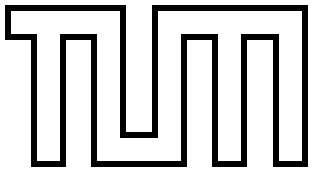
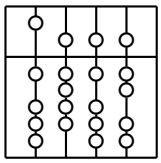


Technische Universität München
Fakultät für Informatik



Systementwicklungsprojekt

Empirical estimation of tracking ranges and application thereof for smooth transition between two tracking devices

Sven Hennauer

Aufgabenstellerin: Univ.-Prof. Gudrun Klinker, Ph. D.

Betreuer: Dipl.-Inf. Martin Wagner

Abgabedatum: 13. Januar 2004

Erklärung

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

München, den 13.01.2004 Sven Hennauer

Abstract

Many augmented reality applications face the problem that the tracking devices being used have limited working areas and therefore do not provide sufficient coverage. If one wants to use multiple trackers in order to extend the overall tracking area, the problem arises how to combine them, so that a smooth transition is obtained while moving from one tracking area to the other.

This thesis deals with this question and proposes two different transition strategies, which do not depend on prior knowledge of specific properties or the setup of the trackers, but instead are able to adapt to the respective tracking areas. Both strategies, being based on convex hulls and neural networks respectively, have been implemented prototypically and have been embedded into the DWARF framework. The design of this implementation allows future developers to add new transition strategies easily.

Contents

Chapter 1

Introduction

This document is a result of my 'Systementwicklungsprojekt' (SEP) conducted at the chair for applied software engineering at the TUM. In this chapter, the task to be solved is motivated, followed by a short overview of related research activities.

1.1 Motivation

In augmented reality applications, the user's view of the reality is augmented with a virtual scene providing additional information. To be able to align the virtual with the real world, so-called trackers continuously have to determine the position and orientation of the user and other objects. Unfortunately, the currently available tracking devices - especially those for indoor use - often have a limited tracking area which is not sufficient for many applications. A simple and obvious idea to solve this problem is to combine multiple (homogeneous or heterogeneous) trackers to extend the overall coverage. An important problem to be solved here is how to manage the transition from one tracker to the other when an object moves across the different tracking areas. Switching from one tracker to the other at a predefined position would lead to a discontinuous transition since both trackers are afflicted with inaccuracies, which for many tracking devices is especially true near the boundaries of their working area (see figure 1.1). Therefore it is crucial for the effective combination of multiple trackers to have a strategy which guarantees a smooth transition between the tracking areas. Furthermore, a solution is desirable for which no prior knowledge of the respective areas is needed and which instead is able to adapt easily to different tracking setups. This SEP deals with this problem and presents a generic, machine learning based solution which is embedded into the DWARF framework for distributed and wearable augmented reality applications. Two transition strategies have been implemented prototypically, one based on convex hulls, the other using a neural network.

This document consists of 8 chapters. After this introduction, chapter 2 gives an overview of the current infrastructure into which this project is to be embedded, namely the available tracking devices and the software architecture. After a summary of the functional and nonfunctional requirements the system design is depicted in chapter 4. It

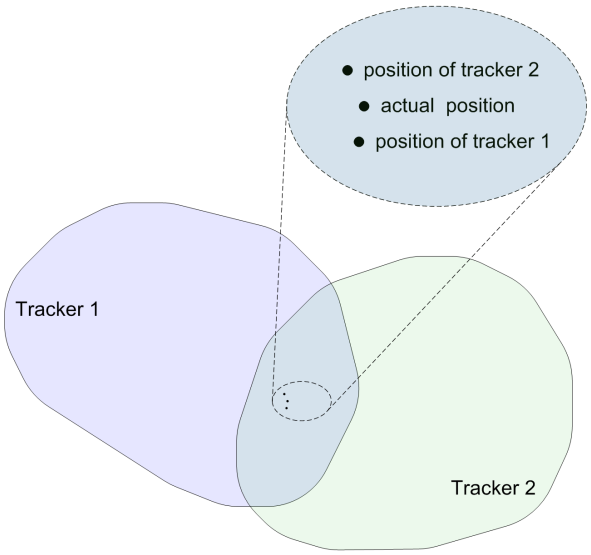


Figure 1.1: Inaccuracies of tracking devices

becomes obvious here that some preparatory tasks such as integrating and calibrating the tracking devices are necessary, which will be described in chapter 5. The subsequent chapters can be regarded as the main part of this SEP and contain the general object design as well as a detailed explanation of the developed transition strategies. Finally, the results and hints for future work are presented.

1.2 Related work

There are many ongoing research activities dealing with the combination of multiple trackers. However, they primarily focus on increasing the tracking accuracy or robustness by using different trackers (hybrid tracking). Popular approaches in this field of research are to combine inertial or magnetic with either vision-based tracking ([1], [2]) or - especially for outdoor use - GPS tracking ([3]). In both cases the shortcomings of a technology (lack of realtime facilities with vision-based, lack of orientation data with GPS, lack of precision and stability of inertial tracking) are compensated by the advantages of the other one (precision and stability with vision-based and GPS, fast orientation data with inertial tracking). Other research projects try to combine arbitrary tracking devices described by quality of service parameters to increase tracking accuracy ([4]).

Note that none of these approaches aims at extending the overall tracking area. Instead, the used trackers all cover the same area and track objects simultaneously to exploit their specific properties. In contrast to that, this SEP deals with almost arbitrary tracking devices which consciously cover different regions and overlap only to a small extent. The only assumption being made on the trackers is that they actually have limited working areas, which is the case for most tracking technologies.

Chapter 2

Current infrastructure

This project has to be embedded into the infrastructure which is currently available at the chair for applied software engineering. It is introduced in the following sections in terms of hardware, i.e. the tracking devices used, and software.

2.1 Hardware

The following sections give an overview of the two tracking devices which were used for developing and testing the system. Both are commercially available and are currently installed in the AR laboratory. Although both trackers deliver 6DOF data, i.e. both position and orientation, only the position is used within this project since this information is sufficient to obtain a smooth transition. Note that these devices only serve as a testbed and could be replaced with arbitrary trackers having limited working areas since the developed system does not depend on their specific properties.

2.1.1 ART tracker

The ARTtrack1 / DTrack system of Advanced Realtime Tracking GmbH (ART, [5]) consists of at least two special-purpose cameras which are rigidly mounted on a frame surrounding the desired tracking area and continuously emit infrared flashes. The objects to be tracked are equipped with a set of at least four retro-reflecting markers which can be recognized by the CCD sensors of the cameras. Each camera computes the position of the markers in 2D-coordinates and transfers this data to the central DTrack PC running under Windows NT, which finally calculates the position and orientation of the object. This system is able to determine position data with sub-millimetre precision within a working area of about four times four metres. For further reference, please consult [6] or the documentation released by the manufacturer ([7]).

2.1.2 InterSense tracker

As another tracking device, we use an IS-600 Mark 2 by InterSense ([8]), a hybrid acousto-inertial tracker which combines an ultrasonic time-of-flight range measuring system with gyroscopes and accelerometers. The following description is extracted from [9] which also can provide additional information if necessary.

In this system, one or more so-called SoniDiscs have to be attached to the object to be tracked. A SoniDisc is a wireless transponder which receives infrared signals and transmits ultrasonic pulses in response. Additionally, at least four so-called ReceiverPods have to be mounted rigidly above the tracking area, preferably at the ends of an X-Bar which provides a fixed frame for positioning them at precise spacing. They broadcast an infrared trigger code which uniquely identifies one of the SoniDiscs. The selected disc responds with an ultrasonic pulse, which in turn is received by the ReceiverPods' microphones. By using the speed of sound, range measurements for each of the ReceiverPods are obtained and the position of the disc can be computed. Afterwards, the next disc is activated by another infrared transmission. Note that with at least two SoniDiscs fixed to the object, its orientation can be calculated as well. Furthermore, a so-called InertiaCube measuring angular rates, linear accelerations and magnetic field components along all three axes can be attached to the object to obtain orientation information. If at least two SoniDiscs are used, the measurements of the InertiaCube are combined with those of the ultrasonic system by applying an extended Kalman filter in order to increase accuracy and stability of both position and orientation data (see [9, p. 13]).

The IS-600 claims to have a usable tracking area of about three times three metres. Own tests showed its ability to retrieve the position far outside of this region, however at the cost of less accuracy and stability. While approaching the boundary of the actual tracking area, the system often suffered from severe outliers sometimes missing the correct position by several metres.

