Master's Thesis

Adaptive Object Tracking for Service Robots

submitted by
Germán Martín García, Matr. Nr. 2206148

at the
Rheinische Friedrich-Wilhelms-Universitaet Bonn
Department of Computer Science III

Contents

1 Introduction **1**

1.1 The Challenge of Tracking 1

1.2 Scientific Contribution 2

1.3 Related Work 3

2 Scientific Background **5**

2.1 AdaBoost 5

2.2 Sequential Probabilistic State Estimation 6

2.2.1 The Bayes Filter 7

2.2.2 The Condensation Algorithm 9

2.3 Scalable Gradient Features 10

2.3.1 Feature Definition 10

2.3.2 Feature Computation 11

2.3.3 The Bin-Gauss Classifier 12

2.4 Features from Surface Normals 15

2.4.1 The Surface Patches 16

2.5 The Surflet-Pair Relation Feature 16

2.6 Adaptive Visual Tracker For Arbitrary Objects 18

2.6.1 The Particle Filter 19

2.6.2 The Object Classifier 20

2.6.3 The Static Feature Pool Initialisation 20

2.6.4 The General Algorithm 20

3 Adaptive Tracking Using Multiple Appearance Modalities **24**

3.1 3D Particle State 24

3.1.1 The 3D State Space and Motion Model 24

3.1.2 Restriction of the Particles' Positions 25

3.1.3 Projection of the Particles onto the Image Plane 26

3.2 Scalable Depth Gradient Features . . . 28
3.2.1 The Depth Image . . . 29
3.2.2 Feature computation . . . 29
3.3 Colour Features . . . 30
3.3.1 The Colour Space . . . 30
3.3.2 Feature Computation . . . 30
3.3.3 The Classifier . . . 31
3.4 Surface Curvature Features . . . 31
3.4.1 Feature Description . . . 31
3.4.2 The Grid Classifier . . . 32
3.5 Adaptive Feature Pool . . . 35
3.5.1 The Dynamic Feature Pool Initialisation . . . 35
3.5.2 Feature Error Measurement . . . 35
3.5.3 Pool Update . . . 36
3.6 Robot Control . . . 38

4 Hardware and Software Environment **40**
4.1 The Kinect RGB-Depth Sensor . . . 40
4.1.1 Characteristics of the sensor . . . 41
4.1.2 The OpenNI Driver . . . 41
4.1.3 Modification of the OpenNI kinect driver . . . 43
4.2 Rhino . . . 43
4.2.1 Coordinate frames . . . 43
4.3 System Design . . . 44
4.4 General Structure of the System . . . 44

5 Evaluation **46**
5.1 Measurements . . . 46
5.2 Hardware Setup . . . 46
5.3 *BoBoT-D* Benchmark . . . 47
5.3.1 Test Set 1: Colour, Depth and Grayscale Features . . . 49
5.3.2 Test Set 2: Colour, Depth, Grayscale and Surface Normal Features . . . 73
5.3.3 Test Set 3: Dynamic Feature Pool Behaviour Disabled with Colour, Depth and Grayscale Features . . . 83
5.3.4 Test Set 4: Depth and Grayscale Features . . . 88

5.3.5 Test Set 5: Colour and Grayscale Features 89
5.3.6 Test Set 6: Colour and Depth Features 89
5.3.7 Test Set 7: 2D State Space and Grayscale Features 92
5.4 *BoBoT* Benchmark 93
5.4.1 Test case 1: Dynamic Feature Pool Disabled 93
5.4.2 Test case 2: Dynamic Feature Pool Enabled 94
5.5 On-board Tracking Experiment 96

6 Summary **98**
6.1 Future Work 99

A Software Environment **100**
A.1 Robot Operating System (ROS) 100
A.1.1 Program Structure in ROS 100
A.1.2 Communication Between Nodes 100
A.1.3 Launching Nodes 101
A.1.4 Rhino Description in ROS 102
A.1.5 Coordinate Frames and the *tf* Package 102
A.2 Running the tracker 102
A.2.1 Free-Hand Camera Mode 102
A.2.2 Onboard Tracker 103
A.2.3 Evaluation mode 103
A.2.4 Parameters 104

B System Design **107**
B.1 The Tracker Node 107
B.1.1 Pre-processing stage: the Examples class 107
B.1.2 The Classifier Family of Classes 111
B.1.3 The strong classifier 111
B.1.4 The WeakClassifierPool Class 118
B.1.5 The Feature Classes 119
B.1.6 The WeakClassifier Class 120
B.1.7 Particle Related Classes 121
B.1.8 The *Particle* Class 121
B.1.9 The *Particle3D* Class 123
B.1.10 The *ParticleCloud3D* Class 124
B.1.11 The *Tracker* Class 126

B.2 Pan-tilt Control Node 126
B.2.1 The *ctrl* Class 129
B.3 Driver Node 129
B.4 Plotter Node 130
B.4.1 Files Generated and Format 130

Chapter 1

Introduction

1.1 The Challenge of Tracking

The challenge of object tracking could be generally and briefly defined as that of determining the state of the target of interest among all the available data over time. The target's state is usually denoted as x and can consist of magnitudes such as position, velocity, acceleration, etc.. The data is usually denoted as z and directly depends on the kind of sensor[s] being used; its nature will constrain the approach used to solve the problem of tracking. For example, when tracking jet fighters by means of radar data, some effort has to be put into the use of motion models. Instead, tracking objects in the image plane when using a video camera as the source of data leads to deeper thinking into the observation model.

One decision to be made is what features will be used for tracking. When dealing with standard video camera data one can choose among grayscale features, colour features, etc. Recently, very cheap sensors have been released commercially that provide visual and depth information at the same time. Such sensors give a good opportunity to exploit this available data for tasks such as object detection, localization and tracking as in the present work.

