers contain an evaluation of what was found to be good or bad in implementing a particular application. Some have extracted principles ([127] coming from philosophy, [128] coming from experience). Some of these have a wider scope than others.

There is, however, a gap of how to get from principles to implementation. A lot of the problems of Ubicomp we have attempted to solve have been pointed out by other people (such as [129]), but the question of how to solve them is the more difficult question. And how to solve them in a repeatable manner is even more difficult.

We can go even further. We have tools and methods to build Ubicomp applications [86] but there is no design approach that combines software design with interaction design and visual design to create a unified process that leads to a well designed product.

In the following we shall try to attempt to provide such an approach.

10.1 The UI as a Knowledge Presentation Problem

One observation we made during the experiments we have not yet mentioned is the following:

- **We can influence the way the user thinks by providing appropriate visualisations**. When converting the teleport application to an interactive application we were faced with the problem of how to visualise "cycling" through Desktops, something first-time users always find difficult, especially given that you can also cycle through machines at the same time. By showing the Desktops in the form of a menu we could change "cycling" to the much more familiar "scrolling", even though the implementation was doing exactly the same.

Similarly, Norman talks about experiments in which the same problem was presented to people as "Towers of Hanoi", "oranges in plates" and "coffee cups in saucers" ([13], p.86). The way the same problem was presented had a considerable effect on the time to solve it (up to 250%) and the number of errors (up to 600%). So, what we need to keep in mind is that we have to design a conceptual model that will make it easier for the user to make sense of what is going on. At the same time, we realise that there is no "right" way of presenting the system to the user. The abstractions we find and choose to visualise will create a mental model in the user's head. There is no unique way of doing it but some ways are easier to handle for the user than others.

The general problem of user interface design we are facing has been described by Mark [130] as following:

> User meets programmer at the user interface. With current interface design techniques, this is only a chance meeting.

The quote refers to desktop UI design and originates from the mid-eighties. The situation we face in Ubicomp today is similar. So, what we are looking for is a *systematic* way to engineer the user interface.

The development of every application starts with an analysis of the domain. The domain is then modelled using object-oriented design methods. The MVC paradigm [20] now assumes that the model presented to the user is based on the model of the domain. After all, the domain model conceptualises the workings in a clear way by identifying entities and how they relate to each other. Each of the entities is then visualised for the user. In other words, the user interface systematically presents knowledge to the user. One thing we have already mentioned was that there is no unique model of the domain. Every designer works out on model according to the task analysis he performed. Different designers can use different methods and different user models and arrive at a different domain model.

We shall use the MVC as the basis of our user interface design. This will automatically provide a skeleton for the user interface design. The filling out of the "gaps", the design of the Views is where room for creativity is left. Even then we shall regard the user interface as a communication between the designer and the user and try to transmit as much knowledge about the workings of the application without overloading the user cognitively. This is where the findings and theory from the last chapter will come in. All in all we shall expect a user interface that can strike just the right balance between predictability and flexibility.

In the experience gained in the design cases we have more than once realised that having the internals surface at the user interface was beneficial. In fact, visualising the internal "zone" was vital for human-computer understanding, as discussed in the last chapter. Many insights presented in the last chapter were only gained because the MVC forces the visualisation of each and every object of interest. Presenting the list of Desktops (this was done as a scrolling menu) was found to be highly beneficial. This, even though the problems associated with not knowing how many Desktops one has or how far one has got "cycling" had not crossed the mind of the initial developers. What was vital was to organise the Views in a way that they seem coherent in order not to overwhelm the user.

MVC proves to be very *exhaustive* in terms of what information to provide to the user. By presenting relevant objects from the inside of the application the user has all the information needed in order to build a syntactical mental model of the application by interacting with it and observing changes. Depending on how well the Views are designed she will internalise its semantics as well.

At the same time due to the fact that Views can be designed arbitrarily and interaction can be chosen arbitrarily MVC supports enough abstraction required in order not to confuse the user.

The last was said in view of criticism the approach of presenting the implementation model to the user provokes. It is often said that software that corresponds to the implementation model is bad. This, however, depends on the implementation model. The examples normally brought forward show a bad design of the software that is reflected in the user interface and not a paradigm problem per se. It is true that if designers have not modelled the domain properly with objects and their responsibilities the user interface cannot be used to conceal it, if you are using MVC. Other than that, a bad UI that has come out of the use of MVC also shows that the designer has not made enough use of the abstraction mechanisms provided when designing the Presentation (Views) and Interaction (Controllers). But the key remains how well the designer has structured his set of information objects and Models.

10.2 Crafting a Unified Experience

In pre-WIMP Desktop times all interactive applications had their bespoke visualisation, interaction paradigms and screen space management. The advent of GUI environments changed this situation. Users were not facing a number of heterogenous applications anymore that required learning but had one familiar interface to all applications. A unified experience was created.

The situation in Ubicomp is somewhat similar. Users face a multitude of devices, active objects and applications each built by individual designers. There is no framework that could manage this heterogeneity at a user interface level.

The architecture presented in this thesis is generic enough to provide such a framework for location-aware interfaces. Merely the fact that using our system we can rely on a certain output facility all the time eliminates some of the uncertainty that makes it so difficult to come up with a common design approach.

We shall attempt to devise a design approach by extracting principles from the experience gained in the experiments and presenting them as steps to follow. It is hoped that it can serve as a basis even for non-AR interaction design in Ubicomp environments.

- **Design for one environment**. Typically every environment will have its own facilities in terms of middleware, tracking systems, services, devices etc. Also, since we are designing location-aware interfaces limiting the scope at the boundaries of the environment makes sense since users experience a change in context when they enter and leave the environment.

- **Create Scenarios**. Think about actions users want to perform and how you want to use location-awareness.

- **Identify Tasks**. Tasks or actions (in Bærentsen's terminology presented in the previous chapter) are central to our analysis. Think about exceptions that can occur, i.e. unpredictable system behaviour. Don't shy away from introducing explicit interaction just because it would break the illusion of "invisible computing". This process needs to be evolutionary. It is not possible to infer one task model that is applicable to all users from such an analysis. No designer can claim to know all his users a priori. Therefore some user testing will prove useful here.

- **Design the interaction**. Keep the cognitive load low and only use explicit interaction if needed. Have a reusable base of interaction facilities that you can extend as appropriate. Combine existing ones if necessary. Make use of known metaphors. Use the simplest possible interaction facility. Tie facilities down to reference points.

  There is an information bandwidth - attention tradeoff. The more information the user can send to the system the more attention is required. You can make use of context-awareness in order to increase the information bandwidth. These are a kind of modes. Make sure modes are made clear to the user ([110], p.13).

  Remember that location-aware applications often pose high demands on short term memory. Perform an analysis of how many items the user needs to keep in her head when interacting with the system. It should be less than 5 [7].