2.2 Software

The system to be developed should be embedded into DWARF, a component-based framework for distributed augmented reality applications. In the following, only the key concepts of DWARF which are essential for this SEP are mentioned, for details see [10] or [11].

The basic concepts of the DWARF framework are distributed services, needs and abilities. Each service runs on a stationary or mobile computer and provides a predefined functionality such as position tracking or 3D rendering. Its interfaces to other services are described by means of so-called abilities (i.e. the functionality it provides) and needs (i.e. the functionality it depends on). A CORBA-based service manager running on each network node is responsible for connecting services with corresponding needs and abilities. Once connected, the services can exchange data via remote method calls, shared memory or event notifications. Since the latter one will be used extensively in this project, it shall be illustrated by a simple example. A service which uses a tracking device to determine

the position and orientation of an object has a `PoseData` ability and therefore continuously sends out `PoseData` events containing position and orientation data (i.e. the 'pose'). On the other hand, a 3D rendering service has a `PoseData` need, therefore receives the `PoseData` events and might interpret them either as an object's position or as the user's viewpoint (if the object is mounted on the user's head) in order to render the scene.

The needs and abilities of a service are described via XML configuration files. They also provide the possibility to handle predicates, specifying more detailedly which services are to be connected.

A service for accessing the ART tracker is already available. It receives the tracking data from the DTrack PC via UDP and sends out `PoseData` events in response.

Chapter 3

Requirements analysis

In the following, both the functional and nonfunctional requirements to be fulfilled are summarized. They specify how the system to be developed shall behave from the user's perspective (functional requirements) and what kind of quality constraints it shall comply with (nonfunctional requirements).

3.1 Functional requirements

The general objective of this SEP is to develop a strategy to combine two tracking devices in order to increase the overall tracking area. For this purpose, the trackers have to be installed in such a way that they cover different regions, but also overlap to a certain extent (see figure 1.1). Obviously, the main focus is on this overlapping area since only there the position has to be computed out of two independent tracking sources. Within this region, the position data from both trackers has to be mixed so that a smooth (i.e. continuous) transition is obtained while moving from one tracking area to the other. It is strongly desirable that the system does not depend on prior knowledge of the trackers' working areas because otherwise a change of their setup or installation location would make a time-consuming and error-prone measuring necessary. Instead, it should be able to adapt itself to, or - in other words - 'learn' the respective tracking areas. This could happen either in a separate phase before the actual application of the system (offline learning) or during its use (online learning).

The described system should be embedded into the DWARF framework. Therefore, a service (in the following called `MultiTracker`) has to be developed which receives the `PoseData` events from two tracking services and sends out a filtered `PoseData` event in response so that a continuous transition is obtained.

Obviously, different strategies are possible in order to obtain a smooth transition. To be able to get an impression of their functionality and to evaluate their effectiveness, a visualization of the algorithms is desirable.

3.2 Nonfunctional requirements

Apart from the functional requirements, some quality constraints concerning performance, flexibility, extendability and documentation have to be addressed. They will be described in the following.

3.2.1 Performance characteristics

It should be able to use the `MultiTracker` service in 'real' augmented reality applications. This requires that it can combine the position data of the two trackers efficiently. More precisely, the service has to be able to deliver the filtered `PoseData` events at a rate which is sufficient for realtime applications. However, depending on the transition strategy and its learning complexity, it may be necessary to perform the learning phase offline, i.e. before the actual use of the system.

3.2.2 Flexibility and extendability

As mentioned above, one can imagine several possible transition strategies. For the user of the `MultiTracker` service, switching to another strategy should be able by simply modifying the XML service description. For the developer, implementing a new strategy and integrating it into the service has to be as easy as possible.

3.2.3 Documentation

The documentation to be delivered with this SEP includes a developer manual (this document) and a user manual (see [11, Documentation]). Apart from that, the source code has to be documented properly so that in combination with this document future developers will be able to modify or extend the system.

Chapter 4

System design

This section describes the interactions between the `MultiTracker` and other DWARF services by means of needs and abilities. An example setup is shown in figure 4.1.

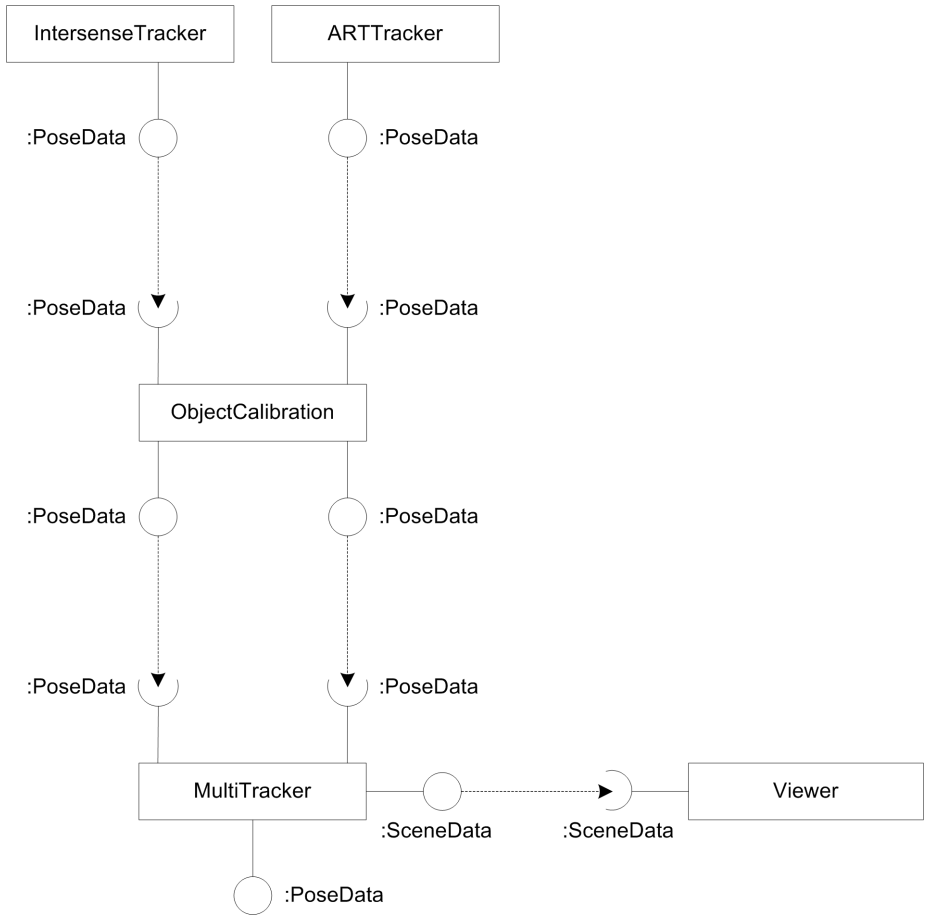


Figure 4.1: System design in terms of needs and abilities

The `MultiTracker` service has two `PoseData` needs in order to receive data from two tracking services, each providing a `PoseData` ability. Since those tracking services (e.g. `IntersenseTracker` and `ARTTracker`) deliver the pose in different coordinate systems, they have to be calibrated to a global coordinate system. This should be done by the `ObjectCalibration` service which therefore has to be interposed between the `MultiTracker` and the tracking services. Note that for this purpose one has to specify which services have to be connected by means of predicates in the XML service description since the tracking services should not interact directly with the `MultiTracker` service.

The `MultiTracker` combines data of two tracking sources and sends out the filtered data, i.e. it provides a `PoseData` ability which can be used by other DWARF services. For visualization purposes, the `Viewer` service ([12]) is used. Since it depends on `SceneData` events to determine what to display, the `MultiTracker` has to provide a `SceneData` ability.

Chapter 5

Preparatory work

In order to develop and test the `MultiTracker` service, some preparatory tasks had to be performed. Since the InterSense tracker should be used, a DWARF service had to be implemented which could retrieve its position and orientation data and deliver them as `PoseData` events. Additionally, the problem how to calibrate two trackers computing the position in different coordinate systems had to be solved. Finally, a so-called `PoseDataLogger` service was developed which writes tracking data to a file and should help deriving possible transition strategies.

