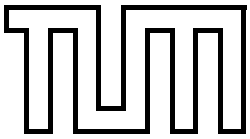
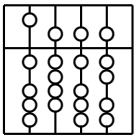


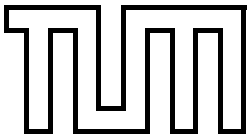
Technische Universität München
Fakultät für Informatik



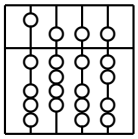
Diplomarbeit

Handling Error in Ubiquitous Tracking Setups

Daniel Pustka



Technische Universität München
Fakultät für Informatik



Diplomarbeit

Handling Error in Ubiquitous Tracking Setups

Daniel Pustka

Aufgabensteller:	Univ-Prof. Gudrun Klinker, Ph.D.
Betreuer:	Dipl.-Inform. Martin Wagner
Abgabedatum:	15. August 2004

Erklärung

Ich versichere, dass ich diese Ausarbeitung der Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. August 2004 Daniel Pustka

Zusammenfassung

Erweiterte Realität (engl. Augmented Reality) kombiniert die reale mit der virtuellen Welt, indem virtuelle Objekte in die Sicht des Benutzers auf die Realität eingeblendet werden. Um diese "Erweiterungen" an der richtigen Stelle anzuzeigen, werden verschiedene Tracking-Technologien eingesetzt, die den Ort und die Blickrichtung des Benutzers feststellen. Dies erfordert hohe Genauigkeit, hohe Aktualisierungsraten und geringe Verzögerungen, um eine überzeugende Darstellung zu erreichen.

Ubiquitous tracking ist ein neuartiges Forschungsprojekt mit dem Ziel, durch die Vereinigung von Ubiquitous Computing und Erweiterter Realität mobile und stationäre Tracking Systeme dynamisch zu integrieren, um mobile und weiträumige Anwendungen der Erweiterten Realität zu ermöglichen.

Um verschiedene, zur Entwicklungszeit nicht bekannte Tracking Technologien dynamisch zur Laufzeit zu kombinieren, sind Statistiken über die Sensorgenauigkeit nötig. Auch bei Transformationen der Messdaten müssen die dazugehörigen Fehlerbeschreibungen aktualisiert werden.

Automatische Umrechnungen zwischen verschiedenen Koordinatensystemen sind möglich, indem das Produkt von unterschiedlichen Tracker Messungen berechnet wird. In heterogenen Sensorumgebungen sind jedoch vor einer solchen Kombination Vorverarbeitungsschritte nötig, denn im Allgemeinen machen verschiedene Tracker ihre Messungen nicht gleichzeitig.

Diese Arbeit beschreibt ein mathematisches Modell sowie eine auf dem DWARF Framework basierende Software Architektur, um Sensorfehler von Positions- und Orientierungsdaten in Ubiquitous Tracking Systemen zu beschreiben. Mit Hilfe eines gaußschen Fehlermodells können Fehler über mehrere Koordinatensystem-Transformationen propagiert werden. Weiterhin wird ein auf Kalman Filtern basierender Ansatz mit getrennten Bewegungsmodellen vorgestellt, um dynamische Sensorfusion, die Gleichzeitigkeit von Messungen, Prädiktion, sowie das Schätzen von statischen Transformationen zu erreichen.

Abstract

Augmented reality is a technology that aims at combining real and virtual reality by integrating virtual objects into the user's view of the real world. In order to place these augmentations at the right locations, different tracking technologies are employed for finding out where the user is and in what direction he is looking. This requires high accuracy, high update rates and low delay to give a convincing impression.

Ubiquitous tracking is a new research project focused at dynamically integrating user-worn and stationary tracking systems by combining the concepts of augmented reality and ubiquitous computing in order to enable mobile wide-area augmented reality applications.

When many different and previously unknown tracking technologies are to be combined dynamically at runtime, statistics about the sensor accuracy are necessary and all transformations of measurements require updating the associated error descriptions.

In order to allow automatic conversions between different coordinate systems, the product of multiple tracker measurements can be computed. In heterogeneous tracking setups however, measurements by different sensors generally are not made simultaneously and therefore require pre-processing before such a combination is possible

This thesis proposes a mathematical model as well as a software architecture based on the DWARF framework for describing sensor errors in position and orientation in Ubitrack systems. Using a Gaussian error model, errors are propagated over multiple chained coordinate system transformations. A Kalman filter-based approach with separate motion models is described for dynamic sensor fusion, measurement simultaneity, prediction, as well as the estimation of static transformations.

Preface

Purpose of this Document This thesis was written as a Diplomarbeit at the Chair for Applied Software Engineering at TU München. From January to August 2004 two other diploma students and I designed and developed an implementation of the ubiquitous tracking concept. In this thesis I would like to explain the ideas behind my work, describe its results and show future implications.

Target Audience This thesis addresses various audiences. Here I would like to give an overview of what might be interesting for you.

General Readers unfamiliar with Augmented Reality should read chapters 1 to 4 in order to get an overview of the issues in Augmented Reality and ubiquitous tracking.

Augmented Reality Researchers are probably mainly interested in the basics of ubiquitous tracking (chapter 2), the error model (chapter 6) and the data flow architecture (chapter 7). Also the results in chapter 9 may be interesting.

Future Ubitrack Developers who already know the basics of AR and Ubitrack should read chapters 6 and 7 where the error model and the runtime implementation concepts are explained. Chapter 5 may be helpful in understanding the underlying mathematics. Also chapter 8 should be useful as a manual of the developer tools developed in this thesis. If you run out of ideas what to do, you can read the future work section in chapter 10.

Acknowledgments This diploma thesis would not have been possible without the help of some people to who I wish to say thank you at this place.

The members of the Ubitrack team, Dagmar Beyer and Franz Strasser for their help in writing down the concepts of Ubitrack and the DWARF implementation.

My supervisors, Martin Bauer, Asa MacWilliams and Martin Wagner for their valuable time in endless meetings.

Gudrun Klinker and Bernd Bruegge for opening up the wonderful research field of Augmented Reality at TU München.

The other attendants of the Salzburg workshop, Dieter Schmalstieg and Thomas Pintaric for their valuable discussions.

Finally, I would like to say thank you to Elissa Sobotta for her patience and unlimited support.

Contents

List of Figures

1 Introduction

In this chapter, I would like to give a short introduction of the context of this diploma thesis and explain its goals.

1.1 What is Augmented Reality?

Augmented reality (AR) is a technique for adding virtual objects to the user's view of reality. In classic AR systems, this is achieved using head-mounted displays (HMDs) which either are transparent (*optical see-through*) or where the real world remains visible by showing the images of a video-camera mounted on the display (*video see-through*). Other approaches also use laptops or hand-held computers as see-through devices or projective technologies [**?**]. Applications of AR range from navigation, where arrows point in the direction of the destination, over maintenance with interactive manuals to medicine, giving doctors "X-ray vision" by overlaying ultrasound or tomographic images onto the patient's body.

According to Azuma et al. [**?**], an Augmented Reality system has the following characteristics:

1. Combines real and virtual objects in a real environment
2. Interactive in real time
3. Registers (aligns) real and virtual objects with each other

The first criterion emphasizes that Augmented Reality adds virtual objects to the user's view of reality, which excludes CAVE environments where multiple people and objects can interact in a purely virtual environment. The second item disqualifies movies like Jurassic Park, that do not allow real-time interactions with the augmentation.

One of the biggest challenge in building Augmented Reality systems lies in the combination of points 2 and 3. In classical HMD-based AR setups, virtual objects (the augmentations) from the user's point of view must appear at certain places in the real environment, and therefore, the exact position and orientation of the user's head has to be known. The interactivity criterion requires that, as the user changes place, the augmentations stay at their real-world location and thus, the new head position must be immediately recognized and a new image has to be generated. This process of automatically determining position and orientation – the *pose* – of moving objects is called *tracking*. Many different tracking technologies exist, an overview is given in chapter 4.

1.2 What is Ubiquitous Computing?

In Weiser's vision [**?**] of future computer systems, today's desktop or mobile devices which require the user's attention to perform their tasks are replaced by many invisible devices embedded into the objects of every day's life. These devices allow a much more natural contact with computers than what is possible with the currently predominant WIMP metaphor on bulky screens. An instance of the ubiquitous computing paradigm is the well-known "intelligent building" where all kinds of devices, from garage doors over refrigerators to telephones are networked and know about the current *context* as well as the user's desires and do things without requiring manual intervention.

An important component of the context in ubiquitous computing applications is the location of users and things. To determine locations in ubicomp systems, usually a large number of sensors is installed in the environment. However, these sensors typically are unuseable for Augmented Reality, as they do not provide the necessary resolution (often, they only sense "proximity" of two objects) and update rate.

1.3 Introduction to Ubiquitous Tracking

Note: *This section was jointly written with Dagmar Beyer and Franz Strasser.*

Augmented Reality is a natural interface to Ubiquitous Computing environments. However, classical AR applications are limited to the range of their tracking systems. When they use wide-area trackers, the update rate and accuracy of the measurements are low and quite often do not fulfill the requirements of AR. Moreover, they assume that sensors are deployed statically in the area of interest and that these sensors are pre-calibrated off-line. These drawbacks inhibit the development of mobile, large-scale, and dynamic systems as they are found in Ubicomp. Every tracking method also suffers from various disadvantages due to the employed technology.

Ubiquitous Tracking combines available tracking technologies from both ubiquitous computing and augmented reality to one large sensor network. It provides unified APIs for applications and transparently delivers measurement results. The data is fused automatically and dynamically from heterogeneous sources.

1.3.1 Motivation for Ubiquitous Tracking

Every tracking technology suffers from various disadvantages: inertial trackers provide high update rate, but make only relative measurements and suffer from drift. Video-based camera systems need lots of computing power for image processing and the accuracy along the line of sight is rather low. Sensors also have a limited range where they can act. In contrast, Ubiquitous Computing environments are typically large-scale with many sensors.

The weaknesses of the different technologies are compensated by using different sensors at the same time. For example, camera-based methods can be paired with inertial tracker to compensate the drift and provide a high update rate. By adding GPS one can even get a global frame of reference.

To extend the range of sensors or the provide higher accuracy, *sensor fusion* combines the measurements of different trackers. You can combine stationary indoor tracking systems with GPS in order to extend the operation range. Significant and pioneering work has be done, amongst others, by Azuma [**?**], et al. (see related work 2.5). However, no attempt has been made so far to automate the integration of sensors into large-scale sensor networks.

Classic AR applications use static tracking platforms designed exactly for that one purpose. So reusing tracking components is hard to achieve. Therefore, Ubiquitous Tracking also tries to abstract from tracking hardware by introducing a reusable, component-based model of data flow and processing.

1.3.2 Goals of Ubiquitous Tracking

The goal is to obtain an optimal estimate of arbitrary geometric relationships and their accuracy at any time, for a user-specified definition of optimality. Such a tracker abstraction can help applications handle the varying levels of tracker uncertainty that affect the registration of virtual objects in 3D space.

The system itself is designed to allow dynamic cooperation between distinct and mobile components. Changes in the infrastructure, i.e. adding new hardware or occurring failures, are recognized whereupon the system tries to minimize the error or even to deliver better results. The developer is shielded from these influences by a software layer which offers a simple, efficient API to the developer.

1.3.3 Research Challenges

These goals however imply problems which have to be addressed:

1. The most obvious issue is the incompatibility between the output of different tracking technologies. For example, one sensor sends its orientation information as rotation matrix the other as Euler angles. One sensor delivers position and orientation whereas the other only accelerations.
2. Tracker measurements vary in their accuracy. A generic error model has to be found to specify all possible measurements errors which may occur.
3. All objects which should be 3D-registered in the application have to be modeled and identified. If new objects are tracked, the model must be updated on the fly.
4. Since Ubiquitous Tracking abstracts from the hardware, it must deal with changes in the infrastructure at runtime. If hardware fails, the application should not be aware of that fact.

To address these issues a new project called *Ubitrack* was started.

1.4 The Ubitrack Project

Note: *This section was jointly written with Dagmar Beyer and Franz Strasser.*

In the winter term 2003/04 the Augmented Reality group of the Applied Software Engineering chair at the Department of Informatics, Technische Universität München offered diploma theses on the topic *Ubiquitous Tracking*. Three students, Dagmar Beyer, Daniel Pustka, and Franz Strasser started to work on an implementation of Ubiquitous Tracking. The AR group in Munich and the Vienna University of Technology, Austria developed a formal Ubiquitous Tracking model [**?**]. The new joint project between these universities was named *Ubitrack* .

The overall purpose of the three diploma theses in Munich was to find an implementation of Ubitrack which should be usable in Augmented Reality applications but should be based on a distributed, decentralized Ubicomp architecture at the same time. In the following months we developed a DWARF system model of Ubitrack , mapped it onto new and existing components of DWARF, implemented a demo setup, and evaluated our own theses with this demo. The detailed concept and models are explained in chapter 2.

1.4.1 Design Goals

To define the design goals and the API, a workshop was held from February, 6th till 8th 2004 at Erentrudisalm near Salzburg, Austria. The resulting design goals are:

Transparency Ubitrack should be framework which provides a device-independent query mechanism for applications. So the framework should shield the application from the tracking hardware as much as possible.

Scalability Since AR applications are enriched with concepts of Ubicomp, the desire for large-scale tracking networks is growing. Such networks must be scalable in respect of the size of available hardware and infrastructure because current intelligent environments may contain thousands of sensors.

Flexibility Ubitrack must be a very flexible framework with respect to changes in the infrastructure. New hardware should be easily integrated at runtime and changes in the communication network should be considered.

1.4.2 Layered Architecture

Transparency is achieved by introducing a three layer architecture model. The application is shielded from the hardware by introducing an intermediate *Ubitrack layer* (figure 1.1). Between these layers there are defined APIs which allow the application and the hardware to communicate with Ubitrack .

All parts of the application that want to communicate with Ubitrack lie in the application layer. The *Query API* is the interface between the application and Ubitrack layers and defines how queries have to be structured and how results are delivered.

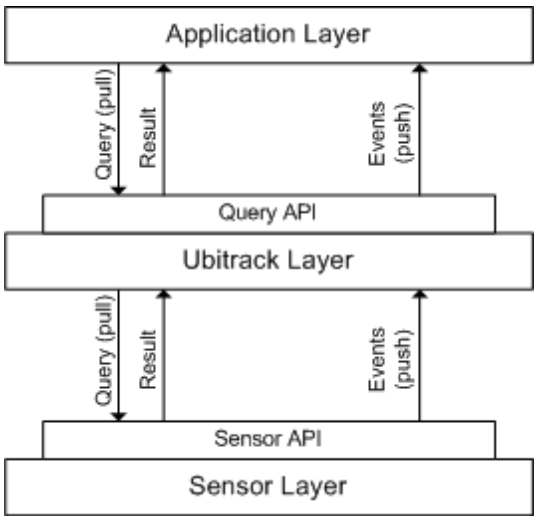


Figure 1.1: The three layer Ubitrack architecture model

In the hardware layer are all components which deliver positional information about objects. They range from sensors to databases where static relationships are stored, e.g. relationships between walls in a building. These low-level results are aggregated in the Ubitrack layer and delivered according to the query to the application. The *Sensor API* defines how low-level results are delivered to the Ubitrack layer.

1.5 Goal of the Thesis

All sensors used in Augmented Reality or ubiquitous computing systems make errors. These are either caused conceptually (e.g. infrared beacons that can only detect proximity), by manufacturing constraints (e.g. the maximum resolution of a camera) or through environmental influences (e.g. magnetic fields or loud noise). As such kinds of errors cannot be removed completely, a description (model) of these errors is required, especially in Ubitrack systems, where arbitrary, previously unknown sensors are to be integrated.

Of course, it would be possible to ignore sensor errors and work with the "best" approximation returned by the tracker – in fact, this is done by many existing system. However, if multiple tracker measurements are to be combined in a sensor fusion process, the accuracy must be known in order to decide which measurement to trust more.

The goal of this thesis is the development of a uniform description of sensor errors and the design of software components necessary for handling these errors in a DWARF-based Ubitrack system.

2 Ubiquitous Tracking and DWARF

This chapter explains the basic ideas of ubiquitous tracking and the DWARF middleware and outlines the key concepts of a distributed Ubitrack implementation in DWARF.

2.1 Formal Framework

The purpose of this section is an explanation of the formal model which builds the theoretic foundations of the ubiquitous tracking system.

2.1.1 Spatial Relationship Graph

Note: *This section was jointly written with Dagmar Beyer and Franz Strasser.*

Within an Augmented Reality environment, it is essential to maintain an awareness of its occupating objects and their physical relationship to another. In addition it must be possible to configure the setup of the environment dynamically and to extend it easily. The goal of the formal model discussed in this section is to provide a method to obtain optimal estimates of the spatial relationships between arbitrary objects within an Augmented Reality environment. For this purpose an coherent, up-to-date spatial model of the environment is essential.

A suitable data structure to store the spatial model is a directed graph. Nodes in this *Spatial Relationship Graph* (SR Graph) represent locatable objects, which could be either active sensors such as cameras or passive objects such as markers or other targets. The directed edges represent the spatial relationships between two objects, i.e. the position and orientation of the object represented by the target node relative to the object represented by the source node.

Real Relationships

From a general point of view, for each pair of objects a spatial relationship can be determined for each point in time. This spatial relationship can be expressed by a transformation of the coordinate frame of the source object to the coordinate frame of the target object, for example by using the computer graphics notation of a 4×4 homogeneous matrix, a $3D$ translation vector and a quaternion, or any alternative representation. In the following, we will represent the transformation by using a 4×4 homogeneous matrix. In figure 2.1 an example for such a Spatial Relationship Graph is given containing three objects A, B, and C.

Defining an binary relation Ω on the object space N (in our example, $N = \{A, B, C\}$), each element $(X, Y) \in \Omega$ can be mapped onto a function w_{XY} describing the transformation of

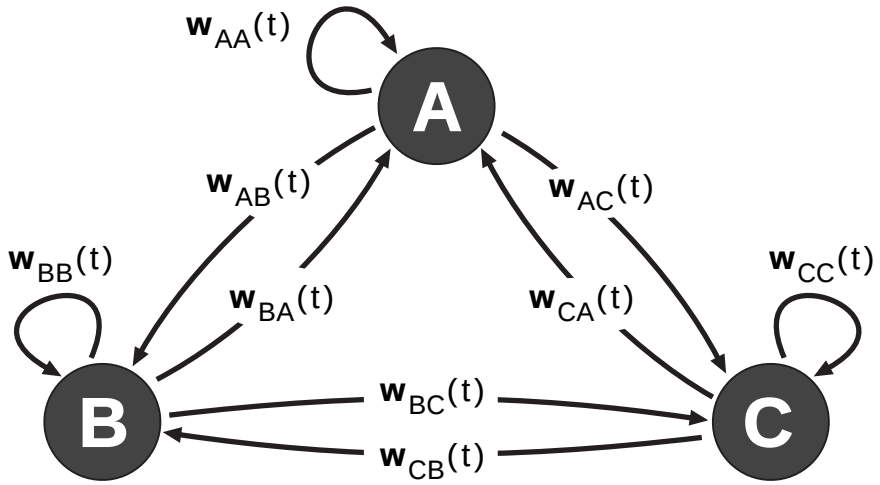


Figure 2.1: Spatial Relationship Graph according to relation Ω, showing the real spatial relationships between objects

the coordinate frame of X to the coordinate frame of Y over time. This attribution scheme is called $\mathbf{W}$.

$$\mathbf{W} : (\Omega = N \times N) \rightarrow w, \text{ where } w : D_t \rightarrow \mathbb{R}^{4\times4} \tag{2.1}$$

Each directed edge in our example Spatial Relationship Graph is annotated by a function w that maps the time domain D_t onto the spatial relationship domain $\mathbb{R}^{4\times4}$. In this complete graph, all spatial relationships between arbitrary objects can be obtained at each point in time.

Measured Relationships

Needless to say, that it is not possible to have such an omniscient view on the environment's underlying geometry as described above. So in real world, only estimates of spatial relationships exist on the basis of measurements taken at a discrete point in time. The measurements representing the real spatial relationships are corrupted by noise and systematic errors, depending on the accuracy provided by the tracking devices. In order to take account for this varying quality of the estimates, a set of attributes is added to each measurement, which describes properties such as the standard deviation of the positional information or latency between the actual measurement and the time this measurement is delivered. In section 2.1.2 a more detailed view on these attributes is given. In order to extend the formal model to these further considerations, a relation Φ and an attribution scheme $\mathbf{P}$ is defined:

$$\mathbf{P} : (\Phi \subseteq N \times N) \rightarrow p, \text{ where } p : D_t \rightarrow \mathbb{R}^{4\times4} \times \mathcal{A} \tag{2.2}$$

A directed edge between two objects X and Y exists only if measurements have been taken. The function p_{XY} maps each distinct point in time where such a measurement is made to a homogeneous matrix representing the measured spatial relationship and a set of

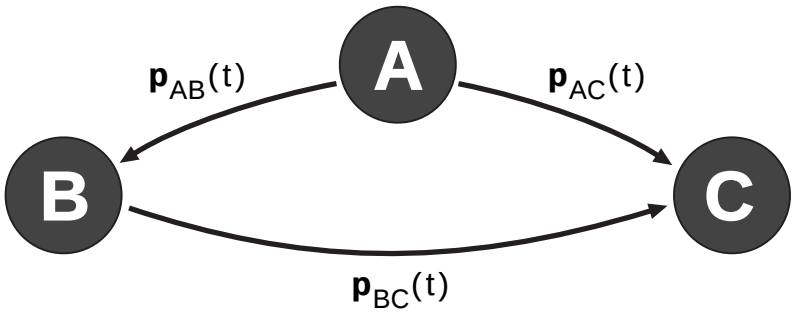


Figure 2.2: Spatial Relationship Graph according to relation Φ, showing the measured spatial relationships between objects

attributes describing the quality of this measurement. An example for a Spatial Relationship Graph according to relation Φ is given in figure 2.2.

Assuming that in the example the geometric relation between A and B was measured twice at time t_1 and t_2, this leads to two homogeneous matrices H_1 and H_2 with according sets of attributes $\mathcal{A}_1$ and $\mathcal{A}_2$. Hence, function p_{AB} is defined as:

$$\begin{array}{rcccl} p_{AB} & : & \{t_1, t_2\} & \rightarrow & \mathbb{R}^{4\times 4} \times \mathcal{A} \\ & & t_1 & \mapsto & (H_1, \mathcal{A}_1) \\ & & t_2 & \mapsto & (H_2, \mathcal{A}_2) \end{array} \tag{2.3}$$

Inferred Relationships

This estimation of spatial relationships is in many ways insufficient. Due to the limited range and the specific properties of tracking devices, it is not possible to provide information about spatial relationships between all objects. As measurements are performed at discrete points in time, it is also impossible to provide estimates of the spatial relationships continuously. Therefore it is very unlikely, that an application's request for a specific spatial relationship can be fulfilled by the measured relationships. It becomes necessary to extend the knowledge of the environment's geometry by inferring new estimates for spatial relationships on the basis of measured relationships. Accordingly the attributes describing the quality of the inferred spatial relationships must be adjusted to reflect the loss of quality in respect to the quality of measured relationships.

In order to provide positional information about objects at arbitrary points in time, interpolation or extrapolation functions can be used, or for a more accurate prediction Kalman Filters. Inverse edges can be added to the Spatial Relationship Graph by inverting the measured relationships. Estimates of spatial relationships at the same point in time can be combined along *paths* in the Spatial Relationship Graph in order to form the *transitive closure* of the graph. Just as measured relationships are provided by different tracking software components, each inferred relationship is provided by a new software component performing this inference on the measured positional information. For a more detailed description of these new software components, see section 2.1.3.

All these possible forms of inferring new spatial relationships lead to the final goal of defining a binary relation Ψ that approximates the idealized relation Ω in real world:

$$\mathbf{Q} : (\Psi \subseteq N \times N) \rightarrow Q, \text{ where } Q = \{q \in Q \mid q : D_t \rightarrow \mathbb{R}^{4\times4} \times \mathcal{A}\} \tag{2.4}$$

Each element $(X, Y) \in \Psi$ is mapped onto a set of functions q'_{XY}, which are describing specific forms of inference.

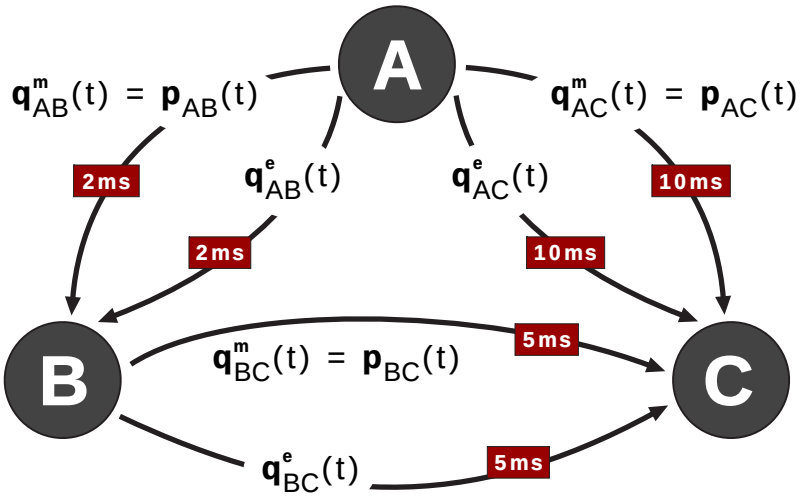


Figure 2.3: Spatial Relationship Graph according to relation Ψ, containing measured relationships as well as inferred relationships.

In figure 2.3, the example Spatial Relationship Graph is shown, according to relation Ψ. The originally measured relationships are now represented by the function q^m, which is identical to the function p in relation Φ. On basis of these measured relationships, new relationships are inferred. Additional edges are inserted, annotated with function q^e, which represent the geometric information derived from extrapolation on basis of the measured relationships. The quality of the estimated spatial relationships is indicated by a single attribute, the latency of the measurement.

Assuming an application requests positional information about object C relative to object C at time $t = T$, this request can be fulfilled with the estimated spatial relationship $q^e_{AC}(T)$. Alternatively a new edge could be added to the Spatial Relationship Graph, annotated with function $q^f_{AC}(T)$, which represents the combination of the estimates $q^e_{AB}(T)$ and $q^e_{BC}(T)$.

In order to choose an optimal estimate for the requested spatial relationship, the application defines an evaluation function, that maps attributes on a real non-negative value. The spatial relationship with the best result value is returned to the application as the result of it's query.

2.1.2 Attributes and Evaluation Function

Each edge in the spatial relationship graph provides, in addition to the actual measurement values, attributes that describe the quality of the measurement. These attributes most impor-

tantly serve as selection criteria when multiple paths fulfill an application's request, but they can also be used as a weight when multiple measurements are averaged or to adapt the user interface of an AR application to the currently available tracking resolution, as demonstrated by [**?**]. Examples of such quality attributes relevant for augmented reality are:

Latency describes the time (in seconds) between the actual measurement and the availability of its result to the rest of the system.

Update frequency measured in $1/\mathrm{s}$, indicates the rate at which trackers make their measurements.

Confidence value attributes are important for optical trackers that may misclassify video images and detect non-existent features. A well suited range would be $[0; 1]$, indicating the probability that the identification is correct.

Pose accuracy determination is the major problem when developing a tracker abstraction to integrate multiple trackers [**?**]. A simple approach is the use of Gaussian noise models. The pose accuracy is then described by a covariance matrix which defines an (hyper-)ellipsoid around the measured position in which the real position lies with a probability of 68%.

Monetary cost per measurement may be significant in the design of tracking setups. The tracking graph can incorporate all the trackers available on the market and then a suitable evaluation function can be evaluated for the desired relationships between objects. The resulting optimal path consists of the trackers that should be installed.

Motion models are relevant when prediction has to be performed in order to compensate sensor and rendering delays in augmented reality applications or when short sensor dropouts have to be bridged. A motion model tells the Ubitrack system which derivations of the measurements to use for prediction, i.e. if the expected motion is constant pose, constant velocity or constant acceleration. The motion model also describes how much the motion may diverge from this simple model.

A special case of a motion model is one which tells the system that the measured relationship between the two objects is static. In this case, the Ubitrack components will assume that all measurements are infinitely valid and not perform any extrapolation but use the value of the last measurement or an average over all measurements instead.

Although these attributes can be defined easily, it is often not trivial to obtain reasonable values from specific trackers. Manufacturers of tracking hardware usually only provide average or even best-case accuracy values in the documentation, but do not make such information available in real-time for each measurement. In addition, when measurements are transformed by inference or filter components, the quality attributes of the resulting edge have to be adjusted accordingly.

Evaluation Functions

In some cases, an application's query may result in multiple paths in the spatial relationship graph that describe relationship between two objects. To resolve this, the developer of a

Ubitrack application has to specify an evaluation function which is used to discriminate between different solutions. An evaluation function takes as an argument a whole path, evaluates the attributes along its edges and computes a scalar value that expresses how well the solution suits the application's need, with lower values meaning better solutions. Equation 2.5 gives an example of an evaluation function that aims to tradeoff latency against update rate.

$$e^t := \sum_{q \in path} \text{lag}(q) + \frac{\lambda}{\text{rate}(q)} \tag{2.5}$$

λ determines the weighting of lag versus update rate. This example also shows that it is possible to create evaluation functions that computes a value from the attributes of a single edge and then sums them up along the path. In this case we can use well-known and efficient search algorithms on weighted graphs, such as the algorithms of Dijkstra or Bellman-Ford [**?**].

2.1.3 Data Flow Graphs

In order to compute a desired relation from a given path in the spatial relationship graph, the Ubitrack middleware constructs a tree of components. Raw measurements are inserted at the leaves by tracking services and propagated to the root, undergoing various transformations at each node. The root is formed by the application which receives the computed measurements. The interior of the data flow graph consists of interpolators, extrapolators, filter, inference and data fusion components. This concept of a data flow graph can easily be mapped onto a network of DWARF services or OpenTracker nodes. An example of a data flow graph is shown in figure 2.4.

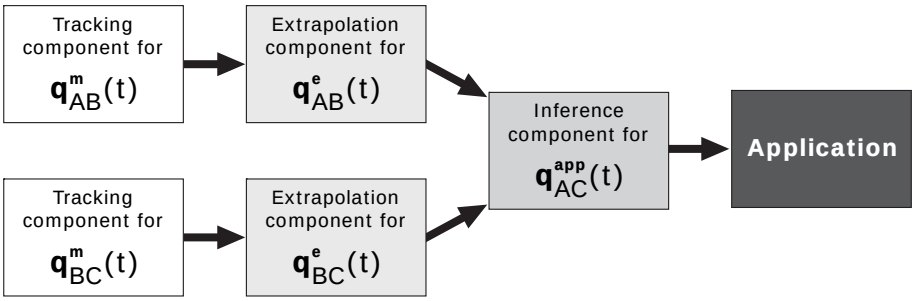


Figure 2.4: Example data flow graph

A typical Ubitrack data flow graph consists of the following components:

Tracking components provide a stream of measurements, consisting of pose information and the quality of the measurement. The measurements usually come from some tracking device such as an optical tracker. It is also possible to introduce previously recorded or simulated data for algorithmic evaluation.

Static components provide static measurements, e.g. from a database, consisting of pose information and the quality of the measurement. This information may have been derived from previous measurements, e.g. by averaging, or may have been entered manually. Examples of static measurements occurring frequently in AR systems are the relations between fixed markers on a wall or the offset between an HMD and an attached camera for optical tracking.

Extrapolation, interpolation and filtering components take a stream of measurements, perform some computation on it and insert the result back into the data flow graph.

Extrapolation and interpolation are of particular importance in a Ubitrack system. When a new relation is computed by combining measurements along a path, this computation only is valid when all measurements are taken at the same time. In inherently heterogeneous Ubitrack systems, this however rarely is the case. Therefore interpolation or extrapolation must be applied in order to compute measurements from the raw sensor data, that are valid at the same moment. A popular tool for extrapolation is the Kalman filter which also smoothes noisy data [**?**], [**?**].

Inference components are the heart of a Ubitrack system. They take two or more measurements that correspond to the edges along a path in the spatial relationship graph and compute the inferred relation from the beginning of the path to its end. As all of the input measurements must be valid at the same moment, inference components in the data flow graph are usually preceded by inter-/extrapolation components.

Fusion components take input measurements from at least two different sensors measuring the same relation and compute a new combined measurement that ideally should be better than each of the inputs alone. In the simplest case the measurements are weighted by their error covariance. More complex sensor fusion setups can combine sensors with different characteristics, e.g. an optical tracker that provides high precision measurements but suffers from high latency and casual dropouts with an inertial tracker that drifts, but has low latency at a high update rate [**?**], [**?**].

Pose inversion components take a measurement as an input and compute the inverse relation. When an application requests the relation between two targets A and B that are tracked by a single optical tracker T, providing the measurements $A \rightarrow T$ and $B \rightarrow T$, $B \rightarrow T$ has to be inverted before the requested relation $A \rightarrow B$ can be computed by an inference component.

2.2 DWARF

Note: *This section was jointly written with Dagmar Beyer and Franz Strasser.*

The Distributed Wearable Augmented Reality Framework (DWARF) is a component-based framework which allows the rapid prototyping of AR applications. It was developed at the Chair of Applied Software Engineering of the Technische Universität München (TUM) [**?**, **?**]. The idea behind DWARF was that using distributed components on several machines and connecting them in an ad-hoc manner will provide more flexibility when developing new AR applications. By combining the aspects of Ubiquitous Computing and Augmented

Reality concepts into DWARF it is possible to build a new AR system using already existing components: e.g. a tracker or viewer components can be used not only by one application but by several applications, sometime even simultaneously.

This section gives an small overview of the concepts of DWARF which are relevant for Ubitrack. It focuses on introducing the high-level modeling concepts and communication mechanisms.

2.2.1 Services

Applications implemented with DWARF use interdependent, collaborative *services*; components which may be distributed over the network. Every service runs as discrete unit, mainly as one process, on a network node. The whole application is formed by combination of services which are in general very specific components and perform a certain task: they are the units of DWARF.

Every service is described by its *service description*. These descriptions are stored in XML files which are evaluated by a middleware component, the DWARF service manager (see section 2.2.3). They contain all relevant information about the service, e.g. what features are provided for others or what features are required. These relationship of dependence are modeled by *needs* and *abilities*.

2.2.2 Needs and Abilities

The relationships between services, i.e. how to connect services together, is modeled by *needs* and *abilities*. Abilities are features which are offered to other services. A need the expression of a desire for an ability. The specification for a need or ability contain the following:

1. The *name* is an identifier to distinguish them
2. The *type* describes what kind of information is delivered or requested. Only when the types of need and ability matches, the communication is established.
3. The *connector protocol* describes what communication mechanism is preferred (see below).

Attributes

Abilities may specify *attributes*[1] which are generic name-value pairs that may contain arbitrary information. Attributes normally ar high-level description about the certain additional features or qualities of the ability. The attributes may be evaluated by the service itself to adapt its behavior at runtime. But mostly they are used by needs to constrain the amount of possible partners because needs can specify predicates.

[1] DWARF attributes are distinct from spatial relationship edge attributes described in section 2.1.2, although SR graph edge attributes can be mapped onto DWARF attributes.

Need Predicates

In cases where there are more than one ability which can satisfy a need, *predicates* restrict the choice of possible partners. If it is declared in the need, the attributes of the potential partner services are evaluated and only if the predicate matches the attribute settings, both will be connected.

Connectors

Every need or ability has one or more *connectors*. For every communication method exist a connector protocol that must be named in the description. DWARF service can currently use four different communication methods.

CORBA remote method invocations can be used for inter-process and inter-network communication. CORBA , the Common Object Request Broker Architecture, is a platfrom-independent, object-oriented architecture and infrastructure standardized by the Object Management Group (OMG) [**?**]. By using the Interface Definition Language (IDL) the developer can define remote class interfaces which are called by remote method invocations. The connector protocol for this type of communication is called `ObjrefExporter`, because this services provides objects which can be imported on the remote side. Thus the partner service uses the `ObjrefImporter` connector protocol.

CORBA notification service allows event-based communication between services. This is the most common way of communication between services because it is an asynchronous method. The information is packed into a CORBA structure and delivered over an *event channel*. The data may be arbitrary however the receiving service must correctly process the structure. In DWARF the type of the need/ability specifies what data is stored in the CORBA event. The connector protocol for sending events is called `PushSupplier` whereas receiving services use the `PushConsumer` connector protocol.

Shared memory The third communication method is shared memory. If the services are on the same computer one can write the information in a POSIX shared memory block from where the other can read it. DWARF only allocates the memory area and provides mechanism for synchronization; the services must handle the concurrent access to the blocks themselves. In contrast to the former mentioned ones, this method only use one connector protocol, the `Shmem` protocol, on both ends of the communication.

By convention, when a service provides data it has an ability; if it requires data, it has needs. However, this data flow direction if not coupled with the need-ability-relationship; the direction is only modeled by the connectors. There are situations where it is sensible to revert the communication direction. This is supported, since the connectors are independent from needs or abilities.

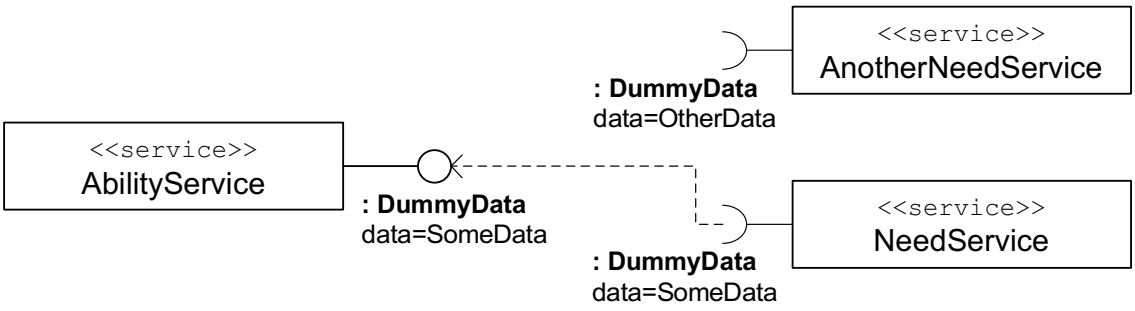


Figure 2.5: This UML diagram shows an example session of three services. Two of them are connected, one is not since the predicate did not match. The annotations in the figure describe the individual UML notations.

IDL interfaces

The connectors describe the communication method, but how actually the interfaces look like is specified in CORBA's Interface Definition Language (IDL). IDL is an type description language which allows to specify object-oriented class interfaces. Mappings of IDL exist for several programming languages including C++, Java and Python. Every of the three available connector protocols use one or more IDL interfaces to implement the communication method.

Any component using a specific protocol may implement an arbitrary IDL interface. However the remote end must know which interface may be called. This information is stored in the ability type. Every interface is designed for a specific ability. So the remote end knows what interface can be called since it has the same ability type.

UML Notation

Services and need-ability-relationships may be modeled by UML diagrams. They are used to represent the system design of DWARF applications. The special DWARF notation extends the normal UML by defining a new need-ability-relationship association. These diagrams will be used throughout this thesis for representation of the DWARF models. Abilities are drawn as filled circles attached by a line (similar to UML interfaces), needs on the other hand are semi-circles. Both are attached by a UML dependency arrow. The need always depends from the ability.

