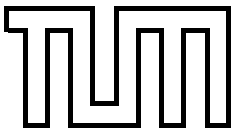
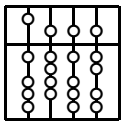


Technische Universität München
Fakultät für Informatik

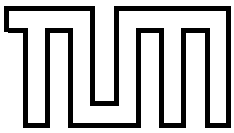


Diplomarbeit

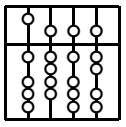
Design, Prototypical Implementation and Testing of a Real-Time Optical Feature Tracker

DWARF: Distributed Wearable Augmented Reality Framework

Martin Wagner



Technische Universität München
Fakultät für Informatik



Diplomarbeit

Design, Prototypical Implementation and Testing of a Real-Time Optical Feature Tracker

DWARF: Distributed Wearable Augmented Reality Framework

Martin Wagner

Aufgabenstellerin:	Prof. Gudrun Klinker, Ph.D.
Betreuer:	Dipl.-Inform. Thomas Reicher
Abgabedatum:	15. Februar 2001

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Februar 2000 Martin Wagner

Zusammenfassung

Die neue Technologie der Erweiterten Realität (engl. Augmented Reality) versucht, reale und virtuelle Umgebungen zu kombinieren. Dies geschieht üblicherweise durch visuelle Erweiterung der Benutzerumgebung mittels brillenähnlicher Head Mounted Displays. In das Display projizierte virtuelle Objekte sind für den Benutzer scheinbar ortsfest. Es ist sogar möglich, mit diesen virtuellen Objekten zu interagieren.

Das am Lehrstuhl für Angewandte Softwaretechnik der Technischen Universität München durchgeführte DWARF Projekt versucht, Methoden des Software Engineering zu nutzen, um die Entwicklung und das Testen neuer Anwendungen der Erweiterten Realität zu beschleunigen.

Eine überaus kritische Aufgabe der Erweiterten Realität und damit des DWARF Projektes ist die Bestimmung der Position und Orientierung, d.h. Blickrichtung, des Benutzers. Diese Parameter müssen mit hoher Genauigkeit, hoher Aktualisierungsfrequenz und geringer Verzögerung ermittelt werden, um passende Überlagerungen realer und virtueller Objekte zu garantieren. Während der letzten Jahre verlagerten sich die hauptsächlichen Forschungsaktivitäten auf dem Gebiet der Ortsbestimmung zu sogenannten optischen Trackern, die auf von Videokameras gelieferten Bilddaten basieren.

Optisches Tracking kann in die Erkennung von Markierungen und die Ortsbestimmung anhand der bei der Markierungserkennung erlangten Daten zerlegt werden. Für beide Teilaufgaben existieren mehrere Ansätze zur Lösung. Zusätzlich kann die in einer Sequenz von Videobildern enthaltene Information dazu benutzt werden, die relative Bewegung der Videokamera von einem Bild zum nächsten abzuschätzen.

In dieser Diplomarbeit wird ein neuer Ansatz vorgestellt, der Information über Relativbewegungen mit Information aus der Markierungserkennung kombiniert. Dieser Ansatz wurde prototypisch implementiert und getestet. Nach weiteren Entwicklungsarbeiten könnte durch ihn Ortsbestimmung ohne künstliche Markierungen ermöglicht werden. Dies würde zu neuen Anwendungsgebieten der Erweiterten Realität führen.

Abstract

Augmented Reality is a new technology that aims at combining real and virtual environments by means of usually visual augmentation using head mounted displays. Virtual objects projected into the display delude the user into believing that they are fixed at real objects and even can interact with the latter.

The DWARF project is conducted at the Chair for Applied Software Engineering of the Technische Universität München and tries to use methods from software engineering to speed up the development and testing of new Augmented Reality applications.

One of the crucial tasks of Augmented Reality and therefore the DWARF project is the estimation of the user's position and orientation. These values have to be determined with high accuracy, high update rates and low delay to enable good overlays of real and virtual objects. During recent years, the focus of the tracking community shifted towards optical trackers relying on data from video cameras.

Optical Tracking can be decomposed in marker detection and pose estimation out of the data obtained in the detection step. Several approaches are possible for both steps. In addition, the information given from the sequence of video images can be used to estimate the relative movement of tracked objects.

The main contribution of this thesis is a new approach of combining the information about relative movement with marker detection. This approach has been prototypically implemented and tested. If it is developed further, it will be an enabler for markerless tracking that allows to use Augmented Reality in new areas.

Preface

Background This thesis has been written to document the development and background of an optical tracker and some auxiliary components for Augmented Reality. This tracker is part of the DWARF project conducted in the second half of 2000 at the Chair for Applied Software Engineering at the Technische Universität München. I worked together with six other Master's students to design and implement a first version of a new Distributed Wearable Augmented Reality Framework. In this thesis, I would like to give a general introduction to optical tracking and discuss specific details necessary to understand the tracker I developed.

Introduction to Optical Tracking Tracking is an important topic of computer science that combines areas such as geometry, numerical analysis and computer vision. The first part of this thesis treats most of the background information that has to be known to understand and successfully implement an optical tracker that allows the three-dimensional estimation of the position and orientation of the tracked object.

Description of the Implemented Tracker One of the goals of the DWARF project was to use methods from software engineering for the development of Augmented Reality applications. They proved to be successful even for a leading-edge research field such as optical tracking. I will describe the implementation of the Optical Tracker using standard techniques from software engineering such as UML diagrams.

Discussion of Actual Implementation There is no specific chapter that describes all implementation issues. If you are only interested in understanding the code I developed, you should read chapter 3 for an in-depth discussion of the World Model and chapters 5.3, 5.4, 6.3, 7.3, 8.1.2, 9 and 10 for a description of all parts of the Optical Tracker. Chapter 11.1 gives some information on our implementation of a testing environment for the optical tracker.

Acknowledgements I am indebted to Prof. Gudrun Klinker, Prof. Bernd Brügge and Thomas Reicher for making this thesis possible and giving helpful advice, guidance and support during the last months.

Without the members of the DWARF team, Martin Bauer, Asa MacWilliams, Florian Michahelles, Christian Sandor, Stefan Riss and Bernhard Zaun, this thesis would not have been possible. It has been a pleasure to work with you guys!

I thank my family for encouragement, giving advice and proofreading.

And finally I would like to thank Susanne Hübscher for giving support and being there whenever it was necessary.

Contents

List of Figures

1 Introduction

We have become used to exponential growth in computing technology. Every 18 months, computing power doubles at a constant price. This has enabled thousands of new applications of computers in fields that were unthinkable a decade ago. One of the best examples for this development is the rapid growth of so-called dot-com-companies struggling for market share in small sectors of electronic business.

However, the basic principle of all these Internet-based companies remains the same. They try to connect huge databases to clever user interfaces in order to facilitate the way we, their costumers, buy or sell things. Although most companies claim they have new, exciting products, in fact they use rather old-fashioned technology to sell ordinary products.

Let us take a trip back to history. Forty years ago, in the 60s, many people believed that it would take only some decades for computers to become as intelligent as man. Indeed, artificial intelligence made huge advantages at that time. The only problem was that the time span for computers to become intelligent remained a constant of several decades. One of the fields of artificial intelligence is computer vision, the science of making a computer "understand" still or video images. This knowledge may be used to gather information about its surrounding like the people around it or to compute its current position.

Up to now, most computer vision people have used large desktop computers in order to get the necessary processing power to work with video images in real time. Fortunately, during the last years, progress in technology has made the "operations per pound" index so large that it now is feasible to think of and try out so-called wearable computers for computer vision tasks.

This thesis describes a new approach to the determination of a computer's position out of a live video feed that minimizes the necessary computing power by combining several known methods of computer vision. It has been used as part of the Distributed Wearable Augmented Reality Framework (DWARF) project conducted at the Chair for Applied Software Engineering at the Technische Universität München.

1.1 What is Augmented Reality?

As we mentioned above, this thesis is part of a project to develop an Augmented Reality software framework. The first question arising is: What is Augmented Reality? Perhaps you have heard of Virtual Reality (VR). The user of such a system is surrounded by a completely virtual scene and moves around in this obviously restricted environment. Common examples of VR systems are flight simulators used to train pilots or sophisticated computer games.

In short, with a VR system the user is taken away from the real world to a computer generated one.

Augmented Reality (AR) aims now to leave the user in the real world and only to *augment* his experience with virtual elements. Note that although the rest of this thesis just deals with visual augmentations, other means of augmentation are thinkable, such as sound, tangible devices and so on.

Azuma [12] gives an overview of the possible applications of Augmented Reality and tries to specify the core properties of each AR systems. He defines

> *AR as systems that have the following three characteristics:*
>
> 1. *Combines real and virtual*
> 2. *Interactive in real time*
> 3. *Registered in 3–D*

Let us note here that although this definition is very broad, most researchers have concentrated on visual augmentation during the last years. This can be seen by looking at the proceedings of ISAR 2000 [5] or IWAR 1999 [4]. An excellent example of this visual augmentation is given by Kato et al.[38] and shown in figure 1.1.

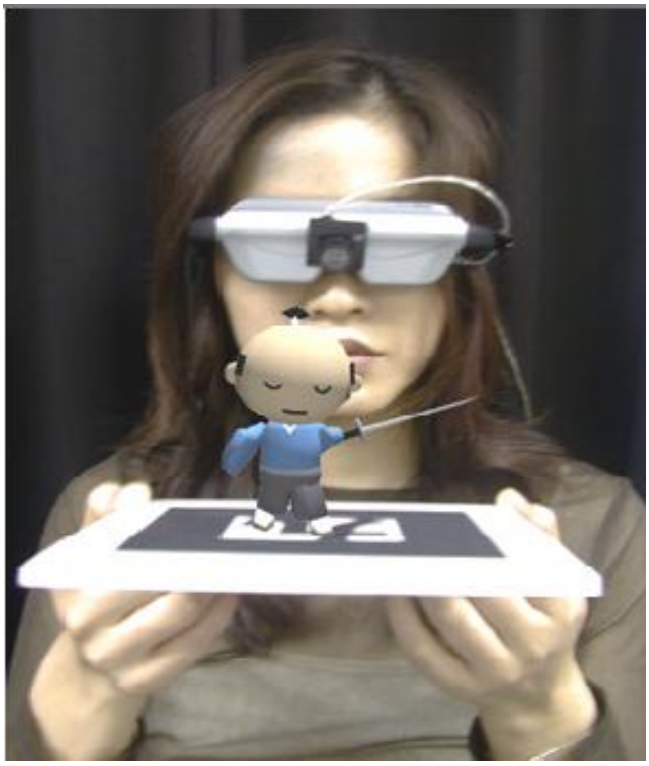


Figure 1.1: An Example of Visual Augmented Reality (Courtesy of Kato et al.)

Visual augmentation works in two possible ways:

Video see–through mode. An image of the real world is recorded by a camera, some analysis is done on it and finally some parts of the video image are changed in a way that adds information. The resulting video stream is displayed either on a monitor or a head-mounted display (HMD) worn by the user. The main advantage of this approach is that the AR system knows exactly what the user sees. However, the disadvantage of viewing the real world in VGA resolution imposes restrictions on the usability of such systems.

Optical see–through mode. The position of the user is determined by a tracking device. Afterwards, visual augmentations are computed that fit for his current position. Finally, these augmentations are displayed in a see-through head-mounted display (STHMD) worn by the user. The main advantage of the user seeing the real world without any modifications is opposed by the disadvantage of the need of exact calibration to align real and virtual objects (see [65] for a recent paper on this topic).

For the DWARF project, we decided to use the optical see-through technique. This thesis mainly describes the development of an optical tracker using video input from a camera mounted on the user's head in order to determine the camera's (i.e. the user's) position and orientation in real time.

1.2 Outline of the Thesis

Although the results of an Augmented Reality system may seem simple (some tiny pixels on a small head-mounted display), the technologies necessary to obtain them are not. Many different tasks from a wide range of computer science and mathematics have to be performed in order to get an AR system working.

The goal of this thesis is on the one hand to give an overview of how to build an AR system in general and on the other hand to give specific details on the implementation issues necessary for an optical tracker. Its outline follows this strategy and is shown in figure 1.2.

We start in this chapter by giving a definition and a general overview of the tasks necessary for Augmented Reality. Once we have analyzed the overall procedure, we can describe the other components of the DWARF system in chapter 2.

One of these components is the central database containing all necessary information about the real and virtual environment of the user. This database is described by a software engineering point of view in chapter 3.

After this brief description of the software environment, we can start analyzing the task of tracking and calibration in chapter 4. In chapter 5, we will see that optical tracking can be decomposed in image analysis (chapter 6) and pose estimation.

For pose estimation, two possible approaches will be described: one is based on the observation of a single frame of the video image and is shown in chapter 7. The other one uses information derived from subsequent video frames to get faster results. We will describe this problem in chapter 8.

The main contribution of this thesis is a new approach of pose estimation combining relative and absolute tracking techniques. The idea behind this approach will be described in chapter 9.

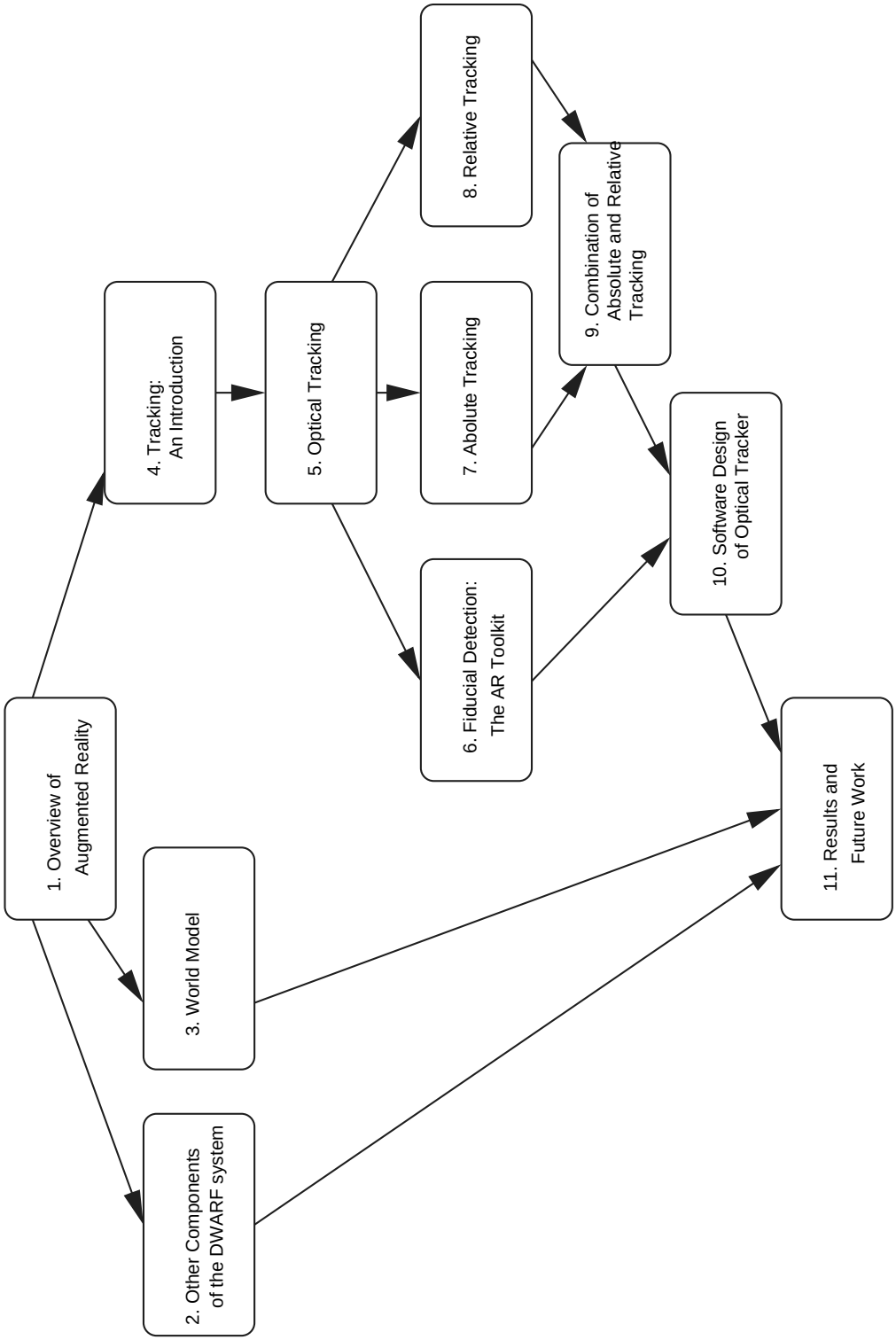


Figure 1.2: Overview of Logical Structure of Chapters

All this knowledge about tracking has been combined by implementing a real optical tracker. The design and the implementation issues of this program will be discussed in chapter 10.

Finally, we will give some results and an interpretation of these results in chapter 11.

Note that the dependency between the content of the various chapters can be modeled as a graph and is shown in figure 1.2.

1.3 Tasks Needed for Augmented Reality

In this section, we will try to give a short impression of all the work that has to be done to build an Augmented Reality application. For this analysis, we will use Azuma's definition (see section 1.1) of AR.

To ensure the registration in 3–D requirement, we have to track the user's position, and sometimes his orientation as well. This task is performed by so-called *Trackers*. We will see in chapter 4 which devices can be used as trackers. For now, we assume the trackers being a black box that outputs the user's position. Note that we may want to track not only a single user for multi-user environments. In addition, it may be advantageous to keep track of the position of some objects the user wants to interact with.

Next, we have to use this information to combine real and virtual worlds. If we spend some time thinking on how to organize this, it is obvious that we need some data structure to store all necessary information about the user's real and virtual environment. We call this structure a *World Model*. A central application takes the data stored in this World Model and combines it with the data delivered by the trackers.

The result of this combination is used to augment the user's perception of the real world. In consequence, we need some output device. Because of the real-time interaction requirement, we may have to combine this output device with a complete *User Interface* to enable the user to interact with the AR system. For the case of visual augmentation, the main task of the output device is to show virtual objects that are registered in 3–D. This may be performed by standard techniques such as OpenGL or VRML.

You may have noticed that we just described a classical model-view-controller (MVC) architecture. Figure 1.3 shows the structure more clearly.

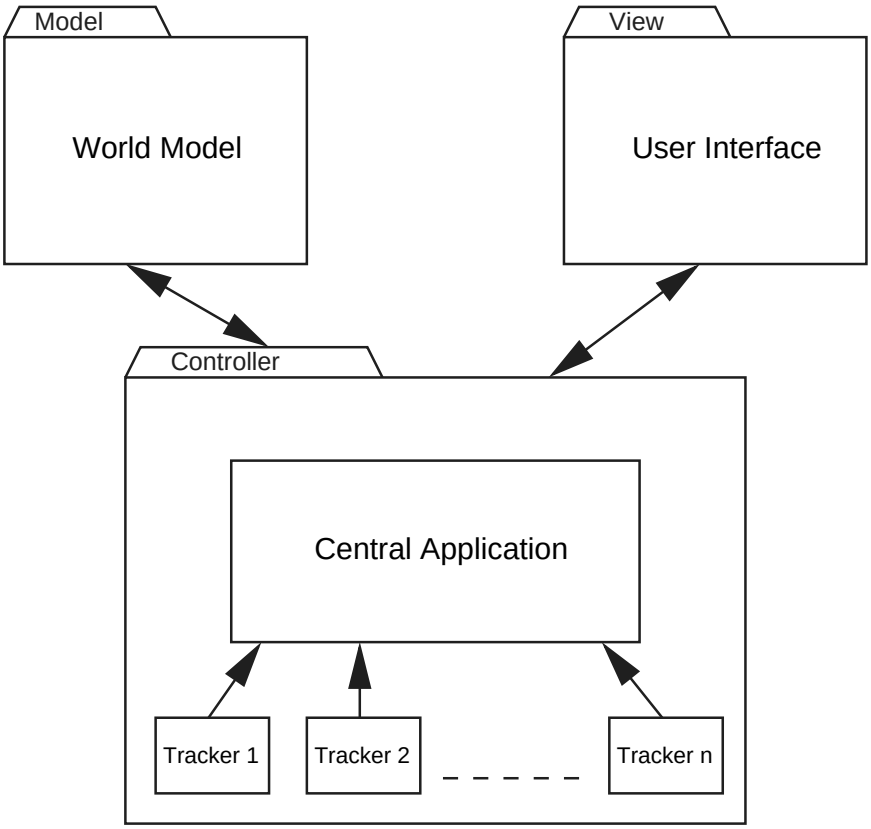


Figure 1.3: MVC Architecture of AR Systems

2 The DWARF Project

Recall figure 1.3. Developing an AR application does not seem too simple, however, we have made a first step by decomposing the problem. What would happen if we used methods from software engineering to define standard interfaces between the components just derived? Could we modularize the AR system?

These questions were the starting point of the *Distributed Wearable Augmented Reality Framework (DWARF)* project ([1]) we started at the Technische Universität München in May 2000. In this chapter we will give an overview of the project's background, its requirements and the general architecture. As a first test of the DWARF ideas, we created a demo application based on the framework components implemented up to now. To conclude this chapter, we will analyze the architecture and walk through the components of the demo.

2.1 Background

Almost all AR systems described up to now are highly specialized monolithic programs (see [8], [9], [35], [11], [59], [61], [16], [38], [12], [40] for details). This is due to AR being a young field of research. Each of these systems handles a small demo setup and tests a particular technology like tracking, calibration or human computer interaction.

For the purpose of specialized research this approach is well suited. However, each of these systems runs into trouble if the requirements—functional or nonfunctional—are changed.

The main idea for the DWARF system is to solve this problem by combining methods from Software Engineering with technology from Augmented Reality. The goal of DWARF was to build a software framework that is extended easily by having clearly defined interfaces between several so-called *Services*.

If such a system could be designed, several people could easily work together on it. Each person can modify, reimplement or even add a single subsystem like a different tracking device without having to worry about interaction issues with the other modules. See [46] for a detailed discussion on the use of software engineering in the domain of Augmented Reality and wearable computing.

The principles for the development of the DWARF system did not differ from those used for "normal" software projects described in [18]. The rest of this chapter follows the structure given by this book. We start with the description of the *Requirements Elicitation*, go on with the *System Design* and finally give a short overview of all DWARF components implemented up to now.

The subsequent sections are a short summary of the much more detailed Requirements Analysis Document (RAD)[14] and System Design Document (SDD)[15] that have been written during the project.

2.2 Requirements Elicitation

Note: The Requirements Elicitation is joint work with Martin Bauer, Asa MacWilliams, Florian Michahelles, Christian Sandor, Stefan Riss, Bernhard Zaun and Thomas Reicher. This summary of the requirements analysis has been written by Martin Bauer and appears as well in his thesis and in Asa MacWilliams' thesis in slightly modified form.

2.2.1 Functional Requirements

The framework we propose is designed as a collection of services that can be combined flexibly to build an efficient and wearable Augmented Reality system. These services can run on separate hardware components that are then networked together to provide the desired functionality. Using the framework to build an Augmented Reality application thus involves selecting the required services and hardware components and configuring their interaction.

The services are designed to be as independent of one another as possible, so an augmented reality system with limited hardware resources can be built that uses only a few of the framework's services. On the other hand, the framework is also designed to scale upwards with increases in available computing resources.

The framework provides services that range from low-level image processing for optical tracking to high-level interpretation of world, user and task models. An application using the framework provides these world and user models and defines the logical interdependencies between events in these models and output to the user—the framework takes care of the rest. Of course, an application can also access the low-level services directly, if this is required.

2.2.1.1 Tracking

The framework contains support for selected tracking hardware devices as well as the possibility to use additional devices the same way. The output of the tracker is accessible for the application including necessary quality of service parameters.

The framework also includes methods for post processing and combining the output of the used tracker and an interface for using and testing of new filtering and prediction algorithms.

2.2.1.2 Presentation

The user interfaces for applications built with the DWARF framework are described in an abstract way in terms of functions, operations and messages. This description is then converted into actual interface elements by the currently running user interface devices. These devices allow multi-modal user interaction depending on the situation and the preferences of the user.

2.2.1.3 Middleware

The communication between different modules of the DWARF system is done using an event based mechanism. The event service provides a publish-and-subscribe mechanism for events, a mediator configures the flow of events between different subsystems and the system service manager is responsible for defining which services are part of the system and which are not.

2.2.1.4 World Model

The World Model stores all information the system has about its environment. All relevant objects of the real world the system is operating in are represented as objects in the World Model. In addition, all virtual objects the system uses to augment the user's reality are stored in the World Model. It should be possible to integrate and migrate between various World Models during runtime.

2.2.1.5 User Model

The system allows users to set personal configurations. Users are able to change, update or annotate information about the environment and their own profile.

The system provides task relevant information and services based on system and individual user context. Context is indicated either explicitly or implicitly by the user. The system is aware of its entities' locations, identities, activities and times.

2.2.1.6 Task Model

The task model contains a description of the tasks the user has to do or wants to do. It receives events from the application when a step is performed and then generates the user interface for the next step.

2.2.2 Nonfunctional Requirements and Pseudo Requirements

The DWARF system development is starting completely from scratch. Nevertheless there are some constraints that have to be taken into account during the design phase.

2.2.2.1 Resource Issues

The entire system allows users to be mobile. Mobile users can access the system by small wearable devices. The system has to deal with changes in connectivity, bandwidth and error patterns, while users move from one area to another. Moreover, mobile users are subject to different security policies as they are connected to different domains.

2.2.2.2 User Interface and Human Factors

The user interface has to be multi-modal and as far as possible not restricting the user during his normal work. The applications using the DWARF system should not require the user to know anything about the internal structure of the DWARF system.

2.2.2.3 Hardware Considerations

The whole system will be deployed on several modules. Modules encapsulate functionality and are independent to a certain extent. The framework allows to develop and test these modules independently from each other. Furthermore, modules can be extended, modified or substituted by new editions. Conceptually every service is mapped to a single special hardware component. Hardware components may be added or removed at runtime.

2.2.2.4 Quality Issues

As the major goal of the project is the development of a framework, industry quality does not have to be reached on the implementation level. On the design level however the overall structure, interfaces, and interaction should be frozen after the first development stage to ensure the immediate usability of the design.

2.2.2.5 Security Issues

User and system service authentication and authorization have to be investigated as well as encryption. Due to performance reasons the system subcomponents should be able to decide upon the security level used. To ensure the fulfillment of this requirement, a hierarchical security model should be provided.

2.2.2.6 Documentation

The primary focus is on design rather than implementation issues. In consequence, the whole design process has to be documented, including the design rationale.

In order to provide a proof of concept, a working prototype using all parts of the framework's functionality has to be implemented.

2.3 Related Work

Note: The study of related work is joint work with Martin Bauer and Asa MacWilliams. It appears also in their theses in slightly modified form.

The DWARF system relies on many fields of ongoing research in computer science. It aims at combining wearable computing with plug-and-play distributedness, easy-to-use human-computer interfaces and advanced tracking technologies. In this section, we will have a brief look at some research projects that are related to the DWARF project and deal with similar problems.

2.3.1 MIThril

MIThril [48] is a context aware wearable computing platform that is currently developed at the MIT Media Lab.

The primary focus of the MIThril project is on the development of a wearable infrastructure using small RISC-processors with low power consumption like the Intel StrongARM, a single-cable power/data connection between the distributed components and a high-bandwidth data connection to the infrastructure of the surrounding.

MIThril Overview. The design of the system can be split into four distinct classes of components. The overall goal for each component class was to provide small, lightweight devices with low power consumption.

The *MIThril computing cores* are the base of the system design. These cores use small RISC-processors with extremely low power consumption to perform all necessary computing tasks. They communicate with one another using an Ethernet connection. All computing cores run a full operating system, currently Linux is used.

The *MIThril body networks* consist of a Body Network and a Body Bus. The Body Network connects the computing cores with one another using a standard twisted pair Ethernet cable modified to include power lines. The Body Bus is used to connect external devices such as microphones or cameras to the computing cores. It consists of two data buses, USB and I^2C and a power supply, all combined into a single cable that can be branched easily.

The *MIThril peripherals* are connected to the system either using the Body Bus or special connectors on some computing cores. Many cheap off-the-shelf components such as microphones, input devices or webcams can be connected via the Body Bus and its USB component. Devices with more complicated interfaces that would be restricted by the 12MBps constraint of the USB bus such as head mounted displays are connected directly to special computing cores.

Finally, the *MIThril software* currently consists of special versions of Linux running on all computing cores. Special drivers have been developed to support the body networks and special hardware such as display drivers for head mounted displays. On top of this basic operating system runs the application. Up to now, almost no real applications have been implemented.

Discussion. MIThril seems to be a perfect match for the DWARF project. It focuses on hardware and operating system issues and can therefore be an excellent basis for really wearable DWARF systems focusing on software issues. The concept of the body network allowing single cable connections between the various components is very promising and should be explored in the future. As the developers of the MIThril project intend to publicize their circuitry and device driver code, it could be a good starting point for extremely small DWARF systems.

2.3.2 UbiCom

The *Ubiquitous Communications (UbiCom)* program is a multidisciplinary research program at Delft University of Technology. The program aims at carrying out research needed for specifying and developing wearable systems for mobile multimedia communications [66, 42].

UbiCom Overview. The basic system architecture combines (non-self-sufficient) mobile units with stationary computing servers. Thus, the research focuses on wireless networks and small mobile systems.

There are many hardware-oriented subprojects involved in this program, such as head-mounted display technologies, high-bandwidth wireless networking, and small wearable systems. One interesting result is the LART (Linux Advanced Radio Terminal) [43], a small yet powerful embedded computer capable of running Linux. Its performance is around 250 MIPS while consuming less than 1 Watt of power. In a standard configuration it holds 32MB DRAM and 4MB Flash ROM, which is sufficient for a Linux kernel and a sizeable ramdisk image.

An important software issue is quality of service, which led to the design of a dedicated quality-of-service architecture [44]. This involves algorithms for predicting network use and protocols for reserving network capacity. As of yet, these are only implemented in a simulator, and there does not appear to be a software framework for actually managing the network connections. Another subproject [68] aims to make the mobile systems adapt their behavior to changing availability of resources.

A third area of research is aimed at the design of multimodal human-computer interfaces, but this is still in the design stage.

Discussion. Ubicom is a broad research project investigating many aspects of building mobile multimedia systems. The program does not aim to build a framework or single system, but rather to develop basic technologies.

2.3.3 jAugment

The *jAugment* project at the computer science department of the University of Rostock, Germany tries to develop a set of applications to be used with different kinds of wearable computers.

jAugment Overview. The jAugment applications take advantage of the unique features of wearable computers. Examples that are given include text editors, email clients and MP3 players as well as a path-finder, street maps and a scheduler for trains and busses.

The applications are classified by their purpose into three groups: *Infrastructure applications* provide interfaces, helper-classes, user interfaces and central services. *Everyday applications* are all the small things that a computer in general is expected to do, but that get a bit complicated in case of a wearable computer. The last group are *special purpose applications*, that only make sense on a wearable computer.

Discussion. The project does feature dynamic adding and removing of input and output devices as well as dynamic access to services. But the applications are basically stand-alone applications that need to bring all their resources with them. There is no direct support for reuse of ressources between different applications, when it is not specially implemented together with the application; the overall architecture of the system is centered around the user interfaces.

2.4 System Design

Note: The System Design is joint work with Martin Bauer, Asa MacWilliams, Florian Michahelles, Christian Sandor, Stefan Riss, Bernhard Zaun and Thomas Reicher. This summary of the System Design Document has been written by Asa MacWilliams and appears as well in his thesis and in Martin Bauer's thesis in slightly modified form.

This section briefly outlines the DWARF system design. Further details can be found in [15].

We stated above that DWARF is designed as a collection of services that can be combined to build an AR system. To avoid confusion, we should point out that the term *service* can mean different things.

DWARF services are the components of the framework, such as a GPS tracking service. These are often referred to simply as "services".

External services are not part of DWARF, but can be accessed using DWARF services, e.g. external printers.

System services are used internally by the components of the framework, e.g. network or middleware services.

Using the framework to build an AR application involves selecting the required services and hardware components. The DWARF middleware will then connect the services together that depend on each other. For simple systems, the application will not have to configure the services' interaction at all, since they find each other automatically. This is even possible dynamically, so that new services can be integrated into a running system as they are found. For more complex systems, the application will need to configure the service's interaction appropriately.

2.4.1 Subsystem Decomposition

Augmented Reality is all about bringing information and things together for the user. Accordingly, DWARF provides services to model things, access information, bring these things together and present them to the user. In addition, DWARF lets the user access external services in the environment. See figure 2.1 for an overview.

The application is generally "shielded" from the low-level devices, such as user interface devices, tracking services, external services and so on. It can access these at a high level of abstraction using the various DWARF services.