5.1 DWARF service for InterSense tracker

To be able to use the IS-600, a DWARF service was developed which continuously retrieves its position and orientation data and delivers them as `PoseData` events. The tracker is connected to the serial port of the computer hosting the service and can easily be accessed via a shared library released by InterSense. For sending the `PoseData` events, the class `PoseSenderService` is used, which provides an interface for setting and sending all relevant information like position, orientation, accuracy and timestamp and thus encapsulates the DWARF-specific tasks. Once the service is started, it enters a loop which retrieves the tracker data, sends it as a `PoseData` event and waits for 20 milliseconds which leads to an update rate of approximately 50Hz. Note that for future releases, it might make sense to specify the update rate in the XML service description. Unfortunately, the version of the shared library which was used in this project (version 3.44) offers no functionality to determine whether the object to be tracked is out of sight of the tracker. Instead, it keeps on emitting the last position which could be retrieved if a current position is not available. In the meantime, updated versions of the library have been released, but have not been tested yet since even InterSense could not state clearly if the firmware (version 3.0201) actually supports this feature.

The `PoseData` events to be sent expect the orientation data as quaternions ([13]). Since the IS-600 can be configured to deliver quaternion data, no explicit conversion concerning the orientation is necessary - apart from reordering the quaternion array from $[w, x, y, z]$ to

$[x, y, z, w]$. However, the origin of its coordinate reference frame is located at the X-Bar's centre hub and thus in general does not coincide with the origin of the other tracker being used (this also applies to the alignment of the reference frame). Therefore, the position data has to be transformed to a single, global coordinate reference frame. This is described in section 5.2.

Since this service uses a general InterSense library to access the tracker, it should work with other InterSense tracking devices such as the IS-300 or IS-900 as well. It was tested successfully with both InterSense trackers being available at the chair for applied software engineering, namely the IS-600 and the InterTrax. For details on how to use the service, please consult [11, Documentation].

5.2 Calibration

As mentioned in the previous section, the InterSense and ART tracker deliver the position data in different coordinate systems. To be able to apply a transition strategy, they have to be calibrated to a global coordinate system so that they retrieve the same position for an object being tracked by both devices. For this purpose, the `ObjectCalibration` service is supposed to be used, which in turn could store the information being necessary for the transformation in the `WorldModel` ([14]). Unfortunately, the `ObjectCalibration` service did not offer this functionality at the time of this SEP's development yet. As a workaround, the `IntersenseTracker` service was modified so that it sent out the `PoseData`s in the coordinate system of the ART tracker. This was achieved by applying the formula

$$\begin{pmatrix} x_{ART} \\ y_{ART} \\ z_{ART} \end{pmatrix} = c \cdot R \cdot \begin{pmatrix} x_{InterSense} \\ y_{InterSense} \\ z_{InterSense} \end{pmatrix} + \vec{t}$$

where the scaling factor c, the rotation matrix R and the translation vector $\vec{t}$ were determined out of a couple of corresponding ART and InterSense points and were hardcoded within the tracking service (many thanks to Jörg Traub who developed this method in the context of his diploma thesis, [15]).

5.3 PoseDataLogger

This is a simple service with one `PoseData` need which writes all `PoseData` events, namely the position, the orientation and the timestamp, to a file. The service whose events have to be logged as well as the name of the output file can be configured via the XML service description. For details concerning the use of this service, the reader is referred to [11, Documentation].

Chapter 6

Object design

The general object model of the `MultiTracker` service is depicted in figure 6.1. For the sake of clarity, the methods concerning DWARF-specific tasks (such as `startService`, `setConsumer`, etc.) are omitted in this class diagram (for details on those see [11, Documentation]), as well as those concerning the visualization, which is described separately in section 6.6. In the following, the classes being developed are explained in detail. Additionally, the interactions between them are clarified afterwards in section 6.5.

6.1 MultiTracker

This is the core component of the service. It uses two `MultiTrackerReceiver` objects to receive `PoseData` events from the two tracking services, an instance of a `MultiTrackerBase` subclass to combine them and a `PoseSender` object to send out the filtered position as a `PoseData` event.

The class implements the `BasicService_PushSupplier` interface and therefore is responsible for initializing and starting the service (the `PushSupplier` interface is needed for the visualization). During startup, the needs and abilities are retrieved from the XML service description. Two `MultiTrackerReceiver` objects each corresponding to a need (i.e. a tracking source) and one `PoseSender` object corresponding to an ability are created and registered with the service manager. Furthermore, the attributes `maxUpdateRate` specifying the maximum update rate and `TransitionType` specifying the transition strategy to be used are retrieved. Depending on the strategy, an instance of the respective `MultiTrackerBase` subclass (e.g. `MultiTrackerConvexHull`) is created by means of a strategy pattern ([16]). New `PoseData` events are received from the `MultiTrackerReceiver` objects via the `newPoseData` method which forwards the event to the respective transition strategy and sends out the filtered `PoseData` events. This mechanism is described in detail in section 6.5.

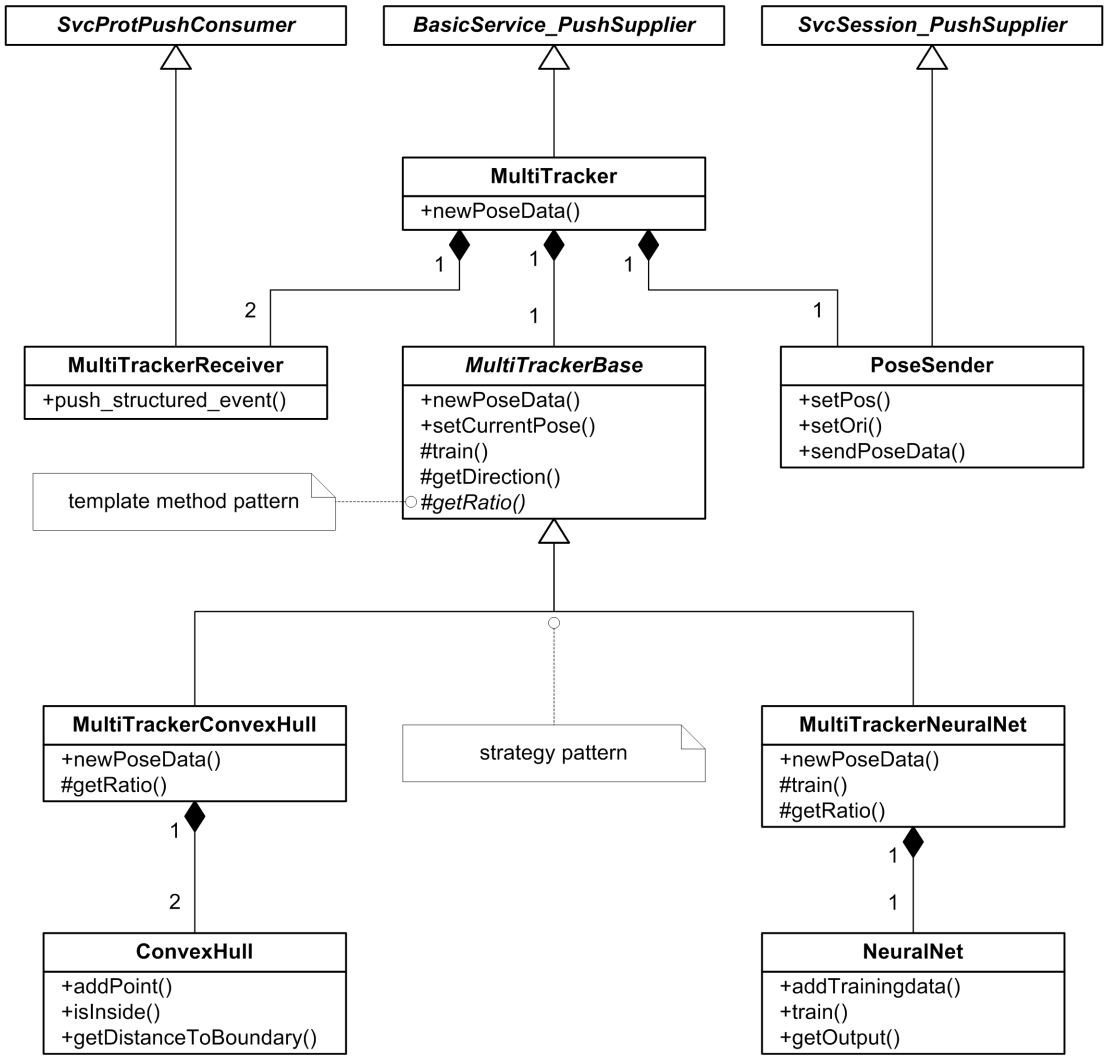


Figure 6.1: Object model of the `MultiTracker` service

6.2 MultiTrackerReceiver

This class is responsible for receiving `PoseData` events from a tracking service and forwarding them to the central `MultiTracker` class. For this purpose, it implements the `SvcProtPushConsumer` interface including the `push_structured_event` method which is called whenever a new event is available.