Figure 2.5 shows three example services. Only two of them are connected since the attributes of one do not match the given predicate.

2.2.3 Middleware

The DWARF *service manager* allows the development of distributed DWARF applications. Every service manager resides on a network node and dynamically finds all other managers in the net. Every service manager knows its own local services by parsing the local XML service descriptions. When a local service is started, it *registers* itself at the manager and become

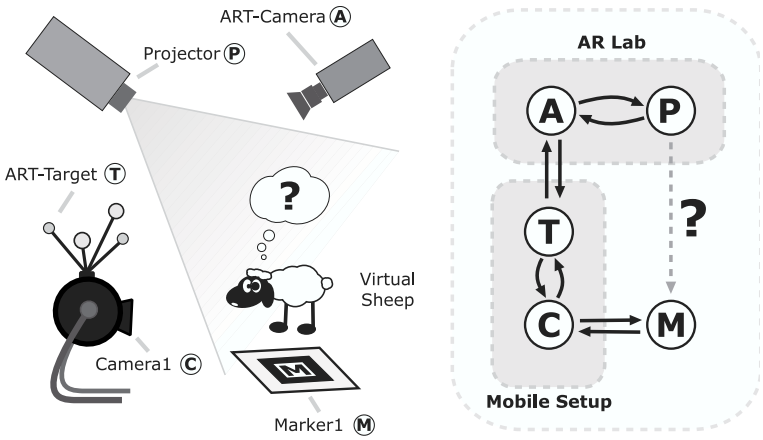


Figure 2.6: The example setup and its spatial relationship graph

active. If it has needs, the service manager tries to find a suitable service by requesting other managers for potential partners. If a service is found both managers establish the desired connectors and create a direct communication channel between the services.

If a service needs an ability and a corresponding service allows it, the service manager can start the ability service on demand. This feature can be used to start whole *service chains* by manually starting only one service.

2.3 An Example

Note: *This section was jointly written with Dagmar Beyer and Franz Strasser.*

This section introduces a simple example scenario which illustrates the the Ubitrack concepts. This example is used to explain how the Ubitrack concepts are mapped onto DWARF in the rest of this chapter. A stationary tracker combined with a mobile camera delivering images to an optical tracker, effectively allows the system to "see around corners". Figure 2.7 shows a camera tracked by an ART tracking system. Figure 2.6 shows both a schematic of the hardware setup, and the corresponding Spatial Relationship Graph. A ceiling-mounted projector P displays a pastoral landscape, populated by a sheep, on a table. The projector is calibrated such that it shares the same coordinate system as the ART tracker A; therefore, a static relationship exists between P and A. A mobile user is equipped with an ARToolkit [**?**] camera C on which an ART target T is mounted, resulting in another static relationship described by another edge in the graph. The camera is attached to a notebook with a wireless network interface. The middleware is running on the lab computers and on the notebook.

When a user enters the room, the two Ubitrack systems connect, and the description of an ARToolkit marker is transferred to the user's notebook enabling the camera to track it. As long as the marker remains in the view of the camera, and the ART target attached to the camera is tracked by the ART system, then the virtual sheep can still be tracked in the coordinate frame of the ART system, even if it is physically out of range. Consequently, when the marker is moved the image of the sheep displayed by the projector can be seen to

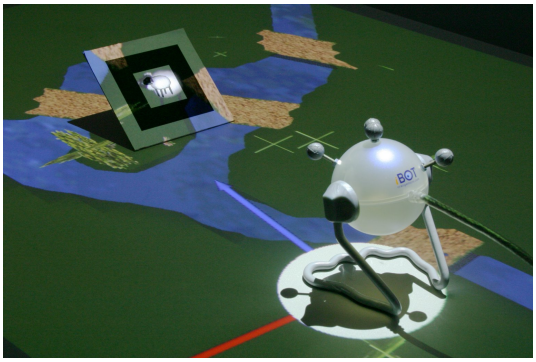


Figure 2.7: Picture of an example scenario implemented in DWARF

move accordingly. If the sheep is moved into a pen outside the range of the projector, it can still be viewed using some other tracked or fixed display.

The combination of these two tracking technologies is not novel in and of itself; however, this example focuses on the inference of spatial relationships and the resultant spontaneous behavior of the system as it reacts by combining tracking sensors.

2.4 Modelling Ubitrack in DWARF

Note: *This section was jointly written with Dagmar Beyer and Franz Strasser.*

Based on the example scenario a suitable model for DWARF was developed. This model takes the Ubitrack concepts and maps them onto existing DWARF techniques, resulting in the integration of existing DWARF components with new ones. This section only introduces the new key concepts. They are explained in the different diploma theses of the Ubitrack project in more detail.

2.4.1 Sensor API

The Sensor API is the interface between the hardware and the Ubitrack layer, which all services that control a sensor and deliver positional information have to implement. The interface is mapped to an ability with type `PoseData`. Every measurement, i.e. every edge in the Spatial Relationship Graph is mapped onto exactly one ability of such tracking services. So every ability has a unique pair of source and target *object identifiers*, the corresponding nodes of the edge. In DWARF, these identifiers are stored in attributes called `UTSource` and `UTTarget`.

This ability delivers positional information in form of an IDL data structure of type *PoseData*. This IDL structure contains the measurement values from the sensor hardware, i.e. spatial measurements of objects in the frame of reference of the sensor. The PoseData fields are described in more detail in chapter 7.

2.4.2 Query API

Applications receive spatial information from the Ubitrack layer through the Query API. In DWARF this is realized by a need of type `PoseData`. What information is desired depends on the predicate of the need. Required predicates are `UTSource` and `UTTarget`. Given a certain evaluation function, quality attributes can also be specified in the predicate which restricts to tracking services with, for example, the given accuracy.

2.4.3 Query mechanisms

The connector of the need implies the delivery mechanism for the results. There are three different query mechanism which work synchronously or asynchronously, with the push or pull principle.

Asynchronous Push

This is the most common query mechanism in current DWARF application for PoseData. It uses the CORBA notification service for sending PoseData events whenever a measurement is possible. Depending on the sensor technology this can be in constant update rate or exactly on recognition of a locatable. The `PoseData` ability uses a `PushSupplier` connector and probably sets an attribute containing the update rate in Hz ($1/s$).

Every ability must implement the `UTPoseDataAsyncPush` IDL interface, which is a derived interface of the `SvcProtPushSupplier` interface. Later is used in current DWARF implementation as default interface for `PushSupplier` connectors. The use of a derived interface in Ubitrack allows future extensions to the asynchronous push mechanism.

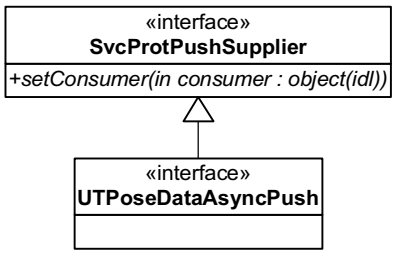


Figure 2.8: The `UTPoseDataAsyncPush` interface

Synchronous Pull

The synchronous pull is modeled by a remote method call, so it uses the `ObjrefExporter` connector to offer its interface for the application. The IDL interface which must be implemented for this pull mechanism is the `UTPoseDataSyncPull` interface. This interface currently consists of only one method: `getPoseData`. The method gets an timestamp as parameter and blocks the execution until a result is available.

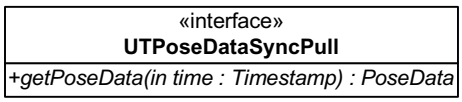


Figure 2.9: The `UTPoseDataSyncPull` interface

Asynchronous Pull

The most complex interface is the asynchronous pull interface: `UTPoseDataAsyncPull`. It aggregates the `SvcProtPushSupplier` interfaces as well as its own method: `wantPoseData`. This method takes a timestamp and returns immediately after scheduling a PoseData event for the given time. When a PoseData event is available it is delivered over the event channel.

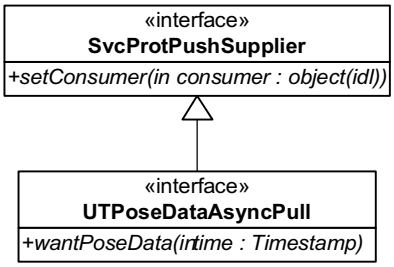


Figure 2.10: The `UTPoseDataAsyncPull` interface

2.4.4 Ubitrack Middleware Agent

Note: *This section was written by Dagmar Beyer and is a summary of her thesis [**?**].*

The *Ubitrack Middleware Agent* (UMA) is a distributed middleware component. Allocated in the Ubitrack Layer of the architecture model, it focuses on three aspects of the Ubitrack concept:

- Aggregation of information about locatable objects and their geometric relationship to build a distributed representation of the Spatial Relationship Graph.
- Aggregation of inferred spatial information between objects requested by applications by executing a distributed path search in the Spatial Relationship Graph.
- Set up of data flow to provide the requested spatial information to an application.

Representation of the Spatial Relationship Graph

On each host in the network, a single UMA maintains a representation of the local Spatial Relationship Graph. Edges in this graph are associated with the abilities of software services to provide measurements of the according spatial relationships. These services have a `PoseData` ability implementing the Sensor API. Attributes describing the quality of the

estimated positional information are mapped on the ability attributes. In order to reflect the infrequently changing quality of the estimates, these attributes have to adjusted in lengthy interval.

In order to obtain information about spatial relationships, the UMA has to parse the service descriptions of local services providing measurements of spatial relationships. Therefore the UMA gets connected to the local service manager, which obtains all service descriptions of locally running services. For initialization, the UMA polls the list of available service descriptions, but after this the UMA receives notification messages from the service manager whenever the properties of a locally running services change. The notification messages also indicate the start up of new services and the shut down of already running services.

A first inference process can extend the knowledge of the spatial relationships by inverting relations and adjusting the corresponding attributes. This leads to a set of inverse edges and results in an undirected Spatial Relationship Graph.

Locatable objects can take part in multiple spatial relationships stored in different UMAs in the network at the same time. UMAs storing spatial relationships to identical objects need to interact in order to find optimal estimates of requested spatial relationships by performing a distributed path search. A bidirectional event channel must be established between them for passing messages during the search process. The UMA indicates for which objects it requires connections to send and receive search messages by configuration of the according needs and abilities in its own service description. The connection of matching needs and abilities, and thereby neighboring UMAs, is done by the DWARF middleware.

This distributed representation of the Spatial Relationship Graph contributes to the systems scalability and flexibility. Centralized storage of the Spatial Relationship Graph or providing coherent copies at each host would lead to an immense communication overhead and long response time to changes in the topology. In a distributed representation, the AR system is easily to extend and mobile clients equipped with their own tracking setup and running AR application can be integrated seamlessly.

Distributed Queries and Path Search

An application indicates an request for a specific spatial relationship by implementing the Query API in a service need. By parsing the service description of the application, the local UMA obtains information about the properties of this request.

The path search between the UMAs is implemented using a distributed asynchronous Bellman-Ford algorithm [**?**]. In a first step, the path search is performed on the local subgraph. Even though a suitable estimate of the requested spatial relationship might be found in the local subgraph, there is no certainty that this estimate is the optimal result to the application' s query. Therefore the path search must be extended to all adjacent UMAs.

Search requests are sent out containing following properties: a search identifier, the current node visited during the path search (initially this is the source node), the target node, and an evaluation function that computes edge weights from attribute sets. By evaluating the attribute sets along the path from the source node to another node, a distance value is obtained on which Bellman-Ford is based. This distance and the according path are stored additionally. When a node is visited during a path search, the distance of this node from the

source node is updated. When a search computes a new minimum distance between a node and a source node, this new distance must be propagated to neighboring UMAs. Therefore a participation identifier is stored additionally for each distance update that is sent out.

When a search event is received by the UMA, the distance associated with the current path is compared with those from earlier participations. If the new distance is less than that of former participations, the path search must be continued again from this node. The distance from the current node to all other local nodes is updated by computing minimum-cost paths and new search events are sent out for them to relevant UMAs.

If the distance is greater than that of the current node or the target node is found, an acknowledgement message is sent back. Each time a UMA participates in a path search, it must wait for acknowledgement messages or a timeout before sending acknowledgement message back on the path towards the source node. After all search messages, sent out by the UMAs containing the source node, have been acknowledged, the UMA chooses the over-all minimum-cost path.

Set Up of Data Flow

After the path search results in an appropriate path, the UMA sets up new Data Flow Components by configuring their service description according to the result path. The Data Flow Components are started by the local service manager and connected to running tracking services. They form the inner part of a data flow graph, combining measurements provided by tracking services and delivering the inferred spatial data to the application.

2.4.5 Data Flow Components

In the DWARF Ubitrack implementation, all nodes of the data flow graph are realized by distinct DWARF services. The root of this graph consists of the application, represented by a service with a need for `PoseData`.

Leaf Components

At the leaves of the data flow graph are the tracking services which provide raw sensor measurements. Tracking services usually send out a PoseData event as soon as a new measurement is made using the asynchronous push protocol. When a component at a higher level in the data flow graph requires measurements at arbitrary times, a Kalman filter component has to be inserted for interpolation or extrapolation. The leaf components fall into the following categories:

Trackers make real measurements of the relation between two objects. Available tracking services include ART DTrack, Intersense and AR Toolkit. All of them were already available, but we adapted them to send measurements in the new extended Ubitrack *PoseData* format.

StaticCalibration sends out a static measurements at a fixed interval. The value of the measurement is determined by the *Position*, *Orientation*, *Covariance* and *Confidence* attributes. The *UTUpdateRate* attribute specified the interval at which the measurement is sent out. In addition to the asynchronous push protocol, the *StaticCalibration* service also supports synchronous pull, as the static measurement is considered valid at all times.

PoseDataPlayer service inserts previously recorded measurements into the data flow graph. This is useful for offline data analysis or for algorithm evaluation where repeatability is extremely important in order to obtain comparable results. Sequences of measurements can be recorded using the *PoseDataLogger* service.

Interior Components

The interior components in the data flow graph perform all the measurement transformations that are necessary in order to deliver inferred, fused and filtered measurements to the application. Each data flow graph component is realized by a single DWARF service and performs only a small part of the whole computation to allow reuse in more complicated graphs.

The service descriptions are created by the UMA in response to an application query or a change in the sensor topology. Service descriptions act as a configuration for the services and make sure that the services connect to the respective partners in order to form a given data flow graph. Interior services communicate with each other mainly using the `UTPoseDataSyncPull` protocol. All interior data flow graph services run in a single process space for higher system performance.

An in-depth discussion of the interior data flow components is found in chapter 7 of this thesis.

2.4.6 Bootstrapping

Note: *This section was written by Franz Strasser and is a summary of his thesis [**?**].*

As already mentioned one design goal is *flexibility*. That means in our case that sensor hardware should be integrated in flexible way. The big picture is the "on-the-fly integration" of devices and software components because the user of an AR application wants to use his/her own tracking equipment in the Ubitrack environment. So we assume that the user's client system is mobile and that the client does not have any knowledge of its ambient infrastructure: this includes the knowledge of the network infrastructure as well as available Ubitrack systems.

Without loss of generality, the user's set-up should exist of a mobile tracking system, a locatable or both. The example scenario above uses an ARToolkit camera system as a mobile tracker. The environment is the stationary counterpart of the mobile setup, i.e. either a locatable for the mobile tracking system or a stationary tracker for the user's locatable. The local client runs a Ubitrack system which is separated from the stationary one.

Merging of the Spatial Relationship Graphs

Bootstrapping in the sense of Ubitrack means to merge two separated Spatial Relationship Graphs to a larger one. This fusion is realized by finding spatial relations between the two Ubitrack systems and add new edges between formerly separated Spatial Relationship Graphs.

At the beginning the two local Ubitrack systems are separate from each other with their own local Spatial Relationship Graph. When the two UMAs get in connection the information about the hardware devices or locatables are exchanged and the devices are (re-)configured. This exchange of configuration information allows the mobile camera to track the locatables in the environment; or — the other way round — allows the stationary sensors to track the mobile locatables. Currently, there are two different ways how to determine new spatial relationships between seperated Ubitrack systems.

Exchanging Configuration

Every node in the Spatial Relationship Graph, i.e. locatables and sensors, may deliver *hardware descriptions*. They contain information about the used hardware and are implemented as DWARF services. Locatables may contain descriptions of their appearance, e.g. marker patterns. Sensor may deliver settings for the services which compute the measurements, e.g. a camera may deliver its intrinsic parameters. By exchanging these configuration information, the sensors can detect new relationships between objects.

Motion Patterns

If no configuration is available, Ubitrack tries to find these new relationships by observing the motion of objects. Currently, it conducts a frequency analysis of the speed and angular velocity of objects. It compares the frequency spectra by the coherence function and determines a probability value for the similarity of the motion. If a certain threshold is exceeded, the system may assume that a static relationship between two objects exists.

2.5 Related Work

VRPN [**?**] implements an abstraction layer for a wide range of virtual reality devices, including trackers. It provides a network-transparent client-service interface with time stamps and clock synchronization between multiple computers. Supported device types include position and orientation sensors, buttons, dials, analog and force feedback devices. Layered devices can be implemented in order to process data from multiple sensors. VRPN however has no notion of different coordinate systems and transforms nor does it identify multiple trackers and tracked objects.

OpenTracker [**?**] implements a "pipes & filters" dataflow model for processing of tracker measurements. Source and sink objects set up connections to trackers and applications and allow the transport of sensor data over the network. Filter nodes perform static and dynamic coordinate system transformations or smooth measurements in order to remove noise.

Many previous research projects deal with data fusion of different sensors in order to improve tracking stability and accuracy in static sensor setups [**?**, **?**, **?**]. Hoff [**?**] also describes a setup that combines measurements from fixed and user-worn optical trackers. However, no attempt has been made to dynamically integrate arbitrary sensors.

The so far the most comparable results come from Höllerer, Hallaway et al. [**?**, **?**]. They describe a system that combines a high-resolution IS 600 tracker with wide-area infrared beacon observations. When none of these trackers is available, the gaps are bridged by a dead-reckoning technique using inertial orientation sensors, a pedometer and environmental knowledge. A Kalman filter is used to fuse measurements from all tracking systems. However, the system is limited to this particular tracking configuration and a single user as they do not have a software architecture to dynamically integrate arbitrary sensors and all sensors must be calibrated to use the same coordinate system. The group also describes an interesting approach for adapting the user interface of a personal navigation application to the currently available tracking resolution.

3 Errors in Ubiquitous Tracking

The previous chapter has given an introduction to the concepts of ubiquitous tracking and its implementation using the DWARF framework. The purpose of this chapter is to explain what kinds of tracking errors can occur in Ubitrack systems, to describe the goals of this thesis and to outline the general approach how to reach these goals.

3.1 Classification of Error

In his survey [**?**], Azuma distinguishes between static and dynamic registration errors. Static errors cause wrong registration even when no motion occurs in the scene. Dynamic error is caused by caused by delays in the sensor and rendering pipeline which make the underlying measurements obsolete at the time the final augmented image is presented to the user. This thesis deals with how both static and dynamic errors can be reduced in Ubitrack systems.

3.1.1 Static Error

Azuma defines static errors as follows:

> "Static errors are the ones that cause registration errors even when the user's viewpoint and the objects in the environment remain completely still." [**?**]

Static errors can further be classified into static field distortion and random noise [**?**].

Static field distortion causes constant registration errors when even when neither the HMD nor objects are moved. The reasons for static field distortion can be located in the tracking system, in the processing of sensor measurements and in the display system.

Reasons for static field distortions caused by the tracking system depend on the underlying technology that is used. Optical systems usually use lenses that do not provide a perfectly linear projection or the sensor may not be installed exactly orthogonal at the center of the optical axis. The field of a magnetic tracker may be distorted by the presence of other ferromagnetic objects and inertial orientation trackers suffer from drift which causes the registration error to increase at an (ideally) constant speed.

Another source of error comes from incorrect processing of the sensor measurements. Besides programming errors, which may be difficult to find, constant transformations, which often are not exactly known, may be a frequent problem, especially in Ubitrack systems which obtain most of their power from inferring measurements along a path in the Spatial

Relationship Graph. In order to increase the tracking range by inferring from multiple sensor measurements, the exact transformation between the tracker's coordinate systems must be known and errors made here can have big impacts on the registration accuracy.

The third important source of registration error in Augmented Reality applications is caused by the HMD, especially for optical see-through techniques. Therefore the offset between the user's eyes and the HMD's display elements must be known, as errors made here cause the registered image to appear in the wrong size and/or shifted. For stereo images, also the distance between the eyes is important.

All these variations of static field distortion have in common that they are systematic and reproducible and therefore can be eliminated by *calibration*. The general approach for calibration is to compare the values that are produced by a tracker or a combination of multiple trackers with the "true" values determined either manually or by some other high-accuracy tracker of higher accuracy that serves as "ground truth". From the difference between truth and measurements, correction factors can be computed.

For optical trackers, a wealth of literature exists about camera calibration [**?**]. Most approaches use a special pattern of known dimensions which is held in front of the camera. Using image processing techniques on the recorded image, the internal camera parameters can be determined automatically. Magnetic trackers also can be calibrated by using lookup tables that contain correction offsets for a number of points in the tracker's working area [**?**]. Similar techniques can be applied to other tracking technologies.

In Ubitrack systems, HMD calibration cannot be done automatically and always requires user intervention as there is no ways for the computer to know if the real and virtual objects are correctly aligned when seen from the user's eyes. The calibration always requires interactive techniques (e.g. [**?**]) and therefore we cannot reasonably perform the calibration automatically in a Ubitrack system – besides storing the calibration result as an edge between the HMD and the user.

In order to calibrate the transformation between two tracking systems, we can again either measure the distance and orientation of the coordinate systems manually or use a special tracking target that can be tracked by both systems. In the simplest case the transformation between the coordinate system consists of a constant offset and a rotation, e.g. This problem is covered in this thesis as it can be solved with the Ubitrack tools that will be developed. Optical distortion and similar problems require more specialized algorithms and are considered future work.

Random noise or jitter is inevitable in any measurement system[**?**] and determines the resolution that can at best be achieved with a tracker. As the noise is neither systematic nor repeatable, it cannot be removed by calibration. Random noise or jitter is present even when the measured value does not change and often increases with the distance from sensor to target. However, if the statistical error distribution is known or can by considered as Gaussian, it is possible to correct the measurements by using statistical methods, e.g. least-squares or Kalman filters. These noise-reduction techniques however work by averaging multiple measurements and therefore result in loss of response to fast movements. If multiple sensors are available with varying accuracies in different directions, also sensor fusion can be used to reduce the random noise.

3.1.2 Dynamic Error

> "Dynamic errors are the ones that have no effect until either the viewpoint or the objects begin moving." [**?**]

Dynamic error is caused by delays introduced by slow trackers (sensor lag), expensive processing of measurements, network transport and rendering. The sum of all delays in an Augmented Reality system is called the *end-to-end system delay* and describes the time difference between the actual sensor measurement and time the rendered image appears on the display. A measurement is at best valid at the instant the sample is taken by the sensor and the accuracy degrades with time. Presenting the user an overlay image that is based on "old" measurements cause the virtual image to lag behind the reality. In case of HMD tracking it should be noted that the rotation of the user's head may cause very rapid movement of objects in the display, which is in fact the most prominent source of dynamic error in Augmented Reality systems.

It is worth mentioning that dynamic errors mostly play a role in Augmented Reality applications of Ubitrack. In the classic ubiquitous computing scenarios, it is usually sufficient to know the approximate location of objects, often only in a room granularity. Therefore all motions are slow with respect to the required accuracy, and the system can be treated as static.

In his SIGGRAPH course pack, Bishop [**?**] classifies dynamic error (delay-induced error) into four different cases:

First-Order Dynamic Error is the most obvious dynamic error and is directly caused by motion of the objects and the end-to-end delay. The amount of registration error, i.e. the distance between the real and virtual objects, depends on both the speed of motion and the total delay Δt:

$$\varepsilon_{\text{dyn, x}} = \dot{x}\Delta t$$

As the measurements become older, the registration error generally increases linearly with time. The visually observed lag may be acceptable where all motion is restricted to slow speeds. Azuma [**?**] computes a visible error of 60mm for a typical 100ms delay and "moderate" head rotation rate of 50 degrees per second for objects that are about an arm's length away from the observer's head.

The Simultaneity Assumption plays a role when data from multiple sensors is collected sequentially but processed as if it was produced simultaneously. This is particularly often the case in Ubitrack systems, where geometric relations are inferred from measurements made by multiple sensors.

Sensor Sample Rate also is a possible factor for dynamic error as it determines the type of motion that can principally be detected by a sensor. Shannon's sampling theorem states that the sampling rate must be at least twice the bandwidth of the measured signal. Assuming that arm or head motion has a bandwidth of 20Hz [**?**], the measurement rate should be at least 40Hz. Any motion that has frequencies higher than half the sampling rate causes

aliasing, i.e. higher frequencies are shifted into the low frequencies. This phenomenon is probably best known from TV recordings of car wheels which appear to turn at a low rate or even backwards although the car is going at a high speed.

Synchronization Delay occurs when the frequency at which a sensor produces measurements is not exactly synchronized to the frequency at which this data is requested by the receiver. In this case, the measurements have to be held in a buffer for a short amount of time, which is called the synchronization delay. The duration of this delay depends on the update rates and may not be constant, when the sensor update interval is not a divisor of the receivers request interval.

Reduction of dynamic errors can be achieved by multiple techniques. The most important step is to minimize the overall end-to-end system delay by using fast trackers, renderers and network transports. Prediction, i.e. an extrapolation of sensor data into the future using a history of measurements, also has shown to improve the dynamic Augmented Reality experience [**?**]. A classic tool for prediction is the Kalman filter, described in chapter 5.3. In order to allow accurate prediction and to reduce the negative effects of the simultaneity assumption, it is necessary to maintain accurate timestamps (in the order of milliseconds) and to synchronize the clocks of all participating computers.

3.2 The Goals of This Thesis

The overall goal of this thesis is the reduction of both static and dynamic errors. For the static ones, the main focus lies on the treatment of random noise. In fact, the original title of the thesis was "Autocalibration in Ubiquitous Computing Environments", i.e. the elimination of systematic errors. However I soon found out that before this can be done, the underlying problem of noise and dynamic error has to be solved, which provided sufficient work for this thesis.

In order to reduce static and dynamic errors, it is necessary to reach the following subgoals:

3.2.1 Consistent Representation of Measurement Errors

The first goal of this thesis is the development of a model of measurement errors for the Ubitrack framework, which should fulfill the following requirements:

Easy Comprehensibility The error representation should be easily understandable by other developers of DWARF applications.

Efficient Computation The computations related to the error model should be able to run in real time on a standard mobile computer while leaving sufficient resources to other Augmented Reality tasks, like rendering and tracking.

Consistent Propagation When measurements are transformed or chained, the single errors must be consistently propagated to the resulting error distribution. In cases when multiple measurements are chained to infer new relations, errors in position and orientation are each added together. Additionally, the rotational error of one measurement together with the position of the other contributes to the final position error. Similarly, when a pose needs to be inverted, the rotational error adds to the error in position. The error model chosen for the Ubitrack framework must be able to reflect these conditions.

Error Interpretation over Time In addition to a description of the error of a single measurement, instructions of how to interpret data as it becomes older is necessary. For instance, a tracking service that delivers "measurements" of fixed relations, e.g. between rooms in a building, does not need to have a high update rate. In this case, we can assume even relatively old values to be still valid. On the other hand, if a camera on a user's head stops to deliver measurements when a tracked feature moves out of its field of view, we cannot assume the user's orientation to be still valid even after one second.

3.2.2 Measurement Simultaneity in Dynamic Systems

When chaining multiple measurements in order to infer new relations ("following a path" in Ubitrack terms), it is important that all participating measurements were generated at the same time, as otherwise dynamic errors, caused by "trust" in old measurements, may accumulate and cause jerky movements of virtual objects in Augmented Reality applications. In quasi-static cases, where both the viewer and the viewed object are only allowed to move slowly, these effects can be neglected, but for dynamic systems this can be fatal. The general solution is to use interpolation and prediction techniques to compute simultaneous measurements before combination.

The approach described in this thesis uses a single Kalman filter for each sensor to generate measurements at common point in time for all participating sensors. These are then used to compute inferred measurements. When the inferred relation has different motion characteristics than its partial measurements, an additional Kalman filter is used for the output.

An example for the measurement simultaneity problem and static relations can be seen in our demo setup 2.3. When the AR Toolkit marker is fixed to the table, we can measure its pose with respect to the ART tracker by combining the ART-to-Camera and AR- Toolkit-To-Marker measurements. Because the AR Toolkit camera is freely moveable, both of these measurements have dynamic characteristics, while the final ART-To-Marker relation has not. Therefore an additional Kalman filter with a restricted motion model is employed to generate a final recursive least-squares estimation of the marker position.

3.2.3 Dynamic Sensor Fusion

In ubiquitous tracking setups it will frequently be the case that multiple sensors track the same relation. The measurements however can have difference characteristics in accuracy, update rate, lag, etc. or even measure different underlying physical quantities like absolute position and orientation, acceleration or angular velocity. Previous works [**?**, **?**, **?**] have

shown that it is often advantageous to fuse these different sensor readings together in order to generate measurements of higher accuracy or update rate.

One goal of this thesis is the development of a software architecture that allows the dynamic construction of sensor fusion filters to take advantage of multiple redundant measurements in complex Ubitrack setups.

3.2.4 Prediction

Previous research [**?**] has shown that dynamic registration errors in Augmented Reality application, i.e. errors occurring when the user quickly moves his head, can be reduced by using prediction techniques. The usual approach is to compute first or second-order derivations of position and orientation and use these to extrapolate the current pose into the future.

The prediction goal is tightly coupled to the measurement simultaneity problem mentioned before as guaranteeing the simultaneity of measurements frequently requires an extrapolation of past sensor data.

3.2.5 Estimating Static Relationships

A special case of relationship between objects appears when two objects do not move relative to each other. In Augmented Reality application this frequently is the case, e.g. when two tracking cameras are installed in a room or when different tracking targets are attached to an object. When enough redundant tracking measurements are available it is possible to estimate this static relationship instead of having to measure it by hand.

This goal is closely related to the error model as in the case of static relationships all measurements can be considered as the truth overlaid with noise. The solution to this problem essentially is again the use of a Kalman filter together with a special motion model that makes the Kalman filter an ordinary recursive least-squares estimator.

3.3 Overview of Related Work

A lot of work in these areas has already been done, especially on the prediction and sensor fusion topics. However, to my knowledge, nobody has so far combined all these aspects into a single system that allows the dynamic integration of sensors at runtime. The purpose of this section is to give the reader a short overview of the most relevant related works from the field of Augmented Reality. More detailed analyses of similarities and differences will be discussed where appropriate in later chapters.

3.3.1 Azuma

In his 1995 Ph.D. thesis "Predictive Tracking for Augmented Reality" [**?**], Ronald Azuma describes a technique to improve dynamic registration in cases when the user quickly moves

his HMD. His approach uses an extended Kalman filter to fuse data from vision, accelerometers and angular velocity sensors in order to determine the HMD's position, velocity, acceleration, orientation and angular velocity. For eliminating the end-to-end delay introduced by slow sensor measurements and the rendering process, these parameters are used to predict a future position and orientation.

In contrast to my work, Azuma does not treat error covariances as actual measurement errors, but rather as ordinary parameters of the Kalman filter that can be tuned in order to yield the desired behavior of the prediction system. Also, a fixed sensor setup is used, while this work aims at allowing the dynamic integration of all available sensors.

3.3.2 Hoff

In [**?**] William Hoff describes a setup to combine the measurements from an "inside-looking-out" camera mounted on the user's head with measurements of an external "outside-looking-in" optical tracking system tracking both the HMD and the target object positions. While the external tracker has a higher over-all resolution, the head-mounted camera has the advantage of always looking in the same direction as the user, resulting in insensitivity to the user's orientation and a higher resolution perpendicular to the viewing direction.

The paper also describes a visualization of tracking errors and derives formulas for computing the error distributions that result when multiple coordinate system transforms are combined. Hoff however only considers quasi-static registration "where objects are stationary when viewed, but can freely be moved."

3.3.3 SCAAT

The 1996 Ph.D. thesis "SCAAT: Incremental Tracking with Incomplete Information" [**?**] by Gregory Welch describes a mathematical method of integrating incomplete measurements of the user's pose into a Kalman filter, allowing for sensor fusion, prediction and even simultaneous autocalibration. An application of the SCAAT algorithm to the UNC HiBall tracking system is presented where single 2D camera measurements are integrated into a full 6DOF pose estimate.

The SCAAT approach is in many ways similar to the one presented in this thesis. SCAAT however does focus on more general mathematical methods, while I will also try to combine this with software aspects of dynamic Kalman filter construction.

3.3.4 Höllerer, Hallaway

In [**?**] Hallaway, Höllerer and Feiner describe a mobile Augmented Reality system that is able to move between the working areas of trackers of dramatically different accuracy. These include an IS 600 magnetic+inertial tracker, a GPS, infrared beacon sightings and – for "bridging the gaps" in between – a dead reckoning module, consisting of a pedometer, orientation sensors and some environmental knowledge which prevents the system from assuming that the uses passes a wall. All available measurements are fused together into a Kalman filter. However, in contrast to the Ubitrack approach, this system does not dynamically integrate

sensors at runtime, nor does it handle transformations between different coordinate frames systematically.

4 An Overview of Tracking Technologies

The goal of Augmented Reality is the perfect alignment of virtual and real objects. For this process, usually called *registration*, the precise position and orientation of the user's head must be known at any time. This chapter gives an overview of commonly used tracking technologies and explains their advantages and drawbacks, as well as the dominating sources of error. The goal of this chapter is an analysis of existing tracking technologies, which will help in the following chapters, where a unified description of tracker measurements and an error model is developed.

4.1 Evaluation Criteria

The following criteria are commonly used for evaluating different tracking technologies in Augmented Reality systems:

Resolution is the smallest difference in position or orientation that can be measured by the tracker.

Accuracy usually is lower than a tracker's resolution. It means the expected absolute error in measurements of a tracker which can be quantified by covariance or a Root-Mean-Square (RMS) value. Accuracy is not necessarily homogenous throughout the entire working area of a tracker.

Latency is the amount of time between the incidence of a measured physical fact and the moment that the measurement result is made available to the rest of the system.

Update rate describes how many measurements a tracker can make in a given time.

Working area is the space where a tracker is able to make measurements. In an operation room AR scenario a tracker only needs to track the patient and the surgeon's HMD in a small area around the operating table. However, architecture or factory maintenance applications often require users to tracked inside or around large buildings. Sometimes the working area also is restricted by requiring the tracked object to be within the line-of-sight of one or more sensors.

Robustness against external influences which range from temperature, moisture or bright sunlight in outside applications to electric and magnetic fields from engines in factories or CRT screens in office environments. Also, if a tracker has insufficient mechanical stability, it may have to be recalibrated frequently.

Obtrusiveness of a tracking system can be described by the size and weight of the tracker and the targets, but also by the difficulty of installing or the visual appearance of tracked features.

Monetary cost always is a relevant criterion when deciding which technology to use.

In practice there exists no ideal tracking system and tradeoffs between these factors have to be made. The following sections give an overview of many popular tracking technologies and describe their advantages and drawbacks. Other overviews can be found in [**?**, **?**, **?**, **?**]. There are many possible distinctions of tracking systems. The approach here lists trackers by the underlying physical medium that is used for measurements.

4.2 Optical Trackers

Optical trackers usually use image processing techniques to locate features in images recorded by an optical camera. Other systems also use lateral effect photo diodes [**?**, **?**] or so-called quad-cells in order to directly determine the location of light-emitting diodes (LEDs) without expensive image recognition.

When three features with known positions on an object are detected by a single camera, it is possible to determine the object's position and orientation relative to the camera using a closed-form solution. This 6-degree-of-freedom measurement from only three points is possible as each point effectively provides two measurements in x and y direction. For increased stability, most optical trackers however track more than three points and use a least-squares-algorithm to solve this now over determined system. Standard least-squares approaches are rather sensitive to outliers which frequently occur in feature detection, and therefore a robust estimation technique such as RANSAC [**?**] can be used.

Many research tracking systems use high-contrast black and white [**?**, **?**, **?**, **?**] or colored [**?**, **?**, **?**] artificial targets placed at known locations in order to determine the position of the camera. A promising approach is the detection of natural features that allow optical trackers to work in unprepared environments [**?**].

Alternatively, active infrared LEDs that transmit an ID by blinking with a unique pattern are used in the HiBall [**?**] and Northern Digital Optotrack systems. The ART DTrack system uses passive retro-reflective targets which are illuminated by infrared flashes integrated into the cameras. Infrared targets generally have the advantage that they can easily be detected using optical filters which block all visible light.

Optical systems provide good resolution in the x and y directions orthogonal to the camera's viewing direction. Using sophisticated models of camera sensor elements, features can be detected with an accuracy of up to $\frac{1}{20}$th pixel. In order to determine orientation or location in z (the viewing) direction, the relative positions of at least three features have to be utilized. In this computation, feature detection errors usually play a bigger role. Therefore accuracy of z-position and orientation is generally lower. As a target moves further away from the camera, the distance between its tracked features becomes smaller while the feature detection error stays constant. A solution to this problem is the use of multiple cameras which track the same target simultaneously.

Another source of errors in optical trackers are optical distortions. They can be caused by a wrong focal distance, or a sensor that is not exactly mounted at the center of the and orthogonal to the optical axis of the lens. Also, wide-angle lenses frequently cause radial distortions that increase with the distance to the center of the image. These distortions however

are systematic and can be compensated by careful camera calibration. A wealth of literature exists about this topic with the classic algorithm being described in [**?**]. All serious optical tracking system require calibration, either done by the manufacturer (e.g. ART) or by the user (ARTk)

Inside-Out vs. Outside-In Tracking is a popular distinction of tracking systems used in Augmented Reality applications, which mainly applies to optical tracking. Inside-Out tracking refers to systems where the tracker is mounted on the user's head and makes measurements in almost the same coordinate system that is used to display information in the HMD. The higher errors in orientation and z direction of virtual objects do not have as big an effect on the user's experience as the x and y positions.

An Outside-In tracker uses camera's installed at fixed locations that track moving targets. For displaying virtual objects in the user's HMD, the HMD orientation must be determined with high accuracy from multiple tracked features. As the orientation is derived from position measurements, the accuracy in orientation is lower than that of the HMD position. This drawback of external tracking systems can partly be compensated by using cameras that are heavier and provide higher quality than those than can be carried by the user.

Advantages: When mounted on the user's head, optical trackers provide excellent resolution in the user's viewing direction and working range is theoretically only limited by the number of fiducials that the user can place or the number of natural features that can be stored and recognized.

Drawbacks: Optical trackers have relatively high latencies, which is caused by the large amounts of data that has to be transported from the camera into the computer's memory and by the time required for expensive image processing algorithms. Also, most trackers require good lighting and the placement of artificial fiducials in the scene. Optical trackers always require a clear sight between camera and feature.

Examples of commercially available optical trackers are the ART DTrack system, Northern Digital's Optotrack and 3rdTech's HiBall tracker. Many research systems [**?**, **?**] are based on the freely available AR Toolkit [**?**]. A wider overview of commercial optical tracking system systems and freely available tools can be found in [**?**].

