Master Thesis

Tai Chi Chuan using wearable Sensors

# Enhancements in Accuracy, Communications and dynamic Motion Recognition

**Lars Widmer**

Dennis Majoe
Adviser

Prof. Dr. Jürg Gutknecht

Computer Systems Institute
Native Systems Group

Swiss Federal Institute of Technology Zurich
ETH

2009


To my cat Rosalie for enthusiastically throwing my pen from the desk
and looking completely innocent a second later.

# Preface

Thanks to Professor J. Gutknecht for his support, trust, interest and help.

Special thanks go out to the advisor Dennis Majoe for the great collaboration, his great trust, patience, interest and all the good ideas.

Philip Tschiemer next to his ongoing studies became a part-time employee at our group and was helping a lot.

Thank you all very much!

# Abstract

This master thesis consists of this report and a set of software. It is based on preceding theses. The project covers new algorithmic development and further software development of a motion sensor system used to support teaching of sports such as Tai Chi, Tae Kwon Do but as well activities such as golf, dance and computer gaming. The project also demonstrated the incremental augmentation of the system at exhibitions and a conference where the application was successful presented and tested by visitors.

The system sensors the angles of the persons limbs and makes this data available to the computer. The project has focused on improving the accuracy of this data as well as how the data is communicated to a number of concurrent processes often running on different machines. In this way a separate machine may be set up to recognise motion gestures for Tai Chi or Dance.

Significant software development was conducted to optimise data transfer over USB and the network. Multiple processes could then exchange data via a server application. A large variety of applications could be combined to define a working whole system, set up in many different ways. For example Oberon AOS based machines, Matlab XP machines and Linux machines all work together to perform a gesture recognition process.

Apart from the optimization algorithms for the sensor angle detection, the work has included sensor data feature extraction and subsequent training and classification of data using Hidden Markov Model methods. Using trained HMM classifiers, token can be published back to the server to finally achieve a motion recognition based user human computer interface. In the end an application can use the classification for control or generating feedback as is demonstrated in the project presentation. Instructions for installation and usage are part of this report. The system is built up in a modular fashion and can hence be extended and reused.

# Zusammenfassung

Die Masterarbeit beinhaltet nebst dem vorliegenden Bericht, als Hauptteil verschiedene Computer Programme. Die Arbeiten basieren auf vorangegangenen Semesterarbeiten

Verschiedene Prozesse tauschen Daten über einen Server aus. Mit einer Vielzahl von neu entwickelten Hilfsprogrammen kann das System beliebig erweitert und auf viele Arten eingesetzt werden.

Daten werden von den neuen Sensoren ausgelesen und auf dem Netzwerk verfügbar gemacht. Anwendungen wie Feature Extraktion, hidden Markov Modelle und eine Nachbearbeitung wandeln den Strom von Sensordaten schrittweise um in eine Abfolge von erkannten Bewegungen. Eine Anwendung kann die erkannten Gesten als Eingabegrössen verwenden um entspechend zu reagieren.

Die erreichte Funktionalität kann in Zukunft verwendet werden um Gesten aus Tai Chi, Karate oder aber auch Golf oder Tanzen auf dem Computer zu erkennen und zu trainieren. Das Projekt beinhaltet nebst der Entwicklung der Software auch deren zwischenzeitige Präsentation an Ausstellungen und einer Konferenz.

Das System verwendet die Betriebssysteme Windows XP, Oberon und Linux. Angaben zur Installation und Verwendung der Programme sind Teil dieses Berichts. Die Software ist modular aufgebaut und kann dadurch einfach angepasst und erweitert werden.

# Contents

# Chapter 1

# Introduction

The report tells about the steps taken in using the ten sensor system for gesture recognition. The accuracy of the basic computations have been improved and new ways of data handling were introduced.

First there is a small excursus at tai chi 1.1 and the idea why to support teaching tai chi on a computer 1.2. In chapter 2 we talk about the goals of the project and reasons for the chosen solutions. The two chapters Achievements 3 and Users Manual 4 tell about what the abilities of the outcome are and how to use it. Finally there is a lot of text which is very important for everybody who is aimed to do a project related to the sensor system at hand. All this texts have been put into Appendix A. In Appendix B you find the generated source documentation.

## 1.1 Tai Chi

Tai Chi is an exercise form that has many benefits for health and well being. Gentle exercises are applied for inner calmness and harmony. Tai Chi originates from ancient China. It has spread allover the world because it positively affects the health of everybody. It's the art of balancing each of body, energy, spirit and emotions. Benefits can be healthiness, a focused mind and spiritual advancement.

Practically Tai Chi consists of smooth and flowing movements (figure 1.1). The movements are carried out particularly slow. Hence hardening in muscles and mind can dissolve. Due to this energetic channels open up and can be refilled.

**Figure 1.1:** Tai chi training

**Figure 1.2:** Some tai chi poses

Tai Chi is also called as mediation in movement and as the art of body and spirit. In order to get the maximum benefit out of Tai Chi one needs to perform the exercises correctly and to understand the way "Qi", or life energy, should be moved in the body using both physical and mental control. Figure 1.2 shows five different tai chi poses.

## 1.2 Motivation

On figure 1.3 we see a mini-golf teacher correcting the pose of a student. The same happens for tai chi on the picture next to it. So getting in the right pose and doing the exercises correctly is mandatory for many sports.

With gesture recognition for computer aided training it's not our general aim to replace sports teachers. Rather we would like to help them and add an extra dimension to sports training and health care. Especially for tai chi there aren't much teachers available. So in order to give people the opportunity to benefit from tai chi it could be helpful to offer computer supported training.

Having the possibility to train at home could also help shy people to overcome stoppages and join a club after exploring the sport at home. Tai chi isn't the only sport to apply the system. It's possible to use it for martial arts, basket ball, golf, volley ball,

**Figure 1.3:** Teachers and their students

dancing and many more.

## 1.3 Prerequisites

To get through the report the following experience will help:

- Programming in C++
- Working with Windows
- Little experience with Unix/Linux
- Basic knowledge of tai chi or a martial art
- Having heard of learning algorithms
- Vector algebra

# Chapter 2

# Project Context

## 2.1 Goal

This project is basically aimed at computer based training of Tai Chi, Qi Gong or even Taekwondo. Cordless motion sensors have already been designed. Preceding work Ranieri & Majoe (2007) and Widmer & Majoe (2008) allows data distribution and comparison of static poses with feedback. But there are lacks in accuracy and refresh rate. As well as the two theses haven't been combined yet.

We aim to improved the refresh rate by the use of USB technology. The accuracy will be optimized up to the limits of the sensors. Our big picture is motion recognition. With a learning algorithm movements will be taught and dynamically recognized.

## 2.2 Project Description

The project covers implementation of software algorithms, sensor interfacing and multimedia applications. There will be larger projects to adapt and reuse, as well as new software will be started on the green field.

### 2.2.1 Basis

The project is based on:

- The work of Dr. Dennis Majoe, researcher and adviser of this thesis, see Majoe, D. (2003)
- Nicola Ranieri's semester thesis, see Ranieri & Majoe (2007)
- Lars Widmer's semester thesis, see Widmer & Majoe (2008)

### 2.2.2 Tasks

Basically there are the following project tasks:

1. Upgrade the connection between the master board and the PC. Currently a serial cable is used. Higher transfer speeds fail because of hardware limitations. As a consequence of this we move to USB for connecting the master to the PC. For this purpose the existing software project has to be adapted.
2. Due to having different compilers, programming languages and operating systems the project will be split up into a set of tools, which can be combined in different ways and offer great flexibility and scalability.
3. The SNServer of Ranieri & Majoe (2007) will serve as a common platform to interconnect the different processes.
4. Improve the accuracy of the Euler angles computation from the raw sensor data. The current algorithm features various states and induces strange behavior at the state transitions. Alongside we found bugs in the computation when doing unusual movements.
5. The drawing of the avatar needs some cleanup as well. Lengths of the limbs and gaps between them will be clearly defined.
6. We need to gather knowledge about learning algorithms.
7. Meaningful features need to be extracted out of the raw sensor data. Learning algorithms usually can't be used with raw data.
8. With the learning algorithm movements will be recognized.
9. The project will again be shown at exhibitions and presentations.
10. There needs to be an application to work with the recognized movements.

## 2.3 Why this thesis

One of the big pictures of our work always was to show the usefulness of the sensors. Therefore the task of improving the sensor accuracy is very important. Up to now it wasn't clear if the sensors or the algorithm induces certain errors. As well the refresh rate. It was always difficult to explain to people that the wireless transmission wasn't the bottleneck. Now by replacing the old serial cable with USB the refresh rate is improved by a factor of ten. These two tasks are quite at a low level. Therefore they can be reused for nearly every future sensor application. Higher level tasks like handling a server for distribution of data, forward kinematics and gesture classification will be reused, too.

Finally, a system without an application is just a framework. For purposes as exhibitions, demonstrations and possibly didactics we were looking for an interesting application. As a consequence of this we adapted a sophisticated game to be played by recognized gestures. A nice application could also be helpful to attract new students doing projects in our group.

## 2.4 Why multiple Processes

A big deal of the work done was to consolidate existing projects. Soon it got clear that this couldn't been achieved in a single program. First there were different compilers (MinGW g++ and VC++). Finally we're even having three different operating systems in use. To get some idea we took a look at the dance project Gutknecht et al. (2008). There are multiple processes as well. Matlab is used there to do the classification. Between the different processes sockets are used to exchange data. Due to this we decided to go in a similar direction.

## 2.5 How does an HMM work

Just in a nutshell. For detailed knowledge please refer to Daniel et al. (2008), Cappé et al. (2005), Liu & Chua (2003), Wu & Huang (1999), Brand et al. (1997) and Kahol et al. (2004). Whereas we mainly used the first two citations.

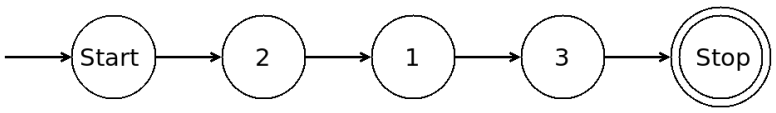



**Figure 2.1:** HMM to recognize the sequence “213”

A hidden Markov Model (HMM) is like a state machine but without the states being visible. For every gesture to recognize we need as much HMMs as there are features available. So for every combination of feature and gesture, there is a single HMM.

At first the HMMs need to be trained. Every HMM is trained on its own with the according feature of the desired gesture. While training the HMMs extracts the probabilities of the trained feature transitions over the defined time slots. In figure 2.1 we see an HMM to recognize the sequence “213”. If the input sequence is “213” the HMM gives a high output else the output is low. With a set of HMMs and any given gesture it’s now possible to produce a set of probabilities. Usually the HMM with the highest output gets declared as the winner. The gesture originally taught to the winner HMM is therefore the recognized movement.

## 2.6 Why Learning Algorithm

To be clear, as long a correct conventional algorithm can be found and tailored to a specific recognition job it will better do the job then any learning algorithm. A conventional algorithm usually has the following properties:

+ Fast
+ Reliable
+ Clear
+ Comprehensible
- Sometimes hard to find
- Needs programming skills to adapt
- Algorithm has to be rewritten for every new movement.

Whereas a learning algorithm more behaves like this:

+ Can be taught by example data
+ No programming needed to change the movement
- Fuzzy
- Often time consuming
- Not traceable

The ability of training is a nice gift. With a learning algorithm every user can train the system different. This won't be possible if we would use a conventional algorithm.

It is one of our goals to recognize movements from incomplete data. If we only use two or four sensors it's impossible to do the forward kinematics to reproduce the whole body position. Depending on the application 10 sensors might be to expensive. Learning algorithms can catch the characteristics of a particular movement from incomplete data. Not to confuse: It is possible to write a conventional algorithm for incomplete data, but it's hard.

Based on this two reasons we chose to use a learning algorithm.

## 2.7 Why HMM

There are different learning algorithms to chose from. For time dependent data like movements or speech usually HMMs are recommended. In Gutknecht et al. (2008), Kwon & Gross (2005) and other projects HMMs are used for gesture recognition. Nevertheless we thought about alternatives as well.

### 2.7.1 Backpropagation Network

With a multilayer Backpropagation Network (BPN) or a Perceptron a number of inputs can be mapped to a number of outputs. A BPN is basically a very simple form of a brain. There are connections and nodes (synapses) forming a network. At the beginning there are input nodes at the end there are output nodes. Figure 2.2 shows

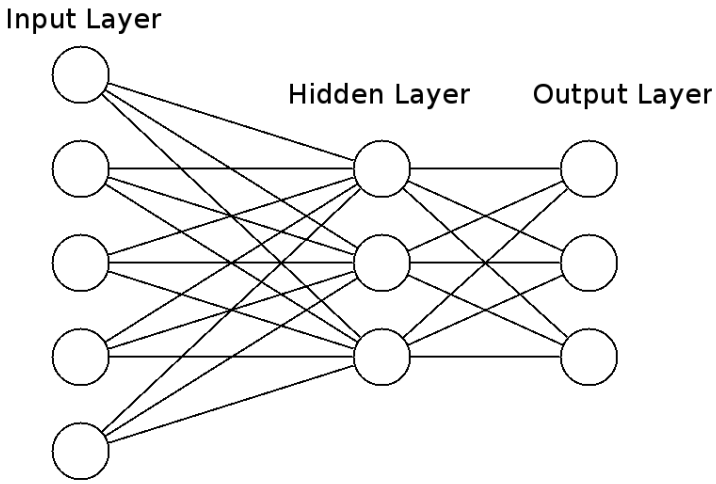



**Figure 2.2:** BPN with one hidden layer

such a BPN. An activation function computes the output of a node given the input values. There are different activation functions available. The step activation function for example implements a simple threshold. If the sum of all inputs exceeds the threshold, the node fires a signal. Every connection has its weight. The weights get adapted during teaching.

But where do we put time when using a BPN? The idea of taking time as an input doesn't fit because we don't want answers from the network every time slot. We would like to get one classification for the whole movement. So we could use a FIFO buffer for storing the outputs over time. After the movement another BPN could evaluate the buffer and produce a single output for the whole movement.

But to train a BPN a set of training data is needed which shows the correlation of input to output data (supervised learning). But we in general don't know what the first BPN has to output to the FIFO as input to the second BPN. In order to solve this we need to split up the problem and define some sort of a intermediate language. This language would be a stream of recognized static poses. The second BPN would output the gesture based on this stream in the FIFO.

### 2.7.2 Kohonen Map

The intermediate language would be defined automatically if we use an algorithm for unsupervised learning. That's why we came up with the idea of using a Kohonen Map for the first part of the recognition. Unsupervised means that we only define how many different classes (let's say n classes) we want to distinguish and prepare some input data. The map itself decides which inputs are closer to each other in feature space. The output we get is 1..n for the classification. Nevertheless a Kohonen map doesn't take time as an input. So we still need to output the data into a FIFO and use a second learning algorithm to classify from the FIFO.

### 2.7.3 Conclusion

Despite the fact we never tried, we think a Kohonen Map with an attached BPN would possibly work for gesture recognition. But still we agree that this system will be more cumbersome than a set of HMMs. That's why we agreed to use HMMs.

## 2.8 Hardware

The hardware is still the same as it had been used for the semester thesis Widmer & Majoe (2008). It's the second sensor generation. But now we use ten sensors instead of five.

A third sensor series is planned. With the current series there was a problem with the chip antennas. Therefore small wires are currently used as replacement. In the third series chip antennas will offer a much more reliable transmission over even longer distances.

# Chapter 3

# Achievements

## 3.1 Milestones

1. Use Oberon gateway to enhance throughput
2. Augment system to 10 sensors
3. Exhibition: Researchers Night
4. Improve accuracy of the euler angles calculation, redo whole vector algebra
5. Sorting of the limb lengths and improvements in spheres-drawing for avatar-construction and forward kinematics
6. Reuse SNServer, compile and test under Windows and Linux; small adpation; Write C++ classes to work with the SNServer.
7. Write first set of tool applications (Receive, Send, ...).
8. USB to SNServer connector.
   With this piece of software the raw sensor data gets published to the SNServer.
9. Develop second set of tool applications (Forward, Merge, ...).
   Adapt the tools to compile under Linux and Windows.
10. Extend the game SuperTux, that it can be controlled over the SNServer.
11. FeatureExtraction: Reduce sensor data to relevant features.

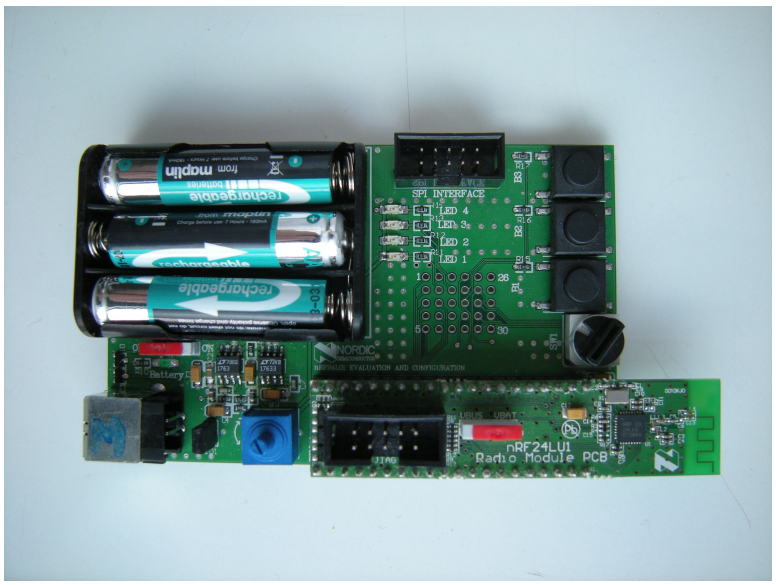



**Figure 3.1:** The new master board for the USB connection

12. Connect to SNServer from Matlab.

13. Reuse Matlab HMM application, send HMM-output to server.

14. PostProcessing: Interpret HMM output.

## 3.2 Oberon gateway

The serial connection for getting data from the sensor network had become too slow. Mainly this was because the old master board didn't support high baud-rates. With a higher baud-rate even with a serial cable connection there would a much better refresh rate. Therefore a new master board was prepared. Whereas the old board used a serial cable to connect to the PC, the new master in figure 3.1 offers a USB-connection. With USB we achieve a much better throughput. Unfortunately there wasn't any software ready to receive the USB data on the PC side. But still the PC side at least was able to receive serial data in high baud-rates. As a temporary work-around a gateway was used. An Oberon-PC was reading the USB data and forwarded it over a serial cable to the windows PC with the Avatar-Player. In Oberon it was much easier than in windows to read the USB data from the new master board.

## 3.3 Ten Sensor System

With the Oberon gateway it was now possible to use ten instead of five sensors. Due to the refactoring in Widmer & Majoe (2008) it was easy to adapt the FiveSensorSystem for the use with 10 sensor. The old FiveSensorSystem offered a refresh rate of about 1Hz. Now with even ten sensors the refresh rate was around 5Hz.

## 3.4 Researchers Night

The Researchers Night was a big public exhibition. The TenSensorSystem was shown on a booth beside the Swarm-Dance application. The TenSensorSystem is able to handle movements and static poses and offers capabilities to capture, replay, compare and provides feedback. Next to the booth there was a large stage where throughout the evening projects were presented. It was a pleasure for us to give a short talk to the audience about our system and goals. Afterwards people visited us on the booth. There were a lot of positive reactions. The images in figure 3.2 give an impression.

## 3.5 Federal Office of Culture

The federal office of culture "Bundesamt für Kultur" came to visit us. At this occasion the TenSensorSystem was shown once again. We did a short presentation with power-point, movies and live demonstration.

## 3.6 Refactoring

The current work didn't start from scratch. Figure 3.3 shows the FiveSensorSystem after the last semester thesis. But still more changes were needed.

- The basic field structure got changed. The C++ vector class replaced the old array structure. By the use of vectors a lot of memory leaks can be prevented. In the sense of:
  *"Use vector and string instead of arrays." Bjarne Stroustrup*


**Figure 3.2:** Pictures from the researchers night

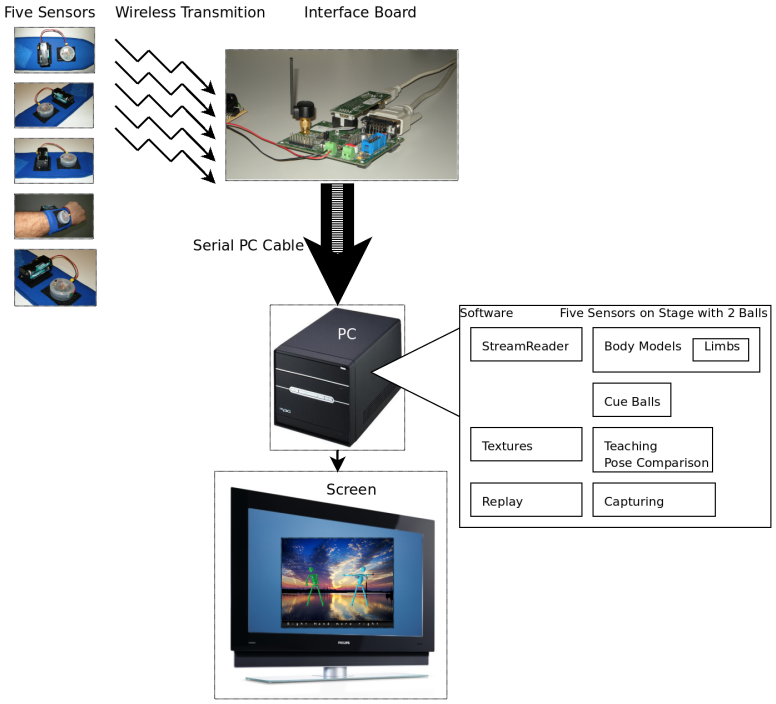



**Figure 3.3:** The system after the last semester thesis

- The class SensorData was split into StreamTransformer, SerialData and Calibration. Due to this StreamTransformer and Calibration were reused in another class. StreamTransformer does the vector computation to get the Euler angles from the raw sensor data. Calibration reads the offsets for every sensor from the files on disk.
- SerialData replaced the old class SensorData.
- Data got changed from being a data carrier to an interface for SerialData and FileData. Every source of sensor data has to implement this interface. Therefore BodyModel and Capture abstract from the physical source and just use Data.
- The handling of the limb sizes got improved as well. One has to keep in mind that a sphere is defined by three radiuses. Therefore the length of the sphere in x dimension is two times the given x radius.

The new class diagram can be found in section A.4.


**Figure 3.4:** The EuroSSC conference in Zürich

## 3.7 EuroSSC

The TenSensorSystem got demonstrated once more at the EuroSSC conference in hosted at the ETH Zürich. We submitted a demo paper and went to show the system on a booth. In figure 3.4 one can see some impressions from the EuroSSC conference. Beside showing the system we attended the conference and acquired some useful information.

## 3.8 Enhancing the Accuracy

When watching the old visualization the limbs were often a little trembling. In fact this wasn't only because of the noise in the analog sensor devices. There was a stateful algorithm used to compute the Euler angles. Because there wasn't a hysteresis at certain poses the states changed rather quickly. When moving to unusual positions some limbs flipped to a 90° wrong orientation. To overcome this problems the whole

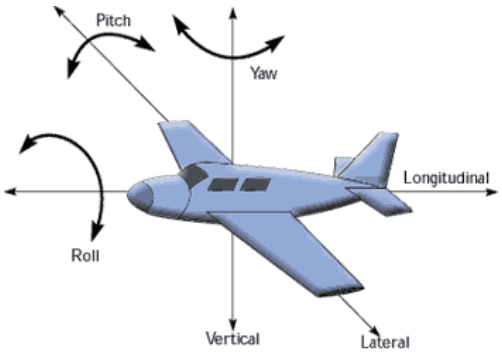



**Figure 3.5:** Pitch, Roll and Yaw on a plane

computation got rewritten into a stateless vector computation.

We will use the terms of pitch, roll and azimuth. One can think of it as an airplane.

1. Roll is applied first when aligning the plane from the outside. Roll means if the plane leans to the left or right. Leaning to the left means the left wing goes down and the right wing goes up. Roll doesn't affect the direction in which we are flying. Only or passengers tumble on the floor or the ceiling.

2. Pitch is applied second when aligning the plane from the outside. Pitch means that the plane is flying up or down. The position in which it is rolled doesn't matter. Pitch determines if we rise of fall.

3. Azimuth is applied at third when aligning the plane from the outside. Therefore azimuth is always in the plane rectangular to the earth magnetic field. Azimuth is the direction where we fly e.g. north or west. Azimuth is independent from rising, falling or rolling the plane.

Figure 3.5 shows Pitch, Roll and Yaw on a plane. Yaw and Azimuth are synonyms.

To go through the algebra.

1. The sensor offsets get subtracted from the data.

2. From the acceleration the absolute values and signs are computed. We will use this information for later reintroducing the signs of the Euler angles.

3. The orientation of the magnetometer data has to be corrected. The original sensor values use two different coordinate systems. This was a bug really hard to

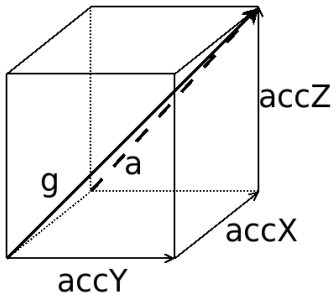



**Figure 3.6:** Acceleration vector g in the coordinate system of the plane

find! The x and y axis need to be inverted. Now the coordinate system are unified and we can interpret the sensor values as two vectors. The acceleration vector is pointing downwards and the magnetometer vector points in the direction of north.

4. Compute the length of the magnetometer and the acceleration vector. On a limited system (embedded device) this values can be replaced by constants or only be computed once.

5. Using the lengths we normalize the vectors. The sum of these two vectors should be constant, no matter how you turn and flip the sensor. This concept was used to benchmark the computation.

6. The vectors are in our coordinate system. Our coordinate system is fixed to the plane. We are sitting (let's say glued to our seat) in the airplane. Therefore the gravitation vector rotates around us, when we pitch and roll.

7. The x axis points from the tail to the nose of the plane. The y axis goes from the left wing to the right wing, when looking to the nose. The z axis goes from the floor to the ceiling.

**Pitch:**

1. For calculating the pitch we take a look at figure 3.6. It shows the gravitation vector $\vec{g}$ in the local coordinate system of the plane. Gravitation is pointing slightly upwards. This means we're flying upside down.

2. The value $\vec{accX}$ is the projection of $\vec{g}$ on the x axis. Same for $\vec{accY}$ and the y axis and $\vec{accZ}$ and the z axis.

3. The measurements of the accelerometer actually are $\vec{acc}X_X$, $\vec{acc}Y_Y$ and $\vec{acc}Z_Z$.

4. Pitch can be thought as the angle between out x axis and $\vec{g}$. But we'd like it the way that a pitch of zero means we're neither rising nor falling. That's why we subtract $90°$ from this value.

5. For calculating the pitch we build the following rectangular triangle. The hypotenuse is $\vec{g}$. One cathetus is $\vec{acc}X$. The other cathetus is the vectorial sum $\vec{acc}Y + \vec{acc}Z = \vec{a}$.
$|\vec{a}| = \sqrt{\vec{acc}Y_Y^2 + \vec{acc}Z_Z^2}$

6. Pitch is now the angle $\alpha$ between $\vec{g}$ and $\vec{a}$.

7. We use trigonometry to calculate this angle.
$\sin\alpha = \frac{\vec{acc}X_X}{|\vec{g}|}$

8. From which follows:
$\alpha = \arcsin\frac{\vec{acc}X_X}{|\vec{g}|}$

**Roll:**

1. For calculating roll we build another rectangular triangle. The hypotenuse now is $\vec{a}$. One cathetus is $\vec{acc}Z$. The other cathetus is $\vec{acc}Y$.

2. Roll is the angle between $\vec{acc}Z$ and $\vec{a}$. If there is no roll we have the following behavior: $\vec{acc}Z = \vec{a} = \vec{g}$.

3. From trigonometrie follows:
$\cos\beta = \frac{\vec{acc}Z_Z}{|\vec{a}|}$

4. From this we get:
$\beta = \arccos\frac{\vec{acc}Z_Z}{|\vec{a}|}$

5. But $\cos$ is a symmetrical function. Therefore we lose the sign when using $\arccos$.

6. Happily the sign of roll is identical to the sign of $\vec{acc}Y_Y$.

**Azimuth:**

1. For the computation of azimuth we introduce a plane $G$ through the point $(0, 0, 0)$ and orthogonal to the gravitation vector $\vec{g}$. This is exactly the place where we want to measure azimuth!
2. All we now have to do is to do a projection of the magnetometer vector $\vec{m}$ to this plane and compute the angle between this projection and a reference.
3. For the reference we take the vector $(1, 0, 0)$ and do a projection to $G$.
4. We define the plane $G$ by its normal vector $\vec{g} = (\vec{accX}_X, \vec{accY}_Y, \vec{accZ}_Z)$. $G = \vec{accX}_X \cdot x + \vec{accY}_Y \cdot y + \vec{accZ}_Z \cdot z$ whereas $(x, y, z)$ is a point in $G$
5. Now we're looking for a point $A$ in $G$ whereas $\vec{0A} = \vec{x}$ is exactly the projection of $\vec{m}$ to $G$.
6. We set $\vec{m} = \vec{0M}$. One has to draw it on a figure to understand it.
7. $\vec{x} + \vec{AM} = \vec{m}$ and $\vec{AM} || \vec{g}$.
8. But $\vec{AM}$ and $\vec{g}$ aren't of the same length. Therefore there is a stretching factor $f$.
9. We set $\vec{AM} = f \cdot \vec{g}$
10. So we get $\vec{x} = \vec{m} + f \cdot \vec{g}$
11. Because $A \in G$ it follows that $\vec{x} \cdot \vec{g} = 0$
12. All together we have $(\vec{m} + f \cdot \vec{g}) \cdot \vec{g} = 0$
13. We get $\vec{m} \cdot \vec{g} + f \cdot \vec{g} \cdot \vec{g} = 0$
14. $\vec{m} \cdot \vec{g} + f \cdot \vec{g}^2 = 0$
15. $f \cdot \vec{g}^2 = -\vec{m} \cdot \vec{g}$
16. $f = -\frac{\vec{m} \cdot \vec{g}}{\vec{g}^2}$
17. $\vec{x} = \vec{m} + f \cdot \vec{g}$
18. For the reference vector $\vec{r} = (1, 0, 0)$ we do the same projection: $l = -\frac{\vec{r} \cdot \vec{g}}{\vec{g}^2}$

19. The projection of $\vec{r}$ is $\vec{y}$.
20. $\vec{y} = \vec{r} + l \cdot \vec{g}$
21. Azimuth $\gamma$ is the angle between $\vec{x}$ and $\vec{y}$
22. $\cos\gamma = \frac{\vec{x}\cdot\vec{y}}{|\vec{x}|\cdot|\vec{y}|}$
23. $\gamma = \arccos\frac{\vec{x}\cdot\vec{y}}{|\vec{x}|\cdot|\vec{y}|}$
24. $\cos$ again, therefore we need to reconstruct the sign.
25. Therefore we construct another plane $S$.
26. The plane is defined by $\vec{g}$ and $\vec{m}$ both being in the plane.
27. We can define the plane by three points:
    $(0,0,0)$, $(\vec{g}_X, \vec{g}_Y, \vec{g}_Z)$ and $(\vec{m}_X, \vec{m}_Y, \vec{m}_Z)$
28. But we only need this plane in our imagination to determine to which side of it $\vec{r}$ points.
29. We set $\vec{t}$ as the normal vector of $S$.
30. Therefore $\vec{t}$ is orthogonal to $\vec{g}$ and $\vec{t}$ is orthogonal to $\vec{m}$.
31. This means $\vec{t} = \vec{g} \times \vec{m}$
32. We do a projection of $\vec{r}$ on $S$.
33. The same way as already we look for a factor $k$ such that $k$ again depicts the distance of $\vec{r}$ above $S$.
34. $k = -\frac{\vec{t}\cdot\vec{r}}{\vec{t}^2}$
35. Now we can use the sign of k as the sign of Azimuth.
36. done

The source code for applying the vector algebra can be found at section A.6.