- **Analyse the Tasks**. Identify information objects for each task. Refer to Olsen's method [109]. Chapter 7 contains an example of how to perform this. Information objects map to Models, related ones can be accumulated in one Model.

  Also identify information required in the user interface, information required for affordances, feedback and mappings. There will be a large overlap to the information identified as information objects but some information will be purely interaction-related. This information is targeted at operations (in Bærentsen's terminology [118]).

  Affordances were defined to be all information that signals to the user what the system affords (can do). Mapping information is needed so that the user can decide what to do in order to get from the current state to a desired state. Feedback is all information needed in order to determine whether the task was completed successfully.

  Decide how to present the information (text, lists, outlines, something else). We can distinguish three types of representational aspects of artifacts. Representational aspects that say something about its usage, show its internals and aspects that help us offload cognitive effort.

- **Visualising Semantics**. The syntax of the interaction is concerned with *how* you use the system. Semantics is about *what* you are actually doing. This is again targeting another cognitive process (please refer to the previous chapter).

  So far, we have only used text and colour encoding to visualise semantics. Books on Information Visualisation [131] contain more suggestions on how to present meaning to the user. This is where symbolic communication and conventions will play an important role.

  The method proposed here is to break down the knowledge required by the user into schemata and to encode each visually. Given that this kind of knowledge has been broken down into basic units according to our theory users should be able to recognise and recall the right schema at the right time. In order for this to work the concept you are representing visually needs to be identifiable. If you want to convey the notion of "depends on" you should use the same visual elements, say, colours, in order to encode each participant and/or process of the schema. Participants in the "depends on" case, for example, can be called subject (the controlling entity) and object (the dependent entity).

  As the number of applications grows users will gradually learn these conventions. This is similar to the way GUIs have evolved.

- **Create the MVC architecture**. Use an application base to plug in your user interface.

The approach outlined here is grounded in the author's own experience. It has been extracted from experience in building interactive location-aware applications. One of the main merits is that it tries to deliver the unified experience we have already talked about. It does this by allowing various applications to be built using the same philosophy, same principles and same approach. It allows a number of applications to share the same interaction facilities base and the same application skeleton. Every application built can present a system image made up of the same components.

This is hopefully a first step in achieving the same predictability of user interface we know from GUIs. In traditional GUI environments every application receives some screen space where it can visually show affordances, provide feedback and propose courses of action to the user. Users quickly learn a visual vocabulary that is used by all applications in a particular environment. Applications that users are not familiar with are instantly learnt and later foster quick recall. Knowledge of a basic visual vocabulary gives users enough guidance to autonomously explore and use unfamiliar inter- and intra-application functionality. The visual symbols together with their static (how they are arranged visually) and dynamic (the order in which you manipulate them) composition will reveal something about the inner workings of the application leading to a deeper understanding of it.

10.3 A Proposal for an Iconic Ubicomp Interface

In the previous section we briefly talked about visualising semantics of the interaction. What is meant by visualising semantics is to show to the user the meaning of an action. In our design examples we mainly used colour coding and text labels for that purpose.

By using symbolic communication we can increase the expressive power of our visualisation manifold. Since we are using Augmented Reality we can easily visualise anything anywhere. And so, adding symbolic communication to our system is technically trivial.

Generally, it would be difficult to design a consistent set of symbols without the use of Augmented Reality. This is perhaps why symbolic communication has hardly been used in Ubicomp. One of the few examples is by Tarasewich et al. [132]. They use arrays of LEDs and encode messages to colour and position. In this way they can enable primitive (less than 5 bits) communication.

One of the earlier Ubicomp applications was a memory prosthesis [133] called Forget-me-not. People working in an office fitted with sensors could have their actions logged. A small personal device would then show them their diary as a memory prosthesis. The display of the device was very small, so the designers used icons in order to show the owner's actions. The actions would appear as, a time followed by a few icons that could be interpreted as a sentence, e.g. "At 10:59 Mike went to the kitchen". The most remarkable property of this device was the expressiveness of just a few icons. Given the facilities Augmented Reality offers us making use of a visual language seems appropriate.

Three types of icons seem to be suitable for our purposes.

10.3.1 Command Icons

These icons are well-known from desktop computing. In our Active Bat menu we have so far used text to describe the menu items. For our teleport application it was appropriate since the menu items (Desktops) were not different enough to justify different icons. There are, however, cases where you want to distinguish often used functions by using icons, for example, if you are using different modes.

10.3.2 Identity Icons

These are the types of icons that were already used in Forget-me-not. People, places, devices and document types were given icons.

In our context, we can solve some difficult problems by using this simple communication facility. In the previous chapter we looked at a hypothetical device we might encounter in the future.

> Your alarm clock might ring later than usual if it logs on to find out that you don't have to get the kids ready for school – [it is] snow day!

It is not difficult to think of a scenario where this clock can go wrong. We could reduce its "autonomy" by adding an icon that indicates its dependencies. The View of the clock would have an icon representing the Internet, showing that this device is more than it seems.

In fact many problems in the scenarios analysed are a result of unpredictable dependencies. If every device's View were to have a coloured bar with icons of other devices, services or sensors its operation depends on users would have a better chance of figuring out causal connections.

Consider again this quote:

> [...] while I was at University of Toronto, we did some early work on what later became known as the "reactive room". Initially versions of the conference room were known as the "possessed room". Lights turned on and off when you moved or ceased moving. Equipment "spoke to you" through an electronic voice and told you that you had 5 seconds to "take action" or it would power down. Video signals switched to channels and "forgot" how to switch back.

Imagine how much more understandable this could be made by visualising dependencies on the lights and equipment.

10.3.3 Icons for Schema Recall

In the previous chapter we mentioned schemata can be recalled. One of the concepts we mentioned was "Dependent On". The previous section has shown a way to recall this concept using icons.

We can make use of this insight by generally trying to cue the recall of schemata by icons. According to schema theory small cues start a process in the head of the human that can lead to a comprehensive understanding of the whole situation by making use of the human's associative powers.