4.3 Inertial

Inertial trackers are based on elementary physical laws of inertia which state that a force must be exerted on a body in order to change its velocity or direction of motion. Therefore they do not need any special tracking infrastructure in the environment as all measurements are made in relation to the world. This section gives a short description of accelerometers used for measuring linear accelerations and gyroscopes that measure angular velocities. More detailed overviews of the underlying principles are given in [**?**, **?**].

Accelerometers can measure accelerations that act upon a body by gravity, changes in velocity or rotation, if the accelerometer is not mounted at the center of rotation. Based on Newton's law $F = ma$, accelerometers do not measure acceleration directly, but the force that is exerted on a known reference mass. When the reference body is attached to a spring, a displacement between the housing and the mass can be observed that is proportional to the acceleration. This displacement can be measured by a potentiometer that changes its resistance as the wiper is moved. Analog Devices' ADXL202 small-size two-axis acceleration sensor uses capacitor plates mounted on both the sensor housing and a moveable silicone body. The capacity increases as the distance between both plates is reduced. Another possibility for the construction of on-chip accelerometers is the use of piezoelectric sensors that create an electric charge when under mechanical pressure.

The main problem when using accelerometers for tracking is that the measured accelerations have to be integrated twice in order to obtain a relative position measurement and therefore even small measurement errors accumulate quickly to big errors in position. This can be observed as drift which cause the measured position to move away from an the real location, even when the sensor is not moved. In addition to true accelerations, accelerometers also measure static forces such as gravity or centrifugal forces which have to be subtracted before integration.

Gyroscopes can be used to measure angular velocities and are based on the principle of conservation of angular momentum. Gyroscopes in navigation use a heavy spinning mass which, when torque is applied, starts rotating around the axis orthogonal to its rotation axis and the axis of torque, a phenomenon which is called precession. When the motion of the rotation axis is constrained by springs, the amount of torque can by determined by measuring the displacement of the axis' anchor.

An alternative used in modern avionics are laser gyroscopes. Light from a laser is split into two rays that travel in different directions around the axis of rotation in an optical fiber. At the end, the two rays are combined again, causing interference. As the whole device rotates, the ray traveling in the direction of the rotation takes a shorter path, causing a phase shift which results in a change of intensity of the combined signal that can be measured.

However both mechanical and laser gyroscopes are too big and expensive for augmented reality applications. Small and light sensors that can be integrated into silicone chips exploit the coriolis effect. Kuchling [**?**] defines the coriolis force as follows: "When a body is moving in a rotating reference system radially to the inside or outside, its track speed is changing. It is therefore experiencing tangential acceleration caused by the coriolis force." The coriolis effect is also responsible for weather phenomenons as the rotation of storms. In integrated devices, the required motion is usually generated by oscillating a mass, e.g. a silicone tuning fork. When the system is rotated around an axis perpendicular to the direction of oscillation, the oscillating mass picks up an acceleration orthogonal to the rotation axis and the direction of oscillation. This acceleration depends on the angular velocity of the system, but not on the sensor's position with respect to the center of rotation.

As gyroscopes only measure the angular velocity, the signal has to be integrated from a known position in order to determine the absolute orientation. Therefore gyroscopic orientation sensors also suffer from drift, but because the measurements are only integrated once, the result is substantially better than what can be gained from accelerometers.

Advantages: Inertial sensors can be used anywhere as they are self-contained and therefore no special infrastructure is necessary. They offer good resolution at a low latency and are relatively robust against external influences. Highly integrated silicone sensors are small, light-weight and inexpensive.

Drawbacks: All inertial sensors only provide relative position and orientation measurements. Integration from a known initial location is necessary in order to determine absolute values, in which even small measurement errors accumulate and cause drift. Therefore inertial trackers can only be in addition to absolute position/orientation sensors.

Examples of commercial inertial tracking systems are Ascension's 3D Bird and Intersense's IS-300/600 and InterTrax trackers.

4.4 Acoustic

Acoustic trackers measure the time-of-flight of an acoustic signal (usually ultrasound at about 40kHz) from a transmitter to a receiver and compute the traveled distance by multiplying the time by the speed of sound. This restricts the location of the receiver to lie on a sphere around the transmitter. By using tree transmitters or receivers, the location can be restricted to two points, one of which can usually be rejected due to known constraints, e.g. when the transmitters/receivers are mounted on the ceiling. In order to compute a three-dimensional position, either one receiver and three transmitters or one transmitter and three receivers can be used, but the components with three instances must be located at fixed locations in the reference coordinate system. For full 6 degrees-of-freedom pose measurements, three receivers and three transmitters are necessary. More details about acoustic tracking can be found in [**?**].

Advantages: Acoustic systems can be relatively small and light-weight while providing low-latency measurements at a good update rate. The necessary hardware components (ultrasound senders and receivers) are relatively inexpensive.

Drawbacks: Ultrasound trackers are restricted to working areas where transmitters or receivers are installed and they require a clear line-of-sight between them. Accuracy is in the area of a few millimeters to centimeters, but tracking quality can be degraded by reflections of sound waves from walls or other objects which cause the signal to arrive at the receiver in multiple copies. Also, the speed of sound depends on temperature and air pressure and cannot be treated as constant.

Examples of existing acoustic trackers are the Intersense IS-600/900 systems (sensing orientation with an inertial system) or the AT&T Active Bat system [**?**], which uses ceiling-mounted receivers and a user-worn sender that can be activated on demand by a RF signal in order to determine the location of people in a ubiquitous computing application. [**?**]

demonstrates an inside-out acoustic system that is attached to the HMD and tracks targets that the user carries in his hand.

4.5 Magnetic

Magnetic trackers use a sender with three orthogonal coils that are activated sequentially with dc or low frequency ac currents in order to generate magnetic fields in the working area. The receivers also have orthogonal triaxial coils in which a current is induced by magnetic field of the sender. The orthogonal arrangement of the coils allow the receivers to determine both strength and orientation of each of the field from the sender. Unlike the acoustic case, the strength of the magnetic field does not directly indicate the distance to the receiver, as the streamlines do not have a spherical, but rather odd shape. Three measurements from each of the three source fields allow the receiver to determine a full 6 degrees-of-freedom pose. [**?**] gives a more detailed description of the underlying principles.

As magnetic fields can pass through most materials, the receivers do not need to have a clear line-of-sight to the sender. The field can, however, be distorted by the presence of any ferromagnetic object near the tracker or by artificial magnetic fields e.g. from engines or CRT screens.

A special case of a magnetic tracker is a compass that does not generate its own field, but instead relies on the magnetic field of the earth. This allows the tracker to determine the angle to the flux lines of the earth's magnetic field, which pass horizontally from the south to the north. A compass is often used together with GPS in simple outside navigation scenarios. The problem with this approach is that the earth's magnetic field is relatively weak and can be distorted even by slightly ferromagnetic objects such as buildings and rocks and therefore often yields unusable results in urban environments.

Advantages: Magnetic trackers have a high update rate, low latency and do not require a clear line-of-sight between sender and receiver.

Drawbacks: Similarly to acoustic trackers, they need a special infrastructure for tracking. Ferromagnetic objects or artificial magnetic fields from engines or CRT screens may seriously distort measurements.

Examples of magnetic tracking devices are Polhemus' FASTRAK, Ascension's Flock of Birds and Northern Digital's Aurora. Virtual Realities' VirtuaTrack does not generate its own magnetic field, but uses the ambient field instead for orientation tracking.

4.6 Mechanical

Mechanical trackers usually consist of a number of angle and length sensors that are more or less rigidly attached to the objects whose relation is to be measured. Although a mechanical tracker was used in the pioneering HMD VR work by Sutherland [**?**] in 1968, the have most

heavily been used for motion capture [**?**] where trackers range from whole body suits with heavy exo-skeletons that measure the motions of all relevant body parts of an actor to gloves that track the movements of all fingers. Angles and lengths can be measured by potentiometers or optical sensors based on light-barriers. Newer products also use strain gages that change their resistance when bent.

Advantages: Mechanical trackers give good accuracy, low latency, high update rates and can also be combined with force feedback.

Drawbacks: They often are relatively bulky constructions that may seriously constrain the freedom of the user.

4.7 GPS

The Global Positioning System (GPS) provides geographic location information from time-of-flight measurements of electromagnetic signals sent by specially designed satellites. The system consists of a control segment, a space segment and a user segment [**?**, **?**]

The space segment is a network of 24 satellites (21 active + 3 spare) flying in high orbit (about 19.000km) above the earth. The orbits are arranged such that every place on earth with a clear view of the sky can always receive the signals of at least four satellites. Every satellite contains multiple atomic clocks that allows it to stay synchronous with the global GPS time. The space segment is monitored by the control segment, a network of 5 ground stations that compute precise orbital data and clock corrections for each satellite. Each satellite transmits the L1 signal at 1575MHz which is modulated by the 1MHz C/A-Code (Coarse Acquisition), the 10MHz P-Code (Precise) and a 50Hz navigation message which contains the current position of the satellite and a correction factor for satellite's internal clock. The C/A-Code is a pseudo-random sequence which repeats every 1023 bits (about one millisecond). The P-Code is a very long (seven days) pseudo-random sequence that usually is encrypted to limit its access to specially-equipped authorized users.

GPS receivers (the user segment) internally compute the same pseudo-random sequence and correlate it with the received signal in order to measure the time shift between the internal clock and the satellite's signal. By multiplying the delay by the speed of light, the distance to the satellite can be computed. This constrains the receiver's position to intersecting spheres around the known position of three satellites, using the measured distance as the radius. However, as receivers typically only use a cheap oscillators which are not exactly synchronous to the global GPS clock, the signal from a fourth satellite is needed in order to compute both the three-dimensional position and the global GPS time simultaneously. Specially equipped users can utilize the encrypted P-Code in order to determine a more precise correlation and additionally use the signal on the L2 frequency (1227.60 MHz) in order to compensate for ionospheric delay.

The C/A-Code useable by civilian receivers provides an accuracy of about 100 meters horizontally and 156 meters vertically at a correctness probability of 95%. In order to increase accuracy, the differential GPS (DGPS) technique can be used. This requires the setup of a

reference receiver at a fixed known location that transmits RF correction signals for each satellite to surrounding receivers. Using DGPS it is possible to achieve accuracies of 1 to 10 meters, depending on the distance between mobile and reference receiver. Because of this low precision, GPS in Augmented Reality is usually used outside for navigation or to augment relatively distant objects.

Advantages: The Global Positioning System is available everywhere on the planet without the need to install a special infrastructure – unless DGPS is used.

Drawbacks: GPS provides a very coarse resolution and low update rates. In order to make measurements, it is necessary to maintain a clear line-of-sight to at least four satellites. Therefore GPS generally can only be used outside and away from high buildings (urban canyons).

4.8 Other Techniques for Location Estimation

Most ubiquitous computing applications use substantially different techniques for estimating the location of the users. Unlike Augmented Reality, where the precise location and orientation must always be known, location mostly is used to trigger events and change the user's context. Therefore the focus is on small, cheap sensors that can be installed in large (usually indoors) environments. Because of their low precision and update rate, these sensor reading cannot directly be used in Augmented Reality applications. However, as Ubitrack aims at providing a unified location sensing framework for both Augmented Reality and ubicomp, it is still worth looking at typical ubicomp techniques. A more comprehensive overview of ubicomp location system can be found in [**?**].

Dead Reckoning Dead reckoning had been used in navigation before celestial techniques came available at the end of the 15th century. It was also used by Columbus when he discovered the new world, and his logs still cause dispute over where the first landfall was made. The principle is simple: When starting from a known location, it is possible to determine the current location if the direction and the covered distance is known. In classical sea navigation, the direction is determined using a compass and the traveled distance by measuring the speed and multiplying it by the elapsed time. Every time the course is changed, the initial location is set back to the currently assumed coordinates. The problem with this technique is that errors in both direction and speed accumulate and the accuracy of the estimated location degrades with time.

Dead reckoning is also used in car navigation systems (e.g. Alpine Nav-200) in order to support GPS in situations where the satellite view is blocked by buildings, rocks or tunnels. The direction usually comes from a compass and/or gyroscopes and the speed is taken from sensors at the wheels. Odometry-based dead reckoning using wheel encoder also is the most popular navigation technique in mobile robotics [**?**].

In order to apply dead reckoning to personal navigation, usually pedometers are used to count the user's steps and get an estimate of the step size [**?**, **?**]. Pedometers consist of one or more accelerometers which are attached to the foot or hip of the walking person where each

step creates a characteristic acceleration pattern. For determination of walking direction, a compass and/or gyroscopes can be used. In order to improve location estimation, environmental knowledge about walls, furniture, etc. can be used to exclude impossible paths [**?**].

ID Tags Another technique mainly used in ubiquitous computing does not give precise location information, but determines proximity of a sender and a receiver. When the receiver is within range of the sender's signal and one of the two is at a known position, the location of the other can be restricted to lie in a certain area. An example of this are RFID tags [**?**] which send a low-power radio signal to a closely located receiver. The tags can be powered by a battery (active tags) or by induction of a magnetic field from the receiver (passive tags). RFID tags are inexpensive and have recently become popular for contactless object identification and security applications. The Active Badge [**?**] system consists of user-worn badges that transmit a unique infrared ID signal every 10 seconds which is detected by network of receivers at fixed locations. [**?**] uses strong infrared emitters whose ID signal is received by ordinary Palm Pilot PDAs.

Wireless Networks In order to provide mobile telecommunication services and wireless network access, a lot of infrastructure is already installed which can be exploited for location estimation. For position in the large scale, cell phone-networks like GSM or UMTS are suitable, and in office environments wireless LANs can be used. The infrastructure consists of base stations at fixed locations which can be uniquely identified. Simple cell-based approaches work similar to ID tags. By receiving the signal of a particular base station with known location, the receiving mobile device can be assumed to be near the base station. More sophisticated approaches determine the total signal strength or the signal-to-noise-ratio of multiple base stations. As the RF signal is severely weakened by walls and other objects, it is not possible to determine the distance to the sender and compute the location as an intersection of multiple spheres as in the acoustic or GPS cases. Microsoft research's RADAR [**?**] system uses floor maps and a signal propagation model to solve this problem. Other projects [**?**, **?**] use learning approaches. These can be divided into two phases: In the offline training phase the signal strength of base stations is recorded at selected locations. In the online location determination phase the recorded data is searched for the location that matches best the currently measured signal strengths. Wireless network trackers can cover a wide area and in many cases it is not necessary to set up a special infrastructure.

4.9 Hybrid Tracking

The previous sections have shown that there is no ideal tracker that provides low latency, high update rate, unlimited range and good precision at a low price. Therefore one usually has to decide which tracking technology fulfils best the requirements of a particular application. However, it is possible to combine multiple tracking technologies to a hybrid tracker such that the advantage of one technology can cover the drawbacks of another. A classic head tracking example is the combination of inertial sensing with optical tracking in order to get the high-resolution absolute positioning from the vision system and the high update

rate with a low lag from the inertial. The inertial system can also fill short tracking gaps when optical tracker does not work, e.g. when no fiducials are visible.

Ubitrack systems also are inherently hybrid as new sensors can be dynamically integrated. In this section I distinct the hybrid approaches by the level of integration between the different tracking techniques.

Tightly Integrated Systems can improve the measurements of one sensor by integrating measurements from another. An example is a vision system that gets updates from an inertial angular velocity sensor [**?**, **?**, **?**]. The vision tracker can use this information to predict the new position of a previously detected fiducial and limit the search area in the image, which results in better tracking performance and higher robustness against wrongly detected features. The systems described in [**?**, **?**] describe integrate vision and magnetic tracking. [**?**] uses GPS coordinates to compute the horizon silhouette from a digital elevation model of the landscape and matches is against a recorded image in order to obtain the user's orientation.

Sensor Fusion Systems combine measurements from multiple independent trackers in order to obtain a better location estimate. In contrast to tightly integrated systems, this fusion is done after the sensor measurements, and the single trackers do not profit from each other. The classic tool for sensor fusion is the Kalman filter which is described later in chapter 5.3. Examples of fusion of vision and inertial sensors can be found in [**?**, **?**, **?**]. [**?**] combines head-mounted and fixed optical sensors in order to reduce the general optical weaknesses along the line of sight and in rotation. Inclinometers (static acceleration sensors that measure the direction of gravity) can be fused with a compass and gyroscopes to combat the drift of the integration [**?**, **?**]. [**?**] combines position and orientation measurements from three ultrasonic "Bats" [**?**] with rotational information from an inertial tracker. The commercial IS-900 product from Intersense also uses inertial position and orientation measurements together with acoustic sensors.

Alternative Tracking Systems switch between multiple tracking technologies or use them for different purposes, but do not try to combine measurements from multiple sensors in order to get a better location estimate. I particularly also consider systems that use different sources for position and orientation to be in this group. [**?**] uses a compass and inclinometers for orientation and GPS for position. [**?**] also describes an outdoor system using GPS for position determination and a sensor-fusion system for orientation measurements. Another common application is the use of GPS for outdoor tracking and a vision system indoors [**?**]. A later approach by the same group [**?**] uses the optical system only for position determination inside buildings and takes the orientation completely from an IS-300 inertial system.

5 Mathematical Basics

For being able to describe measurements and errors in ubiquitous tracking systems, a few basic mathematical methods are necessary. Representation of orientation, multidimensional statistics and optimal estimation are or particular importance for this thesis and therefore explained in this chapter. These mathematical basics will be used in the next chapter to develop an error model for ubiquitous tracking system.

5.1 Representing Orientation

Orientation is an important part of tracker measurements, but there exist many different ways to represent orientation. The orientation of a body is usually described by a rotation that aligns the axes of the reference coordinate system with those of the body. Theoretically, a rotation in 3D-space can be described using only three scalars. However, it can be proven that it is impossible to map $\mathbb{R}^3$ onto SO(3), the group of rotation matrices of $\mathbb{R}^3$, without singularities.

The ideal representation has as few elements as possible and should exhibit no singularities or ambiguities that can become problematic in certain computations. Additionally, it should have an approximately linear behavior for all values to allow the use of numerical algorithms based on derivations. In Ubitrack frameworks, chaining of coordinate system transforms along a path (inference) is an important operation. Therefore the rotation representation should have efficient product and inversion operations. In order to compute measurements at arbitrary times in a system with unsynchronized sensors, also a simple interpolation of rotation sequences is needed. Finally, as the rotation describes a coordinate system transform, it should be possible to rotate vectors without having to change the representation.

5.1.1 Matrices

The matrix representation is the most commonly used in computer graphics. A 3×3 matrix can be used to represent rotation and scaling operations. By extending the coordinate system to four-dimensional homogeneous coordinates where vectors representing positions have a 1 in the fourth entry. A 4×4 homogeneous matrix can be used to represent translation, rotation, scaling and even perspective projection, which are the most important vector operations in a rendering pipeline. All the elementary operations can be combined by matrix multiplication, allowing the efficient traversal of a whole scene graph using a matrix stack that combines all operation from the root to the leaves. A useful property of the matrix representation is that the columns can be interpreted as the direction of the transformed coordinate axes in the source coordinate system.

Advantages: A 3×3 rotation matrix (or the upper left submatrix of a 4×4 homogeneous matrix) always has a determinant of 1 and is orthogonal, which allows efficient inversion by matrix transposition. The product of multiple rotations can easily be computed by matrix multiplication, and the rotation of a vector simply is a matrix-vector product. The matrix representation is free from singularities and unambiguous as each orientation can only be represented by a single rotation matrix.

Drawbacks: By having 9 elements, rotation matrices are a relatively heavy representation and the product is computationally expensive (27 multiplications). Also the direct interpolation is not possible as addition or subtraction of matrices almost certainly results in a non-rotation matrix

5.1.2 Euler Angles

Euler angles are the most simple and probably also most intuitive representation of orientation. A rotation in 3D can be expressed as three successive rotations around different coordinate axes. A particular instance of Euler angles is the *Yaw-Pitch-Roll* representation. The three rotations can be represented by the following rotation matrices [**?**]:

$$\text{Rotation } \psi \text{ about } z \text{ axis (Roll)}, \quad R_1 = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Rotation } \theta \text{ about } y \text{ axis (Yaw)}, \quad R_2 = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$\text{Rotation } \phi \text{ about } x \text{ axis (Pitch)}, \quad R_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}$$

The product of the three matrices which can express arbitrary rotations in 3D-space can be written as:

$$\begin{aligned} R &= R_3 R_2 R_1 \\ R &= \begin{bmatrix} \cos\psi\cos\theta & \sin\psi\cos\theta & -\sin\theta \\ \sin\phi\cos\psi\sin\theta - \cos\phi\sin\psi & \cos\phi\cos\psi + \sin\phi\sin\psi\sin\theta & \sin\phi\cos\theta \\ \cos\phi\cos\psi\sin\theta + \sin\phi\sin\psi & \cos\phi\sin\psi\sin\theta - \sin\phi\cos\psi & \cos\phi\cos\theta \end{bmatrix} \end{aligned}$$

As all angles wrap around at 2π, ϕ and ψ usually are restricted to the interval $]-\pi;\pi]$ and θ to $]-\pi/2;\pi/2]$. A particular problem with Euler angle representations occurs when $\theta = \pi/2$. In this cases any value of ϕ can be compensated by choosing a different ψ and the representation of rotation is effectively reduced to the two dimensions θ and $(\phi - \psi)$. This phenomenon is often called *Gimbal Lock* .

Advantages: Euler angles provide a simple and easy to understand representation of orientation with a minimal number of elements. Unlike matrices, every Euler angle combination does represent a rotation and we cannot fall out of the space of rotations.

Drawbacks: Angles cannot be represented unambiguously using Euler angles and gimbal lock may cause problems in certain computations (e.g. interpolation) and require explicit treatment by the programmer. Computing the product of multiple rotations or applying the rotation to a vector is not directly possible and interpolation of Euler angles does not result in nice looking rotation as each of the angles is treated independently.

5.1.3 Quaternions

Quaternions were introduced into the domain of computer graphics by Ken Shoemake [**?**] and have been used extensively in the field of animation and kinematics.

This section gives a rather informal introduction in quaternion representation and some elementary operations. Formal proofs for can be found in [**?**, **?**]. Quaternions historically were developed as a generalization of complex numbers in three dimensions. They consist of a scalar (real) and a vector (imaginary) part and can be written in several ways:

$$\begin{aligned} \mathbf{q} &\equiv \mathbf{i}x + \mathbf{j}y + \mathbf{k}z + w \quad , \quad w, x, y, z \in \mathbb{R} \\ &\equiv [x, y, z, w] \quad , \quad w, x, y, z \in \mathbb{R} \\ &\equiv [\mathbf{v}, s] \quad , \quad s \in \mathbb{R}, \mathbf{v} \in \mathbb{R}^3 \end{aligned}$$

Basic Quaternion Algebra

For addition and subtraction, quaternions can be treated like ordinary vectors:

$$\mathbf{q}_1 + \mathbf{q}_2 = [x_1 + x_2, y_1 + y_2, z_1 + z_2, w_1 + w_2] = [\mathbf{v}_1 + \mathbf{v}_2, s_1 + s_2] \tag{5.1}$$

Multiplication can be derived from the **i-j-k**-notation by using the rules $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1, \mathbf{ij} = -\mathbf{ji} = k, \mathbf{jk} = -\mathbf{kj} = \mathbf{i}$ and $\mathbf{ki} = -\mathbf{ik} = j$, which were defined by Hamilton, the inventor of quaternions. In the scalar-vector form this can be written as

$$\mathbf{q}_1 \otimes \mathbf{q}_2 = [\mathbf{v}_1 \times \mathbf{v}_2 + s_1\mathbf{v}_2 + s_2\mathbf{v}_1, s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2] \tag{5.2}$$

where $\cdot$ and $\times$ denote the standard scalar and vector products in $\mathbb{R}^3$. Like matrix multiplication, the quaternion product is associative, distributes across addition, but is generally not commutative.

Similar to complex numbers, quaternions also can be conjugated. This is defined by negating the vector part:

$$\mathbf{q}^* = [-x, -y, -z, w] = [-\mathbf{v}, s] \tag{5.3}$$

For multiplication of conjugated quaternions, the following formula applies:

$$(\mathbf{p} \otimes \mathbf{q})^* = \mathbf{q}^* \otimes \mathbf{p}^* \tag{5.4}$$

Also of interest is the quaternion norm, $N(\mathbf{q})$, which is a scalar quaternion (having a vector part of $\mathbf{0}$), and is defined identically to the Euclidean norm of a general vector:

$$N(\mathbf{q}) = \|\mathbf{q}\| = \sqrt{x^2 + y^2 + z^2 + w^2} = \sqrt{\mathbf{q} \otimes \mathbf{q}^*} = \sqrt{\mathbf{q}^* \otimes \mathbf{q}} \tag{5.5}$$

Using conjugation and norm, the inverse of a quaternion can be computed as

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{N(\mathbf{q})^2} \tag{5.6}$$

Quaternions of norm $N(\mathbf{q}) = 1$ are normalized and called *unit quaternions*. In this case, the inverse $\mathbf{q}^{-1}$ of $\mathbf{q}$ is $\mathbf{q}^*$

Representing Rotation with Quaternions

In order to describe rotation, only unit quaternions ($\|\mathbf{q}\| = 1$) are used, which means that all valid rotation quaternions lie on a four-dimensional hyper sphere. If we express a rotation by an angle θ and an axis $\mathbf{v}$ where $\mathbf{v}$ is a 3D unit vector, then the corresponding rotation quaternion can be constructed as

$$\mathbf{q}_{\theta,v} = [\sin(\frac{1}{2}\theta)\mathbf{v}, \cos(\frac{1}{2}\theta)] \tag{5.7}$$

This gives quaternions the intuitive meaning that the vector part always points in the direction of the rotation axis and the scalar is the cosine of $\frac{1}{2}\theta$. The identity rotation with $\theta = 0$ is represented by the quaternion $[0, 0, 0, 1]$, a 90 degree rotation about the x axis by $[\sqrt{\frac{1}{2}}, 0, 0, \sqrt{\frac{1}{2}}]$ and so on. Now also the conjugation which inverts the rotation has an intuitive meaning in that it simply changes the direction of the rotation axis. As with matrices, multiple consecutive rotations can be expressed by the quaternion product.

Application of rotations to vectors also is easy with quaternions. By converting the vector $\mathbf{x}$ to a vector quaternion $\tilde{\mathbf{x}}$ where the scalar part is 0, we can express the rotated vector as

$$\tilde{\mathbf{x}}' = \mathbf{q} \otimes \tilde{\mathbf{x}} \otimes \mathbf{q}^* \tag{5.8}$$

It can be proven that this always results in another vector quaternion.

Unfortunately the quaternion representation is not completely unambiguous, as each rotation can be expressed by two different quaternions. In other words, $\mathbf{q} = [x, y, z, w]$ and $-\mathbf{q} = [-x, -y, -z, -w]$ represent the same rotation and one has to be careful to always choose the closest quaternion when analyzing sequences of rotations or performing interpolation.

Numerical algorithms that work on derivations in a small neighborhood around a fixed point usually work well with quaternions as the curvature of the hyper sphere is small. However, each step usually moves the quaternion off the unit sphere and therefore the algorithms have have be be extended to include frequent renormalization.

Interpolating Quaternions

Quaternions have shown to be well-suited for interpolation between two given rotations. The simplest approach is linear interpolation of the four quaternion components independently (*Lerp*):

$$\mathit{Lerp}(\mathbf{p}, \mathbf{q}, h) = \mathbf{p}(1 - h) + \mathbf{q}h \tag{5.9}$$

As this causes the interpolated quaternion to move through the inside of the unit hyper sphere, the output has to be normalized after each step. This results in a motion that always takes the shortest path between the two orientations, however, the angular velocity is not constant and increases to the center.

In order to correct this, the spherical linear interpolation (*Slerp*) was developed, which keeps the resulting quaternion on a great arc on the unit sphere while maintaining constant angular velocity. Slerp can be expressed by the following formula:

$$Slerp(\mathbf{p}, \mathbf{q}, h) = \mathbf{p} \otimes (\mathbf{p}^* \otimes \mathbf{q})^h \tag{5.10}$$

Three other equivalent formulations of the *Slerp* algorithm are possible [**?**]. $(\mathbf{p}^* \otimes \mathbf{q})$ effectively expresses a difference quaternion that has to be multiplied to $\mathbf{p}$ in order to reach $\mathbf{q}$. As this difference is increased from $(\mathbf{p}^* \otimes \mathbf{q})^0$, which results in the identity quaternion, to $(\mathbf{p}^* \otimes \mathbf{q})^1$, the output quaternion smoothly travels from $\mathbf{p}$ to $\mathbf{q}$ on the unit sphere. An alternative formulation of the *Slerp* algorithm avoids the quaternion exponentiation, which I have not yet defined, by adding correction factors to the *Lerp* method:

$$\begin{aligned} \theta &= \arccos(\mathbf{p} \cdot \mathbf{q}) \\ Slerp(\mathbf{p}, \mathbf{q}, h) &= \frac{\mathbf{p}\sin((1-h)\theta) + \mathbf{q}\sin(h\theta)}{\sin\theta} \end{aligned} \tag{5.11}$$

As θ is derived from the dot-product of $\mathbf{p}$ and $\mathbf{q}$, it represents the angle between the two quaternion in 4D-space. This expression however has an undefined result when $\mathbf{p} = \pm\mathbf{q}$.

Similarly to the *Slerp* interpolation between two rotations, spherical Bézier curves can be defined for smooth interpolation of longer orientation sequences [**?**, **?**].

Advantages: Quaternions are well-suited for the representation of rotations as they are free from singularities, numerically well-conditioned and typical Ubitrack operations a inversion, product and interpolation can be performed easily. Also, direct rotation of vectors is possible.

Drawbacks: The fact that $\mathbf{q}$ and $-\mathbf{q}$ represent the same rotation requires some attention in the analysis of rotation sequences and numerical algorithms have to be designed carefully in order to avoid "falling off" the unit hyper sphere.

5.1.4 Exponential Maps

Exponential maps are another method for representing rotations using three scalar elements. They map a vector in $\mathbb{R}^3$ to a corresponding rotation, which can be represented either by a unit quaternion or a rotation matrix. A rotation quaternion can be computed from a vector $\mathbf{v}$ using the following series expansion [**?**]:

$$\mathbf{q}_\mathbf{v} = e^\mathbf{v} = \sum_{n=0}^{\infty} (\frac{1}{2}\tilde{\mathbf{v}})^n, \qquad \mathbf{v} \in \mathbb{R}^3 \tag{5.12}$$

where $\tilde{\mathbf{v}}$ is the vector quaternion produced from $\mathbf{v}$ by setting the scalar part to 0, and $\frac{1}{2}\tilde{\mathbf{v}}$ is raised to the power using standard quaternion multiplication. Fortunately, it is not necessary to explicitly compute this power series, as a simpler version of the quaternion exponential map exists:

$$\mathbf{q}_\mathbf{v} = e^\mathbf{v} = [\frac{\cos(\frac{1}{2}|\mathbf{v}|)}{|\mathbf{v}|}\mathbf{v}, \cos(\frac{1}{2}|\mathbf{v}|)] \tag{5.13}$$

In order to generate a rotation matrix with the exponential map, three generator matrices are used [**?**]:

$$G_1 = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad G_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \qquad G_3 = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The final rotation matrix is created from $\mathbf{v}$ by weighting the three generator matrices with the entries of the vector and computing an infinite power series expansion:

$$\begin{aligned} G &= v_1 G_1 + v_2 G_2 + v_3 G_3 \\ M &= e^G = \sum_{n=0}^{\infty} \frac{1}{n!} G^n \end{aligned} \tag{5.14}$$

From equation 5.13 an intuitive meaning of the vector $\mathbf{v}$ can be derived. Similar to quaternions, the direction of the vector points in the direction of the rotation axis. The angle of rotation however is determined by the length of the vector.

The conversion from a vector to a quaternion defined by the exponential map in fact is an exponentiation of a vector quaternion. The inverse operation, the log map, can be defined as:

$$\mathbf{v} = \log \mathbf{q} = \frac{2\cos^{-1} q_w}{|\mathbf{q}_v|}\mathbf{q}_v \tag{5.15}$$

where q_w and $\mathbf{q}_v$ are the scalar and vector parts of the quaternion $\mathbf{q}$. In many cases it is tempting to assume that the property $ab = \log a + \log b$ known from scalar algebra also holds for quaternions, but this generally isn't true, as quaternion multiplication is not commutative.

Every representation of rotation in $\mathbb{R}^3$ must exhibit singularities. In case of exponential maps, these are all vectors of length $|\mathbf{v}| = 2k\pi, k = 1, 2, 3, \ldots$, as a rotation of 2π around any axis is equivalent to no rotation at all [**?**]. However, because the direction of rotation can be changed by inverting $\mathbf{v}$, all rotations can be expressed by vectors $|\mathbf{v}| \leq \pi$, which are far away from any singularity.

Advantages: Like Euler angles, exponential map representations have small state vector with only three elements and therefore there is no danger of falling off a meaningful subspace, but the singularities can be avoided more easily than with Euler angles. Linear interpolation often gives results that are close to what can by obtained by the *Slerp* algorithm.

Drawbacks: Like all representations in $\mathbb{R}^3$, also the exponential map has singularities and there is no simple form for the product of multiple rotations and the rotation of vectors.

5.2 Representing Uncertainty

The value of a measurement by itself doesn't tell us anything about its quality. However, knowing the quality is important when we wish to make decisions based on it, e.g. by weighting multiple measurements, adapting the user interface to different resolutions [**?**] or by simply ignoring bad sensor readings. By describing measurements as random variables, we can express how precise a measurement is, and what other values can be considered with a certain probability. Using error propagation methods, we can also compute the errors that result when measurements are transformed or combined.

5.2.1 Multi-Dimensional Random Variables

Although it is somewhat basic, I included this section into my thesis, because the standard math courses taught in computer science at TU München (HM 1/2, DS 1/2) do not address statistics with multi-dimensional random variables. I do, however, assume that the reader is familiar with scalar random variables and probability distributions. For a more detailed introduction including complete derivations and proofs, I refer to [**?**, **?**].

But why is it important to treat multi-dimensional measurements different from scalar ones? After all, one could simply describe each measurement dimension as a distinct random variable. The reason is that using multi-dimensional error distributions, we can include additional information about dependencies between the dimensions. Consider an optical tracker that delivers three dimensional measurements, where the z values are derived from multiple x and y positions. If we know how the position in z depends on x and y, and we have an additional tracker that measures the z dimension with a higher resolution, the this knowledge can be used to further increase not only the z, but also the x and y values.

Probability distribution functions in scalar statistics give the probability that the value of a random variable is less than a value x:

$$F(x) := P(X < x) \tag{5.16}$$

This can be generalized to multiple dimensions and is called *Joint Probability Distribution Function*:

$$F(x_1, \ldots, x_n) := P(X_1 < x_1 \wedge \cdots \wedge X_n < x_n) \tag{5.17}$$

Similarly to the scalar probability distribution function, the vectorial one is monotonically increasing for each dimension and has the following special values:

$$\begin{aligned} F(-\infty, \ldots, -\infty) &= 0 \\ F(+\infty, \ldots, +\infty) &= 1 \\ F(x_1, \ldots, x_{i-1}, -\infty, x_{i+1}, \ldots, x_n) &= 0 \end{aligned}$$

When one dimension is set to $+\infty$, it disappears from the density function and the remainder is called *Marginal Probability Distribution*. This is illustrated for the two-dimensional case:

$$F(x_1, +\infty) = P(X_1 < x_1) = F_1(x_1) \tag{5.18}$$

Probability density functions for scalar random variables are defined to be the derivation of the distribution function:

$$f(x) := \frac{d}{dx}F(x) \tag{5.19}$$

The same holds for the n-dimensional case:

$$f(x_1, \ldots, x_n) := \frac{\partial^n}{\partial x_1 \ldots \partial x_n}F(x_1, \ldots, x_n) \tag{5.20}$$

Because the density function is the derivative of a monotonic function, $f(x_1, \ldots, x_n)$ always is above or equal to 0. The improper integral over the whole density function must always be 1:

$$\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(x_1, \ldots, x_n)\, dx_1 \ldots dx_n = F(+\infty, \ldots, +\infty) = 1 \tag{5.21}$$

Using the density function, the probability of rectangular areas can be computed by integration:

$$P(l_1 \leq x_1 < u_1, \ldots, l_n \leq x_n < u_n) = \int_{l_1}^{u_1} \ldots \int_{l_n}^{u_n} f(x_1, \ldots, x_n)\, dx_1 \ldots dx_n \tag{5.22}$$

Expectation indicates the mean of the random variable, which in the scalar case is computed by weighting each possible value with the corresponding value of the probability density function:

$$\mu = E[X] = \int_{-\infty}^{+\infty} x f(x)\, dx \tag{5.23}$$

For vectorial random variables, $E[X] = \mu$ also is a vector with

$$\mu_i = E[X_i] = \int_{-\infty}^{+\infty} \ldots \int_{-\infty}^{+\infty} x_i f(x_1, \ldots, x_n)\, dx_1 \ldots dx_n \tag{5.24}$$

Variance of scalar random variables is a measure for how much the variable is expected to diverge from its expectation μ:

$$\text{Var}[X] = E[(X - E[X])^2] = \int_{-\infty}^{+\infty} (x - \mu)^2 f(x)\, dx = E[X^2] - E[X]^2 \tag{5.25}$$

The root of the variance $\sigma = \sqrt{\text{Var}[X]}$ is called *Standard Deviation* and has the advantage of having the same unit as the random variable itself.

In case of multiple scalar random variables, besides the variance, also the *covariance* $\text{Cov}[X_i, X_j]$ can be computed, which gives an indication about how much the two variables correlate:

$$\begin{aligned} \text{Cov}[X_i, X_j] &= E[(X_i - E[X_i])(X_j - E[X_j])] = E[X_i X_j] - E[X_i]E[X_j] \\ &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} (x_i - \mu_i)(x_j - \mu_j) f(x_i, x_j)\, dx_i dx_j \end{aligned} \tag{5.26}$$

A special case is $\text{Cov}[X_i, X_i]$, which is the same as the normal variance $\text{Var}[X_i]$. The covariance can be normalized, yielding the *correlation coefficient* ρ:

$$\rho = \frac{\text{Cov}[X_i, X_j]}{\sqrt{\text{Var}[X_i]\text{Var}[X_j]}} \tag{5.27}$$

The correlation coefficient always is in the interval $-1 \leq \rho \leq 1$. If $\rho = 0$, X_i and X_j are called uncorrelated; a value of $\rho = \pm 1$ implies linear dependency.

For two vectorial random variables X and Y of dimensions n and m, the covariances between the single entries of X and Y can be combined into a $n \times m$ *cross covariance matrix* , where $\text{Cov}[X, Y]_{i,j} = \text{Cov}[X_i, Y_j]$. Using vector arithmetic, this can be expressed as:

$$\text{Cov}[X, Y] = E[(X - E[X])(Y - E[Y])^T] = E[XY^T] - E[X]E[Y]^T \tag{5.28}$$

When $X = Y$, this matrix is called the *covariance matrix* of X:

$$\begin{aligned} \text{Cov}[X] &= E[(X - E[X])(X - E[X])^T] = E[XX^T] - E[X]E[X]^T \\ &= \begin{bmatrix} \text{Var}[X_1] & \text{Cov}[X_1, X_2] & \cdots & \text{Cov}[X_1, X_n] \\ \text{Cov}[X_2, X_1] & \text{Var}[X_2] & \cdots & \text{Cov}[X_2, X_n] \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}[X_n, X_1] & \text{Cov}[X_n, X_2] & \cdots & \text{Var}[X_n] \end{bmatrix} \end{aligned} \tag{5.29}$$

The covariance matrix is always square and symmetric, as $\text{Cov}[X_i, X_j] = \text{Cov}[X_j, X_i]$, and the diagonal contains the variances of the single entries $\text{Cov}[X]_{ii} = \text{Var}[X_i]$.