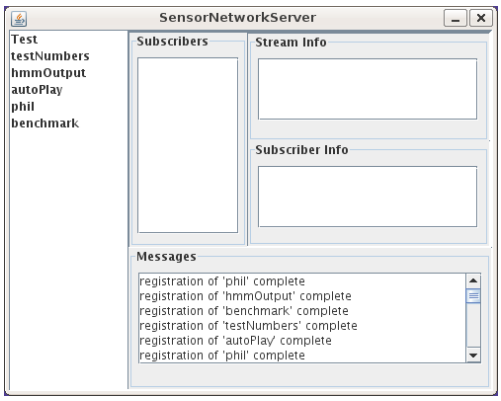



**Figure 3.7:** The SNServer Java application

## 3.9 Multiprocess solution

The final project isn't anymore running as a single process. As to be seen in section 2.4 there were reasons for splitting up to a multi process solution. We had to use different compilers and operating systems in order to achieve our goals.

The SNServer has been build by Ranieri & Majoe (2007). Figure 3.7 shows the application in use. We chose to use it because it offers a better way than just passing data via sockets from one application to another. Data producers cab publish to the server without knowing about any subscribers. Other applications subscribe to receive the data.

To interface with server a new C++ class had to be built. The server is written in Java so we had to start from scratch. The finished class `HandleServer` offers a high level interface and completely abstracts from the server protocol. `HandleServer` servers as the basis for all work following from here.

## 3.10 Server PC

At the beginning of the project we were looking for some place to store and exchange data in an easy way. Therefore a server has been set up in Winterthur. So the location fitted well. A lot of work has been done outside of the office. At the beginning FTP and SSH were used. The server runs a version of Ubuntu Linux. Unlike other servers we were forced to use a graphical subsystem. Because the SNServer wasn't

able to run without a GUI. Cablecom as an Internet provider sometimes changes the IP address of its customers. To overcome this a dynamic system has been introduced to automatically check every five minutes if the actual server address and the address stored in the DNS entry still are the same. Therefore the server always can be reached via the URL *mona.lawi.ch*. To be protected against the usual threats all the unused services as Samba and NFS have been disabled. VNC has been used for remote access and administration. Later the server allowed us to leave a certain setup continuously running. This simplified our work because there were less applications to run on our local PCs. And everyone was able to receive and publish sensor data from wherever he was. On the server our system proved to be able to run over a week without a crash.

## 3.11 USB Connector

Finally it was achieved to read the USB data directly from the master board in Windows, too. So a network connector was prepared to publish the raw sensor data to the SNServer. Because WinUSB was used, this application only runs on Windows. With the USB connector the Oberon Gateway got replaced. Now even higher speeds are possible because there isn't any more serial cable in the line from the master board to the PC.

## 3.12 Tools

Except ControlKeys and the AvatarPlayer all the tools can be compiled for Windows and Linux using two different Makefiles.

### 3.12.1 TestSender

Tool for publishing test data to the server. Generate a random stream. Every second a packet is sent. Each packet consists of a random byte value encoded into two characters.

### 3.12.2 Receiver

Tool for receiving any streams from the server. It get a stream and print it to the command line (standard-out). Via pipes on command line this tool can be used to provide any server data to other processes.

### 3.12.3 Sender

Tool for sending data from standard-in to the server. With this tool one can basically send anything using pipes on the command line. A combination of Receiver and Sender with a pipe can be used to replace the Forward tool.

### 3.12.4 Log

Tool to write a stream to disk. But not only the server data gets captured. The tool also measures and stores the delay between the packets. Therefore Log can be used as well to analyze the timing behavior of server applications.

### 3.12.5 Replay

Replay a logged stream to the server. This tool uses the same file format as Log. Due to the timing support the playback stream can be used to replace a real sensor data stream. This was very helpful for debugging.

### 3.12.6 List

Tool to display the available streams on the server.

### 3.12.7 GetLast

Demonstrating non blocking read from the UDP socket. Test Software for a nonblocking getter of the UDP-packets.

### 3.12.8 ControlKeys

Software to play SuperTux (see section 3.16) over the SNServer. For playing the game simple commands like “jump”, “left” or “right” are used. In the end these commands will be sent from the PostProcessing tool based on the HMM output. ControlKeys therefore is sort of a mockup to test if SuperTux runs nicely with the remote commands. It basically sends the pressed keys to the server in commands like “left down”. The first word tells about the command. The second word displays the key event. If the key is pressed a “down”-event is produced. If the key is released a “up”-event is produced. ControlKeys is the tool which currently can’t be compiled under MS Windows. Presumably it would be possible to compile this under Windows as well, but one needs all the SDL-Libraries installed.

### 3.12.9 Forward

Tool to duplicate a stream or forward it from one server to another. The tool takes two complete sets of server and stream information. If the user enters two times the same server but with different streams, the result is the duplication of a stream. If two different servers are used, a forwarding of the stream results. This can be used to introduce redundancy.

### 3.12.10 Merge

Tool to merge two streams into one. This tool is useful when there are two applications producing data and their output streams need to be merged into one stream. For example it could be possible to save computation time by using multiple PCs for HMM classification. Each computer would do a separate job. Finally the classification results can be merged and appear as if one computer has published it.

## 3.13 FeatureExtraction

This tool works as a subscriber and a publisher at the same time. It subscribes for raw sensor data and reduces it to relevant features. The features are published back to the

same server or another server. The tool can be adapted quite easy. Currently it's set to discard magnetometer data. The values of the accelerometers are quantized into 5 values for each dimension.

- 0 means -2 times gravitation
- 1 means -1 times gravitation
- 2 means low activity
- 3 means 1 times gravitation
- 4 means 2 times gravitation

For one sensor the feature output in rest typically looks like this "223" when the sensor is put flat on the table. This means low acceleration in the X and Y axis. The Z axis measures the usual earth gravitation. If we turn the sensor upside down we get "221".

Beside the tool for converting raw sensor data to features we use a modified version of the USB connector. Instead of raw sensor data it directly calculates the features from the data and publishes them to the server.

## 3.14 HMM

For the HMMs we currently have two solutions.

- Matlab: From Daniel Roggen & Crew from the ETH electronics laboratory we received a working Matlab demo application to train a set of HMMs and use them for classification. They use this application for the exercise sessions in lectures about wearable computing. But there were different sensors used and we had to find a way how to interface from Matlab to the SNServer.
- C: LI, Guoliang, PhD candidate at the School of Computing at the National University of Singapore provided us a working C software to run an HMM.

Currently we are using the Matlab version. The application has been adapted to the feature data from our sensors. For the server connection we used the TCP/UDP/IP

Toolbox 2.0.6 from Peter Rydesäter available at the mathworks website. Special thanks go out to Philip Tschiemer for his work!

Matlab offers quite slow processing speed and we would like to switch to the C code as soon as possible. There has already a running application been prepared to show that the library actually is working fine.

## 3.15 PostProcessing

Every HMM basically outputs a value as a measurement for the probability that the current movement is the one which was taught to this HMM. From Matlab the set of output values are again published to the SNServer. The PostProcessing application subscribes to the hmmOutput stream and evaluates the data in order to extract the desired information. In the case of SuperTux (see section 3.16) the desired data packages are command events like those published by the ControlKeys (section 3.12.8) tool. For example if the user lifts his left arm "left down" is sent to simulate a key-down event of the left cursor key.

## 3.16 Application

Especially at exhibitions nice applications are attractive for the audience. People rate a technology by its application. It's a bit a pity to stand at a booth and to know that the sensors are quite sophisticated but the application could be better. That's why we decided to use an existing game to use the HMM classification. So we started to look for an open source game. Unfortunately we didn't find a beat'em up game like Street Fighter, One must fall or Tekken. At least we found a free engine to realize fighting games. Maybe we will use this later. We further took a look at Ego-shooter games like OpenArena (figure 3.8) and Jump'n Run games like SuperTux (figure 3.9). OpenArena is based on the Quake3-engine which was made open source by id software. SuperTux is a clone of the well known Super-Mario game. But instead of Mario a penguin (called Tux) jumping and running.

Because we had children on past exhibitions we chose to use SuperTux. In a first step the source code had to be viewed if it's suitable for our purpose. The code quality is


**Figure 3.8:** OpenArena Ego Shooter


**Figure 3.9:** SuperTux Jump'n Run

good enough and the source is structured. After getting an overview it wasn't difficult to plug our HandleServer with a singleton wrapper class into the game.

With ControlKeys the game can be played without noticing a difference. Even if the SNServer runs in Winterthur and we're playing in Zürich.

Yet we compiled SuperTux only in Linux. But there is a version available for Windows and the code is prepared to be compiled in Windows as well. The game is platform independent because it uses the OpenGL graphics library.

## 3.17 Whole System

Figure 3.10 shows all the software needed from reading the sensors up to playing SuperTux. Most of the tools are converters. They are subscriber and publisher at the same time.

With tools like Forward and Merge it is possible to build up whole networks. Figure 3.11 shows an example with two servers, one converter, two publisher, three subscriber and one forwarder.

The system offers great extendibility. Therefore even larger networks like in figure 3.12 are thinkable.

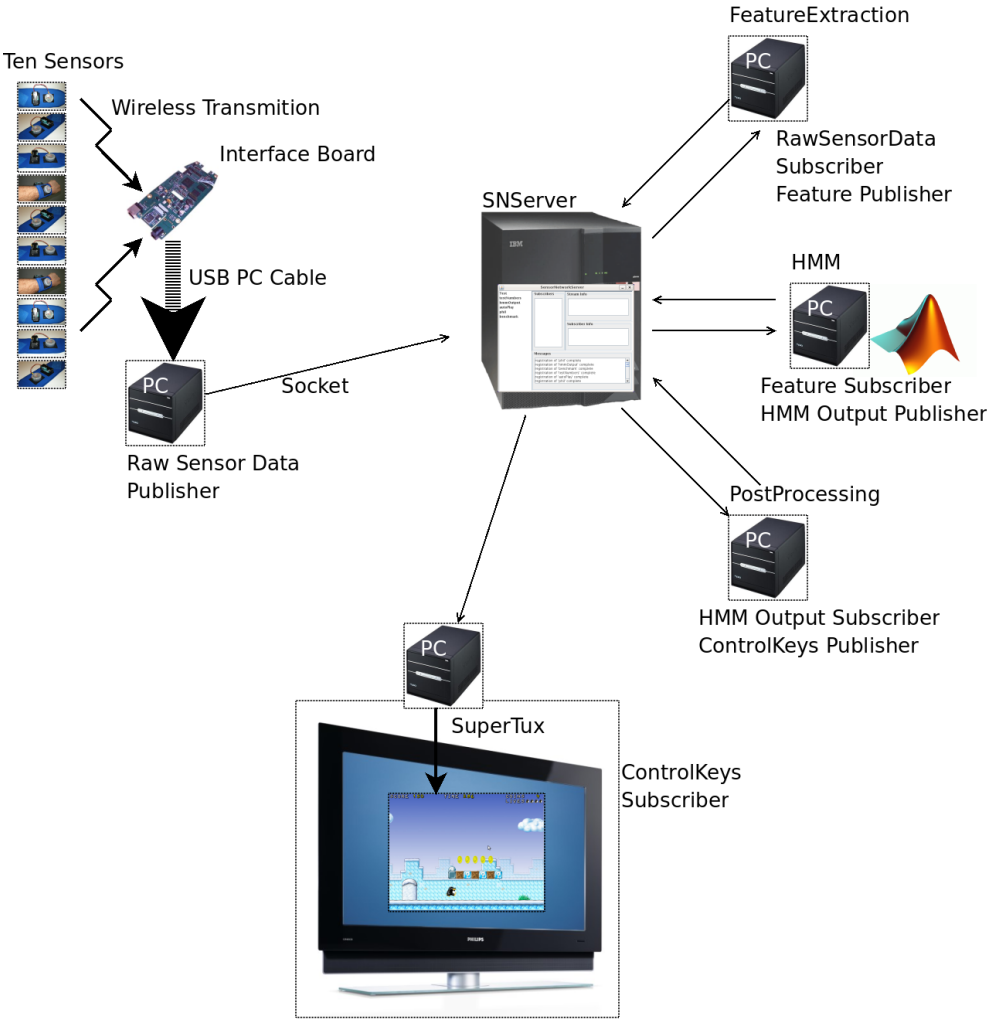



**Figure 3.10:** The whole demo system

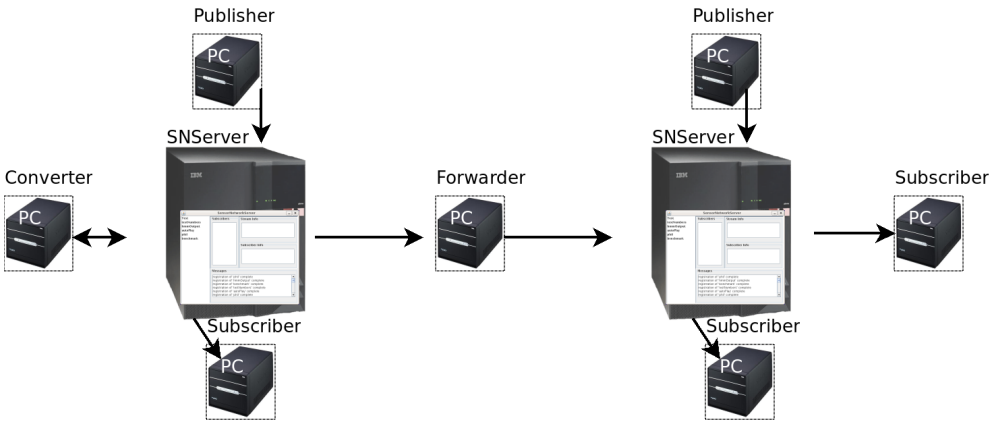



**Figure 3.11:** A small network with two servers

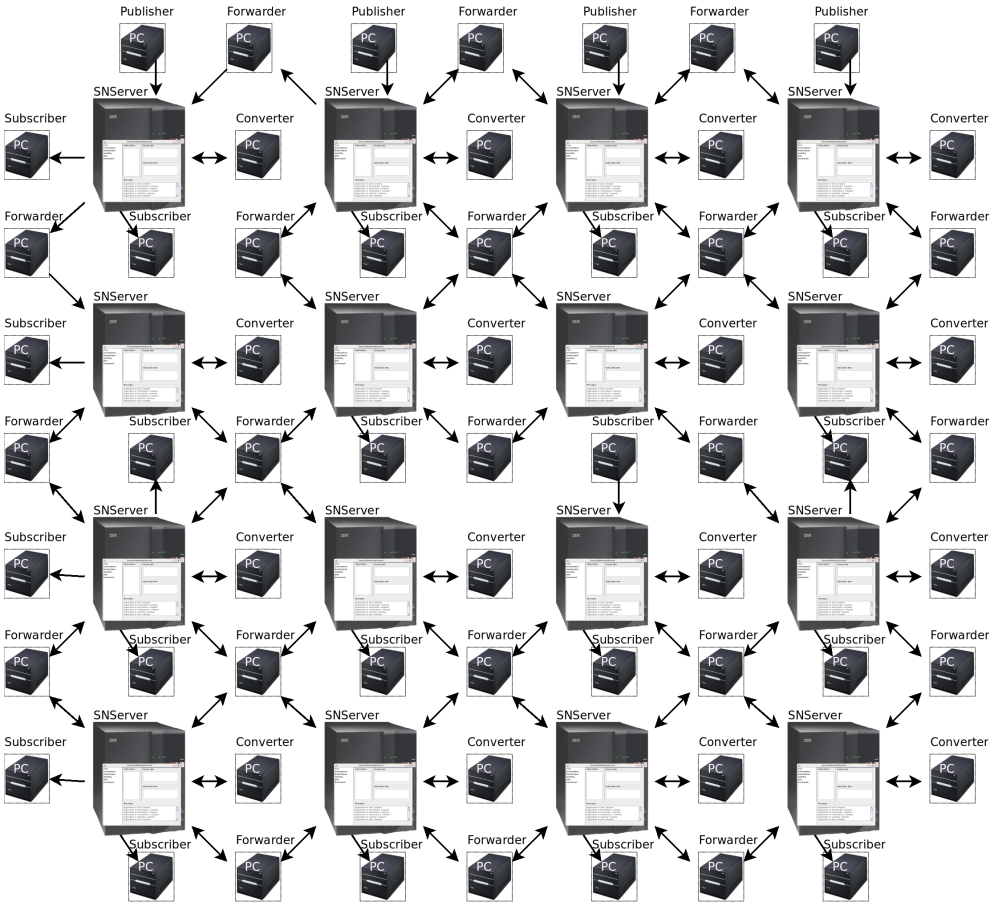



**Figure 3.12:** A larger network

# Chapter 4

# Users Manual

## 4.1 SNServer

The server can be ran in Linux as well as in Windows. For starting the server one has to enter the directories Java and SNServer using a command shell. There one has to enter *java Main.Main*. The port numbers are used in default state. After start the server is ready there is no further interaction needed.

## 4.2 Tools

The tools can be ran in a command shell, too. Needed command line parameters are listed if a tool gets started with only one parameter. One just has to try *tool -help*. When started without parameters the tools use default settings. This usually means the server is expected running at the same computer and listening on TCP port 2345. If there aren't any executables available for your OS see section A.2 for compiling the tools.

## 4.3 Matlab

The Usb2Features application has to be running in order to provide the input for the HMMs.

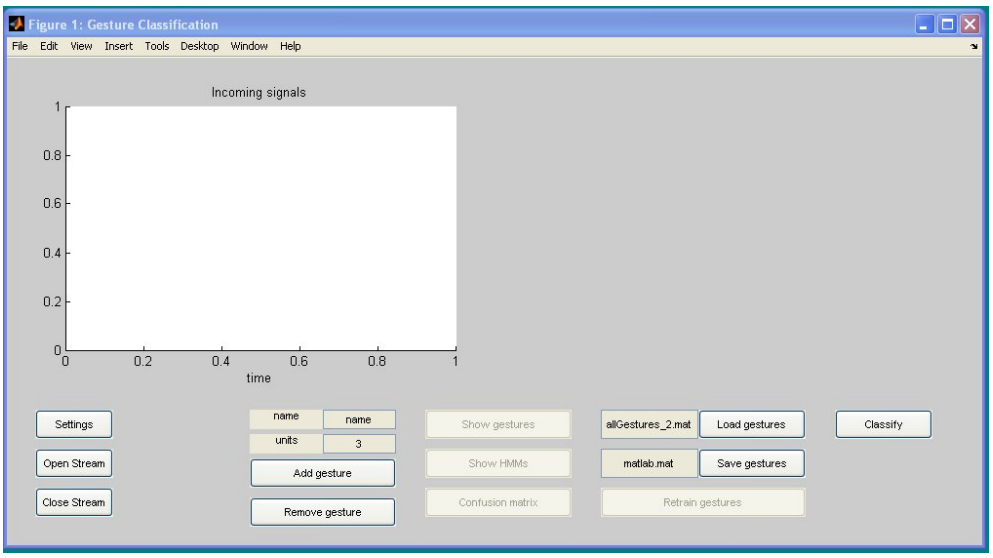



**Figure 4.1:** Screenshot from the Matlab HMM application, first window

Open Matlab and change into the folder “mod” (type cd mod). Then start the application “mygui”. You get a GUI as in figure 4.1. Now you can add gestures and train them. There is a prepared training file on disk. Just enter allGestures_2.mat as shown in figure 4.1 and click on the “Load Gestures” button. Before we can open a stream we have to activate the server connection. Therefore we click on the classify button. A window as to be seen in figure 4.2 opens up. Click on activate to register at the server. With the Resume button the application gets ready to classify. Now in the first window the stream can be opened. Automatically the HMMs should start to classify. With the Receiver tool the output can be watched. Using PostProcessing the command events for SuperTux can be generated out of the HMM output.

## 4.4 SuperTux

The game is really self explaining. Just enter the folder SuperTux/src and start the executable ./supertux.

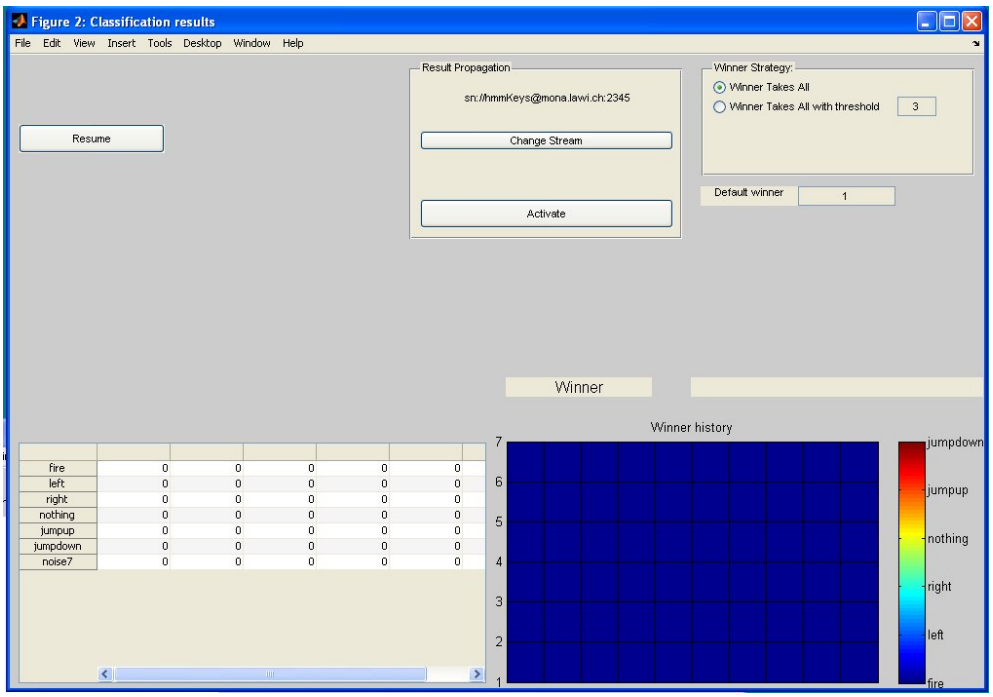



**Figure 4.2:** Screenshot from the Matlab HMM application, classification window

## 4.5 AvatarPlayer

If you just want to run the software, enter the project directory on the project CD and run the executable there. For the AvatarPlayer the following has to be running as well.

- SNServer
- Usb2Network
- Interface board “Master”
- Interface board “Beacon”
- Sensors

The port numbers and server addresses need to be configured correctly for every application. Now the screen with the two avatars should appear.


**Figure 4.3:** Screenshot from the ten sensor system software

### 4.5.1 Training

The AvatarPlayer software still has the same functionalities as described in Widmer & Majoe (2008). Just with the difference that we now use ten instead of five sensors. As a consequence of this it's now possible to model the whole body on the computer instead of only the over body.

When moving around you will recognize the texture of your avatar to change. The better you manage to copy the pose of the teacher on the left side, the more light blue your body gets. The more wrong a limb is positioned, the more red it appears. The system also chooses your worst limb and advises you in the text line at the bottom of the screen how to improve your position.

Figure 4.3 shows a screenshot from the running application.

Next to the pose comparison there is a virtual ball on the screen. The ball follows the hands of the user and can be used to play with. Using only one ball we implemented throwing it to the teacher and getting it back from him. This could serve as a basis for a game in future.

### 4.5.2 Keys Used

When presenting the software to an audience too many buttons on the screen would just distract the audience. Currently there aren't any buttons implemented. Instead the keyboard was used to control the software.

- B: Switch the virtual cue ball on.
- N: Switch the virtual cue ball off and unmerge teacher and student.
- P: Start playing teacher movement.
- O: Stop playing the teachers movement and stay at the current frame.
- I: Show next teacher pose. This feature is used to step through teaching data in a frame by frame manner. If playing of the teacher movement was activated before, it is stopped by this key.
- M: Merge the two avatars.
- S: Start capturing continuous data from the students movements.
- A: Stop capturing data from the students movements.
- D: Capture a single data frame from the students pose.
- 1: Load capture file capture01.txt
- 2: Load capture file capture02.txt
- 3...8: Load capture file capture03...8.txt
- 9: Load capture file capture09.txt
- Esc: Exit software.

## 4.6 Sensors

Sensors and interface board indicate their function using green LEDs. The sensors LEDs should be flashing. Every flash means a successful transmitted data package. So

**Figure 4.4:** Close view to a sensor

if the flashing gets slow this means you're losing too much data. A reason could be too long distance between the sensor and the interface board (receiver). If the flashing of the sensor boards LEDs gets weak this means the batteries are empty and should be replaced.

The sensors have to be attached carefully to the body of the user. Whereas one has to keep in mind that they measure angles and not positions. When you stand straight and have your palms beside your legs, all the sensors have to point forward. The battery packs have to point away from your body. Alternatively you can check that the tantal electrolytic capacitor on the print is pointing downwards.

The battery packs as they can be seen on figure 4.4 are only used to save coin cell batteries. Actually it's possible to have the batteries within the same casing as the sensor are right now.

The Ten Sensors System uses these sensors attached to the following limbs:

- A6: Left arm
- A7: Left upper arm
- A8: Chest
- A9: Right upper arm
- AA: Right arm
- A1: Left leg

- A2: Left upper leg
- A3: Waist
- A4: Right upper leg
- A5: Right leg

This is how the sensors are labeled. Actually the corresponding node numbers are different. They are from B1 to BA. On the labels there is just an A instead of a B as the first letter.

## 4.7 Development

See section A.2 for changing and compiling the parts of the software.

## 4.8 Software security

Of course the software has been developed in a careful manner. But the software is still experimental and clearly in beta status.

# Chapter 5

# Testing

## 5.1 White Box Testing

There aren't yet any test classes. For the graphics output there don't need to be any test classes. For the calculation of the Euler angles it would have been a good idea to define test cases and check them in an automated way.

## 5.2 Black Box Testing

The following tasks have been successfully tested:

- Run the SNServer on Linux and Windows.
- Tools
  - TestSender tool publishing test data to a server.
  - Receiver tool printing a stream from a server to the command line.
  - Sender tool sending data from standard-in to a server.
  - Log tool writing a stream to disk.
  - Replay replaying a logged stream.
  - List displaying the available streams on a server.
  - GetLast showing a non blocking read from the UDP socket.

  - ControlKeys sending simple command words for pressed keys to a server.
  - Forward tool duplicating a stream on a server.
  - Forward tool forwarding a stream from one server to another.
  - Merge tool merging two streams into one and sending back to a server.

- UsbToNetwork Connector sending the data from one to ten sensors to a server at a rate of 10Hz.
- FeatureExtraction extracting six features from two sensors and sending them to a server at a rate of 30Hz.
- HMM
  - Teaching multiple movements by one or three examples for each movement. The examples were recorded using the Log tool.
  - Analyzing the movements out of a stream from the server using a set of 42 HMMs.
  - Sending the HMM output back to the server.
- PostProcessing analyzing the HMM output and generating commands to play SuperTux.
- SuperTux
  - Run the application playing it the usual way
  - Run the application using SNServer and ControlKeys to play it from another PC. It's playable without a problem even if the SNServer runs 20km away from the two PCs running ControlKeys and SuperTux.
- AvatarPlayer
  - Display avatars
  - Play teacher
  - Stop teacher
  - Step through teacher movement frame by frame
  - Capture a student movement

- Capture some student poses
- Take new captured files as teacher poses and movements
- The more wrong a students limb gets positioned the more red it appears
- There is a meaningful advice at the bottom of the screen what the student should correct next
- Switch the task to another running software
- Terminate the software

The following tasks have yielded problems which will get fixed ASAP:

- Return from switching the task to another running software. The screen looks scrambled and colors appear wrong. Presumably there's something wrong with reinitializing DirectX.

# Chapter 6

# Summary and Contributions

It was once again an interesting and enjoyable project. The range of sensors, data protocols, data structures, algorithms and computer graphics offers a nice variety of interesting topics. Having public presentations, exhibitions and conferences in prospect is a source of motivation. Gladly the presentations went well and lead to positive feedback. Hopefully the ongoing projects will push much further and will make the project even more interesting. We're looking forward to visit conferences and present our work.

## 6.1 Future Work

- Discharge the todo-list (see A.3)
- Check for bottle necks in computation and network throughput.
- Prepare a application for teaching tai chi using the new gesture recognition.
- Work for the NanoTerra project.
- Prepare a beat'em up game for presentation of the sensor abilities.
- Use sensors to simulate a theremin musical instrument.

## 6.2 Other Projects

Four out of five projects are basically the same as already mentioned in Widmer & Majoe (2008). The Nanoterra project is new and already in development. Planet Penguin Racer was replaced by SuperTux as mentioned in section 3.16. SwarmDance has been shown at the researchers night but hasn't been used since then.

### 6.2.1 Nanoterra

The nanoterra is a large scale project. We will be part of it and prepare an application of HMM and sensors for didactical purposes. The contribution is in collaboration with the electronics laboratory at ETH.

### 6.2.2 Oberon HMM

Philip Tschiemer is working on an HMM running in Oberon. Applications would be visuals for parties and musical performances. Another idea is that he will continue the mind machine project.

### 6.2.3 Beat Em Up-Game

A beat em up-game will be controlled by our sensors. At first this will demonstrate the use of gesture recognition.

There are two approaches:

- The player wearing the sensors has to perform a desired move (kick or punch) completely in order to score. So on the screen there will be more or less the same movement as the player does. This would be nice for martial artists in the sense of shadow boxing. So the players don't touch each other but there's a full contact fight on the computer screen as motivation.
- The player just has to indicate a movement in a way and the computer recognizes the move and plays a stored movement instead. This way there's a stronger focus

on the fun aspect then on sports. Instead the game would be easier to play and less exhausting.

But also for the first possibility there would be a need for classifying movements by learning algorithms. The movements are going to be too fast for the sensors to provide data for the forward kinematics as we are doing it now. For example when kicking fast, the acceleration sensors measure a sum vector of the acceleration of the feet and the gravitation. You no more get a proper plumb line.

### 6.2.4 Joystick Simulation

Instead of using open source games to control them by our sensors, it's possible to control them by joystick. But instead of a usual joystick we intend to use the sensor system and a computer to simulate a joystick. So there is a game like Street Fighter or Tekken. When playing this game using a joystick special kicks and movements can be achieved by pressing the keys of the joystick and moving it in the according direction. Joystick movement and keystrokes could be simulated by a computer acting as a joystick. This is how it would be possible to use our sensors for a PlayStation or similar equipment without touching the game code.

### 6.2.5 Theremin

There is the idea to implement a theremin using the new 10 sensor system. A theremin is an electronical music instrument (figure 6.1). It basically produces a single tone at a time. Pitch and volume are controlled by the distance of the players hands from the antennas of the theremin. Using our sensor system we get the absolute coordinates of every limb. Therefore it will be easy to calculate the distance between the hands and a virtual antenna. But a usual theremin is so sensitive that even small movements of a single finger change the pitch by a halftone. Therefore for better accuracy applying one sensor per hand to the fingers would be helpful. Maybe it would be easiest to mount a single sensor to trigger finger and fore finger both together. in the end a virtual theremin might be quite attractive for the audience at future exhibitions.

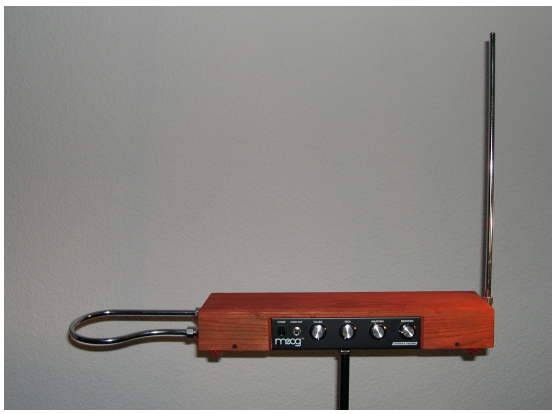


**Figure 6.1:** New model of a theremin

### 6.2.6 OpenGL

Unfortunately there is no DirectX on any MacOS or Linux. At least there is a simulation available using Wine but that's quite limited. Depending on the direction where the project is going, it might be a good idea to ponder about using OpenGL instead of DirectX. OpenGL runs on Windows, MacOS, Linux and other platforms. There is already an open source library GLUT possibly capable to replace the SNGLibrary.

## 6.3 Drawbacks

This section talks about discrepancies from the original project description.

- There was no merge with the existing Motion Capturing System (MoCapSys). There was long meeting on this topic. Within this project the MoCapSys hasn't been used. It doesn't offer a fullscreen mode. That's why it wasn't useful at the exhibitions. If was never planned to be used at such presentations. Nevertheless it offers nice graphics and allows to load any body model from file. It's compatible with the Maya 3D software. Therefore captured movements can be played back in Maya. The MoCapSys might be reused later. It could serve well for cutting and organizing captured movements. For this purpose it only will have to be adapted to interface with the new sensors. Using the SNServer offers an abstraction to directly reading the sensor data. Due to this in future only the connector will have to be adapted to new sensors.