6.3 MultiTrackerBase

This abstract class serves as the base class for the different transition strategies. It provides the public methods `newPoseData` to indicate that a new `PoseData` event is available and

`setCurrentPose` to combine the `PoseData` events of both tracking services to a single position. In order to achieve flexibility and extendability, the strategy and template method patterns ([16]) are used.

The method `newPoseData` provides common functionality like storing the last `PoseData` events of both tracking services (to be able to compute the current moving direction) and starting training as soon as 1000 points have been received from each tracker (for offline learning only). Subclasses should override (and call) this method to perform tasks which are specific for the respective transition strategy. Furthermore, the `train` method has to be overridden for offline learning strategies, while online learning can be done in the subclasses' `newPoseData` method.

With the method `setCurrentPose`, the most recent `PoseData` events of both trackers are combined to a single position using the respective transition strategy. Obviously, such a strategy has to be applied only if current position data of both tracking devices is available. Therefore, the case of at least one obsolete (i.e. older than 0.1 seconds) `PoseData` event is handled first: with one obsolete position, the other one is returned as the current position, with both tracking data being obsolete, no current position can be obtained. If, however, both `PoseData` events are valid, the abstract method `getRatio` is called which has to be implemented by the subclasses (template method pattern) and has to return a value between 0 and 1 indicating at which ratio the `PoseData` events should be mixed. For example, a value of 0.7 would instruct the `setCurrentPose` method to take 70% of the first and 30% of the second tracker.

The `MultiTrackerBase` class additionally provides a protected method `getDirection` which returns the current moving direction (i.e. the average direction of both trackers) and can be used by subclasses whose transition strategy depends on this knowledge.

6.4 Transition-specific classes

The classes `MultiTrackerConvexHull` and `MultiTrackerNeuralNet` are prototypical and exemplary implementations of different transition strategies based on convex hulls and neural networks respectively. They are described in detail in the sections 7.1 and 7.2, as well as the classes `ConvexHull` and `NeuralNet` which they depend on.

6.5 Control and data flow

Figure 6.2 shows a typical control and data flow triggered by the arrival of a new `PoseData` event. A `MultiTrackerReceiver` object receives the event via the `push_structured_event` method and forwards it to the core `MultiTracker` component, which in turn forwards it to the object realizing the transition strategy (e.g. `MultiTrackerNeuralNet`). This object calls the `newPoseData` method of the base class in order to perform the common tasks described above and to trigger the learning phase by calling the template method `train` if the service is not trained yet and enough points (currently 1000) of both trackers have

been collected. Note that a service being capable of online learning is regarded as 'trained' from its startup and therefore does not have to implement the `train` method. Instead, the learning-specific tasks are to be done directly in the `newPoseData` method. After the `PoseData` was processed by the `MultiTrackerBase` subclass, the `MultiTracker` object calls `setCurrentPose`, which in turn usually uses the template method `getRatio` described above to retrieve the ratio at which the positions of both trackers have to be mixed. If the current `PoseData` could be determined, it is finally sent out by means of the `PoseSender` object. Retreiving and sending the current position is performed only if the last `PoseData` event was not delivered recently so that the desired maximum update rate is not exceeded.

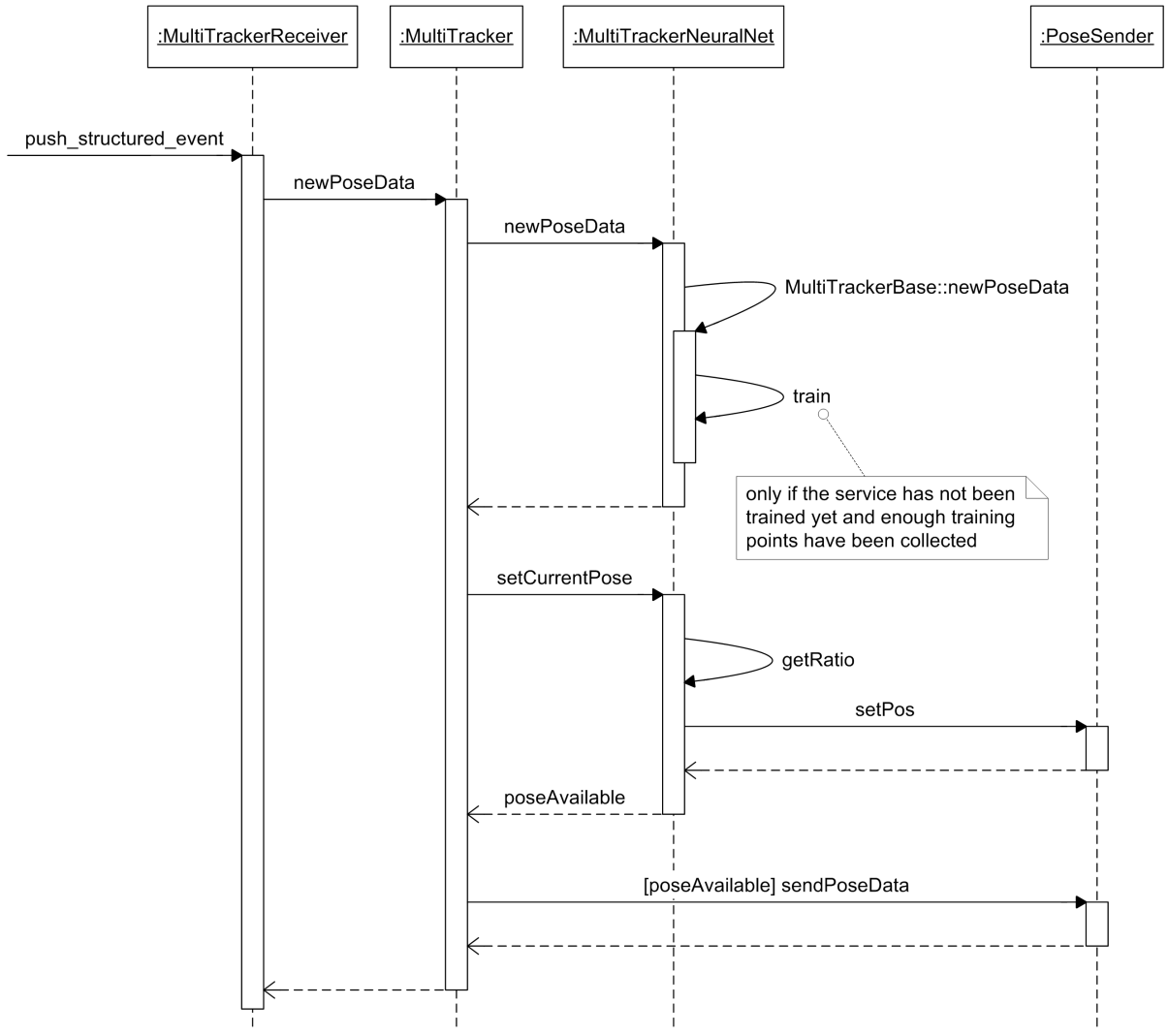


Figure 6.2: Control and data flow

6.6 Visualization

For visualization purposes, the service has to generate scene descriptions in the Open Inventor format and send them as `SceneData` events to the `Viewer` service. This scene creation is conducted in several steps in order to be able to incorporate different kind of information. This is illustrated in figure 6.3 while figure 6.4 depicts the methods of the above classes which are relevant for the visualization. The process is again triggered by a `MultiTrackerReceiver` object which forwards a `PoseData` event to the `MultiTracker` object. That creates a string which will receive the scene description and initializes it with some general scene data (currently, this is the room's door and some points indicating the location of the ART cameras in order to facilitate the orientation for the user). Afterwards, it requests the object implementing the transition strategy (e.g. `MultiTrackerConvexHull`) to append its scene data by calling `appendScene`. This method inserts two points indicating the current position of both trackers and calls the template method `getScene` to retrieve the scene description which is specific for the respective transition strategy. Finally, the `MultiTracker` object encapsulates the now complete scene description into a `SceneData` event and sends it out by calling the method `push_structured_event` of the `StructuredPushConsumer` object.