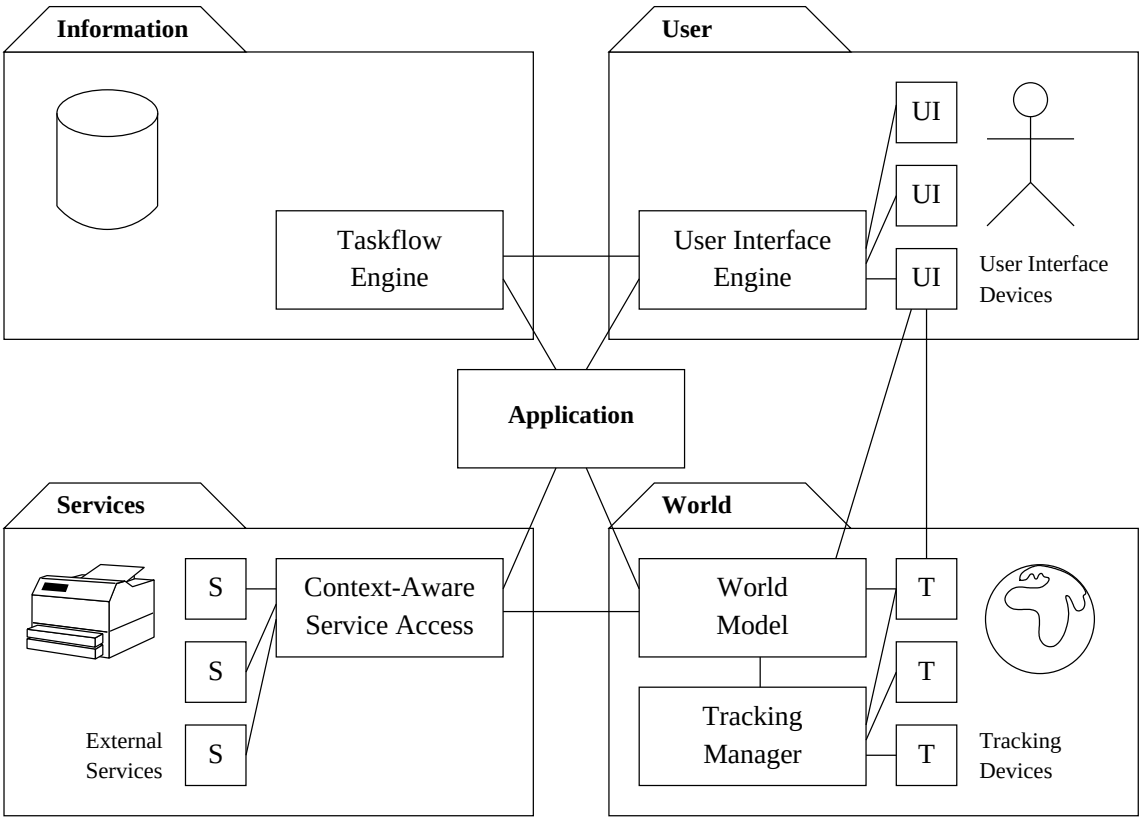


Figure 2.1: DWARF System Design. The DWARF services are conceptually divided into four different areas of functionality. Note that there are connections between the services that bypass the application. For example, the combination of the World Model, a user interface device and a tracker can correctly register virtual objects in the real world.

Modeling the World and Things in it For Augmented Reality to work, the system must have an idea of where the things in the real world are and what they look like. For this, DWARF includes several different *Trackers*, which can establish the position of things, and a *World Model*, which can store position information and other attributes of real and virtual things.

Accessing Information Augmented Reality can display information about things to the user that the user could otherwise not see easily. One type of such information is instructions on how to deal with the things, such as maintenance instructions or geographical directions. The information associated with things in the real world often involves a sequence of tasks, such as in maintenance or in navigation. DWARF models this sort of dynamic information in the *Taskflow Engine*.

Other models for information that can be presented to the user can be added to the framework in the future.

Interacting With the User This is the central part of Augmented Reality. DWARF provides several different kinds of *User Interface Devices* and a *User Interface Engine* that lets the application access these devices in a high-level fashion. This allows multimodal user interfaces to be created easily.

External Services Additionally, DWARF lets the user access external services in the environment (even those that are not part of DWARF), such as printers or display devices. To allow the selection of such services based on user preferences and context such as geographical position, DWARF provides a *Context-Aware Packet Service*.

System Services The low-level system services allow DWARF to function as a distributed system. This includes the *Network Service*, the *Communication Services* and the *Middleware*, which is used by all other DWARF services. These are not shown in figure 2.1—in a sense, they are omnipresent in the DWARF system.

Application The application is outside of the framework; it uses the framework to build a complete Augmented Reality system.

The application interfaces with the DWARF Middleware, and can access all of the DWARF services and configure them, if it needs to. A simple Augmented Reality application, however, can consist only of initializing the World Model and the Taskflow Engine, and the information associated with the tasks will be displayed to the user, registered with the real world in three dimensions.

Whenever application logic cannot be modeled by the framework's services, the application has to provide this itself.

2.4.2 Hardware/Software Mapping

The DWARF services can be distributed onto as many computers as is desired; the middleware will let them find each other, as long as they have a network connection. This allows computation-intensive services such as an optical tracker to run on dedicated hardware which can be added to or removed from the system at run time.

This way, the user can configure his mobile Augmented Reality system so that he never has to carry around more with him than is necessary.

DWARF takes advantage of existing software components where this is useful. For example, the user interface devices use existing VRML rendering and voice recognition software, and the middleware makes use of CORBA and third-party event services.

2.4.3 Persistent Data Management

Persistent data in DWARF is stored in various markup languages. This includes *VRML (Virtual Reality Markup Language)* for three-dimensional models, and various *XML (eXtensible Markup Language)* dialects for Taskflows, the World Model, user interface descriptions, context-aware service requests, and descriptions and configuration of DWARF services.

2.4.4 Global Software Control

Since DWARF consists of many cooperating services, there are many simultaneously running processes, and many threads of control. The DWARF services communicate with one another using the middleware's event service and remote procedure mechanisms.

2.4.5 Boundary Conditions

The framework's services can be started on demand by the middleware when other services access them. Thus, startup of the whole DWARF system is automatic. The same mechanism allows new services to be intergrated into the system on the fly, and replace services that have failed or cannot be accessed due to the loss of network connectivity.

2.5 Component Walkthrough

We have now seen the requirements elicitation and the system design. This work was the starting point of a much longer development process aiming at the implementation of a first prototype using the DWARF framework.

This prototype had to fulfill a demo scenario that involved all DWARF components developed up to now. This section deals with the description of the demo scenario and a short functional description of all DWARF components, including the ones described later in this thesis. The order of the components described is derived from their order of appearance in the demo.

2.5.1 Demo Scenario

For our demonstration system, we chose a rather complex scenario. A visitor is headed for a meeting at a room on the campus of the Technische Universität München, and his mobile AR system navigates him to the meeting room and lets him print out his handouts on the way.

The choice of our scenario was driven by three goals:

- The scenario should involve not only classical AR, but also the dynamic use of services,
- it should take place in different types of environment to show the flexibility of our framework,
- and it should use and test all of the DWARF components developed up to now.

Scenario: **Demonstration Scenario**

Actor instances: `Fred: User`

Flow of Events:

1. Fred is invited to a meeting with some software engineering students at the TU München. He is equipped with a backpack with two laptops, a head-mounted display with an attached digital video camera, a headset and microphone for voice input, a GPS/compass combination and a RFID tracking device. Fred has a PostScript handout on one of his laptops, and has already registered the handout to be printed as soon as he reaches the TU main building. The students Fred is supposed to meet with have told him to take the subway to the station Königsplatz.
2. Fred emerges from the Subway station and walks towards the exit. As he comes within reach of an information terminal on his way, an option appears on his display offering him to download personalized navigation instructions to the meeting room. He says "yes" to accept this data transfer. He sees a a message that the download is in progress. After a while a message appears, saying that the data transmission is complete.
3. On Fred's head-mounted display, a three-dimensional map of the area appears. It shows his own position with a red dot and rotates as he turns, showing his current orientation. A blue arrow indicates his destination. Fred uses this map to guide him to the entrance of the TU.
4. As Fred reaches the TU, an option appears to let him send off the print job for his handouts by wireless LAN. Fred confirms this by saying "yes" again.
5. Inside the building, Fred is guided by a schematic two-dimensional map, indicating which room he is currently in (the position is read from RFID tags), to the hallway outside of the meeting room.
6. Here, he sees a red arrow appear in his head-mounted display, pointing to one of two printers, which has printed his handouts.
7. Fred picks up his handouts from this printer, says "ready", and a three-dimensional blue arrow appears, pointing him to the meeting room.

8. Fred enters the meeting room, takes off his head-mounted display and backpack and greets the students.

2.5.2 System Architecture and Subsystem Communication

The system architecture is obviously based on the framework design depicted in figure 2.1. Note that most communication between the services is direct and does not use the application.

Subsystem Communication The DWARF subsystems, so-called *services* communicate using CORBA, the Common Object Request Broker Architecture. All services implement CORBA interfaces for communication with other services. This assures platform independence, as the definition of the interfaces can be transformed to most object-oriented languages as C++ or Java. Most of the communication is done via the CORBA Notification Service, an extension of the CORBA Event Service.

If a service wants to send or receive events, it has to register for it. An easy-to-use publish and subscribe mechanism is provided for this reason.

The implementation is based on free ORBs such as OmniORB for C++ or the JavaORB for Java. Further details on the subsystem communication can be found in [46].

2.5.3 Context-Aware Service Selection and Execution

Recall the description of the demo scenario. We mentioned that the user would like to print handouts on the way to the meeting room. This is basically the first task that happens in the scenario. It is handled by the *Context-Aware Packet (CAP)* service.

The major problem for this task is the user's current printer configuration. Even technically simple tasks such as printing can lead to huge problems in unknown computing environments, as a lot of contextual information such as the preferred paper size has to be regarded for successful execution.

The basic idea of the CAP service is now to encapsulate such information in packets that are further processed by software devices that route them in a suitable way. For the printing example, all of the user's configuration data, e.g. paper size, preferred color model etc., is stored in such a packet. The CAP router gathers all information necessary for an optimal fulfillment of the given task of printing from the other DWARF subsystems and executes the print job.

Further details on the CAP service can be found in [47].

2.5.4 Bluetooth Communication

The demo scenario involves an information terminal that allows the user to download location dependent data. Several requirements have to be fulfilled to make this possible:

1. The communication of the user's wearable computer with the information terminal should be wireless for easy handling.
2. The bandwidth of the communication channel should be sufficiently high to minimize download times.
3. Spontaneous connection should be possible to increase the usability even more. In an ideal world, the user gets all useful data automatically during he walks by the information terminal.
4. The devices should be cheap to facilitate widespread use.

Bluetooth [6] is a new industry standard for low range wireless networks that fulfills all of these requirements. We implemented the information terminal using this technology.

2.5.5 World Model

Once the user of the demo application got the data, the DWARF system has to store it in a well organized fashion. The first place to store all data describing the user's natural and virtual environment is the World Model service. It can be seen as a large database that holds entries for every real or virtual object. Examples for real objects are buildings, floors, furniture in rooms etc. Virtual objects may consist of virtual stickies attached to real objects or highlighting information such as virtual arrows to indicate the directions the user has to take.

The crucial point for all these objects is their three-dimensional position and orientation towards each other. The World Model service provides facilities that allow easy description and computation of these relations.

As almost all DWARF components rely on such data, the World Model is a heavily used component. In consequence, efficiency was one of the major design goals.

We will describe the World Model service in detail in chapter 3.

2.5.6 Taskflow Engine

The Taskflow engine stores and handles the remaining data transferred from the information terminal. The basic idea behind the development of the Taskflow engine was to provide an easy-to-use possibility for the description of linear flows of tasks or more general events.

The advantages of this concept arise immediately if we think of maintenance applications using Augmented Reality technologies. Most maintenance tasks are characterized by a fixed flow of steps that have to be performed one after another. Using the Taskflow engine, these steps can be described using a dialect of the *Extended Markup Language (XML)*.

Nevertheless, this feature is very useful for other application domains as well. We used it for the demo application to describe the navigational information. Internally, the Taskflow engine may be seen as a state machine that switches to new states if it is triggered by certain incoming events. For navigation, every state is a textual or graphical description of a navigation task, e.g. "Go up the stairs" or an image of the stairs that have to be taken. The events are

the updates on the user's position and orientation. By evaluating them, the Taskflow engine realizes the necessity to switch to new navigation task descriptions.

Further details on the Taskflow engine can be found in [53].

2.5.7 User Interface Engine

All services described up to now did only process the data but did not provide any means of direct user interaction. All such tasks are encapsulated into the User Interface engine.

Its main task is to display and process the user interface scenes provided by the application or the Taskflow engine using multi-modal human-computer interfaces. The DWARF framework has been designed to support a large variety of application domains. In consequence, we can not rely on a fixed class of user interface devices such as head-mounted displays or usual computer terminals. It may even be possible that the output device changes during the runtime of an application.

To handle these constraints, the User Interface engine separates the description of the user interface from its actual instantiation. The input of the engine consists of a XML-based description of the user interface's functionality that does not contain much information about its final look and feel. This input is then transformed *(rendered)* to a concrete user interface displayed on a single or multiple available devices.

This approach allows high flexibility and reduced development time for highly platform independent AR applications. Further details can be found in [56].

2.5.8 Tracking Devices

Up to now, we have described components of a wearable system that knows absolutely nothing about any dynamic position data. Only if this knowledge is introduced into the system, we can talk of a real Augmented Reality system. As the DWARF system has been designed as an AR framework, we implemented a variety of rather different tracking devices and integrated the data obtained from them.

Simple Trackers This first class of tracking devices is characterized by giving less than six-dimensional data (three translational and three rotational components) necessary for real three-dimensional registration. However, advantages in availability, speed and reliability make them well suited for basic tracking needs.

A first device was a Global Positioning System receiver. Its output data consisted of three-dimensional translational data available only under open sky and one-dimensional compass data obtained from the earth's magnetic field. We used this receiver for outdoor coarse navigation.

We investigated a second promising tracking technology, radio frequency ID (RFID) tags. These passive tags are attached to known locations in the real world, like doors or significant points in hallways. A special active RFID Tag reader mounted on the user's wearable computer identifies their ID every time they pass by. In consequence, it is possible to obtain precise location information for a short period of time. Unfortunately it was not possible to

find a distributor of such tags in the narrow time frame of the project. As an alternative, we implemented a software simulation of the RFID tag tracker.

Further details and an extensive survey on simple trackers can be found in [13].

Optical Tracker An optical tracker processing live video input from a camera mounted on the user's head is one possibility to obtain realtime six-dimensional data of the user's position and orientation. This data is crucial for performing "real" AR applications.

The basic principle of the optical tracker we implemented is simple. The video stream is analyzed for markers that are attached to known locations in the three-dimensional world. As a result, correspondences between two-dimensional image points and three-dimensional real world points are established. Sophisticated algorithms are now used to compute the camera's six-dimensional pose out of these correspondences.

The advantage of high accuracy and update rates of the optical tracker is opposed by the extraordinary hardware requirements. Basically, a single computer has been used exclusively to provide sufficient computational power for the optical tracker.

Further details on the optical tracker are described from chapter 4 onwards of this thesis.

Tracking Manager The tracking manager has been implemented to provide transparent access to a variety of trackers. It seems reasonable to combine the output data of many tracking devices before sending it to other DWARF components that process the data.

In addition, facilities such as sensor fusion or movement prediction may be added to the tracking manager. In short, its task is to make the whole of the trackers more than the sum of the parts. Further details can be found in [13].

2.5.9 Summary

The development of the DWARF system has been highly modularized. However, it was possible to implement a working prototype within a short period of time. For setting up the demo application out of the existing components, no more than three weeks of work were necessary.

However, the development of the framework is far from complete. New components and missing functionality have to be added and existing components to be redesigned. Only if more applications will be developed using the framework it will be possible to increase its usefulness.

3 Representing the Real World: The DWARF World Model

We have now identified the general tasks necessary for an Augmented Reality system and described most of the components. If we recall figure 1.3, we see that the last chapter was about the Central Application and the User Interface. We still have to specify the tracking components and the World Model. The latter will be looked at in detail in this chapter.

Again, we will follow the structure described in [18] to ensure a well structured methodology that allows other developers to reuse the World Model's components.

3.1 Requirements Elicitation

Let us start with an analysis of the requirements for a World Model.

We have seen in section 1.3 that this DWARF component may be seen as the central database holding all necessary information about the real and virtual objects in the user's environment. As such, it is definitely not sufficient to take an ordinary database and impose a certain structure on it, as the specific requirements for AR systems have to be kept in mind and the World Model's interfaces have to be defined in a way that facilitates the development of AR applications.

3.1.1 Functional Requirements

The World Model stores information about the real and virtual objects that may be important for the user's interaction with the AR system.

As for most databases, there is no direct user interaction with the World Model. This service itself does not provide any output capabilities beyond debugging purposes. All access to the World Model is done via software interfaces.

Every real or virtual object has an associated position and orientation, its *pose*. To facilitate the development of applications, the World Model has to provide means to compute the pose of one object relative to an arbitrary other.

The objects stored in the World Model are highly variable. To handle this variability in a wide range of possible applications, it has to be possible to store an arbitrary amount of arbitrary information associated with each object.

In addition, it must be possible to change the World Model's content dynamically. As several DWARF services may access the World Model at the same time, there shall be mechanisms that allow consistent multi-threaded access. After every change to the World Model's

content, all services wishing to do so must be notified by an efficient event mechanism about the details of this change.

Finally, the World Model has to register for events that indicate the change of an object's position or orientation. Every such change has to be processed by the World Model and, if necessary, stored in the internal data structure.

The DWARF system is designed to be able to work in a large variety of different settings. In consequence, it may occur that more than one World Model service is present. This situation has to be handled.

3.1.2 Nonfunctional and Pseudo Requirements

Nonfunctional Requirements. The most crucial nonfunctional requirement for the World Model is performance. Many DWARF services access it simultaneously and some of them, e.g. optical see-through display devices, are constrained by hard real-time requirements. It is therefore necessary to ensure an extremely low response time.

As the DWARF system is intended to run on wearable computing devices, the World Model should be implemented in a way that ensures low memory consumption beyond the consumption of the data stored in the Model.

Pseudo Requirements. The World Model will be implemented in an object oriented language that allows seamless integration with CORBA for communication with all other DWARF services.

It has to be written in a way that ensures a maximum of platform independence, due to the fact that the Model will be part of every application using the overall framework.

To facilitate rapid prototyping and debugging of AR applications, it should be possible to specify data stored in the Model in a human-readable format.

3.1.3 Scenarios

Let us now have a look at some scenarios that may occur for the World Model service. Usually, *"A scenario is a concrete, focused, informal description of a single feature of the system from the viewpoint of a single actor"*[18]. Due to the World Model being an internal service of the DWARF system not interacting directly with the user, the scenarios described below are somewhat more technical than those for a complete system with user interaction.

Scenario: **Information Download**

Actor instances: `Alois: User`

Flow of Events:
1. Alois has an appointment at the TUM campus. He takes his wearable computer with a DWARF system running on it with him and exits the subway station next to TUM.
2. He walks to a public terminal that allows the download of information about the buildings at the TUM campus.

3. His application downloads a file containing this information about Alois' environment and hands it to the World Model.
4. The World Model service stores the data in its internal data structures and notifies the other DWARF services running on Alois' wearable.

Scenario: **Tracker Update**
Actor instances: `Erwin: User`
Flow of Events:
1. Erwin is wearing a DWARF application that includes a GPS tracker that determines Erwin's current position. He has been inside a building, so his tracker could not get his position.
2. Erwin is leaving the building. The GPS tracker gets signals from several satellites and starts sending position events.
3. The World Model service analyzes these events and updates the position of all objects that are moving with Erwin.
4. All other DWARF services are notified by the World Model service about these changes.

Scenario: **Optical Tracker Initialization**
Actor instances: `Hermine: User`
Flow of Events:
1. Hermine is wearing an AR helmet with a camera mounted on it. The camera is attached to a computer with an optical tracking program running on it. This program uses marker detection to compute Hermine's current position in real-time. Currently, Hermine is in a room with no markers in it. The tracker is idling.
2. Hermine enters a new room. A radio frequency tag tracker mounted in her backpack detects the ID of this new room.
3. The optical tracker gets notified and asks the World Model about the availability of markers suitable for tracking in this new room.
4. The World Model delivers information about markers and the optical tracker initializes itself with this data. It detects several markers and computes Hermine's position.

3.1.4 Use Cases

The scenarios described above allow us to derive use cases specifying further the flow of events of the World Model service. There are only two possible actors that can initiate a use case: the user of a DWARF system (`User`) or other components of the system (`External`). Figure 3.1 describes the relationship between the use cases described below.

Use Case: RequestData

1. *Entry Condition:* A DWARF service needs information stored in the World Model.

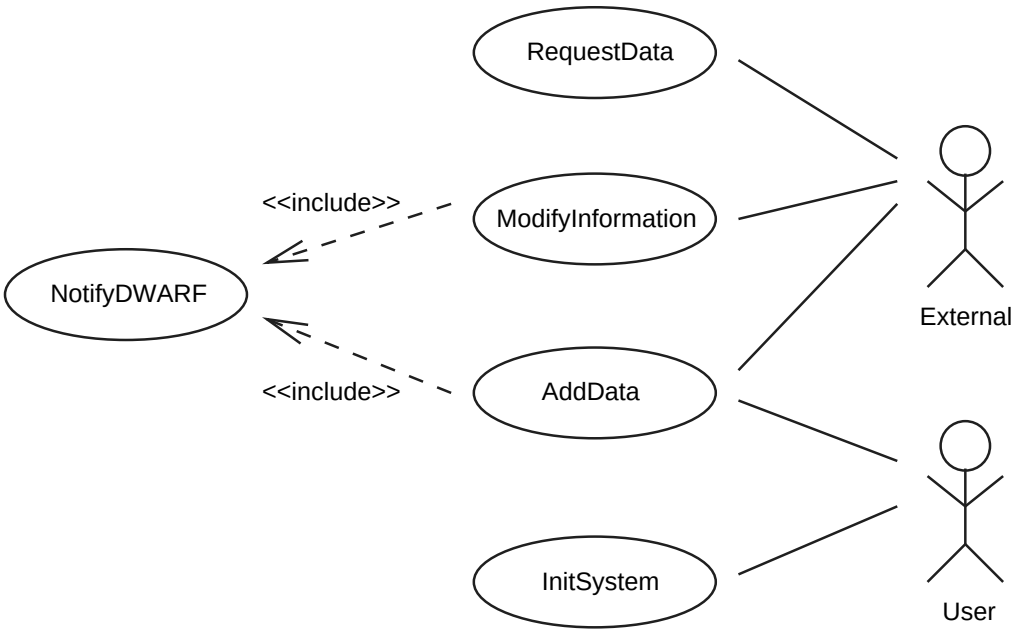


Figure 3.1: Use Cases Describing the World Model's Behavior

2. The external service sends an information request regarding a certain object to the World Model.
3. If this object does not exist, the World Model notifies the external service. Otherwise, the requested information is returned.
4. *Exit condition:* The DWARF service processes the information.

Use Case: NotifyDWARF

1. *Entry Condition:* Some data in the World Model has been changed, removed or added.
2. The World Model sends events on the DWARF event bus containing information about which objects are affected by the change and what has been changed.
3. *Exit Condition:* Other DWARF services use this information to update their knowledge about the user's environment.

Use Case: ModifyInformation includes `NotifyDWARF`

1. *Entry Condition:* Another DWARF service has detected a change in the user's environment.
2. The DWARF service either calls the World Model directly or sends an event to the DWARF system bus in order to insert the new information into the World Model database.
3. The World Model changes its data and uses `NotifyDWARF` to inform the other DWARF services.

4. *Exit Condition:* A consistent state of information in the overall system is maintained.

Use Case: AddData includes `NotifyDWARF`

1. *Entry Condition:* Either the user or another DWARF service gets a bunch of new information about the user's environment.
2. The caller invokes the World Model with this information.
3. The World Model adds the given data to its internal memory and uses `NotifyDWARF` to inform the other DWARF services.
4. *Exit Condition:* A consistent state of information in the overall system is maintained.

Use Case: InitSystem

1. *Entry Condition:* The user wants to start a DWARF system.
2. The user starts the World Model either with initial data or in an empty state.
3. The World Model initializes itself, registers to the DWARF system bus and reads the data given by the user.
4. *Exit Condition:* The World Model is in a valid state and can be used by other DWARF components.

3.2 System Design

Having seen the requirements for the World Model service, the system design follows straightforward. The core component of the information about a real or virtual object is its pose. Perhaps the single most important functional requirement for the World Model is the ability to compute one object's pose relative to an arbitrary other.

The system design of the World Model is centered around this capability. In the remainder of this section, we will explain the basic approach we chose to handle this problem and give some details about the subsystem decomposition and the persistent data management.

3.2.1 A Hierarchical Approach Towards Representation of Real and Virtual Objects

Every set of real or virtual objects can be grouped hierarchically in a tree data structure . To give an example, consider a table in a room at the TUM campus. The top-level object may be, at the DWARF system designer's choice, something like a map of Munich or a UTM coordinate system. One child of this top-level object should be the TUM campus. The coordinates of this campus may well differ from the UTM coordinates so we have to store rules how to convert the UTM system to the TUM campus system.

Again, the campus object has children. We may want to take every floor of every building at the campus as a child. These floor objects will then have single rooms as children. Finally, the table we are looking for is represented as a child of the room it is standing in.

Using this general structure, it is easy to add objects without knowing their position in the top-level coordinate system. If we add a virtual TV set to the table in our room, we only have to give the coordinate transformation from the virtual object to the table in order to allow the World Model service to compute the TV set's pose relative to every other object in the tree structure.

Mathematical Background Let us now discuss how to perform the computation just mentioned efficiently. The first question arising is, how can we represent an object's pose relative to its parent object's in a way that allows us easy inversion of this relationship?

Pose, in general, is represented as a combination of a three-dimensional *translational* vector

$$\mathbf{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \tag{3.1}$$

and an orthonormal 3×3 *rotational* matrix

$$\mathbf{R} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix} \tag{3.2}$$

with r_1, r_2, r_3 being the x, y and z axis of the object to be described in the root coordinate system. Figure 3.2 illustrates these values.

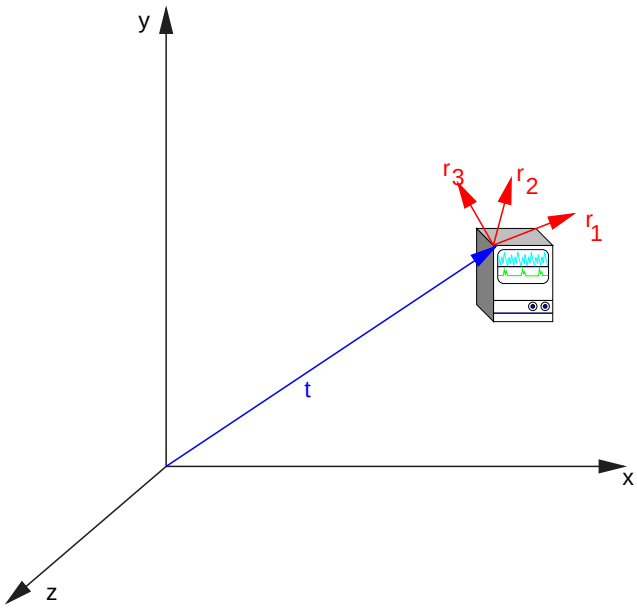


Figure 3.2: General Coordinate Transformation

If we now want to obtain the absolute coordinates $(x', y', z')^T$ of a point with coordinates

$(x_0, y_0, z_0)^T$ in our object's coordinate system, we have to do the following calculation:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \mathbf{R} \cdot \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + \mathbf{t} \tag{3.3}$$

Unfortunately, we do not have a linear equation here. Direct inversion (i.e. we have the point's absolute coordinates and want its coordinates in the object's coordinate system) is therefore not possible. To solve this problem, we have to move to *homogeneous coordinates*. We simply introduce a fourth value to every coordinate vector that allows us to combine the position vector and the rotation matrix to a non-singular 4×4 matrix

$$\mathbf{H} = \begin{pmatrix} r_1 & t_x \\ r_2 & t_y \\ r_3 & t_z \\ 0 & 1 \end{pmatrix} \tag{3.4}$$

and end up with the following relation:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \mathbf{H} \cdot \begin{pmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R} \cdot \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \\ 1 \end{pmatrix} \tag{3.5}$$

With $\mathbf{R}$ being an orthonormal matrix, it is obvious that $\mathbf{H}$ is always invertible. In consequence, we obtain

$$\begin{pmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{pmatrix} = \mathbf{H}^{-1} \cdot \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \tag{3.6}$$

by simple matrix inversion. Finally, the homogeneous coordinates allow us to do multiple coordinate system transformations by simple matrix multiplication. If we want to get a point's coordinates P_0 in system S_0 given the point's coordinates P_2 in system S_2 and the homogeneous transformation matrices $\mathbf{H}_1$ describing the transformation from system S_0 to S_1 and $\mathbf{H}_2$ describing the transformation from system S_1 to S_2, we just have to compute

$$P_0 = \mathbf{H}_1 \cdot \mathbf{P}_1 = \mathbf{H}_1 \cdot \mathbf{H}_2 \cdot P_2. \tag{3.7}$$

It should now be clear that the tree structure proposed above is well suited for arbitrary object pose systems if we use homogeneous coordinates.

3.2.2 Subsystem Decomposition

In the last section, we fixed the central data structure to be a tree. In this section, we will think about the overall software organization of the World Model service.

As we can see in figure 3.3, the structure is simple. We have one central object, the `WorldModel`. This object handles the initialization of the service, the CORBA communication and some high-level functionality as loading new information out of files. The actual data is stored in a set of `Thing` objects organized in a tree data structure. Each `Thing` object has exactly one parent `Thing` and an arbitrary amount of children. To facilitate the loading of files containing World Model entries, we provide an `XMLParser` object that encapsulates all necessary tasks for reading files as described in section 3.2.3.

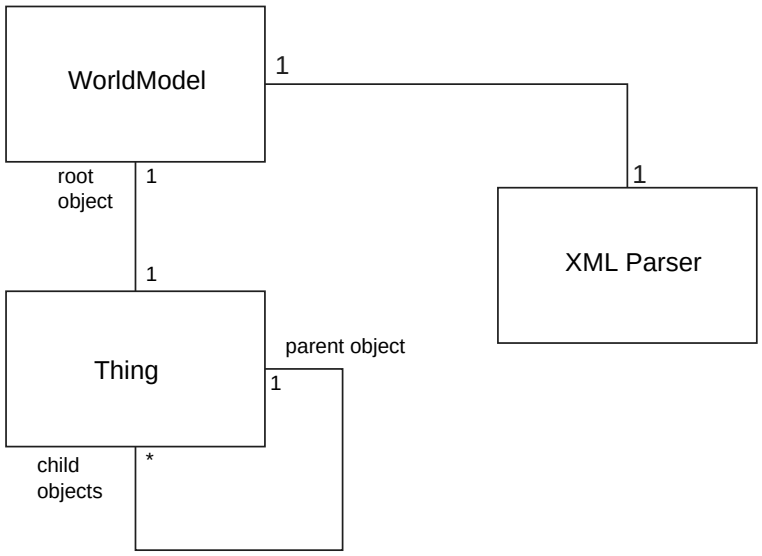


Figure 3.3: Subsystem Decomposition of the World Model Service

3.2.3 Persistent Data Management

In the current state of development, the World Model holds all data in memory at runtime. However, it may be necessary to add a large amount of data at some point in time, e.g. during startup or at a situation similar to the Information Download scenario.

It seems reasonable to create a possibility to hand a file of arbitrary size to the World Model that has the following properties:

1. Every type of information that can be stored in the World Model can be specified in the file as well.
2. The file should be readable by humans, it has to be possible to modify or create such a file with a simple text editor.
3. There has to be a possibility to add comment lines to the file.
4. It must not be possible to take the World Model in an inconsistent state by reading a malformed file.

A natural choice for these requirements is to use a variant of XML, the *eXtended Markup Language* [69]. With XML, it is possible to use a large variety of existing parsers that perform all error handling based upon a so-called *Data Type Definition (DTD)* that defines the syntax of well-formed documents describing content of the World Model.

3.2.4 Access Control and Security

The World Model contains most of the information stored in a DWARF system. If such a system is used in a multi-user environment, security issues must be respected. However, in the current implementation no such provisions have been taken, as no confident data is stored in the demo applications that are implemented using the DWARF framework up to now.

3.3 Object Design

This section documents the actual implementation of the World Model. We start with an UML class diagram and describe the main functionality of the World Model interface. Afterwards, we will show the structure of the XML dialect specified for World Model content. To make things more clear, we will show how to use the World Model in a DWARF system by giving a sequence diagram of an example scenario.

3.3.1 Class Diagram

The World Model may be seen from two sides. The one is the user's view with the user being a developer of a DWARF system. The only thing the user wants to know is a definition of the World Model interface to know how to access it. The other view is from the implementation side.

Figure 3.4 shows the class diagram of the World Model interface. This interface is accessible from all other DWARF components that communicate with the World Model via CORBA. It is defined using the *Interface Definition Language (IDL)* and shown in appendix C.

A detailed overview of all the methods and variables in the actual C++ implementation of the World Model can be derived easily from the source code. At the current state of implementation, heavy use of C++ standard template library (STL) features has been made. In addition, the XML parsing of configuration files is done with the standardized SAX (Simple API for XML) API. Portability is thus guaranteed.

In the remainder of this section we will give a short overview of the concepts that have been used for the definition of the World Model. Note that we will not give any implementation details. We refer the reader to the documentation in the source code contained in the accompanying CD.

Things and ThingIDs As we have seen in section 3.2, a `Thing` object is the core data structure used to store information in the World Model. However, we do not think it is a good idea to use copies of or pointers to `Thing` objects for referencing them across various

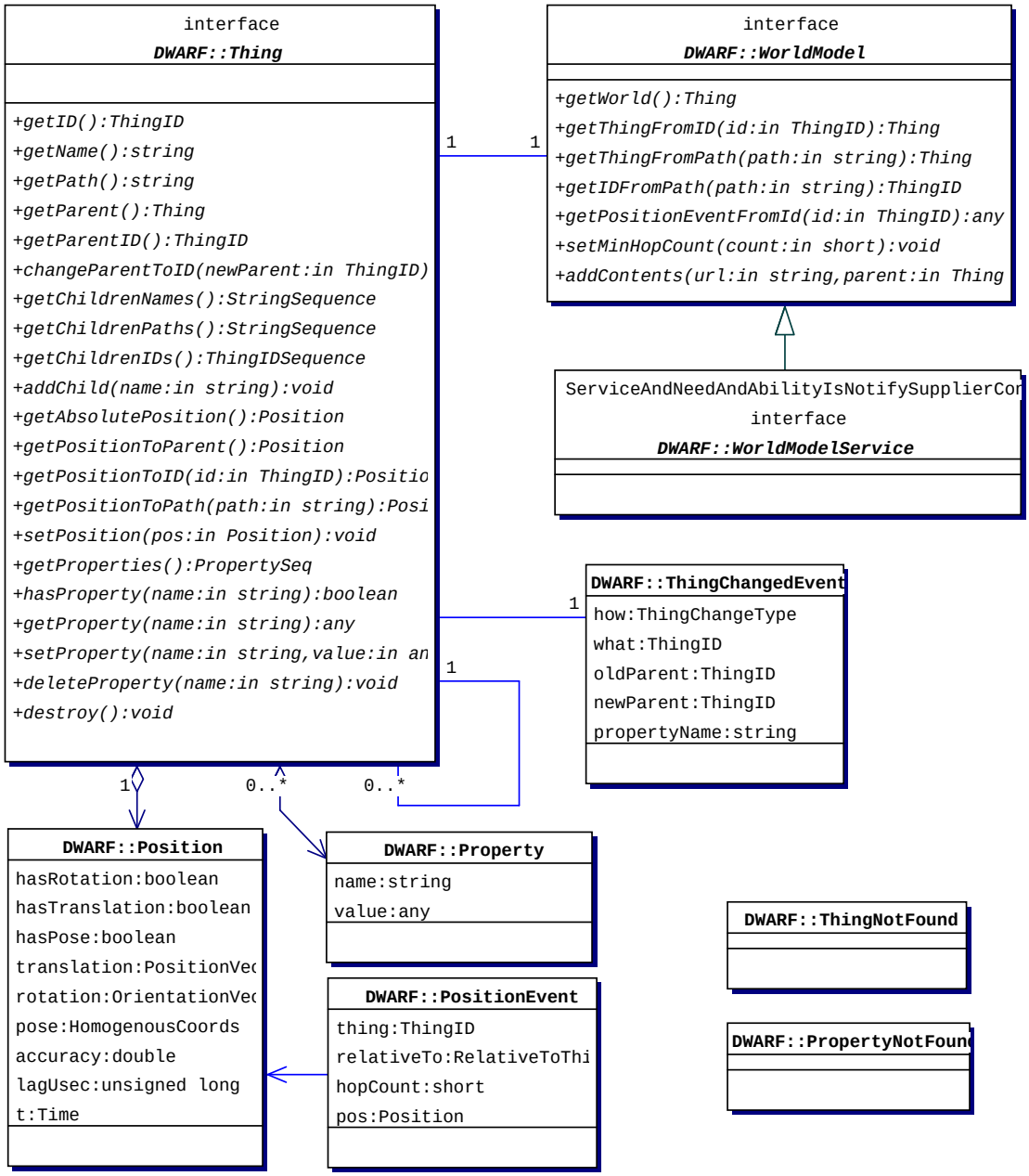


Figure 3.4: Class Diagram of the World Model Interface

DWARF services, as this approach is likely to lead to problems with platform independence and confusion for the programmer.