- Abstract visualizations weren't considered much. But for the researchers night a system was presented to play with a swarm of cubes. Given the movements the swarm changed its behavior.

## 6.4 The Thesis & Myself

Why did I choose this thesis? I'm a student in computer science. I wrote this master thesis during my 11th semester. My master subject is software engineering. From the minor subjects I've chosen two. On one hand I took didactics and on the other hand I took robotics. Sensors play an important role in robotics. Didactics will help me for the NanoTerra project. Beside work I really enjoy sports. But I'm not practising Tai Chi, at least not yet. I'm into another martial art, Taekwondo.

## 6.5 Epilogue

It has been a a lot of work - at home and in the office - but nevertheless a big pleasure! As one can see in the section 6.1 about future work, there's still a lot of work left and plenty of ideas. Hopefully the project will continue and persist in a successful way. Personally I'm really looking forward to continue to work in the group. It's now to decide what we want to push on next. There are upcoming projects, exhibitions and conferences.

# Appendix A

# Appendix

## A.1 Operating Systems

Various operating systems (OS) have been used within this project. Prof. Gutknecht once said that the OS should be a gift and not a burden. We agree very much. There is no point in discussing which one is the best OS. It always depends on the application. We tried to use the most suitable OS for each case and kept the software platform independent as much as possible. Most of the tools can be compiled and ran under Linux and Windows. The SNServer should run on every platform since it's implemented in Java.

### A.1.1 Why Oberon

The Oberon OS has the nice property of being light weight. It was therefore very useful to realize a gateway to read the data from USB and send it in a fast stream over the serial connection to a windows computer. We thank Sven Stauber for his good work. Currently there is ongoing work to do much more with the sensor data on Oberon. Finally there will be a complete HMM running on AOS offering better performance than other OS.

### A.1.2 Why Windows

Windows is the usual OS. In Ranieri & Majoe (2007) DirectX is used for 3D drawing. DirectX is published by Microsoft. The AvatarPlayer is based on this work and therefore runs under MS Windows. A lot of commercial games run on Windows. Maybe that was the reason to choose MS Windows. Thats why the AvatarPlayer is difficult to run on Linux. But still Wine and CrossOver manage to run software depending on DirectX (World of Warcraft, Half-life) on Linux. Therefore they do some sort of emulation of DirectX using OpenGL. So it might be possible to run the AvatarPlayer under Linux but always coupled with a loss of performance.

### A.1.3 Why Linux

Linux has been used because of the wide availability of open source applications. To demonstrate the abilities of the gesture recognition we were looking for a nice game. What we found was SuperTux. It's a jump and run game and quite common under Linux. Open source games can easily be extended to be playable using our sensors. Later it will be possible to compile SuperTux for windows as well. But Linux has been chosen for development since it seemed to be the easiest way. Generally most of the Linux software can be ported to windows. Happily OpenGL is available for a wide variety of OS which happens to be untrue for DirectX.

## A.2 Setup and Compilation

### A.2.1 Dev-Cpp, Eclipse

Descriptions how to install Dev-Cpp and how to migrate to eclipse for the AvatarPlayer can be found in Widmer & Majoe (2008).

### A.2.2 Usb2Network & Usb2Features

Usb2Network and Usb2Features are VC++-projects. They have only been compiled using VC++ on Windows. VC++ is available for free on the internet. Use it to adapt

and recompile this two projects.

### A.2.3 In General

Every other part of the software has to be compiled with the GNU compiler *g++* and the *make* tool. These two tools are mandatory and need to be part of the path variable.

In Windows you can add them the following way if you have Dev-Cpp installed. If Dev-Cpp is located in C:\Dev-Cpp you can add it to the path using the following command line input. `path = %path%;C:\Dev-Cpp\bin;` If you have Dev-Cpp or the MinGW in another directory installed, you just need to adapt the path.

For Linux the following packages need to be installed: *g++*, *libsdl-dev* and *make*.

### A.2.4 AvatarPlayer

AvatarPlayer has been edited last using eclipse. The best way to edit it is to use eclipse again because it's quite a large project. Nevertheless it can be compiled on the command-line with *make* as well. Because the AvatarPlayer relies on DirectX it can only be used in Windows.

### A.2.5 Tools

All the other tools can be compiled using one *Makefile* for all. They are small projects and can be edited with the text editor of your choice. All the tools compile in Linux and Windows, except ControlKeys which has only been tested in Linux. There are two different files for *make Makefile* is for Linux, *Makefile.win* is for Windows. When compiling the first time us the according shell script instead of directly calling *make*. In Linux call *./makeLinux* in Windows us *makeWin.bat*.

### A.2.6 SuperTux

SuperTux has only been tested to compile in Linux. Basically it should be able to compile and run in Windows, too. To compile it enter the SuperTux directory with a

command shell. If you're compiling SuperTux for the first time on this PC, type *./configure*. Install all the missing software using synaptic. It's very easy. Configure offers a clear output what you have to look for in synaptic. Finally *./configure* should run without complains. Then you can type *make*. Make should run fine until it tries to link the application. Linking will fail because it doesn't automatically link our additional classes. Therefore enter the src directory and compile Helper.cpp, PracticalSocket.cpp, HandleServer.cpp and HandleServerSingletonKeeper.cpp by hand. You therefore use the following commands:

- g++ -c Helper.cpp -o Helper.o
- g++ -c PracticalSocket.cpp -o PracticalSocket.o
- g++ -c HandleServer.cpp -o HandleServer.o
- g++ -c HandleServerSingletonKeeper.cpp -o HandleServerSingletonKeeper.o

Now we can try again to link the SuperTux. But make alone will still fail because the Makefile as ./configure has generated it doesn't include our object files. Therefore we prepared another link script. Just type ./link.sh and everything should link fine. Now you can run ./supertux.

For SuperTux you can't enter the server to use on command line. If you need to change the address of the used server, streamname or port numbers for SuperTux, look in the HandleServerSingletonKeeper.cpp. Afterwards you only have to recompile this file and link the project again.

### A.2.7 Matlab

Matlab uses a directory "MATLAB" in your home folder. Put the directory "mod" from the CD-ROM to this folder.

Open your browser and go to the following URL:

http://www.mathworks.com/matlabcentral/fileexchange/345

Download the package and install it according to the instructions enclosed.

## A.3 Todo-List

### A.3.1 SNServer

- Allow bigger UDP packets than 255 Bytes.
- Check if actually for every packet send a new thread gets created. Try to reuse threads in order to save performance.
- Prepare server with a proper command line mode in order that it can run on a Linux server without graphical subsystem.

### A.3.2 FeatureExtraction

- Usb2Features works fine. But in the Tool application for feature extraction out of a raw data stream there isn't yet a sure check that a sensor always appears at the same position.

### A.3.3 HMM

- Use the existing C-code instead of Matlab.
- Test performance.

### A.3.4 Avatar Player

Source files with work apparently left open. There are remarks in the source files as well. So this list was generated by searching the source files for TODO remarks. But the descriptions have been rewritten in order to make the tasks more clear.

#### A.3.4.1 Camera.cpp

- Line number: 29
  Currently there is just a fixed camera position. At least the class for camera movement is prepared. Now also the calculation routines and variables for moving the camera should be prepared.

#### A.3.4.2 Capture.cpp

- Line number: 2
  Make two capturing classes:
  - CaptureXML (this one)
  - CaptureCSV

  Therefore Capture will become an interface. FileData should be adapted to deal with both formats. It should be able to determine the format on its own and provide the data transparent.

#### A.3.4.3 CueBall.cpp

- Line number: 141
  Remove the simple implementation of the decision to which limb the ball is attracted. For the first demonstration we just decided based on the sign of the balls X position.
- Line number: 178
  Should be (ox - x) * G3. Try out this changed formula. It has to evaluate to zero if only one ball is present. Also it would be nice to make the algorithm generic to n balls.
- Line number: 180
  Store old mx and add 1/2 oldmx to mx. This would give the ball some sort of a mass. Maybe there will some thought about the balls physics be needed as well.

#### A.3.4.4 Defines.cpp

- Line number: 3
  Adapt methods using generics. Most of the helper methods could be made generic.
- Line number: 4
  Change the names of the 2string-functions, so they use overloading

- Line number: 80
  Re-implement the sorting feature using insertion sort. This will be more efficient and the code will look better.

#### A.3.4.5 Limb.cpp

- Line number: 330
  Make getXRot also return the fixedRot if there's no sensor

#### A.3.4.6 MainClass.cpp

- Line number: 219
  Just have one method to read the keyboard and react on the keys. Rewrite parts of the MainClass and change the main routine in order to have key management in a single feature. If possible use a simple sort of events. So listeners can subscribe themselves for key-pressed events.

#### A.3.4.7 Various

- Fix re-initialization of DirectX when switching the task back from another application.
- Fix reloading of the first capture data 01.txt after showing other movements.
- Add support for mouse and icons to the software.

### A.3.5 Generally

- Correct typing mistakes in comments
- Add whitebox tests
- Fix re-initialization of DirectX when switching the task back from another application.
- Fix reloading of the first capture data 01.txt after showing other movements.

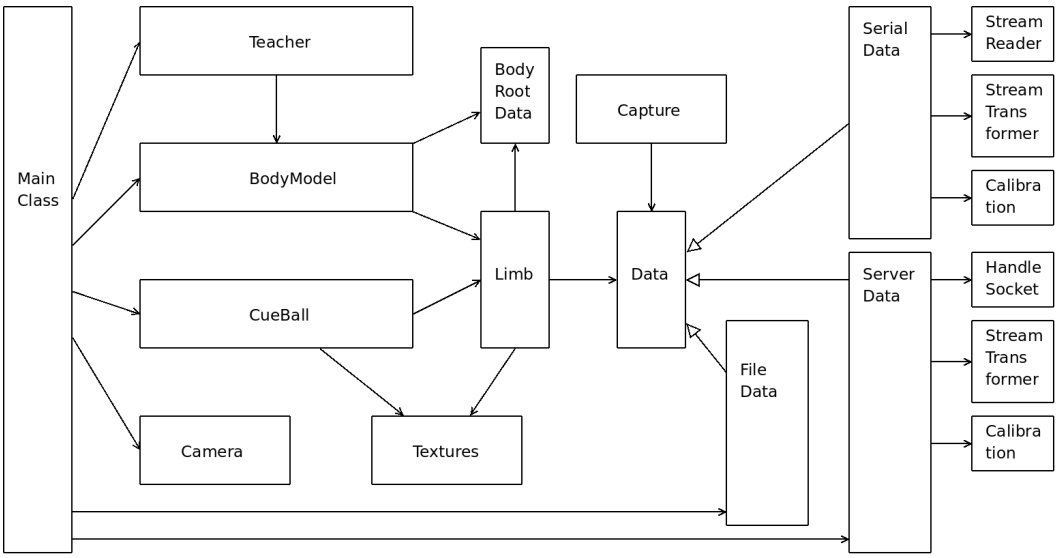



**Figure A.1:** New class diagram of the software project

- Add support for mouse and icons to the software.
- Run tests under heavy load.
- Arrange projects in a proper structure so that every file appears only once. This means shared files (classes etc.) are accessed from every tool at the same location.
- If useful prepare USB connector for Linux.

## A.4 Class Diagram

Most of the tools are not too large. Except the avatar player. That's why a closer look at its class diagram in figure A.1 will be presented . Documentation of the SNServer can be found at Ranieri & Majoe (2007).

You can see the class diagram for the software structure in figure A.1.

The smaller arrows depict an aggregation ("has-a" relationship). The bigger with a closed head arrows depict an inheritance ("is-a" relationship).

MainClass serves as a helper for the main-routine of the software. With main-routine I mean the top-level routine which is executed automatically, when the software is

started. So the usage of the MainClass makes the main-routine smaller and more manageable.

For the avatars we need two instances of BodyModel. BodyModel itself has a lot of instances of Limb to form the body. And the Limb objects have a Data object to get the current angles. Actually there is only one object of FileData and ServerData but the Limb objects all have a pointer to it. Data is a superclass of FileData, SerialData and ServerData. Therefore you're free to you use FileData, SerialData or ServerData to drive a BodyModel as an avatar. StreamTransformer does the vector calculation to get the Euler angles from the raw sensor data. FileData allows to read those text files and use them instead of actual sensor data. SerialData and ServerData provide the actual sensor data for displaying the avatar according to the pose of the user wearing the sensors. SerialData is used to read the data over the serial port. It serves as an abstraction layer on top of StreamReader. SerialData has an object of type StreamTransformer and Calibration. But it's currently unused because we use ServerData to read data from the SNServer. Instead of StreamReader ServerData has an instance of HandleServer. StreamTransformer prepares the Euler angles given the raw sensor data. Calibration offers a comfortable way to calibrate the sensors using stored files with the prepared calibration data.

The Teacher class implements the comparison of the two avatars and provides the according feedback. Therefore a Teacher instance needs references to the two BodyModel instances in order to compare them. The Camera class will implement camera movement. For now there's just a fixed point of view. The CueBall class holds the code for a virtual ball to play with. In our application the ball follows the hands. Therefore every CueBall instance has to have a reference of a limb which it will follow. You're basically free to hand over any limb you wish. New there is also sort of a throwing mode supported. In this mode the CueBall instance keeps a list of limbs. It always gets attracted to the nearest limb on the list. In this mode you can pass the ball from one limb to another. Limb and CueBall use Textures. A texture means a skin for an object on the screen. The Textures class allows an easy use of textures based on graphic files. BodyRootData is just a small class storing a displacement for a whole avatar. The reason why it's not part of BodyModel is because all the Limb instances also need the displacement information. Therefore for every Avatar there should be one instance of BodyRootData and a reference to this instance is stored in every Limb instance. Capture offers a file writer for saving sensor data to a list of XML structures

in a text file.

## A.5 Simple Examples

The idea of this section is to simplify the start with the software for new students. These simple examples can be understood within minutes and provide a basic understanding for the used technologies. Examples for reading data over the serial port and drawing 3D can be found in Widmer & Majoe (2008).

### A.5.1 Reading Server Data (Subscriber)

This example program gives you an overview how to read sensor data. It's a test receiver for the server UDP streams. Most of the work is done in the class Handle Server. Nevertheless this class offers functionality for nearly every application. The test receiver just outputs the data to standard out. Therefore you can use it via pipes and redirections.

Usage: *Receiver[.exe] [<Server> <Server TCP Port> <Server UDP Port> <Stream Name> <Subscriber Name> [<Stream Password>]]*

This source has been successfully compiled and tested under Ubuntu Linux and MS Windows XP.

**Listing A.1:** Receiver.cpp

```
1 #include "../src/PracticalSocket.h"
2 #include "../src/HandleServer.h"
3 #include "../src/Helper.h"
4 #include <iostream>
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int main (int argc, char *argv[]) {
9     HandleServer * hs = new HandleServer(true,argc,argv);
10
11     while (Helper::kbhit() < 1) {
12         string pck = hs->receiveUDP();
```

```
13          cout << pck << endl;
14      }
15
16      delete(hs);
17  }
```

### A.5.2 Sending Data to the Server (Publisher)

A simple test sender. This tool reads from standard input and sends it via UDP socket. Using a command line like:

*Receiver[.exe] mona.lawi.ch 2345 9777 phil larslistener | Sender localhost 2345 phil2*

You can send data from one server to another or duplicate a stream.

Usage: *Sender[.exe] [<Server> <Server TCP Port> <Server UDP Port> <Stream Name> <Subscriber Name> [<Stream Password>]]*

This source has been successfully compiled and tested under Ubuntu Linux and MS Windows XP.

**Listing A.2:** Sender.cpp

```
1  #include "../src/PracticalSocket.h"
2  #include "../src/HandleServer.h"
3  #include "../src/Helper.h"
4  #include <iostream>
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int main (int argc, char *argv[]) {
9      HandleServer * hs = new HandleServer(false,argc,argv);
10
11     string toSend;
12     char in[256];
13     while (!cin.eof()) {
14         toSend.clear();
15         cin.getline(in,256);
16         toSend = in;
```

```
17          hs->sendUDP(toSend);
18      }
19
20      delete(hs);
21  }
```

## A.6 Computation Method for Euler Angles

This method gets used to compute the Euler angles out of the raw sensor data.

**Listing A.3:** transform.cpp

```
1  EulerRotation StreamTransformer::
2      transform(NodeData newData, NodeData calibrationData) {
3
4      EulerRotation result;
5      result.xRotation = result.yRotation
6          = result.zRotation = 0;
7      float sok = false;
8      if(newData.valid) {
9          sok = true;
10         newData.valid=false;
11
12         // Apply calibration
13         float accX = (newData.accX - calibrationData.accX);
14         float accY = (newData.accY - calibrationData.accY);
15         float accZ = (newData.accZ - calibrationData.accZ);
16
17         float magX = (newData.magX - calibrationData.magX);
18         float magY = (newData.magY - calibrationData.magY);
19         float magZ = (newData.magZ - calibrationData.magZ);
20
21         // calculate absolute values
22         float absaccZ;
23         float absaccY;
24         float absaccX;
25         if(accZ < 0.0f)
```

```
26              absaccZ = -accZ;
27          else
28              absaccZ =  accZ;
29          if(accY < 0.0f)
30              absaccY = -accY;
31          else
32              absaccY =  accY;
33          if(accX < 0.0f)
34              absaccX = -accX;
35          else
36              absaccX =  accX;
37
38          // calculate the signs
39          int signX = (int)(accX/absaccX);
40          int signY = (int)(accY/absaccY);
41          int signZ = (int)(accZ/absaccZ);
42
43          // correct the orientation of the magnetometer data
44          // Afterwards the coordinate system for the magn.
45          // and the acc. data is the same
46          magX *= -1.0;
47          magY *= -1.0;
48
49          // calculate raw vector sizes
50          // On a limited system (embedded device) this values
51          // can be replaced by constants
52          // Nevertheless, performance is a piece of cake
53          // compared to semantic correctness
54          float accVecSize =
55              sqrt(pow(accX,2)+pow(accY,2)+pow(accZ,2));
56          float magVecSize =
57              sqrt(pow(magX,2)+pow(magY,2)+pow(magZ,2));
58
59          // normalize vectors
60          accX /= accVecSize;
61          accY /= accVecSize;
62          accZ /= accVecSize;
```

```
63          magX /= magVecSize;
64          magY /= magVecSize;
65          magZ /= magVecSize;
66
67          // compute helping values
68          // square of gravitation
69          float sqrg = pow(accX,2)+pow(accY,2)+pow(accZ,2);
70          float grav = sqrt(sqrg); // gravitation
71          float gravYZ = sqrt(pow(accY,2)+pow(accZ,2));
72
73          // compute pitch and roll
74          float pitch = asin(accX/grav);
75          float roll = signY * acos(accZ/gravYZ);
76
77          // For the computation of azimuth we introduce
78          // a plane G through (0,0,0) rectangular
79          // to the gravitation
80          // This is exactly the place where we want
81          // to measure azimuth!
82          // compute the height of the mag-vector above G
83          float heightA =
84              - (accX*magX + accY*magY + accZ*magZ) / sqrg;
85          // compute the orthogonal projection
86          // vector x of the mag-vector in G
87          float x1 = magX + accX*heightA;
88          float x2 = magY + accY*heightA;
89          float x3 = magZ + accZ*heightA;
90          // length of x
91          float lenX = sqrt(pow(x1,2)+pow(x2,2)+pow(x3,2));
92
93          // reference vector
94          float r1 = 1.0;
95          float r2 = 0.0;
96          float r3 = 0.0;
97          float lenR = sqrt(pow(r1,2)+pow(r2,2)+pow(r3,2)); // 1
98
99          // compute the height of the reference vector above G
```

```
100         float heightB =
101             - (accX*r1 + accY*r2 + accZ*r3) / sqrg;
102         // compute the orthogonal projection vector
103         // y of the reference vector in G
104         float y1 = r1 + accX*heightB;
105         float y2 = r2 + accY*heightB;
106         float y3 = r3 + accZ*heightB;
107         // length of y
108         float lenY = sqrt(pow(y1,2)+pow(y2,2)+pow(y3,2));
109
110         // we now have x and y in the same plane
111         // whereas the angle between
112         // them is our beloved azimuth
113         float xy = x1*y1 + x2*y2 + x3*y3; // scalar product
114         float azimuth = acos(xy/(lenX*lenY)); // azimuth angle
115
116         // find vector product of acc & mag
117         // ==> vector t, orthogonal to acc & mag
118         float t1 = accY*magZ - accZ*magY;
119         float t2 = accZ*magX - accX*magZ;
120         float t3 = accX*magY - accY*magX;
121
122         // length of the projection of
123         // the reference vector on t
124         float b =
125             - (t1*r1+t2*r2+t3*r3) /
126                 (pow(t1,2)+pow(t2,2)+pow(t3,2));
127
128         // reintroduce the sign of the azimuth
129         if (b < 0) {
130             azimuth *= -1.0;
131         }
132         result.xRotation = roll;
133         result.yRotation = pitch;
134         result.zRotation = azimuth;
135     }
136     return result;
```

```
137 }
```

# Appendix B

# Source Documentation

The documentation is based on comments in the source code. It has been generated using the Doxygen software. You can find the whole source code on the CD.

## B.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

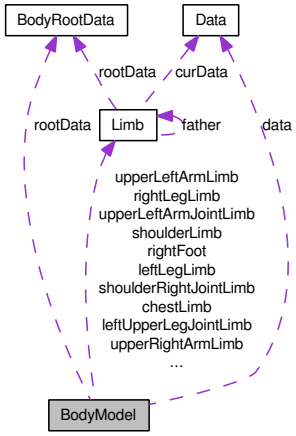
## B.2 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

## B.3 BodyModel Class Reference

`#include <BodyModel.h>`

Collaboration diagram for BodyModel:




### Public Member Functions

- BodyModel (Data *d, BodyRootData *rd)
- virtual ~BodyModel ()
- void draw ()
- Limb * getLimb (string)
- Limb * getLimb (int)
- vector< Limb * > getLimbs ()
- void setSensors (vector< int >)
- vector< int > getSensors ()
- void setTexture (string)

### Private Member Functions

- void initialize ()

## Private Attributes

- Data * **data**
- BodyRootData * **rootData**
- Limb * **chestLimb**
- Limb * **headLimb**
- Limb * **shoulderLimb**
- Limb * **shoulderLeftJointLimb**
- Limb * **upperLeftArmLimb**
- Limb * **upperLeftArmJointLimb**
- Limb * **leftArmLimb**
- Limb * **leftArmJointLimb**
- Limb * **shoulderRightJointLimb**
- Limb * **upperRightArmLimb**
- Limb * **upperRightArmJointLimb**
- Limb * **rightArmLimb**
- Limb * **rightArmJointLimb**
- Limb * **pelvisLimb**
- Limb * **leftPelvisJointLimb**
- Limb * **leftUpperLegLimb**
- Limb * **leftUpperLegJointLimb**
- Limb * **leftLegLimb**
- Limb * **leftFoot**
- Limb * **rightPelvisJointLimb**
- Limb * **rightUpperLegLimb**
- Limb * **rightUpperLegJointLimb**
- Limb * **rightLegLimb**
- Limb * **rightFoot**
- vector< Limb * > **limbs**

### B.3.1 Detailed Description

Class for modelling a complete human body. E.g. If you want to model a cat you will have to adapt this class. The class has instances of Limb, Data and BodyRootData.

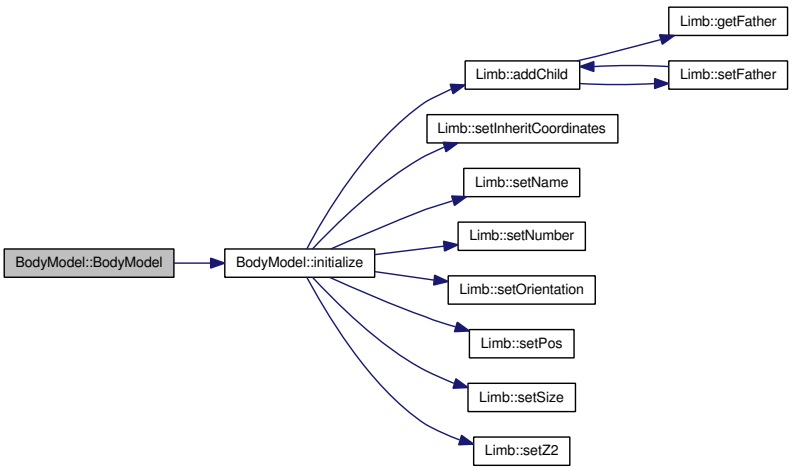
Author: Lars;

### B.3.2 Constructor & Destructor Documentation

#### B.3.2.1 BodyModel::BodyModel (Data * *d*, BodyRootData * *rd*)

Constructor: References to objects of type Data and BodyRootData have to be handed over. Data holds the sensor measurements determining the angles for every limb. BodyRootData sets the positioning of the whole body. There isn't any constructor without parameters.

Here is the call graph for this function:




#### B.3.2.2 BodyModel::~BodyModel () [virtual]

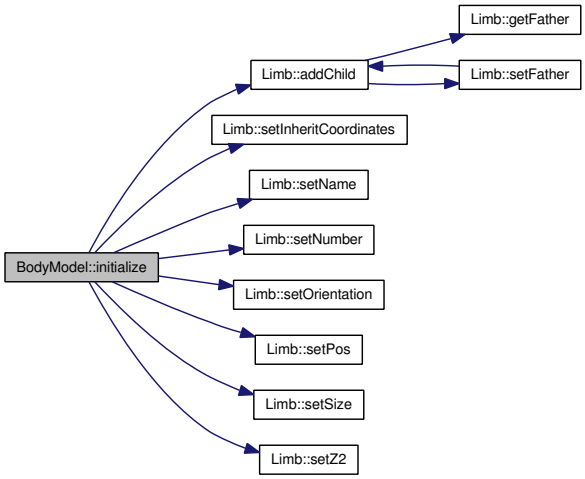
Destructor: Disposes all the limbs of the body

## B.3.3 Member Function Documentation

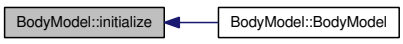
### B.3.3.1 void BodyModel::initialize () [private]

Private separate method used to initialize and set up the whole structure of the body.

Here is the call graph for this function:



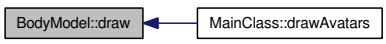

Here is the caller graph for this function:




### B.3.3.2 void BodyModel::draw ()

Draws the body to the screen. It therefore calls the draw methods of the limbs it consists of.

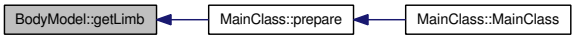
Here is the caller graph for this function:



#### B.3.3.3 Limb ∗ BodyModel::getLimb (string *str*)

Returns a reference to a limb of the body. Therefore it searches through the limbs for the first limb with the same name as the given string. Returning a reference is leaking, the private limb can be changed from outside.

Here is the caller graph for this function:




#### B.3.3.4 Limb ∗ BodyModel::getLimb (int *i*)

Returns a reference to a limb of the body. The limb returned is the one at the given position in the internal vector. That's leaking, the private limb can be changed from outside.

#### B.3.3.5 vector< Limb ∗ > BodyModel::getLimbs ()

Returns the whole vector of limbs This is leaking. The private vector can be changed from outside through the link.

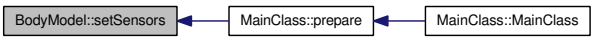
Here is the caller graph for this function:


#### B.3.3.6 void BodyModel::setSensors (vector< int > *sv*)

Assigns the sensors to the body model in a predefined order: 0: left Wrist, hand 1: left upper arm, elbow 2: chest, shoulders, head 3: right upper arm, elbow 4: right wrist, hand

Here is the call graph for this function:

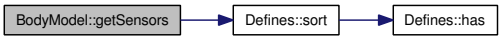
Here is the caller graph for this function:



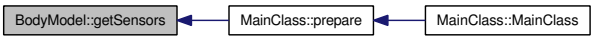

#### B.3.3.7 vector< int > BodyModel::getSensors ()

Returns a vector containing all the sensors of the body. The list contains every value only once and is in increasing order.

Here is the call graph for this function:



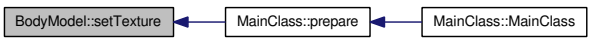

Here is the caller graph for this function:




#### B.3.3.8 void BodyModel::setTexture (string *str*)

Sets the texture for the whole body. The texture is given by the filename of the picture file. Nether path nor extension have to be given. Textures reside in the resources directory.

Here is the caller graph for this function:




The documentation for this class was generated from the following files:

- BodyModel.h
- BodyModel.cpp

## B.4 BodyRootData Class Reference

`#include <BodyRootData.h>`

### Public Member Functions

- BodyRootData ()
- virtual ~BodyRootData ()
- void setShift (float, float, float)
- float getShiftX ()
- float getShiftY ()
- float getShiftZ ()
- void setRoot (float, float, float)
- float getRootX ()
- float getRootY ()
- float getRootZ ()

### Private Attributes

- float **shiftX**
- float **shiftY**
- float **shiftZ**
- float **rootX**
- float **rootY**
- float **rootZ**

### B.4.1 Detailed Description

Basically this class is used to store a displacement in every dimension. The displacement is called shift. We use it for the Body Models. So every BodyModel has a BodyRootData. The shift is used to draw the models separate or merge them.

Author: Lars;

## B.4.2 Constructor & Destructor Documentation

### B.4.2.1 BodyRootData::BodyRootData ()

Constructor: Set shift to zero.

### B.4.2.2 BodyRootData::~BodyRootData () [virtual]

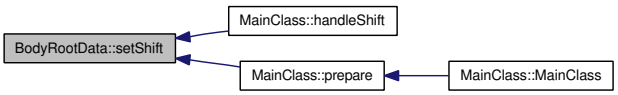
Empty destructor

## B.4.3 Member Function Documentation

### B.4.3.1 void BodyRootData::setShift (float *x*, float *y*, float *z*)

Set the shifts in every dimension in one call.

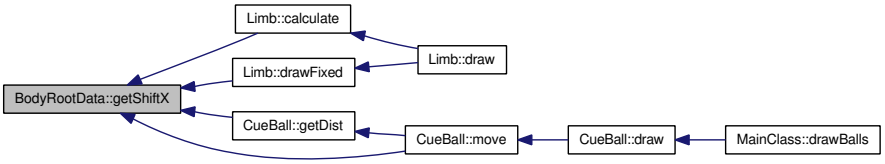
Here is the caller graph for this function:




### B.4.3.2 float BodyRootData::getShiftX ()

Returns the X part of the Shift.

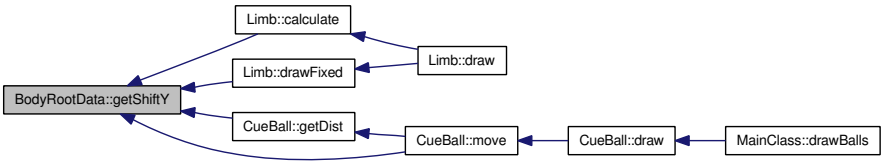
Here is the caller graph for this function:



#### B.4.3.3 float BodyRootData::getShiftY ()

Returns the Y part of the Shift.

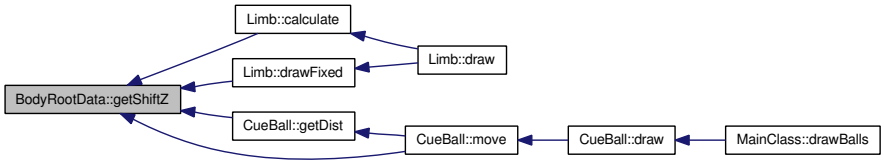
Here is the caller graph for this function:




#### B.4.3.4 float BodyRootData::getShiftZ ()

Returns the Z part of the Shift.

Here is the caller graph for this function:




#### B.4.3.5 void BodyRootData::setRoot (float *x*, float *y*, float *z*)

Set the root Values in every dimension in one call.

#### B.4.3.6 float BodyRootData::getRootX ()

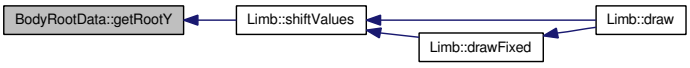
Returns the X root value.

Here is the caller graph for this function:

#### B.4.3.7 float BodyRootData::getRootY ()

Returns the Y root value.