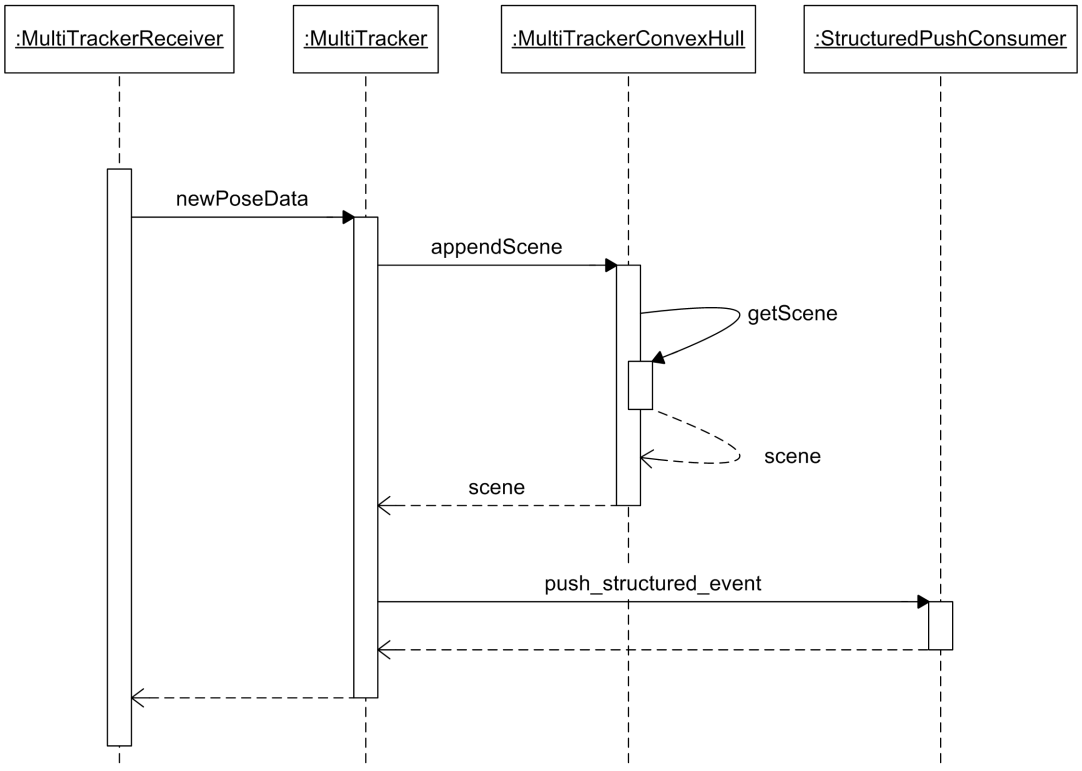


Figure 6.3: Control and data flow (visualization-specific)

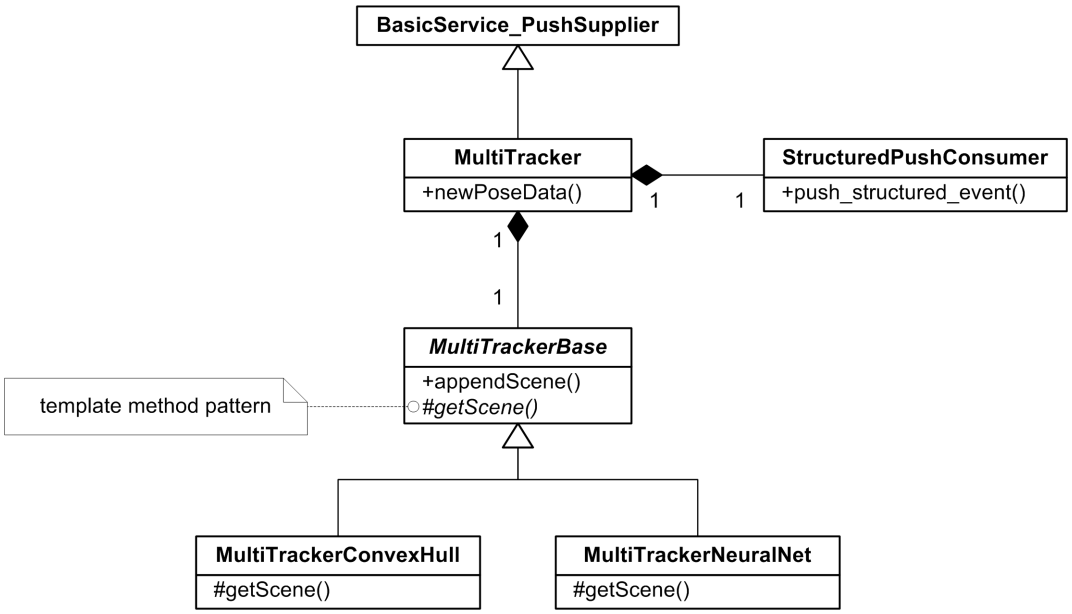


Figure 6.4: Object model (visualization-specific)

Since the scene creation can be a pretty time-consuming task, two optimizations are applied. First, the whole process is conducted only if a consumer for `SceneData` events is available. Second, the scene description generated by `getScene` is cached and is created only if the scene has changed since the last creation (note that the scene can remain unchanged although new points are received from the trackers, for example if all the new points are within the respective convex hulls).

Chapter 7

Transition strategies

During the course of this SEP, several ideas for possible transition strategies arose. Two of them were elaborated and implemented, the one being based on convex hulls, the other using a well-known concept in machine learning, namely neural networks.

7.1 Convex hull

This transition strategy aims at estimating the working areas of both trackers based on all `PoseData` events they delivered so far. For this purpose, we assume that if two points are visible for a tracker (i.e. inside the working area), all points on a straight line between them are in the tracking area as well. In other words, the tracking areas are supposed to be convex, which should at least approximately be true for most tracking devices. Therefore, the working area of a tracker can be represented by the convex hull of all the points it can retrieve. From this observation, the following transition strategy consisting of the learning and application phase is derived. The distinction of both phases is only conceptual since they can be conducted simultaneously (online learning).

7.1.1 Learning phase

During learning, the convex hulls of both tracking sources are computed. This is done incrementally, which means that whenever a new `PoseData` event is received, the respective convex hull is extended by this point. Since this computation can be conducted efficiently with the `ConvexHull` class described in section 7.1.3, this transition strategy is able to provide online learning.

7.1.2 Application phase

The actual transition is based on the idea that an interruption of tracking data is the more probable the nearer the object is to the boundary of the tracking area. As a consequence for mixing two tracking sources, the ratio of a tracker has to be decreased when we approach the boundary of its working area. This is illustrated in figure 7.1:

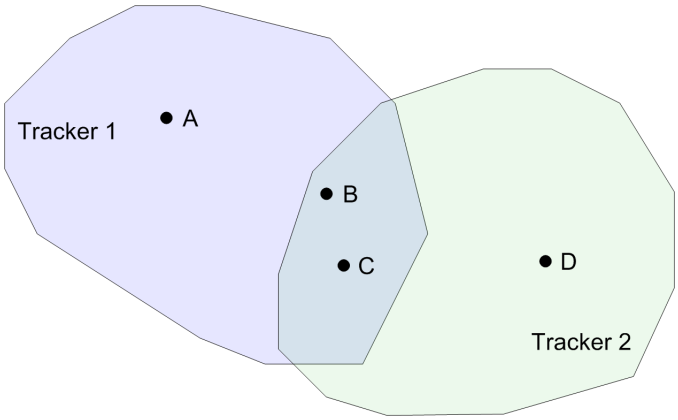


Figure 7.1: Convex hulls representing the tracking areas

- Point A (resp. D) is only in the convex hull / working area of tracker 1 (resp. 2), i.e. mixing is not necessary and the position of tracker 1 (resp. 2) can be returned.
- Point B is in both tracking areas, but near the boundary of tracker 2. Assuming the distance to this boundary is 10cm and to the boundary of tracker 1 is 90cm, the tracking sources are mixed at a ratio of 10% of tracker 2 and 90% of tracker 1.
- Point C is in the middle of the overlapping area. Therefore the mixing ratio is 50/50.

From these observations, the following algorithm taking the position of two tracking devices (p_1 and p_2) and returning the ratio at which they should be mixed is derived:

- Take the point in the middle between p_1 and p_2 as the current position.
- If the current position is within the convex hull of only one tracker, the ratio of this tracker is 100%.
- If the current position is within the convex hull of both trackers,
  - compute its distances (d_1 and d_2) to the boundary of both convex hulls.
  - return $\frac{d_1}{d_1+d_2}$ as the ratio for tracker 1.