Instead, we use so-called `ThingID`s to reference `Thing` objects in a unique way. Every `ThingID` is unique. The central `WorldModel` object that handles all CORBA communication allows access to `Thing` objects using only these `ThingID`s.

Paths and Names Although the `ThingID` identifies a `Thing` object uniquely, it is not guaranteed to be the same for every instance of the World Model service. To allow an easy, human readable way to access arbitrary `Thing` objects, we represent the tree-shaped hierarchy of `Thing` objects by so-called *paths*. These paths are similar to file systems with the difference that every internal node holds information itself. We use the following simple rule to get a `Thing` object's path out of its name:

if $parent = nil$
  then $name = "/"$;
    $path = "/"$;
  else if $parent.path = "/"$
    then $path = parent.path + name$;
    else $path = parent.path + "/" + name$;
  fi;
fi;

WorldModel and Thing Objects As we stated in section 3.2, the central `WorldModel` object handles the initialization of service and the CORBA communication. The latter is described in [46], so we will not discuss details of the communication.

However, the reader should know the distribution of tasks beyond simple CORBA communication between the `WorldModel` object and the `Thing` objects.

In general, the `WorldModel` allows high-level access to specific `Thing` objects. It provides methods as `getWorld`, `getThingFromID`, `getThingFromPath`, `getPositionEventFromID` and `getIDFromPath` to either determine a `Thing`'s ID or position or get a reference to the actual `Thing` object. Note that `getWorld` is equivalent to `getThingFromPath`("/") and delivers the root object that is allocated during initialization of the World Model.

To get more specific details, the `Thing` interface offers a variety of methods. They can be grouped as follows and are described in detail in the source code:

General Information. `getID`, `getName`, `getPath`, `getParent`, `getParentID`, `changeParentToID`

Children Information. `getChildrenNames`, `getChildrenPaths`, `getChildrenIDs`, `addChild`

Position Information. `getAbsolutePosition`, `getPositionToParent`, `getPositionToID`, `getPositionToPath`, `setPosition`

Property Information. `getProperties`, `hasProperty`, `getProperty`, `setProperty`, `deleteProperty`

The methods dealing with a `Thing`'s position return `Position` or `PositionEvent` objects that hold, in addition to position data formatted as homogeneous pose matrices or translational and rotational vectors as used in VRML [3], information about the accuracy, the time lag and the time when the information has been created.

Properties The properties are part of every `Thing` object to give the possibility to store arbitrary information. A `Property` object is a simple key-value pair. Note that special properties may be used to introduce logical subtyping into the `Thing` class. However, this has not been used up to now but should be considered in future applications of the framework.

3.3.2 XML Specification of Input Files

To fulfill the requirements mentioned in section 3.1.1, we have to add the possibility to read in files containing information about World Model content to be added. The structure of these files has been modeled using XML. For every XML dialect, a so-called *Data Type Definition* has to be given that specifies the valid grammar for the dialect. We have specified a *World Model Modeling Language (WMML)*. Its structure is shown graphically in figure 3.5. The Data Type Definition is given in appendix D.

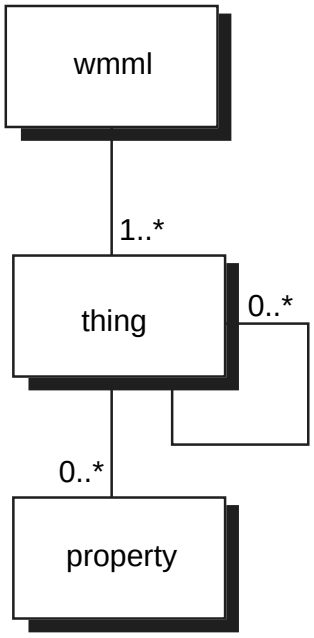


Figure 3.5: Structure of the World Model Modeling Language

To parse the XML files given to the World Model service, we used the freely available xerces implementation [63] of a SAX *(Simple API for XML)* XML parser.

3.3.3 An Example: Entering a New Room

To sum up the discussion of the World Model, we will give a simple example. The problem is as follows. The user of a DWARF system enters a new room and the system somehow realizes it. Part of the DWARF system is an optical tracker that relies on fiducials. In consequence, the tracker asks the World Model about information on these fiducials.

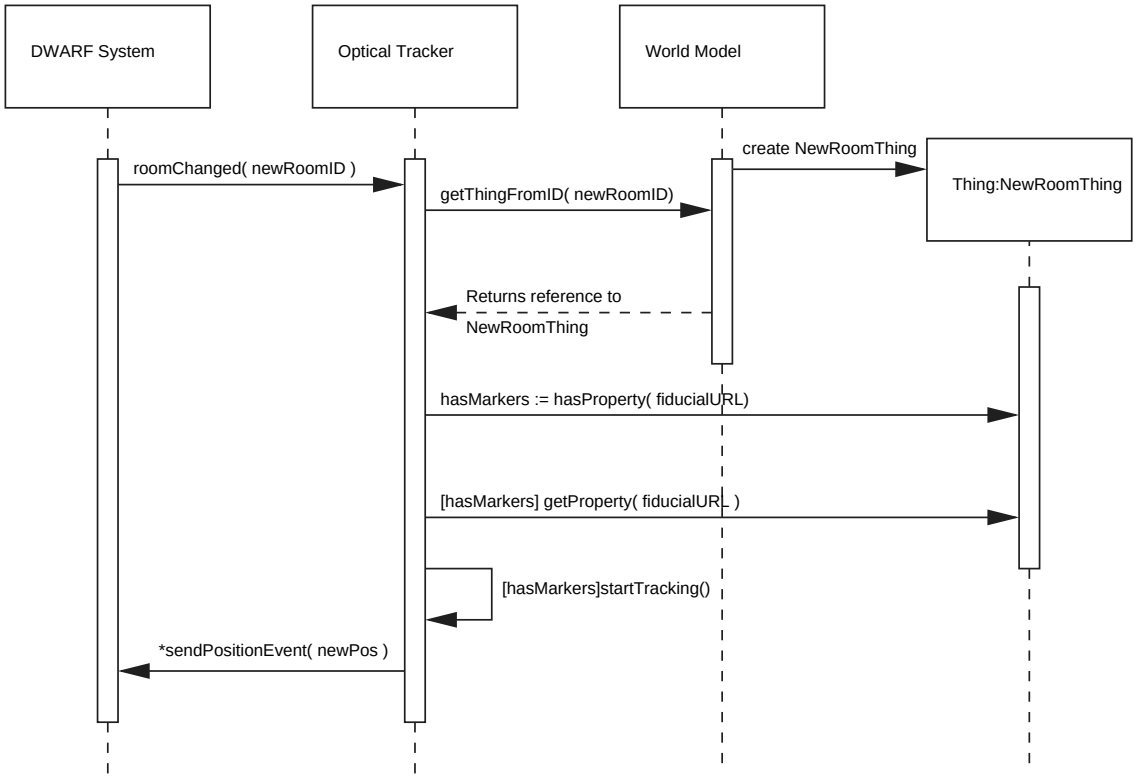


Figure 3.6: Sequence Diagram: Entering a New Room

As you can see in figure 3.6, the use of the World Model is convenient and does not require too many function calls.

4 An Overview of Tracking

We have seen a general overview of the tasks that have to be done for Augmented Reality. If we recall Azuma's definition of AR (see chapter 1.1), all things we have done up to now assume that we know the dynamic position and orientation of the user or at least of some objects in the scene.

In this chapter, we will see what has to be done to get this data. We start with a general description including the definition of the most important terms, go on with a short overview of the options of obtaining the position and orientation of an object and finally give an introduction to the mathematical background of optical trackers.

4.1 General Description

According to Azuma's definition of Augmented Reality, registration in 3D is a crucial requirement for every AR application. The DWARF system has been developed as a more general framework that supports not only "real" AR systems, but every system that depends on the position of some actors.

In consequence, we refer to a *Tracker* as a device that gives us some kind of information about the position of an object that is part of the DWARF system.

This definition has two parameters to be specified. The first is the nature of the "object". The most obvious thing that comes into one's mind is the AR system's user. However, we may want to track other devices as well. We just have to think of the user interacting with a real object like an engine or a part thereof or a virtual object that may be used for applications in construction. In addition, the system may have several user's that have to be tracked separately. If this degree of complexity is still not enough, we just have to add some parts of the users' bodies.

The second parameter is the "position". Although other approaches are thinkable (see [13] for an in-depth discussion), we will remain in a standard Cartesian three-dimensional coordinate system for the rest of this thesis. For this coordinate system, six parameters are necessary to describe the complete three-dimensional position and the three-dimensional orientation of an arbitrary rigid object in space.

Pose and Position From now on, we will call the six-dimensional tuple the object's *pose*. *Position* refers to the object's translation relative to the origin of the coordinate system and *orientation* to the rotation relative to the coordinate system's unity vectors. The representation of the position (translation) is straightforward – a three-dimensional vector $\mathbf{t}$.

Representing the Orientation It is a little bit more tricky to represent the orientation. In section 3.2.1, we have seen a 3×3 rotational matrix as the possibility used for the internal storage in the DWARF World Model. Another possibility is to decompose the entire rotation in three parts. First, we rotate around the x-axis of the reference frame, second, around the y-axis of the once rotated system and finally around the z-axis. If we call the x-axis rotation angle ω, the y-axis angle ϕ and the z-axis angle κ, we have the representation

$$\mathbf{R} = \mathbf{R}(\omega, \phi, \kappa) = \mathbf{R}(\kappa) \cdot \mathbf{R}(\phi) \cdot \mathbf{R}(\omega). \tag{4.1}$$

If we represent each rotation around a single axis as a rotational matrix,

$$\begin{aligned} \mathbf{R}(\omega) &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\omega & \sin\omega \\ 0 & -\sin\omega & \cos\omega \end{pmatrix} \\ \mathbf{R}(\phi) &= \begin{pmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{pmatrix} \\ \mathbf{R}(\kappa) &= \begin{pmatrix} \cos\kappa & \sin\kappa & 0 \\ -\sin\kappa & \cos\kappa & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{4.2}$$

we obtain the conversion from the angle representation to the orthonormal matrix representation:

$$\mathbf{R}(\omega, \phi, \kappa) = \begin{pmatrix} \cos\phi\cos\kappa & \sin\omega\sin\phi\cos\kappa + \cos\omega\sin\kappa & -\cos\omega\sin\phi\cos\kappa + \sin\omega\sin\kappa \\ -\cos\phi\sin\kappa & -\sin\omega\sin\phi\sin\kappa + \cos\omega\cos\kappa & \cos\omega\sin\phi\sin\kappa + \sin\omega\cos\kappa \\ \sin\phi & -\sin\omega\cos\phi & \cos\omega\cos\phi \end{pmatrix} \tag{4.3}$$

If we set

$$\mathbf{R}(\omega, \phi, \kappa) = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

we can obtain the angle representation from the orthonormal representation using

$$\begin{aligned} \sin\phi &= r_{31} \\ \tan\omega &= -r_{32}/r_{33} \\ \tan\kappa &= -r_{21}/r_{11} \end{aligned} \tag{4.4}$$

and have therefore proven the equivalence of the two representations.

Another important representation of the orientation is derived by rotating around a vector $\mathbf{v}$ with angle α. Every rotation can be represented in this way, and we get four parameters, three for the vector (whose length is arbitrary) and one for the angle. This representation is often used for computer graphics applications. Both VRML [3] and OpenGL [71] use such a vector representation. For the following discussion, we assume the vector to be normalized. To get the orthonormal rotation matrix out of the rotation vector and angle, we first have to

compute the matrix $\mathbf{V}$ that is the linear transformation that computes the cross product of the vector $\mathbf{v}$ with any other vector $\mathbf{x}$.

$$\mathbf{V} = \begin{pmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{pmatrix} \tag{4.5}$$

Now we can write the rotation matrix in terms of $\mathbf{V}$:

$$\mathbf{R} = e^{\mathbf{V}\alpha} = \mathbf{I} + \mathbf{V}\sin\alpha + \mathbf{V}^2(1 - \cos\alpha) \tag{4.6}$$

The derivation of equation (4.6) can be found in [22]. To convert the other way round, we use the following formulae from [52]:

$$\begin{aligned} \cos\alpha &= \frac{\text{trace}(\mathbf{R}) - 1}{2} \\ (\mathbf{R} - \mathbf{I}) \cdot \mathbf{v} &= 0 \end{aligned} \tag{4.7}$$

There exist quite a few additional modes to represent three-dimensional orientation. Perhaps the most important is the *quaternion representation* discussed in [24], pp. 140–143.

It should now be clear that there are many representations of orientation that are mathematically equivalent. However, we have not yet motivated why we should use different representations for different applications. The reason is simple. Certain algorithms for pose estimation depend on a special representation to be numerically stable.

Note that for certain applications, it may not be necessary to get the complete six-dimensional pose. Sometimes, the two- or three-dimensional position may suffice, and in some cases we may be interested only in the current orientation. We even may think of a simple tracker giving only the two-dimensional position of an object relative to the user's display. This tracker would be sufficient for augmenting reality using so-called stickies, virtual objects containing textual information attached to real objects.

Tracking and Calibration We have now specified the parameters we want to obtain from our trackers. As every AR system has to work in real-time, we can split the pose determination in two steps. During *Calibration* we measure all parameters that are at least assumed to be invariant over time. To give an example, if we are given an optical tracker we have to obtain the camera parameters such as the focal length during this step. Note that calibration is an off-line procedure that has to be performed only once for every setup.

The actual task of *Tracking* determines the parameters that vary over time, i.e. the pose of an object and uses the knowledge obtained in the calibration step.

4.2 Types of Trackers

Having a general knowledge about the data we want to obtain, we can now discuss some different tracking devices and have a look at their advantages and disadvantages. The following discussion is based on [13] and we refer the reader to this thesis for more detailed information.

We distinguish two classes of trackers. *Absolute Trackers* give the pose information relative to a (fixed) coordinate system. *Relative Trackers*, in contrast, give only information about the change in pose during the last time frame Δt. Note that although it is mathematically feasible to use relative trackers for absolute pose estimation, this approach fails because of drift and error propagation.

4.2.1 Absolute Trackers

Let us start with some trackers that determine the absolute position of an object to be tracked.

4.2.1.1 Mechanical Methods

This is perhaps the most obvious method. Mechanical methods measure changes in pose by physically linking the tracked object to a known point in the reference coordinate system. Examples are sensors on wheels in vehicle navigation systems or robot arms attached to the tracked object.

The major advantage of mechanical tracking is its accuracy and speed. However, it is mostly very obtrusive and therefore not suited for wearable AR systems that use DWARF components. In addition, the range is usually not very large.

4.2.1.2 Magnetic Trackers

Magnetic tracking uses either the earth's magnetic field for orientation estimation *(passive magnetic tracking)* or an artificially generated electromagnetic field for pose estimation *(active magnetic tracking)*.

The active variant has been used in the majority of existing AR systems, as it allows millimeter precision at high update rates. In addition, it is commercially available, e.g. the "Flock of Birds" tracker from Ascension Technology [10]. It is possible to have a single emitter for multiple sensors, leading to inexpensive tracking of many objects simultaneously.

Nevertheless, magnetic tracking is distorted heavily by the presence of ferromagnetic material. This and the accuracy diminishing quadratically with the distance from the emitter make magnetic trackers ill-suited for applications outside specially designed laboratories.

4.2.1.3 Global Positioning System (GPS)

Global positioning systems are space-based radio positioning systems that provide twenty four hours, all-weather, three-dimensional position information in a common reference system for any point in the world.

Since the launch of the first satellite, the Russian "Sputnik", in 1957, GPSs exist. The system known today as "GPS" is NAVSTAR (Navigation System with Timing and Ranging) funded and controlled by the U.S. Department of Defense. It consists of 24 active satellites with four satellites in each of six nearly circular orbits.

GPS is perhaps the most often used tracking system worldwide. All car navigation systems use it, and most commercial aircraft are equipped with three-dimensional GPS receivers. The receivers are available in extraordinarily small size ($10 \times 5 \times 2$ cm^3) and at low cost.

We have to keep in mind the major drawbacks of GPS. First, GPS works only outdoors with huge parts of the sky being visible, as at least three satellites must be visible simultaneously. Second, it takes quite a long time to initialize the system (approximately one minute). Third, accuracy is limited to the range of ten meters, allowing the use of GPS only for coarse tracking purposes.

4.2.1.4 Tag Trackers

In this section, we will examine trackers that send data every time they reach a known point. Simple variants are *RFID (radio frequency ID) tags* using electromagnetic fields to detect passive tags in the vicinity of the RF emitter or bar code readers.

These trackers give a precise position in real-time, but only once. However, they may well be used for coarse indoor tracking.

With more and more wireless communication technology being installed in buildings and urban areas, it is feasible to develop trackers that evaluate information from this communication infrastructure. To give an example, the cells used for mobile telephony are quite small and may be used as a substitute for GPS systems that works even indoors. Another possibility is to use the cell IDs of WaveLAN systems for tracking purposes.

The advantage of these systems to be very cheap (the hardware already exists most of the times) is opposed by the relatively low accuracy and the dependence on infrastructure.

4.2.1.5 Optical Feature Based Trackers

Feature based optical tracking is based on detecting and tracking certain features in an image given by a video camera. These can be lines, corners or any other features that are easy and reliable to detect. To track the user's or an object's position in a room, obviously some models have to be given according to the intended application.

If we know the position of at least three non-collinear model points in the camera's 2D coordinate system, we may compute the six-dimensional pose.

Unfortunately, using today's image understanding techniques, the computational expense needed to perform the task of tracking without any artificial aids given is in general too high to be done in real time. In consequence, artificial tracking aids referred to as *fiducials* have to be attached to well-known locations in the real 3D world, either to the surrounding room and/or to the tracked object. We will elaborate on this problem in chapter 6.

Note that we have not mentioned where the camera has to be placed. An immediately arising approach is to mount it on the user's head, as usually a display device has to be carried anyway. The advantage is that we do not have to determine the position of the user in the environment coordinate system anymore, we just have to determine the position of a tracked object relative to the camera. Unfortunately, if the user moves his head (i.e. changes

his orientation) too fast, the video image may get blurred and in result the tracker fails. If we put the camera fixed in the real world, we do not have this problem, but we have to track both the user and the object to be augmented. A combination of both techniques, i.e. one camera on the user's head and one or several others fixed on places of interest in the room, seems to be the most promising technology and is a topic of ongoing research [41].

The main advantage of optical tracking is its "natural" approach to tracking. People normally use the things they see as the major information in order to determine where they are, in consequence we could say that humans use optical tracking as well. In addition, the tracking procedure is usually not influenced by special circumstances as ferromagnetic materials in a room, although varying lighting conditions lead to problems. Finally, optical tracking seems to be well suited for large scale applications.

On the other hand, we have to keep in mind the computational expense of getting information out of a video image. The better the tracking quality we wish to achieve, the more we will hurt the real-time constraint crucial for good AR-systems. In addition, the determination of a 3D pose out of 2D input data is all but optimal. If we want to overcome this disadvantage, we would have to use a stereo camera system. This would inevitably lead to a doubling of the input data rate, eventually hurting our real-time constraint even more. This and other effects lead to a smaller accuracy compared to magnetic trackers.

4.2.2 Relative Trackers

As mentioned above, Relative Trackers output the change in pose during a time interval Δt. In some situations, this information may suffice. However, care has to be taken to compensate for effects like drift and error propagation.

4.2.2.1 Inertial Trackers

Inertial Trackers determine the pose using principles of Newton's law. They usually consist of two parts.

The *Accelerometer* is used to obtain linear acceleration values. If acceleration is double integrated over time, it is possible to obtain the translational value, i.e. the tracked object's position. However, this double integration is heavily distorted by errors in the accelerometer that can not be eliminated completely. Most accelerometers use three masses coupled to the instrument case through an elastic restraint. Each mass is allowed only one degree of freedom. The pressure of each mass is converted to electric signals using the piezoelectric effect. Note that the accelerometer must compensate for gravity. In consequence, its rotation must be known all times.

The *Gyroscope* delivers the tracked object's rotation. It measures the speed of rotation around three axes, the absolute rotation value is obtained by single integration.

The major advantage of inertial tracking systems is their complete selfcontainedness. They are not influenced by the earth's magnetic field, varying light conditions or whatever material in the surrounding. Nevertheless, its use for Augmented Reality applications is rather limited. Because of their concept of measurement, an error linear (for rotation values) or even quadratical (for translation values) in time is introduced to the output. In addition,

high precision devices are rather clumsy and prohibitively expensive. Smaller and cheaper devices exist, but show significant drift effects. These can be circumvented by combining optical methods with inertial tracking, although this is computationally expensive.

4.2.2.2 Optical Flow Trackers

Recall our discussion of feature based optical trackers. We stated that we have to create a model of the tracker's environment in order to determine its pose. What information can we get without any model? Obviously, this information can only be relative to a certain point in time. *Optical Flow Trackers* use the brightness information in a sequence of video images to derive relative pose estimation. Horn ([29], p. 280) defines optical flow as follows:

> *Brightness patterns in the image move as the objects that give rise to them move.* Optical flow *is the apparent motion of the brightness pattern.*

If we assume the objects in the image being fixed in space and only the camera moving, it is obvious that we can get information about the camera's movement using optical flow.

There exists a variety of algorithms to compute the optical flow for an image sequence (see [31], chapter 18, for a discussion of some). All algorithms deliver vectors for some or all image points telling the flow of the point during the last time frame. Most of these algorithms are quite fast, allowing optical flow computation in real time. However, the computation of the pose change out of the optical flow is rather time consuming. In addition, drift is a major problem. Note that optical flow methods are influenced by varying lighting conditions as well, but if the variations are slow, the effect will not be as disturbing as for fiducial based optical trackers.

The only reasonable use of optical flow methods for tracking seems to be in combination with absolute fiducial based optical trackers to compensate for the drift. This has been the major goal of this thesis.

4.2.3 Summary

We have seen a few tracking devices and discussed their pros and cons. It should now be clear that the optimal tracker does not exist. There is always a tradeoff between cost, size, speed, accuracy and dependency on infrastructure or stable environment conditions such as lighting conditions or ferromagnetic material. An extensive registration error analysis can be found in [26].

The choice of the right tracker is at the beginning of the development of every Augmented Reality system and has to be well thought of. It has been a goal of the DWARF project to allow flexible integration of various tracking devices without having to rewrite any code. More details on this can be found in [13].

The remainder of this thesis is about optical tracking, the only tracking technology discussed above that is still a topic of ongoing research. For all other technologies, most error sources and limitations are known and have to be or are already solved using engineering methods. In addition, optical tracking depends solely on semiconductors. We therefore can expect almost automatic increases in speed and/or accuracy by the current exponential advances in computer hardware.

5 Optical Tracking

This chapter serves several purposes. We want to give an introduction to the basic principles of optical tracking and decompose the problem. Furthermore, we want to start with some mathematics to give the background information necessary for understanding the algorithms discussed in the subsequent chapters. Finally, this chapter serves as a requirements analysis for the development of an optical tracker.

5.1 Principles of Optical Tracking

The first question for every problem that has to be solved in computer science is about the decomposition of the problem in manageable steps. In this section, we will show all the steps that have to be performed to implement a working feature-based optical tracker.

From now on, we will assume that the output of the tracker consists solely of its camera's pose. If we use this tracker for Augmented Reality applications, it seems reasonable to mount the tracker's camera on the user's head in order to get the user's pose.

Creating A Model If we want the optical tracker to give permanent updates of its pose in a well defined coordinate system, e.g. a room's system, the first thing we have to do is to set up this coordinate system. As the tracker is feature-based, we have to create such features. Normally, artificial landmarks (*fiducials*) are chosen. We now have to place these fiducials in all areas of interest such that the tracker is able to identify them by whatever means.

To finish the model, we have to measure the fiducial's position (based on the room coordinate system) and hand them over to the tracker.

Note that the model just described is very simple. We may add certain aspects like occlusion properties of some objects up to an arbitrary level of complexity.

Calibrating the Camera Now we have to determine the optical properties of the camera. We will discuss which properties have to be obtained in the next section. In addition, we may want to measure the camera's pose relative to the user's eyes (or the head mounted display) in order to allow well aligned video overlays.

After calibration, we know all values of our tracking environment that are basically invariant over time. However, if we want to use the tracker in a large environment, it should be possible to reuse some fiducials. To make this feature available, we have to provide mechanisms to reconfigure the tracker, especially the fiducial's positions, during runtime.

Tracking Tracking combines the data obtained during calibration with the live video image obtained from the video camera. Sophisticated algorithms are used to process the data in real time.

Most of these algorithms, including the one we describe in this thesis, can be decomposed in two steps:

1. Detect the fiducials' two-dimensional positions in the video image frame by means of computer vision.
2. Compute the camera's three-dimensional position in the real world.

The result of this computation is the camera's pose. Ideally, we get the pose immediately after the video frame has been moved into the computer's memory. If we send the pose to the other components of the AR system, it can be used to compute correctly aligned video images in a see-through head mounted display.

Figure 5.1 illustrates the steps necessary for successful optical tracking.

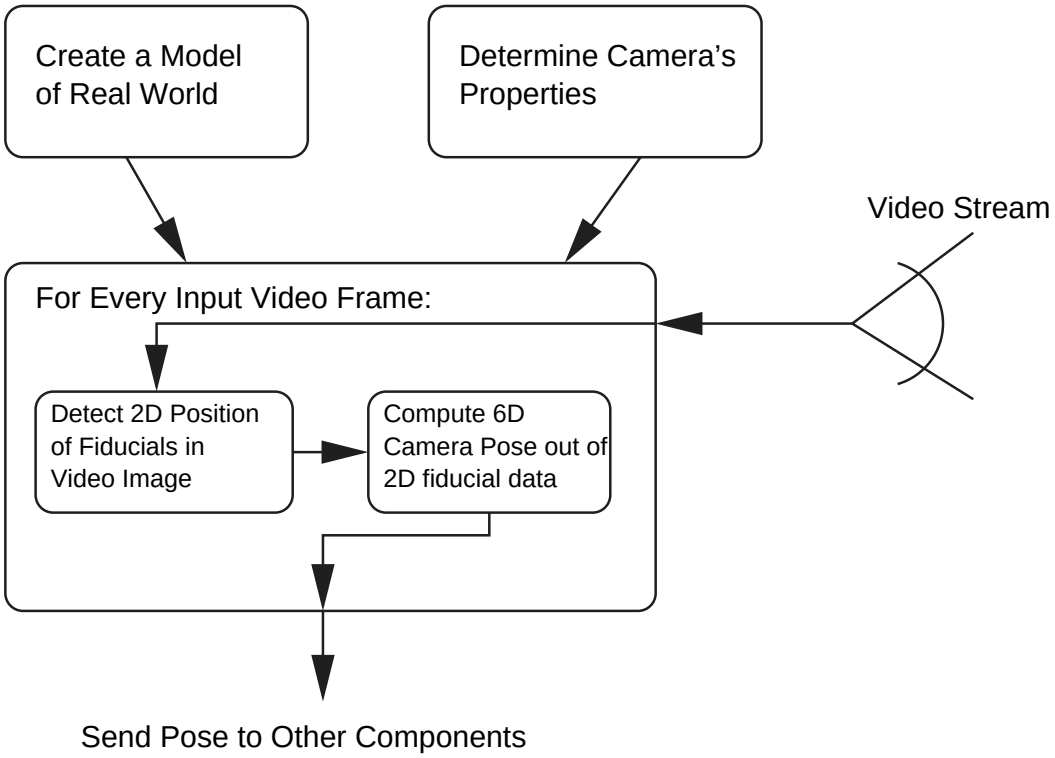


Figure 5.1: Task Decomposition for an Optical Tracker

5.2 Mathematical Background of Optical Tracking

Like for all tracking methods, at the very heart of optical tracking are mathematical methods. In this section, we will explain the underlying camera model for the tracker presented in the

following chapters. We will discuss the parameters that have to be known and discuss how we can obtain them. This discussion is based on [58] and [70].

The central question of the camera model is the mapping of three-dimensional points in the real world to two-dimensional points in the camera plane. To make life easy, we want to have a linear mapping from 3D to 2D points. In addition, we split up this projection in the parts that can be determined offline during calibration, the so-called *intrinsic camera parameters*, the parts that result from the model of projection, and finally the parts that are determined during tracking, the *extrinsic camera parameters*. We therefore can write

$$x_{\mathbf{cam}} = \mathbf{P} \cdot x_{\mathbf{world}} = \mathbf{H} \cdot \mathbf{P}_{\mathbf{per}} \cdot \mathbf{D} \cdot x_{\mathbf{world}} \tag{5.1}$$

with

$$\begin{aligned} \mathbf{D} &: \text{extrinsic camera parameters} \\ \mathbf{P}_{\mathbf{per}} &: \text{perspective projection matrix} \\ \mathbf{H} &: \text{intrinsic camera parameters} \end{aligned}$$

The remainder of this section derives the matrices of equation (5.1).

5.2.1 Pinhole Camera Model

The pinhole camera model is a very simple camera model that is completely sufficient for the purpose of tracking. Using this model, the image plane is perpendicular to the z-axis of the three-dimensional camera coordinate system. A 3D point's image is computed by the intersection of a ray from itself to the center of projection O_C with the image plane. the distance f from O_C to the image plane is called *focal distance*. Figure 5.2 shows the pinhole model.

Using simple geometry, we obtain the following relation from 2D to 3D coordinates:

$$x' = f \cdot \frac{x}{z} \qquad \text{and} \qquad y' = f \cdot \frac{y}{z} \tag{5.2}$$

These equations are called the *perspective projection*. Note that they are not linear. To linearize them, we have to use *homogeneous coordinates* discussed in section 3.2.1. We basically add a third coordinate to every two-dimensional point:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} \mapsto \begin{pmatrix} x' \cdot z \\ y' \cdot z \\ 1 \cdot z \end{pmatrix} = \begin{pmatrix} fx/z \cdot z \\ fy/z \cdot z \\ z \end{pmatrix} = \begin{pmatrix} fx \\ fy \\ z \end{pmatrix} = \begin{pmatrix} x'' \\ y'' \\ z \end{pmatrix} \tag{5.3}$$

The reverse process, the *dehomogenization*, works as follows:

$$\begin{pmatrix} x'' \\ y'' \\ z \end{pmatrix} \mapsto \begin{pmatrix} x''/z \\ y''/z \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix} \tag{5.4}$$

Homogenization and dehomogenization for three-dimensional coordinates work in the same manner.

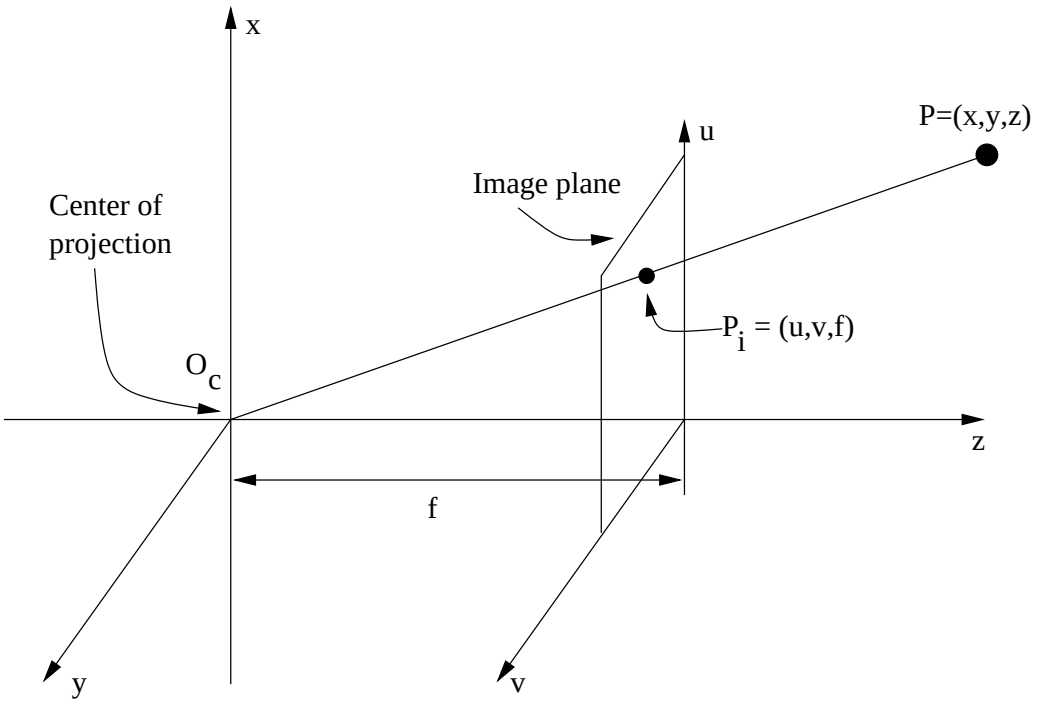


Figure 5.2: Pinhole Camera Model

We now are able to write the perspective projection as linear mapping, i.e. as a matrix. As a first step, we homogenize the image points:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} fx/z \\ fy/z \end{pmatrix} \mapsto \begin{pmatrix} fx \\ fy \\ z \end{pmatrix} = \begin{pmatrix} x'' \\ y'' \\ z \end{pmatrix} \tag{5.5}$$

Now we have the linear projection matrix that projects three-dimensional homogeneous coordinates on two-dimensional ones:

$$\begin{pmatrix} x'' \\ y'' \\ z \end{pmatrix} = \begin{pmatrix} fx \\ fy \\ z \end{pmatrix} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \mathbf{P_{per}} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{5.6}$$

5.2.2 Calibration with the Pinhole Camera Model

Recall our definition of section 4.1. Calibration is the determination of all parameters that are at least assumed to be invariate over time. Obviously, the parameter f of the projection matrix $\mathbf{P_{per}}$ is part of the calibration. We will now discuss the additional values that can be measured offline.