Tracking can be useful in many kinds of applications. The ones that quickly come into mind are those of surveillance or target tracking for military uses. The applications in the field of robotics are various; one could think of a mobile platform using the tracker to guide its motion towards a target. Or a robotic arm that uses the target information to grasp, manipulate and move the object. This would be particularly useful in the context of domestic robots, when performing tasks such as grabbing and moving objects in a house environment. In the field

of soccer robots, a tracker can give valuable information about the situation of the opponents as well as the position and movement of the ball. There would be applications as well in human computer interaction: a system could track the human body that is standing in front of it and communicating orders with the use of body gestures.

1.2 Scientific Contribution

The tracker of this thesis is an extension of that of [Dominik A. Klein and Cremers, 2010]. At its core is the condensation algorithm and its key strength is an adaptive observation model. The observation model is constructed by boosting features from a feature pool. The features were originally scalable gradient features applied on the grayscale plane. The pool will be made more heterogeneous by extending the feature types to the depth and colour layers, as well as a sort of shape feature defined on the point cloud data.

The number of features of each kind in the pool will be dynamically adjusted according to the average error made by each of the kinds. The purpose of this is that the system adapts to the different conditions in which the tracker can find itself: when tracking objects that stand out more in some of the feature types, the pool should contain more features of the kind that turned out to be more helpful in distinguishing the target from the background and other objects that populate the environment. Finally the tracker will be embedded in a robotic platform that is able to follow at some distance a target of interest. For this reason the first part of this work involves deriving a 3D state representation of the target, to improve the prediction part of the algorithm and ease the control command loop to approach a target.

The present work is structured as follows: Chapter 2 goes through the scientific foundations that were used. Chapter 3 covers the contributions of this thesis. An overview on the design of the system and the hardware environment is given in 4. Further details can be found in Appendix A . The evaluation of the system is contained in Chapter 5; the different experiments and results will be thoroughly discussed.

1.3 Related Work

The topic of object tracking in the context of computer vision has been tackled from many different perspectives in the past twenty years. A general view of the problem and a classification of the different approaches that have been taken can be seen in [Yilmaz et al., 2006]. One big category that comprises many of the proposed solutions is that of Bayesian state estimation; in this framework one considers the state of the object of interest as a random variable x whose state we want to infer, given a sequence of measurements z over time, in the form of a conditional probability density function $p(x|z)$.

One very significant progress was the Condensation algorithm of [Isard and Blake, 1998a]; they presented an algorithm that represented these conditional pdfs as collections of samples, and allowed the tracking of objects via parametrised curve features. This work was extended in [Isard and Blake, 1998b] by incorporating the notion of importance sampling to the sampling stage of the algorithm: a technique used to draw samples from regions that looked more promising as suggested by an importance function.

One category of trackers includes those that use a model of the object. In [Giebel et al., 2004] a representation of the target's shape is learnt in a training stage in the form of Dynamic Point Distribution Models; shape, texture and depth cues are integrated in the state propagation in a particle filter. In the work of [Wu and Nevatia, 2007] we find a fixed model of humans that is used to detect the different parts of persons' bodies independently, and thus allows them to deal with occlusions.

In [Avidan, 2007] an ensemble of weak classifiers are boosted into a strong one to classify every pixel as either object or background. The mean-shift algorithm [Fukunaga and Hostetler, 1975] is then used to find the likelihood peak of the foreground pixels, that serves as an estimate of the object's position. In this thesis, boosting of features is also done to learn the appearance of the object; the difference is that the features are defined over regions in the image, and not pixel-wise. A component-based tracker was introduced in [Simone Frintrop and Schulz, 2010]; high contrast components in intensity and colour channels are found and integrated in a descriptor. This descriptor is then used as part of the observation model of a particle filter.

Our present work is very much related to that of [Dominik A. Klein and Cremers, 2010] and [Klein and Cremers, 2011]. The

first one shows a real-time adaptive tracker that boosts Haar-like features to form a strong classifier, which classifies hypotheses in the image as target or background; in the latter this work was continued with the introduction of region-wise gradient features. Section 2.6 covers the details of this work, since it is the base of the present thesis.

With respect to 3D information of the environment, RGB-D cameras have become very popular since they provide visual and depth information simultaneously. In [Holzer11] it was shown how to use integral images for the computation of 3D surface normals in constant time.

Chapter 2

Scientific Background

2.1 AdaBoost

AdaBoost was introduced in [Freund and Schapire, 1995]. It is defined as a meta-learning algorithm in the sense that it takes as part of its input a collection of weak classifiers together with the training data and produces a strong classifier, as a linear combination of the weak ones. The intuition behind it is to use many simple rules that classify the data slightly better than by making random decisions. Some of these rules are combined into one single strong classifier that is able to classify the data very precisely.

Formally speaking, let the training data be the set $\{(x_1, y_1)...(x_N, y_N)\}$ where $x_i \in \mathcal{X}$ is the instance space and $y_i \in Y = \{+1, -1\}$ is the label set. The algorithm works by keeping a weight distribution over the training examples; intuitively this distribution tells how hard it is to correctly classify an example at each round of the algorithm. This distribution is denoted by $D_t(i)$ meaning what weight corresponds to the $i-th$ example at round t of the algorithm. Algorithm 1 is a sketch of a variant of the algorithm known as Gentle AdaBoost:

About the performance of the algorithm, it can be shown that as long as the individual classifiers make an error below the one made by random decisions, the error of the strong classifier exponentially decays with the number of rounds, being bounded by:

$$error_{total} \leq \prod_t 2\sqrt{\epsilon_t(1-\epsilon_t)} \leq e^{-2\sum_t (1-\epsilon_t)^2}$$

Another source of information about boosting and its properties can be found in [Freund and Schapire, 1999].