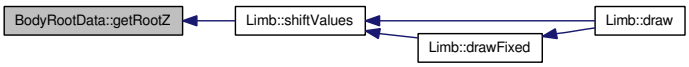
Here is the caller graph for this function:




#### B.4.3.8 float BodyRootData::getRootZ ()

Returns the Z root value.

Here is the caller graph for this function:




The documentation for this class was generated from the following files:

- BodyRootData.h
- BodyRootData.cpp

## B.5 Calibration Class Reference

`#include <Calibration.h>`

### Public Member Functions

- Calibration ()
- virtual ~Calibration ()
- NodeData **calibrateSensor** (int)
- int **getNumOfNodes** ()

### Private Member Functions

- void getcalibrationdata (int, int *, int *, int *, int *, int *, int *, int *, int *)
- string getCalFilename (int)
- int **getIndexOfNode** (int)

### Private Attributes

- vector< NodeData > **dataOffset**
- vector< NodeData > **dataScale**
- vector< int > **nodes**

### B.5.1 Detailed Description

Class for easy calibrating the sensors. The sensors each have different offsets. Therefore when preparing them their offsets are measured and stored into a text file. This class now reads those files and returns the offsets. Reading the files is transparent for the user of the class. The user just asks for the calibration data of a given node. If the file hasn't been read yet, it gets read and evaluated. Afterwards the data is stored in the class fields. So every file gets at most read once.

Author: Lars;

## B.5.2 Constructor & Destructor Documentation

### B.5.2.1 Calibration::Calibration ()

Class constructor:

### B.5.2.2 Calibration::~Calibration () [virtual]

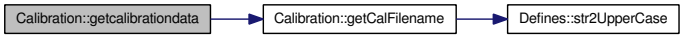
Class destructor:

## B.5.3 Member Function Documentation

### B.5.3.1 void Calibration::getcalibrationdata (int *sensor*, int * *intmagxoffset*, int * *intmagyoffset*, int * *intmagzoffset*, int * *intacczmax*, int * *intacczmin*, int * *intaccymax*, int * *intaccymin*, int * *intaccxmax*, int * *intaccxmin*) [private]

Read the calibration data for one sensor out of a file on disk. The data has to be prepared and stored in a file. The filenames are set by the defines (Defines.h).

Here is the call graph for this function:




### B.5.3.2 string Calibration::getCalFilename (int *sensor*) [private]

The calibration data files have all the same filename expect from a number. Therefore it easy to return the file name. for the given number. That what this method does.

Here is the call graph for this function:

Here is the caller graph for this function:


The documentation for this class was generated from the following files:

- Calibration.h
- Calibration.cpp

## B.6 Camera Class Reference

`#include <Camera.h>`

### Public Member Functions

- Camera ()
- virtual ~Camera ()
- void position ()
- float getPosX ()
- float getPosY ()
- float getPosZ ()
- float getRotX ()
- float getRotY ()
- float getRotZ ()
- float getScaX ()
- float getScaY ()
- float getScaZ ()
- void setPosX (float)
- void setPosY (float)
- void setPosZ (float)
- void setRotX (float)
- void setRotY (float)
- void setRotZ (float)
- void setScaX (float)
- void setScaY (float)
- void setScaZ (float)

### Private Attributes

- float **posX**
- float **posY**

- float **posZ**
- float **rotX**
- float **rotY**
- float **rotZ**
- float **scaX**
- float **scaY**
- float **scaZ**

## B.6.1 Detailed Description

This class should be used to allow the user to move the camera. Currently only a fixed camera position is supported.

Author: Lars;

## B.6.2 Constructor & Destructor Documentation

### B.6.2.1 Camera::Camera ()

Constructor: Set a default position of the camera.

### B.6.2.2 Camera::~Camera () [virtual]

The Destructor is empty.

## B.6.3 Member Function Documentation

### B.6.3.1 void Camera::position ()

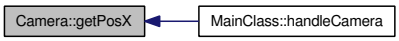
Currently this method just resets the default position. In a more sophisticated state this class should enable to virtually fly around the scene.

Here is the caller graph for this function:

#### B.6.3.2 float Camera::getPosX ()

Returns the X position of the camera.

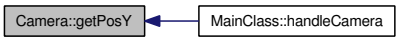
Here is the caller graph for this function:




#### B.6.3.3 float Camera::getPosY ()

Returns the Y position of the camera.

Here is the caller graph for this function:



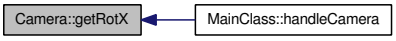

#### B.6.3.4 float Camera::getPosZ ()

Returns the Z position of the camera.

#### B.6.3.5 float Camera::getRotX ()

Returns the X rotation of the camera.

Here is the caller graph for this function:




#### B.6.3.6 float Camera::getRotY ()

Returns the Y rotation of the camera.

Here is the caller graph for this function:


#### B.6.3.7 float Camera::getRotZ ()

Returns the Z rotation of the camera.

#### B.6.3.8 float Camera::getScaX ()

Returns the X scaling of the camera view.

#### B.6.3.9 float Camera::getScaY ()

Returns the Y scaling of the camera view.

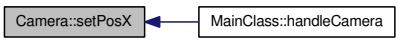
#### B.6.3.10 float Camera::getScaZ ()

Returns the Z scaling of the camera view.

#### B.6.3.11 void Camera::setPosX (float *f*)

Sets the X position of the camera.

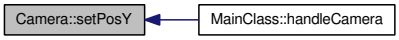
Here is the caller graph for this function:




#### B.6.3.12 void Camera::setPosY (float *f*)

Sets the Y position of the camera.

Here is the caller graph for this function:




#### B.6.3.13 void Camera::setPosZ (float *f*)

Sets the Z position of the camera.

#### B.6.3.14 void Camera::setRotX (float *f*)

Sets the X rotation of the camera.

Here is the caller graph for this function:


#### B.6.3.15 void Camera::setRotY (float *f*)

Sets the Y rotation of the camera.

Here is the caller graph for this function:


#### B.6.3.16 void Camera::setRotZ (float *f*)

Sets the Z rotation of the camera.

#### B.6.3.17 void Camera::setScaX (float *f*)

Sets the X scaling of the camera view.

#### B.6.3.18 void Camera::setScaY (float *f*)

Sets the Y scaling of the camera view.

#### B.6.3.19 void Camera::setScaZ (float *f*)

Sets the Z scaling of the camera view.

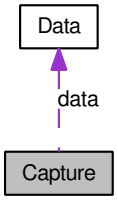
The documentation for this class was generated from the following files:

- Camera.h
- Camera.cpp

## B.7 Capture Class Reference

`#include <Capture.h>`

Collaboration diagram for Capture:




### Public Member Functions

- Capture (string, Data ∗, vector< int >)
- virtual ∼Capture ()
- void write ()

### Private Attributes

- string **fname**
- Data ∗ **data**
- vector< int > **sensors**
- ofstream **file**
- int **frame**

### B.7.1 Detailed Description

This class's object is used for capturing the data. It's not extactly motion capturing. We capture the sensor data. Of course based on this sensor data the motions can be replayed.

Author: Lars;

## B.7.2 Constructor & Destructor Documentation

### B.7.2.1 Capture::Capture (string *fn*, Data * *d*, vector< int > *s*)

Constructor: There is only this version of the constructor. Three things have to be handed over:

- The filename, where the captured data has to be stored.
- The reference to the sensor data.
- A vector containing the numbers (int) of the sensors to be captured. The sensor data behind the reference can change everytime. But the reference will stay the same. That's the reason why we don't need to repeatedly hand over sensor data to capture it.

### B.7.2.2 Capture::~Capture () `[virtual]`

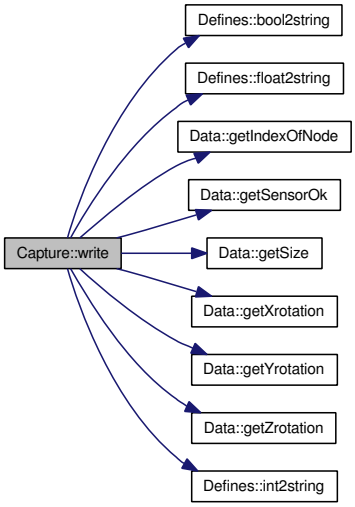
Destructor: Just closes the capturing file.

## B.7.3 Member Function Documentation

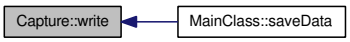
### B.7.3.1 void Capture::write ()

Writes the current sensor data to the capturing file. Some simple sort of an XML structre is used. Therefore the files are human readable

Here is the call graph for this function:




Here is the caller graph for this function:



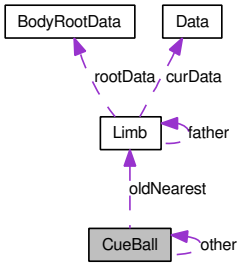

The documentation for this class was generated from the following files:

- Capture.h
- Capture.cpp

## B.8 CueBall Class Reference

`#include <CueBall.h>`

Collaboration diagram for CueBall:




### Public Member Functions

- CueBall ()
- void addLimb (Limb ∗l)
- virtual ∼CueBall ()
- void setOther (CueBall ∗)
- float getX ()
- float getY ()
- float getZ ()
- void draw ()
- void setTexture (string)

### Private Member Functions

- void initialize ()
- void check ()
- void move ()
- float getDist (Limb ∗l)

### Private Attributes

- Limb * **oldNearest**
- vector< Limb * > **limbs**
- CueBall * **other**
- float **mx**
- float **my**
- float **mz**
- float **x**
- float **y**
- float **z**
- float **oldLimbX**
- float **oldLimbY**
- float **oldLimbZ**
- string **texture**

## B.8.1 Detailed Description

The class CueBall is used to get some virtual balls on the screen to play with. If there are two balls they can be linked, so they have an attraction to each other. Basically the balls are attracted to a given limb of a body. The behavior is like having a rubber band connecting the ball with the limb. It's also possible to keep a set of limbs for each ball. The ball itself determines to which of the given limbs its nearest. Thats the limb to which the ball is attracted. The speed (change of position) of the limb has a high influence on the balls movement.

Author: Lars;

## B.8.2 Constructor & Destructor Documentation

### B.8.2.1 CueBall::CueBall ()

Default constructor without parameters. Initializes the properties with default values.

Here is the call graph for this function:


#### B.8.2.2 CueBall::~CueBall () [virtual]

Empty destructor

### B.8.3 Member Function Documentation

#### B.8.3.1 void CueBall::initialize () [private]

Private method for initializing the properties. This method gets called by the constructor.

Here is the caller graph for this function:


#### B.8.3.2 void CueBall::check () [private]

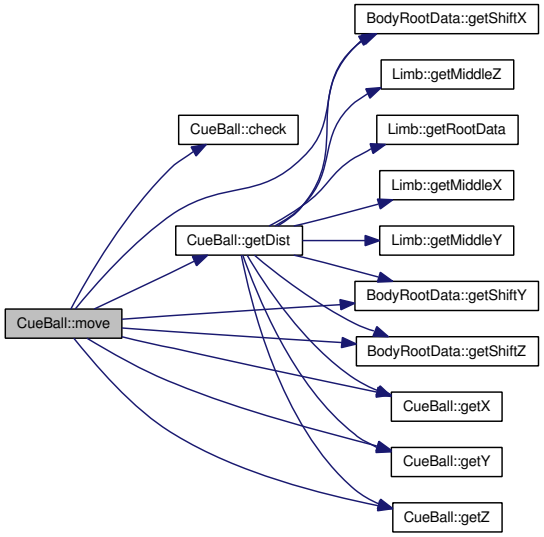
Private method for checking if the ball is properly initialized and therefore ready to fly. This means it has at least one limb to attract to.

Here is the caller graph for this function:


#### B.8.3.3 void CueBall::move () [private]

Private method to gather some measurements and move the ball accordingly.

Here is the call graph for this function:

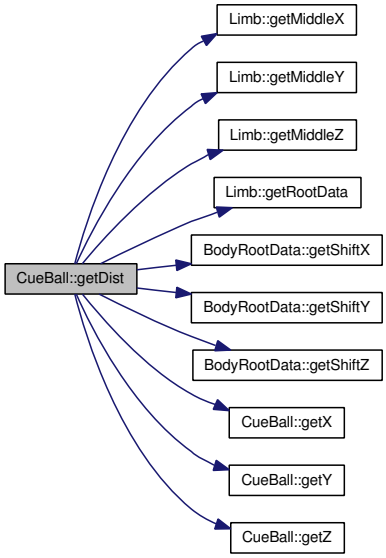



Here is the caller graph for this function:

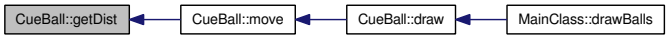

#### B.8.3.4 float CueBall::getDist (Limb * *l*) [private]

Private method to calculate the diagonal distance between a given limb and the ball itself.

Here is the call graph for this function:



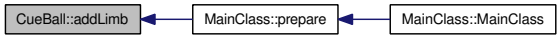

Here is the caller graph for this function:




### B.8.3.5 void CueBall::addLimb (Limb ∗ *l*)

The ball stores a vector of limbs. This function allows to add a limb to this vector. From this vector the ball selects the nearest limb. And to this limb it gets attracted.

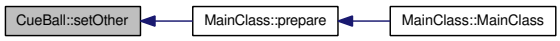
Here is the caller graph for this function:




### B.8.3.6 void CueBall::setOther (CueBall ∗ *cb*)

It's possible to have two balls attracting each other. Every ball then has to know which one is the other ball. For this purpose this method is used.

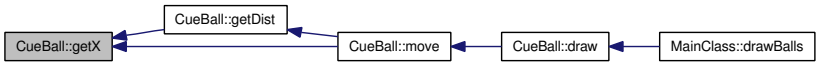
Here is the caller graph for this function:




### B.8.3.7 float CueBall::getX ()

Returns the current x position of the ball.

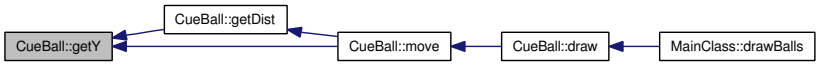
Here is the caller graph for this function:




### B.8.3.8 float CueBall::getY ()

Returns the current y position of the ball.

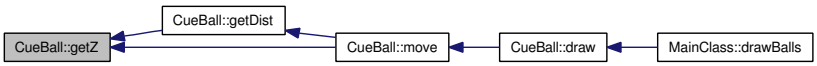
Here is the caller graph for this function:




### B.8.3.9 float CueBall::getZ ()

Returns the current z position of the ball.

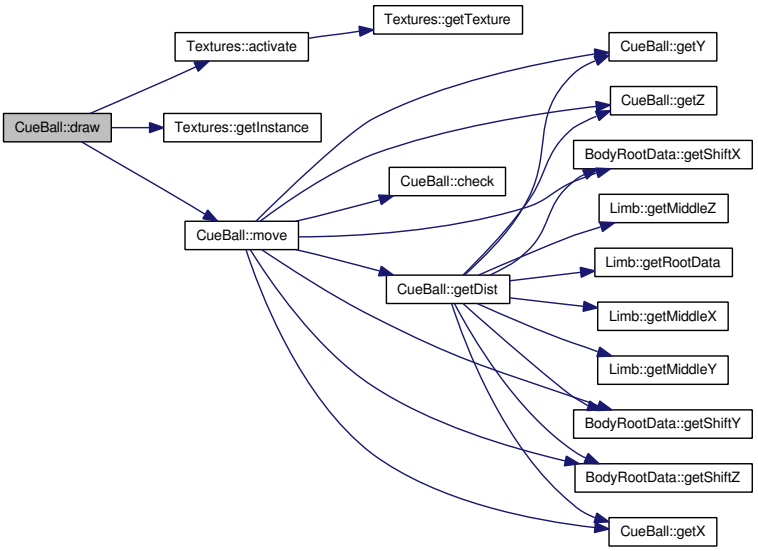
Here is the caller graph for this function:



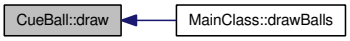
### B.8.3.10 void CueBall::draw ()

Moves and draws the ball.

Here is the call graph for this function:



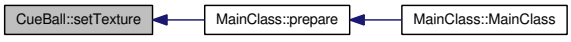

Here is the caller graph for this function:




### B.8.3.11 void CueBall::setTexture (string *str*)

Sets the texture in which the ball will be painted. Only the filename without path or extension has to be provided.

Here is the caller graph for this function:




The documentation for this class was generated from the following files:

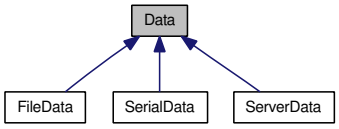
- CueBall.h

- CueBall.cpp

## B.9 Data Class Reference

`#include <Data.h>`

Inheritance diagram for Data:




### Public Member Functions

- Data ()
- virtual ~Data ()
- float **getSensorData** (int)
- EulerRotation getRotation (int)
- float getXrotation (int)
- float getYrotation (int)
- float getZrotation (int)
- bool getSensorOk (int)
- int getNode (int)
- int getIndexOfNode (int)
- int getSize ()
- string getName ()
- void setName (string)
- void **refresh** ()

### Protected Member Functions

- void setSensor (int, int, float, float, float, bool)
- void addSensor (int, float, float, float, bool)
- void setSensor (int, int, EulerRotation, bool)
- void addSensor (int, EulerRotation, bool)

- void addSensor (int)
- void clear ()

## Private Attributes

- vector< float > **sensorOk**
- vector< int > **node**
- string **name**
- vector< EulerRotation > **rotationData**

### B.9.1 Detailed Description

An object of this class is used to hold the sensor data. This class can't be used as it is. It's more an interface. DataSource classes have to inherit from this class. DataSources currently are: FileData, SerialData and ServerData. Then applications like Capture or BodyModel just use an instance of Data. If the get an instance of FileData, SerialData and ServerData doesn't matter.

Author: Lars;

### B.9.2 Constructor & Destructor Documentation

#### B.9.2.1 Data::Data ()

Constructor sets the properties to NULL (uninitialized).

Here is the call graph for this function:


#### B.9.2.2 Data::~Data () [virtual]

Empty Destructor

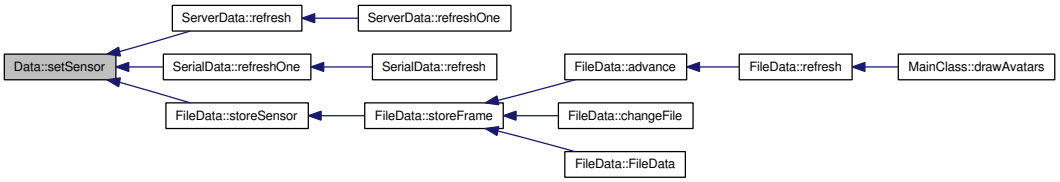
### B.9.3 Member Function Documentation

#### B.9.3.1 void Data::setSensor (int *position*, int *node*, float *xrotation*, float *yrotation*, float *zrotation*, bool *sensorOk*) [protected]

Sets the measurements for a sensor in the internal cache. node is the actual sensor ID. Position is the internal index as returned by getIndexOfNode().

Here is the call graph for this function:

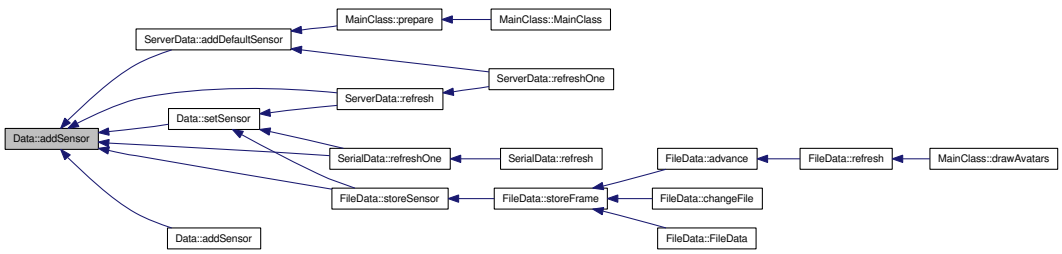

Here is the caller graph for this function:




#### B.9.3.2 void Data::addSensor (int *node*, float *xrotation*, float *yrotation*, float *zrotation*, bool *sensorOk*) [protected]

Adds the measurements for a sensor to the internal cache. node is the actual sensor ID. Use this method if the sensor hasn't been used yet. This is the case if getIndexOfNode() returns the value NO_SENSOR.

Here is the caller graph for this function:



#### B.9.3.3 void Data::setSensor (int *index*, int *node*, EulerRotation *newData*, bool *valid*) [protected]

Sets the euler angles for a sensor in the internal cache. index is the position of the sensor in the cache. index is NOT the actual sensor ID! Use getIndexOfNode() to get Index. Use this method if the sensor has been used yet. If NO_SENSOR is passed for index, automatically addSensor() gets used internally.

Here is the call graph for this function:


#### B.9.3.4 void Data::addSensor (int *node*, EulerRotation *newData*, bool *valid*) [protected]

Adds the euler angles for a sensor to the internal cache. node is the actual sensor ID. Use this method if the sensor hasn't been used yet. This is the case if getIndexOfNode() returns the value NO_SENSOR.

#### B.9.3.5 void Data::addSensor (int *node*) [protected]

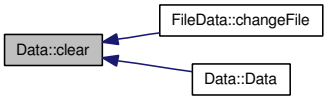
Adds a sensor to the internal cache. For the euler angles default values are set. node is the actual sensor ID. Use this method if the sensor hasn't been used yet. This is the case if getIndexOfNode() returns the value NO_SENSOR.

Here is the call graph for this function:


#### B.9.3.6 void Data::clear () [protected]

Removes the internal cache for sensor data. Afterwards all the sensors have to be added again. For just refreshing the values only call refresh().

Here is the caller graph for this function:



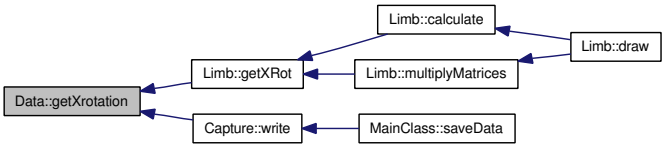

#### B.9.3.7 EulerRotation Data::getRotation (int *index*)

Returns the euler angles of a single sensor inside of a EulerRotation struct. index is the internal position in the cache. Use getIndexOfNode() to get Index.

#### B.9.3.8 float Data::getXrotation (int *index*)

Returns the x rotation of the sensor with the given number.

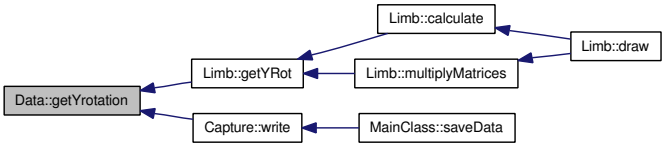
Here is the caller graph for this function:




#### B.9.3.9 float Data::getYrotation (int *index*)

Returns the y rotation of the sensor with the given number.

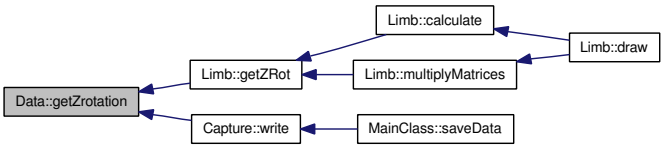
Here is the caller graph for this function:



#### B.9.3.10 float Data::getZrotation (int *index*)

Returns the z rotation of the sensor with the given number.

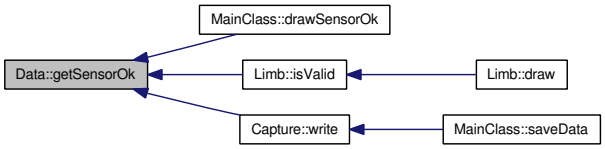
Here is the caller graph for this function:




#### B.9.3.11 bool Data::getSensorOk (int *index*)

Returns the boolean flag if the sensor with the given number is in ok state.

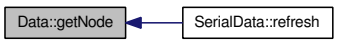
Here is the caller graph for this function:




#### B.9.3.12 int Data::getNode (int *index*)

Returns the ID number of a sensor node at a given internal index position.

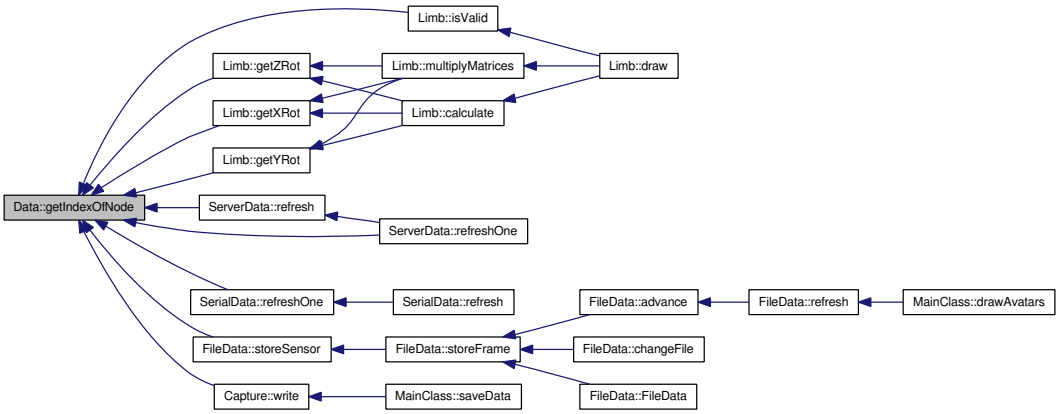
Here is the caller graph for this function:




#### B.9.3.13 int Data::getIndexOfNode (int *node*)

Returns the internal index position of a given ID number of a sensor node.

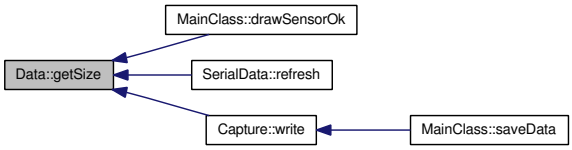
Here is the caller graph for this function:




### B.9.3.14 int Data::getSize ()

Returns the number of internal cached sensors.

Here is the caller graph for this function:



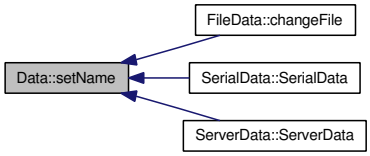

### B.9.3.15 string Data::getName ()

Returns the name of the DataSource. This name can be used for textual output or anything.

### B.9.3.16 void Data::setName (string *newName*)

Sets the name of the DataSource. This name can be used for textual output or anything. Feel free to set it to any value.

Here is the caller graph for this function:




The documentation for this class was generated from the following files:

- Data.h
- Data.cpp

## B.10 Defines Class Reference

`#include <Defines.h>`

### Public Member Functions

- Defines ()
- void **echo** (int d1)
- void echo (int d1, int d2)
- void echo (int d1, int d2, int d3)
- void echo (vector< int > d)
- void echoBr (int d1)
- void echoBr (int d1, int d2)
- void echoBr (int d1, int d2, int d3)
- void echoBr (vector< int > d)

### Static Public Member Functions

- static Defines * **getInstance** ()
- static float abs (float)
- static string float2string (float)
- static string int2string (int)
- static string bool2string (bool)
- static string str2UpperCase (string)
- static bool has (vector< int >, int)
- static vector< int > sort (vector< int >)

### Private Attributes

- int **echoMax**

### B.10.1 Detailed Description

There are just some static helper methods in this class. So there's basically no point in instancing an object of this class.

Next to the simple method this class holds all the macro-definitions used in the code.

Author: Lars;

### B.10.2 Constructor & Destructor Documentation

#### B.10.2.1 Defines::Defines ()

Constructor The class can be used with or without constructor. Both is possible. The echo fetures are only available on a valid instance. This can be done via getInstance or the constructor.

### B.10.3 Member Function Documentation

#### B.10.3.1 float Defines::abs (float *f*) [static]

Static helper method. Returns the absoulute value of the given float value.

#### B.10.3.2 string Defines::float2string (float *f*) [static]

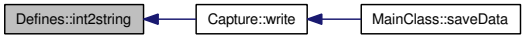
Static helper method. Returns a string containing a textual representation of the given float variable.

Here is the caller graph for this function:


#### B.10.3.3 string Defines::int2string (int *i*) [static]

Static helper method. Returns a string containing a textual representation of the given integer variable.

Here is the caller graph for this function:




#### B.10.3.4 string Defines::bool2string (bool *b*) [static]

Static helper method. Returns a string containing a textual representation of the given bool variable.

Here is the caller graph for this function:


#### B.10.3.5 string Defines::str2UpperCase (string *str*) [static]

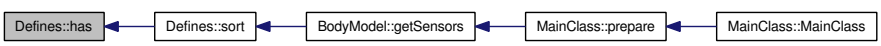
Static helper method. Returns the given string in upper case.

Here is the caller graph for this function:


#### B.10.3.6 bool Defines::has (vector< int > *in*, int *el*) [static]

Static helper method. Returns true if a given integer vector already has the given element.

Here is the caller graph for this function:

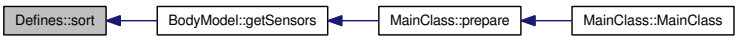


#### B.10.3.7 vector< int > Defines::sort (vector< int > *in*) [static]

Static helper method. Returns the given integer vector sorted in increasing order.

Here is the call graph for this function:


Here is the caller graph for this function:




#### B.10.3.8 void Defines::echo (int *d1*, int *d2*)

Helper method. Echos two integers. The method keeps the length of the largest number yet. If shorter numbers appear, spaces get inserted. Therefore the numbers appear aligned.

#### B.10.3.9 void Defines::echo (int *d1*, int *d2*, int *d3*)

Helper method. Echos three integers. The method keeps the length of the largest number yet. If shorter numbers appear, spaces get inserted. Therefore the numbers appear aligned.

#### B.10.3.10 void Defines::echo (vector< int > *d*)

Helper method. Echos a vector of integers. The method keeps the length of the largest number yet. If shorter numbers appear, spaces get inserted. Therefore the numbers appear aligned.

#### B.10.3.11 void Defines::echoBr (int *d1*)

Helper method. Echos an integer. After the number a line feed gets inserted. The method keeps the length of the largest number yet. If shorter numbers appear, spaces

get inserted. Therefore the numbers appear aligned.

#### B.10.3.12 void Defines::echoBr (int *d1*, int *d2*)

Helper method. Echos two integers. After the last number a line feed gets inserted. The method keeps the length of the largest number yet. If shorter numbers appear, spaces get inserted. Therefore the numbers appear aligned.

#### B.10.3.13 void Defines::echoBr (int *d1*, int *d2*, int *d3*)

Helper method. Echos three integers. After the last number a line feed gets inserted. The method keeps the length of the largest number yet. If shorter numbers appear, spaces get inserted. Therefore the numbers appear aligned.

#### B.10.3.14 void Defines::echoBr (vector< int > *d*)

Helper method. Echos a vector of integers. After the last number a line feed gets inserted. The method keeps the length of the largest number yet. If shorter numbers appear, spaces get inserted. Therefore the numbers appear aligned.

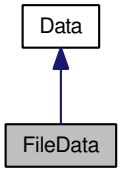
The documentation for this class was generated from the following files:

- Defines.h
- Defines.cpp

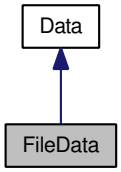
## B.11 FileData Class Reference

`#include <FileData.h>`

Inheritance diagram for FileData:




Collaboration diagram for FileData:




### Public Member Functions

- FileData (string)
- FileData (int)
- virtual ~FileData ()
- void changeFile (int)
- void changeFile (string)
- void refresh ()

### Private Member Functions

- string readLine ()
- string readLine (string *)
- string readFrame ()
- XmlItem getItem (string *)

- void storeFrame (string)
- void storeSensor (string ∗)
- void advance ()

## Private Attributes

- ifstream **inFile**
- string **fname**

### B.11.1 Detailed Description

This class can be used in the same manner as SerialData. Therefore it offers former sensor data from storage. This class is only to read the data. For writing data to disk use the Capture class.