Up to now, we assumed several properties of the camera to be ideal:

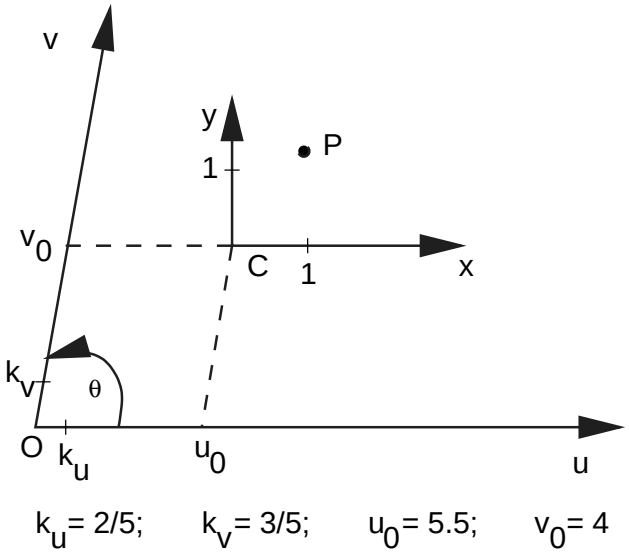


Figure 5.3: Intrinsic Camera Parameters

- The axes of the camera may not be orthogonal. The angle between them is called θ. We assume the axes to be arranged as depicted in figure 5.3, namely the x^*-axis being coincident with the x-axis and the y^*-axis being rotated around an angle of $90° - \theta$ compared to the y-axis. The new coordinates x^* and y^* can therefore be expressed as

$$x^* = x - \frac{y}{\tan\theta}; \qquad y^* = \frac{y}{\sin\theta}.$$

  The corresponding matrix $\mathbf{H_1}$ for two-dimensional homogeneous coordinates is

$$\mathbf{H_1} = \begin{pmatrix} 1 & -\frac{1}{\tan\theta} & 0 \\ 0 & \frac{1}{\sin\theta} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- We assumed the pixels to be square and of union length. From now on, they have size $k_u \cdot k_v$. Note that this operation can be seen as scaling the x-axis by k_u and the y-axis by k_v. This scaling can be expressed by the matrix

$$\mathbf{H_2} = \begin{pmatrix} k_u & 0 & 0 \\ 0 & k_v & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- The origin of the image coordinate system was assumed to be located at the intersection of the optical axis (the z-axis in the camera coordinate system) and the image plane. In general, it is situated at $C = (u_0, v_0)^T$ with C's coordinates given in pixel coordinates, i.e. the $k_u \cdot k_v$ scale. This transformation is expressed by

$$\mathbf{H_3} = \begin{pmatrix} 1 & 0 & u_0/z \\ 0 & 1 & v_0/z \\ 0 & 0 & 1 \end{pmatrix}$$

to transform the homogeneous vector $(x'', y'', z)^T$.

- The optics of the camera may create distortions that can be removed geometrically. However, we will not discuss this topic here and have not used any of this correction in the tracker developed.

The parameters we just introduced are visualized in figure 5.3 with (x, y) being the coordinates in the ideal coordinate system.

The derivation of the matrix $\mathbf{H}$ is straightforward from this figure and the matrices $\mathbf{H_1}, \mathbf{H_2}$ and $\mathbf{H_3}$:

$$\begin{pmatrix} u \\ v \\ s \end{pmatrix} = \mathbf{H_3} \cdot \mathbf{H_2} \cdot \mathbf{H_1} \cdot \begin{pmatrix} x'' \\ y'' \\ z \end{pmatrix} = \mathbf{H} \cdot \begin{pmatrix} x'' \\ y'' \\ z \end{pmatrix} = \begin{pmatrix} k_u & -k_u/\tan\theta & u_0/z \\ 0 & k_v/\sin\theta & v_0/z \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x'' \\ y'' \\ z \end{pmatrix} \tag{5.7}$$

Now we can sum up all steps of the calibration procedure in a single matrix $\mathbf{H} \cdot \mathbf{P_{per}}$:

$$\begin{pmatrix} u \\ v \\ s \end{pmatrix} = \mathbf{H} \cdot \mathbf{P_{per}} \cdot \mathbf{D} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} fk_u & -fk_u/\tan\theta & u_0/z & 0 \\ 0 & fk_v/\sin\theta & v_0/z & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}}_{\mathbf{H}\cdot\mathbf{P_{per}}} \cdot \mathbf{D} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{5.8}$$

As we can see from this matrix, the parameters k_u and k_v can not be observed separately from f. In consequence, we introduce two new parameters instead of the three old ones:

$$\alpha_u = fk_u \qquad \alpha_v = fk_v \tag{5.9}$$

In summary, calibrating the camera consists of the determination of five parameters, $\alpha_u, \alpha_v, \theta, u_0$ and v_0.

5.2.3 Tracking with the Pinhole Camera Model

The determination of the pose is at the heart of the tracker's work. As we discussed in chapter 4.1, we have to determine the translation and orientation in a suitable representation, If we assume to use homogeneous coordinates as we did in the previous sections, we have to obtain a translational vector $\mathbf{t}$ and a rotational matrix $\mathbf{R}$ to get the matrix $\mathbf{D}$:

$$\mathbf{D} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.10}$$

We will see in the following chapters how to obtain these values.

5.3 Functional Requirements for an Optical Tracker

We have discussed several properties of the optical tracker. Let us now sum them up to give a handy reference of the requirements that have to be fulfilled.

The optical tracker determines the pose (translation and orientation) of a video camera. It uses a live video stream from this camera in order to detect some artificial landmarks (fiducials) in every video image. The real-world position of these fiducials has to be given. To ensure communication with other components of the DWARF system, the optical tracker has to be implemented as a DWARF service (see [46] for details).

Due to the nature of its task, the optical tracker does not provide any user interface. The optical tracking service does not provide any output capabilities beyond debugging purposes. All access to the optical tracker (except starting it) is done via software interfaces. To ensure the optical tracking being encapsulated from any visualization tasks, even debugging programs for rapid visualization outside the DWARF framework should communicate using the standard DWARF tracking API.

The following (intrinsic) parameters of the video camera have to be determined in an offline procedure:

1. The focal length multiplied by the pixel size in u and v direction of the image frame: α_u and α_v
2. The center point C of the image frame: $C = (u_0, v_0)$
3. The angle between the u and v axis of the image frame: θ

During runtime, the optical tracker gives regular updates of the camera's pose. This data is sent via the DWARF event architecture to other DWARF components using the data.

To ensure the use of the tracker in a large environment, there have to be mechanisms that allow dynamic reconfiguration of the tracking service's model of the real world. These mechanisms have to be based on data stored in the DWARF World Model service. The tracking service should register for events indicating a change of the user's global position. For every such incoming event, the World Model should be queried for the existence and probably location of fiducials that can be tracked.

5.4 Nonfunctional and Pseudo Requirements for an Optical Tracker

Nonfunctional Requirements The most important nonfunctional requirement for the optical tracker is sufficient performance. The tracker has to give pose information in real time. The rate should be above ten updates per second. Another important criterion for real-time performance is delay. To ensure small drift effects in optical see-through augmented reality, the delay should be below 100 milliseconds.

As the DWARF system is intended to be used on wearable computing systems, the optical tracking service has to be running on a small laptop in the range of a 400 MHz Pentium II processor. Its memory consumption should be low.

Pseudo Requirements Like the rest of the DWARF system, the optical tracker should principally run on a large variety of hardware devices. To ensure this property, all non-standard libraries used for tasks as image acquisition have to be documented and encapsulated in a way that allows easy migration to other platforms.

The optical tracking service will be implemented in an object oriented language that allows seamless integration with CORBA for communication with all other DWARF components.

The task of implementing an optical tracker is extremely time consuming and involves algorithms and methods from many different fields of computer science and mathematics. To ensure a low development time, heavy use of already existing, preferably optimized libraries has to be made.

6 Fiducial Detection

Having seen the overall system decomposition of an optical tracker depicted in figure 5.1, we will see in this chapter how to perform the first major subtask of optical tracking.

The task we want to explain is the detection of artificial landmarks, so-called *fiducials*, in a video frame. We will give a general introduction to the problem, stating important difficulties of computer vision that may not seem obvious at first sight, proceed with the interesting approach of multi-ring color fiducials that avoids the major disadvantage of most fiducial-based systems, and finally describe the detection mechanisms used in the tracker that has been developed for this thesis.

Note that this thesis is not mainly about computer vision. Its goal was to provide a working optical tracker relying on existing technologies already implemented in available software libraries. Hence, we will give only an informal overview of some problems that are discussed in depth in the computer vision literature. A good introduction is given by Horn [29].

6.1 Problem Description

Fiducial detection can be described in a single sentence: Given a digital image, detect certain artificial landmarks known a priori in the image and output their two–dimensional positions.

However, the real task is somewhat harder than it might seem. In this section, we will describe some of the general difficulties that arise.

Human versus Computer Feature Detection The first thing we have to keep in mind is that the human brain is doing an extremely efficient job in compensating for varying conditions such as lighting or reflection. If a video camera is mounted in a shaded room and suddenly the light is switched on, the attached computer "sees" a completely different image with heavily changed brightness values. Even more severe problems arise with color images, as there do not exist good methods to assure color constancy. In addition, we have to rethink common classifications of features into "bad" and "good features to track". Humans tend to track their position via features such as corners, edges or sharp lines. This approach is not possible for computers. Due to limited resolution of each video image, such small features can not be reliably detected. The resolution is mainly limited by processor speed, and for our setup with an Intel Pentium II processor running at 400 MHz, the maximum resolution for real-time processing was 320×240 pixels. The features that can be detected by computer systems tend to be large, homogeneously colored areas. Of course, the edges of

these areas can be extracted to make exact localization possible. Figure 6.1 tries to illustrate these differences.

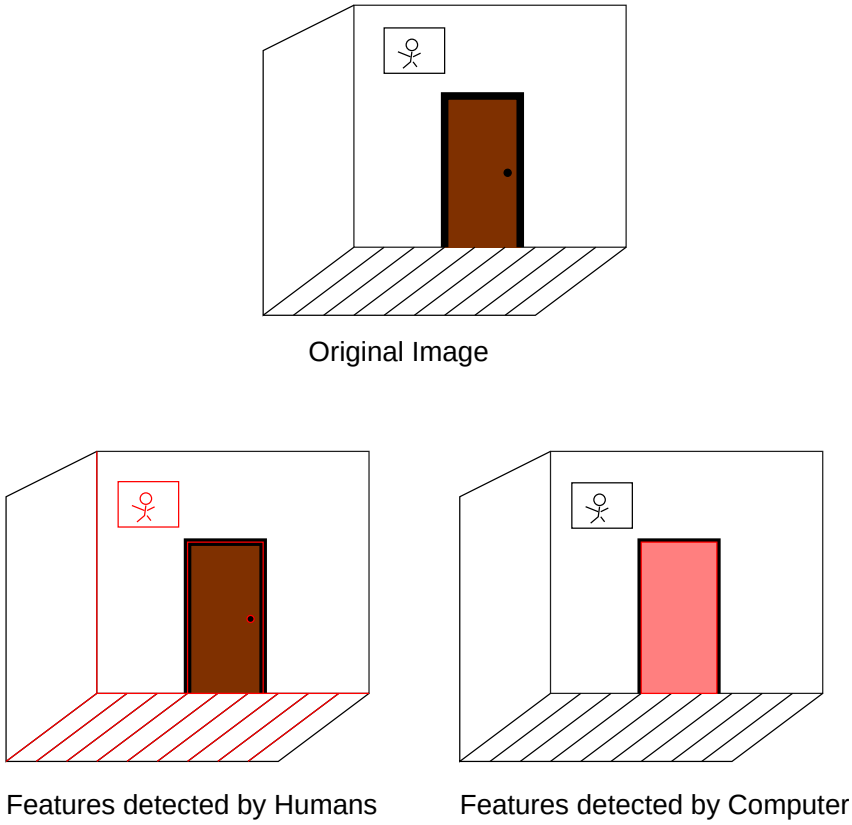


Figure 6.1: Differences in Human and Computer Vision

Processing Time As mentioned above, one of the hardest constraints for fiducial detection is the processing time available. Holloway [26] has shown that a delay of 1ms induces an average mismatch between real and virtual objects of $1/3$mm. It seems reasonable to set a registration accuracy of 3cm as a goal, leading to a maximum delay of the whole tracker of 100ms. If we now assume the fiducial detection to take half of the available time (the other half is needed for pose estimation), we may not afford to take more than 50ms. Many algorithms discussed in literature, e.g. [72], need processing time far from real-time and are therefore not suited for our needs. Nevertheless, there may be application domains that do not need a high registration accuracy, leading to relaxed time constraints.

Limited Resolution Most computer users expect a display resolution of a minimum of 1024×768 pixels. However, if we aim at keeping the processing time down, we must limit ourselves to video resolutions in the area of 320×240. In addition to processing the data, we have to get it into the computer. This might be a problem especially for wearable computers, as the widely available and therefore cheap so-called webcams that are usually connected to

the computer via the *Universal Serial Bus (USB)*, suffer from limited bandwidth. To overcome this disadvantage, compression methods are required that take away precious CPU cycles for decompression.

Nevertheless, with increasing processor speed and new computer interfaces such as *IEEE 1394 (a.k.a. FireWire or iLink)* or *USB2* it will be just a matter of time that resolution increases.

Size of Fiducials As should be clear from the discussion above, tracking can be facilitated by using artificial tracking aids that are specially designed for each detection algorithm, so-called *fiducials*. Most working feature detection algorithms are fiducial-based and do not rely on "natural" features.

The problem with these fiducials is now that they have to be attached to the user's environment such that they appear in the video camera's field of view. Naturally, there exists a tradeoff between a large number of fiducial which is good for the robustness of the tracking algorithm and a non obtrusive setup for the AR system being good for user acceptance. In addition, a fiducial in the camera's field of view has a maximum and a minimum size to be detected by the detection algorithm, leading to the necessity of variable sized fiducials to allow large range tracking. The next section deals with an algorithm aimed at this problem.

Environmental Conditions Environmental Conditions are perhaps the hardest constraint on successful feature detection and in consequence optical tracking. As we stated above, the same scene taken under varying lighting conditions may result in a completely different binary image. Features detected reliably in the morning may be misclassified in the afternoon. The same problem arises for detection algorithms based on color. Phenomena such as reflectance or the vicinity of other colored objects make tracking a lot harder and inaccurate.

These problems are in sharp contrast with the general requirement of the DWARF system to be usable in a large area. Especially for outdoor scenes, constant lighting can not be assured. Things get somewhat easier for indoor scenes, although some users of AR systems may oppose working in areas without natural light that are necessary to create stable lighting conditions for optical trackers.

Summary We have seen that many parameters have to be concerned to implement algorithms that reliably detect features. However, we did not aim at creating new technologies. We simply assumed the environmental conditions to be stable and therefore could use rather robust and fast algorithms.

6.2 Multiring Color Fiducials

So far we have discussed the size of fiducials. If all fiducials are of single size, this facilitates their design and manufacturing but reduces the scalability of the AR system. Ideally, it should be possible to track the video camera's position if it shows wide views as well as detailed views. This goal will not be reached if only fiducials of a single size are used.

However, using markers of various sizes simultaneously poses problems as well. To give an example, we just have to think of what is happening if the camera is only showing a detailed view of a part of a large fiducial. It will not be possible to recover the camera's pose just by this image, as at that moment no marker is detectable.

Cho, Lee and Neumann try in [19] an interesting approach to solve this dilemma. They propose

> *When a camera is too far or too close to a fiducial, the projected image of the fiducial in the input image is too small or too large to detect it correctly. Therefore, an AR system with single-size fiducials has a very limited tracking range. Although each fiducial has a fixed detectable range, the whole tracking range could be extended by combining different detectable ranges of different size fiducials.*
>
> *Multi-ring color fiducials have different number of rings at different size levels. The first level fiducial has one core circle and one outer ring. As the level goes up, one extra ring is added outside of the previous level fiducial. The number of rings in a fiducial tells the fiducial level where it belongs. The core circle and rings are painted with six color (red, green, blue, yellow, magenta and cyan). It introduces many unique fiducials and, it makes fiducial identification easier.*

Such multi-ring color fiducials are depicted in figure 6.2. It is shown in [19] that the optimal size ratio between adjacent levels is $c = 2$.

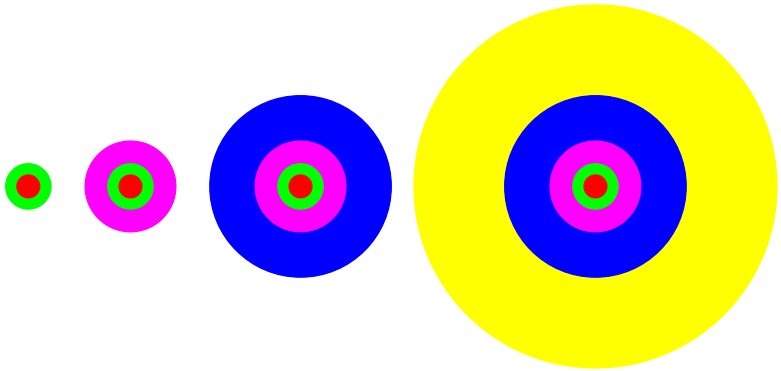


Figure 6.2: Multiring Color Fiducials

The main advantage of this method is that large size fiducials are even detected if only parts of them are in the camera's field of view, although these parts must include the center.

However, it is necessary for this method as for all others to attach many fiducials to the user's environment. These fiducials, especially the large ones, are very obtrusive and need to be attached to plain surfaces such as walls. In industrial environments, such large plain surfaces may not be available.

A final disadvantage for our purposes was that no freely available implementation exists. We did not have the time to implement the solution ourselves.

6.3 The ARToolKit

The *Augmented Reality Toolkit (ARToolKit)* (see [37]) is the only freely available feature tracker aimed at Augmented Reality applications we are aware of. As such, it has been our first choice for feature detection.

The ARToolKit has been developed to facilitate rapid prototyping of User Interface metaphors for AR applications. It provides a feature tracker, routines that compute the camera pose relative to each such feature, calibration methods to align real and virtual objects and OpenGL-based output routines for creating virtual objects. If all these components are used, real-time performance can be achieved on fast Intel based PCs. A major advantage of the ARToolKit is its availability in source code under the *GNU General Public License (GPL, [2])* that allows compilation on virtually all platforms that support ANSI C ([39]).

The fiducials used by the ARToolKit are of quadratic variable size. For each individual fiducial the exact size has to be known. Examples of such fiducials are shown in figure 6.3. All ARToolKit fiducials have the same structure. They are of square size s and have a black

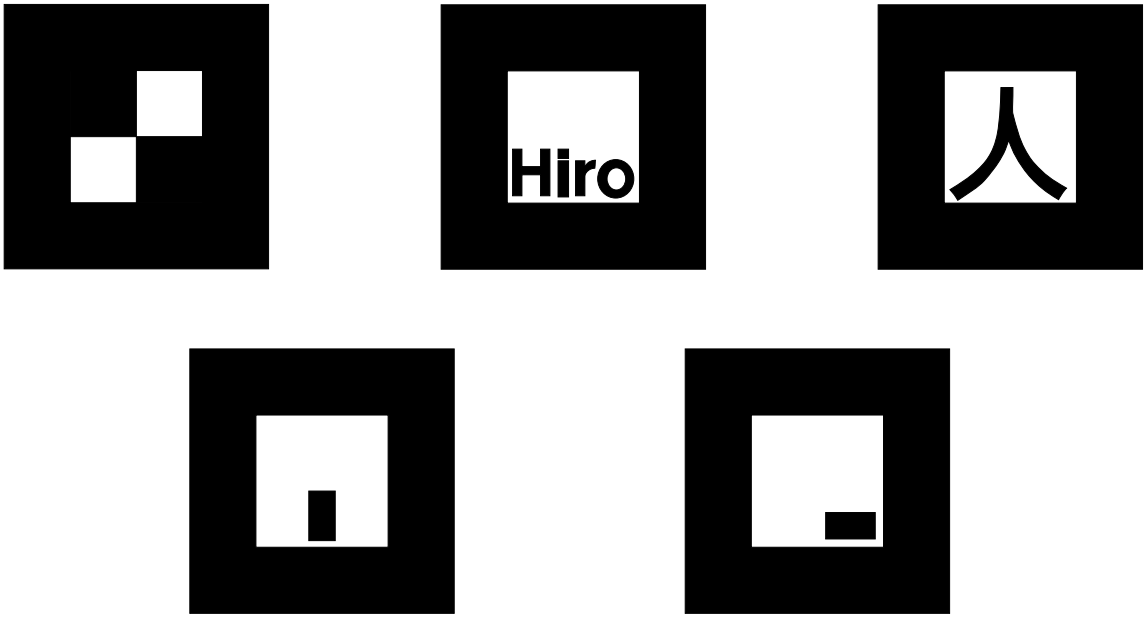


Figure 6.3: ARToolKit Fiducials

border of thickness $\frac{s}{4}$. The area of size $\frac{s}{2} \times \frac{s}{2}$ in the middle of the fiducial is filled with an arbitrary pattern that can be recorded and saved into a file readable by the detection procedures. This recording is done offline using an additional tool included in the software package.

Unfortunately, the source code is completely without comments such that modification or extraction of parts gets difficult. Nevertheless, it was possible to extract the parts necessary for fiducial detection and adopt them to our needs. Note that the algorithm works only on grayscale images and does not include any color processing. As described in [36], the detection algorithm works as follows:

1. Threshold the input image such that a binary image is created.
2. Search the binary image for regions that may be projected squares. More specific, all regions whose outline contour can be matched by four line segments are detected. Of course, many of these regions are not markers.
3. Normalize the regions and compare their interior with all patterns of potential fiducials. This is done by template matching. The templates have been registered in an offline procedure and are loaded during startup.
4. For every found marker, output its center coordinates, the coordinates of its four corners, its ID that identifies it with a pattern, its line segment equations and a confidence value that indicates how reliable the identification is.

We will describe the results of the use of the ARToolKit in chapter 11. In short, there exist better possibilities for feature detection in a general optical tracker that is not limited to the tabletop environment of usual ARToolKit applications. However, it is a good starting point to get a set of correlations from 2D image points to 3D real world points.

In the next chapters, we will see how we can use this data to obtain the camera pose.

7 Absolute Pose Estimation

Recall the discussion from section 4.2.1.5. The task of absolute pose estimation for optical feature-based trackers has several 2D to 3D point correspondences as input and a six-dimensional pose as output.

The input data yields usually from image understanding techniques like the one we discussed in the previous chapter. In this chapter, we will discuss useful mathematical techniques for this task and present two rather different algorithms aiming to solve this problem.

7.1 Problem Description

We assume to use the pinhole camera model described in section 5.2. Equation (5.1) stated

$$x_{\mathbf{cam}} = \mathbf{P} \cdot x_{\mathbf{world}} = \mathbf{H} \cdot \mathbf{P}_{\mathbf{per}} \cdot \mathbf{D} \cdot x_{\mathbf{world}}$$

with $x_{\mathbf{cam}}$ being a two-dimensional point in the image reference frame and $x_{\mathbf{world}}$ being the corresponding three-dimensional point in the real world coordinate system.

As mentioned above, we determined the matrices $\mathbf{H}$ and $\mathbf{P}_{\mathbf{per}}$ in an offline procedure. Note that the matrix $\mathbf{D}$ has only six degrees of freedom as it is describing the camera's pose consisting of three translational parameters t_x, t_y, t_z and three rotational parameters ω, ϕ, κ.

The goal of the absolute pose estimation problem is now to estimate these six values out of a sufficient number of correspondences of points $x_{\mathbf{cam}}$ to $x_{\mathbf{world}}$. As we have six degrees of freedom and get two equations out of every correspondence, three non-collinear points suffice. However, if we can provide more correspondences and solve the resulting overdetermined system in a suitable way, the computed pose should be much more reliable.

7.1.1 Solving Linear Least-Squares Problems: the Singular Value Decomposition

Assume that we have a probably rank-deficient overdetermined system of linear equations. In general, it is not possible to solve this system exactly. However, it is possible to solve it with a minimum-norm solution x according to a least squares criterion:

$$\min \|\mathbf{A}x - b\|_2 \qquad \text{and} \qquad \min \|x\|_2 \tag{7.1}$$

with

$$\mathbf{A} \in \mathbb{R}^{m \times n}; \quad x \in \mathbb{R}^n; \quad b \in \mathbb{R}^m$$

The *singular value decomposition (SVD)* is an efficient means of solving such problems. For every matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, there exists a decomposition

$$\mathbf{A} = \mathbf{U} \cdot \Sigma \cdot \mathbf{V}^T \tag{7.2}$$

with

- $\mathbf{U} \in \mathbb{R}^{m \times m}$, $\mathbf{V} \in \mathbb{R}^{n \times n}$ and $\mathbf{U}\mathbf{U}^T = I$, $\mathbf{V}\mathbf{V}^T = I$
- $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix (here shown for the case $m > n$):

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \dots & \sigma_{\min(m,n)} \\ 0 & 0 & \dots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} \tag{7.3}$$

  The values σ_i are called *singular values*. They hold

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m,n)} \geq 0 \tag{7.4}$$

Let us now have a look at some interesting properties of this decomposition. Further reference can be found in [51], [30] and [52].

1. The rank k of the matrix $\mathbf{A}$ is the number of singular values that are greater than zero. For practical applications, k is the number of singular values that exceed a suitable threshold.
2. The column vectors v_i of the matrix $\mathbf{V}$ with $\sigma_i = 0$ span $\mathbf{A}$'s null space.
3. If $\mathbf{A}$ is square and has full rank, i.e. $k = n = m$, the SVD can be used to obtain the inverse of $\mathbf{A}$:

$$\mathbf{A}^{-1} = \mathbf{V}\Sigma^{+}\mathbf{U}^T \tag{7.5}$$

   with Σ^{+} being a $n \times m$ diagonal matrix with values $1/\sigma_i$.
4. If $\mathbf{A}$ is not square or rank deficient, we define the matrix

$$\mathbf{A}^{+} = \mathbf{V}\Sigma^{+}\mathbf{U}^T \tag{7.6}$$

   and call $\mathbf{A}^{+}$ the *pseudoinverse* of $\mathbf{A}$.
5. The minimum-norm solution of the system (7.1) with rank k is given by

$$x = \mathbf{A}^{+}b \tag{7.7}$$

   To facilitate the computation, it can also be represented as

$$x = \mathbf{V}_k \left(\Sigma_k\right)^{-1} c \tag{7.8}$$

   where Σ_k is the leading k by k submatrix of Σ, the matrix $\mathbf{V}_k$ consists of the first k columns of $\mathbf{V}$ and the vector c consists of the first k elements of $\mathbf{U}^T b$.

7.1.2 Solving Nonlinear Least-Squares Problems

Unfortunately, the absolute pose estimation problem is not linear (a good explanation can be found in [24], pp. 125–129). In consequence, we have to think of methods to solve nonlinear least-square problems.

One approach consists of iteratively solving the linearized problem in which the linearization is taken around the current approximate solution ([24], pp. 129–131). We will describe here only the formulae that have to be applied, but not their derivation. We refer to [24] for further information.

The problem can be stated as follows. The parameters $x_1, x_2, \ldots, x_M$ are the unknown parameters that control the nonlinear transformations $a_1, a_2, \ldots, a_K$. The observed values of $a_1, \ldots, a_K$ are $b_1, b_2, \ldots, b_K$. We now want to minimize the least-squares criterion

$$\epsilon^2 = (b-a)^T(b-a) \tag{7.9}$$

where

$$b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{pmatrix} \quad \text{and} \quad a = \begin{pmatrix} a_1(x_1, \ldots, x_M) \\ a_2(x_1, \ldots, x_M) \\ \vdots \\ a_K(x_1, \ldots, x_M) \end{pmatrix}$$

The system is assumed to be overconstrained, i.e. $K > M$, and therefore has a unique solution.

At iteration i, we are given x^i (we start with an initial assumption x^0) and solve for the adjustment Δx such that

$$x^{i+1} = x^i + \Delta x \tag{7.10}$$

The adjustment is given by

$$\Delta x = \left(\mathbf{A}^{iT}\mathbf{A}^i\right)^{-1}\mathbf{A}^{iT}(b - a^i) \tag{7.11}$$

where

$$a^i = \begin{pmatrix} a_1(x_1^i, \ldots, x_M^i) \\ a_2(x_1^i, \ldots, x_M^i) \\ \vdots \\ a_K(x_1^i, \ldots, x_M^i) \end{pmatrix}$$

and $\mathbf{A}^i$ is the Jacobian given by

$$\mathbf{A}^i = \begin{pmatrix} \frac{\partial a_1}{\partial x_1} & \frac{\partial a_1}{\partial x_2} & \cdots & \frac{\partial a_1}{\partial x_M} \\ \vdots & & & \vdots \\ \frac{\partial a_K}{\partial x_1} & \frac{\partial a_K}{\partial x_2} & \cdots & \frac{\partial a_K}{\partial x_M} \end{pmatrix}$$

In summary, the following steps have to be performed to solve a nonlinear least squares problem:

1. Obtain by another method an initial guess x^0 for x. This guess should be close to the real x.
2. At every iteration i, compute the Jacobian $\mathbf{A}^i$ and obtain the update value Δx according to equation (7.11).
3. Set $x_{i+1} = x_i + \Delta x$ and iterate until Δx falls below a suitable threshold.

Note that we can reformulate equation (7.11) as a *linear* least-squares problem for the following criterion:

$$\epsilon^2 = (b - a^i - \mathbf{A}^i \Delta x)^T (b - a^i - \mathbf{A}^i \Delta x) \tag{7.12}$$

Usually, five to ten iterations should suffice. Note, however, that the computational complexity of this procedure is rather high. If we can compute the Jacobian in an offline procedure, we basically still have to perform a singular value decomposition at each iteration, an operation that requires a running time of $O(n^3 + m^3)$.

7.2 Traditional Solution

Let us now discuss the most straightforward solution for the absolute pose estimation problem. Similar to the previous chapter, we will only show the computations that have to be performed, but not derive them. The derivation can be found in [24], pp. 131–136.

The standard solution solves the non-linear optimization problem discussed above for the six parameters $t_x, t_y, t_z, \omega, \phi, \kappa$. We assume to have a set of N points having known positions $(x_n, y_n, z_n)^T, n = 1, \ldots, N$ in the real world reference frame and the corresponding set of 2D-perspective projections $(u_n, v_n)^T, n = 1, \ldots, N$ as input data. In addition, we assume that the camera calibration matrix $\mathbf{H}$ is the identity matrix. Note that this assumption is made without loss of generality, as we can always transform the observed image coordinates in a way that yields the values for $\mathbf{H}$ as we assume them to be.

We will now show the values of $\mathbf{A}^i, b$ and a^i. They allow us to solve the problem using the techniques shown in the last chapter, using either equation (7.12) or (7.11).