5.2.2 Normal Distributions

The normal distribution, often called Gaussian distribution, is of particular importance in most areas of statistics. This is because the *central limit theorem* states that under certain, but rather general, circumstances, the sum of arbitrarily distributed random variables tends towards the normal distribution, as the number of variables approaches infinity [**?**, **?**]. This is relevant for dealing with sensor errors, because usually there are many independent sources of error which contribute the overall measurement error. Therefore, we can justify describing measurement errors using a normal distribution.

A scalar random variable has a normal distribution $X \sim N(\mu, \sigma^2)$, if it has a density function of

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad \text{where} -\infty < x < \infty \tag{5.30}$$

One can prove that $\int_{-\infty}^{+\infty} f(x)dx = 1$. The density function is symmetric around the expectation μ. The probability, that a normally distributed random variable is in the interval $\mu \pm \sigma$ is 68.3%, $\mu \pm 2\sigma$ is 95.4% and $\mu \pm 3\sigma$ is 99.7%.

An n-dimensional random variable X is normally distributed $X \sim N(\mu, \Sigma)$, if its density function is:

$$f(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n \text{det}\Sigma}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1}(\mathbf{x}-\mu)} \qquad \text{where} -\infty < x < \infty \tag{5.31}$$

It can be proven that μ is the expectation as defined above, and Σ is the covariance matrix. For a more intuitive understanding of eq. 5.31, Σ^{-1} can be decomposed into $\Sigma^{-1} = AA^T$, because a covariance matrix and its inverse always is symmetric and positive definite. By substituting the power term in the equation with $-1/2(A^T\mathbf{y})^T(A^T\mathbf{y})$, we see that the vector $\mathbf{x}$ is offset by the expectation μ, rotated and scaled by the matrix A^T, before the squared distance from the center is computed by the scalar product. To this distance the scalar normal distribution (eq. 5.30) is applied.

Interpretation of Covariance

As we have seen, the covariance matrix effectively describes a scaling and a rotation of space. When looking at points of constant distance to the origin, these also are transformed from a (hyper-)sphere to an arbitrarily oriented ellipsoid. On such an ellipsoid all points of constant probability are located.

In order to compute the ellipsoid, the covariance matrix can be decomposed into its eigenvalues and eigenvectors. The eigenvectors give the directions of the principal axes and the associated eigenvalues determine the variance in that direction.

5.2.3 Error Propagation

Many operations in the Ubitrack dataflow involve transforming and combining measurements. In every step of the computation, also the error statistics in form of covariance matrices must be adapted. It should be intuitively clear, that a rotation of a measurement with a high uncertainty along one axis will also cause this axis to point in a different direction.

Linear transformations are the most basic computations that can be performed on multivariate random variables. The following equations show how expectation and covariance of a random variable X of size n are changed by applying a transformation of the form $Y = AX + \mathbf{b}$, where A is a $n \times n$ matrix and $\mathbf{b}$ a n-vector.

In order to compute the new expectation of the transformed variable, the following equation holds (proof in [**?**]):

$$E[Y] = E[AX + \mathbf{b}] = AE[X] + \mathbf{b} \tag{5.32}$$

Based on this formula, we can also compute the transformed covariance matrix as

$$\mathrm{Cov}[Y] = \mathrm{Cov}[AX + \mathbf{b}] = A\,\mathrm{Cov}[X]\,A^T \tag{5.33}$$

The proof of eq. 5.33 is simple: $\mathrm{Cov}[Y] = E[(Y - E[Y])(Y - E[Y])^T] = E[(AX + \mathbf{b} - AE[X] - \mathbf{b})(AX + \mathbf{b} - AE[X] - \mathbf{b})^T] = AE[(X - E[X])(X - E[X])^T]A^T = A\,\mathrm{Cov}[X]\,A^T$

Non-linear transformations can be arbitrary transformations $Y = \mathbf{f}(X)$ where both X and Y are vectorial random variables.

In order to transform the expectation, the formula $E[Y] = \mathbf{f}(E[X])$ generally does not hold for non-linear functions $\mathbf{f}$. However, it often serves as a good approximation, if X is

concentrated in a small area around its expectation, and $\mathbf{f}(X)$ is approximately linear in this area.

For computing the transformed covariance matrix, $\mathbf{f}(X)$ is linearized using the multi-dimensional Taylor expansion:

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}(\mathbf{x}_0 + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}_0) + \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_0} \Delta\mathbf{x} + \dots \tag{5.34}$$

where $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is the *Jacobian* matrix, which can be considered as the first derivation of a vector-valued function. A Jacobian contains the partial derivatives of all outputs with respect to all inputs:

$$\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)_{i,j} = \frac{\partial f_i(\mathbf{x})}{\partial x_j} \tag{5.35}$$

Now we can rewrite the non-linear transformation to

$$\mathbf{y} = \mathbf{y}_0 + \Delta\mathbf{y} = \mathbf{f}(\mathbf{x}_0 + \Delta\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + J\Delta\mathbf{x}$$

where $\mathbf{x}_0 = E[X], \mathbf{y}_0 = E[Y], \Delta\mathbf{x} \sim N(0, \mathrm{Cov}[X])$ and $\Delta\mathbf{y} \sim N(0, \mathrm{Cov}[Y])$. Applying eq. 5.33 yields the transformed covariance matrix:

$$\mathrm{Cov}[Y] = \mathrm{Cov}[\mathbf{f}(X)] = J\,\mathrm{Cov}[X]\,J^T \tag{5.36}$$

where $J = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\mathbf{x}=E[X]}$. This equation only holds in a small area around the expectation, but usually this is not a problem, as the error can be considered small compared to the random variable.

Combining random variables can be seen as a special case of non-linear transformations, where multiple random variables are combined into one. A frequent Ubitrack example is the chaining of measurements along a path. The combination can be expressed as a function of multiple random variables $Y = \mathbf{f}(X_1, \dots, X_n)$, where each X_i has an associated covariance matrix C_i. By assuming independence between all X_i, we can combine these into a single random variable X' with covariance C':

$$X' = (X_1^T, \dots, X_n^T)^T, \qquad C' = \begin{bmatrix} C_0 & & 0 \\ & \ddots & \\ 0 & & C_n \end{bmatrix}$$

Now the non-linear error propagation formula from the previous section can be applied, resulting in a new covariance matrix $C_Y = J\,C'\,J^T$. The Jacobian J usually is non-square, having fewer rows than columns. Using this and the information about the zero entries in the combined covariance matrix C', a more efficient expression for C_Y can be derived:

$$C_Y = J_1\,C_1\,J_1^T + \dots + J_n\,C_n\,J_n^T \tag{5.37}$$

where each $J_i = \frac{\partial \mathbf{f}(X_1,\dots,X_n)}{\partial X_i}$ is the Jacobian of $\mathbf{f}$ with respect to the random variable X_i.

The unscented transform [?] provides a tool for transforming expectations, covariances and higher-order moments of random variables when the function is highly non-linear or even discontinuous in the relevant area. This is obtained by choosing a number of strategically placed sample points that have the same distribution as the random variable. These sample points are then transformed by the function $\mathbf{f}$ and the relevant properties (expectation, covariance, etc.) are computed from the transformed samples. The unscented transform differs from Monte Carlo methods, such as particle filters, in that it does not select the sample points randomly and therefore a smaller sample is sufficient.

5.2.4 Optimal Estimation

The goal of optimal estimation is to determine a vector of n parameters $\mathbf{x}$ from an m-vector $\mathbf{z}$ of noisy measurements. The measurements $\mathbf{z}$ do not necessarily describe the parameters directly, but may be a function of $\mathbf{x}$ instead. An example of this is an optical tracker, where the 6D-pose is not directly measured, but the 2D-positions of multiple features, which can be described as a function of position and orientation. If the relation between $\mathbf{x}$ and $\mathbf{z}$ is linear, we can describe the measurement process as

$$\mathbf{z} = H\mathbf{x} + \mathbf{v}, \qquad \mathbf{v} \sim N(0, R) \tag{5.38}$$

where H is an $m \times n$ matrix that describes how the measurements relate to the parameters and $\mathbf{v}$ is a vector of measurement errors. $\mathbf{v}$ is not directly known, but assumed to have a Gaussian distribution with an expectation of 0 and a covariance matrix of R. The distribution of $\mathbf{v}$ effectively describes the accuracy of the measurements.

In general $m > n$, i.e. there are more measurements than parameters and equation 5.38 is over determined and therefore in general has no solution. The usual approach is to determine an estimation $\hat{\mathbf{x}}$ of the parameters which minimizes the quadratic distance between the estimate and the measurements:

$$(\mathbf{z} - H\hat{\mathbf{x}})^T(\mathbf{z} - H\hat{\mathbf{x}}) = \text{Min!} \tag{5.39}$$

This approach is usually called *Least-Squares Estimation*. In order to find the $\hat{\mathbf{x}}$, which minimizes equation 5.39, the derivation (Jacobian) is computed and set to zero. This results in the following estimate of the parameters:

$$\hat{\mathbf{x}} = (H^T H)^{-1} H^T \mathbf{z} \tag{5.40}$$

The expression $(H^T H)^{-1} H^T$ is also known as the *Moore-Penrose Inverse* or *Pseudoinverse* H^+. The usual least-squares approach however gives equal weight to all measurements and therefore does not honor the knowledge about the measurement accuracy, which is available in form of the covariance matrix R. We can integrate the known accuracies and modify the cost function by weighting the measurements with the inverse of the covariance matrix, which leads to the *Weighted Least-Squares* problem:

$$(\mathbf{z} - H\hat{\mathbf{x}})^T R^{-1}(\mathbf{z} - H\hat{\mathbf{x}}) = \text{Min!} \tag{5.41}$$

This results in the following parameter estimate:

$$\hat{\mathbf{x}} = (H^T R^{-1} H)^{-1} H^T R^{-1} \mathbf{z} \tag{5.42}$$

A more detailed derivation can be found in [**?**]. If we want to know how accurate the estimate is, we can compute its covariance matrix $P = E[(\hat{\mathbf{x}}-\mathbf{x})(\hat{\mathbf{x}}-\mathbf{x})^T]$ using the following formula:

$$P = (H^T R^{-1} H)^{-1} \tag{5.43}$$

Alternatively, we can derive equation 5.42 by treating the problem as a probabilistic one where we wish to find the point of *Maximum Likelihood* in a Gaussian probability distribution. Therefore the parameters of the density function have to be optimized such that the probability of the measurement $\mathbf{z}$ becomes maximal. The measurement probability can be treated as a conditional one $p(\mathbf{z}|\mathbf{x})$ [**?**]:

$$p(\mathbf{z}|\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^m \det R}} e^{-\frac{1}{2}(\mathbf{z}-H\mathbf{x})^T R^{-1}(\mathbf{z}-H\mathbf{x})} \tag{5.44}$$

In order to maximize this expression, the exponent has to be minimized. We can see that up to a constant factor, this is the same as minimizing the weighted least-squares cost function 5.41 and therefore the result must be equation 5.42.

Recursive Estimation

In practical parameter estimation situations, there often isn't a single big measurement, but the input data is derived from multiple independent measurements. These may come from multiple sensors and/or from multiple measurements taken at different points in time. Instead of combining these measurements and performing one single parameter estimation, we wish to integrate the measurements one-by-one into our estimate of the parameter vector. This has the advantage of better computational efficiency, as the necessary parameter vectors and matrices remain small. Also, intermediate results can be obtained before all measurements are done. This derivation of the recursive estimation formula roughly follow the one by [**?**].

In order to derive the recursive parameter estimation formula, the measurement $\mathbf{z}$ is split up into two independent measurements $\mathbf{z}_{k-1}$ and $\mathbf{z}_k$:

$$H = \begin{vmatrix} H_{k-1} \\ H_k \end{vmatrix}, \qquad \mathbf{z} = \begin{vmatrix} \mathbf{z}_{k-1} \\ \mathbf{z}_k \end{vmatrix} \quad \text{and} \quad R = \begin{vmatrix} R_{k-1} & 0 \\ 0 & R_k \end{vmatrix}$$

Inserting this into equation 5.42 and exploiting the zero entries in R results in the following estimate:

$$\hat{\mathbf{x}} = (H_{k-1}^T R_{k-1}^{-1} H_{k-1} + H_k^T R_k^{-1} H_k)^{-1}(H_{k-1}^T R_{k-1}^{-1} \mathbf{z}_{k-1} + H_k^T R_k^{-1} \mathbf{z}_k) \tag{5.45}$$

In order to create a recursive form of the optimal estimation, we insert the previous estimate $\hat{\mathbf{x}}_{k-1}$ with covariance P_{k-1} as the first part of the measurement:

$$\mathbf{z}_{k-1} = \hat{\mathbf{x}}_{k-1}, \qquad H_{k-1} = I \quad \text{and} \quad R_{k-1} = P_{k-1}$$

This leads to the following new estimate:

$$\hat{\mathbf{x}}_k = (P_{k-1}^{-1} + H_k^T R_k^{-1} H_k)^{-1}(P_{k-1}^{-1} \hat{\mathbf{x}}_{k-1} + H_k^T R_k^{-1} \mathbf{z}_k) \tag{5.46}$$

The equation can be reformulated to (see [?] for a full derivation)

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k-1} + K_k(\mathbf{z}_k - H_k\hat{\mathbf{x}}_{k-1}) \tag{5.47}$$

where

$$K_k = P_{k-1}H_k^T(H_kP_{k-1}H_k^T + R_k)^{-1}$$

The difference $(\mathbf{z}_k - H_k\hat{\mathbf{x}}_{k-1})$ is called the measurement innovation or residual [?], which reflects the difference between the actual measurement $\mathbf{z}_k$ and the expected one $H_k\hat{\mathbf{x}}_{k-1}$. The matrix K determines the amount by which the innovation is integrated into the final estimate. An alternative derivation of the matrix K is obtained by minimizing the trace of the covariance matrix P_k of the estimate [?].

The covariance of the new estimate is computed by

$$P_k = (I - K_kH_k)P_{k-1} \tag{5.48}$$

5.3 Kalman Filters

Until now we have considered estimation in the static case in which all elements of the parameter vector are constant. This would already allow us to compute positions and orientations in a motionless Ubitrack system, where measurements are overlaid by Gaussian noise. In general, the parameters to be estimated (position, orientation, velocity, etc.) however do vary with time as users change place or move objects and therefore the previously developed methods cannot be applied directly, as multiple consecutive measurements usually do not relate to the same system state. In order to allow estimation of parameters from multiple measurements in a dynamic system without completely discarding information from previous sensor readings, the kalman filter was developed.

5.3.1 Linear Dynamic Systems

First we need a general way to describe dynamic systems. We assume that the *state vector* $\mathbf{x}(t)$ completely describes all relevant quantities of the system state at any time. If the location of a moving object is to be determined, the state vector usually includes position, velocity and higher-order derivations which is sufficient to describe the positions in a small time interval around t. When the motion is linear, changes of the state vector in time can be described using a first-order vector-matrix differential equation [?]:

$$\dot{\mathbf{x}}(t) = F(t)\mathbf{x}(t) + L(t)\mathbf{u}(t) + G(t)\mathbf{w}(t) \tag{5.49}$$

The vector $\mathbf{u}(t)$ is a deterministic *control input*, which describes external, but known influences on the state vector, such as the position of the steering wheel which affects the orientation of a car. $\mathbf{w}(t)$ describes the *process noise* which summarizes all unknown influences and is usually modeled as a random variable with a zero-mean distribution. The matrices $F(t)$, $G(t)$ and $I(t)$ determine how state vector, process noise and control input affect the derivation.

Usually, we are not interested in the state vector at all times, but only at discrete points in time. This allows a simplification of the above model:

$$\mathbf{x}_{k+1} = A_k\mathbf{x}_k + B_k\mathbf{u}_k + C_k\mathbf{w}_k \tag{5.50}$$

Now the *transition matrix* A_k describes how the state transforms from time t_k to time t_{k+1}.

As in the static estimation problem, it is often impossible to measure the whole state vector directly, as each sensor actually measures only a function of the state:

$$\mathbf{z}_k = H_k\mathbf{x}_k + \mathbf{v}_k \tag{5.51}$$

The *measurement matrix* H_k describes how the measurement relates to the state vector and the vector $\mathbf{v}_k$ is a statistically distributed *measurement noise* which is added to the measurement.

In order to allow the determination of all state vector elements, the transition and measurement matrices A_k and H_k need to fulfill certain conditions, which are explained in more detail in [**?**]. In general, it is not necessary to measure each element of the state vector directly, but it may suffice, if the transition matrix A_k describes the interrelations between the directly and indirectly measured quantities. For computing a hypothetical Ubitrackstate vector consisting of position and velocity, it is sufficient to measure position at two distinct times if the transition matrix describes how velocity influences position. If all state vector elements can be determined directly or indirectly by measurements, the system is called *completely observable*.

For the Kalman filter, both the process noise vector $\mathbf{w}$ and the measurement noise vector $\mathbf{v}$ are assumed to have a zero-mean Gaussian distribution:

$$\mathbf{w} \sim N(0, Q), \quad \mathbf{v} \sim N(0, R)$$

We also assume that the process noise $\mathbf{w}$ has the same dimension as the state vector and therefore we can set $C_k = I$ in the system model. Furthermore, for ideal Kalman filtering conditions, each of the $\mathbf{w}$ and $\mathbf{v}$ vector must behave like *white noise*, i.e. it must be uncorrelated over time. This condition however cannot be met in real systems, as uncorrelated noise would mean infinite energy, but the white noise assumption often yields sufficient results.

Summarizing this section, the dynamic system whose state vector $\mathbf{x}_k$ is to be estimated by the linear discrete-time Kalman filter using measurements $\mathbf{z}_k$, can be characterized by the following two equations:

$$\begin{aligned} \mathbf{x}_{k+1} &= A_k\mathbf{x}_k + B_k\mathbf{u}_k + \mathbf{w}_k, & \mathbf{w} &\sim N(0, Q) \\ \mathbf{z}_k &= H_k\mathbf{x}_k + \mathbf{v}_k, & \mathbf{v} &\sim N(0, R) \end{aligned}$$

5.3.2 Linear Kalman Filter

A linear Kalman filter integrates all measurements in the order they are made. Each integration is done in two steps according to the *predictor-corrector* principle . In the time update step (predictor), the state vector is advanced to the time of the measurement using the known equations of the linear dynamic system. The measurement step (corrector) integrates the measurement into the predicted state vector using the standard optimal estimation technique for static quantities. The internal state of the Kalman filter consists of the estimated state vector $\hat{\mathbf{x}}$ of the dynamic system and its associated covariance matrix P_k.

The time update step computes an *a priori* estimate $\hat{\mathbf{x}}_{k+1}^-$ from the current estimate $\hat{\mathbf{x}}_k$ by advancing (predicting) the state vector to the time of the measurement $\mathbf{z}_{k+1}$. This is done by computing the state transition formula 5.50 for time-discrete linear systems. The filter's covariance matrix is updated using the error propagation formula 5.33. This leads to the following equations of the time update step:

$$\hat{\mathbf{x}}_{k+1}^- = A_k\hat{\mathbf{x}}_k + B\mathbf{u} \tag{5.52}$$
$$P_{k+1}^- = A_k P_k A_k^T + Q_k \tag{5.53}$$

The process noise vector $\mathbf{w}_k$ does not show up in these equations, but only its covariance matrix Q_k, because $\mathbf{w}_k$ is assumed to have a zero-mean Gaussian distribution.

The measurement update step integrates a new measurement into the a priori estimate of the state vector $\hat{\mathbf{x}}_k^-$ from the time update step using the recursive optimal estimation formulas 5.47 and 5.48. The result is the *a posteriori* estimate $\hat{\mathbf{x}}$.

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \tag{5.54}$$
$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + K_k(\mathbf{z}_k - H_k\hat{\mathbf{x}}_k^-) \tag{5.55}$$
$$P_k = (I - K_k H_k)P_k^- \tag{5.56}$$

5.3.3 Extended Kalman Filter

In real systems we often find that the time update or measurement equations are non-linear, especially when rotation is involved. A Kalman Filter that can handle such situations by linearizing around the current estimate of the state vector is called *Extended Kalman Filter*. The linearization is done in the same fashion as in the error propagations section 5.2.3 by computing Jacobians of the non-linear functions. Due to this linearization, the extended Kalman filter does not always provide an optimal estimation, but rather an approximation, where the quality depends on the condition of the problem.

In a non-linear dynamic system, both the state transition and the measurements may be described not only by matrices, but by arbitrary functions:

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w}_k) \tag{5.57}$$
$$\mathbf{z}_k = h(\mathbf{x}_k, \mathbf{v}_k) \tag{5.58}$$

Because the noise vectors $\mathbf{w}_k$ and $\mathbf{v}_k$ are unknown, we will assume a zero-mean Gaussian distribution and set the vectors to zero, when f or h must be evaluated in subsequent equations.

The time update step of the Extended Kalman Filter, which projects the current state and the associated covariance matrix into the future, now has the following form:

$$\hat{\mathbf{x}}_{k+1}^- = f(\hat{\mathbf{x}}_k, \mathbf{u}_k, 0) \tag{5.59}$$
$$P_{k+1}^- = A_k P_k A_k^T + W_k Q_k W_k^T \tag{5.60}$$

As for the linear case, Q_k is the covariance matrix of the process noise $\mathbf{w}_k$. The Jacobian matrices A_k and W_k are partial derivatives of the time update function f, evaluated at the current estimate of the state vector $\hat{\mathbf{x}}_k$:

$$A_k = \frac{\partial f}{\partial \mathbf{x}}(\hat{\mathbf{x}}_k, \mathbf{u}_k, 0), \qquad W_k = \frac{\partial f}{\partial \mathbf{w}}(\hat{\mathbf{x}}_k, \mathbf{u}_k, 0)$$

The measurement update step of the Extended Kalman Filter has the following form:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \tag{5.61}$$
$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + K(\mathbf{z}_k - h(\hat{\mathbf{x}}_k^-, 0)) \tag{5.62}$$
$$P_k = (I - K_k H_k) P_k^- \tag{5.63}$$

where H_k and V_k are Jacobians of the measurement function h, evaluated at the predicted a priori state estimate $\hat{\mathbf{x}}_k^-$:

$$H_k = \frac{\partial h}{\partial \mathbf{x}}(\hat{\mathbf{x}}_k^-, 0), \qquad W_k = \frac{\partial f}{\partial \mathbf{w}}(\hat{\mathbf{x}}_k^-, 0)$$

5.3.4 Choice of Model Parameters

So far I have explained the mathematical basics of the (extended) Kalman filter. However, it is still not intuitively clear how to set the various parameters involved.

State Vectors must hold all parameters that we are interested in observing, as well as parameters that are necessary for the system dynamics. In a simple example we might want to determine the one-dimensional position of a moving object along an axis. The object has a non-zero mass and therefore the physical rules of inertia hold, which means that, in addition to position, also the current velocity and acceleration are important parameters for the system dynamics. Therefore we have a linear 2nd-order system and could use the following vector to describe the system state:

$$\mathbf{x} = [x, v, a]^T \tag{5.64}$$

Note that if position and/or orientation is to be computed in sensor setups that include inertial sensors, it is not always necessary to include the position or orientation into the state vector. Foxlin [**?**] uses a so-called complimentary Kalman filter, which estimates not the orientation itself, but instead the bias in the integrated signal of the inertial sensor. This reduces the size of the state vector and results in a faster tracker response.

State Transitions are determined by the transition matrix A for linear system, and by the time update function f in case of an extended Kalman filter. In the simplest case, the parameters to be determined are constant and therefore the matrix A can be set to identity. When we also set Q to 0 and there is no control input, the linear Kalman filter reduces to the recursive optimal estimator (eq. 5.47).

For the example system from equation 5.64, both velocity and acceleration need to be integrated and added to the position. In addition, acceleration also adds to velocity. As we have no further information about the dynamics of acceleration, we treat it as constant. This model results in the following transition matrix:

$$A = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix} \tag{5.65}$$

Control Inputs are all deterministic input quantities that have an effect on the dynamic system. An example are the known control impulses that are sent to the motors that drive a robot's wheels. In typical Ubitrack scenarios there usually is no control input, as motions are caused by the user who is moving himself or other objects without announcing it in advance.

Measurements are described by the measurement matrix H for linear systems, and by the measurement function h for the extended Kalman filter. In case of the simple second-order linear example described by equations 5.64 and 5.65, the measurement matrix would simply extract the measured value from the state vector. For a position sensor, H has the following form:

$$H = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

Note that using only this measurement equation repeatedly, both velocity and acceleration are computed by the Kalman filter from position without explicit derivation. This is because the time update step also updates the covariance matrix, which describes how the two derivations depend on position, and the subsequent measurement step can then correct all state vector entries.

Process Noise Covariance Matrices Q are the most difficult to set parameters of the Kalman filter. Intuitively, one can say that high values cause old measurements do decay quickly and new measurements are given a higher weighting. If Q contains small values, the filter tends to react to new measurements very slowly. In the extreme case of $Q = 0$, those state vector entries that are unaffected by the state transition (the acceleration in the above example) are treated as constant values, and the Kalman filter effectively becomes a recursive least-squares estimator.

Unfortunately good values for the process noise covariance are difficult to compute. Therefore these are usually either set manually [**?**] or determined by running a general-purpose optimizer, such as Powell's method, offline over a pre-recorded data set [**?**, **?**]. A good cost function is the squared sum of the innovation term $\mathbf{z}_k - H_k\hat{\mathbf{x}}_k^-$ in the measurement update equation 5.55. The innovation effectively describes how good the filter predicts a new measurement. In order to reduce the dimension of the search space, it is usually sufficient to set all non-diagonal elements of the covariance matrix to zero.

Instead of using a single filter with fixed process noise parameters, some approaches run multiple Kalman filters with different parameters in parallel and select the one that gives the best results for the filter output [**?**]. This can even be extended to algorithms that combine Kalman filtering with hidden Markov models and automatically learn both the filter parameters and the transitions between different filters [**?**, **?**].

Measurement Noise Covariance Matrices R ideally describe the true covariance of each sensor measurement. The accuracy can often be found in the sensor's data sheets, but is better determined by empirical evaluation.

In fixed sensor setups where the measurement accuracy is constant, R can be used in addition to Q for tuning the overall performance of the Kalman filter [**?**]. Such an approach however is generally unsuitable for Ubitrack, as measurement attributes should be comparable for all sensors and not only be valid in a fixed tracker configuration.

5.3.5 Sensor Fusion

Whenever multiple sensors can measure interdependent state vector entries, we can increase the overall accuracy or other qualities of the final state estimate by combining (fusing) measurements from these sensors. There are two different approaches to sensor fusion using Kalman filters.

Synchronous sensor fusion combines multiple sensor values into a single measurement vector, which is integrated into the state vector in one predictor-corrector cycle. Adding both an absolute position sensor and an inertial acceleration sensor to the example from equation 5.64 and 5.65, we get the following synchronous measurement matrix:

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The drawback of the synchronous sensor fusion approach is that the measurements are implicitly assumed to have taken place at exactly the same time. This assumption however is rarely true in ubitrack scenarios where information from many independent sensors is to be combined.

Single-constraint-at-a-time (SCAAT) Kalman filtering aims at integrating as few information as possible in one measurement update step [**?**]. In this case we would perform a distinct update cycle for each sensor, using different measurement matrices or functions. Therefore measurements can be integrated in the order they are made, avoiding the simultaneity assumption.

6 Modelling Errors in Ubitrack

The previous chapter has introduced the underlying mathematical theory of this thesis, but no concrete application was given. This chapter applies the theory of random variables and dynamic systems to pose measurements consisting position and orientation. The goal is to derive all mathematics that are necessary to reach the goals of this thesis and to provide mathematical methods that can directly be implemented. The implementation in DWARF is described in the next chapter.

6.1 Static Errors

The first part of this chapter deals with the description of static random errors, which affect single measurements and are present even in systems without motion. I also describe how errors propagate in a ubitrack system when measurements are chained or inverted.

6.1.1 Error Model

At first we have to decide how measurements and their accompanying errors are to be described and interpreted. Each measurement in a ubitrack system consists of a translation vector $\mathbf{t}$ and a rotation quaternion r. This describes a transformation of a vector $\mathbf{x}$ from one coordinate system into another. At first the rotation r is applied, then the translation is added to the result:

$$\mathbf{x}_{\text{new}} = \mathbf{t} + r\mathbf{x}r^* \tag{6.1}$$

where r^* means quaternion conjugation and all multiplications are quaternion multiplications. For simplicity, I have omitted the $\otimes$ operator. Note that the same transformation can be expressed using a homogeneous matrix, well known from computer graphics.

Each measurement is affected by an error, both in translation and orientation. The positional error $\mathbf{e}_t$ is added to the position and the rotational error, represented by the quaternion e_r, is multiplied to the orientation.

$$\mathbf{x}_{\text{new}} = \mathbf{t} + \mathbf{e}_t + re_r\mathbf{x}e_r^*r^* \tag{6.2}$$

The rotational error e_r is assumed to be small and therefore it can be approximated by the following "small" quaternion:

$$e_r \approx [e_{r,x}, e_{r,y}, e_{r,z}, 1]$$

In fact, $e_{r,w} = \sqrt{1 - e_{r,x}^2 - e_{r,y}^2 - e_{r,z}^2}$ would give a much better approximation for the scalar part of the error quaternion. In our case however, this would make no difference as a first-order approximation is used in the error propagation and Kalman filter formulas, which always linearize around $e_r = [0, 0, 0, 1]$.

The errors are not exactly known, but assumed to be normally distributed, therefore we can describe the error using a covariance matrix. As errors in position and orientation can be dependent, such as in an optical tracker, where orientation is determined from the position of multiple features, both errors are aggregated into a single 6×6 covariance matrix C.

$$(e_{t,x}, e_{t,y}, e_{t,z}, e_{r,x}, e_{r,y}, e_{r,z})^T \sim N(\mathbf{0}, C)$$

This is the covariance matrix that is included in every ubitrack measurement.

6.1.2 Error Propagation: Inference

One of the core techniques in ubitrack systems is the combination of measurements along a path in the spatial relationship graph. If we were using homogeneous matrices to represent the measurements, the matrices could simply be multiplied. However, we don't and therefore a different method is necessary for combining both measurements and the associated error covariance matrices in order to estimate the total error of a path. Here I present the combination of two trackers, however it is possible to apply this computation repeatedly for paths consisting of more than two edges.

The two measurements yield the positions $\mathbf{t}_1$ and $\mathbf{t}_2$ as well as the rotation quaternions r_1 und r_2. If only either rotation of position are present in both measurements, the equations still hold by setting the other value to 0 or 1 respectively. Applying equation 6.2 twice, the combined "inferred" measurement can be expressed as:

$$\mathbf{x}_{\text{new}} = \mathbf{t_1} + \mathbf{e}_{t_1} + r_1 e_{r_1}(\mathbf{t}_2 + \mathbf{e}_{t_2} + r_2 e_{r_2} \mathbf{x} e^*_{r_2} r^*_2) e^*_{r_1} r^*_1 \tag{6.3}$$

this can be rewritten to

$$\mathbf{x}_{\text{new}} = \mathbf{t}_C + \mathbf{e}_{t_C} + r_C e_{r_C} \mathbf{x} e^*_{r_C} r^*_C \tag{6.4}$$

where

$$\begin{aligned} \mathbf{t}_C &= \mathbf{t}_1 + r_1 \mathbf{t}_2 r^*_1 \\ \mathbf{e}_{t_C} &= \mathbf{e}_{t_1} + r_1(e_{r_1}(\mathbf{t}_2 + \mathbf{e}_{t_2}) e^*_{r_1} - \mathbf{t}_2) r^*_1 \\ r_C &= r_1 r_2 \\ e_{r_C} &= r^*_2 e_{r_1} r_2 e_{r_2} \end{aligned}$$

t_C and r_C now represent the total translation and rotation of the chained edges and $\mathbf{e}_{t_C}$ und e_{r_C} are the total errors.

From the formula we can see what should be intuitively clear: The combined positional error $\mathbf{e}_{t_C}$ depends not only on $\mathbf{e}_{t_1}$ and $\mathbf{e}_{t_2}$, but also on the rotational error e_{r_1} and the position $\mathbf{t}_2$. If r_1 represents the angle between the arm and a human body, it should be obvious that the position of the hand depends on it. This dependency increases with the length of the arm, which is described by t_2.

In order to compute the covariance matrix C_C of the combined measurements, the special case of the error propagation formula from equation 5.37 is applied:

$$C_C = J_1 C_1 J_1^T + J_2 C_2 J_2^T$$

where J_1 and J_2 are the Jacobians $J_1 = \partial \mathbf{g} / \partial (\mathbf{e}_{t_1}, e_{r_1})^T$ and $J_2 = \partial \mathbf{g} / \partial (\mathbf{e}_{t_2}, e_{r_2})^T$ of $\mathbf{g}(\mathbf{e}_{t_1}, e_{r_1}, \mathbf{e}_{t_2}, e_{r_2}) = (\mathbf{e}_{t_C}, \mathbf{e}_{r_C})^T$ evaluated at $(\mathbf{e}_{t_1}, e_{r_1}, \mathbf{e}_{t_2}, e_{r_2})^T = \mathbf{0}$.

$$
J_1 = \begin{bmatrix}
1 & 0 & 0 & -4r_{1,y}r_{1,x}t_{2,z} + 4r_{1,z}r_{1,x}t_{2,y} + 4r_{1,y}r_{1,w}t_{2,y} + 4r_{1,z}r_{1,w}t_{2,z} \\
0 & 1 & 0 & -2r_{1,y}^2 t_{2,z} + 4r_{1,y}r_{1,z}t_{2,y} + 2r_{1,z}^2 t_{2,z} - 2r_{1,w}^2 t_{2,z} - 4r_{1,w}r_{1,x}t_{2,y} + 2r_{1,x}^2 t_{2,z} \\
0 & 0 & 1 & -4r_{1,z}r_{1,y}t_{2,z} + 2r_{1,z}^2 t_{2,y} - 2r_{1,y}^2 t_{2,y} - 4r_{1,x}r_{1,w}t_{2,z} - 2r_{1,x}^2 t_{2,y} + 2r_{1,w}^2 t_{2,y} \\
0 & 0 & 0 & r_{2,x}^2 + r_{2,w}^2 - r_{2,z}^2 - r_{2,y}^2 \\
0 & 0 & 0 & 2r_{2,x}r_{2,y} - 2r_{2,w}r_{2,z} \\
0 & 0 & 0 & 2r_{2,x}r_{2,z} + 2r_{2,w}r_{2,y}
\end{bmatrix}
$$

$$
\begin{bmatrix}
2r_{1,x}^2 t_{2,z} - 4r_{1,x}r_{1,z}t_{2,x} + 2r_{1,w}^2 t_{2,z} - 4r_{1,w}r_{1,y}t_{2,x} - 2r_{1,z}^2 t_{2,z} - 2r_{1,y}^2 t_{2,z} \\
4r_{1,y}r_{1,x}t_{2,z} - 4r_{1,z}r_{1,y}t_{2,x} + 4r_{1,z}r_{1,w}t_{2,z} + 4r_{1,x}r_{1,w}t_{2,x} \\
4r_{1,z}r_{1,x}t_{2,z} - 2r_{1,z}^2 t_{2,x} - 4r_{1,y}r_{1,w}t_{2,z} + 2r_{1,y}^2 t_{2,x} + 2r_{1,x}^2 t_{2,x} - 2r_{1,w}^2 t_{2,x} \\
2r_{2,x}r_{2,y} + 2r_{2,w}r_{2,z} \\
r_{2,y}^2 - r_{2,z}^2 + r_{2,w}^2 - r_{2,x}^2 \\
2r_{2,z}r_{2,y} - 2r_{2,x}r_{2,w}
\end{bmatrix}
$$

$$
\begin{bmatrix}
-2r_{1,x}^2 t_{2,y} + 4r_{1,x}r_{1,y}t_{2,x} - 2r_{1,w}^2 t_{2,y} - 4r_{1,w}r_{1,z}t_{2,x} + 2r_{1,z}^2 t_{2,y} + 2r_{1,y}^2 t_{2,y} \\
-4r_{1,y}r_{1,x}t_{2,y} + 2r_{1,y}^2 t_{2,x} - 4r_{1,z}r_{1,w}t_{2,y} - 2r_{1,z}^2 t_{2,x} + 2r_{1,w}^2 t_{2,x} - 2r_{1,x}^2 t_{2,x} \\
-4r_{1,z}r_{1,x}t_{2,y} + 4r_{1,z}r_{1,y}t_{2,x} + 4r_{1,y}r_{1,w}t_{2,y} + 4r_{1,x}r_{1,w}t_{2,x} \\
2r_{2,x}r_{2,z} - 2r_{2,w}r_{2,y} \\
2r_{2,z}r_{2,y} + 2r_{2,x}r_{2,w} \\
r_{2,z}^2 - r_{2,y}^2 - r_{2,x}^2 + r_{2,w}^2
\end{bmatrix}
\tag{6.5}
$$

$$
J_2 = \begin{bmatrix}
r_{1,x}^2 + r_{1,w}^2 - r_{1,z}^2 - r_{1,y}^2 & 2r_{1,y}r_{1,x} - 2r_{1,z}r_{1,w} & 2r_{1,x}r_{1,z} + 2r_{1,w}r_{1,y} \\
2r_{1,y}r_{1,x} + 2r_{1,z}r_{1,w} & r_{1,y}^2 - r_{1,z}^2 + r_{1,w}^2 - r_{1,x}^2 & 2r_{1,z}r_{1,y} - 2r_{1,x}r_{1,w} \\
2r_{1,x}r_{1,z} - 2r_{1,w}r_{1,y} & 2r_{1,z}r_{1,y} + 2r_{1,x}r_{1,w} & r_{1,z}^2 - r_{1,y}^2 - r_{1,x}^2 + r_{1,w}^2 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
$$

$$
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
r_{2,w}^2 + r_{2,x}^2 + r_{2,y}^2 + r_{2,z}^2 & 0 & 0 \\
0 & r_{2,w}^2 + r_{2,x}^2 + r_{2,y}^2 + r_{2,z}^2 & 0 \\
0 & 0 & r_{2,w}^2 + r_{2,x}^2 + r_{2,y}^2 + r_{2,z}^2
\end{bmatrix}
\tag{6.6}
$$

For the actual DWARF implementation, the C code for the generation of these Jacobians was generated automatically by Maple, in order to avoid typos.

6.1.3 Error Propagation: Pose Inversion

An edge in the spatial relationship graph sometimes may point into the direction opposite to the path, which means that the measurement was made in the coordinate system of the target node. In this case the pose described by the edge has to be inverted and the associated error covariance must be updated accordingly.