If we can provide these cues in the form of icons we can channel the user's mind into the right direction. Let us look at the schemata we identified in the example scenario in the previous chapter:

- "Follow me"
- "Output Device"
- "Inferred choice"
- "Controllable"
- "One-time authentication"
- "In Control"
- "Movable"

Figure 10.1 shows simple icons corresponding to some of the schemata mentioned above. The first icon represents "follow me", the second represents "one-time authentication. If the authentication has been done the lock will be displayed as open. The third represents a "movable" session. The last icon represents "being in control of a session". Icons that represent what the device/application affords are outlined in green (first and third). Icons that relate to a static

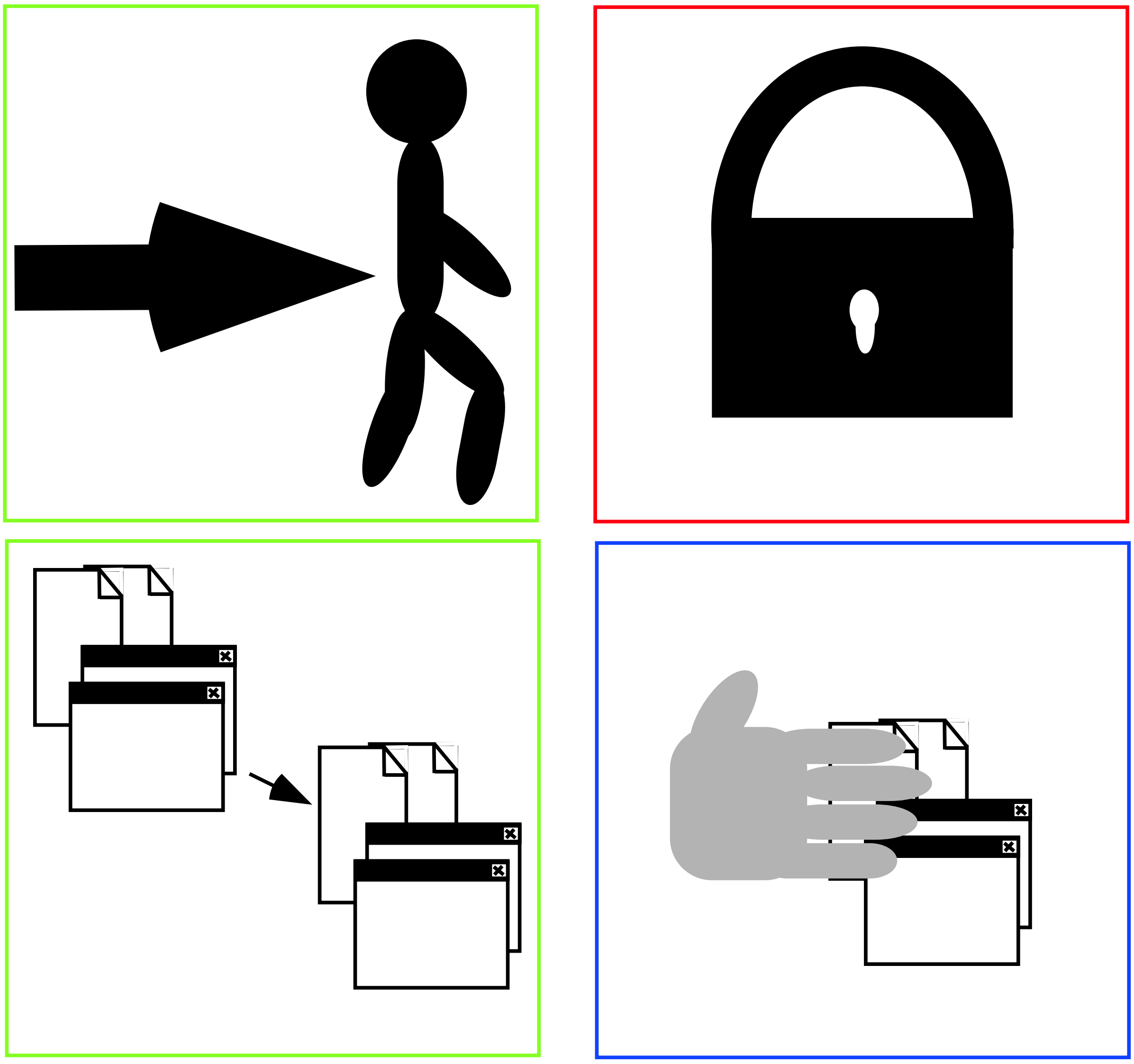


Figure 10.1: Schemata visualised as Icons.

property are outlined in red (the second one). Finally, icons that refer to a dynamic feedback are outline blue (the last one).

Regarding the design of the user interface as a knowledge presentation problem, using icons to (re)present schemata is a natural extension to what we have presented so far.

10.3.4 Creating a Visual Language

We briefly talked about the need to encode schemata visually and make use of this coding when designing the presentation of our user interface. In this way it can make use of them when it tells the user what it is about.

How such a vocabulary looks like depends on your environment, applications, common practice and background of users. It needs to evolve amongst the people and in this aspect it is similar to natural languages. In our lab for example there are a number of people who share the same kind of jargon relating to location-aware applications, such as "region", SPIRIT Button etc.

The starting point for such a language should be a number of schemata previously identified. For our iconic interface we will map each schema to an icon. We will find icons that represent procedural (how) knowledge and icons that present declarative (what) knowledge. Declarative icons will tell us something about the world. Amongst these there will be icons that represent static properties and icons that represent temporary states.

To this base we will gradually add and remove icons according to how they are received by users. The aim is to get a well-designed language that is comparable to what we know from the GUI.

10.4 Related Work

Dourish [127] has provided a framework of principles for interaction in Ubicomp. The principles, however, are quite abstract and more comparable to background theory designers should know before they design interaction. The process given here is much more concrete. The downside is that it makes assumptions about the type of interaction you want to support. Using the background theory from the previous chapter, developers will hopefully be able to port it to other types of interaction. Another difference is that the approach presented here is based on engineering, not philosophy.

Belotti et al. [19] also draw from GUIs in order to seek solutions for hard interaction challenges in Ubicomp. Similar to what is proposed here, they stress the need to improve human-computer communication and classify interaction problems. But they do not provide any solutions. After all, they are more concerned with "questions for designers and researchers".

Selker and Burleson [134] make a case for using insights gained from cognitive science in order to design context-aware systems. They present a number of examples of good design. The principles they propose are very general and do not stem from attempting to tackle hard interaction challenges in Ubicomp.

10.5 Conclusion

In this chapter we have presented an all-integrated design approach. Conceptual design, implementation and visual design are all performed inter-dependently. The strong link between them reduces space for accidents and yet leaves enough room for creativity. At the same time the fact that we have a basic set of principles, architectural constructs and interaction prototypes will lead to a more coherent user experience.

The approach presented is grounded in cognitive engineering and user interface engineering. The background theory from the last chapter showed us the breadth of human-computer communication and we have attempted to make use of different levels of human understanding.

We found that task analysis is central for designing interfaces. In a nutshell, this approach is about decomposing tasks and supporting them most efficiently with visual information. This information varies in its sophistication.

By regarding UI design as a knowledge presentation problem we were able to explore design avenues that had previously been given little attention. For example, symbolic communication has the potential to communicate schemata to users that are required for understanding the system. A better understanding of the Ubicomp environment is vital if Ubicomp is to be deployed at a large scale and to be used daily by everyone.