The vector b contains the $2N$ observed values:

$$b = \begin{pmatrix} u_1 \\ v_1 \\ \vdots \\ u_N \\ v_N \end{pmatrix} \tag{7.13}$$

Things get somewhat more complicated for the vector a^i holding the predicted values of the u_n, v_n assuming that the current values of $t_x, t_y, t_z, \omega, \phi, \kappa$ are correct. First, we have to transform the given three-dimensional coordinates $(x_n, y_n, z_n)^T$ from the world coordinate system to the camera coordinate system:

$$\begin{pmatrix} p_n^i \\ q_n^i \\ s_n^i \end{pmatrix} = \mathbf{R}(\omega^i, \phi^i, \kappa^i) \begin{pmatrix} x_n - x_0^i \\ y_n - y_0^i \\ z_n - z_0^i \end{pmatrix} \tag{7.14}$$

If we now perform the perspective projection on the obtained three-dimensional vector

$$\begin{pmatrix} u_n^i \\ v_n^i \end{pmatrix} = \frac{f}{s_n^i} \begin{pmatrix} p_n^i \\ q_n^i \end{pmatrix} \tag{7.15}$$

we have already a^i:

$$a^i = \begin{pmatrix} u_1^i \\ v_1^i \\ \vdots \\ u_N^i \\ v_N^i \end{pmatrix} \tag{7.16}$$

The Jacobian $\mathbf{A}^i$ uses some derivations of the rotational matrix $\mathbf{R}(\omega, \phi, \kappa) = \mathbf{R}(\kappa)\mathbf{R}(\phi)\mathbf{R}(\omega)$ shown in equation (4.3):

$$\begin{aligned}
\frac{\partial \mathbf{R}}{\partial \omega}(\omega, \phi, \kappa) &= \begin{pmatrix} 0 & \cos\omega\sin\phi\cos\kappa - \sin\omega\sin\kappa & \sin\omega\sin\phi\cos\kappa + \cos\omega\sin\kappa \\ 0 & -\cos\omega\sin\phi\sin\kappa - \sin\omega\cos\kappa & \cos\omega\cos\kappa - \sin\omega\sin\phi\sin\kappa \\ 0 & -\cos\omega\cos\phi & -\sin\omega\cos\phi \end{pmatrix} \\
\frac{\partial \mathbf{R}}{\partial \phi}(\omega, \phi, \kappa) &= \begin{pmatrix} -\sin\phi\cos\kappa & \sin\omega\cos\phi\cos\kappa & -\cos\omega\cos\phi\cos\kappa \\ \sin\phi\sin\kappa & -\sin\omega\cos\phi\sin\kappa & \cos\omega\cos\phi\sin\kappa \\ \cos\phi & \sin\omega\sin\phi & -\cos\omega\sin\phi \end{pmatrix} \\
\frac{\partial \mathbf{R}}{\partial \kappa}(\omega, \phi, \kappa) &= \begin{pmatrix} -\cos\phi\sin\kappa & -\sin\omega\sin\phi\sin\kappa + \cos\omega\cos\kappa & \cos\omega\sin\phi\sin\kappa + \sin\omega\cos\kappa \\ -\cos\phi\cos\kappa & -\sin\omega\sin\phi\cos\kappa - \cos\omega\sin\kappa & \cos\omega\sin\phi\cos\kappa - \sin\omega\sin\kappa \\ 0 & 0 & 0 \end{pmatrix}
\end{aligned}$$

We use these derivations to construct several intermediate matrices; the 3×3 matrix $\mathbf{Q_n}$

$$\begin{aligned}
&\mathbf{Q_n}(\omega^i, \phi^i, \kappa^i, x_0^i, y_0^i, z_0^i) = \\
&\quad \frac{\partial \mathbf{R}}{\partial \omega}(\omega^i, \phi^i, \kappa^i) \begin{pmatrix} x_n - x_0^i \\ y_n - y_0^i \\ z_n - z_0^i \end{pmatrix} \frac{\partial \mathbf{R}}{\partial \phi}(\omega^i, \phi^i, \kappa^i) \begin{pmatrix} x_n - x_0^i \\ y_n - y_0^i \\ z_n - z_0^i \end{pmatrix} \frac{\partial \mathbf{R}}{\partial \kappa}(\omega^i, \phi^i, \kappa^i) \begin{pmatrix} x_n - x_0^i \\ y_n - y_0^i \\ z_n - z_0^i \end{pmatrix}
\end{aligned} \tag{7.17}$$

and the 2×3 matrix $\mathbf{G_n}^i$

$$\mathbf{G_n}^i = \frac{f}{s_n^i} \begin{pmatrix} 1 & 0 & -p_n^i/s_n^i \\ 0 & 1 & -q_n^i/s_n^i \end{pmatrix}$$

and finally the 3×3 matrix $\mathbf{H_n}^i$:

$$\mathbf{H_n}^i = \left(-\mathbf{R}(\omega^i, \phi^i, \kappa^i)\mathbf{Q_n}(\omega^i, \phi^i, \kappa^i, x_0^i, y_0^i, z_0^i)\right)$$

Having done all these preliminary computations, we can now construct the Jacobian $\mathbf{A}^i$:

$$\mathbf{A}^i = \begin{pmatrix} \mathbf{G_1}^i\mathbf{H_1}^i \\ \vdots \\ \mathbf{G_N}^i\mathbf{H_N}^i \end{pmatrix} \tag{7.18}$$

At each iteration, we use equation (7.11) or (7.12) to solve for $\Delta x = (\Delta x_0, \Delta y_0, \Delta z_0, \Delta\omega, \Delta\phi, \Delta\kappa)$ and obtain the new approximate solution by

$$\begin{pmatrix} x_0^{i+1} \\ y_0^{i+1} \\ y_0^{i+1} \\ \omega^{i+1} \\ \phi^{i+1} \\ \kappa^{i+1} \end{pmatrix} = \begin{pmatrix} x_0^{i} \\ y_0^{i} \\ y_0^{i} \\ \omega^{i} \\ \phi^{i} \\ \kappa^{i} \end{pmatrix} + \begin{pmatrix} \Delta x_0 \\ \Delta y_0 \\ \Delta z_0 \\ \Delta\omega \\ \Delta\phi \\ \Delta\kappa \end{pmatrix} \tag{7.19}$$

If the initial approximate values $x_0^0, y_0^0, z_0^o, \omega^0, \phi^0, \kappa^0$ are within a range of 10% of scale for the translational parameter and 15° for the rotational parameters, the algorithm usually terminates within five to ten iterations.

Discussion We have now seen a very simple way to solve the absolute pose estimation problem that uses linear approximation to solve the nonlinear equation systems with the six pose unknowns. Although the formulae may seem complicated, the only tricky thing we have to implement is the Singular Value Decomposition (SVD). However, the SVD is often provided with popular linear algebra libraries like LAPACK (*L*inear *A*lgebra *PACK*age, [7]) or even optimized versions thereof, e.g. the Intel Math Kernel Library ([30]).

Nevertheless, we have to keep in mind some disadvantages of this method. First, its running time is rather slow. At each iteration, we have to do the SVD for a $2N \times 6$ matrix $\mathbf{A}^i$. In addition, several trigonometrical operations and a few matrix multiplications have to be performed.

In addition, we have to give a good initial approximation to obtain a valid solution. During continuous tracking, this may not be a problem, as we can use the values from the last image frame as approximation if we assume that the camera has not been moved too far. However, this requirement may lead to problems during initialization or fast movement.

7.3 Position from Orthographic Scaling

There exists a large variety of different algorithms used for tracking purposes. Horn [28] gives a good overview of some closed-form solutions and Haralick [24] discusses some alternate methods using different representations for the rotational component. Tsai's algorithm ([64]) has been used successfully in a few AR projects. All these algorithms aim to solve the problem exactly.

A rather different approach has been taken by DeMenthon and Davis ([20], [50]). They present an algorithm that sacrifices some accuracy to speed. Due to the hard performance requirements we had for our optical tracker (see section 5.4), we decided to implement this algorithm. The remainder of this section will be a description of the so-called POSIT (*P*osition from *O*rthographic *S*caling with *IT*erations) algorithm.

7.3.1 Background: Scaled Orthographic Projection

In section 5.1 we introduced the perspective camera model as well suited for our tracking needs. Recall that a point M_i with coordinates (X_i, Y_i, Z_i) in the real world is projected onto a point p_i with coordinates

$$x_i = \frac{fX_i}{Z_i} \qquad \text{and} \qquad y_i = \frac{fY_i}{Z_i}$$

The *Scaled Orthographic Projection* camera model is an approximation to the perspective model. It assumes that the depths Z_i of different points M_i in the camera's field of view with coordinates (X_i, Y_i, Z_i) in the camera's coordinate system do not differ too much from one another relative to the overall distance from the camera, i.e. the coordinate system's origin. In consequence, all depths can be set to the depth Z_0 of a reference point M_0. Point M_i is therefore projected onto a point m_i with coordinates

$$x'_i = \frac{fX_i}{Z_0} \qquad \text{and} \qquad y'_i = \frac{fY_i}{Z_0} \tag{7.20}$$

The ratio $s = f/Z_0$ is called the *scaling factor* of the projection. Note that for the reference point M_0, the perspective projection yields the same result as the scaled orthographic projection. In summary, we can rewrite equation (7.20) as follows:

$$\begin{aligned} x'_i &= \frac{fX_0}{Z_0} + \frac{f}{Z_0}(X_i - X_0) = x_0 + s(X_i - X_0) \\ y'_i &= y_0 + s(Y_i - Y_0) \end{aligned} \tag{7.21}$$

Geometric Interpretation Figure 7.1 shows how to construct the perspective projection point m_i and the scaled orthographic projection (SOP) point p_i out of a given 3D point M_i. We obtain m_i as the intersection of the line of sight from the camera coordinate system origin O to M_i with the image plane $\mathbf{G}$. $\mathbf{G}$'s distance from O is the focal length f. For SOP, we have to draw a plane $\mathbf{K}$ parallel to $\mathbf{G}$ at distance Z_0 from O. Now we can orthographically project M_i on $\mathbf{K}$ and obtain a point P_i. The latter is projected on the image plane $\mathbf{G}$ by a perspective projection, finally yielding p_i. Note that m_0p_i is parallel to M_0P_i and scaled down from M_0P_i by the factor $s = f/Z_0$. This is exactly the same as equation (7.21).

Parameters to be Obtained Although we dealt with scaled orthographic projection, we have not yet mentioned how we can obtain the camera pose using SOP. Things should get more clear after a look at figure 7.1. The vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ span the world coordinate system, the vectors $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ the camera's CS. If we normalize $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ and represent them in the $(\mathbf{u}, \mathbf{v}, \mathbf{w})$ system, we already have the rotational matrix

$$\mathbf{R} = \begin{pmatrix} i_u & i_v & i_w \\ j_u & j_v & j_w \\ k_u & k_v & k_w \end{pmatrix} \tag{7.22}$$

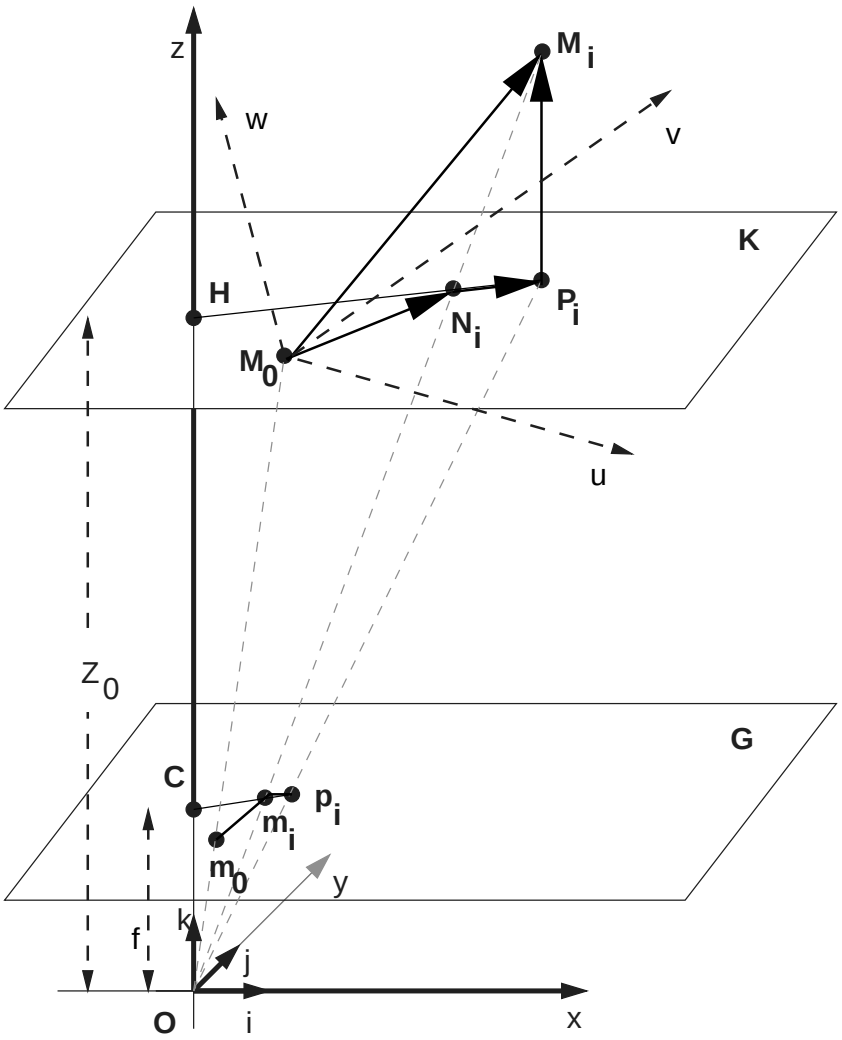


Figure 7.1: Scaled Orthographic Projection and Perspective Projection (Courtesy of DeMenthon and Davis)

To compute the translational component, we assume to have determined $\mathbf{R}$ and the position of M_0 in the camera's coordinate system. We denote this position by $(OM_0)_C$. Obviously, we can represent $(OM_0)_C$ in the world coordinate system:

$$(OM_0)_W = \mathbf{R}^{-1} \cdot (OM_0)_C$$

If we now look at figure 7.2, we realize that we are almost done. We want to obtain the translational vector $\mathbf{t}$ which is identical to the camera's coordinate system's origin O. We have

$$\begin{aligned}
\mathbf{t} &= O_w O = (O_W M_0)_W - (OM_0)_W \\
&= (M_0)_W - \mathbf{R}^{-1}(OM_o)_C \\
&= \begin{pmatrix} X_0^W \\ Y_0^W \\ Z_0^W \end{pmatrix} - \begin{pmatrix} i_u & i_v & i_w \\ j_u & j_v & j_w \\ k_u & k_v & k_w \end{pmatrix}^{-1} \begin{pmatrix} X_0 \\ Y_0 \\ Z_0 \end{pmatrix} \\
&= \begin{pmatrix} X_0^W \\ Y_0^W \\ Z_0^W \end{pmatrix} - \begin{pmatrix} i_u & j_u & k_u \\ i_v & j_v & k_v \\ i_w & j_w & k_w \end{pmatrix} \begin{pmatrix} X_0 \\ Y_0 \\ Z_0 \end{pmatrix}
\end{aligned}$$

using the fact that $\mathbf{R}$ is an orthonormal matrix with $\mathbf{R}^{-1} = \mathbf{R}^T$.

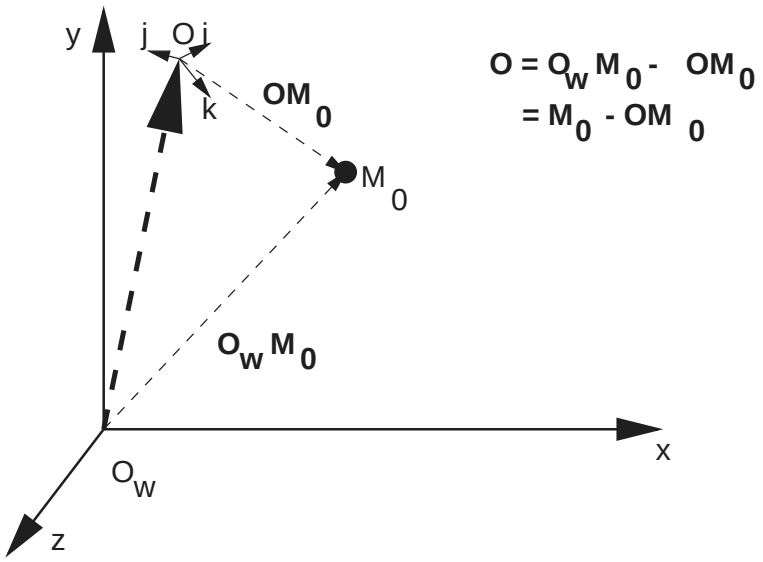


Figure 7.2: Obtaining the Camera's Translation

Note that $\mathbf{R}$ is orthonormal and that therefore $\mathbf{i} \times \mathbf{j} = \mathbf{k}$ holds true. In consequence, it is completely sufficient to determine the vectors $\mathbf{i}$, $\mathbf{j}$ and the parameters X_0, Y_0 and Z_0 to compute the camera's pose. To make life even easier, we have to keep in mind that given Z_0 we readily can compute $X_0 = x_0/s$ and $Y_0 = y_0/s$ using equation (7.20) for the known image coordinates of M_0.

Fundamental Equations We will now derive some fundamental equations that lie at the heart of the POSIT algorithm. We can see in figure 7.1 that the vector M_0M_i is the sum of three vectors:

$$M_0M_i = M_0N_i + N_iP_i + P_iM_i \tag{7.23}$$

The vector m_0m_i is M_0N_i's image. We therefore can write

$$M_0N_i = \frac{1}{s} \cdot m_om_i = \frac{Z_0}{f} \cdot m_0m_i$$

Using simple geometrics, we see that the triangles Om_iC and $M_iN_iP_i$ are similar. The ratio between these triangles is identical to the ratio between P_iM_i's and OC's z–components. This ratio is $M_0M_i \cdot \mathbf{k}/f$. We therefore have

$$\begin{aligned} & \frac{N_iP_i}{M_0M_i \cdot \mathbf{k}} = \frac{Cm_i}{f} \\ \iff \quad & N_iP_i = \frac{M_0M_i \cdot \mathbf{k}}{f} Cm_i \end{aligned}$$

Now the sum (7.23) can be expressed as

$$M_oM_i = \frac{Z_0}{f} \cdot m_0m_i + \frac{M_0M_i \cdot \mathbf{k}}{f} Cm_i + P_iM_i \tag{7.24}$$

If we now finally take the dot product with the unit vector $\mathbf{i}$ with both sides of the equation, we state that $P_iM_i \cdot \mathbf{i} = 0$, that $m_0m_i \cdot \mathbf{i}$ is the x–coordinate $x_i - x_0$ of the vector m_0m_i, and that $Cm_i \cdot \mathbf{i}$ is the coordinate x_i of Cm_i. Doing the same procedure with the unit vector $\mathbf{j}$, we have the equations

$$\begin{aligned} M_0M_i \cdot \frac{f}{Z_0} \cdot \mathbf{i} &= x_i - x_0 + \frac{M_0M_i \cdot \mathbf{k}}{Z_0} x_i \\ M_0M_i \cdot \frac{f}{Z_0} \cdot \mathbf{j} &= y_i - y_0 + \frac{M_0M_i \cdot \mathbf{k}}{Z_0} y_i \end{aligned}$$

Defining the terms ε_i as

$$\varepsilon_i = \frac{1}{Z_0} M_0M_i \cdot \mathbf{k} = \frac{1}{Z_0} M_0M_i \cdot (\mathbf{i} \times \mathbf{j}) \tag{7.25}$$

we now derived the fundamental equations

$$M_0M_i \cdot \frac{f}{Z_0} \cdot \mathbf{i} = x_i(1 + \varepsilon_i) - x_0 \tag{7.26}$$

$$M_0M_i \cdot \frac{f}{Z_0} \cdot \mathbf{j} = y_i(1 + \varepsilon_i) - y_0 \tag{7.27}$$

Consider the vector M_0M_i. It can be represented as $M_0M_i = M_0P_i + P_iM_i$. As m_0p_i is M_0P_i's projection with a scale of Z_0/f, we can write

$$\begin{aligned} & M_oM_i = m_0p_i \cdot \frac{Z_0}{f} + P_iM_i \\ \iff \quad & M_0M_i \cdot \frac{f}{Z_0} = m_0p_i + P_iM_i \cdot \frac{Z_0}{f} \end{aligned}$$

Taking again the dot product with $\mathbf{i}$, we obtain

$$M_0M_i \cdot \frac{f}{Z_0} \cdot \mathbf{i} = x_i' - x_0 \tag{7.28}$$

with x_i' being p_i's x–coordinate. Similarly we can derive

$$M_0M_i \cdot \frac{f}{Z_0} \cdot \mathbf{j} = y_i' - y_0 \tag{7.29}$$

If we compare these equations with equations (7.26) and (7.27), we immediately see that

$$\begin{aligned} x_i' &= x_i(1+\varepsilon_i) \\ y_i' &= y_i(1+\varepsilon_i) \end{aligned}$$

POSIT's Basic Idea If we set

$$\mathbf{I} = \frac{f}{Z_0}\mathbf{i} \qquad \text{and} \qquad \mathbf{J} = \frac{f}{Z_0}\mathbf{j}$$

we can rewrite equations (7.26) and (7.27) as

$$M_0M_i \cdot \mathbf{I} = x_i(1+\varepsilon_i) - x_0 \tag{7.30}$$

$$M_0M_i \cdot \mathbf{J} = y_i(1+\varepsilon_i) - y_0 \tag{7.31}$$

Let us now assume that we assign values to the ε_i. Equations (7.30) and (7.31) provide a linear system that allows the computation of $\mathbf{I}$ and $\mathbf{J}$. As $\mathbf{i}$ and $\mathbf{j}$ are normalized vectors, we obtain them by normalizing $\mathbf{I}$ and $\mathbf{J}$ and Z_0 by the norm of either $\mathbf{I}$ or $\mathbf{J}$.

If the given values of ε_i are not exact, the solutions just obtained are only approximations. However, we can compute *more precise* values for ε_i out of $\mathbf{i}$, $\mathbf{j}$ and Z_0. Iterating a few times, the algorithm should converge towards values of $\mathbf{i}$, $\mathbf{j}$ and Z_0 that correspond to a correct pose.

7.3.2 The Simple Case: Non-Coplanar Points

We still have to discuss how to solve equations (7.30) and (7.31) for a given set of image points $M_0, M_1, \ldots, M_n$. As a first step, we build the $n \times 3$ matrix $\mathbf{A}$ and the n–dimensional vectors $\mathbf{x}'$ and $\mathbf{y}'$:

$$\begin{aligned} \mathbf{A} &= \begin{pmatrix} M_0M_1 \\ M_0M_2 \\ \vdots \\ M_0M_n \end{pmatrix} \\ \mathbf{x}' &= \begin{pmatrix} x_1(1+\varepsilon_1) - x_0 \\ \vdots \\ x_n(1+\varepsilon_n) - x_0 \end{pmatrix} \\ \mathbf{y}' &= \begin{pmatrix} y_1(1+\varepsilon_1) - y_0 \\ \vdots \\ y_n(1+\varepsilon_n) - y_0 \end{pmatrix} \end{aligned}$$

Now we can rewrite the linear system to be solved:

$$\mathbf{AI} = \mathbf{x}', \qquad \mathbf{AJ} = \mathbf{y}' \tag{7.32}$$

If we assume to have at least three *noncoplanar* points other than M_0, i.e. $n \geq 3$, the matrix $\mathbf{A}$ has rank 3 and we have a unique solution in the least square sense given by

$$\mathbf{I} = \mathbf{A}^{+}\mathbf{x}', \qquad \mathbf{J} = \mathbf{A}^{+}\mathbf{y}' \tag{7.33}$$

with $\mathbf{A}^{+}$ being the pseudoinverse of $\mathbf{A}$ discussed in section 7.1.1.

To compute $\mathbf{A}^{+}$, we preferably use the singular value decomposition, as it gives us information about $\mathbf{A}$'s rank before computing the pseudoinverse. Therefore we can check if all points are coplanar before the computation of $\mathbf{I}$ and $\mathbf{J}$.

Normalization of $\mathbf{I}$ and $\mathbf{J}$ yields $\mathbf{i}$, $\mathbf{j}$ and Z_0. These values are afterwards used to compute new values for ε_i and in consequence $\mathbf{x}'$ and $\mathbf{y}'$ at the algorithm's next iteration.

In summary, we have to do *only a single* singular value decomposition during the algorithm's computation. This is a major advantage in speed over the traditional approach that has to perform a SVD at every iteration.

7.3.3 The Hard Case: Coplanar Points

Note that we assumed the image points to be noncoplanar. What happens if this requirement is not fulfilled? Figure 7.3 shows that due to the orthographic projection involved in the SOP process, two solutions are now possible.

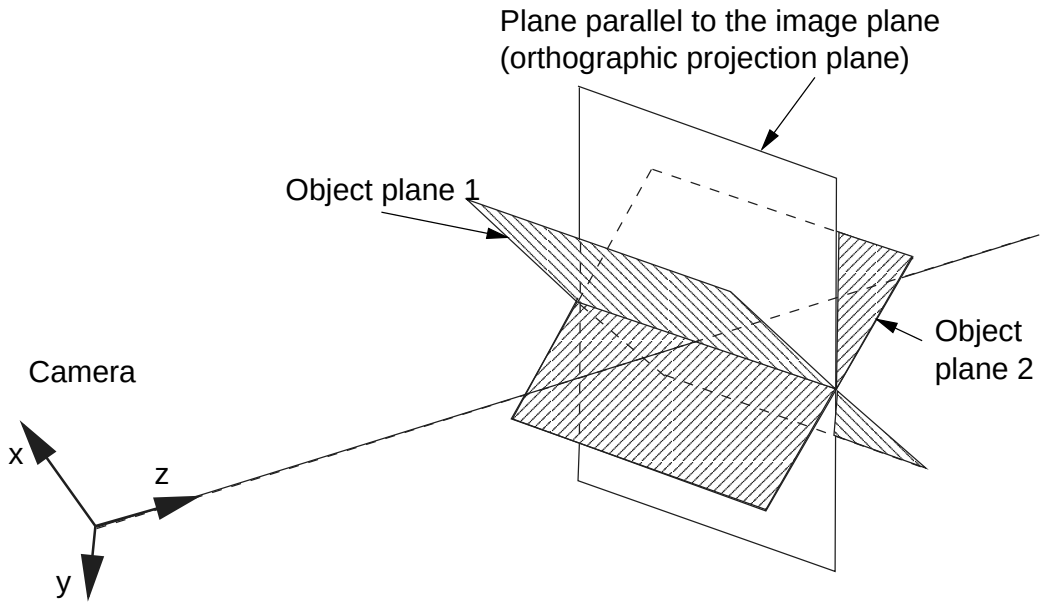


Figure 7.3: Two Possible Solutions from Orthographic Scaling with Coplanar Points (Courtesy of DeMenthon and Davis)

However, the rank of the matrix $\mathbf{A}$ is now only 2. In consequence, we get an infinite number of solutions for $\mathbf{I}$ and $\mathbf{J}$ if we do not impose additional constraints on our system. To derive these constraints, we have to analyze geometrically which solutions are yielded by the linear systems. We have the following equation for the vector $\mathbf{I}$:

$$M_0M_i \cdot \mathbf{I} = x_i(1 + \varepsilon_i) - x_0 = x_i'$$

This expression states that if we take the tail of $\mathbf{I}$ at point M_0, its head projects on M_0M_i at a point H_{x_i} defined by

$$|M_0H_{x_i}| = \frac{x_i'}{|M_0M_i|} \tag{7.34}$$

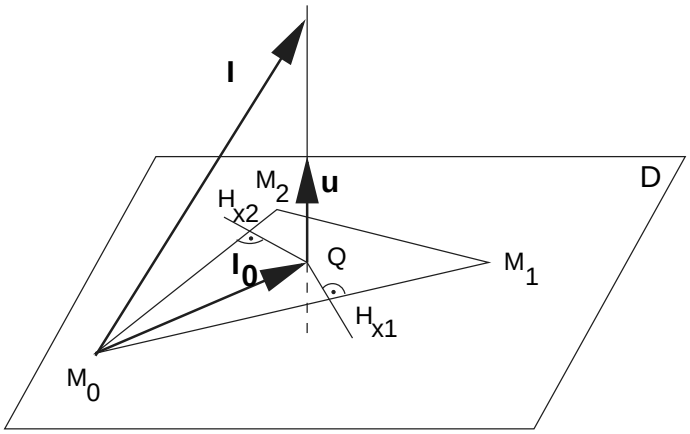


Figure 7.4: Solutions for $\mathbf{I}$ in the Coplanar Case (Courtesy of DeMenthon and Davis)

Every point from the plane perpendicular to M_0M_i at H_{x_i} is projected to H_{x_i}. As shown in figure 7.4 for the case of $n = 2$, the intersection of all planes perpendicular to the points H_{x_i} is a single line (or close parallel lines if noise is in the input data) perpendicular to the plane D containing all coplanar points M_i. In consequence, we can write all solutions as

$$\mathbf{I} = \mathbf{I}_0 + \lambda\mathbf{u} \tag{7.35}$$

with $\mathbf{u}$ being the unit vector perpendicular to D. Note that $\mathbf{u}$ spans the null space of $\mathbf{A}$, we obtain it therefore by taking the column vector corresponding to the smallest singular value of the second orthonormal matrix $\mathbf{V}$ from $\mathbf{A}$'s singular value decomposition. $\mathbf{I}_0$ is the minimum norm solution to the system $\mathbf{AI} = \mathbf{x}'$ and is computed by

$$\mathbf{I}_0 = \mathbf{A}^+\mathbf{x}' \tag{7.36}$$

Similarly, we get

$$\mathbf{J} = \mathbf{J}_0 + \mu\mathbf{u} = \mathbf{A}^+\mathbf{y}' + \mu\mathbf{u} \tag{7.37}$$

It remains to show which two solutions (λ_1, μ_1) and (λ_2, μ_2) are feasible. We have to use two facts:

1. **I** and **J** must be perpendicular:

$$\begin{aligned} & \mathbf{I} \cdot \mathbf{J} = 0 \\ \iff & (\mathbf{I}_0 + \lambda\mathbf{u}) \cdot (\mathbf{J}_0 + \mu\mathbf{u}) = 0 \\ \iff & \mathbf{I}_0 \cdot \mathbf{J}_0 + \lambda\mu = 0 \\ \iff & \lambda\mu = -\mathbf{I}_0\mathbf{J}_0 \end{aligned} \tag{7.38}$$

2. **I** and **J** must have the same length:

$$\begin{aligned} & \mathbf{I} \cdot \mathbf{I} = \mathbf{J} \cdot \mathbf{J} \\ \iff & (\mathbf{I}_0 + \lambda\mathbf{u}) \cdot (\mathbf{I}_0 + \lambda\mathbf{u}) = (\mathbf{J}_0 + \mu\mathbf{u}) \cdot (\mathbf{J}_0 + \mu\mathbf{u}) \\ \iff & \mathbf{I}_0^2 + \lambda^2 = \mathbf{J}_0^2 + \mu^2 \\ \iff & \lambda^2 - \mu^2 = \mathbf{J}_0^2 - \mathbf{I}_0^2 \end{aligned} \tag{7.39}$$

To solve this system for λ, μ we note that the square of the complex number $C = \lambda + i\mu$ is $C^2 = \lambda^2 - \mu^2 + 2i\lambda\mu$ and therefore

$$C^2 = \mathbf{J}_0^2 - \mathbf{I}_0^2 - 2i\mathbf{I}_0\mathbf{J}_0 \tag{7.40}$$

We can now find the two solutions for λ and μ as the real and imaginary parts of the square roots of C^2. To find the square roots, we write C^2 in polar form:

$$\begin{aligned} C^2 &= [R, \Theta], \qquad \text{with} \\ R &= \left((\mathbf{J}_0^2 - \mathbf{I}_0^2)^2 + 4(\mathbf{I}_0 \cdot \mathbf{J}_0)^2\right)^{1/2} \qquad (7.41) \\ \Theta &= \begin{cases} \arctan\left(\frac{-2\mathbf{I}_0 \cdot \mathbf{J}_0}{\mathbf{J_0}^2 - \mathbf{I_0}^2}\right), & \text{if } \mathbf{J}_0^2 - \mathbf{I}_0^2 > 0 \\ \arctan\left(\frac{-2\mathbf{I}_0 \cdot \mathbf{J}_0}{\mathbf{J_0}^2 - \mathbf{I_0}^2}\right) + \pi, & \text{if } \mathbf{J}_0^2 - \mathbf{I}_0^2 < 0 \\ -\text{sign}(\mathbf{I}_0 \cdot \mathbf{J}_0)\frac{\pi}{2}, & \text{else} \end{cases} \qquad (7.42) \end{aligned}$$

The two roots C are $C_1 = [\rho, \theta]$ and $C_2 = [\rho, \theta + \pi]$ with

$$\rho = \sqrt{R}, \qquad \text{and} \qquad \theta = \frac{\Theta}{2} \tag{7.43}$$

The real and imaginary parts of C_1 and C_2 are the two solutions for λ and μ:

$$\begin{aligned} \lambda_1 &= \rho\cos\theta, & \mu_1 &= \rho\sin\theta \\ \lambda_2 &= -\rho\cos\theta, & \mu_2 &= -\rho\sin\theta \end{aligned} \tag{7.44}$$

Choosing the Correct Pose As shown in figure 7.3 and just derived, the basic POSIT equations yield two solutions if all image points are coplanar. After N iterations, we have 2^N possible poses if no additional measures are taken. This is obviously not feasible. DeMenthon and Davis [50] propose and prove the procedure described in the remainder of this section.

We first have to note that the translational component is the same for both solutions obtained at every step (compare figure 7.3). For every solution (λ, μ) we can compute the corresponding rotational matrix $\mathbf{R}$. Once we have this matrix, we obtain easily every M_i's

coordinates in the camera coordinate system. If a single Z_i is negative, we can discard the corresponding pose, as every M_i must be in front of the camera. If this situation occurs at the first iteration, it will occur in all subsequent steps, too.

If after the first iteration both poses are feasible, we pursue both. For all subsequent iterations, we only pursue *the better pose*. To select the better pose, we have to introduce a measure of quality. We use the distance measure E, the average distance between the actual image points m_i and the projected points from the computed pose.

7.3.4 Summary

Now we are done with the description of the POSIT algorithm. To sum it up, we give an implementation in pseudo code:

begin
Fill the matrix **A** with rows M_0M_i
Compute Pseudoinverse $\mathbf{A}^+$ with SVD
for $i := 1$ **to** N **do** $\varepsilon_{i(0)} := 0$ **od**
if $rank(\mathbf{A}) == 1$
then $exit$ Collinear Points
else if $rank(\mathbf{A}) == 2$
then do Coplanar Points
$\mathbf{u}$:= Third row of matrix $\mathbf{V}$ of SVD
Compute the vectors $\mathbf{x}'$ and $\mathbf{y}'$
Compute $\mathbf{I}_0 := \mathbf{A}^+\mathbf{x}', \mathbf{J}_0 := \mathbf{A}^+\mathbf{x}', \lambda_1, \mu_1, \lambda_2, \mu_2, \mathbf{I}_1, \mathbf{J}_1, \mathbf{I}_2, \mathbf{J}_2$
Compute the scale s of projection as average norm of $\mathbf{I}_1$ and $\mathbf{J}_1$
and the translational vector t
Compute unit vectors $\mathbf{i}_1$, $\mathbf{j}_1$ and $\mathbf{i}_2$, $\mathbf{j}_2$
Compute cross products $\mathbf{k}_1 = \mathbf{i}_1 \times \mathbf{j}_1$ and $\mathbf{k}_2 = \mathbf{i}_2 \times \mathbf{j}_2$ to get rotational
matrices $\mathbf{R}_1$ and $\mathbf{R}_2$
Check both poses for feasability
if Both are feasible
then do
Iterate on both by updating ε, always keep the better of the
two poses yielded at each step, and stop if the error
measure E falls below a suitable threshold
od
else do
Iterate on the single feasible pose by updating ε, at every
subsequent step only one pose will be feasible. Stop if
the error measure E falls below a suitable threshold
od
fi
od
fi
else do Noncoplanar Points

repeat
Compute the vectors $\mathbf{x}'$ and $\mathbf{y}'$
Compute $\mathbf{I} := \mathbf{A}^{+}\mathbf{x}', \mathbf{J} := \mathbf{A}^{+}\mathbf{x}'$
Compute the scale s of projection as average norm of $\mathbf{I}$ and $\mathbf{J}$ and the translational vector t
Compute unit vectors $\mathbf{i}$, $\mathbf{j}$
Compute cross product $\mathbf{k} = \mathbf{i} \times \mathbf{j}$ to get rotational matrix $\mathbf{R}$
Compute $\varepsilon_{i(n)} = \frac{1}{Z_0} M_0 M_i \cdot \mathbf{k}$
until $|\varepsilon_{i(n)} - \varepsilon_{i(n-1)}| < threshold$
od
fi
Output current $\mathbf{R}$ and t
end

This ends our discussion of absolute pose estimation. We will give some results on the implementation of the POSIT algorithms in terms of accuracy, stability and speed in chapter 11.