Solving equation 6.2 for $\mathbf{x}$ and exchanging the terms $\mathbf{x}$ and $\mathbf{x}_{\text{new}}$ yields the following equation:

$$\mathbf{x}_{\text{new}} = \mathbf{t}_I + \mathbf{e}_{t_I} + r_I e_{r_I} \mathbf{x} e_{r_I}^* r_I^* \tag{6.7}$$

where

$$\begin{aligned} \mathbf{t}_I &= -r^* \mathbf{t} r \\ \mathbf{e}_{t_I} &= r^* \mathbf{t} r - e_r^* r^* (\mathbf{t} + \mathbf{e}_t) r e_r \\ r_I &= r^* \\ e_{r_I} &= r e_r^* r^* \end{aligned}$$

While the error in orientation is simply rotated to a different coordinate system, the translation error, similarly to the inference case, again depends both on the error in rotation and the length of the translation. The consequence is that applying the formula twice results in the same pose, but in a different – worse – error estimate. Unfortunately it is not possible to determine and eliminate the portion of rotation in the translational error afterwards.

The covariance matrix of the inverted pose is computed using the following equation:

$$C_I = J_I C_1 J_I^T \tag{6.8}$$

Where J_I is the Jacobian $J_I = \partial \mathbf{g} / \partial (\mathbf{e}_t, e_r)^T$ of $\mathbf{g}(\mathbf{e}_t, e_r) = (\mathbf{e}_{t_I}, \mathbf{e}_{r_I})^T$ evaluated at $(\mathbf{e}_t, e_r)^T = \mathbf{0}$.

$$J_I = \begin{bmatrix} -r_x^2 - r_w^2 + r_z^2 + r_y^2 & -2r_y r_x - 2r_z r_w & -2r_z r_x + 2r_y r_w \\ -2r_y r_x + 2r_z r_w & -r_y^2 + r_z^2 - r_w^2 + r_x^2 & -2r_z r_y - 2r_x r_w \\ -2r_z r_x - 2r_y r_w & -2r_z r_y + 2r_x r_w & -r_z^2 + r_y^2 + r_x^2 - r_w^2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ -4r_w r_y t_x + 2r_y^2 t_z - 4r_z r_y t_y - 4r_x r_z t_x - 2r_z^2 t_z - 2r_w^2 t_z + 4r_x r_w t_y + 2r_x^2 t_z \\ -4r_w r_z t_x + 4r_y r_z t_z - 2r_z^2 t_y + 4r_x r_y t_x + 2r_y^2 t_y + 4r_x r_w t_z - 2r_x^2 t_y + 2r_w^2 t_y \\ -r_x^2 - r_w^2 + r_z^2 + r_y^2 \\ -2r_y r_x - 2r_z r_w \\ -2r_z r_x + 2r_y r_w \end{bmatrix}$$

$$\begin{bmatrix} -4r_x r_w t_y - 2r_x^2 t_z + 4r_x r_z t_x + 4r_z r_y t_y + 2r_z^2 t_z + 4r_w r_y t_x + 2r_w^2 t_z - 2r_y^2 t_z \\ 0 \\ -4r_z r_w t_y - 4r_x r_z t_z + 2r_z^2 t_x - 2r_x^2 t_x - 4r_y r_x t_y + 2r_y^2 t_x + 4r_y r_w t_z - 2r_w^2 t_x \\ -2r_y r_x + 2r_z r_w \\ -r_y^2 + r_z^2 - r_w^2 + r_x^2 \\ -2r_z r_y - 2r_x r_w \end{bmatrix}$$

$$\begin{bmatrix} -4r_x r_w t_z + 2r_x^2 t_y - 4r_x r_y t_x - 2r_y^2 t_y - 4r_y r_z t_z - 2r_w^2 t_y + 4r_w r_z t_x + 2r_z^2 t_y \\ -4r_y r_w t_z + 4r_y r_x t_y - 2r_y^2 t_x + 2r_x^2 t_x + 4r_x r_z t_z + 4r_z r_w t_y - 2r_z^2 t_x + 2r_w^2 t_x \\ 0 \\ -2r_z r_x - 2r_y r_w \\ -2r_z r_y + 2r_x r_w \\ -r_z^2 + r_y^2 + r_x^2 - r_w^2 \end{bmatrix} \tag{6.9}$$

6.1.4 Converting Rotational Error

The rotational error was defined as a small quaternion that is multiplied to the actual rotation quaternion. This representation has the advantage of this that only a 3×3 covariance matrix is necessary to describe the rotational error. Covariance however was defined to add to the expectation and for some algorithms, e.g. error visualization or optimal estimation, it is necessary to have additive errors. In such a case the 3×3 matrix must be converted to a 4×4 matrix.

The total rotation r', including the multiplicative error e_m, is represented as follows:

$$r' = re_m$$

Using an additive error representation, this becomes

$$r' = r + e_a$$

It obviously follows that for the additive error e_a the following holds:

$$e_a = re_m - r \tag{6.10}$$

Using the error propagation formula 5.36 we get

$$C_a = J_a C_m J_a^T \tag{6.11}$$

where

$$J_a = \left.\frac{\partial e_a}{\partial e_m}\right|_{e_m=1} = \begin{bmatrix} r_w & -r_z & r_y & r_x \\ r_z & r_w & -r_x & r_y \\ -r_y & r_x & r_w & r_z \\ -r_x & -r_y & -r_z & r_w \end{bmatrix} \tag{6.12}$$

The problem with this formula is that the covariance matrix C_m only describes the distributions of the x, y and z components. Intuitively it should be clear that the variance of the w value always must be 0 at the point $[0, 0, 0, 1]$, where the small error quaternion is assumed to be. This is because there is no rotational error that could change the w part and therefore move the quaternion off the hyper sphere. When extending the covariance matrix to a size of 4×4 and setting both the fourth row and column to 0, the matrix becomes singular and therefore cannot be inverted. Inversion however is necessary for the Kalman filter, and at least the fourth diagonal entry must be set to some non-zero value. My experiments however have shown that the actual value does not have a big effect on the result of a Kalman filter estimation. Therefore I simply set the fourth diagonal entry to 1.

The conversion in the other direction is relevant, when the additive error covariance of a Kalman filter is to be inserted back into the ubitrack system.

$$e_m = 1 + r^* e_a \tag{6.13}$$

The associated Jacobian is

$$J_m = \left.\frac{\partial e_m}{\partial e_a}\right|_{e_a=0} = \begin{bmatrix} r_w & r_z & -r_y & -r_x \\ -r_z & r_w & r_x & -r_y \\ r_y & -r_x & r_w & -r_z \\ r_x & r_y & r_z & r_w \end{bmatrix} = J_a^T \tag{6.14}$$

As we are not interested in the distribution of the scalar quaternion part, we can simply drop the last row and column of the resulting covariance matrix.

If the whole 6×6 or 7×7 pose covariance matrix is to be transformed, we can set the upper left part to identity, put J_a or J_m into the lower right and set the rest to 0, as the position is unaffected by this conversion.

6.1.5 Determination of Sensor Errors

Using a covariance matrix to describe sensor errors is a good technique, but unfortunately most off-the-shelf sensors do not return such information with each measurement. In order to set the error statistics, there are three different approaches:

Accuracy values from datasheets usually do not include complete covariance matrices, but at best the average variance or standard deviation for each axis. In this case, the variances can be inserted into the diagonal of the covariance matrix.

Errors consisting of many single errors which are either known or can be estimated more reasonably than the total error may be used to compute the total error distribution. When the formula that computes the final pose from the single erroneous values is known, the error propagation formula can be applied. This approach is suitable for optical trackers, where the total error, after elimination of static distortion, is composed of simple 2D errors caused by sensor noise or the feature extraction [**?**]. Here the relationship between the Jacobian of a function and its inverse may be helpful:

$$\frac{\partial f^-1(\mathbf{y})}{\partial \mathbf{y}} = \left[\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}\right]^{-1} \tag{6.15}$$

This makes it unnecessary to know the exact formula for pose reconstruction from 2D features, as the error covariance can now be determined from the perspective projection, which usually is easier to compute.

Measuring the sensor noise in the static case usually is the best, but most labor-intensive solution. For that, it is necessary to repeat the same measurements many times. From the resulting data, the covariance can be determined using equation 5.29. Ideally, this procedure is repeated at different places, as the accuracy of a sensor rarely is homogeneous throughout its entire working range.

6.1.6 Related Work

Of the papers that I have read, only in the work of Hoff [**?**], error propagation between trackers with multiple coordinate systems plays a major role. Measurement precision is also described using a 6×6 covariance matrix, but rotation is represented with Euler angles. The error propagation formula 5.36 is derived, but no concrete application to the pose representation is given. Also Coelho [**?**] mentions error propagation from tracker to screen coordinates, but does not give any details except a reference to the unscented transform [**?**].

Smith et al. [**?**] use an exponential map representation of orientation uncertainty. I suspect that the resulting covariance matrices are identical to those of my representation, altough I do not have a formal proof for that.

6.2 Reducing Dynamic Errors with Kalman Filters

Dynamic errors are those that are only visible when motion occurs in the scene. The two types, which are most important to this thesis are:

First-Order Dynamic Errors are caused by the end-to-end delay between trackers and the rendering of the augmentation. In order to compensate for this, I use the same approach as Azuma [**?**]. Therefore a Kalman filter is constructed, which estimates the true position and orientation as well as a number of derivations after each measurement. Similar to the Taylor expansion, these derivations are used to predict the pose at a future time, when the rendered image is expected to appear on the screen.

Measurement Simultaneity is implicitly assumed when measurements of multiple sensors are combined. A prominent example is the chaining of measurements along a path in Ubitrack systems. The problem here is, that measurements are only truly simultaneous in setups where sensors are carefully tuned and synchronized. In a typical heterogeneous Ubitrack system however, this is not the case as all sensors have different clocks and update rates. The solution of this problem is again the use of Kalman filters, which are inserted into the dataflow graph directly after each sensor, and compute a valid measurement for a common point in time by prediction or backward projection.

Kalman Filters were explained in the previous chapter in a rather general fashion. This chapter derives the necessary formulas and matrices for applying the extended Kalman filter to dynamic systems consisting of position and orientation. A Kalman filter consists of a time update step and a measurement step. This partitioning is quite useful, because the time update step describes the behavior of objects and can therefore be associated to nodes in the spatial relationship graph, while the measurement step depends on properties of sensors and single measurements. The rest of this section deals with the parameterization I chose for the Kalman filters.

6.2.1 State Vector

The Kalman filter state vector consists of three values t_i for position and another three values for each derivation. The number of derivations is not fixed, but can be chosen freely in the ubitrack system, depending on the characteristics of motion. If e.g. the measured relation between two bodies is known to be constant or changing very slowly, it may suffice to use no derivation at all. In this case the Kalman filter becomes a least squares estimator which averages measurements and therefore eliminates noise.

The orientation quaternion r is stored in another four entries. For dynamic systems, the state vector also contains the angular velocity ω, which is represented by a three-element exponential map. The angular velocity can be seen as the derivative of orientation. The change in orientation, caused by ω can be described with the following formula:

$$r_{k+1} = r_k \otimes e^{\omega \Delta t} \tag{6.16}$$

Exponentiation of quaternions is defined in equation 5.13. The state vector also may contain an arbitrary number of derivations of the angular velocity. Whether ω and its derivations are present also is not fixed and depends on properties of the measured system.

The contents of the state vector are summarized in the following vector:

$$\mathbf{x} = [t_x, t_y, t_z, t'_x, t'_y, t'_z, t''_x, t''_y, t''_z, \ldots, r_x, r_y, r_z, r_w, \omega_x, \omega_y, \omega_z, \omega'_x, \omega'_y, \omega'_z, \ldots]^T \tag{6.17}$$

6.2.2 Time Update

From the previous chapter we know that the Kalman filter time update step has the following equations:

$$\begin{aligned} \hat{\mathbf{x}}^-_{k+1} &= f(\hat{\mathbf{x}}_k, \mathbf{u}_k, 0) \\ P^-_{k+1} &= A_k P_k A_k^T + W_k Q_k W_k^T \end{aligned}$$

As nothing is known in advance about the user's intentions, the system has no deterministic control input $\mathbf{u}_k$.

Position and its derivations are updated using a linear relationship. For each element of $\mathbf{t}$ and its derivations, the following MacLaurin series expansion holds:

$$t^{(l)}_{k+1} = \sum_{i=0}^{n-l} t_k^{(l+i)} \frac{\Delta t^i}{i!} \tag{6.18}$$

where n is the number of used derivations.

Orientation is a bit more complicated, as the relation between orientation and angular velocity is non-linear. In this case the difference caused by angular velocity has to be multiplied to the orientation quaternion. I compute this using the following formula:

$$r_{k+1} = r_k \otimes d_k \tag{6.19}$$

where

$$d_k = e^b, \qquad b = \sum_{i=0}^{m} \omega_k^{(i)} \frac{\Delta t^{i+1}}{(i+1)!}$$

Again, m is the number of derivations of ω and e^b is computed using the exponential map (eq. 5.13). The sum b effectively computes the logarithm of the difference quaternion d_k using the same extrapolation as in formula 6.18.

Prediction Jacobian $A_k = \partial \mathbf{f} / \partial \mathbf{x}$ is necessary in order to correctly propagate the covariance matrix from time t_k to t_{k+1}. Because position and orientation are computed independently, the Jacobian A_k can be decomposed into two independent Jacobians $A_{T,k}$ and $A_{R,k}$:

$$A_k = \begin{bmatrix} A_{T,k} & 0 \\ 0 & A_{R,k} \end{bmatrix} \tag{6.20}$$

$A_{T,k}$ propagates the errors in position and its derivations, and $A_{R,k}$ those of the orientation. Note that if the covariance matrix P_k contains any statistical dependencies between position and orientation, these will be preserved by the propagation. For better readability, I have dropped the index k in the following equations, but this does not mean that the variables are the same for every update step.

$$A_T = \begin{bmatrix}
1 & 0 & 0 & \Delta t & 0 & 0 & \frac{1}{2}\Delta t^2 & 0 & 0 & \cdots \\
0 & 1 & 0 & 0 & \Delta t & 0 & 0 & \frac{1}{2}\Delta t^2 & 0 & \cdots \\
0 & 0 & 1 & 0 & 0 & \Delta t & 0 & 0 & \frac{1}{2}\Delta t^2 & \cdots \\
0 & 0 & 0 & 1 & 0 & 0 & \Delta t & 0 & 0 & \cdots \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & \Delta t & 0 & \cdots \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \Delta t & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{bmatrix}$$

$$A_R = \begin{bmatrix}
d_w & d_z & -d_y & d_x & & & & & & \\
-d_z & d_w & d_x & d_y & & B\Delta t & & & B\frac{1}{2}\Delta t^2 & \cdots \\
d_y & -d_x & d_w & d_z & & & & & & \\
-d_x & -d_y & -d_z & d_w & & & & & & \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & \Delta t & 0 & 0 & \cdots \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \Delta t & 0 & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \Delta t & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{bmatrix}$$

The size of the matrices A_T and A_R of course depends on the number of derivations in position and orientation that are used. The matrix B describes how the orientation depends

on the angular velocity and its derivations. The complete form of B is given in the next equation. For reasons of size, each matrix entry spans over two lines.

$$B =$$

$$\begin{bmatrix}
\frac{1}{2\phi^3}(2b_x r_z b_y \sin(\frac{1}{2}\phi) - 2b_x r_y b_z \sin(\frac{1}{2}\phi) - b_x r_x \sin(\frac{1}{2}\phi) {b_y}^2 + r_w {b_x}^2 \cos(\frac{1}{2}\phi)\phi - r_z b_y \cos(\frac{1}{2}\phi)\phi b_x + \\
r_y b_z \cos(\frac{1}{2}\phi)\phi b_x - b_x r_x \sin(\frac{1}{2}\phi) {b_z}^2 + 2r_w \sin(\frac{1}{2}\phi) {b_z}^2 + 2r_w \sin(\frac{1}{2}\phi) {b_y}^2 - r_x \sin(\frac{1}{2}\phi) {b_x}^3) \\
-\frac{1}{2\phi^3}(-2r_z \sin(\frac{1}{2}\phi) {b_y}^2 - 2r_z \sin(\frac{1}{2}\phi) {b_z}^2 + r_y \sin(\frac{1}{2}\phi) {b_x}^3 + 2b_x q_w b_y \sin(\frac{1}{2}\phi) - r_z {b_x}^2 \cos(\frac{1}{2}\phi)\phi + \\
r_x b_z \cos(\frac{1}{2}\phi)\phi b_x + b_x r_y \sin(\frac{1}{2}\phi) {b_z}^2 - 2b_x r_x b_z \sin(\frac{1}{2}\phi) + b_x r_y \sin(\frac{1}{2}\phi) {b_y}^2 - r_w b_y \cos(\frac{1}{2}\phi)\phi b_x) \\
-\frac{1}{2\phi^3}(b_x r_z \sin(\frac{1}{2}\phi) {b_y}^2 - r_x b_y \cos(\frac{1}{2}\phi)\phi b_x - r_w b_z \cos(\frac{1}{2}\phi)\phi b_x + r_y {b_x}^2 \cos(\frac{1}{2}\phi)\phi + b_x r_z \sin(\frac{1}{2}\phi) {b_z}^2 + \\
2b_x r_x b_y \sin(\frac{1}{2}\phi) + 2b_x q_w b_z \sin(\frac{1}{2}\phi) + r_z \sin(\frac{1}{2}\phi) {b_x}^3 + 2r_y \sin(\frac{1}{2}\phi) {b_y}^2 + 2r_y \sin(\frac{1}{2}\phi) {b_z}^2) \\
-\frac{1}{2\phi^3}(r_x {b_x}^2 \cos(\frac{1}{2}\phi)\phi + r_y b_y \cos(\frac{1}{2}\phi)\phi b_x - 2b_x r_z b_z \sin(\frac{1}{2}\phi) + b_x r_w \sin(\frac{1}{2}\phi) {b_y}^2 - 2b_x r_y b_y \sin(\frac{1}{2}\phi) + \\
b_x r_w \sin(\frac{1}{2}\phi) {b_z}^2 + r_z b_z \cos(\frac{1}{2}\phi)\phi b_x + 2r_x \sin(\frac{1}{2}\phi) {b_y}^2 + 2r_x \sin(\frac{1}{2}\phi) {b_z}^2 + r_w \sin(\frac{1}{2}\phi) {b_x}^3)
\end{bmatrix}$$

$$\begin{bmatrix}
\frac{1}{2\phi^3}(-2{b_x}^2 r_z \sin(\frac{1}{2}\phi) - 2r_z \sin(\frac{1}{2}\phi) {b_z}^2 - r_x \sin(\frac{1}{2}\phi) {b_y}^3 - r_z {b_y}^2 \cos(\frac{1}{2}\phi)\phi + r_y b_z \cos(\frac{1}{2}\phi)\phi b_y - \\
-2r_y b_z \sin(\frac{1}{2}\phi) b_y - r_x \sin(\frac{1}{2}\phi) b_y {b_z}^2 - {b_x}^2 r_x \sin(\frac{1}{2}\phi) b_y - 2b_x r_w b_y \sin(\frac{1}{2}\phi) + r_w b_y \cos(\frac{1}{2}\phi)\phi b_x) \\
-\frac{1}{2\phi^3}(r_y \sin(\frac{1}{2}\phi) {b_y}^3 + 2b_x r_z \sin(\frac{1}{2}\phi) b_y + r_x b_z \cos(\frac{1}{2}\phi)\phi b_y - r_w {b_y}^2 \cos(\frac{1}{2}\phi)\phi - r_z b_x \cos(\frac{1}{2}\phi)\phi b_y - \\
2r_x b_z \sin(\frac{1}{2}\phi) b_y + {b_x}^2 r_y \sin(\frac{1}{2}\phi) b_y + r_y \sin(\frac{1}{2}\phi) b_y {b_z}^2 - 2q_w \sin(\frac{1}{2}\phi) {b_z}^2 - 2{b_x}^2 r_w \sin(\frac{1}{2}\phi)) \\
\frac{1}{2\phi^3}(r_w b_z \cos(\frac{1}{2}\phi)\phi b_y - r_y b_x \cos(\frac{1}{2}\phi)\phi b_y + r_x {b_y}^2 \cos(\frac{1}{2}\phi)\phi - {b_x}^2 r_z \sin(\frac{1}{2}\phi) b_y - r_z \sin(\frac{1}{2}\phi) b_y {b_z}^2 + \\
2b_x r_y \sin(\frac{1}{2}\phi) b_y - 2r_w b_z \sin(\frac{1}{2}\phi) b_y + 2{b_x}^2 r_x \sin(\frac{1}{2}\phi) - r_z \sin(\frac{1}{2}\phi) {b_y}^3 + 2r_x \sin(\frac{1}{2}\phi) {b_z}^2) \\
-\frac{1}{2\phi^3}(r_w \sin(\frac{1}{2}\phi) b_y {b_z}^2 + r_w \sin(\frac{1}{2}\phi) {b_y}^3 + 2r_y \sin(\frac{1}{2}\phi) {b_z}^2 + 2{b_x}^2 r_y \sin(\frac{1}{2}\phi) + r_y {b_y}^2 \cos(\frac{1}{2}\phi)\phi \\
-2r_z b_z \sin(\frac{1}{2}\phi) b_y + r_z b_z \cos(\frac{1}{2}\phi)\phi b_y + r_x b_x \cos(\frac{1}{2}\phi)\phi b_y + {b_x}^2 r_w \sin(\frac{1}{2}\phi) b_y - 2b_x r_x \sin(\frac{1}{2}\phi) b_y)
\end{bmatrix}$$

$$\begin{bmatrix}
\frac{51}{2\phi^3}(2r_y \sin(\frac{1}{2}\phi) {b_y}^2 + 2{b_x}^2 r_y \sin(\frac{1}{2}\phi) - r_z b_z \cos(\frac{1}{2}\phi)\phi b_y + r_y {b_z}^2 \cos(\frac{1}{2}\phi)\phi + 2r_z b_z \sin(\frac{1}{2}\phi) b_y - \\
{b_x}^2 r_x \sin(\frac{1}{2}\phi) b_z + r_w b_x \cos(\frac{1}{2}\phi)\phi b_z - r_x \sin(\frac{1}{2}\phi) {b_z}^3 - 2b_x r_w \sin(\frac{1}{2}\phi) b_z - r_x \sin(\frac{1}{2}\phi) b_z {b_y}^2) \\
-\frac{1}{2\phi^3}(r_x {b_z}^2 \cos(\frac{1}{2}\phi)\phi - r_z b_x \cos(\frac{1}{2}\phi)\phi b_z + {b_x}^2 r_y \sin(\frac{1}{2}\phi) b_z - r_w b_z \cos(\frac{1}{2}\phi)\phi b_y + 2b_x r_z \sin(\frac{1}{2}\phi) b_z + \\
r_y \sin(\frac{1}{2}\phi) b_z {b_y}^2 + 2r_w b_z \sin(\frac{1}{2}\phi) b_y + 2{b_x}^2 r_x \sin(\frac{1}{2}\phi) + 2r_x \sin(\frac{1}{2}\phi) {b_y}^2 + r_y \sin(\frac{1}{2}\phi) {b_z}^3) \\
\frac{1}{2\phi^3}(-2r_x b_y \sin(\frac{1}{2}\phi) b_z - r_z \sin(\frac{1}{2}\phi) b_z {b_y}^2 - {b_x}^2 r_z \sin(\frac{1}{2}\phi) b_z + 2b_x r_y \sin(\frac{1}{2}\phi) b_z + r_x b_y \cos(\frac{1}{2}\phi)\phi b_z + \\
r_w {b_z}^2 \cos(\frac{1}{2}\phi)\phi - r_y b_x \cos(\frac{1}{2}\phi)\phi b_z + 2r_w \sin(\frac{1}{2}\phi) {b_y}^2 + 2{b_x}^2 r_w \sin(\frac{1}{2}\phi) - r_z \sin(\frac{1}{2}\phi) {b_z}^3) \\
-\frac{1}{2\phi^3}(-2r_y b_z \sin(\frac{1}{2}\phi) b_y + r_w \sin(\frac{1}{2}\phi) {b_z}^3 + 2r_z \sin(\frac{1}{2}\phi) {b_y}^2 + 2{b_x}^2 r_z \sin(\frac{1}{2}\phi) + r_x b_x \cos(\frac{1}{2}\phi)\phi b_z + \\
r_z {b_z}^2 \cos(\frac{1}{2}\phi)\phi + r_w \sin(\frac{1}{2}\phi) b_z {b_y}^2 - 2b_x r_x \sin(\frac{1}{2}\phi) b_z + r_y b_z \cos(\frac{1}{2}\phi)\phi b_y + {b_x}^2 r_w \sin(\frac{1}{2}\phi) b_z)
\end{bmatrix}$$

$\phi = |b|$ is the angle by which the orientation is rotated. From the above equation we can see that its not directly possible to compute B when ϕ is zero. Therefore the following matrix $\tilde{B}$

gives the limit of B as the vector b goes to zero.

$$\tilde{B} = \lim_{b \to 0} B = \begin{bmatrix} \frac{1}{2}r_w & -\frac{1}{2}r_z & \frac{1}{2}r_y \\ \frac{1}{2}r_z & \frac{1}{2}r_w & -\frac{1}{2}r_x \\ -\frac{1}{2}r_y & \frac{1}{2}r_x & \frac{1}{2}r_w \\ -\frac{1}{2}r_x & -\frac{1}{2}r_y & -\frac{1}{2}r_z \end{bmatrix}$$

When the difference between successive orientations is small, $\tilde{B}$ can be used as an approximation for B. This may be the case when angular velocity is small or when a high update rate is used.

Process Noise determines the speed by which the entries of the covariance matrix decay with time. A low process noise causes the Kalman filter to give more weight to the estimated state vector and measurements are trusted less. Previous work [**?**, **?**] has shown that it is sufficient to use only the diagonal entries of the process noise covariance matrix and set the rest to 0. Therefore we set $W_k = I$. Furthermore, the uncertainty should increase as with the length of the prediction interval. So I use the following process noise covariance matrix:

$$Q_k = \begin{bmatrix} (q_1 \Delta t)^2 & & 0 \\ & \ddots & \\ 0 & & (q_n \Delta t)^2 \end{bmatrix} \tag{6.21}$$

where n is the length of the state vector. The exact values of the elements q_i depend on the characteristics of the tracked motion. In order to reduce the degrees of freedom, especially when an optimization algorithm is used, it may make sense to use the same value for the x, y, z and w components of each derivation. This would imply the assumption that the motion has the same qualities in each direction.

6.2.3 Measurement Update

The measurement update step integrates new measurements into the state vector. How this is done depends on the type of sensor that is used. As a reminder, the measurement update equations of the extended Kalman filter have the following form:

$$\begin{aligned} K_k &= P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \\ \hat{\mathbf{x}}_k &= \hat{\mathbf{x}}_k^- + K(\mathbf{z}_k - h(\hat{\mathbf{x}}_k^-, 0)) \\ P_k &= (I - K_k H_k) P_k^- \end{aligned}$$

Measurement Function essentially predicts a new measurement from the estimated state vector. For sensors that directly measure position or one of its derivations, this is easy. The respective measurement function simply has to extract the relevant part from the state vector. The same holds for the derivations of the orientation, as long as the representation is the same.

The only difference is the integration of measurements that directly include an orientation quaternion. As described by Azuma [**?**], the measurement function consists of a quaternion normalization:

$$\tilde{z}_r = h_r(x_k) = \text{Normalize}(r_k) = \frac{r_k}{||r_k||} \tag{6.22}$$

Measurement Jacobian H_k is the derivation of the measurement function with respect to the state vector $\mathbf{x}$. For operations that simply extract components, the Jacobian is the identity. In case of the quaternion normalization, $H_R = \frac{\partial \text{Normalize}(r)}{\partial r}$ is

$$H_R = \begin{bmatrix} \frac{D-r_x^2}{LD} & \frac{-r_x r_y}{LD} & \frac{-r_x r_z}{LD} & \frac{-r_x r_w}{LD} \\ \frac{-r_y r_x}{LD} & \frac{D-r_y^2}{LD} & \frac{-r_y r_z}{LD} & \frac{-r_y r_w}{LD} \\ \frac{-r_z r_x}{LD} & \frac{-r_z r_y}{LD} & \frac{D-r_z^2}{LD} & \frac{-r_z r_w}{LD} \\ \frac{-r_w r_x}{LD} & \frac{-r_w r_y}{LD} & \frac{-r_w r_z}{LD} & \frac{D-r_w^2}{LD} \end{bmatrix} \tag{6.23}$$

where

$$\begin{aligned} D &= r_x^2 + r_y^2 + r_z^2 + r_w^2 \\ L &= \sqrt{D} = ||r|| \end{aligned}$$

If a sensor measures multiple state vector components simultaneously, the Jacobian can be composed of the respective submatrices. The following example gives the measurement function and Jacobian for a sensor that returns a full 6DOF pose:

$$\begin{aligned} h(x_k) &= [t_k, r_k]^T \\ H_k &= \begin{bmatrix} I & 0 & & \cdots & & & 0 \\ 0 & \cdots & 0 & H_R & 0 & \cdots & 0 \end{bmatrix} \end{aligned}$$

Again, the number of zero entries depends on the exact number of derivatives in the state vector.

Measurement Noise Covariance Matrix is taken directly from the measurement. The only obstacle lies in the multiplicative nature of the 6×6 covariance matrix. Therefore it has to be transformed into a 7×7 matrix, where the rotational error adds to the quaternion. This is done by extending the covariance matrix, and using the matrix J_a derived in eq. 6.12 for transforming the orientation part:

$$R_k = \begin{bmatrix} & & & 0 \\ & C_k & & \vdots \\ & & & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix} \tag{6.24}$$

$$V_k = \begin{bmatrix} I & 0 \\ 0 & J_a \end{bmatrix} \tag{6.25}$$

Normalization of the rotation quaternion in the state vector is done after the measurement step to assure that the quaternion stays on the unit sphere. This is not part of the standard Kalman filter, but it has been used by other researchers before [**?**].

6.2.4 Related Work

Many researchers have used Kalman filters before in AR or VR systems. Some examples are [**?**, **?**, **?**, **?**, **?**, **?**]. The Kalman filter is either applied to the output of pre-built sensors [**?**, **?**, **?**] or it is used as an integral part of a vision-based tracking system [**?**, **?**, **?**].

The biggest difference lies in the representation of orientation that is used. [**?**] and [**?**] use quaternions, but [**?**] has distinct Kalman filters for orientation and each component of position. [**?**] uses Euler angles and [**?**] another 3-element representation. The system described by [**?**] does not include the absolute orientation in the state vector, but only an incremental rotation, which is used to update an external quaternion and is then reset to 0 after each measurement. A completely different approach is described in [**?**], where a so-called complimentary Kalman filter is used that does not estimate the absolute orientation of the user, but rather the offset of an inertial sensor, which is then subtracted from the tracker output.

6.3 System Parameters

All remaining free parameters are set in the actual ubitrack system. I separate parameters that describe the type of motion from those that determine sensor measurements.

6.3.1 Motion Model

The motion model includes all parameters that determine the characteristics of motion between two bodies. These parameters are independent from the sensing technique that is used in the spatial relationship graph. The motion model consists of the following three values:

The number of position derivations n_{dp} determines how many derivations of position are included in the state vector. When this value is zero, a static, or very slowly changing relationship is assumed, depending on the entries of the process noise vector. A value of 2 describes a constant acceleration model.

The number of orientation derivations n_{do} describes the same for orientation. A value of 1 indicates constant angular velocity. The values of n_{dp} and n_{do} together determine the size of the state vector, which is $7 + 3(n_{dp} + n_{do})$.

The process noise vector $\mathbf{q}$ determines the diagonal values of Q_k, as defined in equation 6.21. Therefore it has the same number of entries as the state vector.

From these parameters, I dynamically construct the Kalman filter state vector and time update equations, as described above.

6.3.2 Measurement Attributes

The measurement attributes describe the type of the sensor as well as properties of each measurement.

The position derivation l_p describes which quantity is actually measured by the tracker. The value is 0 for absolute sensors and 2 for inertial trackers that measure the acceleration.

The orientation derivation l_o describes the same for the orientation. In this case, for inertial sensors, l_o must be 1, as inertial orientation sensors measure angular velocity, which can be treated as the first derivation of orientation.

The measurement vector $\mathbf{z}_k$ contains the actual measurements, i.e. position and orientation or the respective derivations, if l_p or l_o are greater than zero.

The measurement covariance matrix C_k describes the covariance of the measurement. Usually this is a 6×6 matrix, as defined by the error model.

These four parameters determine the measurement update step of the Kalman filter. The first two describe how the measurement function h and its Jacobian H are constructed.

6.4 Evaluation Functions

The evaluation function is used in Ubitrack for computing how well a path in the spatial relationship graph resembles the requirements of an application. As the covariance matrix describes the accuracy of a measurement, its evaluation is a natural choice for the evaluation function. The approach described here does not take into account other measurement attributes such as update rate or latency, and is therefore only suitable as one part of a real evaluation function. For an evaluation function, it is necessary to compute a single scalar value out of the covariance. A good choice for this is evaluating the trace of the covariance matrix, because the trace of a matrix is the same as the sum of its eigenvalues [**?**], and as described in the last chapter, the eigenvalues are the squares of the lengths of the axes of the confidence ellipsoid for $\sigma = 1$. Therefore the root of the trace is proportional to the length of the diagonal of the cuboid around the ellipsoid. If the application is not interested in the total accuracy, but wants to weight directions differently, e.g. in Augmented Reality, where uncertainties perpendicular to the viewing direction are more crucial, it can supply a weighting matrix A. This results in the following evaluation function:

$$e_p = \sqrt{\text{trace}(AC_pA^T)} \tag{6.26}$$

Normally, it should not be necessary to repeat the evaluation and the graph search for each measurement, as the quality attributes are assumed to change with lower frequency as the measurements. Therefore it is possible to use compute a sliding or exponential average over both measurements and covariance matrix, and use this to update the attributes of the Spatial Relationship Graph at a much lower update rate. For computing the attributes of a whole

path, it is necessary to apply the error propagation formulas for inference and/or inversion on these averaged measurements and covariance matrices. Even some sensor fusion cases may be handled by assuming the static case and using the (recursive) optimal estimation formula for computing the resulting covariance.

7 DWARF Ubitrack Data Flow

The previous chapter explains the mathematics used in the data flow computations of the Ubitrack system. This chapter deals with the software implementation aspects of the DWARF data flow implementation.

7.1 Requirements

Before implementing any system, it is always useful to think about what it is supposed to do and what not. Bruegge [**?**] divides such requirements into functional, non-functional and pseudo requirements.

7.1.1 Functional Requirements

> "Functional requirements describe the interactions between the system and its environment independent of its implementation." [**?**]

Computation of inferred measurements Obviously, the data flow graph part of the Ubitrack system must be able to compute inferred measurements from the available real measurements. For that, measurements must be inverted, chained, predicted and fused, and arbitrary combinations of these operations should be possible. In addition, all operations must modify the sensor error statistics accordingly, in order to provide quality information to the application.

Tracker Abstraction The data flow graph must provide uniform interfaces both to the trackers and to the application in order to make it device- and application-independent. This requirement corresponds to the "Transparency" design goal of the Ubitrack system from section 1.4.1.

Dynamic configurability by the UMA The central component of the DWARF Ubitrack system is the UMA, which holds a distributed representation of the Spatial Relationship Graph , receives application queries and performs the distributed path search. In order to deliver the requested measurements to the application, the data flow graph is configured by the UMA.

7.1.2 Nonfunctional Requirements

> "Nonfunctional requirements describe user-visible aspects of the system that are not directly related with the functional behavior of the system." [**?**]

Performance An critical point in the design of augmented reality systems is the end-to-end system delay. While it is possible to reduce its negative effects to some degree by prediction, the reduction of this delay still should be the a major design goal. Therefore the communication overhead between the data flow components as well as their CPU load should be as low as possible.

Extensibility The first Ubitrack implementations are not meant to be fully working consumer-ready applications, but rather research sandboxes for experimenting with new methods for sensor fusion, distributed tracking, etc. Therefore the data flow graph should be extensible to allows easy addition of new components, and the implementation should focus on small modules that can later be reused for unforeseen purposes.

7.1.3 Pseudo Requirements

> "Pseudo requirements are requirements imposed by the client that restrict the implementation of the system." [**?**]

DWARF The components of the data flow graph should be realized as DWARF services to take advantage of the existing distributed software infrastructure.

7.2 Data Flow Implementation Concepts

This section discusses the general concepts of the data flow graph implementation, which affect all of its components.

7.2.1 Data Structures

In the current data flow implementation, two types of data must be described and communicated among components.

Motion Models

Motion models describe the type of motion that is expected between two objects of the spatial relationship graph. These are used to allow different Kalman filters to be constructed for each relationship.

Motion models are currently contained in the `UTMotionModel` attribute of the sensor service. It should be clear that this can only be a temporary solution, as it contradicts the

principle of separation of sensor and motion information. A possible solution to the problem, by adding motion information to the spatial relationship graph, is outlined in the future work section.

The `UTMotionModel` is a string containing at least two numbers, n_{dp} and n_{do}, which describe the number of derivations of position and orientation in the Kalman filter state vector. If one of the two entries is -1, position or orientation will not be included at all. The meaning of the motion model entries is explained in more detail in section 6.3.1.

$$\texttt{UTMotionModel} = "n_{dp} n_{do} q_1 \dots q_n"$$

Measurements

Each measurement is described using the `PoseData` data structure, which was specified at the beginning of the Ubitrack project in a collaborative work by all project members. The contents of the structure are given in table 7.1.

Field	Description
source	This string contains the object identifier of the source coordinate system in the Spatial Relationship Graph . This is usually the same as the sensor's.
target	This string contains the target's object identifier.
position	The 3D position of the object is stored in an array of double floats in Cartesian coordinates.
orientation	The orientation of an object is delivered in form of a *quaternion*. For a description of quaternions, see section 5.1.3.
timestamp	A data structure containing the absolute timestamp of the measurement. It consists of a `seconds` field, which describes the seconds since January, 1st 1970, 00:00 (standard UNIX time format), and a `microseconds` field, provided for higher resolution.
confidence	A statistical value giving the probability that this measurement exists. This field is relevant, when a tracker is unsure about the identity or even existence of a detected locatable.
covariance	A 6x6 covariance matrix parameterizing the Gaussian error distribution of the measured position and orientation (see section 6.1.1).
Flag fields	Flags that indicate which fields of this structure contain valid data.

Table 7.1: Data fields which are stored in a PoseData structure

In the current implementation, PoseData structures are either sent as events, using the *Asynchronous Push* mechanism, or requested by a *Synchronous Pull* method call. For further explanations, see section 2.4.3. The *Asynchronous Pull* mechanism is currently unused.

In addition to the values contained in the `PoseData` structure, also sensor characteristics that do not change with every measurement, are required. In our case these are the derivations of position and orientation that is actually measured by each sensor. These two

numbers are described by the `UTMeasurementDerivations` of the sensor service, which is a string containing two numbers:

$$\texttt{UTMeasurementDerivations} = "l_p, l_o"$$

For an explanation, see section 6.3.2. If one number is negative, the sensor does not measure this quantity at all.

In order to stay compatible with existing DWARF applications, the measurements always contain absolute quantities. For inertial sensors, the values contained in the `PoseData` structure always are integrated by the sensor. This also has the advantage of higher precision and lower drift, as the integration in the sensor usually can be performed at a much higher accuracy and update rate.