Chapter 11

Conclusion

11.1 Further Work

Now that we have a GUI like platform we can port many of our insights from that field to Ubicomp. The interaction prototypes chapter touched on information-theoretic analysis of interfaces. Raskin [102] has done something similar and more comprehensive for the GUI. Extending that to Ubicomp might be promising.

Extending the interaction base provided in Chapter 6 could be another task. We briefly mentioned that Active Bats can be attached to any object. This avenue can be further explored. We have only worked with one user's Active Bat. Implementing Subject-Verb-Object commands might be the next task.

Dey et al. have implemented a nice context-aware application for GUIs called CyberDesk [135] that lets you select some piece of text and gives you options as to what you might want to do with it. Encouraging synergy (being able to combine services) for Ubicomp environments is all the more important since that will be the edge Ubicomp will need over the powerful desktop computer.

We touched upon multiple users and multiple applications but we did not implement a reliable multi-application/user system. This is another interesting issue.

11.2 Context of this Research Work

Before we sum up the work, let us try to locate it in the research universe. First of all, it needs to be said that the view of Human-Computer Interaction used for this work is not absolute. The interaction model employed here is essentially a dialogue-based model that has its origin in Norman's Seven Stages [7]. Between the user and the machine is a divide that needs to be bridged by gulfs of evaluation and execution. And this bridge is provided by the designer.

In other interaction models such as the one described by Dourish [18] designers move into the background. Their aim becomes not to design every stage of the interaction but to reveal the workings of the system to an extent that users are empowered to make use of the system in unanticipated ways. In this point this work is closer to Dourish than Norman.

Similarly the idea of "invisibility" presented here is influenced by the use of Norman's model which is a *designer's* model. The phenomenon we witness is that some technologies/applications become invisible for users. i.e. they do not realise they are using them at all. Not everyone would agree that invisibility can be achieved by "designing for invisibility". One could argue that invisibility arises through a particular use and this use should or cannot be determined at design time.

The general problem is how can the designer model her user if users are so unpredictable. One way forward is not to work with a fixed user model but a mutable one. Designers need to make initial assumptions about users, design applications accordingly, deploy them, observe users and update their assumptions for the next iteration. At the same time designers should convey the workings of their applications in order to empower users to use them in unanticipated ways.

On the other hand, it is easier to model one user than to model a community of users that influence each other. This work has mainly just dealt with designing for *a* user. The suitability of a single user/single input device **design** assumption for Ubicomp can be challenged, but partly this assumption was inherited from the existing system, that, let us not forget, has been the origin of a number of applications that have not only found a good user reception but also have encouraged users to make use of technology in unanticipated ways (c.f. the SPIRIT Button, see Figure 2.3).

11.3 Summary

This thesis started off with the realisation that many existing Ubicomp applications suffer from fundamental usability problems. These manifest themselves particularly in location-aware computing that is one "flavour" of Ubicomp. The usability problems were found to arise from the lack of a good conceptual model and limited control and feedback. Certain characteristics of Ubicomp such as heterogeneity and its distributed nature further add to making interaction in such environments unmanageable for the user.

It was decided that Ubicomp could benefit from traditional user interface engineering. The hypothesis was that location-aware applications could be made interactive in order to make them more understandable. In desktop computing GUIs accomplish exactly this aim.

The approach used was to adapt this successful kind of interaction paradigm for Ubicomp. Analogously to the desktop where running applications have screen space, applications that ran in the physical world would be able present themselves visually in the physical space. In order to achieve this Augmented Reality was to be used and give the user the illusion that real objects or locations are visually as well as computationally reacting to their actions.

We started by building a AR system that would be suitable for our purposes. The AR system was successfully implemented and registration was found to be good enough for our purposes.

We then moved on to a proof-of-concept chapter. Implications of using AR in an office-scale environment were examined. In order to show that it is indeed feasible to use AR in such an environment we needed to demonstrate that the following was achievable for our system. Firstly, being tetherless and secondly to be able to integrate different tracking technologies and approaches that are commonly used indoors.

The next task was to architect a framework that combines AR and Ubicomp. Additionally, the framework was made to support interactivity at an architectural level. A Model-View-Controller architecture was implemented and its benefits for Ubicomp environments were examined. One of the favourable features found was the ability of the framework to support prototyping and simulation.

The design cases made use of this framework and it proved to be very flexible. After all, they were concerned both with augmenting implicit interaction and introducing explicit interaction into Ubicomp environments. Interaction challenges were systematically categorised and a family of interaction facilities was implemented in order to allow each challenge to be solved by the simplest facility. This base of interaction facilities was made to be pluggable into any location-aware application built with this framework.

This was done by augmenting a legacy location-aware application. A teleport application was made interactive, keeping the already existing functionality in order to deploy the first interactive application running in physical space. The application was received favourably by users.

Studying users who have been using location-aware applications in an environment that shows a glimpse of the future was an invaluable resource in drawing conclusions about how users perceive location-aware applications. Such users are rare and hopefully the results gathered will help other researchers in understanding the user of a Ubicomp environment.

The next chapter dealt with the sense-making process of users in a Ubicomp environment. It was found that there were many approaches, each stressing some other aspect. It was found that basing design on a purely perception-based approach, the black box approach, is not always suitable for a Ubicomp environment. A conception-based approach was introduced and its implications with respect to cognitive load were examined. The conclusion was that different cognitive processes can be addressed by different approaches.

Finally, using the insights gained from the cognitive science perspective and the experience gained in implementing interaction we tried to extract design principles and a design approach. The guidelines provided can be used as a basis for interactive location-aware applications and provide users with a unified experience similar to the GUI.

The following gives the main contributions of this work.

- **Provision of visual feedback for location-aware applications and a study this new form of interaction**. For the first time location-aware applications have had a chance to present their (inner) workings to the user. Most notably, we were able to show spatial aspects (such as regions) of applications that run in the physical world to users. We have seen that such a visual presentation of location-aware applications has not only increased user understandability but introduces new way humans can think about and use context-aware applications.

- **The introduction of a new user interface architecture to support interactivity in Ubicomp**. Over the last 20 years the rise of interactive systems has been accompanied by the development of a number of user interface architectures. Starting with such a typical UI architecture, relevant changes were made in order for it to be used in a physical setting. The architecture presented here can be used to tackle the complexity associated with the introduction of a sophisticated graphical interface to context-aware computing and making context-aware applications more responsive for the user.

- **A reflection on the use of traditional design principles in Ubicomp**. The introduction of a visual interface has allowed us to experiment with using design principles known from desktop computing and other engineering disciplines. Most prominently, the use of Norman's design principles [7] has been attempted in a systematic way. This effort together with a critical evaluation of this approach has resulted in a number of conclusions that provide useful insights for Ubicomp designers.