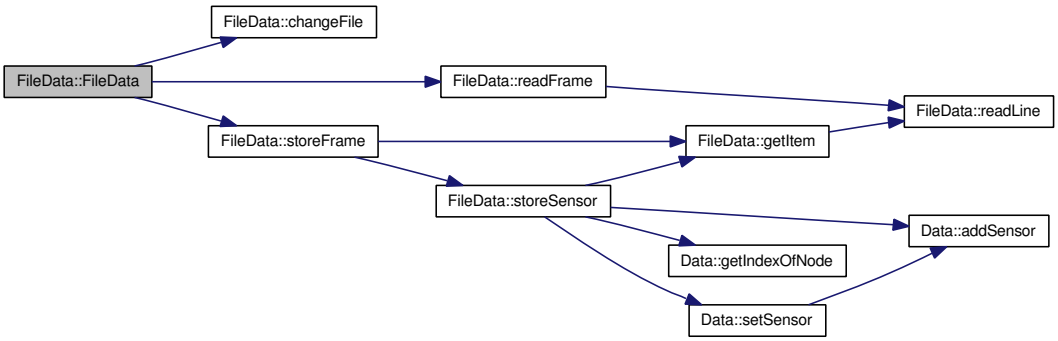
Author: Lars;

### B.11.2 Constructor & Destructor Documentation

#### B.11.2.1 FileData::FileData (string *fn*)

Constructor: The file to read from has to be handed over. The data file can later be changed without creating a new instance of this class.

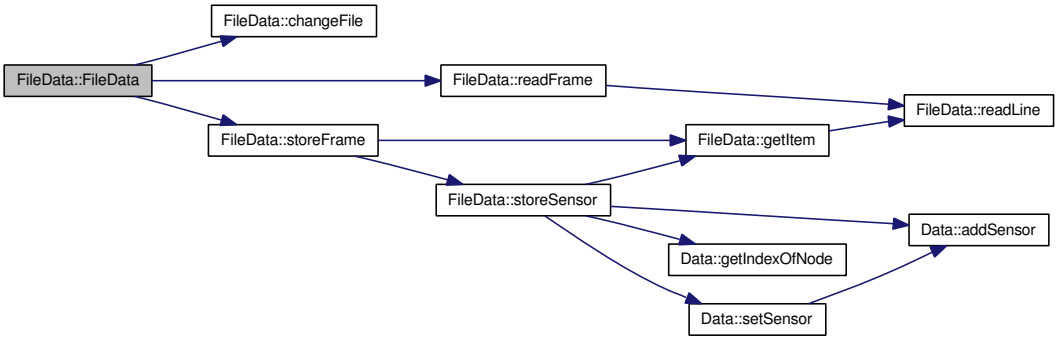
Here is the call graph for this function:



#### B.11.2.2 FileData::FileData (int *fnum*)

Constructor: Instead of a filename a number has to be given. The number referrs to one of the default files. 1 means capture01.txt and so on. capture09.txt is the last file.

Here is the call graph for this function:




#### B.11.2.3 FileData::~FileData () [virtual]

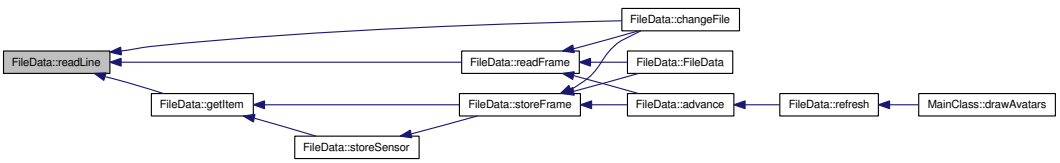
Destructor: Just closes the current file.

### B.11.3 Member Function Documentation

#### B.11.3.1 string FileData::readLine () [private]

Private method reading one line from the data file. If the end of the file is reached, it restarts at the beginning of the file.

Here is the caller graph for this function:



### B.11.3.2 string FileData::readLine (string * *str*) [private]

Private method extracting the next statement out of the given string. This method was needed, because the former line feeds got lost. The method takes three possible signs for the end of a statement.

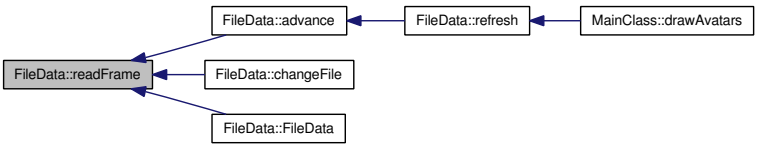
- Line feed character '
  ';
- '>' as the character at the end of a command;
- '<' as the character at the beginning of the next command; The extracted command gets removed of the given string!

### B.11.3.3 string FileData::readFrame () [private]

Private method: Read a whole dataset (called frame) from the file.

Here is the call graph for this function:


Here is the caller graph for this function:

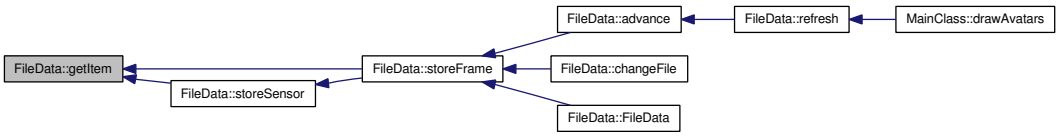



### B.11.3.4 XmlItem FileData::getItem (string * *str*) [private]

This private method extracts a whole XML-DataItem from the given string. Returns a struct defined in the header-file. It just contains two strings. One for the tag (name). One for the data.

Here is the call graph for this function:

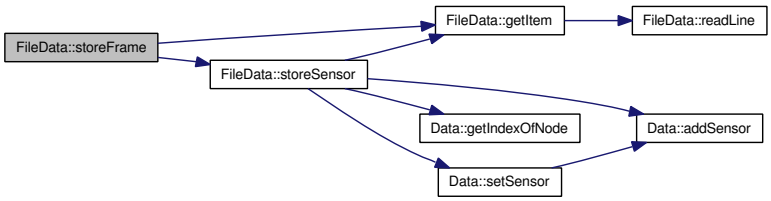

Here is the caller graph for this function:



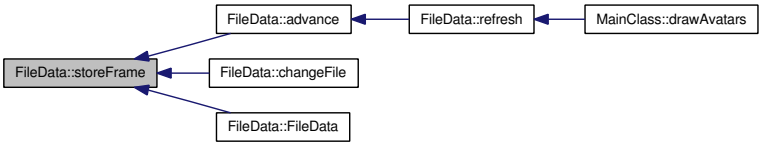

### B.11.3.5 void FileData::storeFrame (string *frame*) [private]

This private method takes the string containing the data for a whole frame. This means there is no more a opening and closing frame-tag in the given string. It then extracts the data and stores it to the class fields.

Here is the call graph for this function:



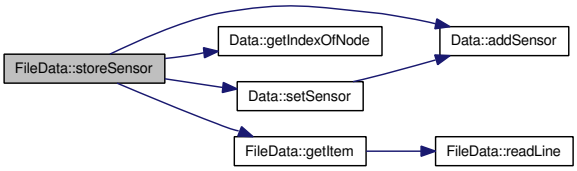

Here is the caller graph for this function:



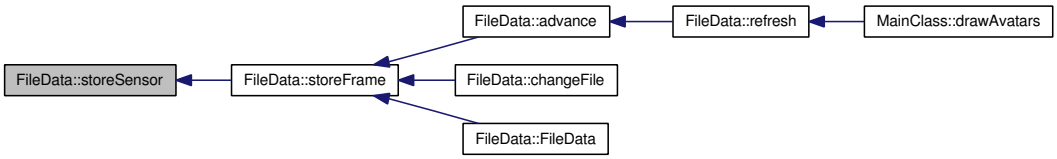
### B.11.3.6 void FileData::storeSensor (string * *str*) [private]

This private method takes the string containing the data for just one sensor. This means there is no more a opening and closing sensor-tag in the given string. It then extracts the data and stores it to the class fields.

Here is the call graph for this function:



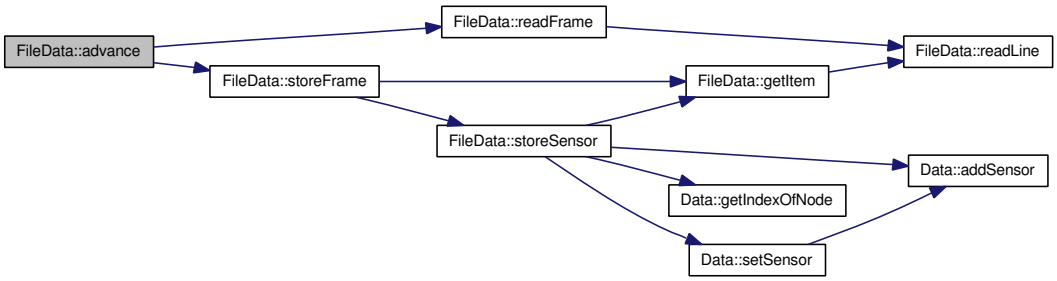

Here is the caller graph for this function:



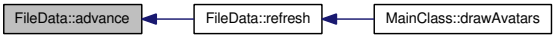

### B.11.3.7 void FileData::advance () [private]

Read the next frame (data set) from the data file and store the data in the class fields.

Here is the call graph for this function:



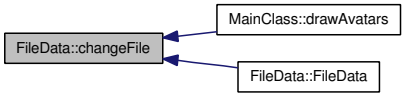
Here is the caller graph for this function:




### B.11.3.8 void FileData::changeFile (int *fnum*)

Changes the current file from where the data is read. This is the version of changeFile which takes an integer number as parameter. The number referrs to one of the default input files. 1 means capture01.txt and so on. capture09.txt is the last file.

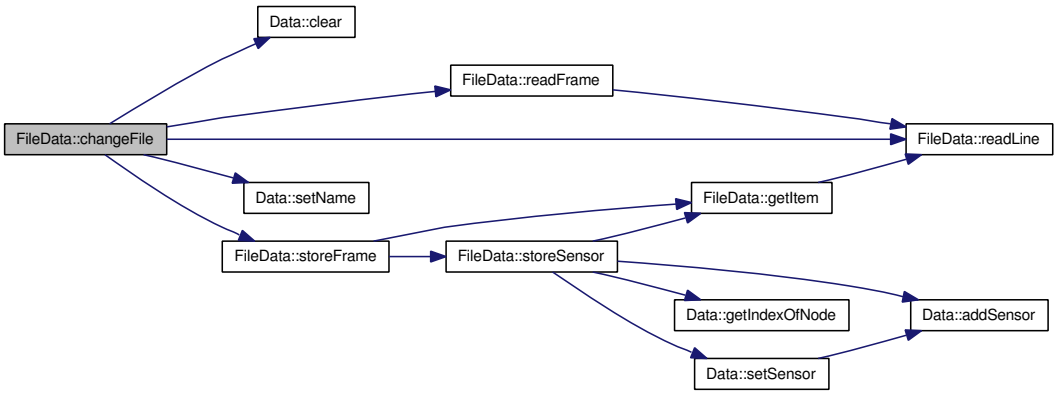
Here is the caller graph for this function:




### B.11.3.9 void FileData::changeFile (string *fn*)

Changes the current file from where the data is read. Just the new filename hast to be handed over. e.g. capture07.txt

Here is the call graph for this function:

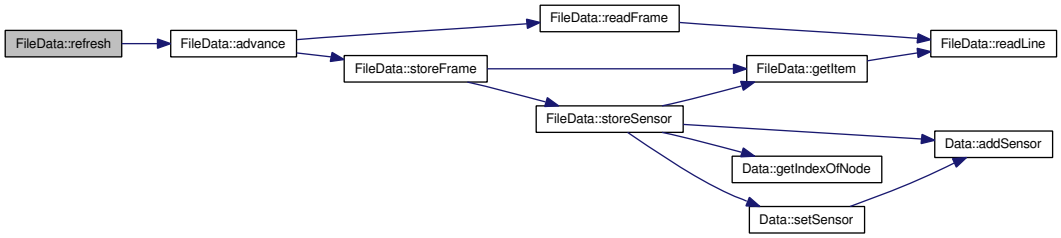


#### B.11.3.10 void FileData::refresh ()

Just a synonym for advance. It reads a new data set from the file and stores it in the class fields.

Reimplemented from Data.

Here is the call graph for this function:




Here is the caller graph for this function:


The documentation for this class was generated from the following files:

- FileData.h
- FileData.cpp

## B.12 HandleServer Class Reference

`#include <HandleServer.h>`

### Public Member Functions

- HandleServer ()
- HandleServer (bool isClient)
- HandleServer (int servTCPPort, string servAddress)
- HandleServer (bool isClient, int argc, _TCHAR *argv[ ])
- HandleServer (unsigned short tcpPort, string servAddr, string streamName, string streamPassword)
- HandleServer (unsigned short udpPort, unsigned short tcpPort, string servAddr, string streamName, string subscriberName, string streamPassword)
- HandleServer (unsigned short tcpPort, string servAddr, string streamName, string streamPassword, int argc, _TCHAR *argv[ ])
- HandleServer (unsigned short udpPort, unsigned short tcpPort, string servAddr, string streamName, string subscriberName, string streamPassword, int argc, _TCHAR *argv[ ])
- ~HandleServer ()
- void registerAtServer ()
- void deregisterAtServer ()
- bool getIsClient ()
- string getServAddress ()
- unsigned short getServTCPPort ()
- unsigned short getServUDPPort ()
- string getSubscriberName ()
- string getStreamName ()
- string getStreamPassword ()
- void sendUDP (string str)
- string receiveUDP ()
- string receiveUDPsave ()

- vector< string > list ()
- string getLast ()
- void pushLast (string)
- int getLastQueueSize ()
- bool lastQueueEmpty ()
- void addFirst (string)
- void clearUdpQueue ()
- bool getTimeToLeave ()

## Static Public Member Functions

- static int getBufsize ()
- static void wait ()
- static unsigned int decodeByte (string code)
- static string **encodeByte** (unsigned int num)
- static int decodeWord (string code)
- static string encodeWord (int num)
- static int decode (unsigned bit, string code)
- static string encode (unsigned bit, int num)

## Private Member Functions

- void initialize (bool isClient)
- void openTCP ()
- void closeTCP ()
- void checkParam (int argc, _TCHAR *argv[ ])
- void readParam (int argc, _TCHAR *argv[ ])
- void registerAsPublisher ()
- void registerAsClient ()
- void handleParam (int argc, _TCHAR *argv[ ])
- void startReregisterThread ()
- char * getLn (char *buffer)

- string getLn ()
- void sendLn (const char *inbuf)
- void sendLn (const int i)
- void sendLn (const string s)
- void openUDP ()
- void closeUDP ()
- void startUp ()
- void unsubscribe ()
- void deregister ()
- int udpPort ()
- void startGetLastThread ()

## Private Attributes

- bool **isClient**
- bool **noConnection**
- bool **lastThreadStarted**
- bool **timeToLeave**
- vector< string > **lastUdpQueue**
- pthread_t **reregisterThread**
- pthread_t **lastThread**
- pthread_mutex_t **mutex**
- string **servAddress**
- unsigned short **servTCPPort**
- unsigned short **servUDPPort**
- string **streamName**
- string **subscriberName**
- string **streamPassword**
- TCPSocket * **tcpSocket**
- UDPSocket * **udpSocket**

## Static Private Attributes

- static const int **BUFSIZE** = 255

## B.12.1 Detailed Description

C++ class to handle the sensor network server of nico ranieri. Because of the wide range of usages this class offers a large amount of methods.

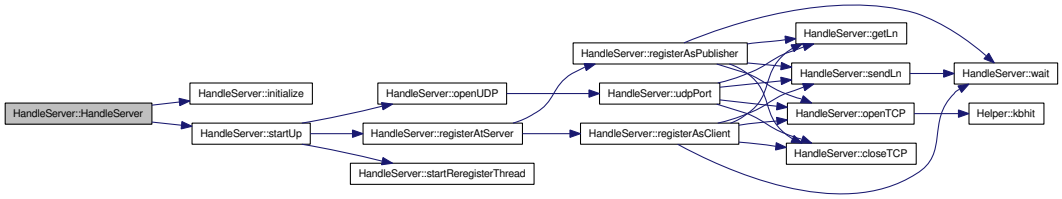
Author: Lars Widmer, www.lawi.ch

## B.12.2 Constructor & Destructor Documentation

### B.12.2.1 HandleServer::HandleServer ()

Default constructor: Always creates a subscriber object with default settings.

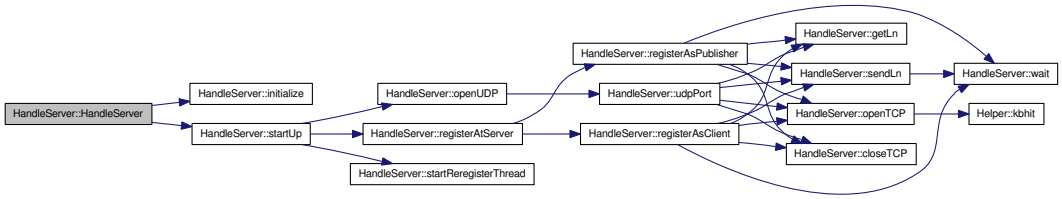
Here is the call graph for this function:




### B.12.2.2 HandleServer::HandleServer (bool *isClient*)

Constructor: Creates a subscriber object if the parameter is true else a publisher is created. For the connection properties default settings are used.

Here is the call graph for this function:




### B.12.2.3 HandleServer::HandleServer (int *servTCPPort*, string *servAddress*)

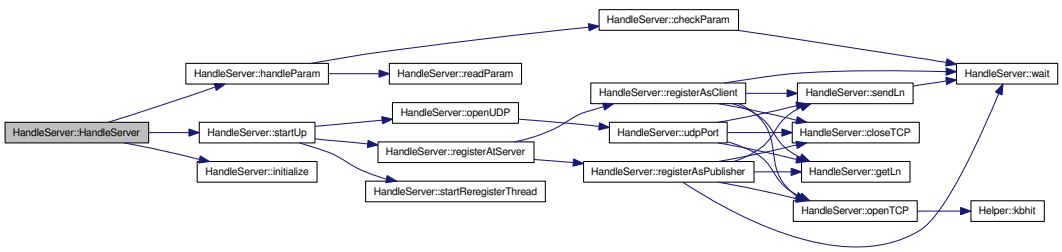
The only constructor which does not register or subscribe at the server. Use this constructor for queries in order to get the list of available streams.

Here is the call graph for this function:


### B.12.2.4 HandleServer::HandleServer (bool *isClient*, int *argc*, _TCHAR * *argv*[ ])

Constructor: Creates a subscriber object if the first parameter is true else a publisher is created. For setting the connection properties the given command line parameters are used.

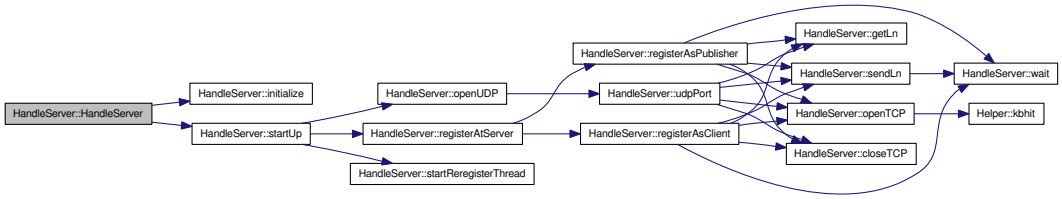
Here is the call graph for this function:



#### B.12.2.5 HandleServer::HandleServer (unsigned short *tcpPort*, string *servAddr*, string *streamName*, string *streamPassword*)

For creating a publisher object. At your option the password can be left blank as well.

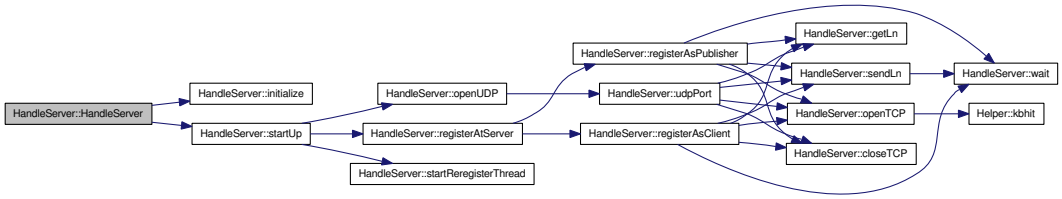
Here is the call graph for this function:




#### B.12.2.6 HandleServer::HandleServer (unsigned short *udpPort*, unsigned short *tcpPort*, string *servAddr*, string *streamName*, string *subscriberName*, string *streamPassword*)

For creating a subscriber object. At your option the password can be left blank as well.

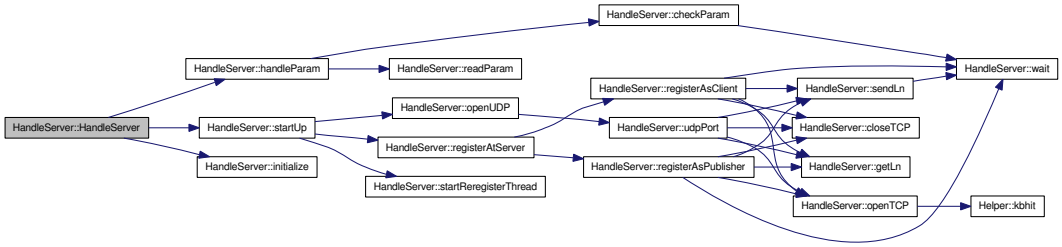
Here is the call graph for this function:




#### B.12.2.7 HandleServer::HandleServer (unsigned short *tcpPort*, string *servAddr*, string *streamName*, string *streamPassword*, int *argc*, _TCHAR * *argv*[ ])

For creating a publisher object. At your option the password can be left blank as well. The given values are used as defaults. But if given, the parameters are evaluated.

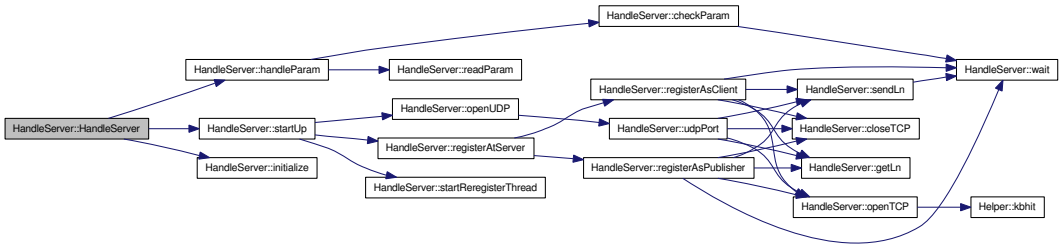
Here is the call graph for this function:




### B.12.2.8 HandleServer::HandleServer (unsigned short *udpPort*, unsigned short *tcpPort*, string *servAddr*, string *streamName*, string *subscriberName*, string *streamPassword*, int *argc*, _TCHAR ∗ *argv*[ ])

For creating a subscriber object. At your option the password can be left blank as well. The given values are used as defaults. But if given, the parameters are evaluated.

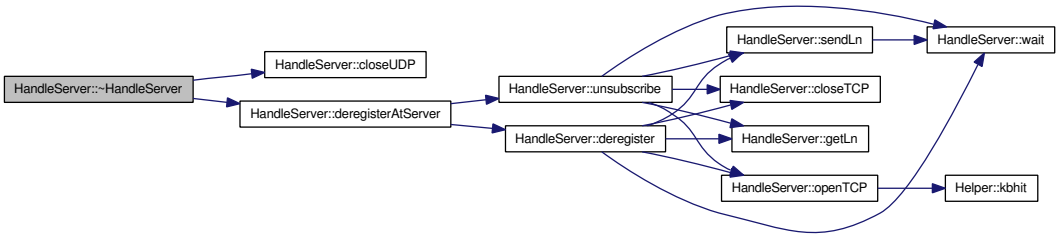
Here is the call graph for this function:




### B.12.2.9 HandleServer::∼HandleServer ()

Deconstructor: Stops the threads and waits for their termination. Deregisters at the server and closes the connections.

Here is the call graph for this function:

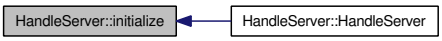



## B.12.3 Member Function Documentation

### B.12.3.1 void HandleServer::initialize (bool *isClient*) [private]

Sets all object fields to default values. The default values are set by the define statements in this cpp-file.

Here is the caller graph for this function:

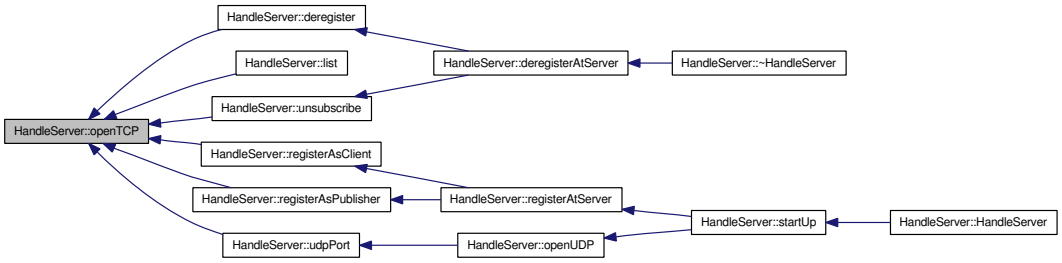



### B.12.3.2 void HandleServer::openTCP () [private]

Low level method to open the TCP socket. If there occurs an error there will be 100 retries until the method gives up and terminates the software. Thats because sometimes there are two or three connection attempts needed. Using this implementation publishers are running stable for days.

Here is the call graph for this function:

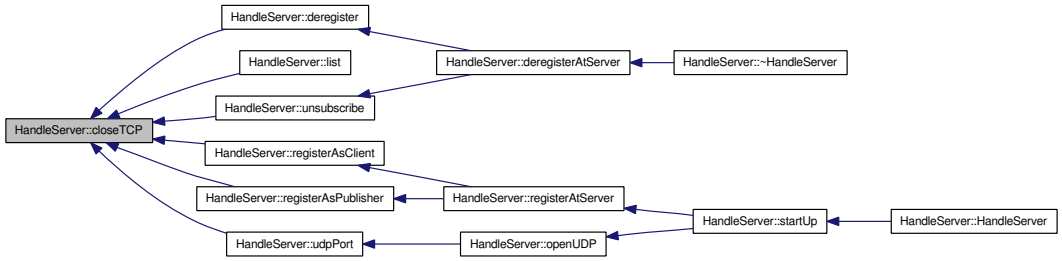
Here is the caller graph for this function:




### B.12.3.3 void HandleServer::closeTCP () [private]

Low level method to close the TCP socket.

Here is the caller graph for this function:




### B.12.3.4 void HandleServer::checkParam (int *argc*, _TCHAR * *argv*[ ]) [private]

Checks if a meaningful number of command line parameters is passed. Zero parameters tell to stick to the default values.

Here is the call graph for this function:

Here is the caller graph for this function:


#### B.12.3.5 void HandleServer::readParam (int *argc*, _TCHAR * *argv*[ ]) [private]

Parses the given parameters and stores the values in the object fields. For a subscriber 6 or 7 parameters are needed:

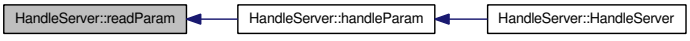
- server address
- TCP port
- UDP port
- stream name
- subscriber name
- stream password (optional)

For a publisher 4 or 5 parameters are needed:

- server address
- TCP port
- stream name
- stream password (optional)

The UDP port for a publisher is chosen by the server.

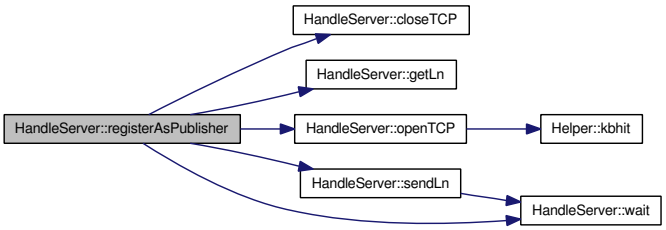
Here is the caller graph for this function:



### B.12.3.6 void HandleServer::registerAsPublisher () [private]

Lower level method to register or reregister at the server as a publisher.

Here is the call graph for this function:

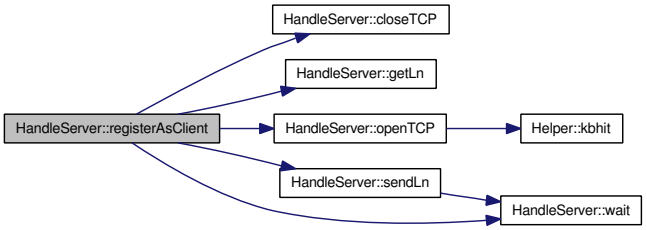



Here is the caller graph for this function:


### B.12.3.7 void HandleServer::registerAsClient () [private]

Lower level method to register or reregister at the server as a subscriber.

Here is the call graph for this function:




Here is the caller graph for this function:

#### B.12.3.8 void HandleServer::handleParam (int *argc*, _TCHAR * *argv*[ ]) [private]

Method to handle the command line parameters. Internally this method just calls checkParam and readParam.

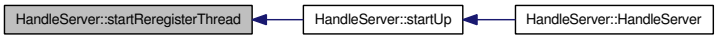
Here is the call graph for this function:


Here is the caller graph for this function:


#### B.12.3.9 void HandleServer::startReregisterThread () [private]

Method to start the thread for automatic reregistering at the server. The constructors indirectly call this method.

Here is the caller graph for this function:




#### B.12.3.10 char * HandleServer::getLn (char * *buffer*) [private]

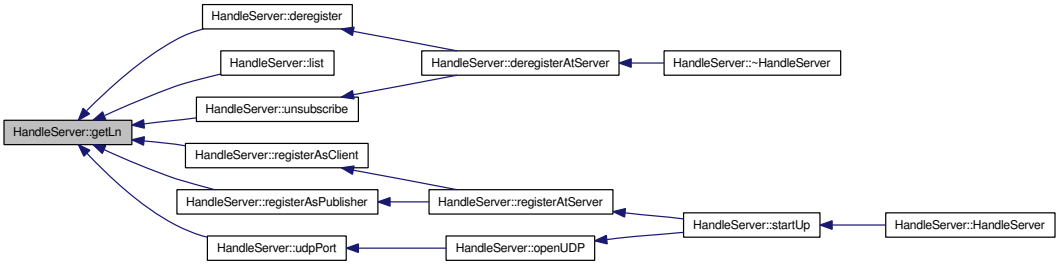
Low level method to receive a text line from the TCP port.

Here is the call graph for this function:

### B.12.3.11 string HandleServer::getLn () [private]

Method to receive a text line from the TCP socket. Returns a string.

Here is the caller graph for this function:

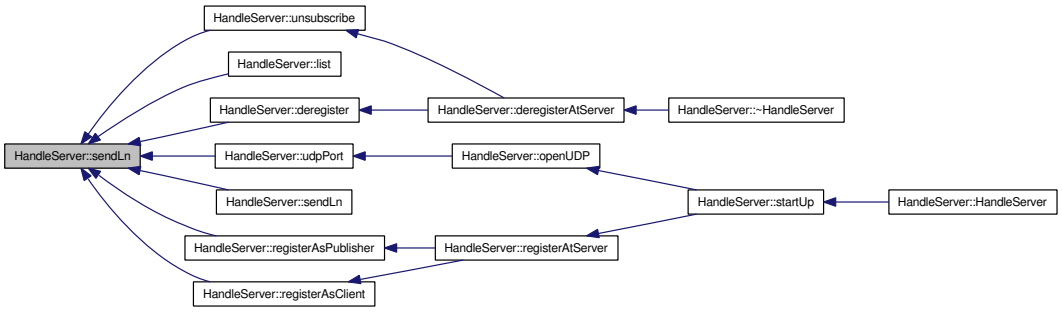



### B.12.3.12 void HandleServer::sendLn (const char * *inbuf*) [private]

Low level method to send a text line over the TCP socket.

Here is the call graph for this function:

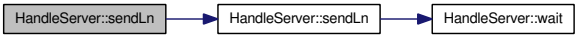

Here is the caller graph for this function:




### B.12.3.13 void HandleServer::sendLn (const int *i*) [private]

Method to send an integer number over the TCP socket.

Here is the call graph for this function:




#### B.12.3.14 void HandleServer::sendLn (const string *s*) [private]

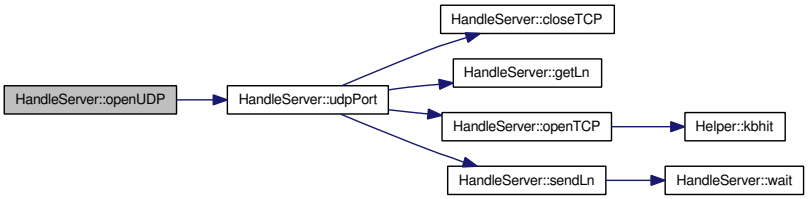
Method to send a string over the TCP socket.