Algorithm 1 AdaBoost

Require: • Labelled training set $\mathbf{T} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ with $x_i \in \mathcal{X}$, $y_i \in \{1, -1\}$
• Probability distribution $D_0 = \{w_1^0, \ldots, w_N^0\}$ about $\mathbf{T}$
• Set of feature functions $\mathbf{F} = \{f_1, \ldots, f_M\}$ with $f_m : X \to \mathbb{R}^{\sim}$
• WEAK-LEARN procedure: input ($\mathbf{T}$, D, f_m), output $h_m^k : X \to [1, -1]$, an optimal weak classifier based on f_m of the kind k

1: Initialize the weights of training examples with $w_i^0 \leftarrow \frac{1}{N}, i = 1 \ldots N$
2: **for** $(t = 0; t \leq T; t{+}{+})$ **do**
3: Calculate by WEAK-LEARN the optimal weak classifiers for the current distribution D_t
4: select weak classifier h_t^k of kind k that minimizes squared classification error $\epsilon_t = \sum_{i=1}^{N} w_i (y_i - h_t^k(x_i))^2$.
5: update the weights: $w_i^{t+1} \leftarrow w_i^t \exp(-y_i h_t(x_i))$
6: normalize the weights: $w_i^{t+1} \leftarrow \frac{w_i^{t+1}}{\sum_{j=1}^{N} w_j^{t+1}}$
7: Add h_t^k to the strong classifier $H_t(x) := H_{t-1}(x) + h_t^k(x)$
8: **end for**
9: **return** the strong classifier $H_T(x)$

2.2 Sequential Probabilistic State Estimation

The goal of probabilistic state estimation is to estimate the state of a system over time from the available data. The internal state of the system and the observable data are treated as random variables and the belief on the state given the data is modelled as a conditional probability density function. The state of the system is denoted generally as X and its value at time k is x_k . The observable data is Z and its value at each time step is denoted as z_k. To define the problem more precisely, given the set of all available data $z_{1:t}$ up to time t we are interested in calculating the conditional probability density function $p(X_t = x_t | Z_{1:t} = z_{1:t})$. This pdf captures information about what is the probability of each possible state of system given the observed data.

The assumption is made that the conditional pdf at time k only depends on the immediately preceding value at time $k - 1$. This is known as the Markov Assumption. The Bayes filter equations give a general solution to the problem of calculating 2.2 in a recursive manner:

$$p(x_t | z^{t-1}) = \int p(x_t | x_{t-1})\ p(x_{t-1} | z^{t-1})\ dx_{t-1} \tag{2.1}$$

$$p(x_t | z^t) = \eta\ p(z_t | x_t)\ p(x_t | z^{t-1}) \tag{2.2}$$

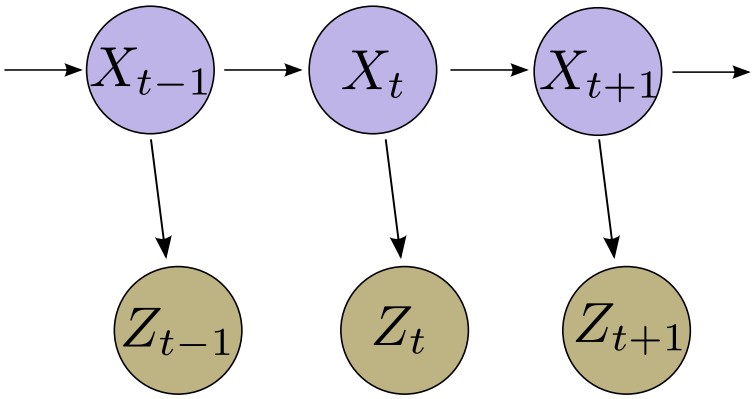

Figure 2.1: A graphical representation of the state estimation problem

In the following sections we define all the elements used in this framework and make a derivation of the equations of the Bayes filter.

Figure 2.1 shows a graphical representation of the problem. The state of the system is to be inferred at time t, represented by the variable X_t. The observations made at each time step are given by the variables Z_t. The arrows go from each state X_k to the following X_{k+1} , meaning that the present state is only dependent on that of the previous time step. They also link states to observations of the same time step denoting a conditional dependence between the two.

2.2.1 The Bayes Filter

Given the time dependent observable data Z_k we want to express the knowledge about the state random variable X_k by means of a conditional pdf of the form $p(X_t = x_t | Z_{1:t} = z_{1:t})$. The Bayes filter equations provide a recursive way for computing this pdf at each time step in terms of the previous time step.

$$p(x_t|z_{1:t}) = p(x_t|z_t, z_{1:t-1}) \tag{2.3}$$

By applying the Bayes rule we get the following:

$$= \frac{p(z_t|x_t, z^{t-1})\ p(x_t|z^{t-1})}{p(z^t)} \tag{2.4}$$

$$= \frac{p(z_t|x_t, z^{t-1})\ p(x_t|z^{t-1})}{\int \mathrm{d}x_t\, p(x_t|z^t)\ p(z^t)} \tag{2.5}$$

$$= \frac{p(z_t|x_t, z^{t-1})\ p(x_t|z^{t-1})}{\int\ \mathrm{d}x_t\ \frac{p(z_t|x_t, z^{t-1})\ p(x_t|z^{t-1})}{\cancel{p(z^t)}}\ \cancel{p(z^t)}} \tag{2.6}$$

$$= \frac{p(z_t|x_t, z^{t-1})\ p(x_t|z^{t-1})}{\int \mathrm{d}x_t\, p(z_t|x_t, z^{t-1})\ p(x_t|z^{t-1})} \tag{2.7}$$