8 Relative Pose Estimation Using Optical Flow

Having discussed the details of fiducial detection and the determination of absolute pose information, we have ignored the fact that we are not analyzing a single image but rather a continuous sequence of images taken every few milliseconds. Typical framerates of video cameras range from 10 to 30 frames per second *(fps)*, and in common environments of trackers we can safely assume that large portions of video frame n are still visible in frame $n+1$.

In this chapter, we will present some methods on how we could use this fact. First, we will discuss a definition of *Optical Flow*, an image understanding technique that gives us information about the changes between frame n and $n+1$. Afterwards, we present an introduction on the mathematical methods that can be used to process this information for pose estimation.

However, we did not use these methods directly in the optical tracker that has been developed for this thesis, and a discussion of the reasons can be found in chapter 11. We used a different approach that will be described in detail in the next chapter.

8.1 Optical Flow

What is Optical Flow? To give a well understandable motivation, assume the following: a video camera takes a picture of a white wall with a black spot on it. The camera is moved counterclockwise. In the recorded video, the spot moves therefore from left to right. Let us assume that at frame n, the spot is located at the coordinates $\mathbf{x}_1 = (100, 100)^T$, and at frame $n+1$, it is located at $\mathbf{x}_2 = (105, 90)^T$. As illustrated in figure 8.1, the optical flow is the vector $\mathbf{d} = \mathbf{x}_2 - \mathbf{x}_1 = (5, -10)^T$.

Let us now define the term optical flow more precisely.

8.1.1 Definition

Note: There exist a few definitions of optical flow that differ in many details. Our definition is based on [17].

We assume I (the first) and J (the second) to be two 2D grayscaled images. The values $I(\mathbf{x}) = I(x, y)$ and $J(\mathbf{x}) = J(x, y)$ denote the grayscale value of the two images at the pixel with location $\mathbf{x} = (x, y)$. We assume the pixel at location $\mathbf{0} = (0, 0)$ to be situated at the top left corner of each image. Both images have width n_x and height n_y. In consequence, the lower right corner pixel has coordinates $(n_x - 1, n_y - 1)$

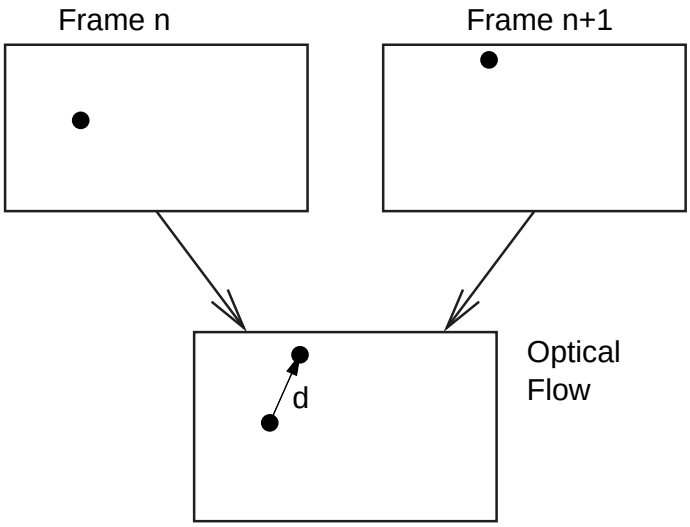


Figure 8.1: Simple Illustration of Optical Flow

Consider now an image point $\mathbf{u} = (u_x, u_y)^T$ in the first image I. We want to detect $\mathbf{u}$'s position in the second image J. We denote this position by $\mathbf{v} = \mathbf{u} + \mathbf{d} = (u_x + d_x, u_y + d_y)^T$. The vector $\mathbf{d} = (d_x, d_y)^T$ is the *image velocity* or *optical flow* at $\mathbf{x}$.

The basic assumption of all optical flow algorithms is that the image brightness is in general invariate over time. In consequence, we have to detect a point $\mathbf{v}$ in image J in a neighborhood of $\mathbf{u}$'s position in I that has the most similar grayscale value. For our purposes, this neighborhood is a so-called *integration window* of width $2\omega_x + 1$ and height $2\omega_y + 1$. We therefore define the optical flow $\mathbf{d}$ as being the vector that minimizes the residual function ε. ε is defined in a least-squares sense such that $\mathbf{d}$ is the vector that transforms all pixels of the integration window W to another window W' such that the pixelwise difference between W and W' gets minimized:

$$\varepsilon(\mathbf{d}) = \varepsilon(d_x, d_y) = \sum_{x=u_x-\omega_x}^{u_x+\omega_x} \sum_{y=u_y-\omega_y}^{u_y+\omega_y} (I(x,y) - J(x+d_x, y+d_y))^2 \tag{8.1}$$

As mentioned above, we assume to be the optical flow rather low. It follows that a choice in the range of $2 \ldots 7$ for ω_x and ω_y is reasonable to minimize computational expense.

8.1.2 Lucas-Kanade Method in Pyramids

There exist many algorithms to compute the optical flow between two images. A description and implementation of several can be found in [31]. In the time-restricted context of this thesis, we chose Intel's pyramidal implementation of the Lucas Kanade feature tracker.

In this section, we will describe the algorithm. However, this discussion is on a rather high level, for algorithmic details we refer the reader to [17].

8.1.2.1 Image Pyramid Representation

In order to handle motions (d_x, d_y) correctly, it is important that the size of the integration window is larger than the maximum expected image velocity, i.e. $\omega_x \geq d_x$ and $\omega_y \geq d_y$. This is a computational problem for fast head rotations resulting in very large flow vectors.

A pyramid representation of the input images can solve this dilemma. Assume we have an image I of size $n_x \times n_y$. Let $I^0 = I$ be the image at level "zero". We now build the images at other levels in a recursive way. First, we create I^1 out of I^0, afterwards we can create I^2 out of I^1, and finally we create I^L out of I^{L-1}. Note that the image I^l has size $n_x^l \times n_y^l$. We now define the image $I^l(x, y)$ as follows:

$$\begin{aligned} I^l(x,y) = \frac{1}{4} I^{l-1}(2x, 2y) + \\ \frac{1}{8}\Big(I^{l-1}(2x-1, 2y) + I^{l-1}(2x+1, 2y) + I^{l-1}(2x, 2y-1) + I^{l-1}(2x, 2y+1) \Big) + \\ \frac{1}{16}\Big(I^{l-1}(2x-1, 2y-1) + I^{l-1}(2x+1, 2y+1) + \\ I^{l-1}(2x-1, 2y+1) + I^{l-1}(2x+1, 2y-1) \Big) \end{aligned} \tag{8.2}$$

In order to process the borders of the images correctly, we define dummy values around the image I^{l-1} for $0 \leq x \leq n_x^{l-1} - 1$ and $0 \leq y \leq n_y^{l-1} - 1$:

$$\begin{aligned} I^{l-1}(-1,-1) &\doteq I^{l-1}(0,0) \\ I^{l-1}(-1,y) &\doteq I^{l-1}(0,y) \\ I^{l-1}(x,-1) &\doteq I^{l-1}(x,0) \\ I^{l-1}(n_x^{l-1},y) &\doteq I^{l-1}(n_x^{l-1}-1,y) \\ I^{l-1}(x,n_y^{l-1}) &\doteq I^{l-1}(x,n_y^{l-1}-1) \\ I^{l-1}(n_x^{l-1},n_y^{l-1}) &\doteq I^{l-1}(n_x^{l-1}-1,n_y^{l-1}-1) \end{aligned}$$

If we look at equation (8.2), it is easy to see that the image sizes of I^l hold

$$n_x^l = \left\lfloor \frac{n_x^{l-1} + 1}{2} \right\rfloor \tag{8.3}$$

$$n_y^l = \left\lfloor \frac{n_y^{l-1} + 1}{2} \right\rfloor \tag{8.4}$$

8.1.2.2 Optical Flow Tracking with Pyramid Representation

Recall our problem statement. Given point $\mathbf{u} = (u_x, u_y)^T$ in image I, we want to find the corresponding point $\mathbf{v} = \mathbf{u} + \mathbf{d}$ in image J.

Equation (8.2) gives us $\mathbf{u}$'s coordinates $\mathbf{u}^l$ in image I^l:

$$\mathbf{u}^l = \frac{\mathbf{u}}{2^l} \tag{8.5}$$

In particular, $\mathbf{u_0} = \mathbf{u}$.

The algorithm works now as follows:

1. Build pyramid representations of I and J at levels $l = 1, \ldots, L$.
2. Initialize optical flow guess $\mathbf{g}^L = (0, 0)^T$
3. **for** $l = L$ **down to** 0 **do**
   a) Compute location of point $\mathbf{u}$ on image I^l: $\mathbf{u}^l = \mathbf{u}/2^l$
   b) Compute the optical flow $\mathbf{d}^l$ at level l (see below for details) using the initial guess $\mathbf{g}^l$.
   c) Compute the guess value for the next level, $\mathbf{g}^{l-1} = 2 \cdot (\mathbf{g}^l + \mathbf{d}^l)$.
4. Compute the final optical flow $\mathbf{d} = \mathbf{g}^0 + \mathbf{d}^0$. The location of the point on image J is now $\mathbf{v} = \mathbf{u} + \mathbf{d}$.

Although we still have to discuss the core optical flow algorithm, it should now be clear that this pyramid approach gives good local tracking accuracy in combination with high robustness, i.e. the ability to handle large movements without prohibiting cost of computation.

8.1.2.3 Iterative Lucas Kanade Algorithm

Finally, we can describe the algorithm at the heart of our optical flow routine. According to equation (8.1), given a point $\mathbf{u}$ for which we want to compute the optical flow, we have to determine the vector $\mathbf{d}$ that minimizes the residual function ε. It should be clear that at this minimum, the first derivate of ε with respect to $\mathbf{u}$ is zero:

$$\frac{\partial \varepsilon(\mathbf{u})}{\partial \mathbf{u}} \stackrel{!}{=} 0 \tag{8.6}$$

To solve this minimization problem, we first have to compute some image derivatives:

$$I_x(x, y) = \frac{\partial I(x, y)}{\partial x} = \frac{A(x+1, y) - A(x-1, y)}{2} \tag{8.7}$$

$$I_y(x, y) = \frac{\partial I(x, y)}{\partial y} = \frac{A(x, y+1) - A(x, y-1)}{2} \tag{8.8}$$

Then we have to determine the change in the image brightness value using our current optical flow estimate $\mathbf{g}$:

$$\delta I(x, y) = I(x, y) - J(x + g_x, y + g_y) \tag{8.9}$$

Applying some further derivations similar to the ones discussed in section 7.1 and first order Taylor expansion approximations described in detail in [17], we finally obtain the optimum optical flow vector

$$\mathbf{d_{opt}} = \mathbf{g} + \nu_{\mathbf{opt}} = \mathbf{g} + \mathbf{G}^{-1} \cdot \mathbf{b} \tag{8.10}$$

with

$$\mathbf{G} \doteq \sum_{x=u_x-\omega_x}^{u_x+\omega_x} \sum_{y=u_y-\omega_y}^{u_y+\omega_y} \begin{pmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{pmatrix} \tag{8.11}$$

$$\mathbf{b} = \sum_{x=u_x-\omega_x}^{u_x+\omega_x} \sum_{y=u_y-\omega_y}^{u_y+\omega_y} \begin{pmatrix} \delta I \cdot I_x \\ \delta I \cdot I_y \end{pmatrix} \tag{8.12}$$

Due to the Taylor expansion approximation used, this computation is only valid if the pixel displacement is small. To ensure a working algorithm for larger displacements, the algorithm has to be iterated through several times. Note that the repeated computation is not as time consuming as it might seem on first sight, as the matrix $\mathbf{G}$ remains constant and only $\mathbf{b}$ has to be recomputed. At every iteration i, we obtain an update value ν_i. This value is added to the current guess value and the algorithm is done again with the new guess value. At iteration i, we have the guess value

$$\mathbf{g}_i = \mathbf{g}_0 + \sum_{j=0}^{i-1} \nu_j \tag{8.13}$$

The process is stopped either after a defined number of iterations or if ν_i falls below a suitable threshold.

Note that we use the pyramid implementation of this algorithm to obtain good initial guess values $\mathbf{g}_0$ at every pyramid step. At the lowest pyramid level, we simply set $\mathbf{g}_0 = (0,0)^T$.

8.2 The Relative Orientation Problem

Having solved the optical flow between two images, let us now resume which data we have:

1. The internal camera parameters, especially the focal length f.
2. A set of N two-dimensional point-to-point correspondences $\{(u_{Ln}, v_{Ln}, u_{Rn}, v_{Rn})\}_{n=1}^{N}$ of points in the left (or first) and right (or second) image frame. The point (u_{Ln}, v_{Ln}) on the left image corresponds to the point (u_{Rn}, v_{Rn}) on the right image if there is some 3D point $\mathbf{x} = (x_n, y_n, z_n)$ such that (u_{Ln}, v_{Ln}) is its perspective projection on the left image frame and (u_{Rn}, v_{Rn}) is its perspective projection on the right image frame.
3. All image points are expressed with identical scale and with respect to their principal point.

Which data do we want to obtain? The relative orientation is defined as the reference frame of the right image with respect to the left image's reference frame. Again, we use the perspective camera model and assume all intrinsic parameters to be known. In consequence, we want to obtain the six degrees of freedom matrix $\mathbf{D}$ describing the right image's reference frame with respect to the left image's.

Since we can determine the necessary six parameters only up to an arbitrary scale factor, we take the distance of the left and right camera's lense's positions along the x–axis as a constant that controls the scale.

In summary, we have to compute the following values:

1. Translation along the y–axis: $y_R - y_L$
2. Translation along the z–axis: $z_R - z_L$
3. Rotation around the x–axis: $\omega_R - \omega_L$
4. Rotation around the y–axis: $\phi_R - \phi_L$
5. Rotation around the z–axis: $\kappa_R - \kappa_L$

8.2.1 Mathematical Background

In this section, we will discuss and derive fundamental equations representing our problem. These equations will be solved in the next section. Note that this derivation has been taken from [24, pp. 144–147]. Further details and alternative solutions can be found in this book.

Let

$$\mathbf{Q}_L^T = \mathbf{R}(\omega_L, \phi_L, \kappa_L) \tag{8.14}$$

be the rotation matrix for the left image and accordingly

$$\mathbf{Q}_R^T = \mathbf{R}(\omega_R, \phi_R, \kappa_R) \tag{8.15}$$

be the right image's rotation matrix. Similarly in notation,

$$\mathbf{x}_L = \begin{pmatrix} x_L \\ y_L \\ z_L \end{pmatrix} \qquad \text{and} \qquad \mathbf{x}_R = \begin{pmatrix} x_R \\ y_R \\ z_R \end{pmatrix} \tag{8.16}$$

are the left and right image's translational vectors. If f_R is the distance between the right image plane and the right lense's front and f_L is the distance between the left image plane and the left lense's front[1], we get from the perspective collinearity equations

$$\begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} = \frac{1}{\lambda_{Ln}} \mathbf{Q}_L^T \begin{pmatrix} x_n - x_L \\ y_n - y_L \\ z_n - z_L \end{pmatrix} \tag{8.17}$$

$$\begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} = \frac{1}{\lambda_{Rn}} \mathbf{Q}_R^T \begin{pmatrix} x_n - x_R \\ y_n - y_R \\ z_n - z_R \end{pmatrix} \tag{8.18}$$

with $\lambda_{Ln} = \frac{s_{Ln}}{f_L}$ and $\lambda_{Rn} = \frac{s_{Rn}}{f_R}$.

We will now see how to solve this system of $2N$ equations using techniques discussed in chapter 7.1.

[1] Note that f_L and f_R are not necessarily identical to the focal length f, as the lens must be moved relative to the image plane to focus on the objects in view.

8.2.2 Standard Solution

Equations (8.17) and (8.18) lead by matrix inversion that is a simple transposition for the orthonormal matrices $\mathbf{Q}_L^T$ and $\mathbf{Q}_R^T$ to

$$\begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = \begin{pmatrix} x_L \\ y_L \\ z_L \end{pmatrix} + \lambda_{Ln}\mathbf{Q}_L \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} = \begin{pmatrix} x_R \\ y_R \\ z_R \end{pmatrix} + \lambda_{Rn}\mathbf{Q}_R \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \tag{8.19}$$

and by subtracting the left part of this equation from the right part to

$$0 = \begin{pmatrix} x_R \\ y_R \\ z_R \end{pmatrix} - \begin{pmatrix} x_L \\ y_L \\ z_L \end{pmatrix} + \lambda_{Rn}\mathbf{Q}_R \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} - \lambda_{Ln}\mathbf{Q}_L \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix}. \tag{8.20}$$

If we now multiply from the left side with the cross product

$$\left[\mathbf{Q}_R \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \mathbf{Q}_L \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right]^T \tag{8.21}$$

we obtain the *coplanarity equation* that has to be fulfilled for all image points:

$$\left[\mathbf{Q}_R \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \mathbf{Q}_L \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right]^T \begin{pmatrix} x_R - x_L \\ y_R - y_L \\ z_R - z_L \end{pmatrix} \stackrel{!}{=} 0, \qquad n = 1, \ldots, N \tag{8.22}$$

To make things simpler, we can, without loss of generality, set $\mathbf{x}_L = 0$ and $\mathbf{Q}_L = I$, the identity matrix. In addition, we take x_R to be known. The system we have to solve now has five unknowns, $y_R, z_R, \omega_R, \phi_R, \kappa_R$, and the following equation for each pair of corresponding points:

$$\left[\mathbf{Q}_R \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right]^T \begin{pmatrix} x_R \\ y_R \\ z_R \end{pmatrix} \stackrel{!}{=} 0, \qquad n = 1, \ldots, N \tag{8.23}$$

To solve this system, we assume an initial approximate solution $y_R^0, z_R^0, \omega_R^0, \phi_R^0, \kappa_R^0$. At each iteration, we linearize the nonlinear equations using first order Taylor approximations and solve the resulting linear system in a least-squares fashion. This is the same method as discussed in section 7.1.2.

To facilitate the notation, we set $R(\omega_R, \phi_R, \kappa_R) = \mathbf{Q}_R$. Note that this is not identical to the right rotational matrix $\mathbf{R}(\omega_R, \phi_R, \kappa_R) = \mathbf{Q}_R^T$. We used the same letter to facilitate the reading of R's derivations already discussed in section 7.2. Using first order Taylor expansion at

iteration t, we now have approximately

$$\begin{aligned}
0 &= \begin{pmatrix} x_R \\ y_R^t + \Delta y \\ z_R^t + \Delta z \end{pmatrix}^T \left[R(\omega_R^t + \Delta\omega, \phi_R^t + \Delta\phi, \kappa_R^t + \Delta\kappa) \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right] \\
&= \begin{pmatrix} x_R \\ y_R^t \\ z_R^t \end{pmatrix}^T \left[R(\omega_R^t, \phi_R^t, \kappa_R^t) \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right] + \\
&\quad \begin{pmatrix} x_R \\ y_R^t \\ z_R^t \end{pmatrix}^T \left[\Delta R(\omega_R^t, \phi_R^t, \kappa_R^t) \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right] + \\
&\quad \begin{pmatrix} 0 \\ \Delta y \\ \Delta z \end{pmatrix}^T \left[R(\omega_R^t, \phi_R^t, \kappa_R^t) \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right]
\end{aligned}$$

with

$$\Delta R(\omega_R^t, \phi_R^t, \kappa_R^t) = \frac{\partial R}{\partial \omega}(\omega_R^t, \phi_R^t, \kappa_R^t)\Delta\omega + \frac{\partial R}{\partial \phi}(\omega_R^t, \phi_R^t, \kappa_R^t)\Delta\phi + \frac{\partial R}{\partial \kappa}(\omega_R^t, \phi_R^t, \kappa_R^t)\Delta\kappa$$

and the derivatives of R defined as in section 7.2.

This system can be rewritten as the linear overconstrained system

$$\begin{pmatrix} a_{11}^t & a_{21}^t & a_{31}^t & a_{51}^t & a_{61}^t \\ \vdots & & & & \\ a_{1N}^t & a_{2N}^t & a_{3N}^t & a_{5N}^t & a_{6N}^t \end{pmatrix} \begin{pmatrix} \Delta\omega \\ \Delta\phi \\ \Delta\kappa \\ \Delta y \\ \Delta z \end{pmatrix} = -x_R \begin{pmatrix} a_{41}^t \\ \vdots \\ a_{4N}^t \end{pmatrix} - y_R \begin{pmatrix} a_{51}^t \\ \vdots \\ a_{5N}^t \end{pmatrix} - z_R \begin{pmatrix} a_{61}^t \\ \vdots \\ a_{6N}^t \end{pmatrix} \tag{8.24}$$

where

$$\begin{aligned}
a_{1n}^t &= \begin{pmatrix} x_R \\ y_R^t \\ z_R^t \end{pmatrix}^T \left[\frac{\partial R}{\partial \omega}(\omega_R^t, \phi_R^t, \kappa_R^t) \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right] \\
a_{2n}^t &= \begin{pmatrix} x_R \\ y_R^t \\ z_R^t \end{pmatrix}^T \left[\frac{\partial R}{\partial \phi}(\omega_R^t, \phi_R^t, \kappa_R^t) \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right] \\
a_{3n}^t &= \begin{pmatrix} x_R \\ y_R^t \\ z_R^t \end{pmatrix}^T \left[\frac{\partial R}{\partial \kappa}(\omega_R^t, \phi_R^t, \kappa_R^t) \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix} \right] \\
\begin{pmatrix} a_{4n}^t \\ a_{5n}^t \\ a_{6n}^t \end{pmatrix} &= R(\omega_R^t, \phi_R^t, \kappa_R^t) \begin{pmatrix} u_{Rn} \\ v_{Rn} \\ f_R \end{pmatrix} \times \begin{pmatrix} u_{Ln} \\ v_{Ln} \\ f_L \end{pmatrix}, \qquad n = 1, \ldots, N
\end{aligned}$$

This system is solved in the least square sense using singular value decomposition. The

adjusted values for the next iteration are then given by

$$\begin{pmatrix} \omega_R^{t+1} \\ \phi_R^{t+1} \\ \kappa_R^{t+1} \\ y_R^{t+1} \\ z_R^{t+1} \end{pmatrix} = \begin{pmatrix} \omega_R^{t} \\ \phi_R^{t} \\ \kappa_R^{t} \\ y_R^{t} \\ z_R^{t} \end{pmatrix} + \begin{pmatrix} \Delta\omega \\ \Delta\phi \\ \Delta\kappa \\ \Delta y \\ \Delta z \end{pmatrix} \tag{8.25}$$

Discussion We have now seen a simple method to solve the relative orientation problem. However, it is by far not optimal. First, the running time is rather high. For every iteration, a singular value decomposition for a $2N \times 6$ matrix has to be computed. Second, it is not very stable. We have to perform several trigonometrical operations at each step to convert the representation of the orientation from the angular form to the orthonormal matrix form. It should be clear that this is a large amplifier for small noise in the input data.

Another problem is that we have to provide an initial approximation that lies within a narrow range of the real values to keep the number of iterations low. This approximation may be provided by movement prediction during continuous tracking, but there will definitely arise problems if the angular velocity gets too high.

A last problem is that the translation along the x–axis is assumed to be known and greater than zero. During real world applications, we have to check by some sophisticated means that this assumption holds true and to change our coordinate systems if not.

However, we implemented a basic version of this algorithm without all the error handling. The results will be discussed in chapter 11.

We will see in the next section how to use the data obtained from the optical flow computation in a less intuitive, but more successful way in order to speed up tracking.

9 Combining Absolute and Relative Tracking

In the previous three chapters, we have discussed almost all steps that are necessary for optical tracking, from fiducial detection to absolute pose estimation and from optical flow to relative pose estimation. All these things have been done several times by several independent research groups, and for all problems there exists a large variety of solutions that work more or less and with stronger or weaker underlying assumptions.

For the reasons mentioned above (mainly availability in libraries), we decided to use the following components:

- The feature detection from the ARToolKit.
- The POSIT algorithm for absolute pose estimation.
- A pyramid implementation of the Lucas Kanade optical flow algorithm for the determination of optical flow for certain feature points.

The main goal of this thesis was to integrate the optical flow computation with ordinary absolute feature tracking. Neumann and You ([49]) use optical flow to verify and make more robust the results of a region tracking algorithm in a closed-loop architecture, and some other groups have used correlation tracking to speed up or verify feature detection. However, up to now relative and absolute pose estimation have not been combined in a general framework architecture that allows simple exchange of the algorithms used.

This chapter deals with the explanation of the strategy we chose to take the best of optical flow and absolute feature tracking in order to build a better six-dimensional optical tracker.

9.1 The Basic Idea: A UML Sequence Diagram

Recall our goal: to use optical flow, fiducial detection, absolute and relative pose estimation simultaneously in order to combine the best of all components. What are the major advantages (✔) and disadvantages (✘) of these components?

Fiducial Detection

- ✔ Gives very exact 2D position of fiducials.
- ✔ Works even on a single frame and does not need a priori knowledge from previous video frames.
- ✘ Needs much computing power. In general, the less obtrusive the fiducials are and the higher the image resolution is, the more processor clock cycles are needed.

Absolute Pose Estimation

- ✔ Gives exact six-dimensional pose.
- ✘ Needs 2D–3D correlation as input data, i.e. has to know the real world position of all detected image features.

Optical Flow

- ✔ In general faster than sophisticated fiducial detection.
- ✔ Running time depends basically only on number of points for which optical flow computation should be performed, not on image resolution.
- ✘ Needs either time to detect "good features to track" or such features as input data.

Relative Pose Estimation

- ✔ Only needs 2D–2D correlations as input data, no knowledge about three-dimensional real world coordinates is needed.
- ✘ Gives pose only relative to the camera's pose at a certain point in time, not relative to the world coordinate system.
- ✘ Gives six-dimensional pose only up to a scaling factor, i.e. we still have to use some real world data such as the 3D distance between two 2D features to obtain the full pose.
- ✘ Susceptible to amplification of small errors if applied subsequently on several small movements.

The basic idea of the combination is now *to use slow fiducial detection for high accuracy and optical flow for high speed*. This may be seen as a sensor fusion approach combining input data from two different trackers, one based on fiducial detection and the other one on optical flow methods. The two trackers run simultaneously on the same image data. Each of these trackers is assumed to have a third of the CPU clock cycles, as the processing of the video data, the communication with other subsystems and the absolute pose estimation needed are assumed to take the remaining third.

The fiducial based tracker takes some time, we assume as an example $4t$ with t being the time between the image frames, to establish correct 2D–3D correlations. The optical flow computation is much faster at approximately $\frac{1}{2}t$. The absolute pose estimator is assumed to need time $\frac{1}{2}t$. After the fiducial detector has established the 2D–3D correlation of the first image frame I_0 (at time $4t$), the resulting 2D points at position P_0 are fed into the optical flow algorithm. The latter now subsequently takes the image frames $I_1, \ldots, I_7$ and computes the positions $P_1, \ldots, P_7$ of the 2D points in these image frames. At time $4t + 7 \cdot \frac{1}{2}t = 7.5t$, the optical flow computation outputs the 2D points position P_7 in the 7th image frame. From our absolute tracker, *we know the corresponding 3D points in the real world.* In consequence, we can feed our absolute pose estimator with a set of valid 2D–3D correspondences and have at time $\frac{1}{2}t + 7.5t = 8t$ the pose corresponding to frame I_7 grabbed at time $7t$ with a delay of only $1t$. This is by far better than the overall computation time of $\frac{4t}{2} + \frac{1}{2}t = 2.5t$ of the fiducial based tracker running alone with two thirds of the CPU clock cycles. Figure 9.1 illustrates the collaboration of the two trackers using a standardized UML sequence diagram (see [23] and [55] for details).

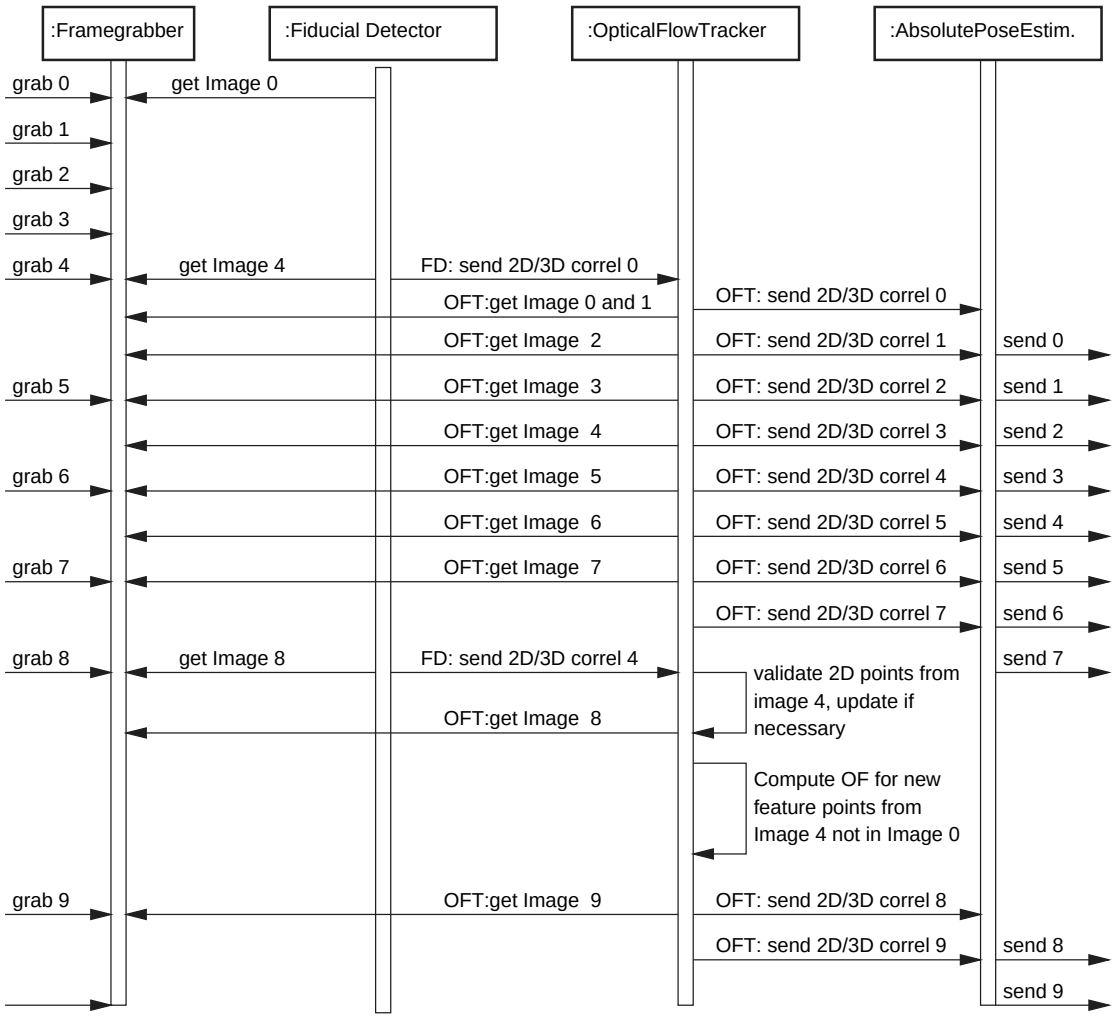


Figure 9.1: Sequence Diagram of Optical Flow and Fiducial Based Tracker. Note that the framegrabber grabs images at constant time intervals.

Care has to be taken to ensure the speed advantage to be constant. Every time the fiducial detector sends new 2D–3D correlation data, we have to verify that the optical flow tracker still tracks valid 2D points. Seen from the sensor fusion point of view, the (sparse) data of the fiducial based tracker overrides the more frequent data of the optical flow tracker. If some of the tracked points are now invalid, the optical flow computation has to be undone for them, and it has to be redone for potential new points. This procedure is not as time consuming as it seems to be. First, if we use the pyramid implementation of the Lucas Kanade optical flow algorithm as discussed in section 8.1.2, the pyramid decomposition is done already. In addition, we have to compute the optical flow only for a few points, reducing the amount of computing time enormously in contrast to the computation for all feature points. In consequence, we assumed this process to take an overall computation time of $\frac{1}{2}t$.

9.2 Implementation

The implementation of this strategy needs an operating system that supports multithreading, i.e. the simultaneous fair execution of various lightweight processes. This assures a minimized overhead for switching from one task to another. In addition, modern multithreaded operating systems such as Linux or Windows 2000 support automated distribution of the threads to the processors on multiprocessor computers.

If this property of the operating system is given, the implementation is straightforward. Each task, the framegrabber, the optical flow tracker, the fiducial detector and the absolute pose estimator is modeled with a thread running concurrently with the other three threads of the overall tracker. Communication between the threads is done using the common address space of threads.

An interesting question is the implementation of the framegrabber unit. On the one hand, it has to be extremely fast, but on the other hand, it should provide many different subsequent image frames simultaneously. This dilemma can be solved using a *cyclical buffer* consisting of a sufficent number of cells.

We last have to mention that the implementation has to be made carefully to exclude inconsistent states. To give an example, if the framegrabber puts its images in a cyclical buffer, no other thread should be able to read a buffer cell during a write operation of the framegrabber. This can be done easily using semaphores and mutexes.

9.3 Extension: Using Relative Pose Estimation

Up to now, we have not yet used any relative pose estimation. Obviously, this should be possible to speed up the process even more. The use of relative pose estimation has the advantage that we do not have to wait for the fiducial detector to output a 2D–3D correspondence.