The problems pointed out in the beginning of this thesis have been recognised before. In fact many researchers have recognised them (to name some: [19], [136], [129], [15]). They appear often appear as "footnotes" in papers describing implemented systems [137]:

> If a pair of headphones with appropriate capabilities were to enter the fray ("I'm a pair of headphones"), redirecting the sound output to them should become an available option.Where these options are displayed, how the user is informed of these options, and how much autonomy the system has remain open questions.

But what had not been done so far, was to classify them and attempt to provide a solution. The solution proposed here might not be ideal for everyone, but what this thesis will hopefully achieve is that researchers will not shy away from tackling interaction challenges lest they break the illusion of invisible computing. And hopefully they will introduce some element of interaction design into their work.

A lot of this thesis has been concerned with bringing structure into the perceived chaos of devices, settings, inferences and applications. The underlying approach employed was to provide abstractions for both designers and users.

The element that will probably attract the most criticism is the use of bulky HMDs. However, it should be kept in mind that this thesis was investigating how humans will interact with computers in the future. At the moment Ubiquitous Computing is no more reality for most humans as are HMDs. And yet there are strong proponents for both.

It is no surprise if computer enthusiasts such as Michael Dertouzos [56] base a large part of their work on future interaction on Ubicomp and HMDs. In fact the most recent issue of IEEE Spectrum has a whole cover story dedicated to the "ultimate display" [138]. But when people whose main concern is usability have accepted HMDs as part of their future lives, it is a different story. Jef Raskin, you could call him the inventor of the Macintosh, in an interview, asked about the future of computing very recently said [139]:

> "We will see a change to simplicity.People are too fed up with the complexity of computers. And I predict we will see more wearable, head-mounted displays."

Donald Norman, another interaction expert, likewise believes that the augmented human being is inevitable [58]. Susan Greenfield, being one of the most respected scientists in Britain, is anything but suspect of being a "geek". The book [1] she wrote after researching how the future will look like, makes interesting predictions regarding the future of computing: the two computing technologies mentioned are Ubicomp and Augmented Reality.

But, the best champion of Augmented Reality probably is Weiser himself. In his seminal Scientific American article [5] he describes something that requires visualisation (most likely on the window, judging from the textual context) and a very sophisticated head tracker.

> Sal looks out of her windows at her neighborhood. Sunlight and a fence are visible through one, but through others she sees electronic trails that have been kept for her of neighbors coming and going during the early morning. Privacy conventions and practical data rates prevent displaying video footage, but time markers and electronic tracks on the neighborhood map let Sal feel cozy in her street.

Appendix A

Guide Questions for the Evaluator

1. How many Active Desktops do you have?
2. Is your Teleporting on or off? Would you prefer to control it from your bat or a button on the wall?
3. What do you know about Teleporting?
4. How does it work? (For novices delay this question, until they have explored the application)
5. *Evaluator*: identify concepts, conventions and prompt user
6. Can you Teleport to the broadband phones?
7. *Evaluator*: Explain Experiment.
8. *Evaluator*: Let user play with invisible application, observe difficulties.
9. *Evaluator*: Ask "what if" questions involving one button press, user movement and a button press and combinations of the two.
10. Imagine you had to give another user a manual for this application. Can you make a drawing instead?
11. *Evaluator*: Let user play with visible application, observe difficulties.
12. *Evaluator*: Ask "what if" questions involving one button press, user movement and a button press and combinations of the two.
13. How does it work?
14. *Evaluator*: identify concepts, conventions and prompt user
15. Teleporting is best described as a property of: Space, Bat, Machine, "System", Bat System, other:

Bibliography

[1] Susan Greenfield. *Tommorow's People.* Allen Lane, 2003.

[2] Stephen Toulmin. *Cosmopolis: The Hidden Agenda of Modernity.* The University of Chicago Press, 1992.

[3] European Commission Research Project. The Disappearing Computer Initiative, 2000. Available at: `http://www.cordis.lu/ist/fetdc.htm`.

[4] M. Weiser. Building Invisible Interfaces. Presentation Slides, 1994. Talk given at UIST'94.

[5] M. Weiser. The Computer for the 21st Century. *Scientific American*, pages 94–104, Sept. 1991.

[6] Kalle Lyytinen and Youngjin Yoo. Issues and challenges in ubiquitous computing. *Communications of the ACM*, 45(12):63pp, December 2002.

[7] D.A. Norman. *The Design of Everyday Things.* The MIT Press, 1998.

[8] H. Ishii and B. Ullmer. Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'97)*, 1997.

[9] R. Want, K.P. Fishkin, A. Gujar, and B.L. Harrison. Bridging Physical and Virtual Worlds with Electronic Tags. In *Proceedings of CHI '99*, pages 370–377, 1999.

[10] Paul M. Aoki and Allison Woodruff. Improving Electronic Guidebook Interfaces Using a Task-Oriented Design Approach. In *Proceedings of the Symposium on Designing Interactive Systems*, pages 319–325, 2000.

[11] R. Want and A. Hopper. Personal Interactive Computing Objects. Technical Report TR 92.2, AT&T Research Laboratories, Cambridge, 1992.

[12] R. Want and A. Hopper. Active badges and personal interactive computing objects. *IEEE Transactions on Consumer Electronics*, 38(1), February 1992.

[13] Donald Norman. *Things that make us smart: Defending Human Attributes in the Age of the Machine.* Perseus Publishing, 1993.

[14] P.J. Brown. Some Lessons for Location-aware Applications. In *Proceedings of first workshop on HCI for mobile devices*, pages 58–63. Glasgow University, May 1998.

[15] B. Harrison. Position Paper for Ubiquitous Computing Workshop. In *Proceedings of the Workshop on Ubiquitous Computing: The Impact of Future Interaction Paradigms and HCI Research at CHI '97*, 1997.

[16] Andy Hopper. The Royal Society Clifford Paterson Lecture, 1999. Available at: `http://www.uk.research.att.com/pub/docs/att/tr.1999.12.pdf`.

[17] Roy Want, Andy Hopper, Veronica Falcao, and Jonathon Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[18] Paul Dourish. *Where the action is.* MIT Press, 2001.

[19] Victoria Bellotti, Maribeth Back, W. Keith Edwards, Rebecca E. Grinter, D. Austin Henderson Jr., and Cristina Videira Lopes. Making Sense of Sensing Systems: Five Questions for Designers and Researchers. In *Conference on Human Factors in Computing Systems*, pages 415–422, 2002.

[20] G. Krasner and S. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[21] A. Butz, J. Baus, A. Krueger, and M. Lohse. A hybrid indoor navigation system. In *Proceedings of IUI.* ACM, 2001.