Focusing on the term $p(x_t|z^{t-1})$, the knowledge about the previous state is integrated as follows:

$$p(x_t|z^{t-1}) = \int \mathrm{d}x_{t-1}\, p(x_t, x_{t-1}|z^{t-1}) \tag{2.8}$$

$$= \int \mathrm{d}x_{t-1}\, p(x_t|x_{t-1}, z^{t-1}) p(x_{t-1}|z^{t-1}) \tag{2.9}$$

Here the Markov assumption allows for the following simplification:

$$p(x_t|x_{t-1}, z^{t-1}) = p(x_t|x_{t-1}) \tag{2.10}$$

since the previous state x_{t-1} captures all the information needed to know the current state x_t, all the past measurements z^{t-1} give no additional information. We will now plug in the result from 2.9 into 2.7 yielding the following:

$$p(x_t|z^t) = \frac{p(z_t|x_t)\ \int \mathrm{d}x_{t-1}\, p(x_t|x_{t-1})\ p(x_{t-1}|z_{t-1})}{\int \mathrm{d}x_t\, p(z_t|x_t) \int \mathrm{d}x_{t-1}\, p(x_t|x_{t-1})\ p(x_{t-1}|z_{t-1})} \tag{2.11}$$

As can be seen, the denominator of the right term of the equation above is nothing but a normalization term over all the possible values of the state variable x_t . Therefore, the notation can be simplified and denote the normalization term as η giving:

$$p(x_t|z^t) = \eta p(z_t|x_t) \int \mathrm{d}x_{t-1}\, p(x_t|x_{t-1})\ p(x_{t-1}|z_{t-1}) \tag{2.12}$$

The posterior pdf $p(x_t|z_{1:t})$ is expressed in terms of the posterior conditional pdf of the previous iteration of the algorithm $p(x_{t-1}|z_{1:t-1})$. The pdf $p(x_t|x_{t-1})$ is known as the transition model, and is directly integrated into the belief of the previous iteration. The previous belief will become more uncertain as a result of applying the motion model and the pdf will spread as a result. The term $p(z_t|x_t)$ allows to update the resulting pdf obtained from applying the motion model. Applying the observation model can be informally read as "how well does the predicted state fit the observed data". It is important to notice that both the transition and the observation model are functions that can be time-dependent.

2.2.2 The Condensation Algorithm

The Bayes filter equations provide us with a general framework, but an important question to answer is how to actually compute them? If one assumes that these pdfs are Gaussian and the motion and observation models are linear, then the solution to each time step posterior is given by the well known Kalman filter equations [Kalman, 1960].

A different approach is that of the particle filter [Gordon et al., 1993], also known as Condensation algorithm in the computer vision community [Isard and Blake, 1998a]. This brand of algorithms is also known as sequential Monte Carlo methods.

The Monte Carlo methods let us represent the posterior conditional pdfs $p(x_t|z_t)$ at each time step by sets of N weighed samples of the form:

$$\{(\pi_t^{(1)}, s_t^{(1)})...(\pi_t^{(n)}, s_t^{(n)})\}$$

where $\pi_t^{(i)}$ is the weight and $s_t^{(i)}$ is the state of the i-th particle/sample at time t.

Algorithm 2 Condensation Algorithm

1: initialize $\pi_1(i) \Leftarrow 1/m$
2: **for** $n = 1..N$ **do**
3: $S_t'^{(n)} \Leftarrow$ draw with replacement $S_t^{(n-1)}$ with probability proportional to $\pi_t^{(n-1)}$ {this step is known as resampling and implies that many of the drawn particles are actually the same}
4: $\pi_t^{(n)} \Leftarrow 1/N$
5: **end for**
6: sample $S_t^{(n)}$ from $p(X_t = S_t^{(n)}|X_{t-1} = S_t'^{(n)})$ {this step is nothing but applying the motion model to all the particles that survived the previous step}
7: $\pi_t^{(n)} \Leftarrow p(Z_t|X_t = S_t^{(n)})$ {weights the particles with the likelihood function}
8: normalize the weights so that $\sum_i^N \pi_t^{(i)} = 1$

Algorithm 2 shows the steps of the Condensation algorithm. Line 3 involves resampling from the prior: this step results in having an equivalent representation of the pdf where all the particles have equal weights. This is a transformation of the pdf representation from a weight distribution to a distribution of frequency. After this step we usually have that many of the drawn particles are actually the same.

In line 6 we sample from the proposal distribution $p(X_t|Z^{t-1})$, this step can be seen as moving the particles according to the motion model. Finally, in line 7

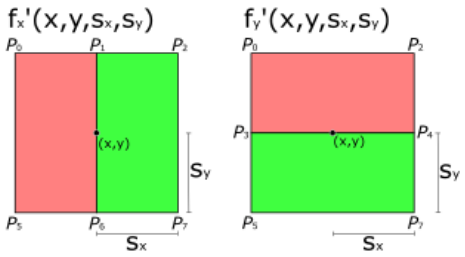

Figure 2.2: Derivations of image regions in x and y dimensions are computed using differences of the means of scalable regions. Obtained from [Klein and Cremers, 2011]

we adjust the weights of the particles. In the previous step we predicted particles from the proposal distribution given by the motion model. We now have to rate them so that the collection of samples follows the target distribution $p(X_t|Z^t)$: this is achieved by weighting the particles with the likelihood function of this measurement given the discriminative object appearance model $p(Z_t|X_t)$. Finally the weights are normalized so that $\sum_{i=0}^{N} \pi_i = 1$.

2.3 Scalable Gradient Features

The region-based scalable gradient visual features were presented in [Klein and Cremers, 2011]. They aim to capture the gradient properties of an image given some position and scale.