Note that this algorithm does not consider the moving direction, but determines the mixing ratio only on the basis of the current position and the convex hulls of both trackers.

When the object to be tracked now moves from point A to D, a transition as depicted in figure 7.2 is obtained. Within the overlapping area, the ratio of the one tracker continuously increases from 0% to 100% while the impact of the other tracker decreases accordingly.

As mentioned above, this transition strategy is capable of online learning. However, a training phase has to be conducted before the system can be used effectively since the convex hulls have to be created. Nevertheless, this can be regarded as online learning because the system keeps on learning during its application and therefore is able to improve its output permanently - with certain restrictions which will be explained in section 8.1.1.

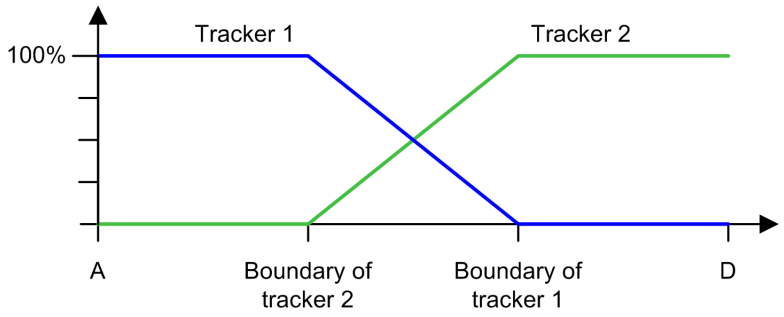


Figure 7.2: Tracking ratios of the convex hull strategy

7.1.3 ConvexHull class

The computations concerning convex hulls are encapsulated within the `ConvexHull` class, which will be described in this section.

A convex hull is represented by a list of facets each consisting of three edges and a normal vector determining its orientation and pointing towards the outside of the hull. In turn, an edge is given by two vertices with their respective x-, y- and z-coordinates. To be able to navigate through the data structure efficiently, the facets and edges are linked bidirectionally, i.e. a facet knows its edges and an edge knows the facets it belongs to.

The concept of 'visibility' plays an essential role in the following algorithms and therefore shall be clarified beforehand. A facet is regarded as visible from a given point if a person being positioned at this point is able to see that side of the facet which is oriented towards the outside of the hull. This can easily be checked by computing the dot product of the facet's normal vector and the vector from the given point to an arbitrary point on the facet. If this dot product is negative, the facet is visible.

Since for the desired application the convex hull should be constructed incrementally, the `ConvexHull` class has to provide a method `addPoint` to extend the hull if necessary. This method's implementation is based on the 'incremental algorithm' described in [17]. After the first four points, which are connected manually to a convex hull (i.e. a tetrahedron), the following algorithm is applied to all other points inserted afterwards:

1. Determine which facets of the convex hull are visible from the new point.
2. If no facets are visible, the point is inside the convex hull and no further modifications are necessary.
3. Otherwise, determine which edges separate a visible from an invisible facet. They have to be connected with the new point, i.e. new edges and facets have to be created.
4. Finally, all visible facets from (1) and the edges and vertices belonging to them are removed - except those which are needed for the edges and facets created in (3).

In order to determine whether a given point is inside the convex hull, the method `isInside` can be used. It iterates through all the facets and checks whether they are visible from the given point. If there is no visible facet, the point is inside, otherwise it lies outside. Finally, the method `getDistanceToBoundary` provides the possibility to calculate the shortest distance from a given point to the boundary of the convex hull. This is done by calculating the distances to all facets by means of simple geometry and returning the minimum.

7.1.4 Visualization

The visualization of this strategy is pretty obvious: the convex hulls for both trackers have to be rendered in different colours. This is done by means of an `IndexedFaceSet` specifying which points (i.e. vertices) belong to which facet. Thus the scene description for a convex hull would look like the following:

```
Separator
{
   Material
   {
      diffuseColor 1 0 0
   }
   ShapeHints
   {
      vertexOrdering COUNTERCLOCKWISE shapeType UNKNOWN_SHAPE_TYPE
   }
   Coordinate3
   {
      point [ 2 4 1, 1 2 6, 5 7 2, 4 8 7, 9 5 3, ... ]
   }
   IndexedFaceSet
   {
      coordIndex [ 1, 2, 3, -1, 2, 4, 5, -1, ... ]
   }
}
```

In this arbitrary example, two facets are created, one with the vertices (2,4,1), (1,2,6) and (5,7,2), the other with the vertices (1,2,6), (4,8,7) and (9,5,3). For further details on the Open Inventor format, please consult [18].

7.2 Neural network

This approach does not deal directly with the working areas of both trackers. Instead, it tries to calculate a single decision boundary so that the points of tracker 1 are on its one

side, those of tracker 2 are on its other side and as few points as possible are not classified correctly (note that there always will be misclassified points if the tracking areas overlap). As illustrated in figure 7.3, this decision boundary (not to be mixed up with the tracking boundary of the convex hull strategy) crosses the overlapping tracking area approximately in the middle. Therefore, the tracking sources should be mixed at an equal ratio on this boundary. Furthermore, if the current position is on the side of tracker 1, its ratio should increase the larger the distance to the boundary gets.

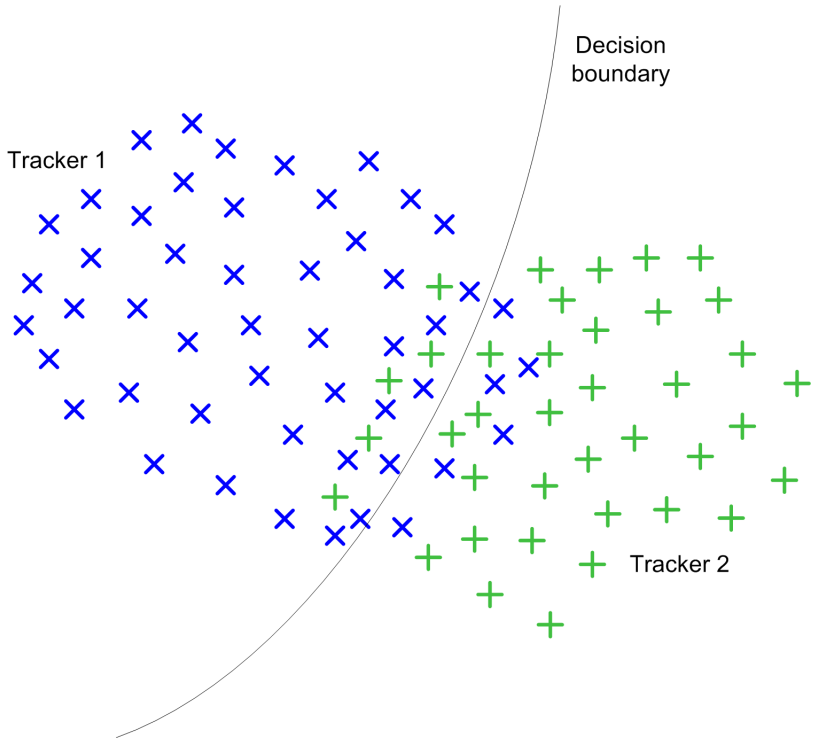


Figure 7.3: Decision boundary generated by a neural network

Since we do not know beforehand if the decision boundary is a plane, a parabolid or maybe something completely different, a universal classificator is needed for this task. As neural networks have this property, a multi-layer feedforward net was chosen to prove its promising, but rather theoretical abilities within this practical application. An introduction to neural nets can be found in [19]. For this project, a network was used having three inputs for the x-, y- and z-coordinate, one hidden layer with three neurons and one output to indicate whether the respective point is on the side of tracker 1 (output > 0) or tracker 2 (output < 0).

7.2.1 Learning phase

During the learning phase, all the points which are received from the tracking sources are stored as training data (input: x, y, z; output: +1 for tracker 1, -1 for tracker 2)

within the neural net class. As soon as 1000 points of each tracker have been collected, the network is trained using the standard backpropagation algorithm. During training, the decision boundary generated by the net is incrementally refined, so that the error (i.e. the number of misclassified points) becomes as small as possible. Training is stopped if the error remains fairly stable, i.e. it decreases less than 0.001% per iteration. Since usually a few thousand iterations are necessary in order to achieve this, online learning is - at least with this implementation - not possible.