Here is the call graph for this function:


#### B.12.3.15 void HandleServer::openUDP () [private]

Opens the UDP connection to the socket in order to transfer data. This method gets used for publishers and subscribers. For a publisher the port number on which the server accepts packages gets obtained automatically.

Here is the call graph for this function:

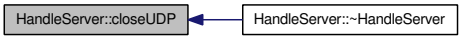



Here is the caller graph for this function:

#### B.12.3.16 void HandleServer::closeUDP () [private]

Closes the UDP connection to the server. This method gets used for publishers and subscribers.

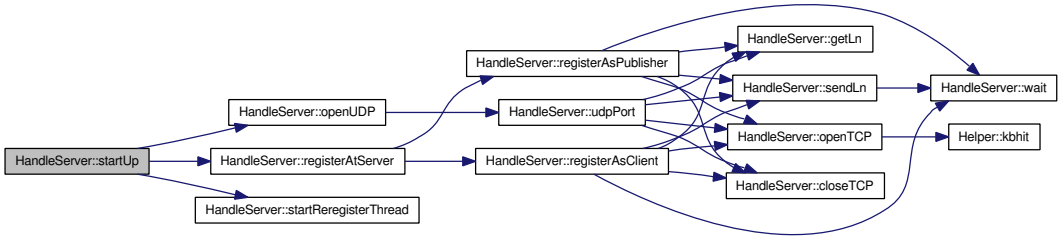
Here is the caller graph for this function:




#### B.12.3.17 void HandleServer::startUp () [private]

Method to prepare the connections for data transfer. The constructors call this method.

Here is the call graph for this function:

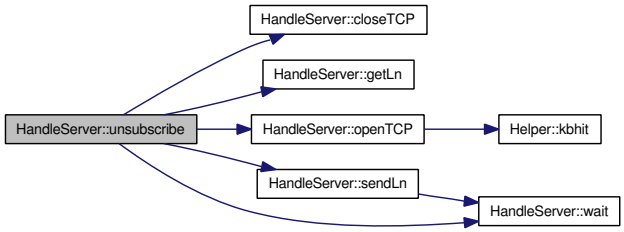



Here is the caller graph for this function:

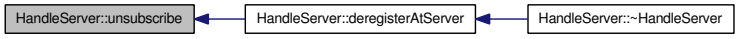

#### B.12.3.18 void HandleServer::unsubscribe () [private]

Method to unsubscribe from the server. This means the server won't send any UDP data to this subscriber afterwards. Use deregisterAtServer as a general purpose method.

Here is the call graph for this function:



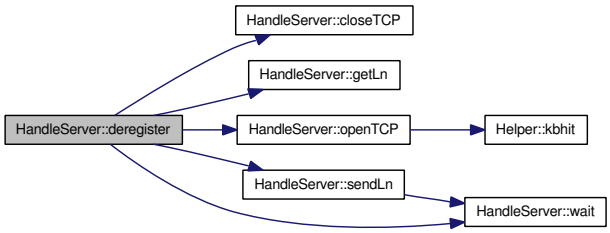

Here is the caller graph for this function:



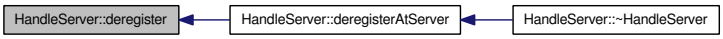

#### B.12.3.19 void HandleServer::deregister () [private]

Method to deregister a publisher from the server. Afterwards the server doesn't accept anymore UDP data packets from this publisher. Use deregisterAtServer as a general purpose method.

Here is the call graph for this function:



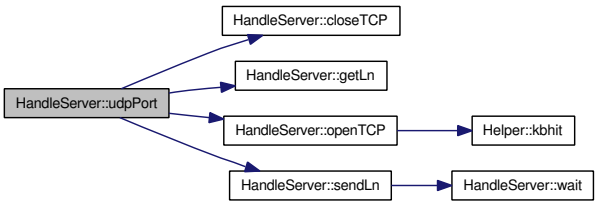

Here is the caller graph for this function:



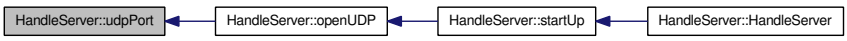
#### B.12.3.20 int HandleServer::udpPort () [private]

Method to ask the server for the current udp port where a publisher has to send its data. openUDP uses this method.

Here is the call graph for this function:



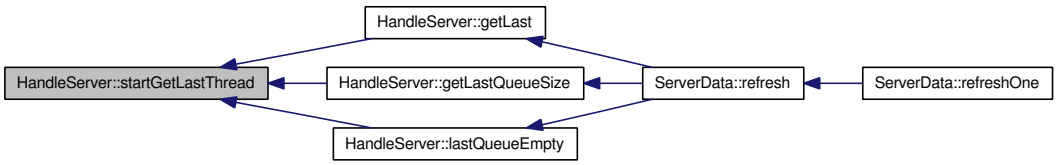

Here is the caller graph for this function:




#### B.12.3.21 void HandleServer::startGetLastThread () [private]

Method to start the thread for reading from the UDP socket in the background. Using this getLast offers a non blocking receive. This method gets called when getLast is used the first time. Afterwards the usual receiveUDP methods can't be used anymore. Because there isn't anything left to receive when the thread is running.

Here is the caller graph for this function:

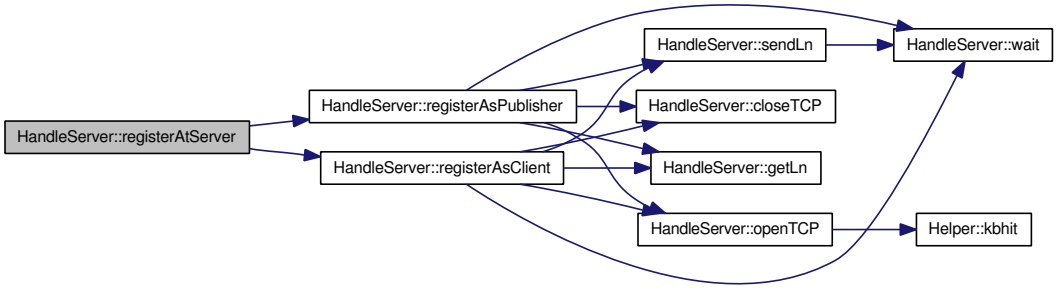



#### B.12.3.22 void HandleServer::registerAtServer ()

Higher level method to (re-)register at the server. This is usually done by the constructor and automatically in the background every 3mins. The appropriate register

methods are called based on the field settings if the object is a publisher or subscriber.

Here is the call graph for this function:

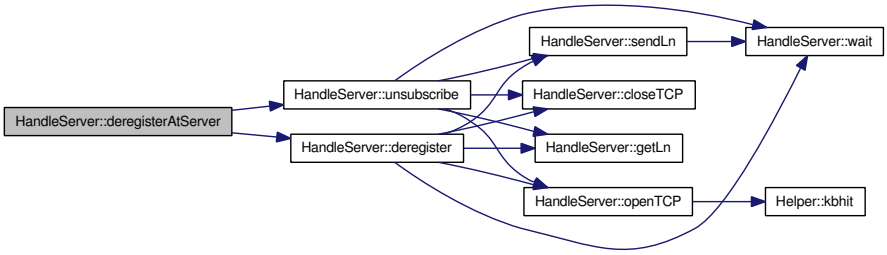



Here is the caller graph for this function:


### B.12.3.23 void HandleServer::deregisterAtServer ()

Method to deregister a publisher or a subscriber from the server.

Here is the call graph for this function:




Here is the caller graph for this function:

#### B.12.3.24 int HandleServer::getBufsize () [static]

Returns the size of the used char buffers.

#### B.12.3.25 bool HandleServer::getIsClient ()

Returns true if the current object is a subscriber and and false it the current object is a publisher.

#### B.12.3.26 string HandleServer::getServAddress ()

Returns the current server address.

#### B.12.3.27 unsigned short HandleServer::getServTCPPort ()

Returns the current TCP port.

#### B.12.3.28 unsigned short HandleServer::getServUDPPort ()

Returns the current UDP port.

#### B.12.3.29 string HandleServer::getSubscriberName ()

Returns the current subscriber name.

#### B.12.3.30 string HandleServer::getStreamName ()

Returns the current stream name.

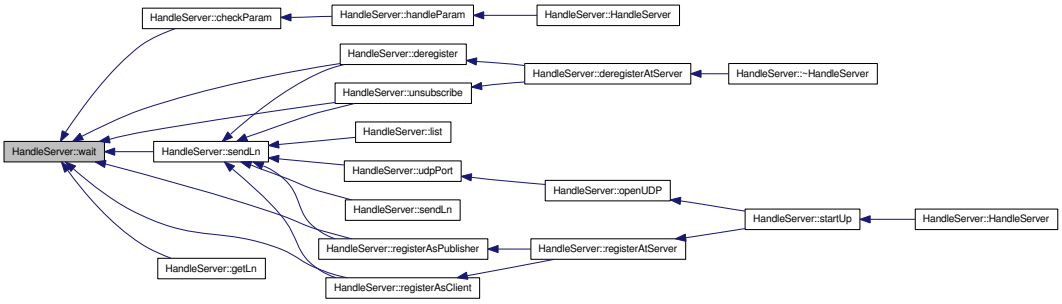
#### B.12.3.31 string HandleServer::getStreamPassword ()

Returns the current stream password.

#### B.12.3.32 void HandleServer::wait () [static]

Method to blocks the program and wait for enter. The Message "Hit enter to terminate program ..." gets displayed. But this method itself doesn't terminate the program.

Here is the caller graph for this function:



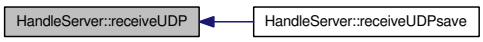

#### B.12.3.33 void HandleServer::sendUDP (string *str*)

Send string data over the UDP socket. This method internally adds the headers as demanded by the server protocol.

#### B.12.3.34 string HandleServer::receiveUDP ()

Blocks until the next packet is received from UDP. Then the packet is returned. ATTENTION: If you call this method after calling getLast you will always get blocked! GetLast at its first run starts a thread to read from the UDP socket. This thread automatically reads every available data from the socket. Therefore there's nothing left to read with this method.

Here is the caller graph for this function:



#### B.12.3.35 string HandleServer::receiveUDPsave ()

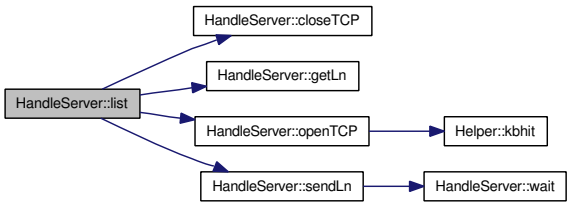
Blocks until the next packet is received from UDP. Then the packet is returned. Use this method from outside to be sure not to get blocked by getLast. TODO: Implement two different flags to stop the threads independently.

Here is the call graph for this function:


#### B.12.3.36 vector< string > HandleServer::list ()

Method to query the server for the available streams. Returns a vector of strings.

Here is the call graph for this function:




#### B.12.3.37 unsigned HandleServer::decodeByte (string *code*) [static]

Helper method to decode a byte for sending as a text. In the input string only the characters from A to P can be used. Therefore one byte is stored in two characters.

Here is the call graph for this function:


Here is the caller graph for this function:

#### B.12.3.38 int HandleServer::decodeWord (string *code*) [static]

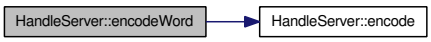
Helper method to decode a word after receiving it as a text. We assume 16bit values. This is what the standard says about the minimum size of int. Therefore the function exits if there accur larger values. This function returns values from -32768 to 32767.

Here is the call graph for this function:


#### B.12.3.39 string HandleServer::encodeWord (int *num*) [static]

Helper method to encode a word for sending as a text. We assume 16bit values. This is what the standard says about the minimum size of int. Therefore the function exits if there accur larger values. This function handles values from -32768 to 32767.

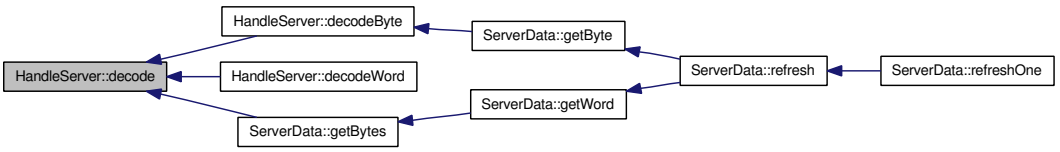
Here is the call graph for this function:




#### B.12.3.40 int HandleServer::decode (unsigned *bit*, string *code*) [static]

Decodes a given string of a given number of bytes. Every character gets decoded into 4 bits. Therefore the given number of bits has to be a multiple of 4.

Here is the caller graph for this function:



#### B.12.3.41 string HandleServer::encode (unsigned *bit*, int *num*) [static]

Encodes a given value of a given number of bytes. Every 4 bits are encoded in one character. Therefore the given number of bits has to be a multiple of 4.

Here is the caller graph for this function:


#### B.12.3.42 string HandleServer::getLast ()

Method for getting the last packet received from the UDP socket. Every packet gets returned only once. This method always returns immediatly. Therefore it implements a non blocking receive. To achieve this a thread is running in the background and receiving every available packet from the UDP socket. Received packets are written to a fifo buffer. If the buffer gets to large an error is issued. If there aren't any packets in the buffer getLast returns an empty string. Race conditions between getLast and the receiver-thread are controlled using posix mutex.

Here is the call graph for this function:


Here is the caller graph for this function:


#### B.12.3.43 void HandleServer::pushLast (string *last*)

Method to push a new value at the end of the Queue of the received packets for getLast. If this method would be used in the getLastThread we would obtain a stack behaviour instead of fifo. For some applications this might be appropriate.

#### B.12.3.44 int HandleServer::getLastQueueSize ()

Method to get the size of the Queue of the received packets for getLast.

Here is the call graph for this function:


Here is the caller graph for this function:


#### B.12.3.45 bool HandleServer::lastQueueEmpty ()

Boolean method to get if the size of the Queue of the received packets for getLast is zero.

Here is the call graph for this function:


Here is the caller graph for this function:

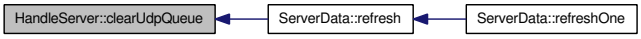

#### B.12.3.46 void HandleServer::addFirst (string *first*)

Method to add a new value at the beginning of the Queue of the received packets for getLast.

#### B.12.3.47 void HandleServer::clearUdpQueue ()

Method to clear the queue of received UDP packets for getLast. Use this method to avoid an overflow of the queue.

Here is the caller graph for this function:




#### B.12.3.48 bool HandleServer::getTimeToLeave ()

Method to return the timeToLeave-flag. This flag get's set by the destructor in order to terminate the two threads.

The documentation for this class was generated from the following files:

- HandleServer.h
- HandleServer.cpp

## B.13 HandleServerSingletonKeeper Class Reference

`#include <HandleServerSingletonKeeper.h>`

### Static Public Member Functions

- static HandleServer * getInstance ()

### B.13.1 Detailed Description

Singleton class limiting to one instance of HandleServer. Therefore the getInstance method always returns the pointer to the same instance of HandleServer.

Author: Lars Widmer, www.lawi.ch

### B.13.2 Member Function Documentation

#### B.13.2.1 HandleServer * HandleServerSingletonKeeper::getInstance () [static]

Singleton constructor limiting to one instance of HandleServer. Therefore this method always returns the pointer to the same instance of HandleServer.

The documentation for this class was generated from the following files:

- HandleServerSingletonKeeper.h
- HandleServerSingletonKeeper.cpp

## B.14 Helper Class Reference

`#include <Helper.h>`

### Static Public Member Functions

- static int getch ()
- static int kbhit ()
- static void usleep (unsigned nanoseconds)

### B.14.1 Detailed Description

Class offering a keyboard lookahead monitor for Linux and Windows. The functions are similar to getch and kbhit of conio.h under Windows.

Author: Floyd L. Davidson <http://www.ptialaska.net/~floyd> Ukpeagvik (Barrow, Alaska) Adaptions: Lars Widmer, www.lawi.ch

### B.14.2 Member Function Documentation

#### B.14.2.1 int Helper::getch (void) [static]

getch() – a blocking single character input from stdin

Returns a character, or -1 if an input error occurs.

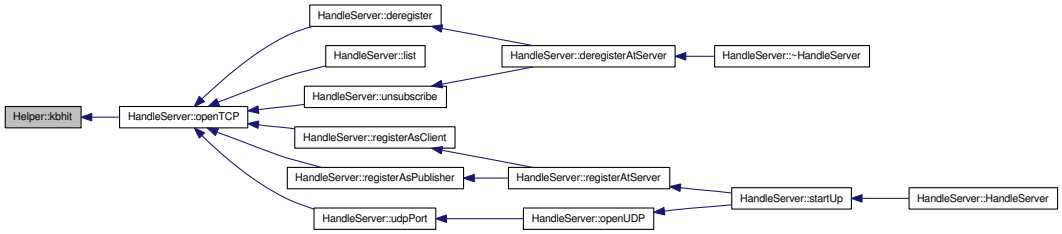
Conditionals allow compiling with or without echoing of the input characters, and with or without flushing pre-existing existing buffered input before blocking.

#### B.14.2.2 int Helper::kbhit (void) [static]

kbhit() – a keyboard lookahead monitor

returns the number of characters available to read.

Here is the caller graph for this function:




### B.14.2.3 void Helper::usleep (unsigned *nanoseconds*) [static]

Simple platform independent sleep function. TODO: produces a strange segmentation fault Author: Lars

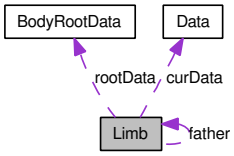
The documentation for this class was generated from the following files:

- Helper.h
- Helper.cpp

## B.15 Limb Class Reference

`#include <Limb.h>`

Collaboration diagram for Limb:




### Public Member Functions

- Limb ()
- Limb (Limb &l)
- Limb (Data *)
- Limb (Data *, BodyRootData *)
- void setData (Data *)
- Data * getData ()
- virtual void draw ()
- void setOrientation (int o)
- void setInheritCoordinates (int=2)
- void setSize (float, float, float)
- void setPos (float, float, float, float, float, float)
- void setX1 (float)
- void setY1 (float)
- void setZ1 (float)
- void setX2 (float)
- void setY2 (float)
- void setZ2 (float)
- void setFixedRot (float, float, float)
- float getX1 ()
- float getY1 ()

- float getZ1 ()
- float getX2 ()
- float getY2 ()
- float getZ2 ()
- float getNX1 ()
- float getNY1 ()
- float getNZ1 ()
- float getNX2 ()
- float getNY2 ()
- float getNZ2 ()
- float getMiddleX ()
- float getMiddleY ()
- float getMiddleZ ()
- float getXRot ()
- float getYRot ()
- float getZRot ()
- virtual void setRootData (BodyRootData *)
- BodyRootData * getRootData ()
- void setFather (Limb *)
- Limb * getFather ()
- void addChild (Limb *)
- string getName ()
- void setName (string)
- int getNumber ()
- void setNumber (int)
- void print ()
- void setBasisShiftX (float)
- void setBasisShiftY (float)
- void setBasisShiftZ (float)
- void setTexture (string)
- bool isValid ()

## Protected Member Functions

- void check ()
- virtual void initialize ()
- void drawSphereByMatrix (D3DVECTOR ∗, D3DVECTOR ∗, D3DVECTOR ∗)
- void matrixmultiply (float, float, float, float, float, float, float ∗, float ∗, float ∗)
- void drawSphere ()
- void drawFixed ()
- void multiplyMatrices ()
- void shiftValues ()
- void calculate ()

## Protected Attributes

- float **xSize**
- float **ySize**
- float **zSize**
- float **x1**
- float **y1**
- float **z1**
- float **x2**
- float **y2**
- float **z2**
- float **nx1**
- float **ny1**
- float **nz1**
- float **nx2**
- float **ny2**
- float **nz2**
- float **middleX**
- float **middleY**

- float **middleZ**
- float **basisShiftX**
- float **basisShiftY**
- float **basisShiftZ**
- float **fixedRotX**
- float **fixedRotY**
- float **fixedRotZ**
- int **orientation**
- D3DVECTOR **posV**
- D3DVECTOR **rotV**
- D3DVECTOR **scaV**
- BodyRootData * **rootData**
- Data * **curData**
- Limb * **father**
- vector< Limb * > **children**
- string **name**
- int **number**
- int **inheritCoord**
- string **texture**

### B.15.1 Detailed Description

This class is used to represent a limb. The limbs can have a father and chlidren. Therefore they form a structure. Any form of a body model can be build up by connecting limbs accordingly.

Author: Lars;

### B.15.2 Constructor & Destructor Documentation

#### B.15.2.1 Limb::Limb ()

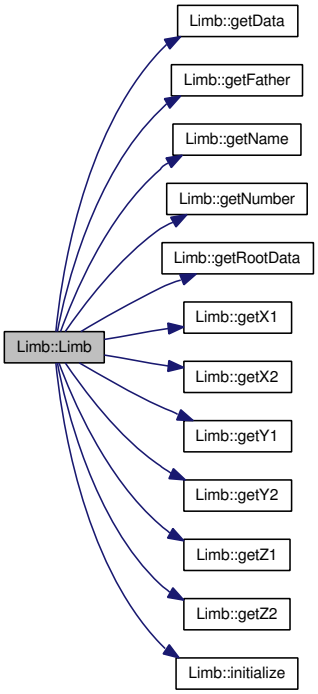
Default constructor. The pointers to the rootData and the current data get initialized with NULL.

Here is the call graph for this function:


### B.15.2.2 Limb::Limb (Limb & *l*)

Copy constructor. The pointers to the rootData and the current data get inherited.

Here is the call graph for this function:




### B.15.2.3 Limb::Limb (Data ∗ *d*)

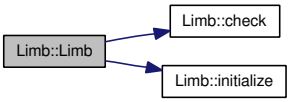
Constructor. The pointer to the current data gets initialized with the given value. The rootData is set to NULL.

Here is the call graph for this function:

#### B.15.2.4 Limb::Limb (Data * *d*, BodyRootData * *rd*)

Constructor. The pointers to the rootData and the current data get initialized with the given values.

Here is the call graph for this function:

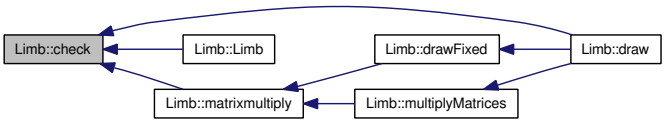



### B.15.3 Member Function Documentation

#### B.15.3.1 void Limb::check () [protected]

Private method checks if current data and root data are initialized (not NULL). If thats ok, this object can be used. If the check fails, the program is aborted.

Here is the caller graph for this function:




#### B.15.3.2 void Limb::initialize () [protected, virtual]

Private initialization method. It gets called by the constructors.

Here is the caller graph for this function:

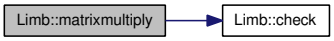
#### B.15.3.3 void Limb::drawSphereByMatrix (D3DVECTOR * *pV*, D3DVECTOR * *rV*, D3DVECTOR * *sV*) [protected]

Draws a sphere by the given matrix data.

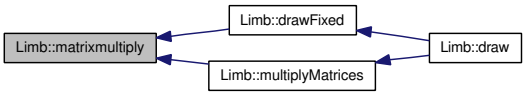
#### B.15.3.4 void Limb::matrixmultiply (float *x*, float *y*, float *z*, float *r1*, float *r2*, float *r3*, float * *a1*, float * *a2*, float * *a3*) [protected]

Rotates the point at the given coordinate. This method rotates first around the x, then y and finally around the z axis. The new coordinates are returned in the last three parameters.

Here is the call graph for this function:



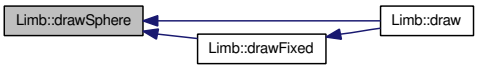

Here is the caller graph for this function:




#### B.15.3.5 void Limb::drawSphere () [protected]

Draws a sphere for the limb. The needed mesurements are stored in the object fields.

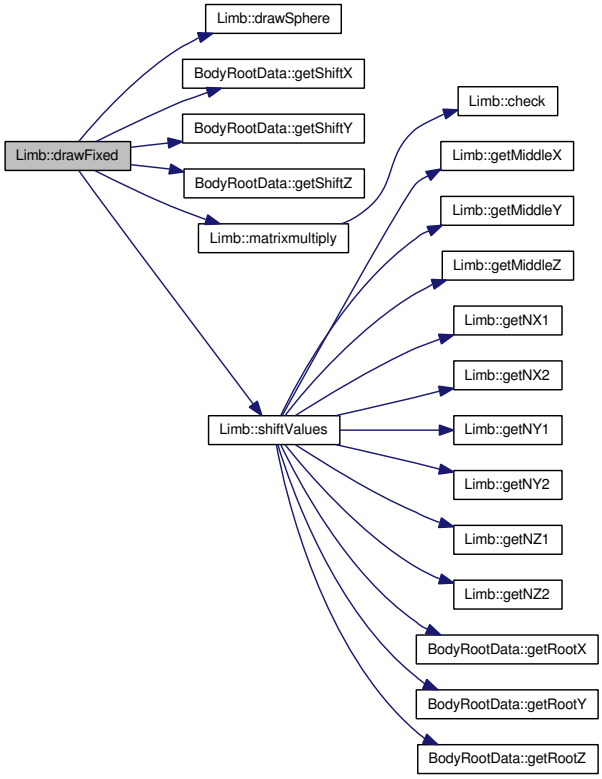
Here is the caller graph for this function:



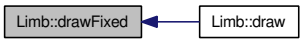

#### B.15.3.6 void Limb::drawFixed () [protected]

Draws the limb at a prefixed angle. This method is used for fixed limbs with no sensors assigned.

Here is the call graph for this function:



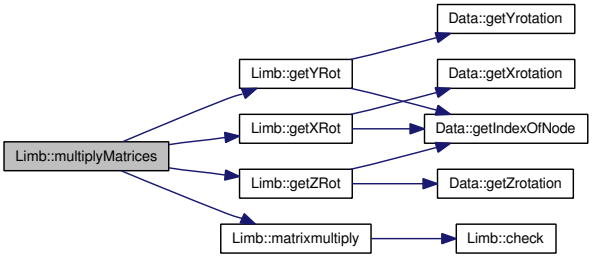

Here is the caller graph for this function:




### B.15.3.7 void Limb::multiplyMatrices () [protected]

Rotates the start and ending points of the limb.

Here is the call graph for this function:

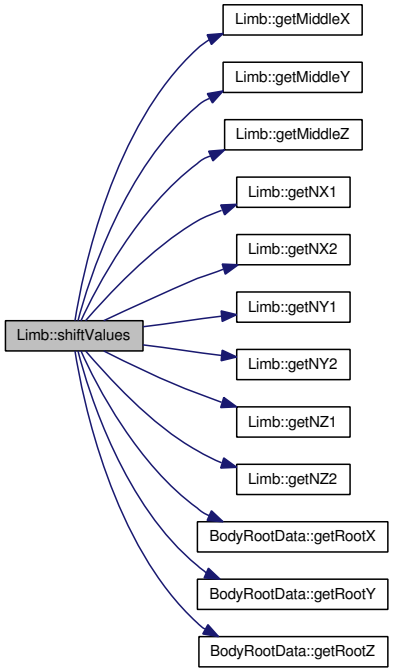



Here is the caller graph for this function:

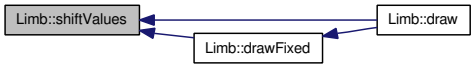

#### B.15.3.8 void Limb::shiftValues () [protected]

Get the starting coordinates from the father. Which coordinates we use, can be influenced by the set InheritCoordinates method. If we are the root limb we use the root coordinates in the root data.

Here is the call graph for this function:



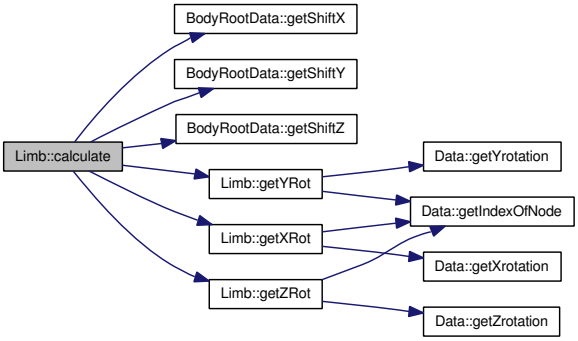

Here is the caller graph for this function:




### B.15.3.9 void Limb::calculate () [protected]

This method calculates the rotations for this limb according to the sensor data and the azimuthstate.

Here is the call graph for this function:




Here is the caller graph for this function:


#### B.15.3.10 void Limb::setData (Data * *d*)

Setter method for the pointer to the used data set. The data can come from file or actual sensors. This method automatically passes the data pointer down to all its children. And they will pass it on to their children. Therefore just calling the method on a root limb is sufficient to set the data for a whole structure.

#### B.15.3.11 Data * Limb::getData ()

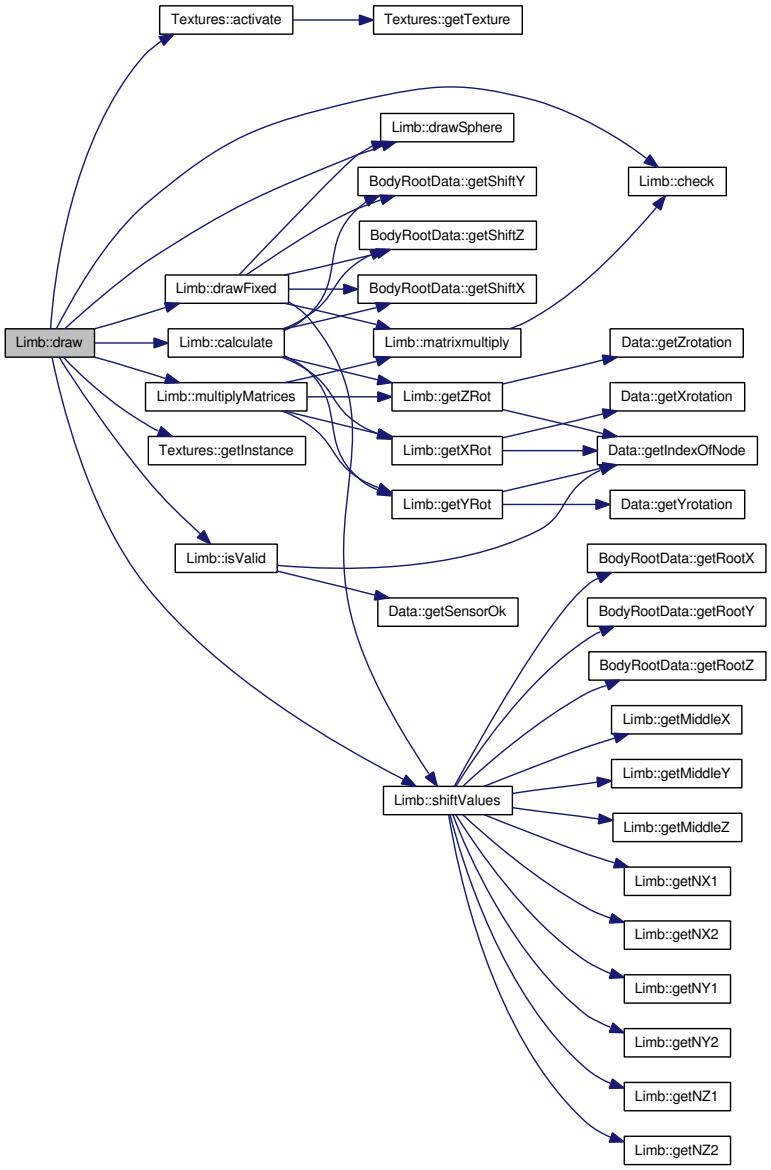
Getter method for the pointer to the used data set. The data can come from file or actual sensors.

Here is the caller graph for this function:

### B.15.3.12 void Limb::draw () [virtual]

Just use this method to draw the limb. This method checks, rotates and draws the limb.

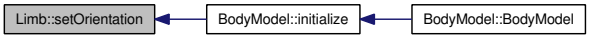
Here is the call graph for this function:



#### B.15.3.13 void Limb::setOrientation (int *o*)

Sets the orientation of the limb. Possibilities currently are 0 for downwards by default and 1 for upwards by default (e.g. chest).