2.3.1 Feature Definition

The gradient features have four parameters, namely the $u \in [0..1]$ and $v \in [0..1]$ relative pixel coordinates, and $w \in [0..1]$ and $h \in [0..1]$ the dimensions of the feature rectangle. In our framework, features are defined in relative terms with respect to the positive or negative example on which they are calculated. Each example is at the same time specified as a rectangular region in the image.

Figure 2.3 will help us understand how this works. Features and examples are always defined with respect to its predecessor in the sequence {image, positive or negative example, feature}. The positive and negative examples are defined in relative terms with respect to the complete image. Therefore, the $u_{\text{pos}}, v_{\text{pos}}$ tell us

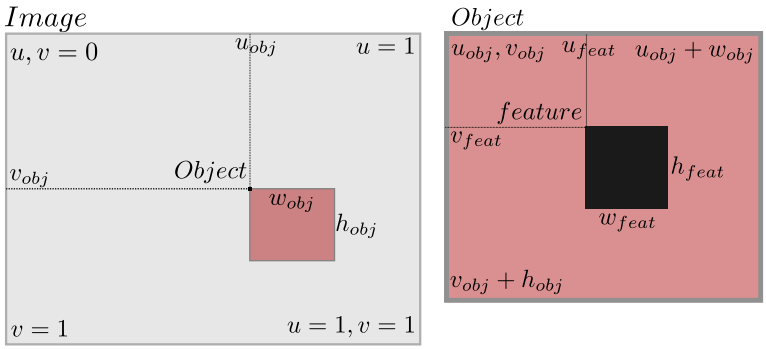

Figure 2.3: Layout of the image plane, positive/negative examples, and features

in relative terms, where in the image the example's upper corner is located. The $w_{\text{pos}}, h_{\text{pos}}$ values, also defined in the [0..1] interval, say how much of the image the example occupies. For example, a width and height of 0.3 and 0.5 respectively would mean that the example height is 30% of the image height and 50% of the image width.

The features are defined with the same mechanism but with respect to the positive and negative examples in which they are contained. Thus, the feature coordinates $u_{\text{feat}}, v_{\text{feat}}$ tell us where in the example's rectangle the feature is located. The same happens with the width and height $w_{\text{feat}}, h_{\text{feat}}$; they say how much of the example dimensions the feature occupies.

2.3.2 Feature Computation

The feature value is a $2D$ gradient, a tuple $(magnitude, orientation)$. To compute it, the partial derivatives on each axis of the image are calculated. Thus, the feature rectangle is divided according to Figure 4.1. The partial derivatives are computed by querying the integral image following the notation of Figure 2.2:

$$\nabla_u = \frac{(p_1 + p_7 - p_2 - p_6)}{n_{u1}} - \frac{(p_0 + p_6 - p_2 - p_5)}{n_{u2}} \tag{2.13}$$

$$\nabla_v = \frac{(p_3 + p_7 - p_4 - p_5)}{n_{v1}} - \frac{(p_0 + p_4 - p_2 - p_3)}{n_{v2}} \tag{2.14}$$

Where $n_u and n_v$ are the number of pixels in each sub-region. All that was done was to subtract the average intensity values of the two subsections of the feature. By using integral images, the averages can be computed on constant

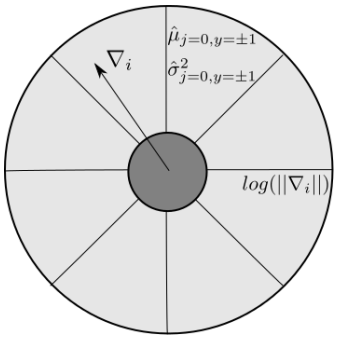

Figure 2.4: The gradient features classifier

time. The result is a vector (∇_u, ∇_v) which for convenience is converted to polar coordinates $(magnitude, orientation)$.

2.3.3 The Bin-Gauss Classifier

The classifier works with the feature values of the positive and negative examples of the training set. It has to classify, according to each feature, how likely it is that some test example is positive.

Learning

The learning stage takes as an input the feature values ∇_i of the positive and negative examples of the training set, together with the weights w_i as chosen by the boosting algorithm. The classifier consists of a center, and a parametrisable number of b outer bins. A gradient is meant to fall into the center bin if its magnitude is too low. The outer bins are equally distributed and discretise all the possible directions. The decision of whether an example falls into the inner or outer bins is fuzzy and depends on two thresholds t_0 and t_1:

$$\begin{aligned} \zeta(x) &= \begin{cases} 0 & \text{, if } t_0 \geq \log(x) \\ \frac{\log(x)-t_0}{t_1-t_0} & \text{, if } t_0 < \log(x) < t_1 \\ 1 & \text{, if } t_1 \leq \log(x) \end{cases} \\ w_{i,\mathrm{o}} &= \zeta(||\nabla_i||) \cdot w_i \\ w_{i,\mathrm{c}} &= (1 - \zeta(||\nabla_i||)) \cdot w_i. \end{aligned} \tag{2.15}$$

Those gradients that are strong enough are assigned to the outer bins. The radial distance of the gradient to the bin centres is calculated, and the weight of

the example is linearly distributed according to this value. We define the radial distance function as

$$\mathrm{dist}(\gamma, j) = \frac{\left(\left|\gamma - \frac{(j+0.5)\cdot 2\pi}{\mathrm{b}}\right|\right)_{\pi}}{(2\pi/\mathrm{b})}$$