7.2.2 Application phase

During the application phase, the service continuously computes the approximate time to reach the decision boundary (i.e. the area where the neural net output equals 0) based on the current position, moving direction and speed. For this purpose, the following numerical and iterative algorithm is used:

1. Starting from the current position, go a certain distance d further along the moving direction.
2. If we are still on the same side of the decision boundary, continue with (1).
3. If we are on the other side of the decision boundary, change the direction, halve the distance d and continue with (1).
4. Stop as soon as d has dropped below a threshold value.

This has to be done simultaneously both along the current moving direction and against it since we do not know beforehand if we are approaching or departing the decision boundary. The algorithm approximately delivers the point at which the decision boundary will be crossed. In combination with the current moving speed, the remaining time t to the boundary can be computed, which in turn is used to determine the ratio at which the trackers are to be mixed (positive t denote an approach, negative t a departure from the decision boundary):

- If $|t|$ is more than a second, only the position of the tracker is considered on whose side we currently are.
- If $|t|$ is less than a second, the trackers are mixed according to figure 7.4 (for example, the ratio would be 75/25% for $t = 0.5s$ and 50/50% for $t = 0s$).

So far, the algorithm depends on the ability to compute the actual moving direction and speed. The case of the object not moving at all has to be addressed separately. The solution for this is to return only the position of this tracker on whose side of the boundary the object is.

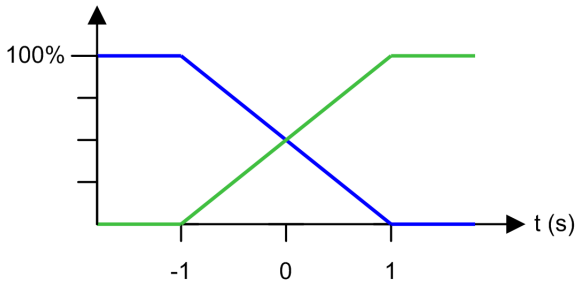


Figure 7.4: Tracking ratios of the neural net strategy

7.2.3 NeuralNet class

The neural net-specific tasks are encapsulated by the `NeuralNet` class. It implements a general feedforward network with one hidden layer and an arbitrary number of input, output and hidden neurons, which can be trained using the standard backpropagation algorithm. This general approach allows to change the number of hidden neurons easily and thus to influence the shape of the decision boundary (for example, a network with only one hidden neuron would generate a plane). Minor changes of the sourcecode would make it even possible to specify this via the XML service description. For further information on the background of neural nets and the algorithms applied here, please consult [19].

7.2.4 Visualization

To visualize this transition strategy, the decision boundary, i.e. the surface where the net output equals zero has to be rendered. Unfortunately, there is in general no closed solution to compute the points of this surface. As a simple workaround, the points of a grid where the decision boundary is supposed to be are examined and are added to the scene if the respective network output approximately equals zero. With this method, one can achieve pretty good results if enough points are considered, however at the cost of a horrendous computing time which completely disqualifies this approach for realtime applications. There are certainly better solutions, but they are beyond the scope of this project.

Chapter 8

Results and future work

This chapter depicts the results of the SEP, i.e. the specific properties of both transition strategies being observed during their practical application. Emanating from that, some hints for future work are mentioned in the last section.

8.1 Results

Both the convex hull and neural net approach are supposed to lead to a smooth transition, which becomes obvious by comparing the figures 7.2 and 7.4. However, each strategy has its specific strengths and weaknesses.

8.1.1 Convex hull

The convex hull transition strategy has proven to work well as long as the position data of the tracking devices was reliable. Figure 8.1 shows a view from above onto a scene rendered by the `Viewer` service during the application of the system in the AR lab. It depicts the four ART cameras and the room's door (black) as well as the convex hulls of the ART tracker (blue) and the InterSense tracker (green). Both convex hulls reflect the actual working areas of the respective tracking devices.

During the application of the `MultiTracker` service, the ratio of a tracker within the overlapping area continuously decreases from 100% to 0% (resp. increases from 0% to 100%) while moving from one tracking area to the other and therefore behaves as expected. This is illustrated in figure 8.2 which shows a view from above onto the measured positions of the InterSense (green) and the ART tracker (blue) and the output of the `MultiTracker` service (red) while moving an object from the left to the right. Note that the two tracking devices actually deliver different coordinates in spite of tracking the same object and being calibrated as described in section 5.2. However, the `MultiTracker` service succeeds in filtering the data so that a smooth transition is obtained.

If, however, an erroneous position of a tracker is inserted into the convex hull, this hull possibly does not reflect the actual working area anymore and then obviously a smooth

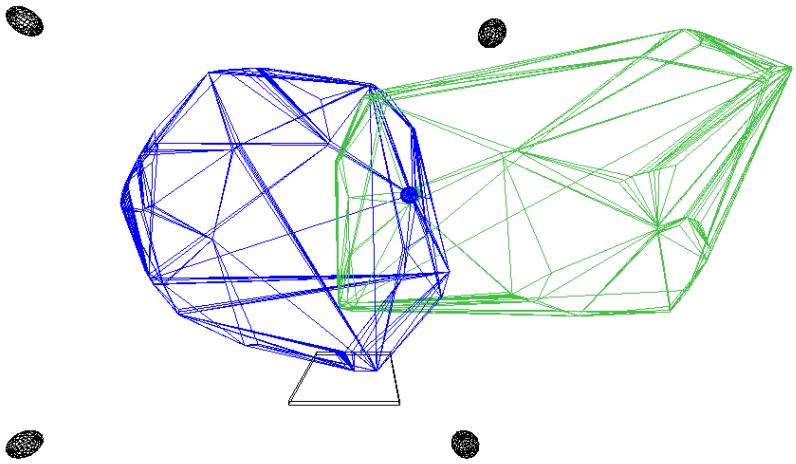


Figure 8.1: Convex hulls of the ART and InterSense tracker

transition cannot be ensured. As mentioned in section 2.1.2, especially the IS-600 turned out to deliver such outliers near its tracking boundary. Note that such a single outlier can corrupt the convex hull (see figure 8.3) and with it a successful transition. Therefore, it would make sense to implement some kind of outlier detection in future releases or - even more important - to prevent the trackers from delivering error-prone position data.

An advantage of this transition strategy is its ability to provide online learning since the algorithms applied are efficient enough even for realtime applications. Therefore, the convex hulls are continuously updated even during the application of the system. However, this approach increases the probability that the convex hulls will be corrupted due to the above reasons. Therefore, it might be better to conduct a separate training phase and let the convex hulls unchanged during the application phase (note that this is not supported in the current implementation).

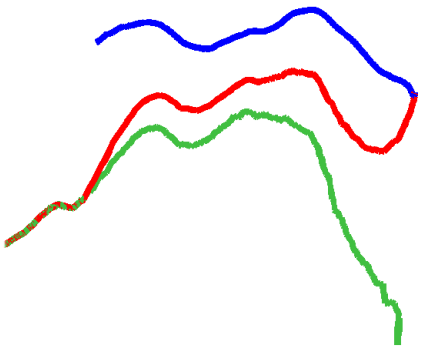


Figure 8.2: Output of the `MultiTracker` service (convex hull strategy)