Here is the caller graph for this function:

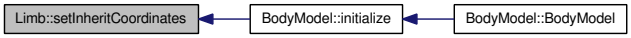



#### B.15.3.14 void Limb::setInheritCoordinates (int *i* = 2)

Sets which coordinated have to be inherited from the father limb. Possibilities are:

- 1 for the starting coordinates;
- 2 (default) for the ending coordinates;
- 3 for the center coordinates;

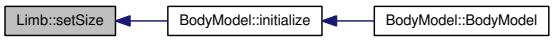
Here is the caller graph for this function:




#### B.15.3.15 void Limb::setSize (float *x*, float *y*, float *z*)

Limbs are always spheres. This method allows to set the size of the sphere in every dimension.

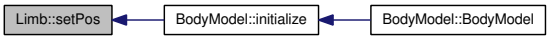
Here is the caller graph for this function:



#### B.15.3.16 void Limb::setPos (float *x1f*, float *y1f*, float *z1f*, float *x2f*, float *y2f*, float *z2f*)

Sets the default position of the limb. This means from this position the limb is rotated by the angle given by the assigned sensor. Using this method you can set all the needed coordinates at once.

Here is the caller graph for this function:




#### B.15.3.17 void Limb::setX1 (float *f*)

Sets the default position of the limb. This means from this position the limb is rotated by the angle given by the assigned sensor. Using this method you can set X coordinate of the starting position.

#### B.15.3.18 void Limb::setY1 (float *f*)

Sets the default position of the limb. This means from this position the limb is rotated by the angle given by the assigned sensor. Using this method you can set Y coordinate of the starting position.

#### B.15.3.19 void Limb::setZ1 (float *f*)

Sets the default position of the limb. This means from this position the limb is rotated by the angle given by the assigned sensor. Using this method you can set Z coordinate of the starting position.

#### B.15.3.20 void Limb::setX2 (float *f*)

Sets the default position of the limb. This means from this position the limb is rotated by the angle given by the assigned sensor. Using this method you can set X coordinate of the ending position.

#### B.15.3.21 void Limb::setY2 (float *f*)

Sets the default position of the limb. This means from this position the limb is rotated by the angle given by the assigned sensor. Using this method you can set Y coordinate of the ending position.

#### B.15.3.22 void Limb::setZ2 (float *f*)

Sets the default position of the limb. This means from this position the limb is rotated by the angle given by the assigned sensor. Using this method you can set Z coordinate of the ending position.

Here is the caller graph for this function:


#### B.15.3.23 void Limb::setFixedRot (float *frx*, float *fry*, float *frz*)

Sets the rotation values for this limb if there is no sensor assigned to this limb. This values only get used if there is no sensor assigned! So its basically a simulation of the missing sensor. Use this feature to add fixed limbs to a body.

#### B.15.3.24 float Limb::getX1 ()

Gets the default position of the limb. This means you get the position of the limb before it is rotated by the angle given by the assigned sensor. Using this method get the X coordinate of the starting position (e.g. left side of the limb).

Here is the caller graph for this function:

#### B.15.3.25 float Limb::getY1 ()

Gets the default position of the limb. This means you get the position of the limb before it is rotated by the angle given by the assigned sensor. Using this method get the Y coordinate of the starting position (e.g. left side of the limb).

Here is the caller graph for this function:


#### B.15.3.26 float Limb::getZ1 ()

Gets the default position of the limb. This means you get the position of the limb before it is rotated by the angle given by the assigned sensor. Using this method get the Z coordinate of the starting position (e.g. left side of the limb).

Here is the caller graph for this function:


#### B.15.3.27 float Limb::getX2 ()

Gets the default position of the limb. This means you get the position of the limb before it is rotated by the angle given by the assigned sensor. Using this method get the X coordinate of the ending position (e.g. right side of the limb).

Here is the caller graph for this function:


#### B.15.3.28 float Limb::getY2 ()

Gets the default position of the limb. This means you get the position of the limb before it is rotated by the angle given by the assigned sensor. Using this method get

the Y coordinate of the ending position (e.g. right side of the limb).

Here is the caller graph for this function:


#### B.15.3.29 float Limb::getZ2 ()

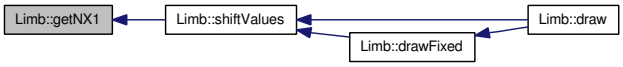
Gets the default position of the limb. This means you get the position of the limb before it is rotated by the angle given by the assigned sensor. Using this method get the Z coordinate of the ending position (e.g. right side of the limb).

Here is the caller graph for this function:


#### B.15.3.30 float Limb::getNX1 ()

Gets the actual position of the limb. This means you get the position of the limb after it is rotated by the angle given by the assigned sensor. Using this method get the X coordinate of the starting position (e.g. left side of the limb).

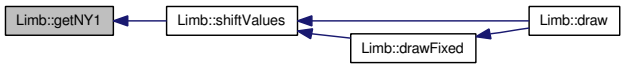
Here is the caller graph for this function:




#### B.15.3.31 float Limb::getNY1 ()

Gets the actual position of the limb. This means you get the position of the limb after it is rotated by the angle given by the assigned sensor. Using this method get the Y coordinate of the starting position (e.g. left side of the limb).

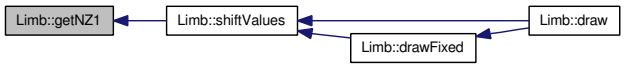
Here is the caller graph for this function:




#### B.15.3.32 float Limb::getNZ1 ()

Gets the actual position of the limb. This means you get the position of the limb after it is rotated by the angle given by the assigned sensor. Using this method get the Z coordinate of the starting position (e.g. left side of the limb).

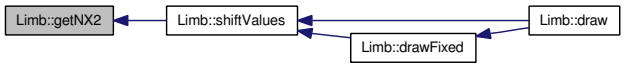
Here is the caller graph for this function:




#### B.15.3.33 float Limb::getNX2 ()

Gets the actual position of the limb. This means you get the position of the limb after it is rotated by the angle given by the assigned sensor. Using this method get the X coordinate of the ending position (e.g. right side of the limb).

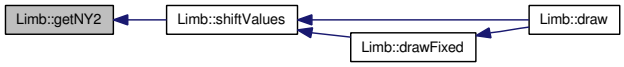
Here is the caller graph for this function:




#### B.15.3.34 float Limb::getNY2 ()

Gets the actual position of the limb. This means you get the position of the limb after it is rotated by the angle given by the assigned sensor. Using this method get the Y coordinate of the ending position (e.g. right side of the limb).

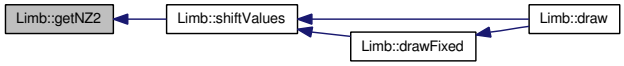
Here is the caller graph for this function:




#### B.15.3.35 float Limb::getNZ2 ()

Gets the actual position of the limb. This means you get the position of the limb after it is rotated by the angle given by the assigned sensor. Using this method get the Z coordinate of the ending position (e.g. right side of the limb).

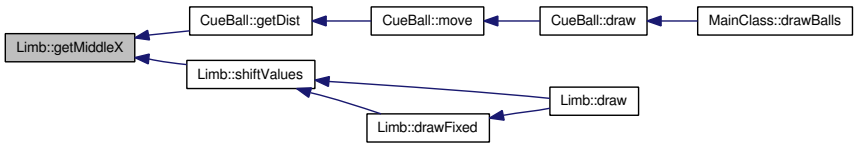
Here is the caller graph for this function:




#### B.15.3.36 float Limb::getMiddleX ()

Gets the actual position of the limb. This means you get the position of the limb after it is rotated by the angle given by the assigned sensor. Using this method get the X coordinate of the center point of the limb.

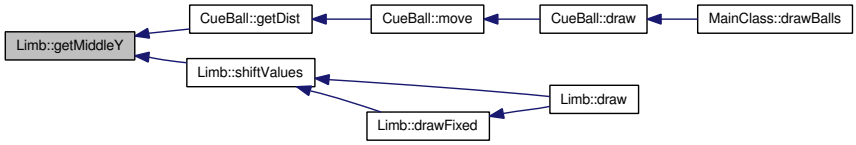
Here is the caller graph for this function:




#### B.15.3.37 float Limb::getMiddleY ()

Gets the actual position of the limb. This means you get the position of the limb after it is rotated by the angle given by the assigned sensor. Using this method get the Y coordinate of the center point of the limb.

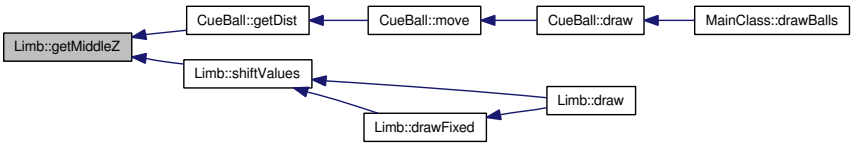
Here is the caller graph for this function:




#### B.15.3.38 float Limb::getMiddleZ ()

Gets the actual position of the limb. This means you get the position of the limb after it is rotated by the angle given by the assigned sensor. Using this method get the Z coordinate of the center point of the limb.

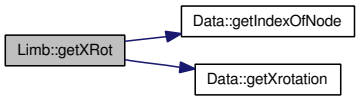
Here is the caller graph for this function:



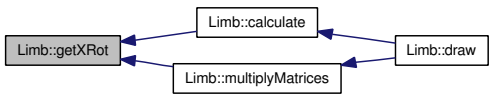

#### B.15.3.39 float Limb::getXRot ()

Returns the current rotation around the X axis. Currently you only get the sensor value. If there is no sensor present this method doesn't work. The X rotation gets inverted according to the set orientation.

Here is the call graph for this function:



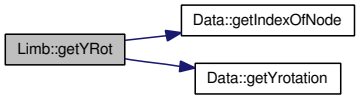

Here is the caller graph for this function:



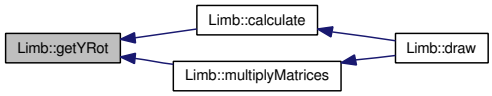
#### B.15.3.40 float Limb::getYRot ()

Returns the current rotation around the Y axis. Currently you only get the sensor value. If there is no sensor present this method doesn't work currently.

Here is the call graph for this function:



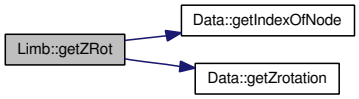

Here is the caller graph for this function:



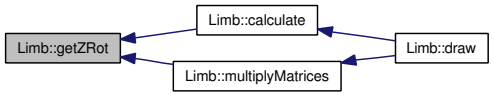

#### B.15.3.41 float Limb::getZRot ()

Returns the current rotation around the Z axis. Currently you only get the sensor value. If there is no sensor present this method doesn't work currently.

Here is the call graph for this function:




Here is the caller graph for this function:



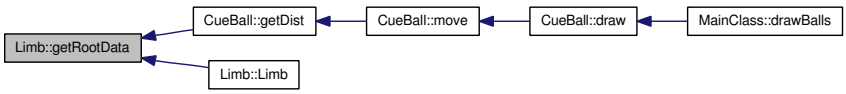
#### B.15.3.42 void Limb::setRootData (BodyRootData * *brd*) [virtual]

Setter method for the root data. This data is used to shift a whole body. This method automatically passes the root data pointer down to all its children. And they will pass it on to their children and so on. Therefore just calling the method on a root limb is sufficient to set the root data for a whole structure.

#### B.15.3.43 BodyRootData * Limb::getRootData ()

Returns the reference to the root data of this limb. That's leaking. This means the private root data can be modified from outside by the provided reference.

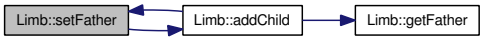
Here is the caller graph for this function:



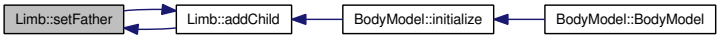

#### B.15.3.44 void Limb::setFather (Limb * *l*)

Sets the father limb for this limb. If the father is Null this means the current limb is the root limb of the structure. Fathership is used to describe a whole body consisting of limbs.

Here is the call graph for this function:



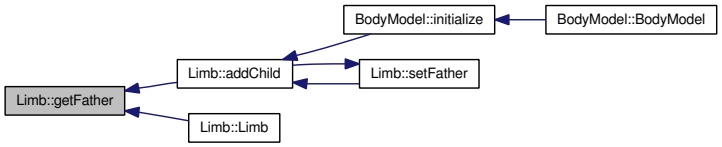

Here is the caller graph for this function:



#### B.15.3.45 Limb ∗ Limb::getFather ()

Returns the father limb for this limb. If the father is Null this means the current limb is the root limb of the structure. Fathership is used to describe a whole body consisting of limbs.

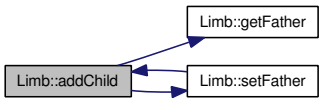
Here is the caller graph for this function:



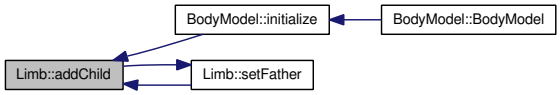

#### B.15.3.46 void Limb::addChild (Limb ∗ *l*)

Adds a child to this limb. This means this limb becomes the father of the given limb. Fathership is used to describe a whole body consisting of limbs.

Here is the call graph for this function:



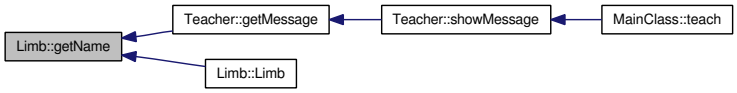

Here is the caller graph for this function:




#### B.15.3.47 string Limb::getName ()

Getter method for the limb name. Every limb has a name. The name is used for the textual feed back to the user.

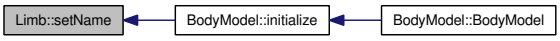
Here is the caller graph for this function:




#### B.15.3.48 void Limb::setName (string *n*)

Setter method for the limb name. Every limb has a name. The name is used for the textual feed back to the user.

Here is the caller graph for this function:




#### B.15.3.49 int Limb::getNumber ()

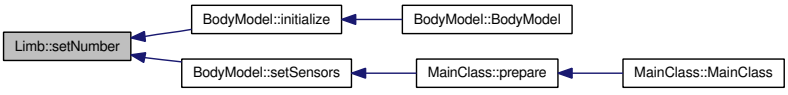
Returns the number of the assigned sensor.

Here is the caller graph for this function:


#### B.15.3.50 void Limb::setNumber (int *n*)

Sets the number of the assigned sensor.

Here is the caller graph for this function:



#### B.15.3.51 void Limb::print ()

Prints the current coordinates of this limb to the command line. Use this method for testing purpose.

#### B.15.3.52 void Limb::setBasisShiftX (float *bs*)

Sets the basis Shift in X direction This shift is applied before rotating the limb.

#### B.15.3.53 void Limb::setBasisShiftY (float *bs*)

Sets the basis Shift in Y direction This shift is applied before rotating the limb.

#### B.15.3.54 void Limb::setBasisShiftZ (float *bs*)

Sets the basis Shift in Z direction This shift is applied before rotating the limb.

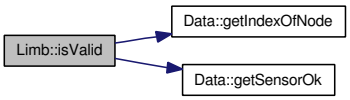
#### B.15.3.55 void Limb::setTexture (string *str*)

Sets the texture of this limb. The given texture isn't passed on to the child limbs. So the texture can be set to a different value for every limb. Use the filename without path or extension to describe the texture.

#### B.15.3.56 bool Limb::isValid ()

Returns true if the assigned sensor data is valid.

Here is the call graph for this function:



Here is the caller graph for this function:

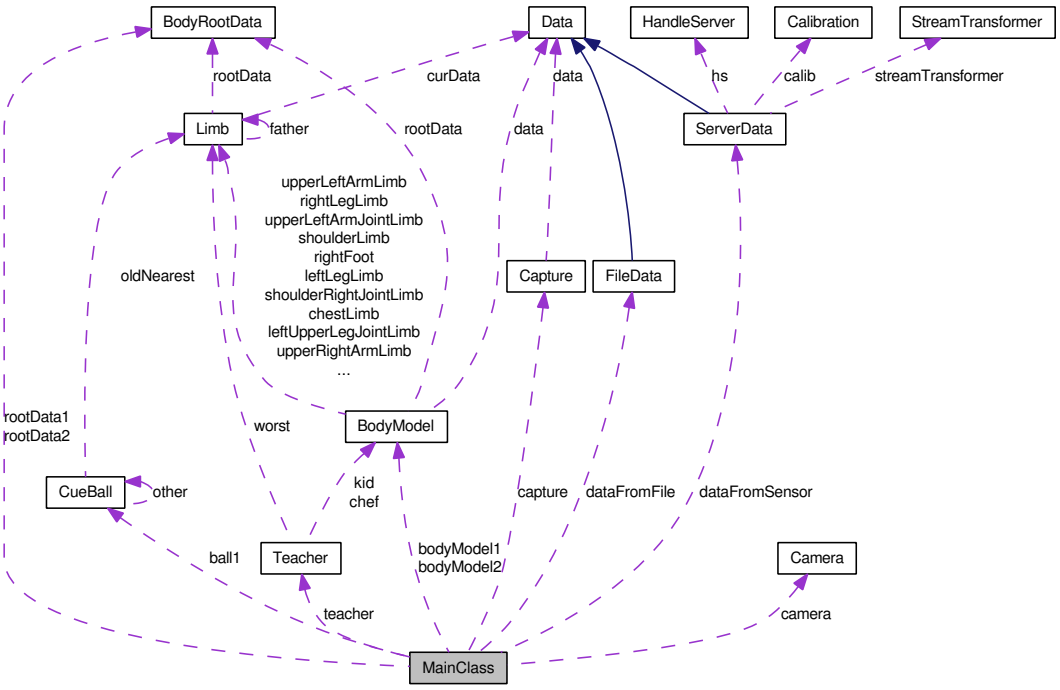

The documentation for this class was generated from the following files:

- Limb.h
- Limb.cpp

## B.16 MainClass Class Reference

`#include <MainClass.h>`

Collaboration diagram for MainClass:




### Public Member Functions

- MainClass ()
- MainClass (bool directX)
- virtual ~MainClass ()
- void drawAxisLines ()
- void drawSensorOk ()
- bool checkForEscape ()
- void readKey ()
- void saveData ()
- void handleShift ()
- void handleCamera ()

- void drawBackground ()
- void drawAvatars ()
- void drawBalls ()
- void teach ()
- void openScene ()
- void closeScene ()
- void **refreshData** ()

## Private Member Functions

- void getcalibrationdata (char ∗, int ∗, int ∗, int ∗, int ∗, int ∗, int ∗, int ∗, int ∗, int ∗)
- float modPi (float)
- void prepare ()

## Private Attributes

- ServerData ∗ **dataFromSensor**
- FileData ∗ **dataFromFile**
- BodyModel ∗ **bodyModel1**
- BodyModel ∗ **bodyModel2**
- BodyRootData ∗ **rootData1**
- BodyRootData ∗ **rootData2**
- CueBall ∗ **ball1**
- Capture ∗ **capture**
- Camera ∗ **camera**
- Teacher ∗ **teacher**
- Bitmap ∗ **background**
- Material ∗ **material**
- char **lastKey**
- bool **capturing**
- bool **playing**
- bool **showBalls**

### B.16.1 Detailed Description

This class just offers all basic steps for the function of the software. It's like a layer of abstraction between the main-method and the basic classes. Due to this class the main method is quite simple and easy to understand.

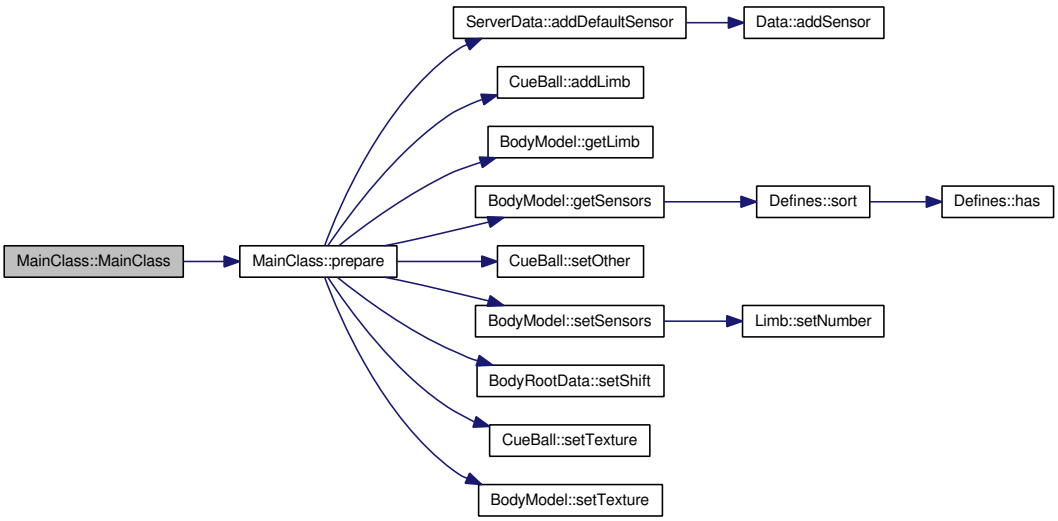
Author: Lars;

### B.16.2 Constructor & Destructor Documentation

#### B.16.2.1 MainClass::MainClass ()

Constructor: Prepares the body models, cueballs, sensors, background and material properties.

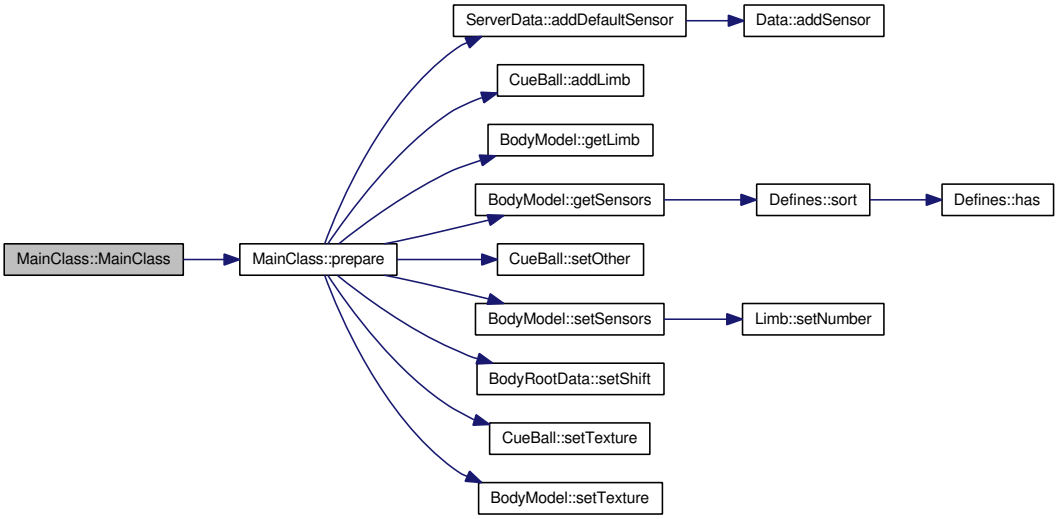
Here is the call graph for this function:




#### B.16.2.2 MainClass::MainClass (bool *directX*)

Constructor: Prepares the body models, cueballs, sensors, background and material properties.

Here is the call graph for this function:




#### B.16.2.3 MainClass::~MainClass () [virtual]

Destructor: Frees all the resources.

### B.16.3 Member Function Documentation

#### B.16.3.1 void MainClass::getcalibrationdata (char * *fname*, int * *intmagxoffset*, int * *intmagyoffset*, int * *intmagzoffset*, int * *intacczmax*, int * *intacczmin*, int * *intaccymax*, int * *intaccymin*, int * *intaccxmax*, int * *intaccxmin*) [private]

Reads the calibration data for one sensor. In the first parameter you have to provide the file name of the calibration file. The rest of the parameters is used to return you the desired values. Maybe a struct as return type would be more beatiful.

#### B.16.3.2 float MainClass::modPi (float *f*) [private]

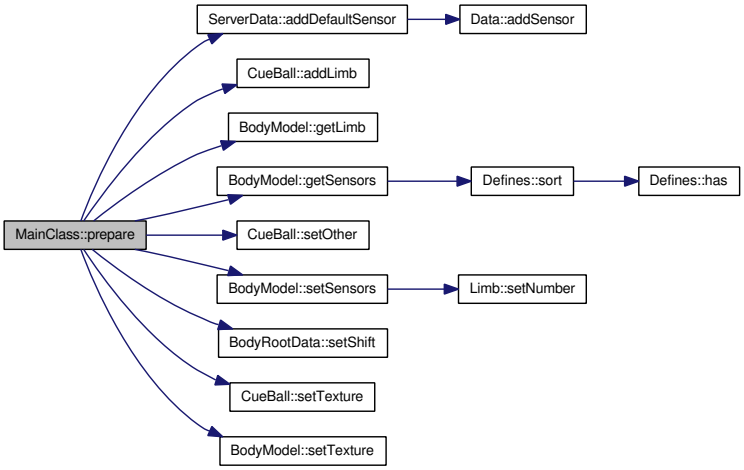
Simple helper method. It just returns the given float value modulo 2*PI.

Here is the caller graph for this function:


### B.16.3.3 void MainClass::prepare () [private]

Gets called by the constructor. Does everything the constructor does.

Here is the call graph for this function:

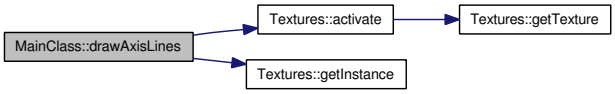



Here is the caller graph for this function:


### B.16.3.4 void MainClass::drawAxisLines ()

If like to get some type of coordinate axes, use this method. It just draws three long and connected cubicles.

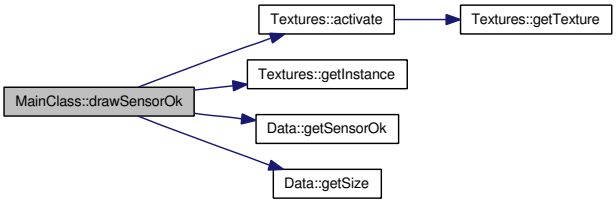
Here is the call graph for this function:




#### B.16.3.5 void MainClass::drawSensorOk ()

This method draws small cubes to indicate if the sensors are working properly. Like this its easier to determine which sensor is low on battery.

Here is the call graph for this function:




#### B.16.3.6 bool MainClass::checkForEscape ()

Checks if the last key hit was escape.

#### B.16.3.7 void MainClass::readKey ()

Simply reads a key from the keyboard if there is any. The program doesn't get blocked. If there's a key read, it gets stored in a field of the class.

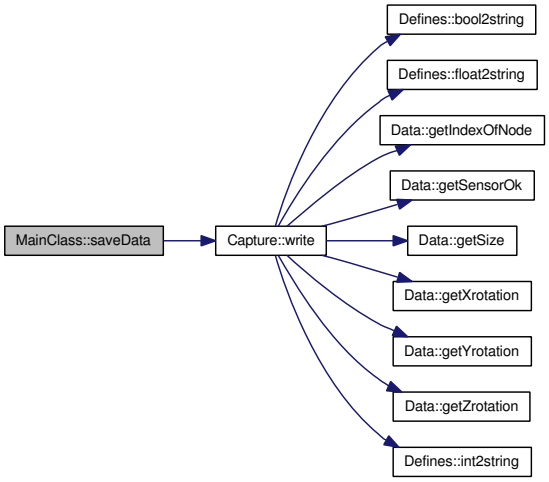
#### B.16.3.8 void MainClass::saveData ()

Method to control the saving of the sensor data to disk. There are different modes:

- No recording;
- Single shot recording (one frame each keypress);

- Continous recording;

Here is the call graph for this function:




#### B.16.3.9 void MainClass::handleShift ()

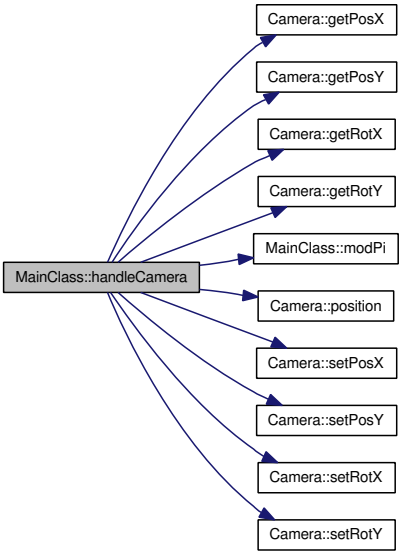
Method to change the shift values of the two body models according to pressed keys. Therefore it's possible to merge the avatars or display them beside each other.

Here is the call graph for this function:


#### B.16.3.10 void MainClass::handleCamera ()

Method to handle the camera. Using the arrow keys you can move the camera around the scene.

Here is the call graph for this function:



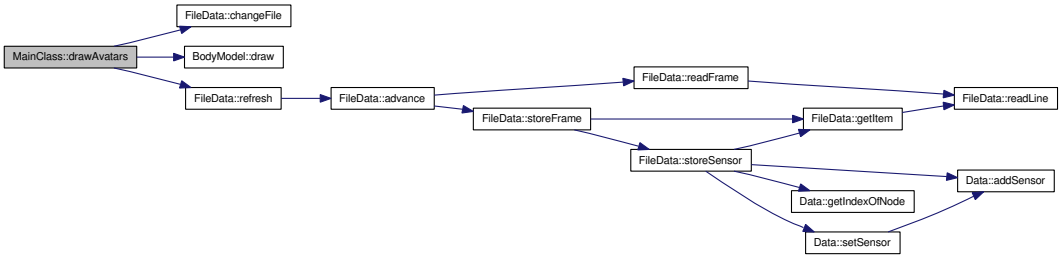

### B.16.3.11 void MainClass::drawBackground ()

Draws the background.

### B.16.3.12 void MainClass::drawAvatars ()

Draws the body models on the screen.

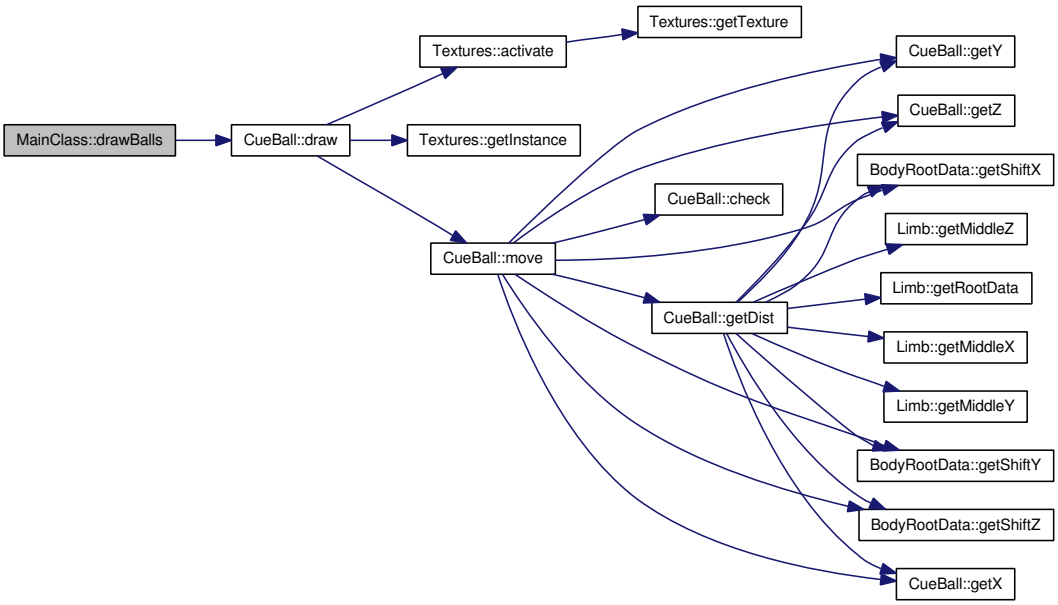
Here is the call graph for this function:



### B.16.3.13 void MainClass::drawBalls ()

Draws the cueballs to play with on the screen.

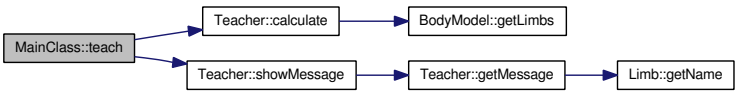
Here is the call graph for this function:




### B.16.3.14 void MainClass::teach ()

Runs the teacher on the body models. The teacher shows a message on the screen and changes the color of the limbs.

Here is the call graph for this function:




### B.16.3.15 void MainClass::openScene ()

Starts and clears a new direct X scene.

#### B.16.3.16 void MainClass::closeScene ()

Closes the direct X scene.