7.2.2 Timing Issues

In an ideal AR system, all trackers make measurements simultaneously, these are immediately passed through the data flow components, and given to the renderer. As soon as the renderer is finished, a new video scan-out starts. In such a system, no time-critical information would stay in a buffer longer than necessary, resulting in a low end-to-end system delay. Most high-performance AR systems have been built with this ideal setup in mind, which requires a tight synchronization between trackers, renderer and screen. That however usually leads to difficult to understand monolithic applications and real-time operating systems.

When building an AR system using off-the-shelf components, neither the tracker measurements nor the video scan-out can be controlled, as both components are driven by their internal clocks. Therefore tracking data always is buffered in some form, maybe as a 6D pose or as a rendered image buffer, waiting to be displayed on the screen.

For our Ubitrack data flow graph, we basically only have the choice, where to buffer measurements. If we use an asynchronous push architecture, where new measurements are immediately pushed through all dataflow components to the renderer. Depending on render speed and sensor update rate, the computed measurement will probably land in a buffer until the renderer is ready to compute the next image buffer. If we use a synchronous pull architecture, the renderer demands a new measurement from the data flow graph, which again asks the trackers. Most likely these do not make a new measurement, especially if it's an optical tracker, but instead return the most recent one from a buffer.

As neither of the two approaches gives a real advantage regarding the end-to-end system delay, other factors can be considered. If we look at the number of computations that are performed, it depends on the concrete system. In case of the event-based system, each new measurement triggers a recomputation of the whole data flow graph. So, if the system has more measurements than image renderings – and we hope it has – the pull architecture has a slight advantage here. Another advantage of the pull method is that the renderer can give an estimate of the time that the image will be displayed on the screen and we can use prediction to compute an estimate this is better than what we get by simply using the last measurement.

Finally, the *Measurement Simultaneity* problem cannot be solved using an event-based architecture unless all sensors are synchronized by some external mechanism. For the pull

approach, prediction or interpolation of old sensor data can at least give the impression of simultaneous measurements.

When measurements from multiple unsynchronized trackers are to be combined, the question remains whether all sensors inputs are extrapolated to the time of the most recent measurement or even to the current time, or whether we use interpolation for computing a measurement valid at the time of the last measurement by the slowest sensor. This depends on what the data is used for. In case of real-time AR systems, we certainly prefer the extrapolation method to give the user the impression of a zero-lag system. If the data however is used for estimating a static relationship or for non-time critical ubicomp applications, we can afford to interpolate at the advantage of higher precision.

For determining how old measurements are, we need precise time stamps. Therefore all involved computer systems need synchronized clocks. If we use a time synchronization protocol like NTP (Network Time Protocol) on a cable-based local-area network to update the computer clocks frequently, we should be able to get precise synchronization in the order of milliseconds. Alternatively, a GPS receiver could be used, which maintains an estimate of the global GPS time. However, the delay caused by the usual low-speed serial connection might be a problem. In any case, the clock synchronization problem should be further investigated.

7.2.3 Performance Considerations

Usually each DWARF service runs in its own process space. This slows down communication between services, as an expensive process context switch is necessary for each message. So far this hasn't been a big problem, because tracking was passed through at most one additional component. In the Ubitrack data flow however, we can have chains of five or more components, and context switching may become an issue.

As process switching is the problem, the solution is obvious: Keep all tracking components in one process. In this case all CORBA method invocations are optimized by the ORB to local function calls, which are very cheap, compared to inter-process communication.

An implementation of this concept puts all the data flow components into one executable. The first process that is started then registers the `SvcLoad` interface for all other data flow services. Subsequent instantiations by the service manager simply cause the creation of a new object instead of a new process. The advantage of this approach is that the instantiation and selection of communication partners is still controlled by the service manager and it is also possible to insert external components into the data flow graph, at the expense of higher communication overhead.

The described method however is only suitable for method invocations. Event based communication currently is dispatched by an external daemon and therefore process switching always occurs. Therefore we should try to use the Synchronous Pull mechanism as often as possible for communication between the data flow graph components.

If the communication between the data flow graph and the application proves to become a problem, the data flow components could be linked as a library into the application's process space. As suggested by Asa MacWilliams, it may even be possible to link the notification service libraries into the service process, enabling fast event-based communication. Such an

approach as well as the real causes for delays in DWARF-based data flow implementations, should be investigated in further works.

7.3 Data Flow Components

This section describes the functionality and implementation details of each of the data flow graph components.

7.3.1 Inference

Inference services are responsible for combining multiple measurements along a path, which is one of the core principles of the Ubitrack framework. The mathematics behind this is explained in section 6.1.2. An inference service instance may have an arbitrary number of input `PoseData` needs and has one ability for providing the resulting computation. As all input measurements must be made simultaneously, the inference only supports the `UTPoseDataSyncPull` protocol for both input and output. If the preceding component, e.g. a sensor, only provides event-based communication, a KalmanFilter component must be inserted in between. Inference components can be instantiated for partial paths, e.g. if there is a sensor fusion step in between.

The mathematics of the Inference service is contained in the `EasyPoseData::Product` method, which now is part of the DWARF library. This allows the reuse of the functionality in other services. The method is documented in more detail in the Doxygen source code comments.

7.3.2 KalmanFilter

KalmanFilter services mainly provide a continuous-time `UTPoseDataSyncPull` protocol to those leaf components that only make measurements at discrete-times by sending events over the `UTPoseDataAsyncPush` protocol. The Kalman Filter maintains an internal state of the relation, usually including position, orientation and a number of derivations. This state is used to compute measurements at past or future moments as they are demanded by other components in the data flow graph. The filter also maintains a queue of old measurements which are interpolated when a measurement is requested at a time earlier that what can reasonably be derived from the current internal state.

A Kalman Filter service instance can handle an arbitrary number of input needs that all measure the same relation. In this case data fusion is performed by integrating measurements from all sensors into the internal state. This way, also trackers with different characteristics can be combined, e.g. inertial and absolute sensors. However, how inertial sensors are described in the spatial relationship graph remains an open problem (see section 10.3.3). As inertial sensors always transmit integrated absolute measurements, a differentiator is included in the KalmanFilter service for generating the appropriate measurement vectors is such a case.

Usually the Kalman Filter receives new measurements using the `UTPoseDataAsyncPush` protocol and each incoming measurement is immediately

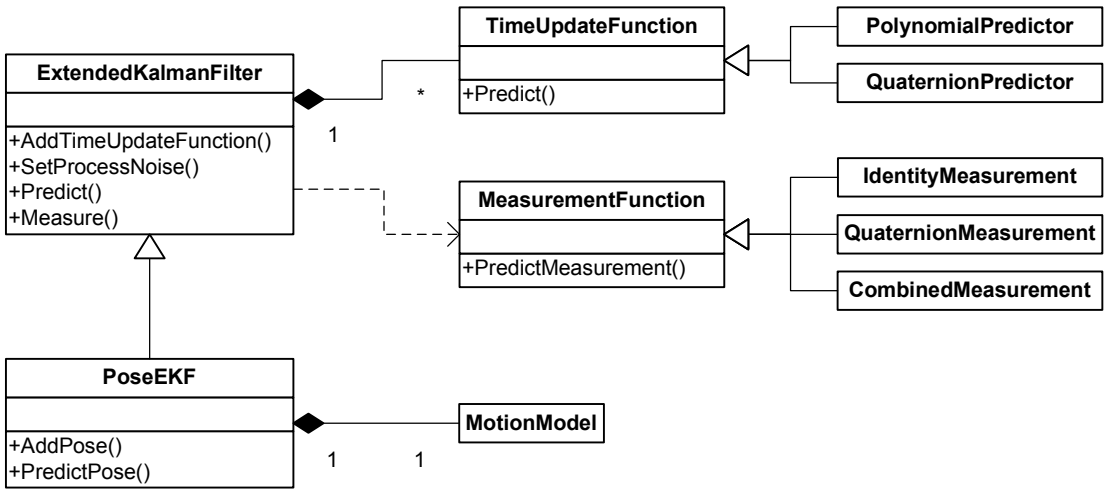


Figure 7.1: UML diagram of the classes involved in the dynamic construction of Kalman filters

integrated into the internal state. For cases where the filter is used for sensor fusion inside a more complex data flow graph, also the `UTPoseDataSyncPull` protocol is supported. The filter output can only be requested via `UTPoseDataSyncPull`. Such setups however should be avoided, as the integration of new measurements then is only triggered when a filter output is requested.

When the internal Kalman Filter state is updated by a new measurement, some assumptions about the type of motion are required to be able to decide how to weight the measurement with respect to the internal state. If more weight is given to the internal state, a bigger portion of the incoming measurements is considered noise, resulting in a tighter filtering. The motion model, which describes this behavior, is read from the `UTMotionModel` attribute of the sensor. When no motion model is specified, a default model using first and second derivations of both position and orientation is used.

Implementation Notes

The KalmanFilter service is implemented using a general-purpose Kalman filter class hierarchy with dynamic state vector generation. I give some documentation here, because these classes could be reused in other projects, e.g. for autocalibration. A class diagram is given in figure 7.1. A more detailed description including all methods and parameters is found in the Doxygen documentation included in the header files.

ExtendedKalmanFilter is the main class of the general Kalman filter implementation. It contains the state vector and the process noise. The time update step is generated by adding one or more `TimeUpdateFunctions` using the `AddTimeUpdateFunction` method. For computing an extrapolated measurement, `Predict` is used. For each measurement, a `MeasurementUpdateFunction` must be supplied to the `Measure` method. A call to `SetProcessNoise` changes the motion model.

TimeUpdateFunction is a virtual base class for all functions that describe how the state vector is advanced. The general idea is that each `TimeUpdateFunction` (TUF) operates only on a subset of the state vector contents. Therefore one TUF implementation can be reused for different state vector parts. The subset on which the TUF operates is specified when calling the `AddTimeUpdateFunction` method of the `ExtendedKalmanFilter`.

The only method which must be implemented by derived classes is `Predict`. This returns both the updated state vector parts and the associated Jacobian A (see 5.3 for an explanation of the Kalman filter).

MeasurementUpdateFunction predicts a new measurement out of a subset of the state vector contents and return both the predicted measurement and the measurement Jacobian H.

PolynomialPredictor is an implementation of the `TimeUpdateFunction`. It computes a polynomial prediction is the style of the MacLaurin expansion:

$$x_{k+1} = \sum_{i=0}^{n} \frac{x_k^{(i)}}{i!} \Delta t$$

QuaternionPredictor works similar to the `PolynomialPredictor`, but creates a quaternion. For the mathematical details, see section 6.2.2.

IdentityMeasurement is an implementation of the `MeasurementUpdateFunction` which simply extracts a part of the state vector. Therefore the Jacobian always is the identity.

QuaternionMeasurement performs a quaternion normalization. For details of the mathematics and the computation of the Jacobian, see section 6.2.3.

CombinedMeasurement combines multiple `MeasurementUpdateFunction`s. This is relevant when a sensor measures both position and orientation simultaneously and returns a combined covariance matrix.

PoseEKF provides a simpler interface for Kalman filters that handle `PoseData`. New measurements can be integrated using the `AddPose` method. For computing an predicted measurements, the `PredictPose` method can be called.

7.3.3 MeasurementInverter

`MeasurementInverter` services invert measurements when edges in a path point in the wrong direction. They have exactly one input and one output. Two different implementations exist for the `UTPoseDataSyncPull` and `UTPoseDataAsyncPush` protocols. Usually the pull version is inserted after a `KalmanFilter`. The underlying mathematics are explained in section 6.1.3 and implemented in the `EasyPoseData::inverse` method, which is part of the DWARF libraries.

7.3.4 MeasurementSampler

`MeasurementSampler` services are inserted at the end of a data flow graph if an application prefers to be notified of new measurements at a fixed update rate via the `UTPoseDataAsyncPush` protocol. The service has a `UTPoseDataSyncPull` need, usually connected to an inference or Kalman filter, and an `UTPoseDataAsyncPush` ability. The update rate at which events are generates is specified by the `UTUpdateRate` attribute at the receiver's side.

A MeasurementSampler also generates the correct timestamps for prediction. This behavior can by controlled by the receiver using the `UTTimeOffset` attribute, which simply specifies the milliseconds to predict into the future, but also negative values are possible.

7.4 Data Flow Graph Construction Rules

This section explains how the components must be instantiated in a data flow graph in order to compute something useful. It also shows some problems that result. The data flow graph construction usually starts with a path or a subgraph that results from a distributed path search (see [**?**] for details). From that an optimal data flow graph must be determined that fulfils the following criteria:

Correct Computation is the most obvious criterion for the data flow graph. If the path search results in a single path, then the measurements simply are chained by an `Inference` component. Edges that point in the wrong direction are prepended by a `MeasurementInverter`. If the search result is a whole subgraph with circles, then subpaths that can be expressed by other subpaths are chained by an `Inference` and all equivalent subpaths must be combined by inserting a `KalmanFilter` component which performs the fusion.

Optimal Computation is not guaranteed by the previous criterion. If for example all edges of a path point in the wrong direction, it would be more efficient to invert just the result of the `Inference` instead of the original measurements. Also multiple inversions of the same measurement in more complex scenarios lead to wrong estimations of the positional error (see sec. 6.1.3). In order to reduce such problems, more advanced methods, similar to the optimization of mathematical formulas have to be developed.

Communication Protocols may also differ in the data flow graph. Most sensors only provide the `AsynchronousPush` protocol, while the inference components require the use of the `SynchronousPull` mechanism. Therefore, additional `KalmanFilter` components must be inserted before such sensors. If the application again requires `AsynchronousPush` from an `Inference`, a `MeasurementSampler` must be added to the end of the data flow.

Resource Allocation is important in distributed systems. Usually the computation of the whole data flow graph on the client system should provide the lowest lag, as no measurements are transported to intermediate hosts. However, if the client is a mobile system with insufficient resources, the data flow or some parts of it may be allocated on different computers or common sub-computations shared.

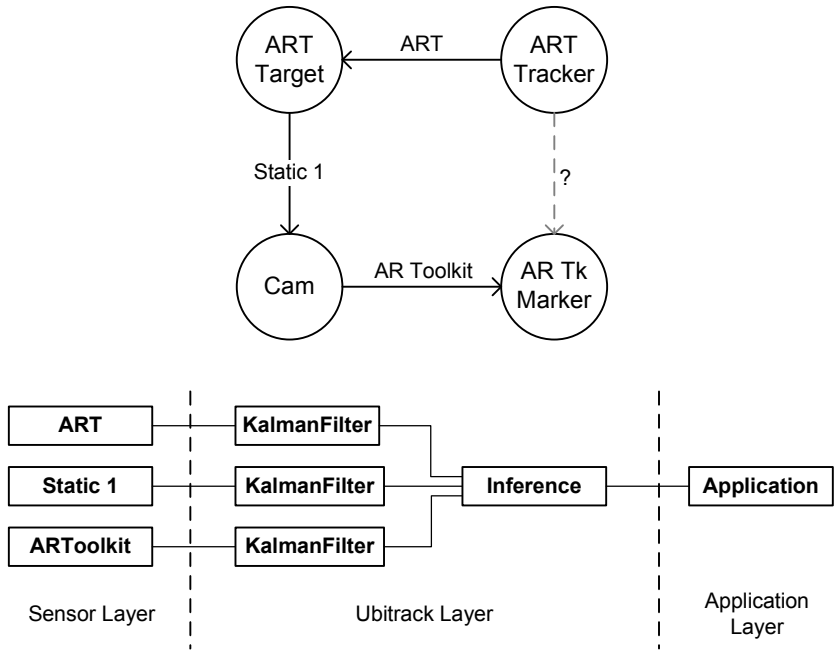


Figure 7.2: Spatial relationship and data flow graphs of the demo setup.

7.4.1 Example Dataflows

This section gives some example dataflow graphs to illustrate the construction rules.

Demo Setup from Introduction

Figure 7.2 shows how the data flow components are used in order to compute the position of the virtual sheep in the example from section 2.3. The `KalmanFilter` after the static tracker merely serves as a buffer to reduce the communication overhead that would result from using the `SynchronousPull` interface. The computational overhead introduced by the Kalman filter is not crucial, as the static tracking service only has a low update rate and a state vector with only seven entries is used, because no derivations are required. Alternatively, a special buffer service could be written for this case.

Estimating the Static Edge in the Demo Setup

In figure 7.3 we want to estimate the static edge between the ART target and the camera. For that, the AR Toolkit marker is placed at a known location which either was measured my hand or by using another, already calibrated tracker. For that, the ART and AR Toolkit edges must be inverted and a `KalmanFilter` component with a static motion model is inserted at the end of the data flow graph, serving as a least-squares estimator.

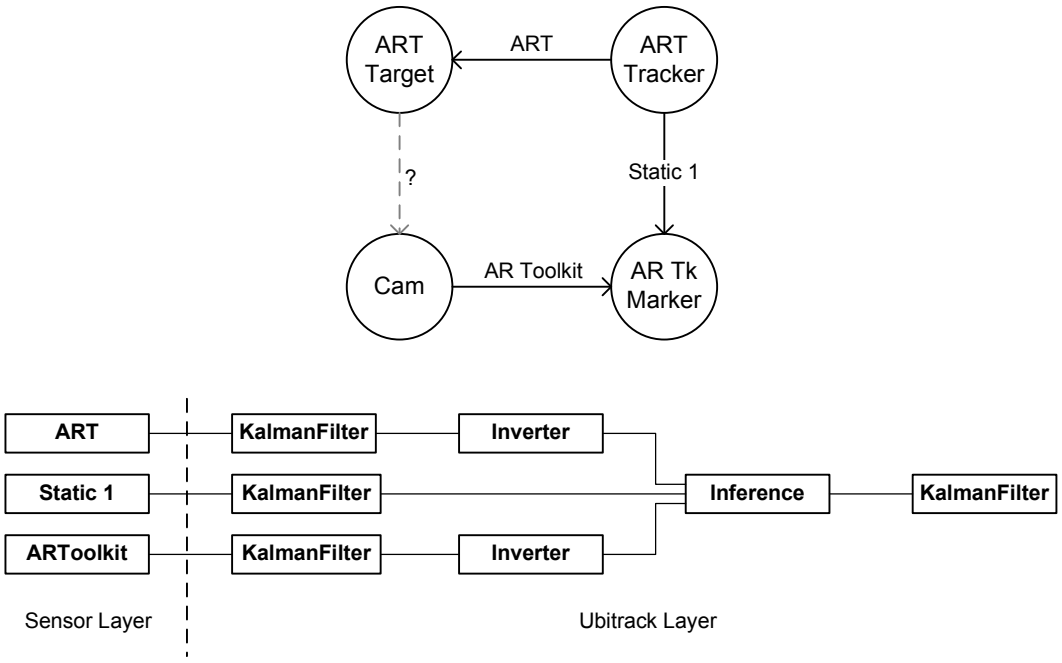


Figure 7.3: Spatial relationship and data flow graphs for estimating the static edge in the demo setup.

Fusion of Head-Mounted and Fixed Sensors

Figure 7.4 shows a scenario in which a data from a head-mounted camera using the AR Toolkit and an external ART tracker is fused for higher precision. The setup is similar to that described by Hoff [**?**]. The fusion is performed by the `KalmanFilter` component at the end, which also has a stabilizing effect on the result, as small errors in one measurement may be enlarged by the concatenation of multiple sensors. This time we assume that the application wants tracking data using the `AsynchronousPush` protocol. Therefore, an additional `MeasurementSampler` is inserted at the end of the data flow graph.

7.5 Implementation Status

All the necessary algorithms already are implemented as a library and tested using an offline tool (see next chapter). What still is missing is the integration into DWARF services, but this should not be a lot of work. Unfortunately, also the UMA must be modified, as the current implementation assumes a much simpler data flow architecture using only Inference services.

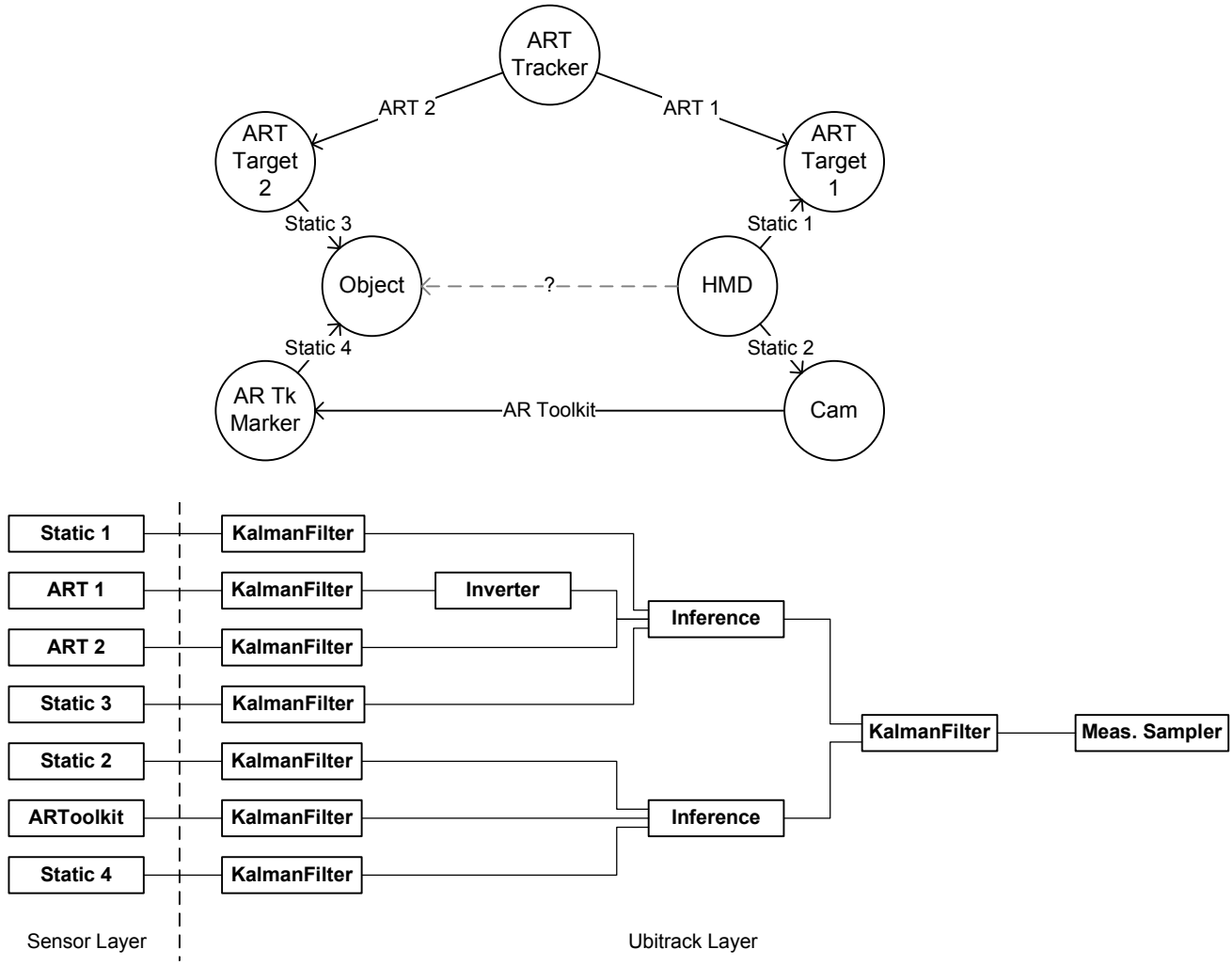


Figure 7.4: Spatial relationship and data flow graphs for fusion of head-mounted and fixed sensors.

8 Developer Tools

The previous chapter described the software components that are supposed to implement the concepts of this thesis in a Ubitrack end system. This chapter is about tools I developed for testing these concepts. The description is included in the thesis, because these tools may be helpful in the development of future systems, and they also are the basis for the next chapter, which describes an evaluation of the concepts.

8.1 PoseTool

PoseTool is a small application that allows an offline analysis of Ubitrack systems by simulating data flow graphs. Another purpose is the tuning of Kalman filters by manually or automatically adjusting the parameters of the motion model (see sec. 6.3.1).

8.1.1 Requirements Analysis

For the requirements analysis, I again use the distinction between functional and nonfunctional requirements [**?**]. As the PoseTool is an independent application, developed mainly for this thesis, no external constraints in form of pseudo requirements were imposed.

Functional Requirements

Recording PoseData The PoseTool is intended to be used with real sensor data, and therefore some way to record `PoseData` events in a running DWARF system is necessary. Especially the simultaneous recording of multiple sensors should be possible for the simulation of more complex data flows.

Loading and Saving PoseData In order to allow repeatable evaluations, sequences of recorded `PoseData` events should be saved and restored. This also allows interfacing with external programs for more complex modifications or the generation of synthetic tracking data.

Playback of PoseData As it is usually easier to examine the effects of the data flow computation in a real AR system than in an abstract 2D representation, it should be possible to play back a processed sequence.

Modification of PoseData A major problem with the current DWARF tracking services is that the covariance matrices often are guessed and the timestamps do not correctly reflect the time that a measurement was made, but rather give a point later in the processing chain. In order to compensate for such errors, it should be possible to scale

the covariance matrices and to shift the timestamps by a constant offset. However, if such corrections are found to be systematic, they should be included the tracker service.

Data Flow Simulation The main purpose of the PoseTool is the simulation of data flow graphs. Therefore all necessary operations (Inference, Inversion, Kalman Filters) must be supported and it should be possible to apply one filter to the result of another. Also the effect of prediction should be simulated.

Manual Kalman Filter Tuning For tuning a Kalman filter, it is necessary to adjust the parameters of the motion model. This should be done graphically, and the effects must be visible immediately to allow an intuitive approach.

Automatic Kalman Filter Tuning In higher dimensions, manual adjustment of parameters becomes increasingly difficult. One approach for automatic determination of Kalman filter parameters is described by Azuma [**?**]. The PoseTool should include a similar technique.

Visualization of PoseData Sequences To see the effects of a simulated data flow, the application must be able to visualize sequences of `PoseData` events. It also should be possible to view predicted or filtered sequences together with the original data.

Nonfunctional Requirements

The main nonfunctional requirement is the response time. In order to allow an intuitive adjustment of Kalman filter/motion model parameters, the effect of a changed parameter should become visible immediately. This should also work for longer chains of transformations, when a parameter at the beginning is changed.

8.1.2 Use Case Models

The use cases described here show the main principles of how the software is operated and may serve as a reference to anyone who wants to use it. Trivial operations, such as loading or saving are not included as their own use cases. I also do not make a distinction between different windows for adjusting the parameters of different filters, as the general approach is always the same. All of these windows are called `ParameterWindow`.

Select Sequences

This use case describes how one or multiple `PoseData` sequences are selected for being used in subsequent modifications.

Participating actors Initiated by `User`

Entry condition
1. At least one `PoseData` sequence was loaded or recorded and appears on the list in the `MainWindow`.

Flow of events
2. By clicking on one sequence in the `MainWindow`, the first entry is selected and appears in inverted colors.
3. More sequences can be selected by holding the shift or control keys while clicking.

Exit condition
4. One or more sequences in the list in the `MainWindow` are selected and displayed in inverted colors.

Display Sequences

This use case describes how one or more `PoseDataSequences` are displayed.

Participating actors Initiated by `User`

Entry condition
1. One or more sequences are selected (includes `Select Sequences`).

Flow of events
2. The `User` clicks on the "Display" button in the `MainWindow`.
3. A `DisplayWindow` appears which shows all selected sequences in different colors.
4. The `User` can now select which components to view, scroll, zoom, etc.

Exit condition
5. The `DisplayWindow` is visible on the screen and shows the selected sequences.

Begin Modification

This use case describes how a `PoseData` sequence is modified. The main purpose is to explain the general operation of the program, and therefore it applies to all modifications and data flow operations. The functionality and the meaning of the various parameters that occur are either self-explaining or described in detail in the previous chapters.

Participating actors Initiated by `User`

Entry condition
1. One or more sequences are selected (includes `Select Sequences`).

Flow of events
2. The `User` clicks on one of the modification buttons (Scale Covariance, Time Shift, Invert, Infer, Kalman Filter, etc.).
3. A new `ParameterWindow` appears with the parameters of the modification.
4. A new sequence shows up in the `MainWindow` which contains the result of the operation.

Exit condition
5. The `ParameterWindow` is visible on the screen.

Special requirements Multiple modifications can be started simultaneously.

Adjust Parameters

This use case describes what happens when a parameter in a `ParameterWindow` is changed.

Participating actors	Initiated by `User`
Entry condition	1. The `User` modifies a parameter in the `ParameterWindow` of a previously started modification (includes `Begin Modification`).
Flow of events	2. The result sequence is immediately recomputed. 3. All other modifications that have the result sequence as an input and whose `ParameterWindow`s are still opened also recompute their result. 4. `DisplayWindow`s that show one of the recomputed sequences are redrawn.
Exit condition	5. All sequences and displays reflect the change.

End Modification

This use case describes how to stop a result sequence from a modification to be recomputed.

Participating actors	Initiated by `User`
Entry condition	1. A modification was started by the `User` (includes `Begin Modification`).
Flow of events	2. The `User` clicks on the "X" button in the caption bar of the `ParameterWindow`. 3. The `ParameterWindow` disappears.
Exit condition	4. Changes to one of the input sequences no longer cause a recomputation.

8.1.3 User Interface

The most important windows of the PoseTool application are show in figure 8.1.

8.1.4 Automatic Kalman Filter Tuning

Most of the PoseTool features are either trivial to implement or they were already explained in the previous chapters. However, the automatic Kalman filter tuning deserves some attention. Azuma [**?**] explains that he is using Powell's method, a general-purpose non-linear optimizer for computing the parameters of his system. A non-linear optimizer generally

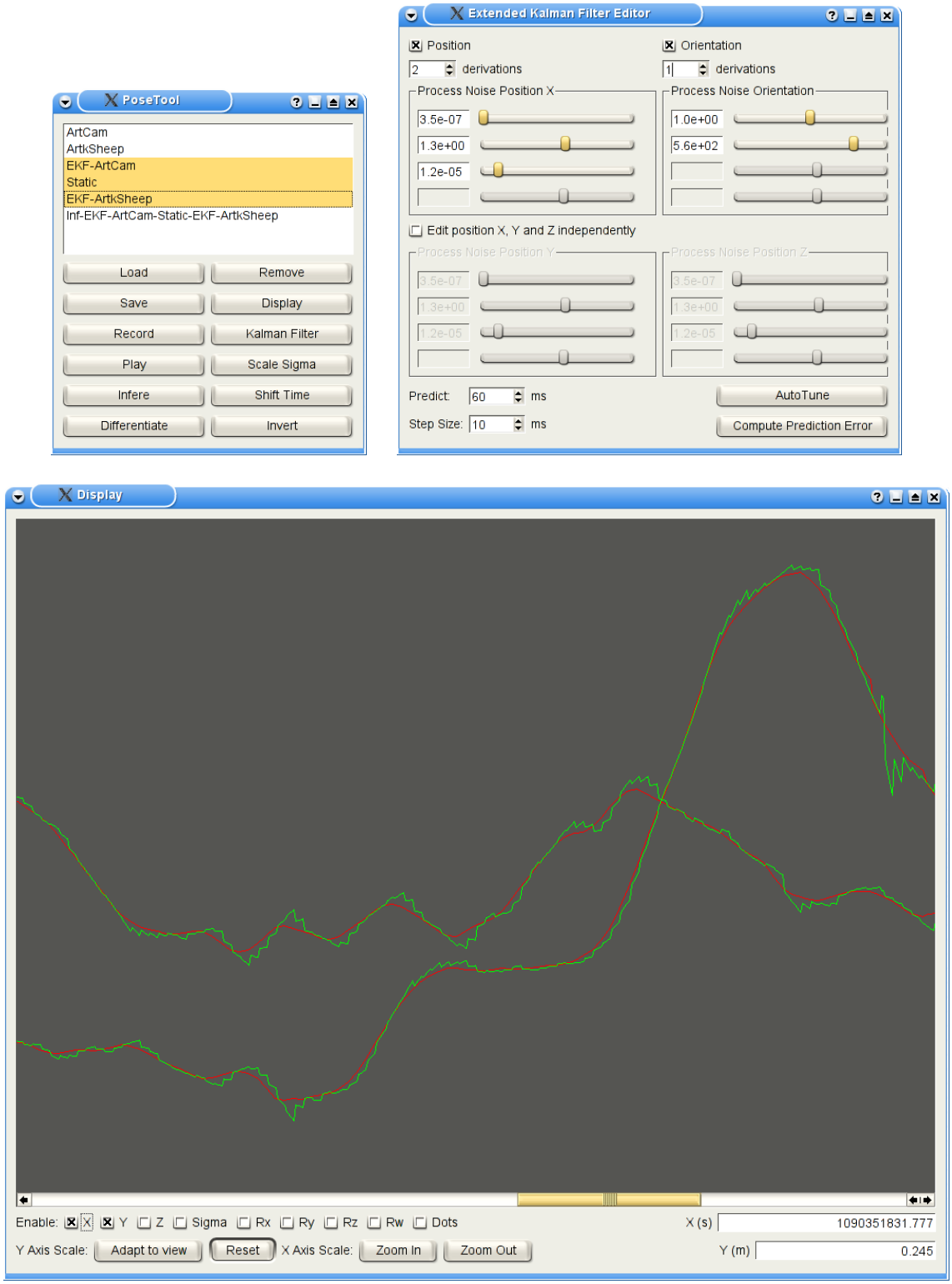


Figure 8.1: Some windows of the PoseTool application.

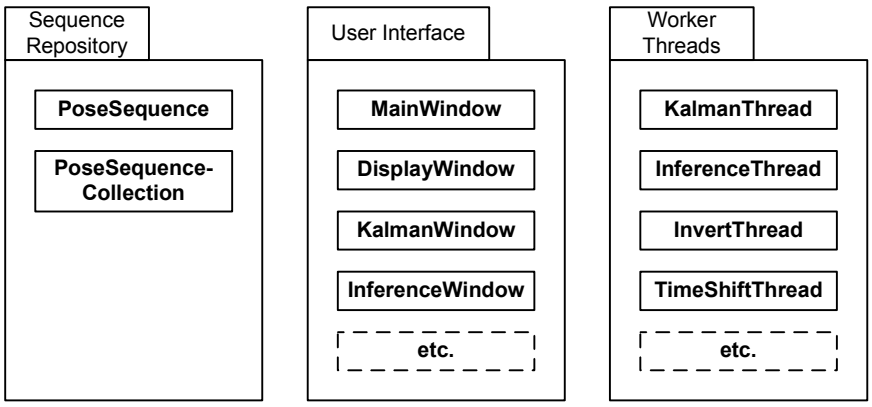


Figure 8.2: Subsystem decomposition and main classes of the PoseTool application.

tries to find a point in parameter space which minimizes a certain cost function. In case of our Kalman filter, about the only thing we can optimize for is prediction. As the tuning is done off-line, it is possible to compute predicted values for certain times and compare these with the known real pose values. Because there are no real measurements available at every time, I use interpolated values instead. So for every measurement, a predicted and a simultaneous interpolated value is computed. This yields the following cost function:

$$e = \sum_{k=0}^{n} |\mathbf{z}_{p,k} - \mathbf{z}_{i,k}|^2 \tag{8.1}$$

The prediction distance is chosen in the `KalmanWindow` and the optimization starts with the parameters manually selected there. Only the process noise parameters which are activated in this window are optimized. Also the number of derivations has to be chosen manually.

8.1.5 System Design

This section gives a rather high-level overview of implementation and is meant as a short introduction for anyone who wants to make extensions to it. A detailed explanation of the single objects is omitted.

8.1.5.1 Subsystem Decomposition

The subsystem decomposition follows the popular model-view-controller (MVC) pattern. The subsystems and their most important classes are depicted in figure 8.2.

PoseSequence Repository is the "Model" part of the MVC pattern. A `PoseSequence` simply is a list of consecutive `PoseData` structures. It also provides a signal to notify other parts of the application when the contents have changed, e.g. when the `User` has adjusted a parameter. The `PoseSequenceCollection` is a singleton object which contains all loaded or recorded `PoseSequence`s.

User Interface contains all the visible parts of the application. The `MainWindow` displays the contents of the `PoseSequenceCollection` and has a number of buttons to start modifications. The `DisplayWindow` visually displays a `PoseSequence`. The other dialogs contain controls for adjusting the parameters of the various modifications and data flow operations.

Worker Threads are started by the user interface when a parameter was changed and therefore a `PoseSequence` has to be recomputed. When the parameter is changed again before the recomputation has finished, a worker thread is automatically stopped and restarted immediately.

8.1.5.2 External Libraries

PoseTool uses the following external libraries:

Qt is used for the user interface parts, the threading, and the signals-and-slots mechanism handles the notification when a `PoseSequence` has changed.

GNU Scientific Library provides an optimization algorithm which is used for the automatic Kalman filter tuning.

TNT/JAMA libraries are used for all linear algebra operations, such as matrix multiplication, inversion or eigenvalue computations.

8.1.5.3 Global Software Control

As PoseTool is based on the Qt library, the global software control is dominated by Qt's event loop and the signals-and-slots mechanism. Worker threads are used for the actual computations.

8.1.6 Implementation Status

So far, everything except the recording and playback functions is implemented. For the evaluation, the recordings were made with the existing `PoseDataLogger` service, which is rather inconvenient, as a new service description needs to be created for each source. Also, no automatic synchronization is available when multiple sources must be recorded simultaneously.

8.2 Runtime Error Visualization

The purpose of the error visualization service is the visualization of errors in position and orientation (covariance matrices) at runtime in a DWARF system. This may help a developer of a Ubitrack system in understanding the effects of sensor uncertainties. As the visualization is just a small tool, the description given here is rather informal.

8.2.1 Requirements Analysis

The error visualization program should fulfill the following requirements:

Visualization of both positional and orientational error. As I have explained in section 5.2.2, the covariance matrix can be visualized as an ellipsoid to show the uncertainty in position. For displaying the orientational error, the position of the coordinate axes, which results from that error, should be shown.

Realtime In contrast to the PoseTool, the error visualization must work in real-time in a running system.

DWARF Integration The error visualization must run in a DWARF system to allow the easy use as a development tool.

8.2.2 System Design

This section describes the general approach of the implementation of the error visualization.

8.2.2.1 Visualizing Errors

From section 5.2.2 we know that it is possible to visualize a covariance matrix as an ellipsoid by computing the eigenvalues and eigenvectors. The eigenvectors give the direction of the axes, and the eigenvalues give the lengths. To visualize the error in position, we can directly apply this to the upper left submatrix of the 6×6 covariance matrix and display the resulting ellipsoid at the position of the tracked object.

In order to visualize the orientational error, we can show the displacement of each of the axes the occurs because of rotational errors. For computing this displacement, a simplified version of the error propagation formula for the inference 6.1.2 can be derived by setting all errors but that of the rotation to 0 and using the direction of the axis as the second translation. The resulting covariance ellipsoid has zero extension in the direction of the axis.

A screenshot of the final visualization is given in figure 8.3. A similar approach is described by Hoff [**?**], but the axes are visualized as cones to represent the rotational errors.

8.2.2.2 DWARF Integration

In order to display the covariance ellipsoid, the DWARF Viewer service is used. This allows us to show the ellipsoid at the same place where the tracked object usually is. Therefore the error visualization service has a *Need* for a `ViewerControl` interface. Another need receives `PoseData` events. For each incoming event which has a different Source-Target-ID pair, a new ellipsoid is created in the viewer. This allows the error visualization service to to display multiple errors simultaneously when appropriate DWARF predicates are chosen in the service description. After the ellipsoid is create, for all other received `PoseData` events with the same Source-Target-ID pair, the orientation and scale of the ellipsoid is computed and the transformations of the objects in the viewer are changed accordingly.