Where γ is the queried gradient angle, and $j \in \{0, \ldots, \mathrm{b}\}$ is the bin center. Then, the weight w_{i,O_j} that corresponds to the outer bin o_j is given by

$$w_{i,\mathrm{O}_j} = \begin{cases} 0 & \text{, if } \mathrm{dist}(\alpha_i, j) \geq 1 \\ w_{i,\mathrm{O}} \cdot (1 - \mathrm{dist}(\alpha_i, j)) & \text{, otherwise .} \end{cases} \tag{2.16}$$

We now have the weights that correspond to each bin for this training example. Then, in each partial training set (the positive and the negative), for each bin we fit a log-normal distribution over the weights:

$$f(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp(-\frac{(\log(x) - \mu)^2}{2\sigma^2}) \tag{2.17}$$

The Maximum-likelihood estimator for this distribution is

$$\begin{aligned} \hat{\mu}_{\mathrm{j,y}} &= \frac{\sum_{i:y_i=y} \left[w_{i,\mathrm{O_j}} \log(||\nabla_i||) \right]}{w_{\mathrm{O_j},y}} \\ \hat{\sigma}^2_{\mathrm{j,y}} &= \frac{\sum_{i:y_i=y} \left[w_{i,\mathrm{O_j}} (\log(||\nabla_i||) - \hat{\mu}_{\mathrm{j,y}})^2 \right]}{w_{\mathrm{O_j},y}} . \end{aligned} \tag{2.18}$$

with

$$w_{\mathrm{O_j},y} = \sum_{i:y_i=y} w_{i,\mathrm{O_j}}$$

Note that the logarithm of the magnitude is used instead of the magnitude alone. This was justified in [Klein and Cremers, 2011] for making the features behave better in the contours of an object where the gradients are more unstable from frame to frame.

The Variance Terms

In the actual implementation, the variance terms $\hat{\sigma}^2_{j,y=\pm 1}$ are added an extra value. This has two purposes:

- that the classifier doesn't overfit to the training distribution
- to have a variance value when no examples fall into some bins

The magnitude of this value is fixed empirically. To conclude this section, the steps carried out by the learning algorithm are summarised in algorithm 3.

Algorithm 3 GradientWeakLearn

Require: • Labelled training set $\mathbf{T} = \{(\nabla_1, y_1), \ldots, (\nabla_N, y_N)\}$ with $\nabla_n \in \mathcal{M}$, $y_n \in \{+1, -1\}$
• Probability distribution $D_0 = \{w_1^0, \ldots, w_N^0\}$ about $\mathbf{T}$

1: **for** $(i = 0; t \leq N; i{+}{+})$ **do**
2: calculate center contribution $w_{i,c}$
3: calculate outer contributions w_{i,o_j}
4: **end for**
5: **for** $(j = 0; t \leq b; j{+}{+})$ **do**
6: fit log-normal distribution over positive example results at bin j
7: fit log-normal distribution over negative example results at bin j
8: **end for**

Querying the Classifier

So far we have covered the learning stage of the classifier. For new examples we want to be able to classify them. First, we calculate the probability that the test example belongs to each class (object or background) for each of the bins:

$$\begin{aligned} \mathrm{P_c}(y|x) &= \frac{\sum\limits_{i:y_i=y} w_{i,\mathrm{c}}}{\sum\limits_{a \in Y} w_{\mathrm{c},a}} = \mathrm{P_{c,y}} \quad \text{(const.)} \\ \mathrm{P_{O_j}}(y|x) &= \frac{w_{\mathrm{O_j},y} \cdot f(x; \hat{\mu}_{\mathrm{j},y}, \hat{\sigma}_{\mathrm{j},y})}{\sum\limits_{a \in Y} \left[w_{\mathrm{O_j},a} \cdot f(x; \hat{\mu}_{\mathrm{j},a}, \hat{\sigma}_{\mathrm{j},a}) \right]}. \end{aligned} \tag{2.19}$$

The classifier output is a combination of the center and outer bin probabilities queried:

$$\begin{aligned} j &= \lfloor \frac{\alpha_x \cdot \mathrm{b}}{2\pi} \rfloor \\ \mathrm{P}(y|x) &= \zeta(x) \cdot \mathrm{P_{O_j}}(y = a|x) + (1 - \zeta(x)) \cdot \mathrm{P_{c,a}}. \end{aligned} \tag{2.20}$$

The classifier is represented in figure 2.4.

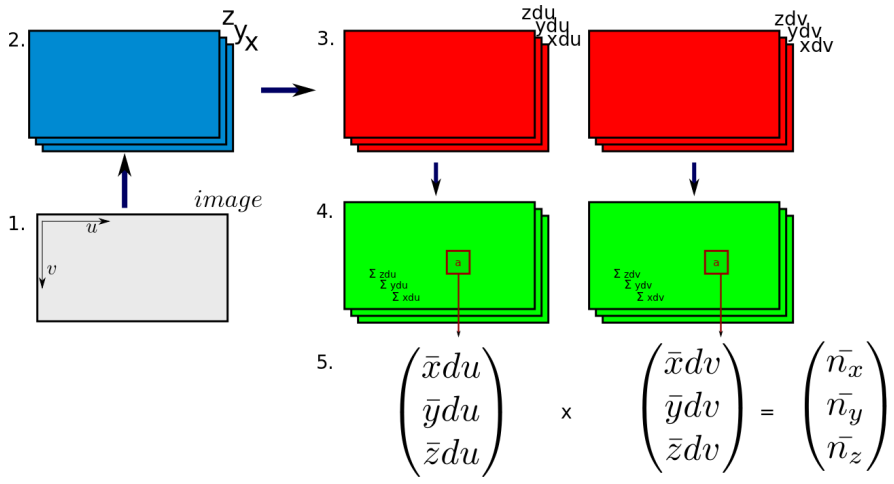

Figure 2.5: The steps of the algorithm for the computation of surface normals as detailed in [Dirk Holz and Behnke, 2011]

2.4 Features from Surface Normals

A surface normal is the vector orthonormal to the plane or surface over which it is computed. A plane in a three dimensional space can be defined by two linearly independent vectors. The cross product of such two vectors gives another vector orthonormal to the plane they define.