Instead, we can track whatever features are suited (see [17] for details) along the image frames. For each such correspondence from frame n to frame $n+1$, we can compute a relative homogeneous transformation matrix H_n^R that is fixed up to a scalar factor. Once the fiducial

detector gives us 2D–3D correspondences, we can impose a metric upon the H_i^R, multiply them and quickly obtain the current absolute transformation matrix:

$$H_n = \prod_{i=n}^{1} H_i^R \cdot H_0 \tag{9.1}$$

However, this approach is numerically not very stable. Every matrix multiplication in equation (9.1) amplifies potential errors in the other relative transformation matrices. In addition, imposing a metric upon the H_i^R can be a rather difficult task.

9.4 Advantages

To conclude the discussion of the combination of optical flow with traditional fiducial detectors, we will enumerate several advantages of this new approach.

Reduced Delay. As shown above, the delay of the combined tracker does not depend on the fiducial detector's speed, but rather on the optical flow algorithm's. As fast optical flow algorithms do exist, we can put more effort in fiducial detectors. For example, we could use more natural markers that are not as obtrusive as fiducials that are used up to now.

Increased Framerate. Obviously, if the optical flow tracker needs low computing power, we can process more image frames per second. This leads to smoother tracking data that can be used to make AR applications seem more natural.

Higher Image Resolution. The running time of optical flow algorithms is by far not this dependent on resolution as the running time of fiducial detectors. In consequence, it may be feasible to slow down the fiducial detector down with high resolution images whilst keeping overall delay low with almost resolution-independent optical flow algorithms.

Parallel Processing. Using the multithreading implementation discussed above, we can speed up the processing time linearly with the number of processors in our computer. Several operating systems such as Linux or Microsoft Windows 2000 support this approach. As the multithreading is handled by the operating system, we even do not have to recompile our tracker to gain significant increase in performance on multiprocessor computers.

Enabling Wearable Computing. Up to now, optical tracking was done mainly on high performance workstations. The problem was now that the user of such a system was very handicapped in his movements, as he always had to be connected by wires to the workstation. One alternative implemented by some people was to use ordinary laptop computers that provided sufficient computing power. However, these laptops suffer from cooling problems and low battery running time. With the new efficient approach, it gets possible for the first time to use wearable computers with ultra low energy consumption and limited computing power for the image processing. This will increase the usability range of AR systems significantly.

10 System Design of the Optical Tracker

We have seen that tracking is a complicated job. However, it can be decomposed in several tasks, and we have discussed most of them in the previous chapters. We even have presented a new approach to speed up optical tracking. Most of this material was rather theoretical due to tracking being based heavily on mathematics in general and geometry in particular.

In contrast to the previous sections, this chapter deals with the practical implementation of the DWARF optical tracking service. According to the systematics of Brügge and Dutoit presented in [18], it may be seen as a System Design Document that describes some details of the implementation.

However, we did not use this systematics, as it seems not appropriate to apply a methodology developed for software engineering tasks to an implementation of relatively few algorithms and their—from a software engineer's point of view simple—combination without modifications.

The original methodology decomposes the description of the proposed system architecture as follows:

1. Overview
2. Subsystem Decomposition
3. Hardware/Software Mapping
4. Persistent Data Management
5. Access Control and Security
6. Global Software Control
7. Boundary Conditions
8. Subsystem Services

In contrast to this proposal, this chapter deals almost exclusively with the subsystem decomposition and specific details of the subsystems and their communication among each other, i.e. the "Subsystem Services".

To give software engineers used to Brügge's method a convenient reference, we treat the remaining items at the end of this chapter.

10.1 Design Goals and Consequences Thereof

The design goals for the optical tracker's system design were derived from the nonfunctional and pseudo requirements listed in section 5.4.

The most important goal was to assure high performance. To achieve this goal, we decided to use optimized software libraries wherever possible. In addition, the choice of programming language was C++. Almost all high performance libraries freely available ([30], [33], [32]) stem from Intel Corporation. In consequence, we had no choice for the processor, we had to use Intel Pentium II or Pentium III CPUs.

Portability of the code was another important topic. We decided to modularize our code in a way that made it possible to extract the platform dependent parts such as image acquisition or displaying the video data from the platform independent parts like the implementation of the algorithms operating on raw image data. In addition, we used standard C++ and components of the Standard Template Library (STL) (see [62] for details) for all feasible purposes. In consequence, only the modules for image acquisition and video generation have to be rewritten to port the optical tracker to other operating systems such as Linux or Mac OS, given that comparable libraries are available for these platforms.

The modularized architecture supported another design goal. We strove for an almost plug-and-play architecture that allowed easy change of various tracker components such as the fiducial detector or the optical flow algorithm.

The last design goal was to ensure usability of the optical tracker on wearable computers, with ordinary laptops defined as "wearable" as well. We therefore had to think of methods of image acquisition that on the one hand did not consume too much CPU time and on the other hand worked without usual framegrabbers that are only available for desktop computers. In general, two solutions are thinkable, so-called Webcams connected via the Universal Serial Bus (USB) or digital cameras that use the FireWire (a.k.a. IEEE 1394 or iLink) interface to connect to the computer. USB cameras tend to be low quality and need some CPU time to decompress the video image, FireWire cameras are quite big and hardly available. Independent of the actual camera model, the Microsoft Windows platform is the operating system family that supports almost all cameras. This was the main reason why we chose Windows 2000 as development platform.

10.2 A Multithreaded Approach

We mentioned above that the architecture of the optical tracker is modularized. Recall the discussion in section 9.2 that proposed a multithreaded architecture for the combination of optical flow, fiducial detectors and absolute pose estimators.

We extended this idea and parallelized all tasks that could be parallelized. In consequence, we decomposed the system in various subsystems that were modeled by a single thread for each subcomponent. Communication between these components is done using the common memory, synchronization is done using semaphores and mutexes.

Figure 10.1 shows a general overview of the subsystems involved in the optical tracker and the shared memory areas they use to communicate with each other. In subsequent sections, we will give detailed explanations to every subcomponent.

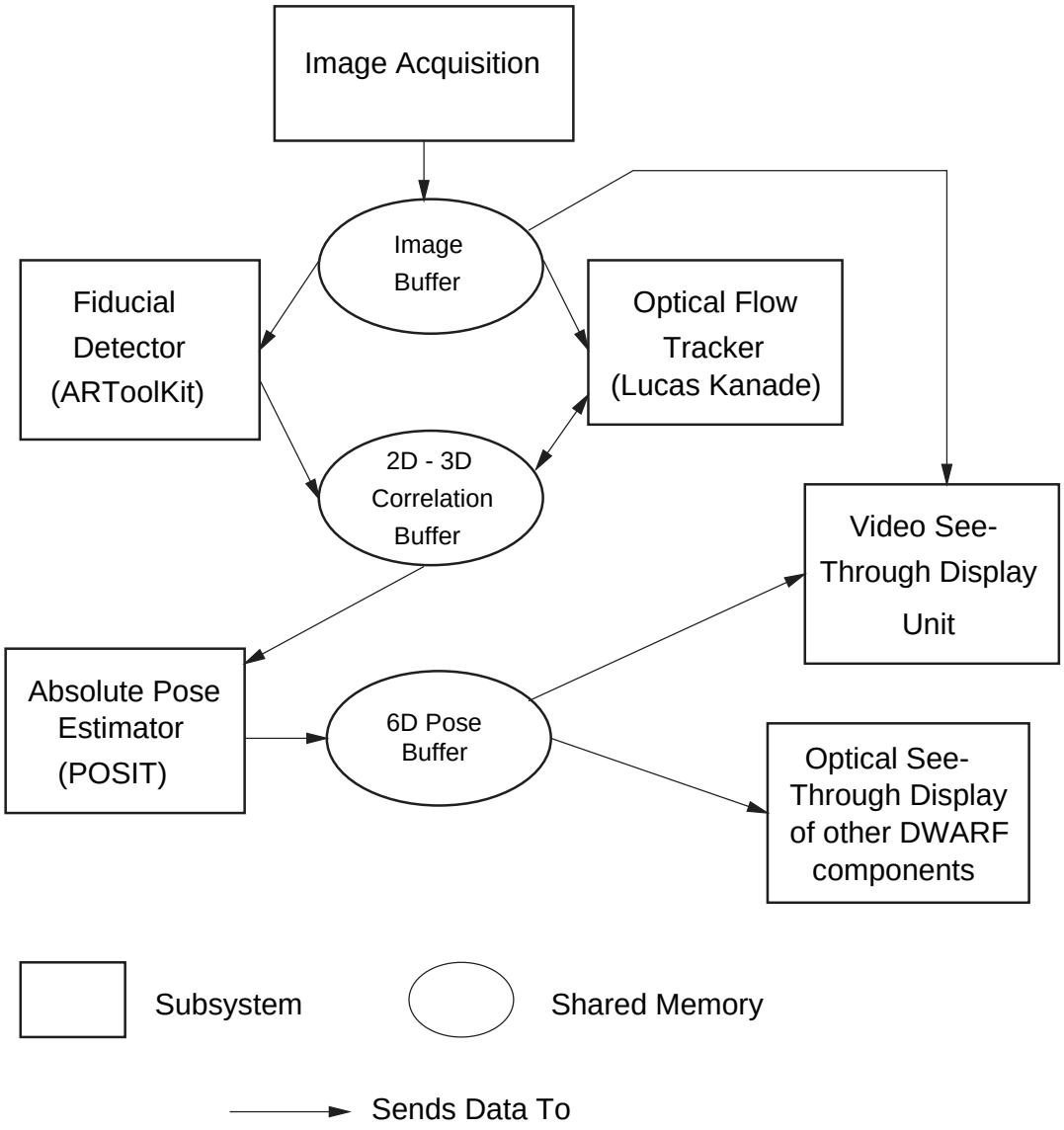


Figure 10.1: Subsystems and Common Memory Areas of Optical Tracker. Note that this is not an UML diagram, as the underlying architecture is not object oriented and separates data from functionality.

A Note on Object Orientation The key concept of object oriented programming is integration of data and functionality. This is the exact opposite of what we did by separating the architecture in threads and shared memory. However, the advantages of object orientation with respect to the intended plug-and-play architecture for new algorithms should lead to further investigations in future work. It could be reasonable to encapsulate algorithms as well as shared memory areas in objects that communicate with one another using exactly specified interfaces. Up to now, if new algorithms should be tested, there has to be put some effort in adopting it to the current architecture.

10.3 Getting the Image Data

The task of this component is simple: get the image of a video camera attached to the computer and transfer the data in 24-bit RGB format to a memory buffer. However, getting the image data requires in-depth knowledge of several operating system dependent *Application Programmer Interfaces (APIs)*. Altogether, we implemented three different methods to acquire video data.

10.3.1 IEEE 1394 Digital Cameras

IEEE 1394 is a new standard for a serial interface working at speeds of 400MBit/s. It is therefore capable of transferring video data at high framerates and resolution without compression. Depending on the manufacturer, there are several different names for the same interface, like FireWire from Apple Computer or iLink from Sony Corporation.

Unfortunately, although the *Digital Video (DV)* standard used in most contemporary camcorders supports transferring data over a IEEE 1394 link, the data gets compressed for this operation. Uncompressing the images takes valuable CPU clock cycles. However, there exists a standard for uncompressed video transfer, but very few cameras support it.

We decided to use the Sony DFW-VL500 digital camera, as it is equipped with remote controllable zoom, focus and iris and does not compress the images. For this camera, a driver library [67] exists for the Microsoft Windows 98 and 2000 operating systems.

Using this library, it became simple to use the Sony camera for image acquisition in the desired way. With this camera all experiments have been conducted.

10.3.2 USB Webcams

Most cheap cameras that are available right now are so-called *Webcams* that get connected to the computer via the *Universal Serial Bus (USB)*. As USB's bandwidth is limited to 12MBit/s, the video data has to be compressed. Uncompressing the data consumes CPU cycles that may otherwise be used for tracking purposes. In consequence, the framerates and/or resolutions achievable with USB cameras are rather low.

However, most webcams have the advantage of being accessible via a standardized interface, Microsoft's *Video for Windows (VfW)*. This architecture stems back from the 16 bit

versions of Windows and is somewhat limited, but nevertheless well suited for our purposes. In addition, it offers the possibility to connect various video cameras simultaneously to the computer.

During startup, the VfW image acquisition module we implemented scans all available cameras and asks the user to choose the right one. Afterwards, it continously updates the image buffers and is not distinguishable in behavior from the IEEE 1394 module.

10.3.3 File Video Data Reader

Both methods we described up to now provide real-time data with high update rates. Especially for reproducable testing of algorithms, it may be desirable to be able to control the speed and sequence of the image frame. For this reason, we implemented a third module that allows framewise control of an image sequence previously recorded to harddisk using a separate tool.

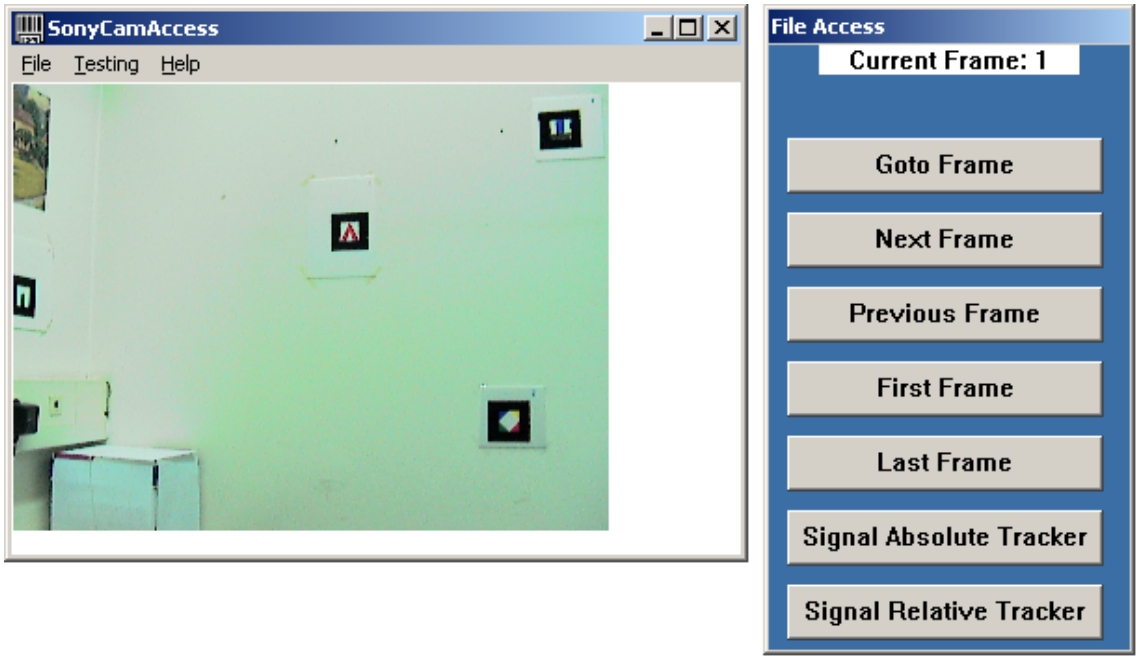


Figure 10.2: Screenshot of the File Video Data Reader

As can be seen in figure 10.2, the module allows comfortable selection of the video frame to be displayed. In addition, it is possible to signal the optical flow tracker and the fiducial detector independently of each other from the advent of a new image frame. We therefore can test the collaboration between the two types of trackers quite efficiently. This signaling was easily made possible using the semaphore mechanisms from the multithreading approach. To signal a tracker, we simply have to release the according semaphore.

10.4 Processing the Image Data

The processing of the image data acquired from one of the modules described above can be divided in three parts: fiducial detection, optical flow determination and absolute pose estimation. In addition, an implementation of the relative pose estimation algorithm described in section 8.2.2 has been provided, although it is not called in the current tracker implementation. We will now briefly describe some implementation issues of each of these components, the algorithmic details can be found in the previous chapters.

Note: Contrary to the intended subsystem decomposition depicted in figure 10.1, the communication between the fiducial detector or optical flow tracker and the absolute pose estimator is not realized using simultaneous running threads and shared memory, but rather using direct function calls. This is due to time constraints during the implementation, but can and should be changed with modest effort in the future.

10.4.1 Fiducial Detection: ARToolKit

As described in chapter 6.3, the Augmented Reality Toolkit [37] has been used to detect the artificial markers mounted in the areas of interest.

The ARToolKit code has been slightly modified to fit our needs, the modifications are marked as such in the source code. For every marker the toolkit detects, a structure `ARMarkerInfo` is put in a shared memory area:

```
typedef struct {
    int     area;
    int     id;
    int     dir;
    double  cf;
    double  pos[2];
    double  line[4][3];
    double  vertex[4][2];
} ARMarkerInfo;
```

These variables now hold the two-dimensional information of the detected markers along with information like a confidence value `cf` and the marker's direction `dir`.

During startup and if the user enters new rooms, the fiducial detection component reads in marker data from the World Model (see section 3.3.3 for a detailed description of this process) and stores it in an array of `arTkMarker` structures:

```
typedef struct {
        int id;
        double center[3];
        double leftUpper[3];
        double rightUpper[3];
        double leftLower[3];
        double rightLower[3];
} arTkMarker;
```

The entries `center`, `leftUpper`, ... hold the three-dimensional coordinates of the marker's center point and corners. Both values, `ARMarkerInfo` and `arTkMarker`, are written to a common memory area and can be used by other components as 2D–3D correlation data.

10.4.2 Optical Flow Determination

We discussed optical flow algorithms in chapter 8.1. As mentioned there, we decided to use a pyramid implementation of the Lucas Kanade optical flow algorithm described in [17].

One of the driving forces for this decision was the availability of this algorithm in the Intel OpenCV library [33], leading to minimized implementation efforts.

However, we had to provide a wrapper around the Intel implementation that allowed the multithreaded implementation of the optical flow estimator. This component uses the 2D–3D correlation data delivered by the ARToolKit to obtain new 2D–3D correlations for subsequent image frames. Care was taken to minimize the computing time by reusing once decomposed pyramid representations of image frames.

Once new correlations are established, the corresponding absolute pose is estimated via a function call and distributed to other DWARF components.

10.4.3 Collaboration Between Optical Flow Tracker and Fiducial Detection

The tracker we developed is thought of as a proof of concept, not a fully functional implementation of the ideas laid out in chapter 9. Due to limited time availability, we decided to implement the collaboration between the optical flow tracker and the fiducial detector in a simpler way than depicted in figure 9.1.

We simply tested the effect of post-calculating 2D–3D correspondences by means of optical flow without worrying about delay in general or delay being constant or minimized. We did not implement the check for still valid 2D points after each update from the fiducial detector as proposed in the last paragraph of section 9.1. Instead, we forced the ARToolKit tracker to update its values at an update rate that was ten times slower than the video camera's frame rate. In consequence, the intermediate nine image frames between the fiducial detector's updates were tracked solely using optical flow.

Communication between the two trackers was done as usual using shared memory and semaphores for access control.

10.4.4 Absolute Pose Estimation

Recall the discussion of absolute pose estimation in chapter 7. We mentioned there that we decided to implement the POSIT algorithm from scratch, as this algorithm promises high speed and sufficient accuracy. In section 7.3.4, we gave an implementation of the algorithm in pseudo code.

The absolute pose estimator is called directly from the optical flow tracker and puts the resulting six-dimensional pose in a shared memory area for further processing.

As we have seen during the discussion of POSIT, it relies heavily on matrix operations like simple multiplications or the Singular Value Decomposition. These operations have been encapsulated in an object oriented library that facilitates the handling significantly.

This library itself relies on procedures from the LAPACK linear algebra package ([7]). The LAPACK package is available for most computing platforms, it should therefore not be too hard a problem to port the code to other operating systems. We used a particular implementation of LAPACK, the *Math Kernel Library (MKL)* [30] from Intel Corporation. The MKL assures highly optimized execution of the code on Intel-compatible processors by using special registers such as the MMX (MultiMedia Extension) register set. In addition, future versions will support new processor architectures such as the Intel Pentium 4 processor, such that our POSIT implementation will use all advantages of these processors by simple recompilation.

10.5 Displaying the Results

Although the DWARF system has its own user interface engine [56] to display the results of the tracking subsystem, it is preferable to have a separate display component that allows testing of the tracker without the whole DWARF system running. In addition, such a component offers an easy possibility to perform video see-through augmented reality by using shared memory to access the camera image.

We implemented this component as a separate process that has to run on the same computer as the tracker. The two processes communicate via shared memory, in Microsoft Windows this is called a *FileMap*. Obviously, the interprocess communication is highly platform dependent. To ensure a maximum portability besides this, we decided to use *OpenGL* [71] as the API to draw the images. First, the camera image is taken and drawn as background image using `glDrawPixels`, afterwards the viewpoint of the OpenGL scene is set and some simple objects like cylinders are drawn at well-known locations to create a testing environment.

As mentioned above, the displaying task is usually performed by the DWARF user interface engine. This engine internally uses the *Virtual Reality Markup Language (VRML)* (see [54] for an introduction) to display correctly aligned virtual objects.

We have defined a VRML scene that allows easy debugging of the tracking algorithm. In this scene, the room the system is used in is shown by two gray walls. Inside this room, the camera's current position is indicated by a blue box as well as its orientation by a blue line of sight. Using this VRML scene depicted in figure 10.3, we can easily debug the dynamic behavior of our system.

10.6 Communication with other DWARF Systems

The optical tracker is quite useless if it is not combined with other components of the DWARF system. This is done using the DWARF event bus architecture described in detail in [46]. The DWARF communication is based heavily on the *Common Object Request Broker Architecture (CORBA)*. To use CORBA, a so-called *Object Request Broker (ORB)* has to be used.

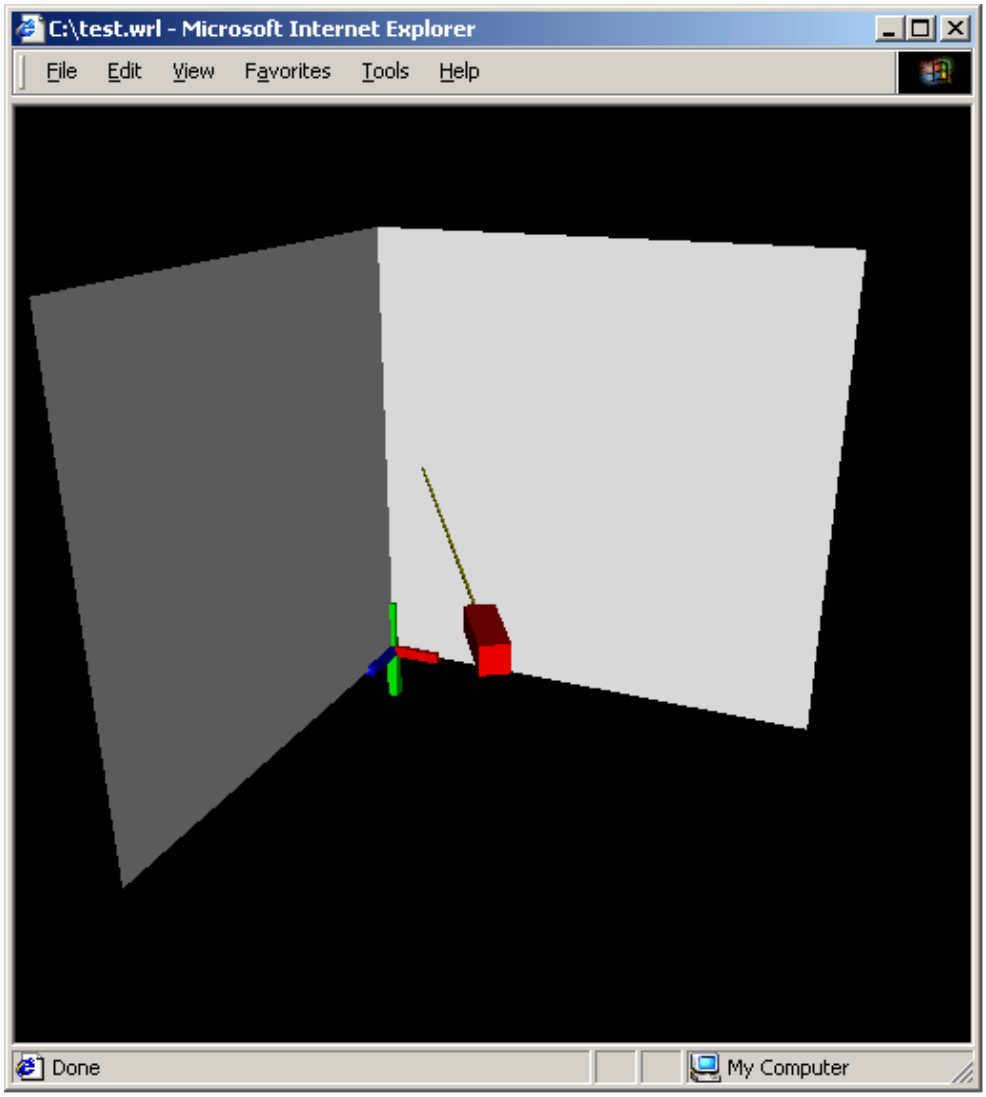


Figure 10.3: VRML scene for Debugging the Optical Tracker

Several commercial and some freely available ORBs exist. We used *omniORB* [45] from AT&T Laboratories.

All communication using CORBA is encapsulated in a single object of the class `DWARF::OpticalTrackerCORBAObject`. This encapsulation ensures that all necessary initialization tasks such as registration with the DWARF system or setting up the CORBA connections are done during the construction of the CORBA object.

Basically, there are only two ways of communication between the optical tracker and the rest of the DWARF system:

Sending Tracking Data Every time the optical tracker has estimated the camera's six-dimensional pose, it encapsulates this pose with some other information such as the time for which the pose is valid or the delay that occurred during the processing of the image frame and sends it to the DWARF event bus.

The encapsulation is done using a CORBA structured event of type `Any` that contains a single object of type `PositionEvent`. The detailed signature of this type is shown in appendix E. Note that the six-dimensional pose is given in VRML coordinates (i.e. Rodriguez coordinates) as well as a homogeneous matrix.

Obtaining Information About the Current Environment To get up to date information about the currently available features to track, the optical tracker registers for so-called `RoomChangedEvents` during startup. Every time the other components of the DWARF tracking subsystem realize that the user has entered a new room, such an event is sent. After receiving it, the optical tracker checks for and loads the marker data of the new environment as described in section 10.4.1.

10.7 Remaining System Design Topics

At the beginning of this chapter we promised to give a full description of the proposed system according to the methodology described in [18]. We will now briefly treat the topics still missing.

Hardware/Software Mapping. We stated during the requirements elicitation that the whole system should be able to run on a single wearable computer. We think that a Sony Vaio PictureBook [60] is a good approximation of a wearable computer without sacrificing too much computing power for small size. The PictureBook is an ordinary Intel-based PC with a 400MHz Pentium II processor and 128MB of RAM. In addition, it has a built-in IEEE 1394 interface that can be used to access our camera and a PCMCIA slot that can be filled with a network card for communication with the other DWARF components running on other computers.

It should be clear that it is not feasible to split the tracking process to two computers. Every time consuming subcomponent needs direct access to the image data. Transferring this data to other computers is too time consuming.

Persistent Data Management. The optical tracker does not have much persistent data. Basically, it only has to store data about the fiducials to be tracked and the corresponding three-dimensional positions. It is, however, not a good idea to hardcode this data into the tracker or even to store it in the local filesystem.

An obvious solution is storing all this persistent data in the DWARF World Model service. Figure 3.6 shows what has to be done to obtain it during runtime.

In addition, calibration data for the camera, i.e. its focal length and principal point, have to be stored. Currently, this data is hardcoded, but it may be transferred to the World Model in future versions.

Access Control and Security. Augmented Reality and especially optical tracking is a field of new research, and up to now no security relevant applications have been developed. In consequence, security issues were not concerned during the development of the optical tracker.

If security has to be implemented in the future, it should be based on a global DWARF security model and assure that pose data is only sent to trusted receivers.

Global Software Control. The optical tracker is a widely self-contained program. As such, it has its own startup procedure that tries to detect other DWARF components. For testing purposes, this detection can be switched off. Once started, the tracker continuously updates the camera's six-dimensional pose whenever fiducials are detected in the video image. There is no possibility to remote control this process. As mentioned above, synchronization between the tracker's subsystems is done using several mutexes and semaphores.

Boundary Conditions. During startup, the tracker registers at the DWARF system and initializes the camera with values the system is calibrated for. If the tracker is shut down, it first disconnects with the rest of the DWARF system (this is implemented in the destructor of the `DWARF::OpticalTrackerCORBAObject` object) and afterwards shuts down the camera.

11 Conclusion

If we recall figure 1.2, we realize that we now have explained Augmented Reality in general, in particular the DWARF system and in great detail the DWARF World Model and Optical Tracker services.

We mentioned above that most of the discussed problems and algorithms have been implemented for this thesis. During testing, we made some interesting experiences that may help the reader get further insight in the difficulties of optical tracking.

These difficulties are by far not solved. We will conclude this thesis by a brief discussion of some important topics that have to be solved in order to make optical tracking as successful as it potentially can be.

11.1 Testing

Brügge and Dutoit state in [18, p. 327]:

> *From a modeling point of view, testing is the attempt of falsification of the system with respect to the system models. The goal of testing is to design tests that exercise defects in the system and to reveal problems.*

In this section we will try to explain our approach to the falsification of the World Model and the optical tracker.

World Model The World Model service was tested extensively during the implementation of the demo scenario explained in section 2.5.1. Almost all other DWARF components used the World Model to store persistent data. During the demo, a single instance of the World Model was running and fed once with a XML file to be parsed into the World Model's database.

Optical Tracker This component was tested using a demo setup consisting of six ARToolKit markers mounted on two perpendicular walls. Using this configuration, it was possible to test the recognition performance of the ARToolKit code and the performance of the POSIT algorithm when running with real time video input.

For every component, we isolated the memory consumption and CPU load in percent. The test system was an IBM compatible PC with an Intel Pentium III processor running at 500MHz and 256MB of RAM. The size of the video images was set to 320×240 pixels.

Special tests were made to detect the behavior of the overall tracker for fast camera movements.

In addition, the file video data reader explained in section 10.3.3 was used to analyze the dynamic behavior of the optical flow computation.

All these tests were analyzed using four output facilities. First was a direct output of the homogeneous matrices computed at each video frame. However, this was not really feasible for real time data, as small deviations in the rotational components can not be seen from the matrices' values. To analyze this data in a first step, the VRML scene depicted in figure 10.3 was used to check if the rough directions of the camera's optical axis were computed correctly. For precise testing we finally used either the OpenGL based video output device described in section 10.5 or VRML scenes describing virtual objects that were projected using a see-through Head Mounted Display.

Finally, we built a second demo setup consisting of ten ARToolKit markers mounted on a wall, a table standing in front of the wall and a printer being on top of the table. However, for this setup we conducted only ARToolKit recognition and some POSIT performance tests.

11.2 Results

This section discusses informally some results from the extensive testing we performed with the World Model and the optical tracker. Due to the limited time frame of this thesis, it was not possible to quantify errors or performance characteristics most of the times. We discuss some problems and hypothesize about their causes and probable solutions. An analysis and implementation of the proposed solutions will be future work.

11.2.1 World Model Database

The World Model completely fulfilled its requirements. Its performance was due to the use of standard efficient algorithms from the C++ standard template library sufficiently good. The World Model's response time was dominated by the network latency, its memory consumption was in the area of a few megabytes and the consumption of processor time was neglectable.

11.2.2 IEEE 1394 as Camera Interface

The decision to use IEEE 1394 as link type to connect the camera was completely justified after the first tests. Using the library [67], it was possible to grab the image frames at rates of 30fps with almost no CPU load. The latter was in the area of 3% including all operation system tasks. Tests conducted with images of size 640×480 pixels yielded a CPU load of 10%.

11.2.3 ARToolKit as Fiducial Detector

The ARToolKit's fiducial detector has been designed for applications where the fiducials' sizes in the video image are quite big. For our application, this assumption did not hold. In consequence, we found several problems that will now be explained briefly.

Dependency on Constant Lighting Although the ARToolKit fiducial detector is using only grayscale values and normalizes image regions being possible fiducials, it depends heavily on lighting conditions being constant. The fiducial's patterns that are given to the detector must be recorded using similar conditions as will be during runtime. Otherwise, detection confidence falls significantly.

Limited Number of Distinct Fiducials The fiducials have to be designed carefully to be detected unambiguously. If the fiducial's sizes in the video image get small such that the interior of the fiducials is in the area of 5×5 pixels, the differences between fiducials have to be rather big to get good detection accuracy. Obviously, with 5×5 pixels it is not possible to design a large number of distinct fiducials with many differences. This property of the ARToolKit code limits its usability in large rooms with many markers.

Dependency of Running Time on Number of Fiducials The ARToolKit code scans the whole input image for potential fiducials and tries to match the detected areas with the patterns internally stored as recognizable fiducials. The implementation of this procedures has complexity linearly increasing with the number of fiducials in the image. If there are only few fiducials in the image (less than four), the fiducial detector takes approximately 25% of the CPU clock cycles at a 30fps update rate. However, if there are nine or ten fiducials simultaneously in the image frame, the CPU load rises to 100% and the framerate drops to $15 - 20$fps.

Limited Accuracy Although the ARToolKit code internally performs subpixel accuracy computations, its resolution was limited to approximately one pixel. To make things worse, we observed significant jitter unless the fiducials filled the whole video image. This effect was especially disturbing if the fiducial's sizes in the video image were rather small. Due to the pose estimation relying on the information about the fiducials' sizes, the results of this estimation were heavily influenced by minor changes in the ARToolKit's output.

11.2.4 Lucas Kanade Optical Flow Algorithm

We had not much time for an in-depth survey of optical flow algorithms. Instead, we chose Intel's pyramidal implementation of the Lucas Kanade feature tracker for the following reasons:

- An optimized implementation exists for Microsoft Windows and Linux [33].
- It supports the computation of the optical flow for only a few image points. Other optical flow algorithms compute only the optical flow for all image points.

- Robustness across a large range of head motions due to the pyramidal search space.
- The existing implementation's speed has been proven to be sufficient for our purposes.