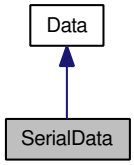
The documentation for this class was generated from the following files:

- MainClass.h
- MainClass.cpp

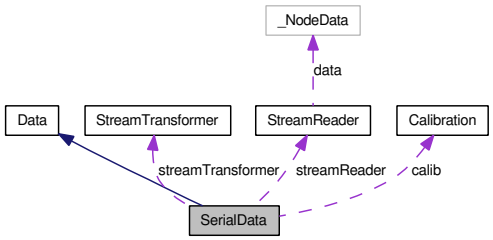
## B.17 SerialData Class Reference

`#include <SerialData.h>`

Inheritance diagram for SerialData:




Collaboration diagram for SerialData:




### Public Member Functions

- SerialData ()
- virtual ~SerialData ()
- void refresh ()
- void refreshOne (int)
- void **addDefaultSensor** (int)
- float getAccX (int)
- float getAccY (int)
- float getAccZ (int)
- float getMagX (int)
- float getMagY (int)
- float getMagZ (int)
- bool getValid (int)

### Private Member Functions

- NodeData readSensor (StreamReader ∗, int)

### Private Attributes

- StreamReader ∗ **streamReader**
- StreamTransformer ∗ **streamTransformer**
- Calibration ∗ **calib**
- vector< NodeData > **sensorData**

## B.17.1 Detailed Description

This class serves as an abstraction layer before the StreamReader. Basically I took all the handling of the sensor data from the given monolithic software and put it to this class.

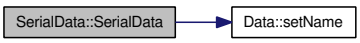
Author: Lars;

## B.17.2 Constructor & Destructor Documentation

### B.17.2.1 SerialData::SerialData ()

Class constructor: Connects to the StreamReader.

Here is the call graph for this function:




### B.17.2.2 SerialData::∼SerialData () [virtual]

Class destructor: Disconnects the StreamReader.

## B.17.3 Member Function Documentation

### B.17.3.1 NodeData SerialData::readSensor (StreamReader * *sr*, int *sensor*) [private]

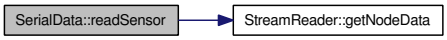
Method for getting the data of a single sensor. A StreamReader and a sensor number have to be passed as parameters.

This class also provides a higher access level to the sensors. Therefore you better use the available get-methods.

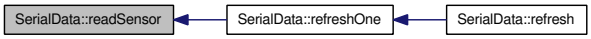
The returntype NodeData of this method is just a small struct. It contains fields for data of compass, magnetometers and battery voltage.

The data is cached by the StreamReader-library. So if you turn the sensors off during the run of the program, the program will continue just using the last valid data package in storage.

Here is the call graph for this function:




Here is the caller graph for this function:

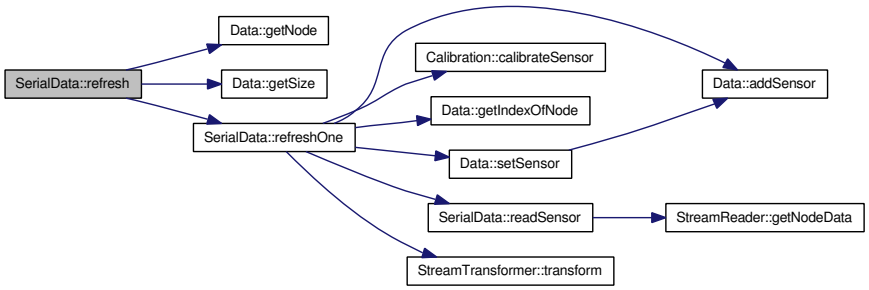



### B.17.3.2 void SerialData::refresh ()

Refreshes the stored data for the known nodes.

Reimplemented from Data.

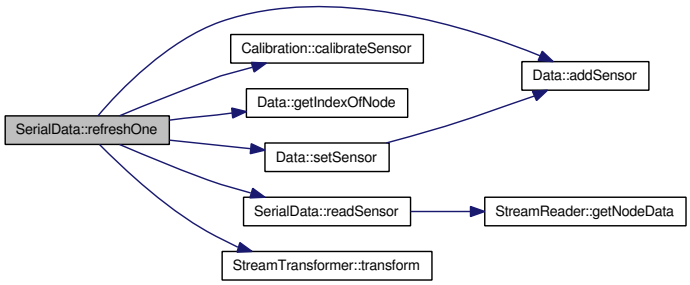
Here is the call graph for this function:




### B.17.3.3 void SerialData::refreshOne (int *sensor*)

Refreshes the stored data of a single sensor. If the sensor node isn't known yet it's added to the known list.

Here is the call graph for this function:




Here is the caller graph for this function:


### B.17.3.4 float SerialData::getAccX (int *index*)

Returns the current acceleration in x destination of the given sensor. Use this method to get raw data for test purpose.

#### B.17.3.5 float SerialData::getAccY (int *index*)

Returns the current acceleration in y destination of the given sensor. Use this method to get raw data for test purpose.

#### B.17.3.6 float SerialData::getAccZ (int *index*)

Returns the current acceleration in z destination of the given sensor. Use this method to get raw data for test purpose.

#### B.17.3.7 float SerialData::getMagX (int *index*)

Returns the current magnetometer data in x destination of the given sensor. Use this method to get raw data for test purpose.

#### B.17.3.8 float SerialData::getMagY (int *index*)

Returns the current magnetometer data in y destination of the given sensor. Use this method to get raw data for test purpose.

#### B.17.3.9 float SerialData::getMagZ (int *index*)

Returns the current magnetometer data in z destination of the given sensor. Use this method to get raw data for test purpose. A refresh for getting new data is automatically done at every call.

#### B.17.3.10 bool SerialData::getValid (int *index*)

Returns true if the data of the given sensor is currently valid. Use this method to get raw data for test purpose. A refresh for getting new data is automatically done at every call.

The documentation for this class was generated from the following files:

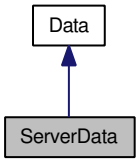
- SerialData.h

- SerialData.cpp

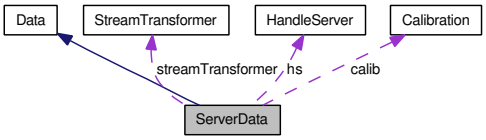
## B.18 ServerData Class Reference

`#include <ServerData.h>`

Inheritance diagram for ServerData:




Collaboration diagram for ServerData:




### Public Member Functions

- ServerData ()
- virtual ~ServerData ()
- void refresh ()
- void refreshOne (int)
- void addDefaultSensor (int)
- float getAccX (int)
- float getAccY (int)
- float getAccZ (int)
- float getMagX (int)
- float getMagY (int)
- float getMagZ (int)
- bool getValid (int)

### Private Member Functions

- int getBytes (int, string, int)
- unsigned getByte (string, int)
- int getWord (string, int)
- void testCalib ()

### Private Attributes

- StreamTransformer * **streamTransformer**
- Calibration * **calib**
- vector< NodeData > **sensorData**
- HandleServer * **hs**

## B.18.1 Detailed Description

This class serves as an abstraction layer on top of HandleServer. HandleServer generally serves for interfacing with the server. This class inherits from Data and represents a source of motion data for the AvatarPlayer.

Author: Lars;

## B.18.2 Constructor & Destructor Documentation

### B.18.2.1 ServerData::ServerData ()

Class constructor: Opens the connection to the server. Change here if you need to adapt the server, port numbers or the stream name. Current settings are: UDP-port 7333, TCP-port 2345, host mona.lawi.ch, stream-name phil, subscriber-name avatarPlayer.

Here is the call graph for this function:

#### B.18.2.2 ServerData::~ServerData () [virtual]

Class destructor: Closes the server connection.

### B.18.3 Member Function Documentation

#### B.18.3.1 int ServerData::getBytes (int *numOfBytes*, string *str*, int *start*) [private]

Extracts a numOfBytes bytes value out of numOfBytes∗2 characters. It uses the numOfBytes∗2 characters in the string str starting at the position start.

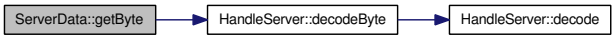
Here is the call graph for this function:


Here is the caller graph for this function:


#### B.18.3.2 unsigned ServerData::getByte (string *str*, int *start*) [private]

Extracts a 1 byte value out of 2 characters. It uses the 2 characters in the string str starting at the position start.

Here is the call graph for this function:



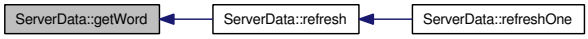

Here is the caller graph for this function:

#### B.18.3.3 int ServerData::getWord (string *str*, int *start*) [private]

Extracts a 2 byte value out of 4 characters. It uses the 4 characters in the string str starting at the position start.

Here is the call graph for this function:

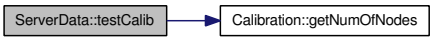

Here is the caller graph for this function:




#### B.18.3.4 void ServerData::testCalib () [private]

Debug method: Returns the number of nodes in the calibration instance. This feature is currently unused.

Here is the call graph for this function:

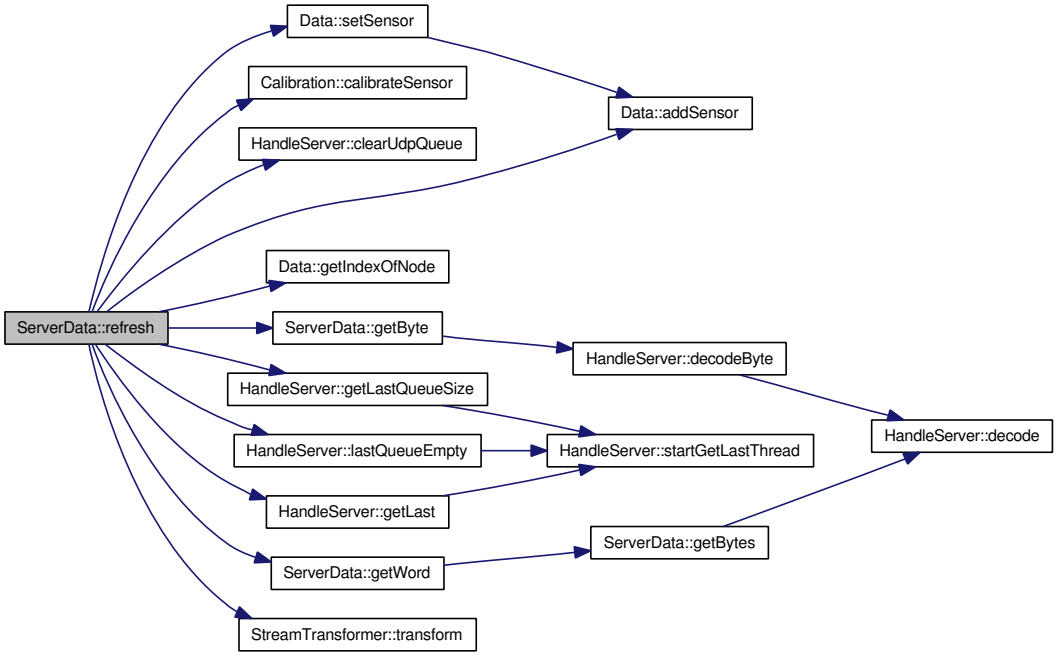



#### B.18.3.5 void ServerData::refresh ()

Refreshes the stored data for the available nodes. Refresh ends when no more UDP packets are available or more than the number of cached sensors packets have been received. TODO: Extract method for translating string to NodeData.

Reimplemented from Data.

Here is the call graph for this function:

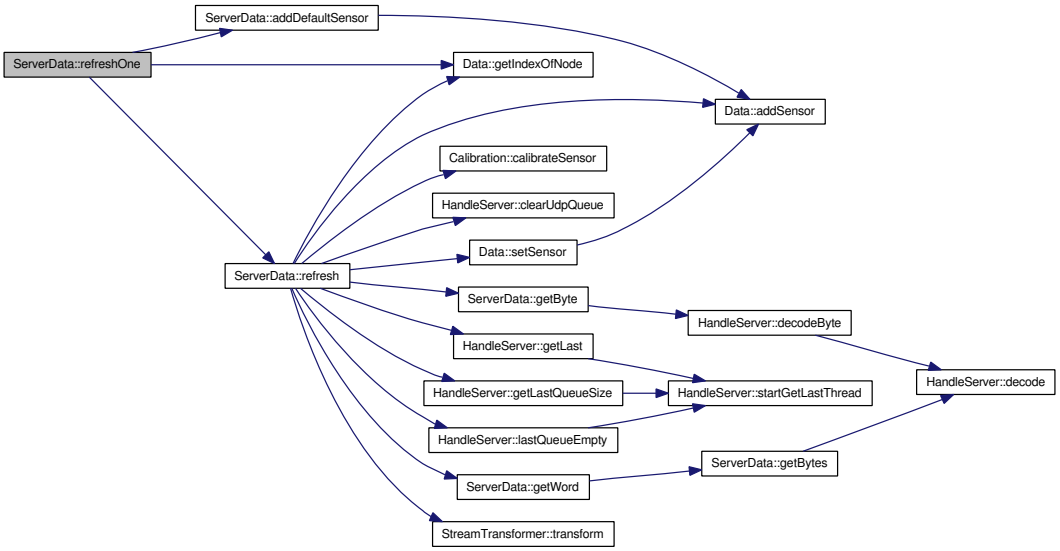



Here is the caller graph for this function:


### B.18.3.6 void ServerData::refreshOne (int *sensor*)

Refreshes the stored data of every sensor. Therefore in this implementation it's merely a synonym for refresh(). But if the given sensor node isn't known yet it's added to the known list. That's the difference between this two. Nevertheless, when reading socket data the available nodes are detected automatically.

Here is the call graph for this function:

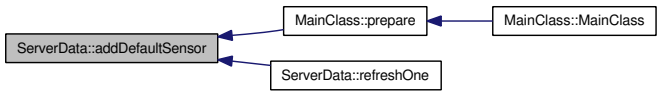



### B.18.3.7 void ServerData::addDefaultSensor (int *node*)

Adds a sensor node to the internal cache. Unneeded in this class. In the stream available sensors get added automatically.

Here is the call graph for this function:


Here is the caller graph for this function:




### B.18.3.8 float ServerData::getAccX (int *index*)

Returns the current acceleration in x destination of the given sensor. Use this method to get raw data for test purpose.

#### B.18.3.9 float ServerData::getAccY (int *index*)

Returns the current acceleration in y destination of the given sensor. Use this method to get raw data for test purpose.

#### B.18.3.10 float ServerData::getAccZ (int *index*)

Returns the current acceleration in z destination of the given sensor. Use this method to get raw data for test purpose.

#### B.18.3.11 float ServerData::getMagX (int *index*)

Returns the current magnetometer data in x destination of the given sensor. Use this method to get raw data for test purpose.

#### B.18.3.12 float ServerData::getMagY (int *index*)

Returns the current magnetometer data in y destination of the given sensor. Use this method to get raw data for test purpose.

#### B.18.3.13 float ServerData::getMagZ (int *index*)

Returns the current magnetometer data in z destination of the given sensor. Use this method to get raw data for test purpose. A refresh for getting new data is automatically done at every call.

#### B.18.3.14 bool ServerData::getValid (int *index*)

Returns true if the data of the given sensor is currently valid. Use this method to get raw data for test purpose. A refresh for getting new data is automatically done at every call.

The documentation for this class was generated from the following files:

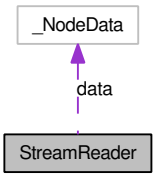
- ServerData.h

- ServerData.cpp

## B.19 StreamReader Class Reference

`#include <StreamReader.h>`

Collaboration diagram for StreamReader:




### Public Member Functions

- NodeData getNodeData (int nodeID)
- StreamReader (char *comPort)
- ~StreamReader ()

### Protected Member Functions

- void run ()

### Private Attributes

- NodeData **data** [256]
- bool **validNodes** [256]
- HANDLE **commHandle**

### B.19.1 Detailed Description

Encapsulates the whole data of a single sensor for one measurement. Reads out the sensor network stream of the COM conection passed as Argument. Starts and stops automatic after con/destruction.

Use getNodeData(int nodeID) to get the current data of a node.

Returns invalid values when: -connection to port failed -no data of this node has been red out yet

This class directly interfaces with the sensors. It runs in an own thread.

## B.19.2 Constructor & Destructor Documentation

### B.19.2.1 StreamReader::StreamReader (char * *comPort*)

Class constructor: Initializes all the fields. Then it opens the com port and initializes the data transfer.

### B.19.2.2 StreamReader::~StreamReader ()

Class destructor: Closes the handle for the com port.

## B.19.3 Member Function Documentation

### B.19.3.1 NodeData StreamReader::getNodeData (int *nodeID*)

Returns the NodeData-struct for the sensor with the given ID number.

Here is the caller graph for this function:


### B.19.3.2 void StreamReader::run () [protected]

Live method of the thread. This method continously reads the sensors and caches the data.

The documentation for this class was generated from the following files:

- StreamReader.h
- StreamReader.cpp

## B.20 StreamTransformer Class Reference

`#include <StreamTransformer.h>`

### Public Member Functions

- EulerRotation transform (NodeData newData, NodeData calibrationData)
- StreamTransformer ()
- ~StreamTransformer ()

### Static Public Member Functions

- static float getPolarAngle (float x, float y)

### B.20.1 Detailed Description

Transforms NodeData to three euler angles, used to represent 360 rotation around each euler axis. DirectX uses a left handed coordinate system!

Currently unused class? The idea was to encapsulate the transformation of the raw sensor data to euler angles in this class. Currently all this stuff is directly done in SerialData. SerialData directly accesses StreamReader.

### B.20.2 Constructor & Destructor Documentation

#### B.20.2.1 StreamTransformer::StreamTransformer ()

Empty class constructor.

#### B.20.2.2 StreamTransformer::~StreamTransformer ()

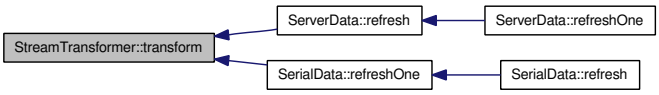
Empty class destructor.

## B.20.3 Member Function Documentation

### B.20.3.1 EulerRotation StreamTransformer::transform (NodeData *newData*, NodeData *calibrationData*)

Does the necessary vector algebra. Computes the euler angles out of the sensor data.

Here is the caller graph for this function:




### B.20.3.2 float StreamTransformer::getPolarAngle (float *x*, float *y*) [static]

The two given floats depict a rectangle. This method returns the angle between x and the diagonal of the rectangle.

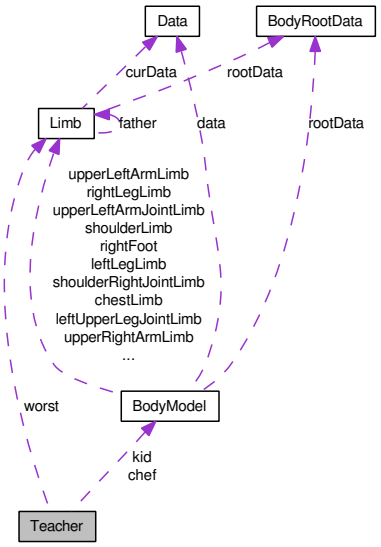
The documentation for this class was generated from the following files:

- StreamTransformer.h
- StreamTransformer.cpp

## B.21 Teacher Class Reference

`#include <Teacher.h>`

Collaboration diagram for Teacher:




### Public Member Functions

- Teacher (BodyModel ∗, BodyModel ∗)
- virtual ∼Teacher ()
- void calculate ()
- void showMessage ()
- string getMessage ()

### Private Attributes

- BodyModel ∗ **kid**
- BodyModel ∗ **chef**
- vector< vector< float > > **diff**
- Limb ∗ **worst**
- int **direction**

- Font * **font**

### B.21.1 Detailed Description

Teacher class. Created on: 24.08.2008; Author: Lars; Allows to compare two body models and provide feedback to one of them.

Author: Lars;

### B.21.2 Constructor & Destructor Documentation

#### B.21.2.1 Teacher::Teacher (BodyModel * *teacher*, BodyModel * *student*)

Class constructor: Takes two body models as parameter. The first for the teacher and the second for the student.

#### B.21.2.2 Teacher::~Teacher () `[virtual]`

Class destructor. Only has to dispose the font used for textual feedback.

### B.21.3 Member Function Documentation

#### B.21.3.1 void Teacher::calculate ()

This method compares the two models. Currently angles aren't compared. Only positions are taken into account. Also in this method the limbs of the student are colored according to their descrepancy to the masters limbs.

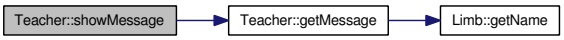
Here is the call graph for this function:


Here is the caller graph for this function:

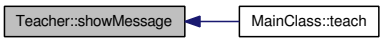
### B.21.3.2 void Teacher::showMessage ()

Plots the Message to the student to the screen.

Here is the call graph for this function:




Here is the caller graph for this function:

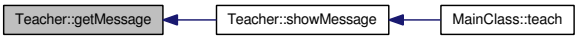



### B.21.3.3 string Teacher::getMessage ()

Returns a message based on the stored information about the worst limb of the student.

Here is the call graph for this function:


Here is the caller graph for this function:




The documentation for this class was generated from the following files:

- Teacher.h
- Teacher.cpp

## B.22 TextureItem Struct Reference

`#include <Textures.h>`

### Public Attributes

- Texture * **pointer**
- string **name**
- string **filename**

### B.22.1 Detailed Description

Encapsulates the texture information.

The documentation for this struct was generated from the following file:

- Textures.h

## B.23 Textures Class Reference

`#include <Textures.h>`

### Public Member Functions

- virtual ~Textures ()
- void activate (string)
- void activateFile (string)

### Static Public Member Functions

- static Textures * getInstance ()

### Private Member Functions

- Textures ()
- void activate (int)
- int loadTexture (string)
- int loadTexture (string, string)
- Texture * getTexture (string)
- Texture * getTextureByFilename (string)
- Texture * getTexture (int)
- string fullFileName (string)

### Private Attributes

- vector< TextureItem > **collection**

### B.23.1 Detailed Description

Simple singleton class for handling textures. Textures always are identified by their filename. That allows to make no difference between already loaded or not yet loaded textures. The filename has to be given without the file extension nor the path. If a not yet loaded texture is desired the system automatically loads the texture. If a request arrives for a already loaded texture the system reuses the loaded texture.

Author: Lars;

### B.23.2 Constructor & Destructor Documentation

#### B.23.2.1 Textures::Textures () [private]

Empty private default constructor.

#### B.23.2.2 Textures::~Textures () [virtual]

The class destructor disposes all the loaded textures.

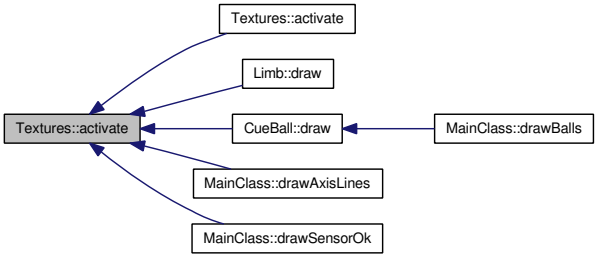
### B.23.3 Member Function Documentation

#### B.23.3.1 void Textures::activate (int *index*) [private]

Activates a texture by the given id number. This method offers a bit more performance. But its unable to load any textures.

Here is the call graph for this function:

Here is the caller graph for this function:

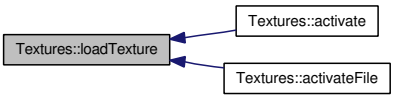



#### B.23.3.2 int Textures::loadTexture (string *name*) [private]

Loads a texture to the system and returns its id number.

Here is the call graph for this function:


Here is the caller graph for this function:



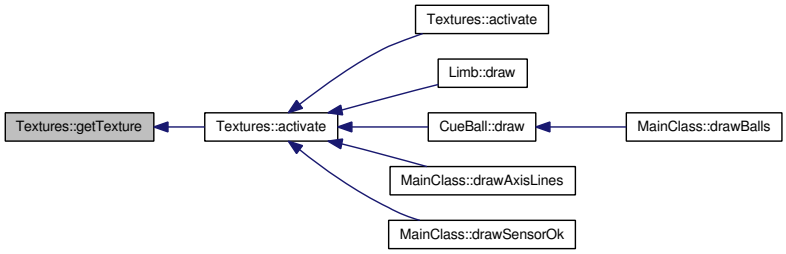

#### B.23.3.3 int Textures::loadTexture (string *name*, string *fname*) [private]

Loads a texture by a given name and filename. Use this method if you want to load a texture from a special path.

#### B.23.3.4 Texture * Textures::getTexture (string *name*) [private]

Searches the internal collection for a texture with the given name. If no texture could be found NULL is returned.

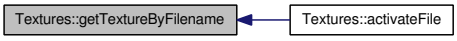
Here is the caller graph for this function:




#### B.23.3.5 Texture * Textures::getTextureByFilename (string *fname*) [private]

Searches the internal collection for a texture with the given file name. If no texture could be found NULL is returned.

Here is the caller graph for this function:



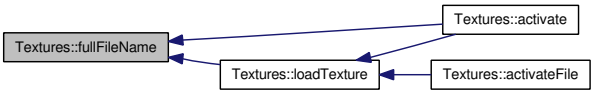

#### B.23.3.6 Texture * Textures::getTexture (int *index*) [private]

Returns the texture from the internal collection with the given ID. If no texture could be found NULL is returned.

#### B.23.3.7 string Textures::fullFileName (string *name*) [private]

Provides the full filename with path and extension for a given texture name. The used path for all textures is defined in Defines.h.

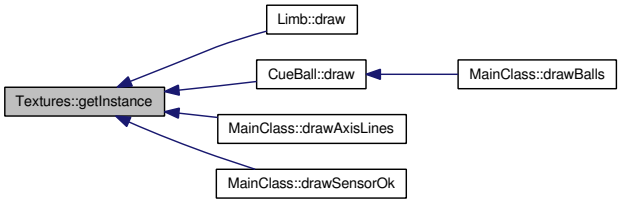
Here is the caller graph for this function:



#### B.23.3.8 Textures * Textures::getInstance () [static]

Returns the one and only instance to the class.

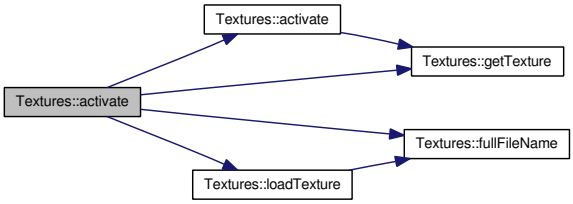
Here is the caller graph for this function:




#### B.23.3.9 void Textures::activate (string *name*)

Activates a texture with the given name. No path nor extension have to be given. If the texture hasn't been loaded yet the system loads the desired texture. Presumably the best idea is just to use only this method.

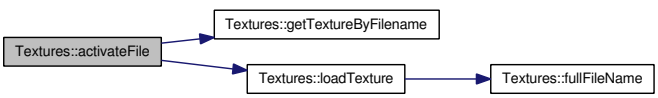
Here is the call graph for this function:




#### B.23.3.10 void Textures::activateFile (string *fname*)

Used to activate a Texture identified by its full filename. Use this method if you need to use a texture stored outside of the default directory.

Here is the call graph for this function:



The documentation for this class was generated from the following files:

- Textures.h
- Textures.cpp

## B.24 XmlItem Struct Reference

`#include <FileData.h>`

### Public Attributes

- string **id**
- string **val**

### B.24.1 Detailed Description

Struct to hold an XML-item. This means, there is a name or id and a value.

The documentation for this struct was generated from the following file:

- FileData.h

# Appendix C

# Indices

## Cite this Thesis

For *bibtex* the following entry can be used.

```
@mastersthesis{WidmerL09,
       author = {Lars Widmer and Dennis Majoe and
         Philip Tschiemer and J\"urg Gutknecht},
       title = "Tai Chi Chuan using wearable Sensors -
         Enhancements in Accuracy, Communications and
         dynamic Motion Recognition",
       year = "2009",
       school = "ETH Z{\"u}rich"
}
```

# Bibliography

Brand, M., Oliver, N., & Pentland, A. (1997), Coupled hidden markov models for complex action recognition, *in* 'Coupled hidden Markov models for complex action recognition'.
**URL:** *http://computer.org/proceedings/cvpr/1997/7822/994; http://csdl.computer.org/comp/proceedings/cvpr/1997/7822/00/78220994abs.htm* 6

Cappé, O., Moulines, E. & Ryden, T. (2005), *Inference in Hidden Markov Models (Springer Series in Statistics)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA. Also available at: www.springerlink.com/content/978-0-387-40264-2. 6

Daniel, R., Lukowicz, P. & Gerhard, T. (2008), 'Hidden markov models in wearable computing - sequence recognition and context awareness', Lecture Script. 6

Gutknecht, J., Kulka, I., Lukowicz, P. & Strieker, T. (2008), 'Advances in Expressive Animation in the Interactive Performance of a Butoh Dance', *Transdisciplinary Digital Art. Sound, Vision and the New Screen: Digital Art Weeks and Interactive Futures 2006/2007, Zurich, Switzerland and Victoria, BC, Canada, Selected Papers* . 6, 8

Kahol, K., Tripathi, P. & Panchanathan, S. (2004), Computational analysis of manner-ism gestures, *in* 'ICPR', pp. III: 946–949.
**URL:** *http://dx.doi.org/10.1109/ICPR.2004.1334685* 6

Kwon, D. Y. & Gross, M. H. (2005), Combining body sensors and visual sensors for motion training, *in* N. Lee, ed., 'Advances in Computer Entertainment Technology', ACM, pp. 94–101.
**URL:** *http://doi.acm.org/10.1145/1178477.1178490* 8

Liu, X. H. & Chua, C. S. (2003), Multi-agent activity recognition using observation decomposed hidden markov model, *in* 'CVS', p. 247 ff.
**URL:** *http://link.springer-ny.com/link/service/series/0558/bibs/2626/26260247.htm* 6

Majoe, D. (2003), Ubiquitous-computing enabled wireless devices, *in* 'Proceedings of the Tenth International Conference on Human-Computer Interaction', Vol. 4 of *Universal access in HCI : inclusive design in the information society*, pp. 444–448. 5

Majoe, D. & Hazzard, N. (1999), Human gesture analysis and recognition for medical and dance applications, *in* 'Proceedings of the Eighth International Conference on Human-Computer Interaction', Vol. 2, pp. 818–822.

Ranieri, N. & Majoe, D. (2007), 'Tai chi chuan using wearable sensors - distribution and interpretation of sensor network data', Semester Thesis. 4, 5, 22, 49, 55

Widmer, L. & Majoe, D. (2008), 'Tai chi chuan using wearable sensors - comparing static poses and providing feedback', Semester Thesis. 4, 5, 10, 13, 35, 43, 49, 57

Widmer, L., Majoe, D., Tschiemer, P. & Gutknecht, J. (2009), Tai chi chuan using wearable sensors - enhancements in accuracy, communications and dynamic motion recognition, Master's thesis, ETH Zürich.

Wu, Y. & Huang, T. S. (1999), Vision-based gesture recognition: A review, *in* 'International Gesture Workshop', p. 103.
**URL:** *http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1739&spage=103* 6

# List of Figures

# Notation

- API: Application programming interface
- BPN: Backpropagation Network (learning algorithm)
- CSV: Comma separated values (actually we use semicolon, not commas)
- DirectX/Direct3D: Microsoft DirectX is a collection of APIs (Direct3D among them for drawing 3D graphics) for handling tasks related to multimedia, especially game programming on Microsoft platforms
- DLL: Dynamic Link Library (shared precompiled library in Windows)
- DNS: Domain Name Service
- FTP: File Transfer Protocol
- SSH: Secure Shell
- GNU: GNU's Not Unix
- GUI: Graphical user interface
- HMM: Hidden Markov Model (learning algorithm)
- IDE: Integrated development environment (e.g. Eclipse, Dev-Cpp, ...)
- IP: Internet Protocol
- MS: Microsoft
- OS: Operating System
- PC: Personal Computer

- SNG-Library: Sensor Network Graphics Library, written by Nico Ranieri.
- TCP: Transfer control protocol (connection based network transmission)
- Ubuntu: The Ubuntu Linux distribution, which is based on Debian Linux.
- UDP: User datagram protocol (packet based network transmission)
- URL: Uniform Resource Locator
- USB: Universal serial bus.
- VC++: Microsoft Visual C++
- VNC: Virtual Network Computing
- XML: eXtensible Markup Language

# Index