In this section we show how to compute normals to surface patches as in the work of [Dirk Holz and Behnke, 2011]. The process involves some pre-computations that will let us query surface normals over surface patches in constant time. The pre-processing stage is represented in figure 2.5.

In step 1. every pixel (u, v) in the image has in 2. its corresponding $[x, y, z]$ 3D coordinates; they are represented here as three different layers. The derivatives of x,y, and z in the u and v dimensions of the image are calculated. This leads to having six derivative images in step 3. This is implemented with a derivative filter that computes differences of 3D positions between adjacent pixels in each of the 3D and image dimensions. To clarify this step, let xdu_img, ydu_img, zdu_img, xdv_img, ydv_img, zdv_img be the six derivative images, and x_img, y_img, z_img the images containing the 3D values of step 2. Then each pixel in the derivative images is calculated as follows:

$$xdu\_img(u, v) = x\_img(u, v) - x\_img(u - 1, v) \tag{2.21}$$

$$ydu\_img(u,v) = y\_img(u,v) - y\_img(u-1,v) \tag{2.22}$$

$$zdu\_img(u,v) = z\_img(u,v) - z\_img(u-1,v) \tag{2.23}$$

$$xdv\_img(u,v) = x\_img(u,v) - x\_img(u,v-1) \tag{2.24}$$

$$ydv\_img(u,v) = y\_img(u,v) - y\_img(u,v-1) \tag{2.25}$$

$$zdv\_img(u,v) = z\_img(u,v) - z\_img(u,v-1) \tag{2.26}$$

The integral images of the six derivatives are computed in step 4. Here concludes the pre-processing stage. To obtain the normal vector over a region, the average derivative vectors $(x\bar{du}, y\bar{du}, z\bar{du})^T$,$(x\bar{dv}, y\bar{dv}, z\bar{dv})^T$ are calculated by querying the integral images. The cross product of them is the surface normal $(\bar{n_x}, \bar{n_y}, \bar{n_z})^T$.

2.4.1 The Surface Patches

The surface patches define the region over which the normal is calculated. They consist of points corresponding to a rectangle in the image. It is important to note that adjacent pixels in the image need not be adjacent points in the cloud. Outlier points can therefore be included in the normal vector computation, but their effect is minimized through the averaging step done over the whole region.

2.5 The Surflet-Pair Relation Feature

In this section we cover the four-dimensional feature introduced in [Wahl et al., 2003] for shape representation. The feature is calculated on pairs of surface normals, and consists of three angles and a distance measure.

Given the points p_1 ,p_2 on which the normals $\vec{n_1}$,$\vec{n_2}$ are calculated, we first choose a reference point/vector such as:

- choose p_1 if $|n_1 \cdot (p_2 - p_1)| < |n_2 \cdot (p_2 - p_1)|$
- choose p_2 if $|n_2 \cdot (p_2 - p_1)| < |n_1 \cdot (p_2 - p_1)|$

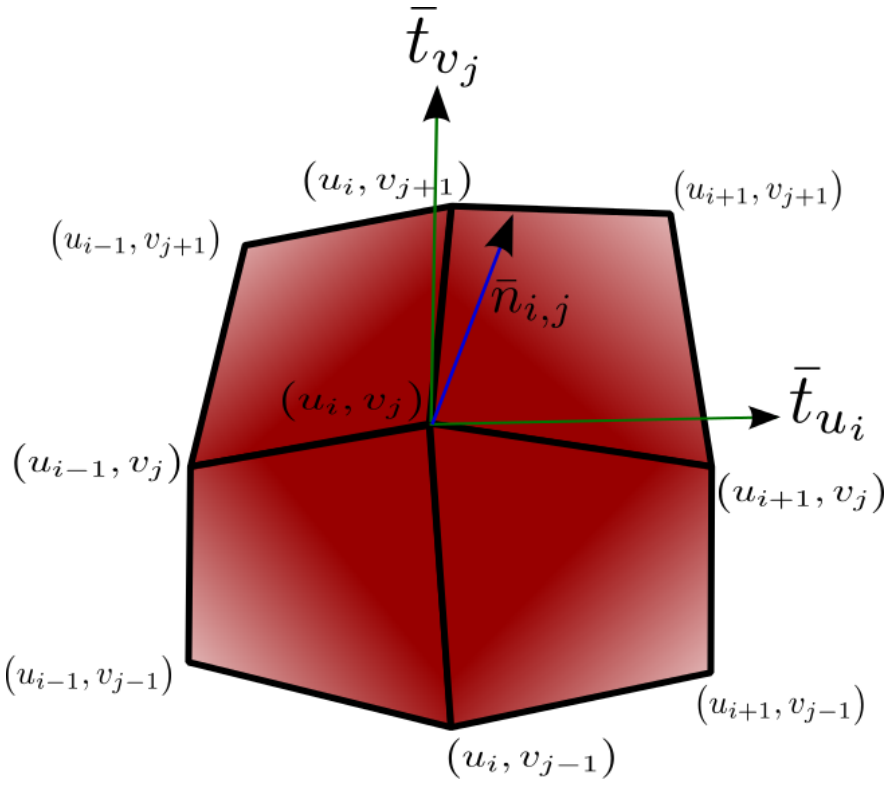

Figure 2.6: Each crossing represent a pixel in the image (u_i, v_i) which has some 3D coordinates $[x_p, y_p, z_p]$. the green vector are the tangents computed in each of the image axis directions. The surface normal (in blue) is the cross product of the two tangents $\vec{t_{ui}}$ and $\vec{t_{vj}}$.