[22] G.D. Abowd, C.G. Atkeson, J. Hong, S. Long, R. Koope, and M. Pinkerton. Cyberguide: A Mobile Context-Aware Tour Guide. Technical report, GVU, Georgia Institute of Technology, 1996.

[23] D. Petrelli, E. Not, M. Sarini, O. Stock, A. Trapparava, and M. Zancanarov. HyperAudio: Location-Awareness + Adaptivity. In *Proceedings of CHI'99, Conference on Human Factors in Computing Systems*, pages 21–22, 1999.

[24] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou. Developing a context-aware electronic tourist guide: Some issues and experiences. In *Proceedings of CHI 2000*, pages 17–24, 2000.

[25] Bradley J. Rhodes. The wearable remembrance agent: A system for augmented memory. In *Proceedings of The First International Symposium on Wearable Computers (ISWC '97)*, pages 123–128, Cambridge, Mass., USA, 1997.

[26] R.W. DeVaul and A. Pentland. The Ektara Architecture: The Right Framework for Context-Aware Wearable and Ubiquitous Computing Applications. Technical report, The Media Laboratory, Massachusetts Institute of Technology, 2000.

[27] T. Hoellerer, S. Feiner, and J. Pavlik. Situated documentaries: Embedding multimedia presentations in the real world. In *Proceedings of the 3rd International Symposium on Wearable Computers*, 1998.

[28] Daniel P. Siewiorek, Asim Smailagic, Leonard J. Bass, Jane Siegel, Richard Martin, and Ben Bennington. Adtranz: A mobile computing system for maintenance and collaboration. In *International Symposium on Wearable Computers*, pages 25–32, 1998.

[29] Steven Feiner, Blair Macintyre, and D. Seligmann. Knowledge-based augmented reality. *Communications of the ACM*, 36(7):53–62, 1993.

[30] L. Degen, R. Mander, and G. Salamon. Working with audio: Integrating personal tape reorders and desktop computers. In *Proceedings of CHI'92*, pages 413–418, 1992.

[31] G.D. Abowd, L.D. Harvel, and J.A. Brotherton. Building a Digital Library of Captured Educational Experiences. In *Proceedings of the 2000 International Conference on Digital Libraries*, 2000.

[32] A.K. Dey, M. Futakawa, D. Salber, and G.D. Abowd. The Conference Assistant: Combining Context-Awareness with Wearable Computing . In *Proceedings of the 3rd International Symposium on Wearable Computers (ISWC '99)*, pages 21–28, 1999.

[33] Y. Sumi, E. Tarneyuki, S. Fels, N. Simonet, K. Kobayashi, and K. Mase. C-MAP: Building a Context-Aware Mobile Assistant for Exhibition Tours. In *Proceedings of the First Kyoto Meeting on Social Interaction and Communityware*, 1998.

[34] N.S.Ryan, J.Pascoe, and D.R.Morse. FieldNote: a handheld information system for the field. In R.Laurini, editor, *Proc. TeleGeo'99, 1st International Workshop on TeleGeo-Processing*, pages 156–163. Claude Bernard University of Lyon, May 1999.

[35] MIT Media Lab C@CML. Chameleon Mug. Electronic Source, 2001. Available at: `http://www.media.mit.edu/context/mug.html`.

[36] MIT Media Lab C@CML. Cutlery that Nose. Electronic Source, 2001. Available at: `http://www.media.mit.edu/context/cutlery.html`.

[37] J.F. McCarthy and T.J. Costa. UniCast and GroupCast: An Exploration of Personal and Shared Public Displays. In *Proceedings of the ACM CSCW 2000 Workshop on Shared Environments to support Face-to-Face Collaboration*, 2000.

[38] L.-E. Holmquist, J. Falk, and J. Wigstroem. Supporting group collaboration with interpersonal awareness devices. *Personal Technologies*, 3:13–21, 1999.

[39] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[40] R. Borovoy, M. McDonald, F. Martin, and M. Resnick. Things that blink: Computationally augmented name tags. *IBM Systems Journal*, 35(3-4), 1996.

[41] P. Yarin. TouchCounters, Interactive Electronic Labels. Electronic Source, 2001. Available at: `http://www.media.mit.edu/~yarin/touchcounters/`.

[42] M. Weiser and J.S. Brown. Designing Calm Technology. Electronic Source, 1995. Available at: `http://nano.xerox.com/hypertext/weiser/calmtech/calmtech.htm`.

[43] A. Schmidt, K. Aidoo, A. Takaluoma, U. Tuomela, K.v. Laerhoven, and W.v. de Velde. Advanced interaction in context. In *Proceedings of HUC*, 1999.

[44] E. Pederson and T. Sokoler. AROMA: Abstract Representation of Presence Supporting Mutual Awareness. In *Proceedings of CHI 97*. ACM Press, 1997.

[45] Rodenstein R and G.D. Abowd. OwnTime: A System for Timespace Management . In *Proceedings of the 1999 Conference on Human Factors in Computing Systems (CHI '99)*, 1999.

[46] Redstroem J., T. Skog, and L. Hallnaes. Informative Art: Using Amplified Artworks as Information Displays. In *Proceedings of DARE 2000 (Designing Augmented Reality Environments)*, 2000.

[47] R. Rodenstein. Employing the Periphery: The Window as Interface. In *Proceedings of Human Factors in Computing Systems '99*, 1999.

[48] M. Tuteja. AnchoredDisplays: The Web on Walls. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'97)*, 1997.

[49] Y. Ayatsuka, N. Matsushita, and J. Rekimoto. HyperPalette: A Hybrid Computing Environment for Small Computing Devices. In *CHI 2000 Extended Abstracts*, pages 133–143, April 2000.

[50] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. In *ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom-99)*, 1999.

[51] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *6th ACM MOBICOM*, 2000.

[52] Steven Feiner, Blair MacIntyre, Tobias Hol̈lerer, and Anthony Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. In *Proceedings of International Symposium on Wearable Computing (ISWC)*, pages 74–81, October 1997.

[53] H. Kato and M. Billinghurst. Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System. In *Proceedings 2nd International Workshop on Augmented Reality.*, pages 85–94, 1999.

[54] T. Kindberg, J. Barton, J. Morgan, G. Becker, I. Bedner, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, C. Pering, J. Schettino, and B. Serra. People, Places, Things: Web Presence for the Real World. In *Proceedings of WMCSA '00*, 2000.

[55] D.A. Norman. *The Invisible Computer.* The MIT Press, 1999.

[56] M. Dertouzos. *What will be.* Piatkus, London, 1997.

[57] R.J.K. Jacob. Uist'007: Where will we be ten years from now? In *Proceedings UIST*, pages 115–118, 1997.

[58] D. Norman. Cyborgs of the new millenium. Electronic Source. Available at: `http://www.jnd.org/dn.mss/Cyborgs.html`.