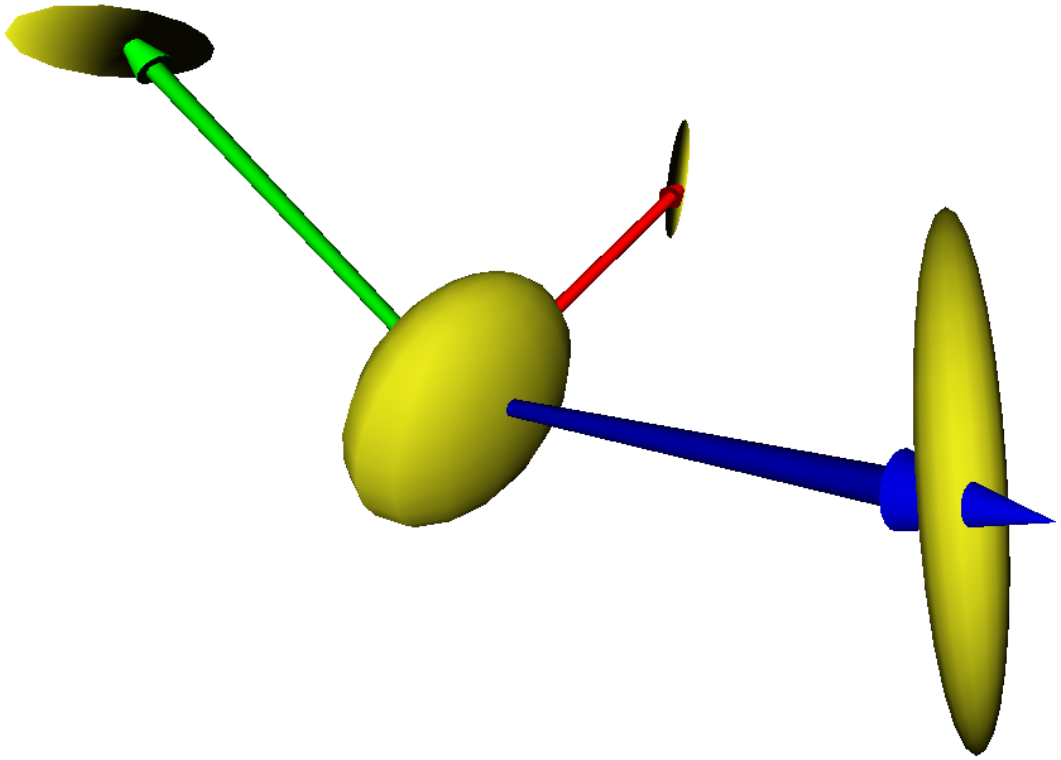


Figure 8.3: A screenshot of the error visualization.

It should be noted that the displayed error only represents the error on the path from the source to the target of the received event. The errors of all other transformations which participate in displaying the objects at the correct position in the viewer do somehow influence the position and orientation of the ellipsoid, but are not explicitly visualized.

8.2.3 Implementation Notes

For computing the eigenvalues and eigenvectors of the covariance matrix, the JAMA library is used. A problem is that the resulting eigenvalues and their associated eigenvectors are sorted by the magnitude of the eigenvalue. Therefore the eigenvectors can have different order in a sequence of `PoseData` events, which may cause sudden rotations of the ellipsoid. In order to compensate for this, I re-sort the eigenvectors every time to match the order from the last event.

9 Evaluation

In the previous chapters, mathematical methods and software components have been described for dealing with errors in ubiquitous tracking systems. In this chapter, I will try out these methods on real measurements to see if the goals of the thesis can be reached.

9.1 Demonstration Setup Using Two Trackers

At first, I describe the tracking setup that was used to generate the raw measurements on which the algorithms are evaluated. The basic setup is the same that was described in section 2.3, with a mobile camera being tracked by a fixed tracker.

9.1.1 Trackers

This setup, consists of two different optical tracking technologies. For connecting to the sensors, existing DWARF services were used.

ART DTrack

The ART DTrack is a commercial optical tracking system. In our lab we have three cameras installed at fixed locations, which effectively realize an outside-in tracking setup. The cameras contain infrared LEDs which illuminate the scene with periodical flashes. ART tracking targets consist of rigid arrangement of retro reflective balls. ART cameras and targets are shown in figure 9.1. The cameras also contain a feature detection logic and send the 2D locations of detected balls to a dedicated tracking computer that combines the measurements of multiple cameras and computes the 6D pose of the targets. The final pose is sent to client computers using UDP packets.

AR Toolkit

The AR Toolkit [**?**] is a freely available software library for optical marker detection and pose reconstruction. It library detects the position and orientation of high-contrast rectangular patterns. Two example markers are shown in figure 9.2. In our setup, a firewire iBot camera is used, which delivers 30 frames per second at a resolution of 640×480 pixels. An ART target is rigidly attached to the camera (see fig. 9.3).

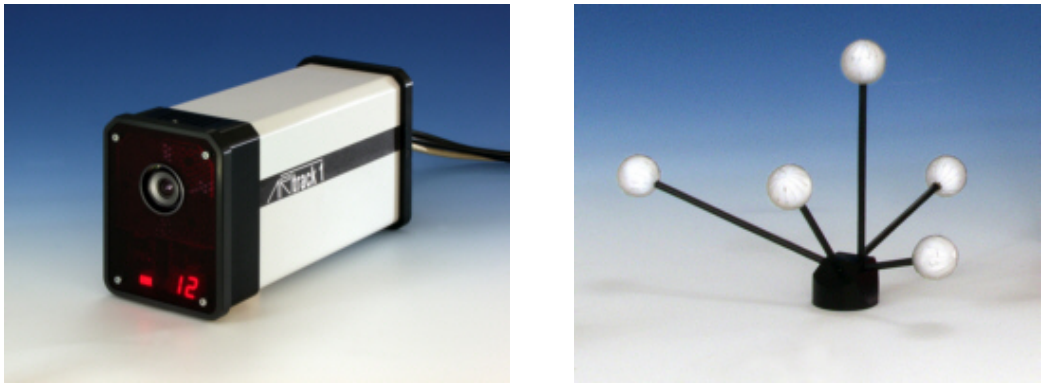


Figure 9.1: ART camera and a typical target

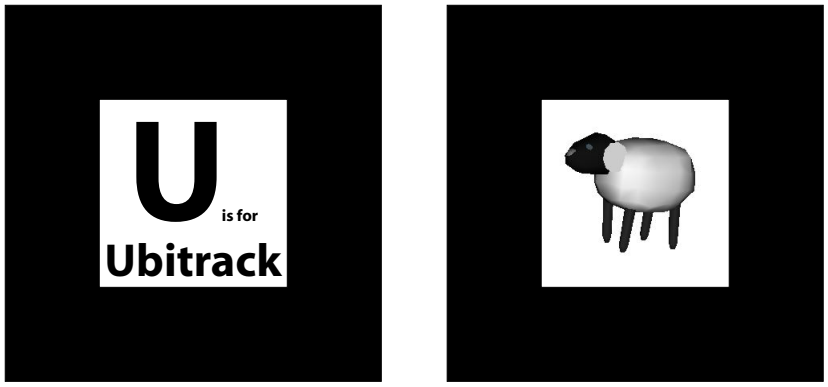


Figure 9.2: AR Toolkit markers

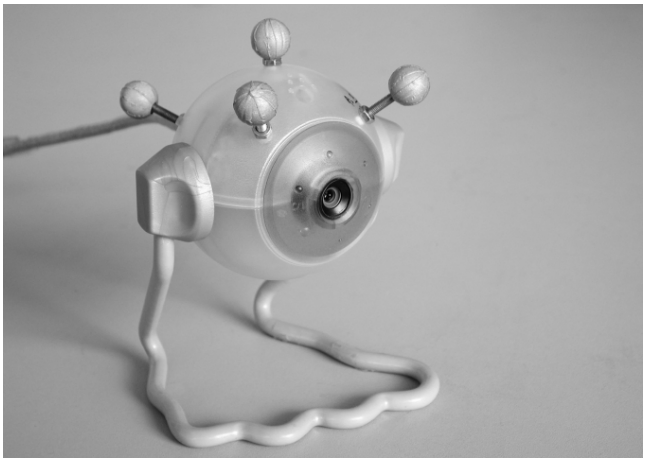


Figure 9.3: iBot camera with attached ART target

9.1.2 Analysis Method

For analyzing the methods derived in this thesis, the measurements of both tracking system were recorded simultaneously using the `PoseRecorder` service, which already was available from a previous project. The `PoseRecorder` had to be slightly modified to accommodate the additional Ubitrack fields in the `PoseData` structure. For higher flexibility, all the dataflow computations (Kalman filter, inference, inversion, etc.) an the analysis were done offline using the PoseTool, described in the previous chapter. A real Ubitrack system, consisting of DWARF components, should deliver the same results.

9.2 Results

The rest of the chapter describes the results of the evaluation. Each section shows how well one of the goals of the thesis (see section 3.2) was reached.

9.2.1 Prediction and Kalman Filter Tuning

For the prediction, the measurements of each of the two sensors were processed and analyzed individually. A Kalman filter was tuned to deliver the best possible predicted values. The same filter also is used in the next section to solve the measurement simultaneity problem.

Manual Kalman Filter Tuning The PoseTool gives a quite intuitive sense for the meaning of the motion model parameters. By displaying both the original and the filtered signals, the effects on the prediction can be seen immediately as the parameters are changed. However, finding good values still is not easy, due to the number of degrees of freedom and the high sensitiveness at certain positions.

Instabilities For certain parameter combinations, the Kalman filter becomes instable, which results in high-amplitude oscillations around the true value. For the position, these combinations however are sufficiently far away from the "good" motion model parameters. This problem is more critical for prediction of orientation, as outliers may cause the angular velocity to jump by $\pm 360°/s$, which from the Kalman filter's point of view generates almost the same motion, but looks dramatically wrong when used for prediction e.g. of head orientation. In addition, it was necessary to include a reset of the Kalman filter whenever the filter matrices became singular.

Automatic Tuning The automatic tuning routine, described in section 8.1.4, works reasonably well, but depends on the quality of the input sequence as well as on the start parameters. Therefore it is usually necessary to choose sufficiently good parameters manually and use the automatic function for the fine tuning only.

In order to reduce the required computation time, position and orientation can be tuned separately, which results in a faster Kalman filter execution and a lower-dimensional search

	Pos 0	Pos 1	Pos 2	Ori 0	Ori 1	Ori 2
Parameter 1 (ART)	$1.1\ 10^{-6}$	1.2	9.7	3.1	250	$1.7\ 10^{-5}$
Parameter 2 (AR Toolkit)	$5.8\ 10^{-5}$	4.3	0.13	5.4	75	$8.4\ 10^{-3}$

Table 9.1: Kalman filter parameters used in the evaluation

space for the optimizer. For optimizing the Kalman filter to an input sequence of about one minute, between 150 and 300 optimizer iterations were necessary, which took less than a minute on a 2 GHz Pentium 4 processor.

When tuning the motion model to the AR Toolkit measurements, a few outliers had to be removed manually before the automatic tuning. This is because the quadratic error function 8.1 weights outliers most heavily and therefore tends towards optimizing the areas around the outliners, which somehow decreases the prediction performance of the rest of the sequence.

Observations The resulting motion models which optimize the prediction performance for the ART and AR Toolkit trackers are given in table 9.1. The parameters were determined as described above by manually pre-tuning and the running the automatic tuning to optimize over a prediction distance of 50ms. The motion model parameters determine how the different derivations are weighted. By examining the results, the following interesting observations can be made:

- The process noise for the absolute position is almost 0, which means that incoming measurements are used to correct only the derivations, and no averaging of absolute positions is performed.
- The ART-optimized motion model gives more weight to the 2nd derivation of position than to the first. This shows that the measurements produced by the ART system are virtually free of noise and therefore even the 2nd derivation produces useable results.
- For the orientation, the 2nd derivation plays no role. There are two possible explanations for this. Either the formulas derived in chapter 6 are completely wrong, or – more likely – the orientation measurements are so noisy that higher derivations effectively are useless.

Qualitative Analysis

To give the reader an idea of how the Kalman filtered results look like, I have applied it to two short sequences from the ART and AR Toolkit measurements and printed the results in figures 9.4 to 9.13. I also included examples with no prediction and a simple linear extrapolation.

The figures show the measured signal together with the prediction result. The dashed lines are the linearly interpolated measurements and only the crosses represent the actual sensor readings. Each page includes a diagram with no prediction, where only the short intervals between measurements are bridged by extrapolation. For the diagrams at the bottoms of the pages, an additional prediction interval of 100ms is added.

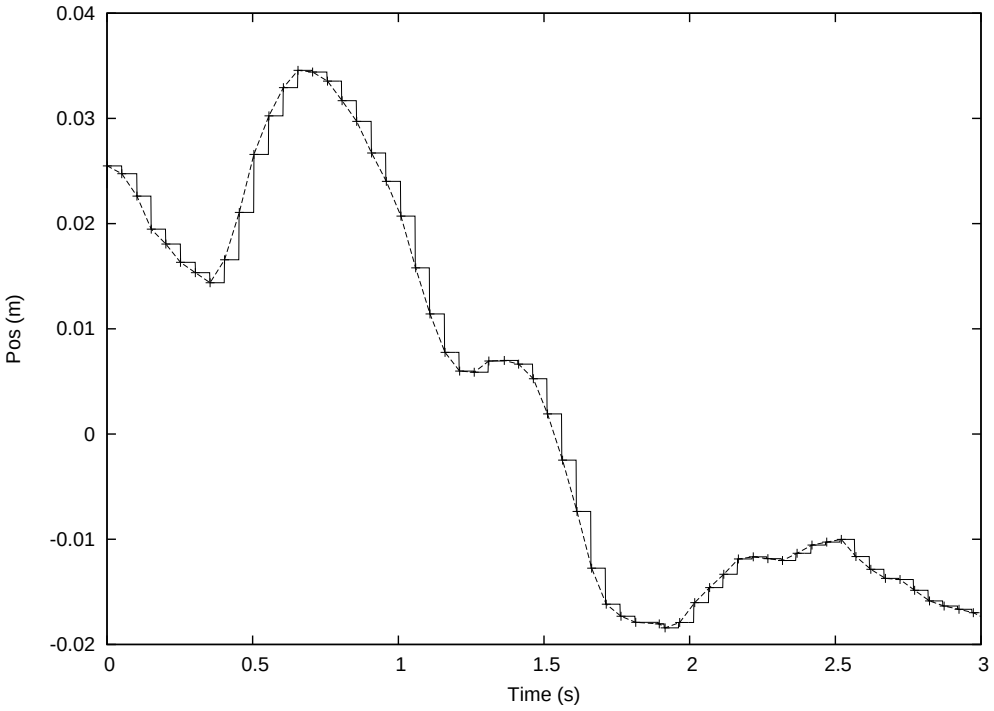


Figure 9.4: ART measurements without prediction.

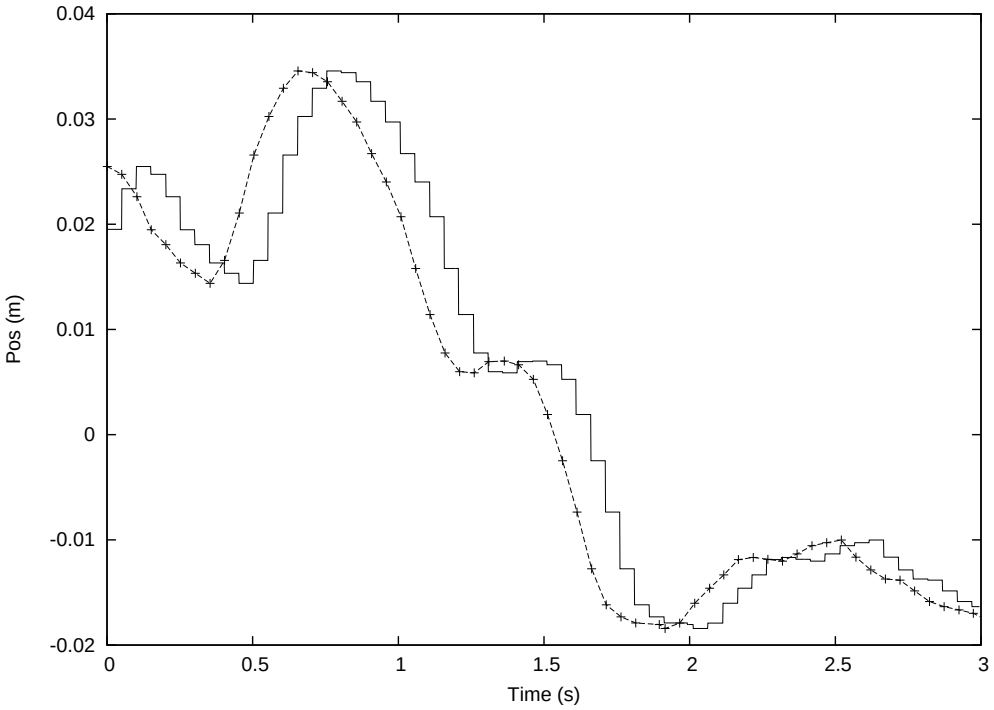


Figure 9.5: ART measurements without prediction shifted by 100ms.

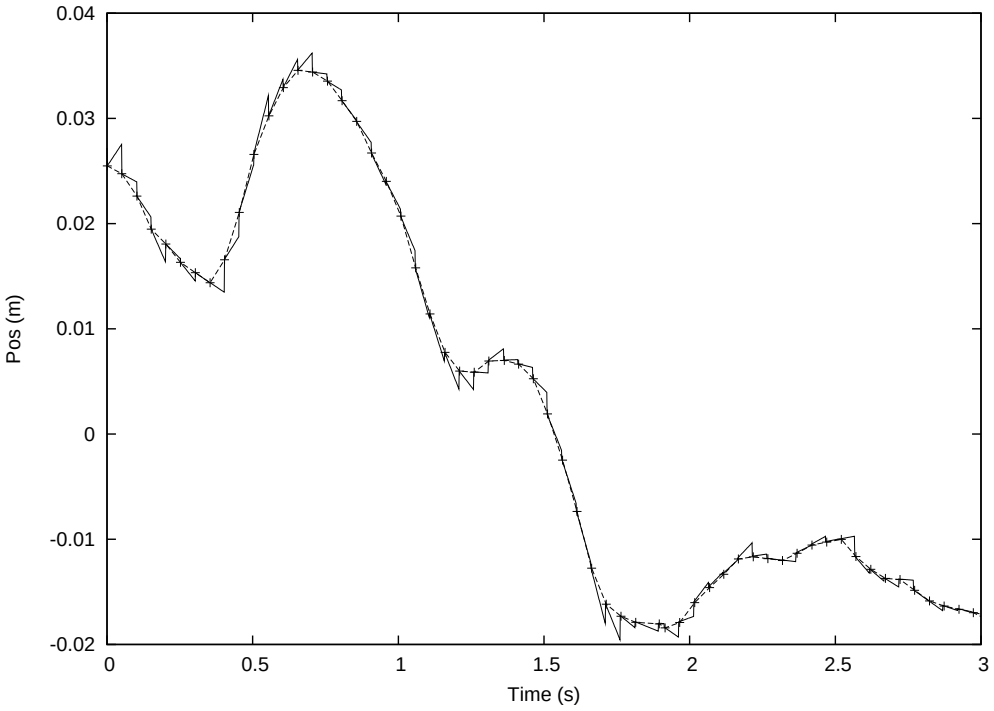


Figure 9.6: ART measurements with linear prediction.

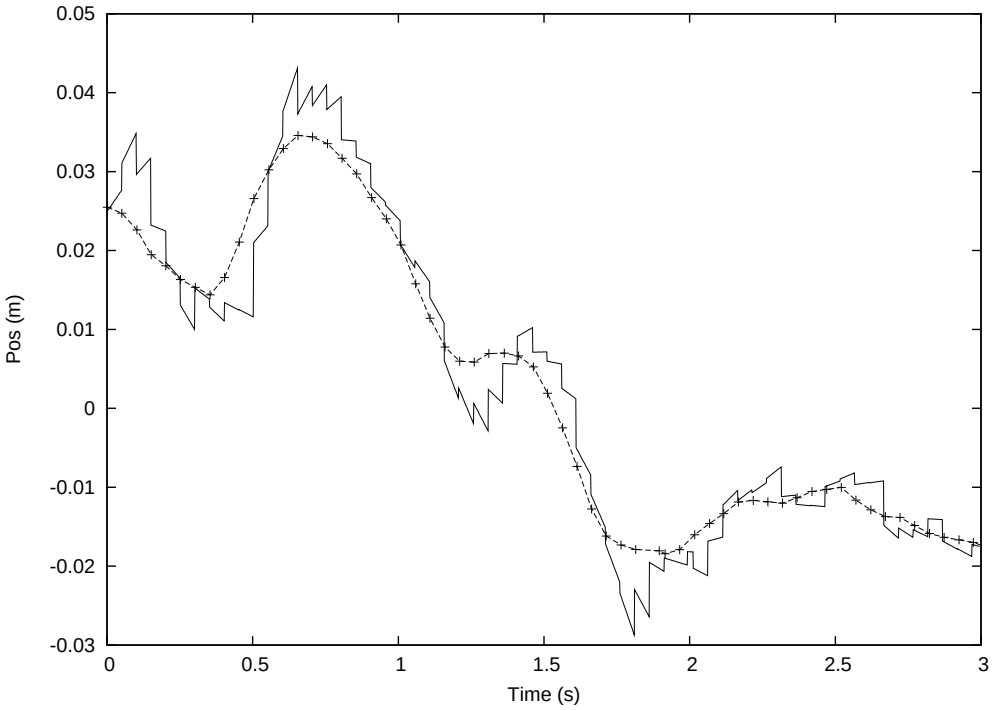


Figure 9.7: ART measurements with linear prediction of 100ms.

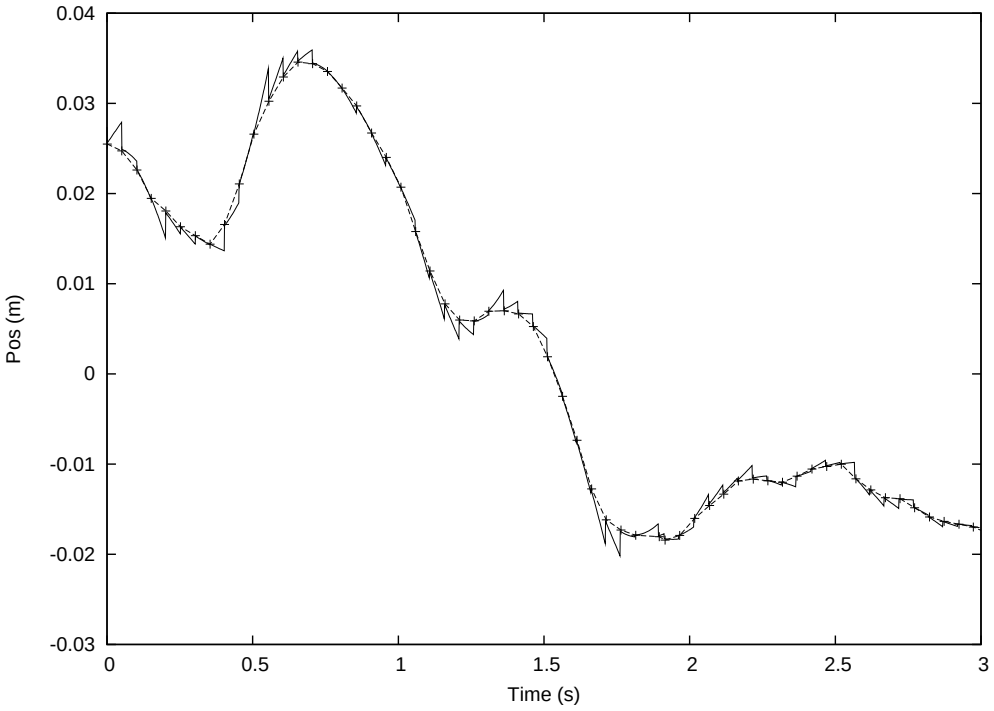


Figure 9.8: ART measurements predicted using a kalman filter.

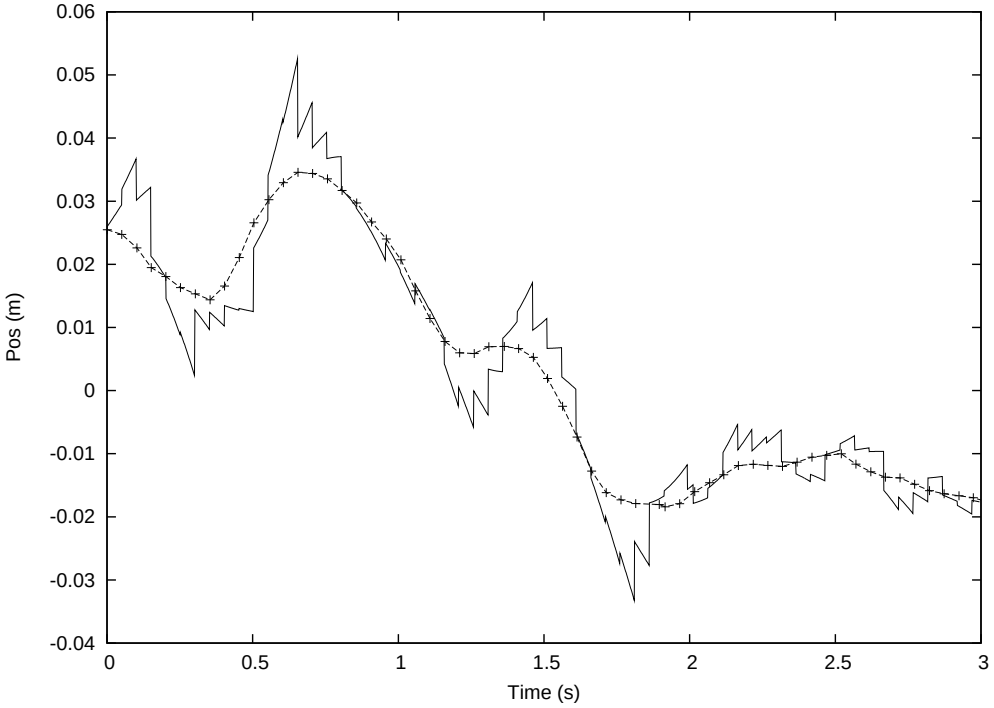


Figure 9.9: ART measurements predicted by 100ms using a kalman filter.

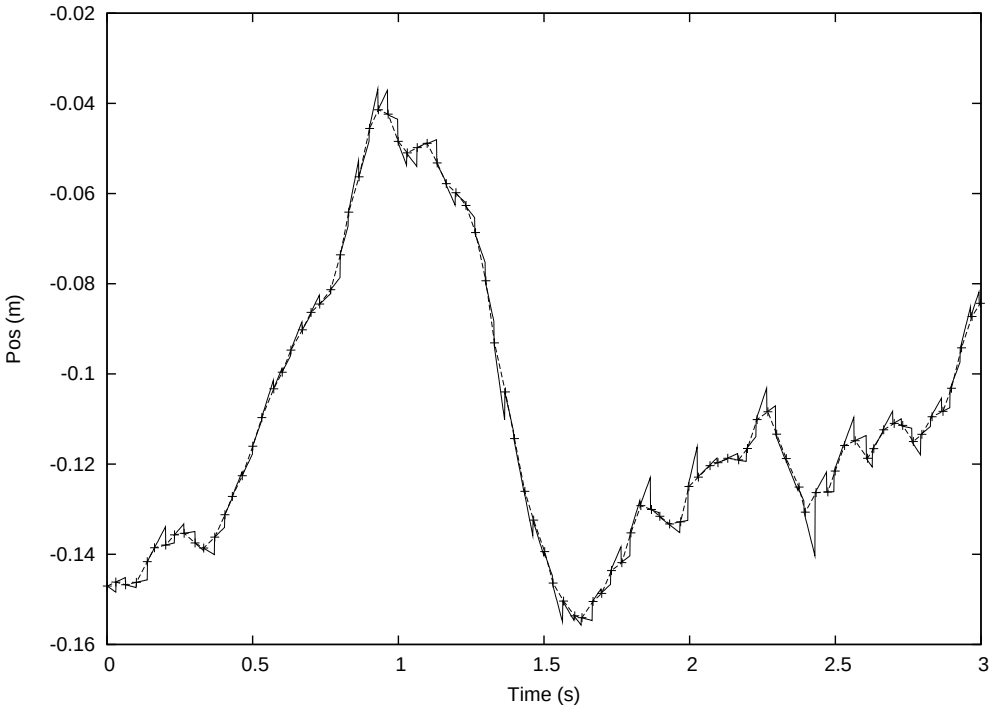


Figure 9.10: AR Toolkit measurements with linear prediction.

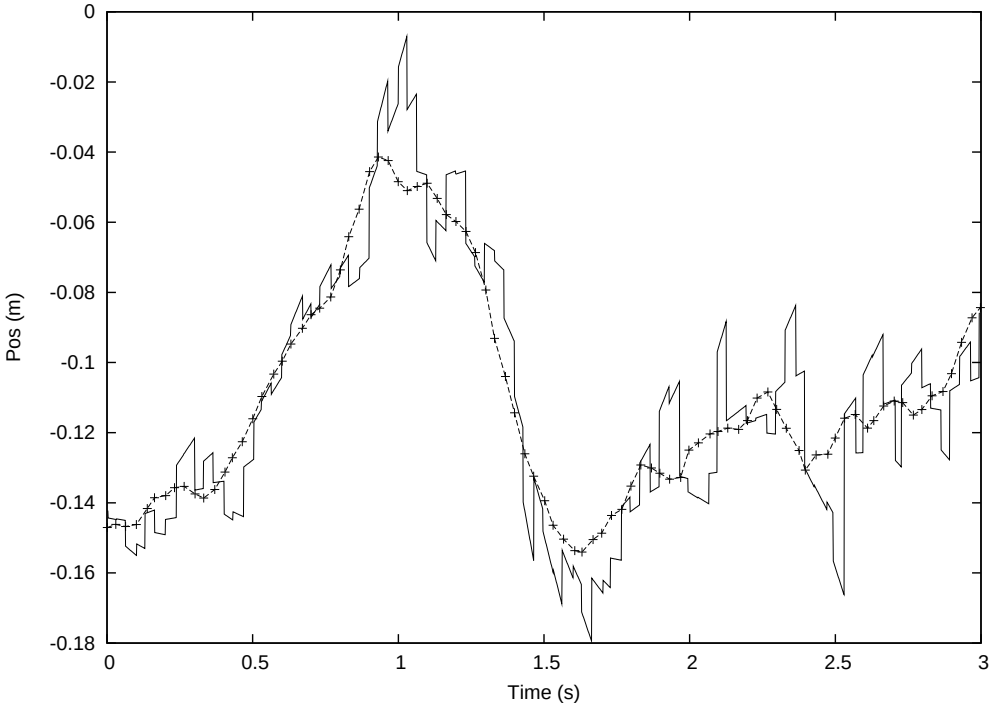


Figure 9.11: AR Toolkit measurements with linear prediction of 100ms.

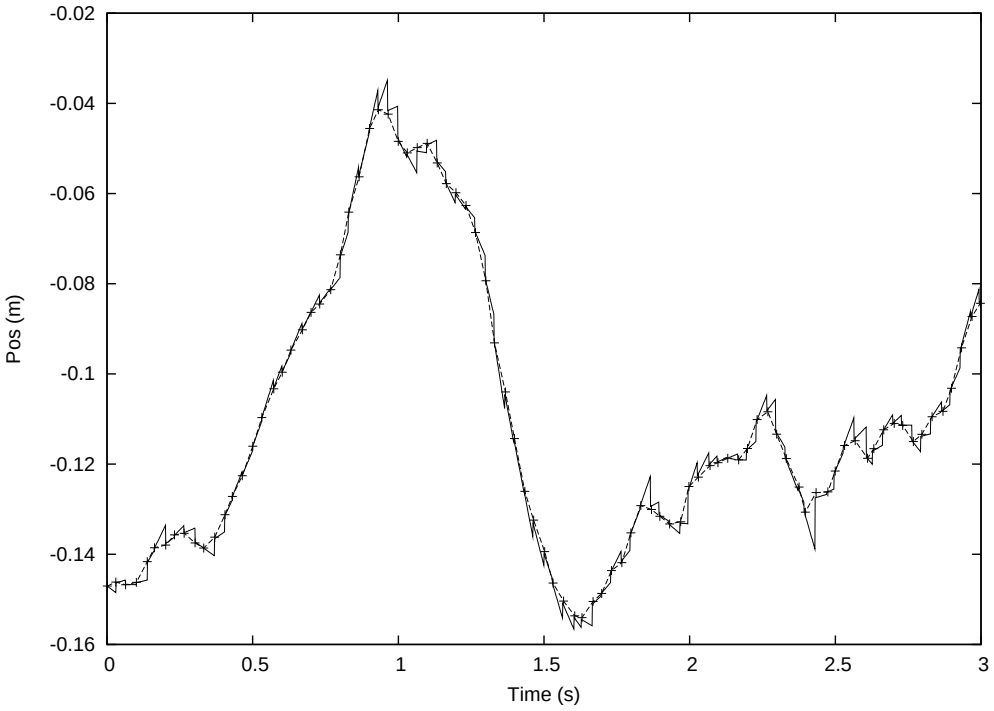


Figure 9.12: AR Toolkit measurements predicted using a kalman filter.

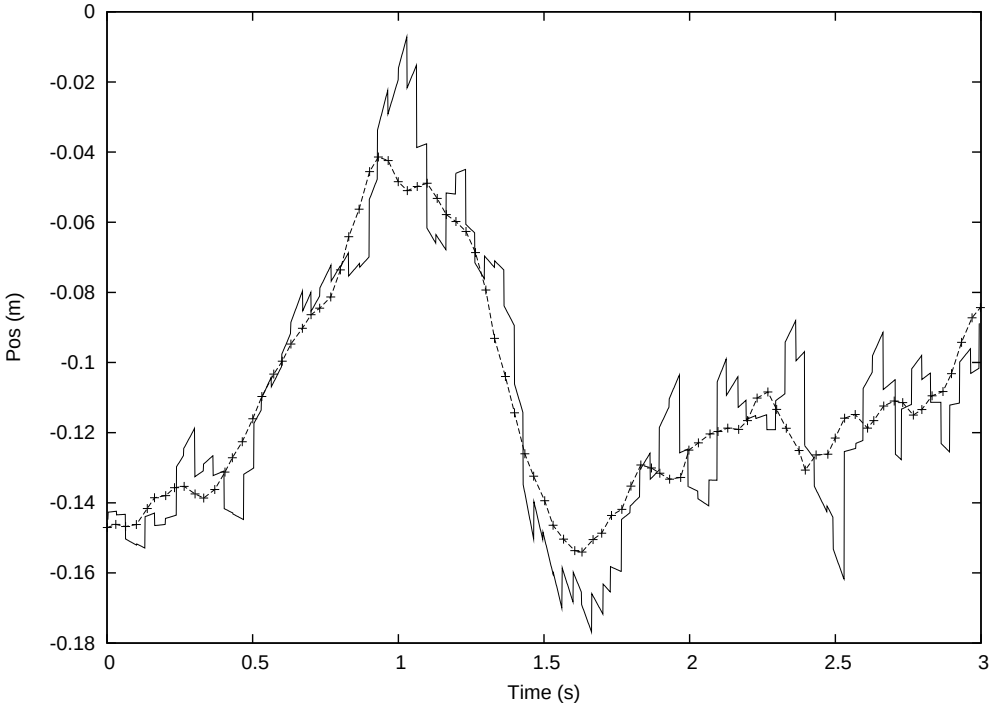


Figure 9.13: AR Toolkit measurements predicted by 100ms using a kalman filter.

Observations By closely examining the images, we can make the following observations:

- The AR Toolkit measurements are much more noisy than those of the ART system, and therefore the linear prediction has higher fluctuations.
- We can see that the Kalman filtered sequences have a bit fewer high-frequency oscillations, but they also react slightly more sluggishly to changes in direction.

Quantitative Analysis

In case of the prediction, a quantitative analysis is relatively easy, as we can directly compare the predicted measurements against the interpolated real measurements and add up the errors. I compute both the average error and the root of the mean squared errors (RMS), as the RMS value weights outliers more heavily. When the difference between both values is high, we know that there must be many single values which greatly differ from the real measurements.

To allow a comparison of different motion models, each of the parameter sets from table 9.1 is applied to both tracker sequences. From that we can also see to which degree a motion model is specialized to a particular type of sensor. I further compared the error against that which appears when no prediction or a simple linear extrapolator is used. Both the "no prediction" and the linear predictor cases can be simulated with the Kalman filter, without the necessity to write dedicated code. No prediction is performed when the motion model includes no derivations and the process noise is set to a very high value, so that new measurements completely override the old value in the state vector. For simulating the linear predictor, the state vector is set to include one derivation, the process noise for the absolute value is set closely to 0, and that of the derivation to a very high one.

Observations The results of the quantitative analysis are given in tables 9.2 to 9.5. An examination yields the following observations:

- The error in position is much higher for the AR Toolkit. This could be caused by the fact that motions in the camera image mainly result from rotations of the camera, which yields much faster movements than changes in the position of the camera body or the tracked target.
- The ART system measurements contain virtually no noise, which makes the Kalman filter give almost the same results as the linear extrapolator.
- The measurement noise of the AR Toolkit is quite high, resulting in parameter set 2 giving much better values than the linear extrapolator, especially in the RMS values. The toolkit's orientation values are so disastrous that no prediction results in almost the same error as the Kalman filter. The linear extrapolator is unusable here.

	10ms		20ms		50ms		100ms		200ms	
	Avg	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**
EKF Parameter 1	0.31	0.44	0.58	0.86	1.44	2.00	3.64	4.82	10.7	13.7
EKF Parameter 2	0.30	0.45	0.60	0.90	1.47	2.07	3.60	4.88	9.98	13.1
Linear	0.29	0.45	0.59	0.90	1.45	2.07	3.57	4.87	9.86	13.1
No Prediction	1.37	1.79	2.74	3.58	6.82	8.88	13.6	17.6	26.8	34.9

Table 9.2: Prediction error: ART, Position (mm)

	10ms		20ms		50ms		100ms		200ms	
	Avg	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**
EKF Parameter 1	2.75	4.02	3.97	5.83	8.03	11.9	16.6	24.4	40.3	58.6
EKF Parameter 2	1.80	2.94	2.95	4.82	6.50	10.7	13.4	21.7	29.9	46.8
Linear	1.57	3.23	3.03	6.17	7.34	15.1	15.4	30.9	33.7	64.4
No Prediction	2.01	3.03	3.89	5.49	8.81	11.9	16.5	21.5	30.2	38.8

Table 9.3: Prediction error: AR Toolkit, Position (mm)

	10ms		20ms		50ms		100ms		200ms	
	Avg	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**
EKF Parameter 1	0.15	0.20	0.30	0.39	0.74	0.95	1.50	1.94	3.30	4.34
EKF Parameter 2	0.19	0.25	0.37	0.47	0.88	1.14	1.70	2.25	3.51	4.71
Linear	0.15	0.20	0.31	0.40	0.76	0.99	1.56	2.02	3.43	4.50
No Prediction	0.22	0.27	0.44	0.56	1.13	1.42	2.14	2.71	4.03	5.07

Table 9.4: Prediction error: ART, Orientation (deg)

	10ms		20ms		50ms		100ms		200ms	
	Avg	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**	**Avg**	**RMS**
EKF Parameter 1	0.33	1.04	0.59	1.32	1.20	2.18	2.18	3.73	4.27	6.92
EKF Parameter 2	0.35	0.57	0.58	0.93	1.13	1.71	1.99	2.81	3.74	5.04
Linear	0.38	0.74	0.74	1.44	1.61	3.10	3.05	5.72	5.96	10.90
No Prediction	0.30	0.50	0.59	0.94	1.22	1.79	2.16	2.94	3.85	4.95

Table 9.5: Prediction error: AR Toolkit, Orientation (deg)

Position	Orientation	Predictions/s	Measurements/s
2 derivations	2 derivations	20.000	5.500
2 derivations	none	58.000	15.000
none	2 derivations	35.000	12.000
absolute only	absolute only	28.000	11.000

Table 9.6: Kalman filter performance evaluation.