The Intel CVLibrary implementation of the algorithm (see [17]) worked well with subpixel accuracy. The CPU load was in the area of 33% whilst running at 30fps. The tracking of feature points was done as it was supposed to, however, care has to be taken that the feature points fed into the algorithm are clearly detectable (i.e. corners of fiducials that are clearly visible as such). Otherwise, the results of the optical flow algorithm get unpredictable and lead to completely useless data from the following pose estimation.

11.2.5 POSIT for Absolute Pose Estimation

Performance The main reason of the decision to try the POSIT algorithm for pose estimation was speed. This requirement was fulfilled completely. Running at 30fps, the pose estimation took only 25% of the CPU time. If this result is compared to other methods such as Tsai's algorithm [64], it equals to a factor of five to ten in performance. The running time of the POSIT algorithm was not affected too much by the number of 2D–3D correspondences fed into it, as long as this number stayed below 30.

Choice of Orthographic Projection Plane However, several problems occurred. First, the POSIT algorithm relies on the choice of a point M_0 that defines the plane of orthographic projection. If this point changes during runtime, the algorithm's results tend to jump to different values corresponding to different local optima of the nonlinear parameter estimation space. It may even occur that M_0 is chosen in a way that makes correct pose estimation impossible. Up to now, we did not conduct any tests about strategies for choosing M_0, but they might lead to a method that yields better results. In general, if the chosen M_0 is located at the intersection of the line of sight with the plane of orthographic projection, we should expect to get good results.

Two Possible Solutions in Coplanar Case As mentioned in section 7.3.3, using POSIT's approach of pose estimation for a set of coplanar input points, two solutions are possible. Although DeMenthon and Davis discuss strategies to derive a correct pose estimation, POSIT does still not guarantee to compute the correct orientation. We realized that this problem is especially severe if the distance of the plane of the collinear points to the image plane is large compared to the distance between the points. The orientation vector tended to jump between the two values and sometimes sticked with the wrong value until the input data was not collinear any more.

Consequences The POSIT algorithm is extraordinarily fast but yields far from perfect results. Many things can be done to extend the algorithm's performance, such as clever strategies to choose the orthographic projection plane and to select the correct orientation in the coplanar case. Nevertheless, the computational expense to yield correct output data is in our opinion too high to justify the use of POSIT any more. In consequence, we think the

POSIT algorithm should be replaced by more robust and exact but slower algorithms for pose estimation.

If accuracy is not a great issue, however, POSIT may be an excellent alternative to common approaches for pose estimation. The hardware requirements are so low that it is feasible to use POSIT on low power wearable computing platforms.

11.2.6 Hybrid Pose Estimation by an Optical Flow Tracker and a Fiducial Detector

Whenever we tried the sensor fusion between the optical flow tracker and the ARToolKit fiducial detector, we could observe an amplification of errors from both types of tracking. If the ARToolKit's output was not optimal due to reasons discussed above, the optical flow computation yielded results that were hardly usable to estimate a correct pose. For exact input data, the optical flow tracking was remarkably stable.

However, the running time convinced us of the promising concept. If both trackers were running simultaneously, the CPU load was in the area of 80% at 30fps. This result shows that much more time can be spent for fiducial detection, leading either to higher accuracy or less obtrusive fiducials. We will elaborate more on this when discussing possibilities of future work.

11.2.7 Standard Solution of Relative Orientation Problem

To put it short, the implementation of the standard solution to solve the relative orientation problem put by the optical flow computation was a classical example of a mathematical theory that is utterly useless in practice.

Some reasons are obvious. The relative orientation problem can only be solved up to a scaling factor. Therefore, the standard solution assumed the translation along the x-axis to be constant. Unfortunately, quite a few movements of our camera did not involve a movement along the x-axis, leading to results that were hardly predictable.

A second disadvantage was the low speed of the algorithm. As discussed in section 8.2.2, several Singular Value Decompositions were necessary to solve the problem, leading to a computational time five to ten times as high as for the POSIT absolute pose estimation.

However, there are many other algorithms to solve this problem (see [27] for details). It may be worth it to try some of these for optical tracking.

11.2.8 Multithreaded Architecture

The multithreaded architecture of the optical tracker allowed the use of object oriented concepts although the algorithms used were implemented in and thought to be used by procedural languages.

This led to various advantages of object orientation without sacrificing performance. All components of the optical tracker can be replaced by new components with small effort. This

was tested using a demo setup that did not need six-dimensional pose data but rather a two-dimensional position of a fiducial in the image plane. The new "pose estimation" component was put in the place of the POSIT algorithm and worked well after recompilation. The other components of the system had not to be modified for this operation.

The multithreaded architecture allows the use of the developed optical tracker as a framework for testing other algorithms for fiducial detection, pose estimation, optical flow and a combination thereof.

11.2.9 Microsoft Windows as Tracking Platform

We described in section 10.1 the design rationale for the choice of Microsoft Windows 2000 as development platform. The main reason was the availability of cameras and libraries for various algorithms.

Although development in general was not disturbed by more annoyances than on other platforms, some deficiencies in the C standard math library provided with Microsoft Visual C++ led to abnormal program terminations that should never have occurred. In detail, some sequences of trigonometrical operations that were mathematically correct resulted in values classified as "Not a Number". As a consequence, such sequences had to be programmed extremely carefully with exact error handling.

We deployed the optical tracker on a Windows 98 system for our demo application. With this operating system, some problems with the multithreading occurred that could be solved by switching off the debug output on a console window. However, due to its instability, we can not recommend the use of Windows 98 as the base platform for optical tracking.

11.3 Future Work

For this thesis, we have implemented a working six-dimensional pose estimator that involved most tasks that can be done for this purpose. As discussed above, the results were promising yet all but excellent. We will conclude this thesis by a brief outlook to future work on optical tracking. Hopefully, the tracker developed can serve as a framework to test some of the following ideas.

11.3.1 Robust Implementation of POSIT

We have seen above that the POSIT algorithm for pose estimation has to deal with several problems. However, its speed is so high that maybe several computations done for more robust results do not hurt the huge advantage it has in speed against conventional pose estimation algorithms.

It would be an interesting project to study how the choice of the feature point M_0 determining the plane of orthographic projection influences POSIT's accuracy. In addition, the use of information obtained in frames processed before the momentary one should be considered. With this approach, it could be possible to solve the "two possible solutions in the coplanar case" problem.

A last proposal for the future development of POSIT deals with wearable computing. Does POSIT's high performance allow simple Augmented Reality applications to run on wearable computers or is its accuracy even too low for this field of use?

11.3.2 Enhanced Integration of Optical Flow and Fiducial Detection

We mentioned in section 11.2.6 that we implemented the collaboration between the optical flow tracker and the fiducial detector in a much simpler way than proposed in chapter 9. This was fine as a proof of concept using a fast fiducial detector such as the ARToolKit's one, but hinders the tracker from gaining all advantages that the concept promises.

It will be an interesting task to think about a detailed concept that allows the update of some points tracked by the optical flow tracker after each delayed output of the fiducial detector. If such a strategy could be implemented efficiently, we could test our concept in detail. After successful testing, it would be possible to use significantly slower fiducial detectors than now.

An additional topic for the optical flow computation would be trying out higher image resolutions. Conceptually, the optical flow algorithm's running time should rise only sublinearly. This may be used to increase the tracker's accuracy and therefore the usability of AR systems in general.

11.3.3 Markerless Tracking

Once the problems of integrating optical flow and fiducial detectors are solved, we could try to start implementing slow fiducial detectors that work completely unobtrusive by using natural features. These markerless trackers are obviously the final goal of the tracker development.

Up to now, several research groups have proposed methods to get rid of artificial markers. Simon et. al. [59] propose to use planar structures in the scene, as such structures occur in most man-made environments. Jebara [34] uses the expectation maximization algorithm [21] to robustly track homogeneously colored areas. Schiele and Crowley [57] use multidimensional histograms to reliably identify objects that may serve as natural markers.

Especially color tracking may be a valuable field of future research. However, strategies for obtaining color constancy have to be developed before being able to track reliably based on color properties.

11.3.4 Fusion of Tracking Data

In recent years, *intelligent environments* were proposed to allow seamless integration of computing devices with the user's reality. Once buildings do have high infrastructure, it should be possible to attach a variety of tracking sensors that collaborate to enhance the quality of their data. Klinker et. al. [41] give an in-depth discussion of the problems arising, Auer and Pinz [11] report about hybrid systems using magnetic and optical tracking simultaneously, Hoff [25] discusses the potential benefits of fusing data from head-mounted and fixed sensors.

It would be interesting to study the kind of outside information that can be used by our optical tracker to enhance its speed and/or accuracy. To give an example, it may be useful to signal the tracker whenever a RFID tag (see section 4.2.1.4) is in its vicinity.

11.3.5 Dynamic Creation of Models

Up to now, we always assumed to have a model of the user's environment. What happens if we are in an unknown area? Do we have to give up optical tracking and rely solely on global systems such as GPS?

In our opinion, it should be feasible to combine all the methods mentioned above in order to build a tracker that automatically identifies natural features that are good to track and obtains their position using optical flow computations and information from other trackers. This tracker could then automatically create a three-dimensional model of its environment. In theory, this approach is possible, as has been shown by Zisserman et. al. [72].

Large progress has to be made to speed up the existing methods to realtime performance. Hopefully, our approach of combining relative and optical tracking can give some advantages.

11.3.6 Enhancing the World Model

The implementation of the DWARF World Model we discussed in chapter 3 has been sufficient for our demo application. If the DWARF framework has to be used in larger applications, it would make sense to integrate a usual database system such as *Oracle* or *MySQL* into the World Model service. This may be done with minor changes to the World Model interface and could enhance the scalability by orders of magnitude.

A second enhancement will allow the World Model to operate as a distributed application. Tasks such as synchronization and consistency checks have to be implemented for this.

11.4 Summary

We have developed and tested two essential components of Augmented Reality applications, a database storing location dependent information and an optical tracker based on artificial markers.

These components were created using some freely available software libraries supporting complex algorithms such as optical flow estimation or fiducial detection. The optical tracker served as a proof of concept of a new approach towards more accurate tracking.

We managed to get a prototypical implementation to work and tested this implementation in a demo scenario. The optical tracker we developed is well suited to serve as a research framework for future work in optical tracking.

Appendix

A Used Libraries

One of the design goals was to use as many libraries as possible to keep the development time low. If not mentioned otherwise, the libraries discussed are only available for Microsoft Windows.

We used the following libraries:

ARToolKit. The Augmented Reality Toolkit provides facilities for fiducial detection, pose estimation and displaying of video or optical see-through Augmented Reality applications. It is available for most platforms at
`http://www.hitl.washington.edu/research/shared_space/download`

Intel Math Kernel Library. This library is an implementation of the LAPACK linear algebra package functions highly optimized for Intel Pentium processors. It is available at
`http://developer.intel.com/software/products/mkl/`

Intel Image Processing Library. The IPL contains convenient data types to describe images in whatever format and many algorithms to manipulate them. The algorithms are highly optimized for Intel Pentium processors and use special register sets such as MMX. The IPL is the basis of the OpenCV library. It is available at
`http://developer.intel.com/software/products/perflib/ipl/index.htm`

Intel Open Source Computer Vision Library. The OpenCV library contains a large variety of algorithms from the field of computer vision. Although the library is still in alpha status, most of the algorithms work well and can easily be used in applications. Most of the library is specially optimized for Intel Pentium processors. There exists a Linux version of the OpenCV library, but we have not tested it up to now. It is available at
`http://www.intel.com/research/mrl/research/cvlib/`

omniORB. omniORB is a robust, high-performance ORB developed by AT&T Laboratories in Cambridge. It can easily be used with C++ applications and has been the base of all the CORBA communication described in this thesis. It is available at
`http://www.uk.research.att.com/omniORB/`

Xerces C++ Parser. Xerces is a validating XML parser that provides facilities to read and process XML files for all C++ programs. It is available for almost all existing computing platforms and can be obtained at
`http://xml.apache.org/xerces-c/`

IEEE-1394 Digital Camera Windows Driver. This library allows easy-to-use access to a video camera that transfers uncompressed video data via the IEEE 1394 interface. It

is available at
http://www.cs.cmu.edu/~iwan/1394/

B Installation of the Program

B.1 Installation of Windows Binary

The Windows binary is installed easily. The following steps have to be performed for the optical tracker:

1. Install the IEEE-1394 camera driver.
2. Put the file `SonyCamAccess.exe` and the dynamic link library `xerces-c_1_3D.dll` in an arbitrary directory.
3. Connect the IEEE 1394 video camera to the computer.
4. Start the file `SonyCamAccess.exe`.
5. Enjoy the program!

The World Model is only a single executable file and does not need any previous installation steps.

B.2 Installation of the Source Code

The source code exists as a set of several Microsoft Visual C++ 6.0 project (`.dsp`) files. The workspace `wagnerm\SonyCamAccess\SonyCamAccess.dsw` comprises all necessary project files. To successfully compile the source code, the following steps have to be performed:

1. Install all libraries mentioned in appendix A into the directory `D:\Developing`
2. Set the library and include paths in Visual C++ according to the documentation of the libraries.
3. Execute the batch file `stub\omni\make.bat` to let build omniORB the classes necessary for CORBA communication out of the IDL definitions.
4. Compile the various project files in the following sequence:
   a) `AR files`
   b) `LinAlgLib files`

c) `WorldModel files`
d) `SonyCamAccess files`
e) `VideoCapture files`
f) `OpenGLViewer files`

5. Enjoy the program!

C IDL Definition of the World Model Interface

```
#include <Position.idl>
#include <Property.idl>

module DWARF {

        enum ThingChangeType {
                ThingAdded, ThingDestroyed, ThingMoved,
                PropertyAdded, PropertyDeleted, PropertyChanged
        };

        struct ThingChangedEvent {
                ThingChangeType how;
                ThingID what;
                ThingID oldParent;
                ThingID newParent;
                string propertyName;
        };

        typedef sequence<string> StringSequence;
        typedef sequence<ThingID> ThingIDSequence;

        exception ThingNotFound {};
        exception PropertyNotFound {};

        interface Thing {
                ThingID getID();
                string getName();
                string getPath();

                Thing getParent();
                ThingID getParentID();
                void changeParentToID(in ThingID newParent)
                     raises(ThingNotFound);

                StringSequence getChildrenNames();
                StringSequence getChildrenPaths();
                ThingIDSequence getChildrenIDs();
                void addChild(in string name);
```

```
                Position getAbsolutePosition();
                Position getPositionToParent();
                Position getPositionToID(in ThingID id)
                         raises(ThingNotFound);
                Position getPositionToPath(in string path)
                         raises(ThingNotFound);
                void setPosition(in Position pos);

                PropertySeq getProperties();
                boolean hasProperty(in string name);
                any getProperty(in string name)
                    raises(PropertyNotFound);
                void setProperty(in string name, in any value);
                void deleteProperty(in string name)
                     raises(PropertyNotFound);

                void destroy();
        };

        interface WorldModel {
                Thing getWorld();
                Thing getThingFromID(in ThingID id)
                      raises(ThingNotFound);
                Thing getThingFromPath(in string path)
                      raises(ThingNotFound);
                ThingID getIDFromPath(in string path)
                        raises(ThingNotFound);
                any getPositionEventFromId(in ThingID id)
                    raises(ThingNotFound);
                void setMinHopCount(in short count);
                void addContents(in string url, in ThingID parent)
                     raises(ThingNotFound);
        };

};
```

D Datatype Definition of XML World Model Description

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
        World Model Modeling Language WMML
        ==================================
        Author:
                Martin Wagner
        Description:
                This dtd describes a part of the DWARF world model.
                All thing objects are assumed to be in the same
                subtree of the World Model. The root Thing object
                of this subtree has to be given outside
                the WMML file.
        Change History:
                - version 1.0: initial revision
-->
<!-- define the backslash
-->
<!ENTITY bs "&#92;">

<!ELEMENT wmml (thing+)>
<!--
        The "thing" element describes a World Model Thing
        object. It always has a name and a pose, given as
        3 rows of a rotational matrix and a translational
        vector, each consisting of 3 consecutive double
        values stored in a string
-->
<!ELEMENT thing (property*, thing*)>
        <!ATTLIST thing
                name    NMTOKEN #REQUIRED
                rot1    NMTOKENS #REQUIRED
                rot2    NMTOKENS #REQUIRED
                rot3    NMTOKENS #REQUIRED
                trans   NMTOKENS #REQUIRED>

<!--
```

```
        The "property" element stores a Property of a Thing
        object. It always has a name and a value. Valid
        types depend on the application, however, the default
        behaviour (if the type is not treated in a special
        way by the application) is to store the value as a
        string.
-->
<!ELEMENT property EMPTY>
        <!ATTLIST property
                type    NMTOKEN #IMPLIED
                name    NMTOKEN #REQUIRED
                value   CDATA #REQUIRED>
```

E IDL Definition of the PositionEvent Type

```
module DWARF {

        typedef double PositionVector[3];

        typedef double OrientationVector[4];

        typedef double HomogenousCoords[16];
        //(row 0,col 0),(row 0, col 1)...
        // like a book

        typedef unsigned long ThingID;
        typedef unsigned long RelativeToThingID;

        struct Time {
                unsigned long seconds;
                unsigned long microseconds;
        };

        struct Position {
                boolean hasRotation;
                boolean hasTranslation;
                boolean hasPose;
                PositionVector translation;
                OrientationVector rotation;
                HomogenousCoords pose;
                double accuracy;
                unsigned long lagUsec;
                Time t;
        };

        struct PositionEvent {
                ThingID thing;
                RelativeToThingID relativeTo;
                short hopCount;
                Position pos;
        };

};
```

F Glossary

API. *see* APPLICATION PROGRAMMER'S INTERFACE

AR. *see* AUGMENTED REALITY

Application Programmer's Interface. "Set of fully specified operations provided by a subsystem" [18]

Augmented Reality. A technique that uses virtual objects to enhance the user's perception of the real world.

Bluetooth. Standard for low range wireless communication.

CORBA. Common Object Request Broker Architecture. CORBA is a specification for a system whose objects are distributed across different platforms. The implementation and location of each object are hidden from the client requesting the service.

Class Diagram. UML class diagrams depict associations, references and inheritance relationships between classes. They also represent the attributes and operations of individual classes.

DTD. Data Type Definition. A DTD describes the alphabet and grammar of an XML language.

Fiducial. An artificial marker used for optical tracking.

FireWire. *see* IEEE 1394

Frame Grabber. A hardware device that gets analog video as input and transfers digitized images into computer memory.

GNU. GNU's Not Unix. An initiative to reimplement the most common UNIX tools on an open source basis.

GPL. GNU Public Licence. A software license developed by the GNU initiative that allows redistribution of software in source code but restricts the rights of the licensee to commercialize software derived from the original code.

GPS. *see* GLOBAL POSITIONING SYSTEM

Global Positioning System. A wide range tracking system based on timing signals from satellites.

HMD. *see* HEAD MOUNTED DISPLAY

Head Mounted Display. A display device similar to glasses. Its user either sees only the display or the display information projected optically onto the real world (See-Through Head Mounted Display)

IDL. *see* INTERFACE DEFINITION LANGUAGE

IEEE 1394. Specification of a high speed (400 MBit/s) serial interface used to connect storage facilities or video cameras to computers. Branded *FireWire* by Apple and *iLink* by Sony.

iLink. *see* IEEE 1394

Interface Definition Language. A language that allows the programming language independent specification of software interfaces. It is used to describe the interfaces of CORBA objects.

LAPACK. Linear Algebra Packet. A set of useful routines for the solution of linear algebra problems. Available for most computing platforms.

MMX. Multimedia Extension. A set of assembler commands specially suited for operations on multimedia content of most current processors compatible to the Intel x86 architecture.

MVC. Model–View–Controller. A common principle of software decomposition.

Middleware. A piece of software that is used to combine various subsystems of a software system.

ORB. Object Request Broker. An ORB is at the heart of a CORBA system. Every process communicating via CORBA must have its own ORB running.

OpenGL. An API for simple programming of threedimensional computer graphics available on most operating systems.

POSIT. Position from Orthography and Scaling with Iterations. An efficient algorithm to estimate the position of a camera.

RAD. Requirements Analysis Document. A document describing the requirements of a software project and the way they were derived.

RFID Tags. Passive tags holding an unique ID that can be read using induction by an active RFID reader located nearby.

SAX. Simple API for XML. An standardized easy to use API for parsing XML documents.

SDD. System Design Document. A document describing the general design of a software system. It serves as a basis for the implementation.

STHMD. *see* HEAD MOUNTED DISPLAY

STL. Standard Template Library. A set of template classes provided with ANSI C++ implementations.

SVD. Singular Value Decomposition. A linear algebra method to decompose arbitrary matrices in a three matrix representation that allows convenient soultions of least square problems.

Sequence Diagram. "UML notation representing the behavior of the system as a series of interactions among a group of objects. Each object is depicted as a column in the diagram. Each interaction is depicted as an arrow between colums." [18]

Tracker. A device determining the position and orientation of a tracked object.

UML. *see* UNIFIED MODELING LANGUAGE

USB. *see* UNIVERSAL SERIAL BUS

Unified Modeling Language. "A standard set of notations for representing models." [18]. See [55] for details.

Universal Serial Bus. A convenient medium speed (12 MBit/s) serial interface available at most modern computers.

Use Case Diagram. UML notation to represent the functionality of a system.

VR. *see* VIRTUAL REALITY

VRML. Virtual Reality Markup Language. Allows the convenient description of virtual objects and scenes for AR and VR applications.

VfW. *see* VIDEO FOR WINDOWS

Video for Windows. An API to access video cameras from the Microsoft Windows operating system family.

Virtual Reality. A computer based technology that allows its user to act in purely virtual environments.

WaveLAN. A midrange wireless communication standard used to replace common Ethernet connections.

XML. Extensible Markup Language. XML is a simple, standard way to delimit text data with so-called tags. It can be used to specify other languages, their alphabets and grammars.

Bibliography

[1] *DWARF Project Homepage*. http://www.augmentedreality.de.

[2] *GNU General Public License*. http://www.gnu.org/copyleft/gpl.html.

[3] *VRML97 specification*. http://www.vrml.org/technicalinfo/specification/vrml97/index.htm.

[4] *Proceedings of the IEEE International Workshop on Augmented Reality – IWAR 1999*, San Francisco, 1999.

[5] *Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000*, Munich, 2000.

[6] *Bluetooth Special Interest Group Homepage*. http://www.bluetooth.com, January 2001.

[7] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. D. CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, and D. SORENSEN, *LAPACK Users Guide*, Society for Industrial and Applied Mathematics (SIAM), 1994.

[8] H. AOKI, B. SCHIELE, and A. PENTLAND, *Realtime Personal Positioning System for Wearable Computers*, Tech. Rep. 520, MIT Media Laboratory, Cambridge, MA, 1998.

[9] H. AOKI, B. SCHIELE, and A. PENTLAND, *Recognizing Personal Location from Video*, in Proceedings of the Perceptual User Interfaces Workshop (PUI '98), 1998.

[10] ASCENSION TECHNOLOGY CORPORATION, *Homepage*. Internet, December 2000. http://www.ascension-tech.com.

[11] T. AUER and A. PINZ, *Building a Hybrid Tracking System: Integration of Optical and Magnetic Tracking*, in Proceedings of the IEEE International Workshop on Augmented Reality – IWAR 1999, San Francisco, 1999, pp. 13 – 22.

[12] R. T. AZUMA, *A Survey of Augmented Reality*, Presence, 6 (1997), pp. 355–385.

[13] M. BAUER, *Distributed Wearable Augmented Reality Framework (DWARF) Design and Implementation of a Module for the Dynamic Combination of Different Position Tracker*, Master's thesis, Technische Universität München, 2001.

[14] M. BAUER, A. MACWILLIAMS, F. MICHAHELLES, C. SANDOR, S. RISS, M. WAGNER, B. ZAUN, C. VILSMEIER, T. REICHER, B. BRÜGGE, and G. KLINKER, *DWARF: Requirements Analysis Document*. Unpublished, 2000.

[15] M. BAUER, A. MACWILLIAMS, F. MICHAHELLES, C. SANDOR, S. RISS, M. WAGNER, B. ZAUN, C. VILSMEIER, T. REICHER, B. BRÜGGE, and G. KLINKER, *DWARF: System Design Document*. Unpublished, 2000.

[16] R. BEHRINGER, C. TAM, J. MCGEE, S. SUNDARESWARAN, and M. VASSILIOU, *A Wearable Augmented Reality Testbed for Navigation and Control, Built Solely with Commercial Off-The-Shelf (COTS) Hardware*, in Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000, Munich, 2000, pp. 12 – 19.

[17] J.-Y. BOUGUET, *Pyramidical Implementation of the Lucas Kanade Feature Tracker. Description of the algorithm*. Part of the Intel Computer Vision Library Documentation, 2000.

[18] B. BRÜGGE and A. H. DUTOIT, *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[19] Y. CHO, J. LEE, and U. NEUMANN, *A Multi–Ring Color Fiducial System and a Rule–Based Detection Method for Scalable Fiducial–Tracking Augmented Reality*, 1st International Workshop on Augmented Reality (IWAR), (1998).

[20] D. F. DEMENTHON and L. S. DAVIS, *Model-Based Object Pose in 25 Lines of Code*, in Computer Vision–EECV 92, Lecture Notes in Computer Science 588, G. Sandini, ed., Springer Verlag, 1992, pp. 335–343.

[21] A. DEMPSTER, N. LAIRD, and D. RUBIN, *Maximum likelihood from incomplete data via the EM algorithm*, Journal of the Royal Statistical Society, B39 (1977).

[22] L. DOWNES and A. BERG, *Computing Rotations in 3D*. `http://www.cs.berkeley.edu/~ug/slide/pipeline/assignments/as5/rotation.html`.

[23] M. FOWLER and K. SCOTT, *UML Distilled*, Addison Wesley, Reading, MA, 2nd ed., 2000.

[24] R. M. HARALICK and L. G. SHAPIRO, *Computer and Robot Vision*, vol. II, Addison Wesley, Reading, MA, 1993.

[25] W. A. HOFF, *Fusion of Data from Head–Mounted and Fixed Sensors*, 1st International Workshop on Augmented Reality (IWAR), (1998).

[26] R. L. HOLLOWAY, *Registration Error Analysis for Augmented Reality*, Presence, 6 (1997), pp. 413–432.

[27] B. K. HORN, *Relative Orientation Revisited*, Journal of the Optical Society of America A, 8 (1991), pp. 1630 – 1638.

[28] B. K. HORN, H. M. MILDEN, and S. NEGAHDARIPOUR, *Colsed-Form Solution of Absolute Orientation Using Orthonormal Matrices*.

[29] B. K. P. HORN, *Robot Vision*, The MIT Press, Cambridge, MA, 1986.

[30] INTEL CORPORATION, *Intel Math Kernel Library Reference Manual*, 3.2.1 ed., 1999. Part of the Intel Math Kernel Library Distribution, available at `http://developer.intel.com/software/products/mkl/index.htm`.

[31] INTEL CORPORATION, *Computer Vision Library Reference Manual*, 2000.

[32] INTEL CORPORATION, *Intel Image Processing Library*. `http://developer.intel.com/software/products/perflib/ipl/`, January 2001. Available for Linux and the Microsoft Windows Platform.

[33] INTEL CORPORATION, *Open Source Computer Vision Library*. `http://www.intel.com/research/mrl/research/cvlib/`, January 2001. Available for Linux and the Microsoft Windows Platform.

[34] T. JEBARA, *Action-Reaction Learning: Analysis and Synthesis of Human Behavior*, Master's thesis, MIT Media Laboratory, 1998.

[35] T. JEBARA, B. SCHIELE, N. OLIVER, and A. PENTLAND, *DyPERS: Dynamic Personal Enhanced Reality System*, Tech. Rep. 463, MIT Media Laboratory, Cambridge, MA, 1997.

[36] H. KATO and M. BILLINGHURST, *Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System*, in Proceedings of the IEEE International Workshop on Augmented Reality – IWAR 1999, San Francisco, 1999, pp. 85–94.

[37] H. KATO, M. BILLINGHURST, R. BLANDING, and R. MAY, *ARToolKit PC version 2.11*, December 1999. available at `http://www.hitl.washington.edu/research/shared_space/download`.

[38] H. KATO, M. BILLINGHURST, I. POUPRYEV, K. IMAMOTO, and K. TACHIBANA, *Virtual Object Manipulation on a Table-Top AR Environment*, in Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000, Munich, 2000, pp. 111 – 119.

[39] B. W. KERNIGHAN and D. M. RITCHIE, *The C Programming Language, Second Edition, ANSI C*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

[40] G. KLINKER, K. AHLERS, D. BREEN, P.-Y. CHEVALIER, C. CRAMPTON, D. GREER, D. KOLLER, A. KRAMER, E. ROSE, M. TUCERYAN, and R. WHITAKER, *Confluence of Computer Vision and Interactive Graphics for Augmented Reality*, Presence, 6 (1997), pp. 433–451.

[41] G. KLINKER, T. REICHER, and B. BRÜGGE, *Distributed User Tracking Concepts for Augmented Reality Applications*, in Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000, Munich, 2000, pp. 37 – 44.

[42] R. L. LAGENDIJK, *The TU-Delft Research Program "Ubiquitous Communications"*, in 21st Symposium on Information Theory in the Benelux, May 2000.

[43] *LART Project Home Page*. `http://www.lart.tudelft.nl/`.

[44] A. LELE, S. NANDY, and D. EPEMA, *Hamony – An Architecture for Providing Quality of Service in Mobile Computing Environments*, Journal of Interconnection Networks, (2000).

[45] S.-S. LO, D. RIDDOCH, and D. GRISBY, *The omniORB version 3.0 User's Guide*, AT&T Laboratories, Cambridge, October 2000. available at `http://www.uk.research.att.com/omniORB/`.

[46] A. MACWILLIAMS, *Using Ad-Hoc Services for Mobile Augmented Reality Systems*, Master's thesis, Technische Universität München, 2001.

[47] F. MICHAHELLES, *Designing an Architecture for Context-Aware Service Selection and Execution*, Master's thesis, Ludwig–Maximilians–Universität München, January 2001.

[48] *MIThril Home Page*. `http://www.media.mit.edu/wearables/mithril/index.html`.

[49] U. NEUMANN and S. YOU, *Integration of Region Tracking and Optical Flow for Image Motion Estimation*, in IEEE International Conference on Image Processing (ICIP'98), October 1998.

[50] F. OBERKAMPF, D. F. DEMENTHON, and L. S. DAVIS, *Iterative Pose Estimation using Coplanar Feature Points*, CVGIP: Image Understanding, 63 (1996). Available at `http://www.cfar.umd.edu/~daniel/daniel_papersfordownload/CoplanarPts.pdf`.

[51] W. H. PRESS et al., *Numerical Recipes in C: the art of scientific computing*, Cambridge University Press, 2nd ed., 1992.

[52] L. RÅDE and B. WESTERGREN, *Springers Mathematische Formeln*, Springer, Berlin; Heidelberg; New York; Barcelona; Budapest; Hongkong; London; Mailand; Paris; Santa Clara; Singapur; Tokio, 1996.

[53] S. RISS, *A XML based Task Flow Description Language for Augmented Reality Applications*, Master's thesis, Technische Universität München, 2000.

[54] B. ROEHL, J. COUCH, C. RHEED-BALLREICH, T. ROHALY, and G. BROWN, *Late Night VRML 2.0*, Ziff Davis Press, 1997.

[55] J. RUMBAUGH, I. JACOBSON, and G. BOOCH, *The Unified Modeling Language Reference Manual*, Addison Wesley, Reading, MA, 1999.

[56] C. SANDOR, *CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces*, Master's thesis, Technische Universität München, 2000.

[57] B. SCHIELE and J. L. CROWLEY, *Probabilistic Object Recognition using Multidimensional Receptive Field Histograms*.

[58] J. SCHMIDT, *Aufnahmegeometrie*, in Erweiterte Realität: Bildbasierte Modellierung und Tragbare Computer, B. Brügge, H. Niemann, and T. Reicher, eds., Technische Universität München, October 1999, pp. 152 – 174.

[59] G. SIMON, A. W. FITZGIBBON, and A. ZISSERMAN, *Markerless Tracking using Planar Structures in the Scene*, in Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000, Munich, 2000, pp. 120 – 128.

[60] SONY CORPORATION, *Sony Vaio PictureBook Homepage*. http://www.ita.sel.sony.com/products/pc/notebook/pcgc1xs.html, January 2001.

[61] D. STRICKER and T. FRÖHLICH, *The Augmented Man*, in Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000, Munich, 2000, pp. 30 – 36.

[62] B. STROUSTRUP, *Die C++ Programmiersprache*, Addison Wesley, 1998.

[63] THE APACHE XML PROJECT, *Xerces C++ Parser*. http://xml.apache.org/xerces-c/, January 2001.

[64] R. TSAI, *A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses*, IEEE J. Robotics and Automation, 3 (1987), pp. 323 – 344.

[65] M. TUCERYAN and N. NAVAB, *Single point active alignment method (SPAAM) for optical see-through HMD calibration in AR*, in Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000, Munich, 2000, pp. 149–158.

[66] *UbiCom Home Page*. http://www.ubicom.tudelft.nl/.

[67] I. ULRICH, *IEEE-1394 Digital Camera Windows Driver DLL*, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 4.0 ed., May 2000. available at http://www.cs.cmu.edu/~iwan/1394/.

[68] H. VAN DIJK, K. LANGENDOEN, and H. SIPS, *ARC: a Bottom-Up Approach to Negotiated QoS*, in 3rd IEEE Workshop on Mobile Computing Systems and Applications, October 2000.

[69] W3C XML WORKING GROUP, *The annotated XML specification*. http://www.xml.com/axml/testaxml.htm, January 2001.

[70] M. WAGNER, *Calibration and Tracking*, in Erweiterte Realität: Bildbasierte Modellierung und Tragbare Computer, B. Brügge, H. Niemann, and T. Reicher, eds., Technische Universität München, October 1999, pp. 132 – 151.

[71] M. WOO, J. NEIDER, T. DAVIS, and D. SHREINER, *OpenGL 1.2 Programming Guide*, Addison Wesley, Reading, MA, 3rd ed., 1999.

[72] A. ZISSERMAN, A. W. FITZGIBBON, and G. CROSS, *VHS to VRML: 3D Graphical Models from Video Sequences*, in Confluence of Computer Vision and Computer Graphics, NATO ARW, August 1999.