[59] J. Newman, D. Ingram, and A. Hopper. Augmented Reality in a Wide Area Sentient Environment. In *Proceedings ISAR (International Symposium on Augmented Reality)*, 2002.

[60] Polhemus. *3SPACE TRACKER: User's Manual.*

[61] Francis S. Hill. *Computer Graphics Using Open GL, Second Edition.* Prentice Hall, 2000.

[62] Jackie Neider. *The OpenGL Programming Guide: Release 1.* Addison Wesley, 1993.

[63] Olivier Faugeras. *Three-dimensional Computer Vision: A Geometric Viewpoint.* MIT Press, 1996.

[64] Anton L. Fuhrmann, Rainer Splechtna, and Jan Prikryl. Comprehensive calibration and registration procedures for augmented reality. In *Eurographics Workshop on Virtual Environments*, 2001.

[65] M. Tuceryan, Y. Genc, and N. Navab. Single point active alignment method (spaam) for optical see-through hmd calibration for augmented reality. *Presence: Teleoperators and Virtual Environments*, 11(3):259–276, June 2002.

[66] E. McGarrity, M. Tuceryan, C. Owen, Y. Genc, and N. Navab. A new system for online quantitative evaluation of optical see-through augmentation. In *IEEE and ACM International Symposium on Augmented Reality*, 2001.

[67] Josie Wernecke. *The Inventor Mentor*. Addison-Wesley, 1994.

[68] Volodymyr Kindratenko. Electromagnetic tracker calibration. Electronic Source, 2004. Available at: `http://www.ncsa.uiuc.edu/VEG/VPS/emtc/index.html`.

[69] P. Malbezin, W. Piekarski, and B. H. Thomas. Measuring artoolkit accuracy in long distance tracking experiments. In *In 1st Int'l Augmented Reality Toolkit Workshop*, 2002.

[70] Holger T. Regenbrecht and Michael T. Wagner. Interaction in a collaborative augmented reality environment. In *CHI '02 extended abstracts on Human factors in computing systems*, pages 504–505. ACM Press, 2002.

[71] Intel. *Intel OpenCV Website*, 2004. Available at: `http://www.intel.com/research/mrl/research/opencv/`.

[72] University of Utah. Artoolkit patternmaker website, 2004. Available at: `http://www.cs.utah.edu/gdc/projects/augmentedreality/`.

[73] N. Adly, P. Steggles, and H. Andy. Spirit:a resource database for mobile users. In *Conference on Human Factors in Computing Systems (CHI '97)*, 1997.

[74] Hani Naguib and George Coulouris. Location information management. In *Ubicomp*, pages 35–41, 2001.

[75] Jeffrey Hightower and Gaetano Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, 2001.

[76] Ulf Leonhardt and Jeff Magee. Multi-sensor location tracking. In *4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 203–214. ACM Press, 1998.

[77] A. Narayanan. Realms and states: A framework for context-aware mobile computing. In *Intelligent and Interactive Assistance & Mobile Multimedia Computing*, 2000.

[78] Gerhard Reitmayr and Dieter Schmalstieg. An open software architecture for virtual reality interaction. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 47–54. ACM Press, 2001.

[79] Erwin Aitenbichler and Max Mühlhäuser. The Talking Assistant Headset: A Novel Terminal for Ubiquitous Computing. Technical Report TK-02/02, Fachbereich Informatik, TU Darmstadt, 2002.

[80] AT&T Research Laboratories, Cambridge. Sentient computing project home page. Electronic Source. Available at: `http://www.uk.research.att.com/spirit/`.

[81] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, July 1995.

[82] Andreas Zajic. Application management for three-dimensional user interfaces. Master's thesis, TU Wien, 2003.

[83] Martin Bauer, Bernd Bruegge, Gudrun Klinker, Asa MacWilliams, Thomas Reicher, Stefan Riss, Christian Sandor, and Martin Wagner. Design of a component-based augmented reality framework. In *Proceedings of ISAR 2001*, 2001.

[84] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[85] Gregory D. Abowd, Elizabeth D. Mynatt, and Tom Rodden. The human experience. *IEEE Pervasive Computing*, 1(1):48–57, 2002.

[86] A. Dey, G.D. Abowd, and D. Salber. A context-based infrastructure for smart environments. In *Managing Interactions in Smart Environments (MANSE '99)*, pages 114–128, 1999.

[87] R. Hull, P. Neaves, and J. Bedrod-Roberts. Towards situated computing. In *1st International Symposium on Wearable Computers(ISWC '97)*. IEEE Press, 1997.

[88] J. Pascoe. Adding generic contextual capabilities to wearable computers. In *2nd International Symposium on Wearable Computers*, pages 92–99, 1998.

[89] Paul Dourish. What We Talk About When We Talk About Context. *Personal and Ubiquitous Computing*, 8(1):19–30, 2004.

[90] Manuel Roman and Roy H. Campbell. A distributed object-oriented application framework for ubiquitous computing environments. In *12th Workshop for PhD Students in Object-Oriented Systems*, 2002.

[91] Ben Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *Proceedings of the 2nd international conference on Intelligent user interfaces*, pages 33–39. ACM Press, 1997.

[92] John Barton and Vikram Vijayaraghavan. Ubiwise, a simulator for ubiquitous computing systems design. Technical Report HPL-2003-93, HP Labs, 2003.

[93] J.C. de Oliveira. Synchronized world embedding in virtual environments. *IEEE Computer Graphics and Applications*, 24(4):73–83, 2004.

[94] James J. Gibson. *The Ecological Approach to Visual Perception*. Lawrence Erlbaum Associates, 1986.

[95] M. Weiser. Does Ubiquitous Computing Need Interface Agents? Presentation Slides, October 1992. Available at: `http://www.ubiq.com/hypertext/weiser/Agents.ps`.

[96] James Hollan, Edwin Hutchins, and David Kirsh. Distributed cognition: toward a new foundation for human-computer interaction research. *ACM Trans. Comput.-Hum. Interact.*, 7(2):174–196, 2000.

[97] Felix Hupfeld and Michael Beigl. Spatially aware local communication in the raum system. In *IDMS*, pages 285–296, 2000.

[98] E. Horvitz. Principles of mixed-initiative user interfaces. In M.G. Williams, M.W. Altom, K. Ehrlich, and W. Newman, editors, *Human Factors in Computing Systems, CHI'99*, pages 159–166, 1999. http://research.microsoft.com/ horvitz/UIACT.HTM.

[99] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., 1988.

[100] T.D. Hodes, R.H. Katz, E. Servan-Schreiber, and L.A. Rowe. Composable ad-hoc Mobile Services for Universal Interaction. In *Proceedings of MOBICOM '97*, pages 1–12. ACM, 1997.

[101] Wesley Chan. Using coolbase to build ubiquitous computing applications. Technical Report HPL-2001-215, HP Labs, 2001.