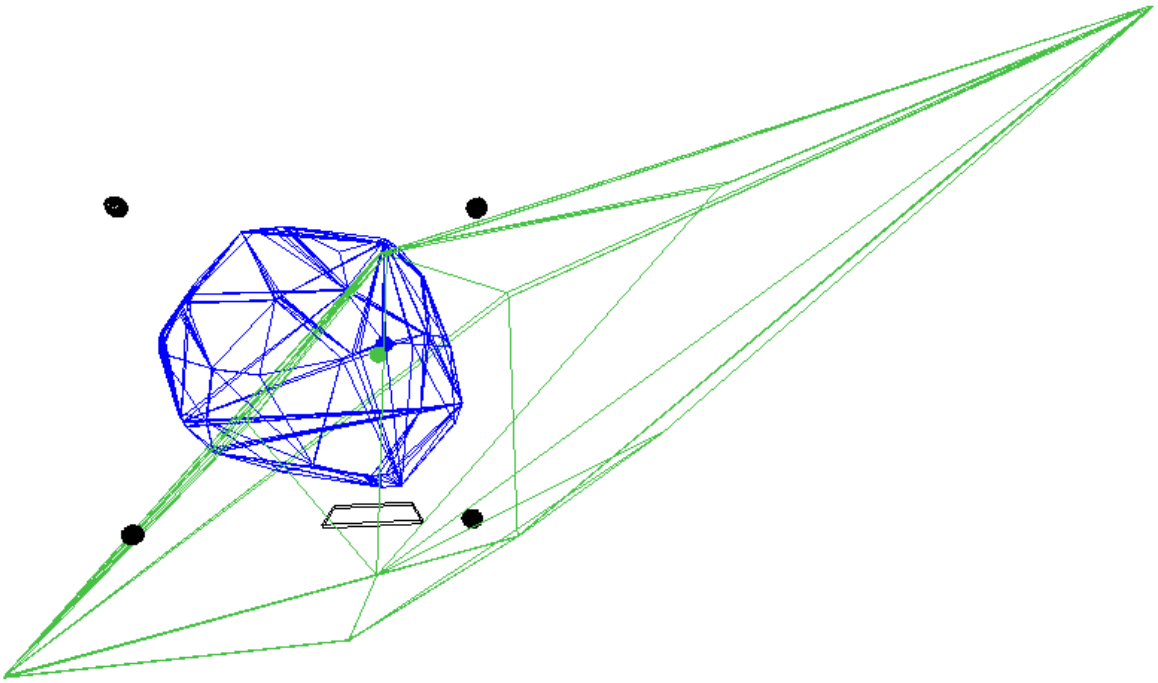


Figure 8.3: Corrupted convex hull of the InterSense tracker

8.1.2 Neural network

The neural net approach is much less sensitive to erroneous positions than the convex hull strategy since an outlier just slightly displaces the decision boundary and therefore has a negligible effect if hundreds or thousands of correct points are available. Therefore, this strategy actually computes a decision boundary which is placed and oriented as expected (see figure 8.4).

However, the transition often is not as smooth as desired although at first glance it resembles the convex hull transition (compare figures 7.2 and 7.4). The reason for that is that the threshold of one second (figure 7.4) was chosen somehow arbitrarily and has turned out not to be appropriate for all tracking setups since one second before reaching the decision boundary not both trackers are guaranteed to be available. If, for instance, the overlapping area is very small and begins only 0.5 seconds before the boundary, there would be a discontinuity (i.e. a jump from 100% to 75%) at the point where suddenly both trackers are available. This is illustrated in figure 8.5. Of course, the threshold could be decreased, but then the actual transition would take place in a very small region and therefore would not be regarded as smooth by the user. The ideal solution would be to choose this threshold dynamically so that it always coincides with the beginning of the overlapping area.

As expected, the learning phase cannot be conducted online, since a few thousand

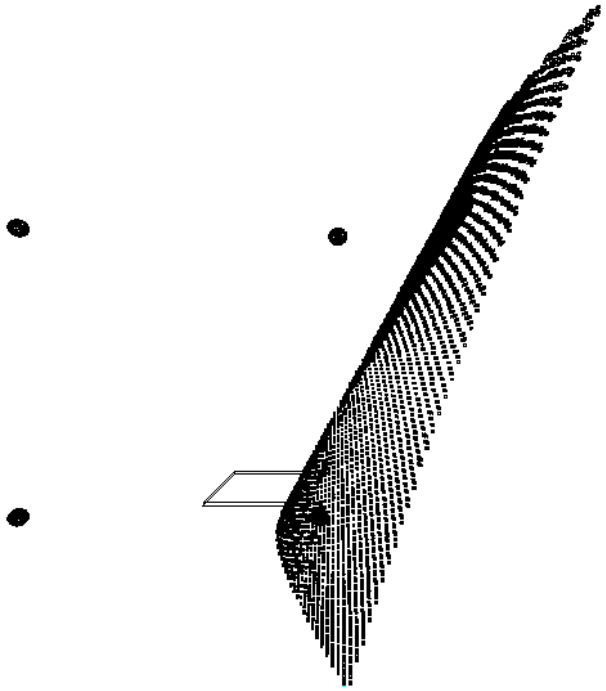


Figure 8.4: Decision boundary generated by the neural net

iterations with a relative large training set (2000 points) have to be done. However, once the network is trained, this transition strategy has proven to be sufficiently fast to be used in realtime applications.

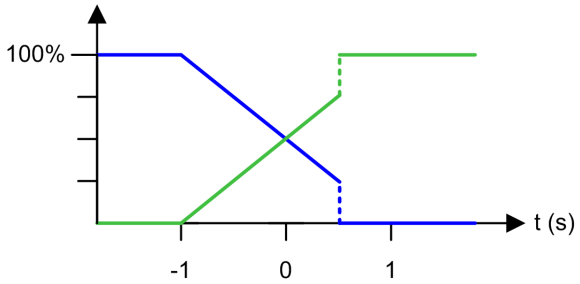


Figure 8.5: Discontinuity of the neural net strategy

8.1.3 Conclusion

In spite of the deficiencies concerning its outlier sensitivity, the convex hull strategy seems to be more suitable for the problem. The neural net approach should be seen rather as a

'proof of concept' and has produced interesting and promising results, but is in its current implementation not really applicable within real systems (of course it is much better than just switching from one tracker to the other at a predefined position).

8.2 Future work

Both transition strategies presented in this SEP have their specific disadvantages (compare section 8.1), which should be addressed during the future development. Apart from further refinement and fine-tuning of the strategies, it might be interesting to combine both approaches in order to compensate the weaknesses of the convex hull with the strengths of the neural net and vice versa.

In the chapters above, there were already given some hints for possible improvements (e.g. neural network visualization). Additionally, it should be able to make the training results (e.g. the weights of the neural net or the facets of the convex hulls) persistent, so that the learning phase does not always have to be repeated after the service startup. Furthermore, a graphical user interface (GUI), e.g. for surveying or starting and stopping the training, could help both the developer and the user to understand the processes during learning and application phase more easily.

Bibliography

[1] Suya You and Ulrich Neumann. Fusion of Vision and Gyro Tracking for Robust Augmented Reality Registration. In *IEEE Conference on Virtual Reality*, pages 71–78, 2001.

[2] Andrei State, Gentaro Hirota, David T. Chen, William F. Garrett, and Mark A. Livingston. Superior Augmented Reality Registration by Integrating Landmark Tracking and Magnetic Tracking. *Computer Graphics*, 30(Annual Conference Series):429–438, 1996.

[3] Stelian Persa and Pieter Jonker. Hybrid Tracking System for Outdoor Augmented Reality. In R.L. Lagendijk, R. Heusdens, W.A. Serdijn, and H. Sips, editors, *2nd Int. Symposium on Mobile Multimedia Systems and Applications (MMSA)*, pages 41–47, 2000.

[4] Martin Bauer. Design and Implementation of a Module for the Dynamic Combination of Different Position Trackers, 2001. Master's Thesis, TU München.

[5] Homepage of A.R.T. GmbH.
`www.ar-tracking.de`.

[6] Bernhard Zaun. Calibration of Virtual Cameras for AR, 2003. Master's Thesis, TU München.

[7] A.R.T. GmbH. *ARTtrack1 & DTrack Manual*, 2003.

[8] Homepage of InterSense Inc.
`www.isense.com`.

[9] InterSense Inc. *User Manual for IS-600 Mark 2 Systems*, 1999.

[10] Martin Bauer, Bernd Bruegge, Gudrun Klinker, Asa MacWilliams, Thomas Reicher, Stefan Riss, Christian Sandor, and Martin Wagner. Design of a Component-Based Augmented Reality Framework. In *Proceedings of ISAR 2001*, 2001.

[11] Homepage of the DWARF project.
`www.augmentedreality.de`.

[12] Otmar Hilliges. Development of a 3D-Viewer Component for DWARF, 2003. SEP Thesis, TU München.

[13] Using Quaternions to Represent Rotations.
`http.cs.berkeley.edu/~laura/cs184/quat/quaternion.html`.

[14] Marcus Tönnis. Data Management for Augmented Reality Applications, 2003. Master's Thesis, TU München.

[15] Jörg Traub. Design of an Intra-Operative Augmented Reality Navigation Tool for Robotically Assisted Minimally Invasive Cardiovascular Surgery, 2003. Master's Thesis, TU München.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[17] Der inkrementelle Algorithmus.
`www-student.cs.uni-bonn.de/~wichmann/writings/convex_hull/node3.html`.

[18] Documentation of Coin3d.
`doc.coin3d.org`.

[19] Sven Hennauer. Neuronale Netze. Hauptseminar 'Machine Learning for Context Aware Computing', TU München, 2003.