Performance Measurements

For real-time applications, it is important to know how much processing time the Kalman filter consumes. I have made separate measurements for the integration of measurements (measurement update step) and the output of a predicted measurement (time update step). In real systems, both steps are required, but the ration does depend on the sensor update rate and the request rate (e.g. video update rate), and therefore giving combined results does not make much sense.

Table 9.6 shows the results for different state vector sizes. The first row gives the runtime of a "typical" 6 DOF Kalman filter with two derivations for both position and orientation. The following two rows contain the results for a filter run only on position or orientation. Finally, an EKF for the static case (position and orientation, but no derivations) is given. All measurements were made on a Pentium 4 processor with 2 GHz.

The results clearly show that it is possible to run multiple Kalman filters in parallel on one computer, while only consuming a small percentage of available processing power.

9.2.2 Inference and Measurement Simultaneity

In order to analyze the measurement simultaneity problem, the ART–Marker1 edge in the example setup from section 2.3 was compute. As only the camera, but not the marker was moved, the resulting sequence should remain constant.

When analyzing the inferred measurement sequence, four different error sources could be identified:

Constant Time Offset

The timestamps of the measurements of the two sensors are separated by a constant offset, because the timestamps are not set on the sensor itself, but at the time the measurements arrive at the host computer. For the DWARF AR Toolkit implementation, this problem is increased by the fact that the timestamp is not set by the camera, but after two intermediate processing steps.

The result of the offset can be seen in figure 9.14, which shows a small part of the measurement sequence. The solid line is the combined measurement, and the dashed line gives one position component of the AR Toolkit measurements. In the middle of the figure, a rapid motion of the toolkit data can be seen, which should be compensated by another motion in

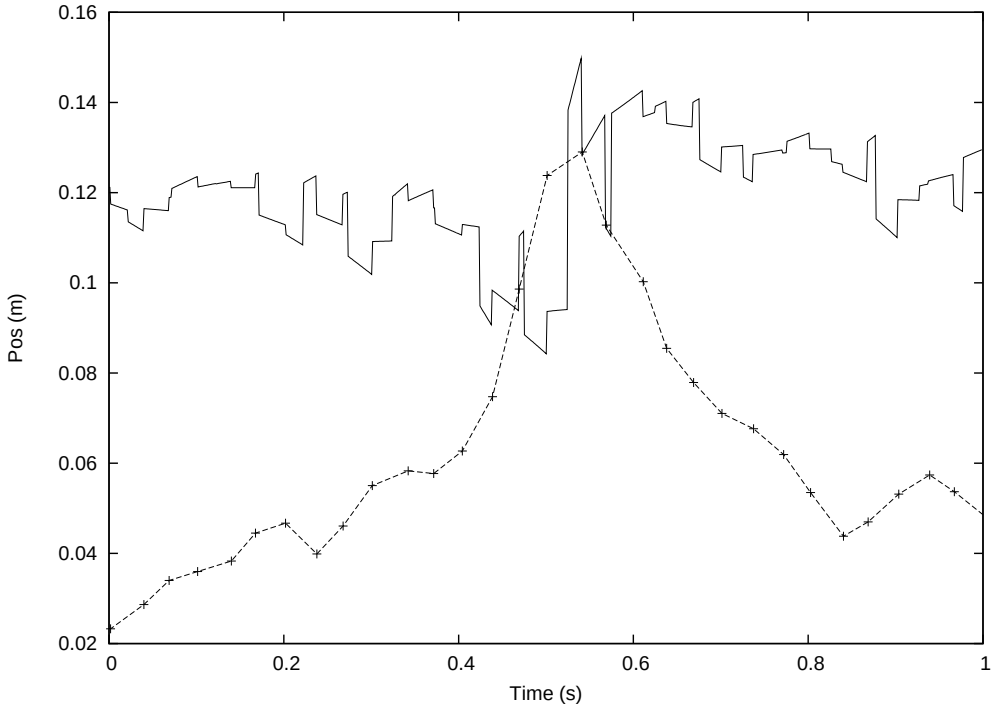


Figure 9.14: Combined measurements with constant time offset.

the ART data. The image however shows that the combined position makes a swing in both directions.

This swing can be compensated by predicting the AR Toolkit measurements 30ms into the future (figure 9.15). However, an even stronger swing to the upside remains, but now this is caused by errors in the prediction.

Spontaneous Timestamp Errors

At some places in the sequence, single erroneous timestamps could be observed, which shift the measurements by a few milliseconds. The order of measurements however is not affected. Such errors may happen when the computer is busy when a new measurement arrives, and the measurement is held in a buffer before the timestamp is set. The timestamps are required in the computation of derivations, and therefore wrong timestamps also cause swings in the resulting measurements.

Prediction Errors

The most visible jitter-like errors in the combined signal are caused by prediction errors which result from both tracker noise and fast motions. This can be seen in the signal as an increasing deviation from the true constant value between two measurements. Prediction errors mainly occur when there are fast motions or accelerations. They can be reduced by using a negative prediction distance, which effectively makes the EKF an interpolator of

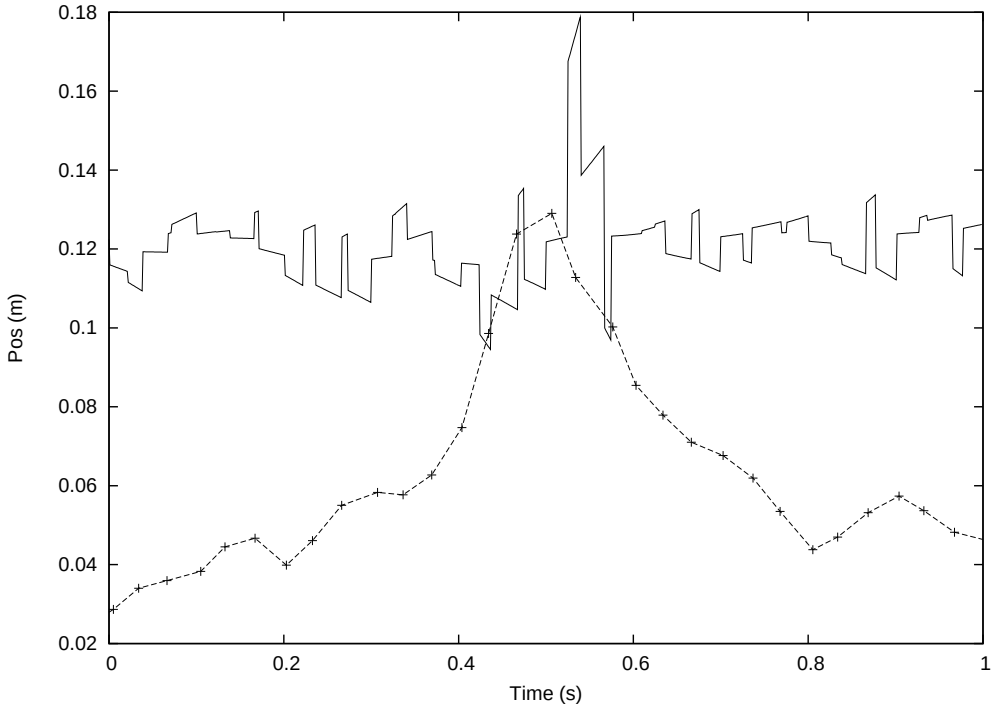


Figure 9.15: Combined measurements with time offset corrected by prediction of one sensor.

old measurements. The drawback is the introduction of an additional delay, and therefore this technique is not suitable for real-time AR augmentations. The combined signal using interpolated measurements is shown in figure 9.16.

Systematic Sensor Errors

Strong low-frequency deviations from the constant value are caused by systematic errors in the sensor measurements, which are not compensated by the other tracker. In such cases, usually a correlation between the deviation and the measurements of one sensor can be observed. In order to reduce systematic errors, all participating trackers must be calibrated carefully.

9.2.3 Estimating Static Relationships

In order to evaluate the estimation of static relationships, the combined measurements from the previous section were ran through another Kalman filter. This time the purpose of the filter was to remove all noise by computing a long-time average of the measurements. Therefore the motion model contained no derivations and the process noise was set close to zero. This worked very well and the filter quickly converged to a constant value for both position and orientation. An example sequence, which shows the beginning of such an estimation is given in figure 9.17.

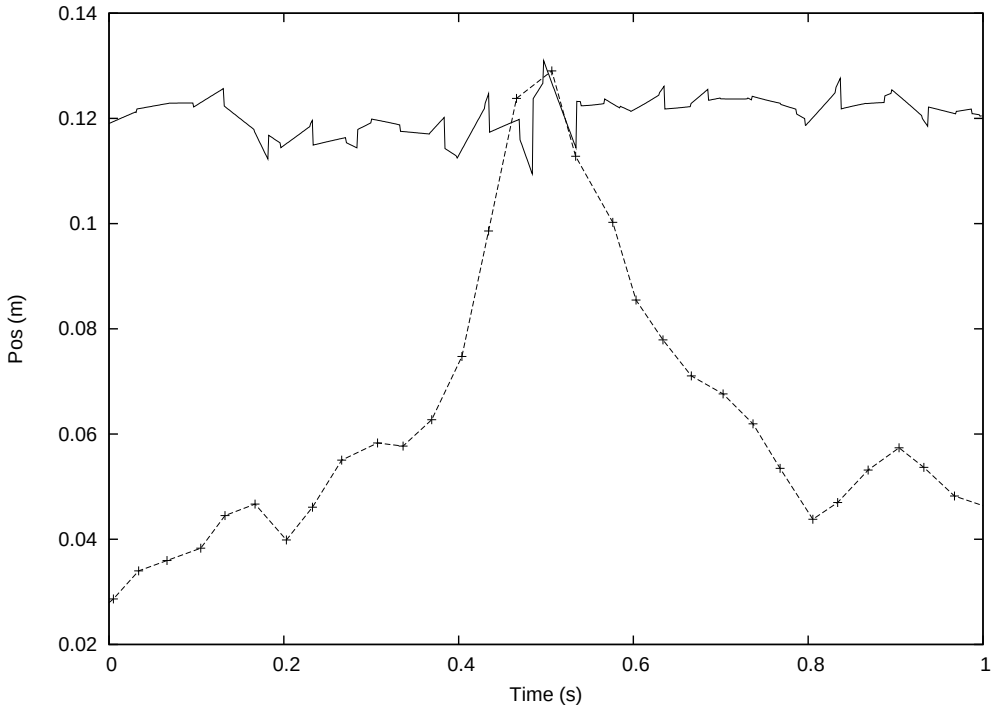


Figure 9.16: Combined measurements with time offset corrected by smoothing.

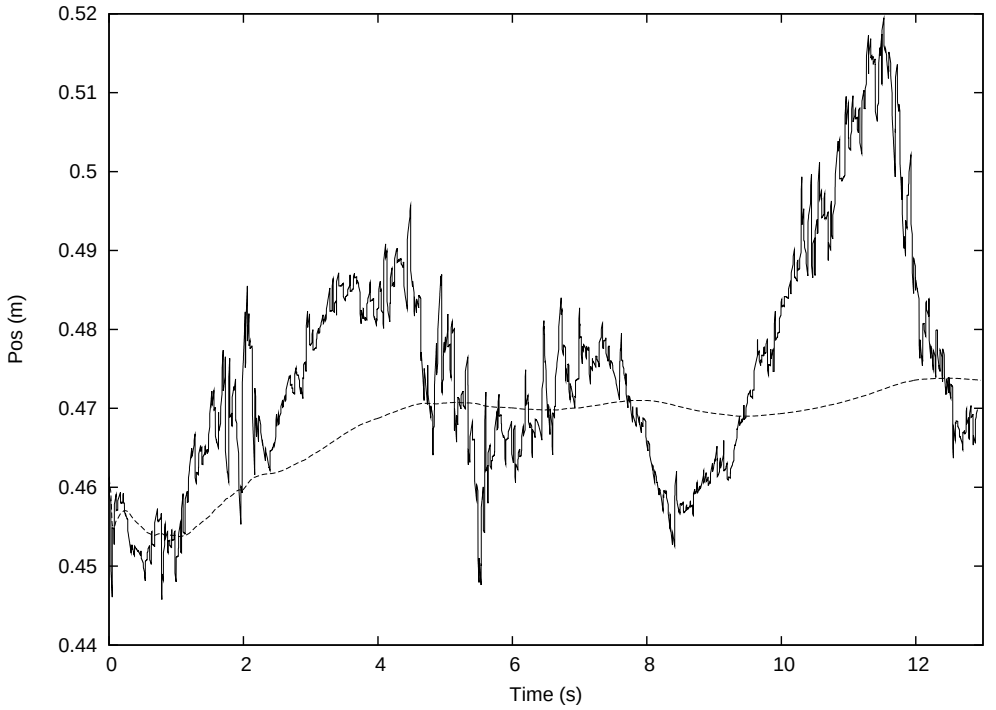


Figure 9.17: Estimating a static relationship using a Kalman filter.

10 Conclusion and Future Work

This last chapter contains a short summary of the thesis' contents and the results, as well as an outlook to potential future research directions.

10.1 Summary

In this thesis, mathematical methods and software implementation concepts have been described for modelling and reducing both static and dynamic errors in ubiquitous tracking systems. The most important results of the thesis are:

Error Model

The error model described in chapter 6 using the "small quaternion" representation of rotational errors has proved to be very useful for ubiquitous tracking systems. Typical data flow operations can be expressed easily and the propagation of errors is achieved by computing Jacobian matrices.

Motion Models

For describing the dynamics of movements, motion models have been derived, which allow an interpretation of measurements over time. They are used in order to construct Kalman filters for optimal estimation of dynamic processes. For generating new motion models, the PoseTool application was developed, which allows intuitive parameter adjustments by immediately showing the effects on recorded measurement sequences.

Dynamic Kalman Filter Construction

The Kalman filter is a very powerful and flexible component in ubiquitous tracking data flows. The time update and measurement equations for 6D pose estimation and prediction are dynamically constructed from motion models and measurement attributes. In Ubitrack, kalman filters can serve a number of purposes:

Prediction The Kalman filter can estimate the current position and orientation as well as dynamic parameters, such as velocity and acceleration, which are used to compute future poses.

Measurement Simultaneity By predicting future or past measurements, Kalman filters can be used to generate valid data at a common point in time, even for non-synchronized sensors.

Sensor Fusion The Kalman filter is able to integrate measurements from different sensors and perform data fusion by weighting measurements according to their covariance matrix. Also the integration of relative quantities from accelerometers or gyroscopes is possible, though some theoretical problems remain (see future work).

Estimating Static Relationships By using a special motion model, a Kalman filter effectively becomes a recursive least-squares estimator, which can be used to estimate constant transformations from multiple observations.

10.2 Lessons Learned

The setup of ubiquitous tracking system is more difficult than that of monolithic AR applications, because all components must handle measurement attributes such as timestamps and error covariance matrices carefully. All timestamps must accurately describe the moment a measurement was made and not the time of some subsequent processing or transmission. This also requires the careful synchronization of system clocks.

Once the basic principles are understood, the mathematics required for error propagation and Kalman filters is not difficult. However, the resulting Jacobian matrices may become rather big and cumbersome to handle. A remedy for this may come in form of the unscented transform and the unscented Kalman filter [**?**].

When used for prediction of measurements from good sensors that are almost noise-free, the Kalman filter does not perform significantly better than a simple linear extrapolator. However, for really bad measurements, such as the orientation values of the AR Toolkit, the linear extrapolation is virtually unusable, while the Kalman filter at least delivers predictions which are not worse than doing no prediction at all.

10.3 Future Work

The thesis finishes with an outlook to possible future directions of research by listing a number of problems which were identified during the work and deserve further attention.

10.3.1 Code Optimizations

While the current code does run sufficiently fast to compute multiple Kalman filters simultaneously on a modern portable computer, there still are numerous possible optimizations to make in run faster. Many matrices used for error propagation and Kalman filtering contain submatrices which either are the identity or zero. By using block matrix operations, these steps could be significantly accelerated.

10.3.2 Adding Motion Models to the Spatial Relationship Graph

In chapter 6, the parameters used in the dynamic construction of Kalman filters have been separated into measurement attributes, which describe properties of a sensor measurement,

and motion models, which describe the dynamics of the relation between two objects. In the current DWARF implementation, for simplicity, motion models also are sensor attributes, which violates this separation. The problem could be solved by adding the motion models as distinct edges to the spatial relationship graph. In fact, this would not even be a misuse of the spatial relationship graph concept, because a motion model also describe a certain kind of geometrical relationship between two objects.

Computing inferred motion models as new edges also is thinkable. From knowing that the relationship between a room and a desk is static as well as the relationship between the room and a lamp, the Ubitrack system can also infer a static relationship between the desk and the lamp. Even more sophisticated transformations could be possible for inferring the motion properties of features detected by a head-mounted camera from the dynamic properties of the user's head rotation.

10.3.3 Integration of Inertial Sensors

Previous work (e.g. [**?**]) has shown that the integration of inertial sensors significantly increases the accuracy of prediction. But even for the simple case of an ART-tracked HMD with an attached inertial tracker it is not easy to decide where to insert the edge of the inertial measurements into the Spatial Relationship Graph. Strictly speaking, an inertial tracker does measure the relative motion of its own coordinate frame, so it would make sense to add the measurements as a loop. Such a loop however, would not help the UMA in finding a path between the inertial tracker and the ART system for performing data fusion. Alternatively, the tracker could make its measurements relative to the "World" or some other fixed object, but in this case the UMA also would need to know that the ART system is at a fixed location, e.g. by a motion model edge. In any case, the addition of inertial trackers does require special logic, as it cannot be dealt with using solely the distributed path search.

10.3.4 Error Representations

The concepts described in this thesis rely on exactly known positions and orientations which are deteriorated by Gaussian measurement noise. However, many sensors used e.g. in ubiquitous computing can only tell if an object is in a certain area, but not where. For such cases, other methods of error representation have to be used. In robotics and recent Ubicomp applications, particle filters [**?**] have become a popular tool for describing arbitrary probability distributions. Particle filters represent all possible locations using a high number (often a few hundred) of representative samples. Also the integration of inertial sensors is possible, as shown by [**?**] with very promising results.

10.3.5 Using Time Error

The current PoseData structure also includes a field for "time error", which is defined to be the standard deviation of the measurement time. In the current implementation of the error model, this information is not used, but it might be useful for modelling an increase in the absolute error at high velocities. How this value can be determined reasonably and how to exploit this information in the computations could be the subject of future research.

10.3.6 Estimating Sensor Delay

As I have already explained in the previous chapter, precise timestamps are necessary for good prediction accuracy. Because it is not always possible to get well-synchronized timestamps from all sensors, the delay between the measurement and the arrival on the host computer sometimes must be determined by other means. Previous work [**?**] uses a manual method by adjusting the delay between two sensors until the observed motions on the screen match. However, in Ubitracksystems it should be possible to estimate the delay automatically from motion characteristics, when the same object is tracked by two sensors.

10.3.7 Estimating Sensor Error

For the measurements used in the evaluation of this thesis, the covariance matrices of the measurements were more or less guessed. For future Ubitrackapplications, it should be investigated, if it is possible to determine the sensor accuracy automatically in a running system, possibly by using reference measurements or by computing the covariance from sequences of slow motion.

10.3.8 Autocalibration

A big source for static tracking errors is the *static field distortion* (see section 3.1.1), which can be removed by calibration. In optical systems, this is usually caused by intrinsic camera parameters, such as focal length or lens distortion, which are either completely unconsidered or the exact values are unknown. In Ubitrack systems, were redundant tracking may be available at certain places, this could obviously be exploited to compute the distortion parameters automatically.

In this thesis, an approach for measuring the rotation and translation of static edges has already been described, but for other parameters like lens distortion, different techniques are necessary. The approach described in [**?**] temporarily includes the sensor parameters into the Kalman filter state vector. The problem is, that also the measurement and/or time update functions have to be modified to model the effects caused by the calibration parameters. Therefore, for the general case, it will be necessary to annotate each calibration parameter in a Ubitrack system with a formula that describes how it distorts the measurements.

10.3.9 Build Bigger Setups

In order to allow for better real-world evaluations of the methods described in this thesis, bigger tracking setups with a higher number and a higher variety of sensors are required. It also might be interesting to see how existing multi-tracker setups from the literature can be duplicated in Ubitrack systems.

Bibliography

[1] A. AUER and A. PINZ, *Building a Hybrid Tracking System: Integration of Optical and Magnetic Tracking*, in Proceedings of the International Workshop on Augmented Reality, 1999, pp. 13–22.

[2] R. AZUMA, *Predictive Tracking for Augmented Reality*, PhD thesis, University of North Carolina at Chapel Hill, 1995.

[3] R. AZUMA, *A Survey of Augmented Reality*, Presence: Teleoperators and Virtual Environments, 6 (1997), pp. 355–385.

[4] R. AZUMA, Y. BAILLOT, R. BEHRINGER, S. FEINER, S. JULIER, and B. MACINTYRE, *Recent Advances in Augmented Reality*, IEEE Computer Graphics Applications, 21 (2001), pp. 34–47.

[5] R. AZUMA and G. BISHOP, *Improving Static and Dynamic Registration in an Optical See-through HMD*, in Proc. Siggraph'94, Orlando, July 1994, pp. 194–204.

[6] R. AZUMA, B. HOFF, H. NEELY III, and R. SARFATY, *A Motion-Stabilized Outdoor Augmented Reality System*, in Proceedings of the IEEE Virtual Reality, IEEE Computer Society, 1999, p. 252.

[7] P. BAHL and V. N. PADMANABHAN, *RADAR: An In-Building RF-Based User Location and Tracking System*, in IEEE INFOCOM (2), March 2000, pp. 775–784.

[8] H. BALTZAKIS and P. TRAHANIAS, *A Hybrid Framework for Mobile Robot Localization: Formulation Using Switching State-Space Models*, Autonomous Robots, 15 (2003), pp. 169–191.

[9] M. BAUER, B. BRUEGGE, G. KLINKER, A. MACWILLIAMS, T. REICHER, S. RISS, C. SANDOR, and M. WAGNER, *Design of a Component-Based Augmented Reality Framework*, in Proceedings of the 2nd IEEE and ACM International Symposium on Augmented Reality (ISAR 2001), New York, NY, October 2001, IEEE Computer Society.

[10] R. BEHRINGER, *Registration for Outdoor Augmented Reality Applications Using Computer Vision Techniques and Hybrid Sensors*, in Proceedings of the IEEE Virtual Reality, IEEE Computer Society, 1999, p. 244.

[11] D. BEYER, *Construction of Decentralized Dataflow Graphs within Ubiquitous Tracking Environments*, Master's thesis, Technische Universität München, 2004.

[12] G. BISHOP, G. WELCH, and B. D. ALLEN, *Tracking: Beyond 15 Minutes of Thought*. `http://www.cs.unc.edu/~tracker/media/pdf/SIGGRAPH2001_CoursePack_11.pdf`, 2001. SIGGRAPH Course Pack.

[13] J. BORENSTEIN, H. R. EVERETT, and L. FENG, *Where am I? Sensors and Methods for Mobile Robot Positioning*. `http://www-personal.engin.umich.edu/~johannb/position.htm`, April 1996. Report Prepared by the University of Michigan for the Oak Ridge National Lab D&D Program and the United States Department of Energy.

[14] K. BRAMMER and G. SIFFLING, *Kalman-Bucy-Filter: Deterministische Beobachtung und stochastische Filterung*, Oldenbourg, München; Wien, 3rd ed., 1989.

[15] K. BRAMMER and G. SIFFLING, *Stochastische Grundlagen des Kalman-Bucy-Filters: Wahrscheinlichkeitsrechnung und Zufallsprozesse*, Oldenbourg, München; Wien, 3rd ed., 1990.

[16] B. BRUEGGE and A. H. DUTOIT, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[17] A. BUTZ, J. BAUS, and A. KRUGER, *Augmenting buildings with infrared information*, in Proceedings of the International Symposium on Augmented Reality ISAR 2000, October 2000, pp. 93–96.

[18] P. CASTRO, P. CHIU, T. KREMENEK, and R. R. MUNTZ, *A Probabilistic Room Location Service for Wireless Networked Environments*, in Proceedings of the 3rd international conference on Ubiquitous Computing, Springer-Verlag, 2001, pp. 18–34.

[19] L. CHAI, K. NGUYEN, B. HOFF, and T. VINCENT, *An Adaptive Estimator for Registration in Augmented Reality*, in Proceedings of Second IEEE and ACM Int'l Workshop on Augmented Reality (IWAR) '99, October 1999.

[20] 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.*, in Proceedings of the First International Workshop on Augmented Reality, November 1998.

[21] Y. CHO, J. PARK, and U. NEUMANN, *Fast Color Fiducial Detection and Dynamic Workspace Extension in Video See-through Self-Tracking Augmented Reality*, in Fifth Pacific Conference on Computer Graphics and Applications, 1997, pp. 168–177.

[22] J. C. K. CHOU, *Quaternion Kinematic and Dynamic Differential Equations*, IEEE Transactions on Robotics and Automation, 8 (1992), pp. 53–64.

[23] E. M. COELHO and B. MACINTYRE, *High-level Tracker Abstractions for Augmented Reality System Design*, in International Workshop on Software Technology for Augmented Reality Systems, October 2003, pp. 12–15.

[24] P. H. DANA, *Global Positioning System Overview*. `http://www.colorado.edu/geography/gcraft/notes/gps/gps_f.html`.

[25] A. DAVISON, W. MAYOL, and D. MURRAY, *Real-Time Localisation and Mapping with Wearable Active Vision*, in Proceedings of IEEE International Symposium on Mixed and Augmented Reality, IEEE Computer Society Press, October 2003.

[26] T. DRUMMOND and R. CIPOLLA, *Real-Time Visual Tracking of Complex Structures*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 24 (2002), pp. 932–946.

[27] *DWARF Project Homepage*. `http://www.augmentedreality.de`.

[28] F. EDDINE ABABSA, J. Y. DIDIER, M. MALLEM, and D. ROUSSEL, *Head Motion Prediction in Augmented Reality Systems Using Monte Carlo Particle Filters*, in Proceedings of the 13th International Conference on Artificial Reality and Telexixtence (ICAT 2003), December 2003.

[29] M. L. ERIK B. DAM, MARTIN KOCH, *Quaternions, Interpolation and Animation*, tech. rep., Department of Computer Science, University of Copenhagen, 1998.

[30] S. FEINER, B. MACINTYRE, T. HÖLLERER, and A. WEBSTER, *A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment*, in Proceedings of the International Symposium on Wearable Computing (ISWC'97), October 1997, pp. 74–81.

[31] M. A. FISCHLER and R. C. BOLLES, *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography*, Commun. ACM, 24 (1981), pp. 381–395.

[32] D. FOX, J. HIGHTOWER, L. LIAO, D. SCHULZ, and G. BORRIELLO, *Bayesian Filtering for Location Estimation*, IEEE Pervasive Computing, (2003).

[33] E. FOXLIN, *Inertial Head-Tracker Sensor Fusion by a Complimentary Separate-Bias Kalman Filter*, in Proceedings fo the 1996 Virtual Reality Annual International Symposium (VRAIS 96), March 1996, p. 185.

[34] E. FOXLIN and M. HARRINGTON, *WearTrack: A Self-Referenced Head and Hand Tracker for Wearable Computers and Portable VR*, in Proceedings of International Symposium on Wearable Computers (ISWC 2000), October 2000.

[35] A. GELB, *Applied Optimal Estimation*, MIT Press, Cambridge, Massachusetts; London, 15th ed., 1974.

[36] Z. GHAHRAMANI and G. E. HINTON, *Variational Learning for Switching State-Space Models*, Neural Computation, 12 (2000).

[37] F. S. GRASSIA, *Practical Parameterization of Rotations Using the Exponential Map*, jgt, The Journal of Graphics Tools, 3 (1998).

[38] D. HALLAWAY, T. HÖLLERER, and S. FEINER, *Bridging the Gaps: Hybrid Tracking for Adaptive Mobile Augmented Reality*, Applied Artificial Intelligence, Special Edition on Artificial Intelligence in Mobile Systems, 25 (2004).

[39] J. HIGHTOWER and G. BORRIELLO, *A Survey and Taxonomy of Location Systems for Ubiquitous Computing*, Tech. Rep. UW-CSE 01-08-03, University of Washington, Computer Science and Engineering, August 2001.

[40] T. HÖLLERER, D. HALLAWAY, N. TINNA, and S. FEINER, *Steps Toward Accommodating Variable Position Tracking Accuracy in a Mobile Augmented Reality System*, in Second Int. Workshop on Artificial Intelligence in Mobile Systems, August 2001, p. 31.

[41] W. HOFF and T. VINCENT, *Analysis of Head Pose Accuracy in Augmented Reality*, IEEE Transactions on Visualization and Computer Graphics, 6 (2000), pp. 319–334.

[42] W. A. HOFF and K. NGUYEN, *Computer vision-based registration techniques for augmented reality*, in Proceedings of Intelligent Robots and Computer Vision XV, November 1996, pp. 538–548.

[43] M. C. JACOS, M. A. LIVINGSTON, and A. STATE, *Managing Latency in Complex Augmented Reality Systems*, in Proceedings of the 1997 Symposium on Interactive 3D Graphics, April 1997.

[44] S. JULIER and J. UHLMANN, *A General Method for Approximating Nonlinear Transformations of Probability Distributions*, tech. rep., Dept. of Engineering Science, University of Oxford, November 1996.

[45] S. JULIER and J. UHLMANN, *Unscented Filtering and Nonlinear Estimation*, Proceedings of the IEEE, 92 (2004), pp. 401–422.

[46] H. KATO and M. BILLINGHURST, *Marker Tracking and HMD Calibration for a Video-Based Augmented Reality Conferencing System*, in Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality, IEEE Computer Society, 1999, p. 85.

[47] G. KLEIN and T. DRUMMOND, *Tightly Integrated Sensor Fusion for Robust Visual Tracking*, in Proc. British Machine Vision Conference 2002, vol. 2, BMVA, September 2002, pp. 787–796.

[48] K.-R. KOCH, *Parameterschätzung und Hypothesentests in linearen Modellen*, Dümmler, Bonn, 1980.

[49] D. KOLLER, G. KLINKER, E. ROSE, D. BREEN, R. WHITAKER, and M. TUCERYAN, *Real-time Vision-Based Camera Tracking for Augmented Reality Applications*, in Proceedings of the Symposium on Virtual Reality Software and Technology (VRST-97), September 1997, pp. 87–94.

[50] M. KOUROGI and T. KURATA, *Personal Positioning based on Walking Locomotion Analysis with Self-Contained Sensors and a Wearable Camera*, in Proceedings of the Second IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR '03), IEEE Computer Society, 2003.

[51] H. KUCHLING, *Taschenbuch der Physik*, Fachbuchverlag Leipzig im Carl Hanser Verlag, München; Wien, 16th ed., 1999.

[52] F. LEDERMANN, G. REITMAYR, and D. SCHMALSTIEG, *Dynamically Shared Optical Tracking*, in Proceedings of the IEEE First International Workshop on ARToolKit, Available as technical report TR-188-2-2002-12, Vienna University of Technology, 2002.

[53] M. A. LIVINGSTON and A. STATE, *Magnetic Tracker Calibration for Improved Augmented Reality Registration*, PRESENCE: Teleoperators and Virtual Environments, 6 (1997).

[54] N. A. LYNCH, *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc, 1996.

[55] U. NEUMANN, S. YOU, Y. CHO, J. LEE, and J. PARK, *Augmented Reality Tracking in Natural Environments*, in International Symposium on Mixed Realities, 1999.

[56] J. NEWMAN, D. INGRAM, and A. HOPPER, *Augmented Reality in a Wide Area Sentient Environment*, in Proc. of IEEE and ACM Int. Symp. on Augmented Reality (ISAR 2001), New York, NY, October 2001, pp. 77–86.

[57] J. NEWMAN, M. WAGNER, T. PINTARIC, A. MACWILLIAMS, M. BAUER, G. KLINKER, and D. SCHMALSTIEG, *Fundamentals of Ubiquitous Tracking for Augmented Reality*, Tech. Rep. TR-188-2-2003-34, Vienna University of Technology, 2003.

[58] OBJECT MANAGEMENT GROUP, *Common Object Request Broker Architecture (CORBA/IIOP)*, March 2004. current revision 3.0.3.

[59] W. PIEKARSKI, B. AVERY, B. H. THOMAS, and P. MALBEZIN, *Hybrid Indoor and Outdoor Tracking for Mobile 3D Mixed Reality*, in Proceedings of the Second IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR '03), IEEE Computer Society, 2003.

[60] L. RÅDE and B. WESTERGREN, *Springers mathematische Formeln*, Springer Verlag, Berlin; Heidelberg; New York, 2nd ed., 1997.

[61] C. RANDELL, C. DJIALLIS, and H. MULLER, *Personal Position Measurement Using Dead Reckoning*, in Proceedings of the Seventh IEEE International Symposium on Wearable Computers, 2003, p. 166.

[62] G. REITMAYR and D. SCHMALSTIEG, *OpenTracker - An Open Software Architecture for Reconfigurable Tracking based on XML*, in Proceedings of the Virtual Reality 2001 Conference (VR'01), March 2001, p. 285.

[63] M. RIBO, *State of the Art Report on Optical Tracking*, Tech. Rep. TR VRVis 2001 025, VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH, Vienna, Austria, September 2001.

[64] K. RÖMER and S. DOMNITCHEVA, *Smart Playing Cards: A Ubiquitous Computing Game*, Springer/ACM Personal and Ubiquitous Computing, 6 (2002), pp. 371–378.

[65] C. SANDOR, A. MACWILLIAMS, M. WAGNER, M. BAUER, and G. KLINKER, *SHEEP: The Shared Environment Entertainment Pasture*, in IEEE and ACM International Symposium on Mixed and Augmented Reality ISMAR 2002, 2002.

[66] K. SHOEMAKE, *Animating Rotation with Quaternion Curves*, Computer Graphics (Proceedings of SIGGRAPH'85), 19 (1985), pp. 245–254.

[67] P. SMITH, T. DRUMMOND, and K. ROUSSOPOULOS, *Computing MAP trajectories by representing, propagating and combining PDFs over groups*, in Proceedings of the 9th IEEE International Conference on Computer Vision, vol. II, October 2003, pp. 1275–1282.

[68] A. STATE, G. HIROTA, D. T. CHEN, W. F. GARRETT, and M. A. LIVINGSTON, *Superior augmented reality registration by integrating landmark tracking and magnetic tracking*, in Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, ACM Press, 1996, pp. 429–438.

[69] F. STRASSER, *Bootstrapping of Sensor Networks in Ubiquitous Tracking Environments*, Master's thesis, Technische Universität München, 2004.

[70] I. E. SUTHERLAND, *A Head-Mounted Three Dimensional Display*, in Proceedings of the 1968 Fall Joint Conference, vol. 33, 1968.

[71] R. M. TAYLOR, II, T. C. HUDSON, A. SEEGER, H. WEBER, J. JULIANO, and A. T. HELSER, *VRPN: a device-independent, network-transparent VR peripheral system*, in Proceedings of the ACM symposium on Virtual reality software and technology, ACM Press, 2001, pp. 55–61.

[72] B. THOMAS, B. CLOSE, J. DONOGHUE, J. SQUIRES, P. D. BONDI, M. MORRIS, and W. PIEKARSKI, *ARQuake: An Outdoor/Indoor Augmented Reality First Person Application*, in Proceedings of the Fourth International Symposium on Wearable Computers (ISWC'00), October 2000.

[73] R. Y. TSAI, *A versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses*, IEEE Journal of Robotics and Automation, RA-3 (1987), pp. 323–344.

[74] M. TUCERYAN, Y. GENC, and N. NAVAB, *Single-Point active alignment method (SPAAM) for optical see-through HMD calibration for augmented reality*, Presence: Teleoperators and Virtual Environments, 11 (2002), pp. 259–276.

[75] R. WANT, A. HOPPER, V. FALCAO, and J. GIBBONS, *The Active Badge Location System*, ACM Trans. Inf. Syst., 10 (1992), pp. 91–102.

[76] A. WARD, A. JONES, and A. HOPPER, *A New Location Technique for the Active Office*, IEEE Personal Communications, 4 (1997), pp. 42–47.

[77] M. WARD, R. AZUMA, R. BENNETT, S. GOTTSCHALK, and H. FUCHS, *A demonstrated optical tracker with scalable work area for head-mounted display systems*, in Proceedings of the 1992 symposium on Interactive 3D graphics, ACM Press, 1992, pp. 43–52.

[78] M. WEISER, *The Computer for the 21st Century*, Scientific American, (1991).

[79] G. WELCH and G. BISHOP, *An Introduction to the Kalman Filter*, Tech. Rep. TR 95-041, Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.

[80] G. WELCH, G. BISHOP, L. VICCI, S. BRUMBACK, K. KELLER, and D. COLUCCI, *The HiBall Tracker: High-Performance Wide-Area Tracking for Virtual and Augmented Environments*, in Proceedngs of the ACM Symposium on Virtual Reality Software and Technology 1999 (VRST 99), December 1999.

[81] G. WELCH and E. FOXLIN, *Motion Tracking: No Silver Bullet, but a Respectable Arsenal*, IEEE Computer Graphics and Applications, 22 (2002), pp. 24–38.

[82] G. F. WELCH, *SCAAT: incremental tracking with incomplete information*, PhD thesis, University of North Carolina at Chapel Hill, 1996.

[83] S. YOU and U. NEUMANN, *Fusion of Vision and Gyro Tracking for Robust Augmented Reality Registration*, in Proceedings of the Virtual Reality 2001 Conference (VR'01), IEEE Computer Society, 2001, p. 71.

[84] S. YOU, U. NEUMANN, and R. AZUMA, *Hybrid Inertial and Vision Tracking for Augmented Reality Registration*, in Proceedings of the IEEE Virtual Reality, IEEE Computer Society, 1999, p. 260.

[85] M. A. YOUSSEF, A. AGRAWALA, and A. U. SHANKAR, *WLAN Location Determination via Clustering and Probability Distributions*, in Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom) 2003, March 2003, p. 143.

Index