[102] Jeff Raskin. *The Humane Interface.* Addison-Wesley, 2000.

[103] Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. istuff: a physical user interface toolkit for ubiquitous computing environments. In *Proceedings of the conference on Human factors in computing systems*, pages 537–544. ACM Press, 2003.

[104] Shwetak N. Patel and Gregory D. Abowd. A 2-Way Laser-Assisted Selection Scheme for Handhelds in a Physical Environment. In *UbiComp 2003*, pages 200–207, Seattle, WA, October 2003. Springer.

[105] Ivan Poupyrev, Desney S. Tan, Mark Billinghurst, Hirokazu Kato, Holger Regenbrecht, and Nobuji Tetsutani. Developing a generic augmented-reality interface. *IEEE Computer*, 35(3):44–50, March 2002.

[106] K. Rehman. 101 Ubiquitous Computing Applications. Web Site, May 2001. Available at: `http://www-lce.eng.cam.ac.uk/~kr241`.

[107] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, Jan/Feb 1998.

[108] Anind K. Dey, Albrecht Schmidt, and Joseph F. McCarthy, editors. *UbiComp 2003: Ubiquitous Computing, 5th International Conference, Seattle, WA, USA, October 12-15, 2003, Proceedings*, volume 2864 of *Lecture Notes in Computer Science.* Springer, 2003.

[109] D.R. Olsen. *Developing User Interfaces.* Morgan Kaufmann, 1998.

[110] Apple Computer Inc. *Macintosh Human Interface Guidelines*, chapter Human Interface Principles, pages 12–13. Addison Wesley, 1992.

[111] Donald A. Norman. Some observations on Mental Models. In Genter and Stevens, editors, *Mental Models*, pages 7–14. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[112] Michael D. Byrne and Susan Bovair. A Working Memory Model of a Common Procedural Error. *Cognitive Science*, 21(1):31–61, 1997.

[113] Karl B. Schwamb. Mental models: A survey. Available at: `http://www.isi.edu/soar/schwamb/papers/mm-survey.ps`.

[114] Panos Fiambolis. Virtual retinal display technology. Electronic Source, 1999. Available at: `http://www.cs.nps.navy.mil/people/faculty/capps/4473/projects/fiambolis/vrd/vrd_full.html`.

[115] M. Helander, T. K. Landauer, and P. Prabhu, editors. *Handbook of Human-Computer Interaction*, chapter The Role of Metaphors in User Interface Design, pages 441–462. Elsevier Science B.V., 1997.

[116] David Kirsh. The intelligent use of space. *Artif. Intell.*, 73(1-2):31–68, 1995. Available at: `http://icl-server.ucsd.edu/~kirsh/Articles/Space/AIJ1.html`.

[117] Steven Levy. The new digital galaxy. *Newsweek*, May 1999. Available at: `http://www.emware.com/news/emware%20news/1999/newsweek.html`.

[118] Klaus B. Bærentsen and Johan Trettvik. An activity theory approach to affordance. In *Proceedings of the second Nordic conference on Human-computer interaction*, pages 51–60. ACM Press, 2002.

[119] Frank Halasz and Thomas P. Moran. Analogy considered harmful. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 383–386. ACM Press, 1982.

[120] Karl E. Weik. *Sensemaking in Organizations*, chapter Belief-Driven Process of Sensemaking, page 133. Sage, 1995.

[121] J. M. Caroll. *The Nuremberg Funnel: Designing Minimalist Instruction for Practical Computer Skill.* Erlbaum, 1990.

[122] N. Kohtake, J. Rekimoto, and Y. Anzai. InfoStick: An Interaction Device for Inter-Appliance Computing. In *Proceedings of HUC'99*, pages 246–258, 1999.

[123] Donald Norman. Affordances and design. Electronic Source, 1999. Available at: `http://www.jnd.org/dn.mss/affordances-and-design.html`.

[124] David E. Rumelhart. *Handbook of Social Cognition*, chapter Schemata and the Cognitive System, pages 161–187. Lawrence Erlbaum Associates, 1984.

[125] Roger H. Bruning, Gregory J. Schraw, and Royce R. Ronning. *Cogintive Psychology and Instruction.* Prentice Hall, 2nd edition, 1995.

[126] Barry Brumitt, John Krumm, Amanda Kern, and Steven Shafer. Easyliving: Technologies for intelligent environments. In *Handheld and Ubiquitous Computing*, 2000.

[127] Paul Dourish. *Where the action is*, chapter Moving Towards Design. MIT Press, 2001.

[128] Saul Greenberg. Context as dynamic a construct. *HCI*, 16:257–268, 2001.

[129] W. Keith Edwards and Rebecca E. Grinter. At Home with Ubiquitous Computing: Seven Challenges. In *Proceedings of the 3rd international conference on Ubiquitous Computing*, pages 256–272. Springer-Verlag, 2001.

[130] William Mark. *User-Centered System Design*, chapter Knowledge-based Interface Design, pages 219–238. Lawrence Erlbaum Associates, 1986.

[131] Robert Spence. *Information Visualization.* Pearson Addison Wesley, 2000.

[132] Peter Tarasewich, Christopher S. Campbell, Tian Xia, and Myra Dideles. Evaluation of visual notification cues for ubiquitous computing. In *Conference on Ubiquitous Computing*, pages 349–366, Seattle, WA, October 2003. Springer.

[133] M. Lamming and M. Flynn. Forget-me-not: intimate computing in support of human memory. In *Proceedings FRIEND21 Symposium on Next Generation Human Interfaces*, 1994.

[134] T. Selker and W. Burleson. Context-aware design and interaction in computer systems. *IBM Systems Journal*, 39(3&4), 2000.

[135] Anind K. Dey, Gregory D. Abowd, and Andrew Wood. Cyberdesk: a framework for providing self-integrating context-aware services. In *Proceedings of the 3rd international conference on Intelligent user interfaces*, pages 47–54. ACM Press, 1998.

[136] Anind K. Dey, Peter Ljungstrand, and Albrecht Schmidt. Distributed and Disappearing User Interfaces in Ubiquitous Computing. In *CHI '01 Extended Abstracts on Human factors in Computing Systems*, pages 487–488. ACM Press, 2001.

[137] B. Brumitt and S. Shafer. Better Living Through Geometry. In *Proceedings of the Workshop on Situated Interaction in Ubiquitous Computing at CHI '00*, 2000.

[138] John R. Lewis. In the eye of the beholder. *IEEE Spectrum*, pages 16–25, May 2004.

[139] Chris Hunter. Jef raskin, macintosh inventor, looks to the future of computing. *Pacifica Tribune Online*, January 2004. Available at: `http://www.pacificatribune.com/Stories/0,1413,92~3247~1920582,00.html`.