Assuming $\vec{n_1}$ was taken as the reference normal, the vectors $\vec{u}$,$\vec{v}$ and $\vec{w}$ are chosen as:

$$\vec{u} = \vec{n_1} \tag{2.27}$$

$$\vec{v} = \frac{p_2\vec{p}_1 \times \vec{u}}{||p_2\vec{p}_1|| \times \vec{u}} \tag{2.28}$$

$$\vec{w} = \vec{u} \times \vec{v} \tag{2.29}$$

For simplicity we denote the vector going from point p_1 to p_2 as $p_2\vec{p}_1$. The feature values are formed by the tuple $(\alpha, \beta, \gamma, \delta)$ and are computed as follows:

$$\alpha = atan2(\vec{w} \cdot \vec{n_2}) \tag{2.30}$$

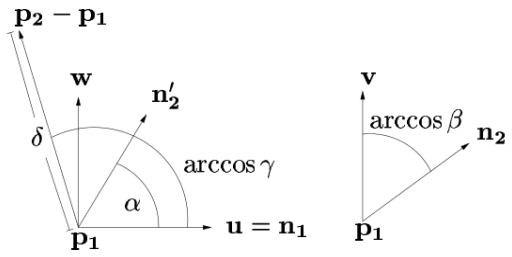

Figure 2.7: The feature angles. The figure was taken from [Wahl et al., 2003]. The n'_2 vector is the projection of n_2 on the vw plane

$$\beta = \vec{v} \cdot \vec{n_2} \tag{2.31}$$

$$\gamma = \vec{u} \cdot p_2\vec{p}_1 \tag{2.32}$$

$$\delta = |p_2 - p_1| \tag{2.33}$$

Figure 2.7 illustrates the four dimensions of the feature.

2.6 Adaptive Visual Tracker For Arbitrary Objects

This section covers the adaptive real-time capable tracker for arbitrary objects of [Dominik A. Klein and Cremers, 2010] and [Klein and Cremers, 2011]. It is the starting point of this thesis and the new contributions have been added to it.

The core of the tracker is the Condensation algorithm; it keeps a set of particles that estimate the location and appearance of the target over time. The object's appearance is learnt as an ensemble of boosted classifiers. In [Dominik A. Klein and Cremers, 2010] these were threshold classifiers defined over Haar-like grayscale features; in [Klein and Cremers, 2011] the new scalable gradient features and their classifier were introduced into the boosting framework. At frames where it is confident enough, the classifier is re-trained to be able to cope with new appearances of the target. In the following subsections we will go through the details of the algorithm.

2.6.1 The Particle Filter

The core of the system is a particle filter that approximates a probability density function $p(X_k|Z^k)$ of the possible states of the target given the observations over time. Such pdf is approximated by maintaining a set of particles or samples $S_k = \{s_k^j\}$ with $j \in \{1..J\}$ at each time step $t = k$. Every particle is defined as $s_k^j = \{\pi_k^j, x_k^j\}$ where π_k^j is the weight and x_k^j its state. The general steps of the process are defined in Algorithm 4.

The Particles' State Space

The state space of the particles consists of two pixel coordinates u and v and their first derivatives $\dot{u}$ and $\dot{v}$; the width w and height h of the bounding box containing the object as well as their first derivatives $\dot{w}$ and $\dot{h}$; and finally the target classifier C:

$$x_k^j = \{u_k^j, v_k^j, \dot{u}_k^j, \dot{v}_k^j, w_k^j, h_k^j, \dot{w}_k^j, \dot{h}_k^j, C_k\} \tag{2.34}$$

The Motion Model

Applying the motion model involves adding white noise to the particle's pixel coordinate and bounding box size's current velocities $\dot{u}, \dot{v}, \dot{w}, \dot{h}$.

$$\dot{u_k} = \dot{u_{k-1}} + \mathcal{N}(0, \sigma_{uv}^2), \tag{2.35}$$

$$\dot{v_k} = \dot{v_{k-1}} + \mathcal{N}(0, \sigma_{uv}^2), \tag{2.36}$$

$$\dot{w_k} = \dot{w_{k-1}} + \mathcal{N}(0, \sigma_{wh}^2), \tag{2.37}$$

$$\dot{h_k} = \dot{h_{k-1}} + \mathcal{N}(0, \sigma_{wh}^2). \tag{2.38}$$

Then the pixel coordinates and bounding box sizes are updated taking into account the current velocities at time k, some fixed time interval, and the values of the previous time step:

$$u_k = u_{k-1} + \Delta t \times \dot{u_k}, \tag{2.39}$$

$$v_k = v_{k-1} + \Delta t \times \dot{v_k}, \tag{2.40}$$

$$w_k = w_{k-1} + \Delta t \times \dot{w_k}, \tag{2.41}$$

$$h_k = h_{k-1} + \Delta t \times \dot{h_k}. \tag{2.42}$$

2.6.2 The Object Classifier

The object appearance is captured in an ensemble of boosted weak classifiers; a binary classifier is trained to distinguish the object from the background. For the purpose of this section we will refer to the scalable gradient features and their classifier of [Klein and Cremers, 2011] explained in section 2.3.

A pool of features of different sizes and positions is defined before the tracking process begins. These features are specified in relative terms with respect to the positive and negative examples on which they will be evaluated. More insight in how this is achieved can be found in Section 2.3.1. The boosting algorithm will evaluate these features over the positive and negative examples and select those that together better classify the object from the background. The number of selected weak classifiers is a parameter that can be tuned. The boosting algorithm is carefully detailed in Section 2.1.

The object classifier C returns a confidence value between 0 and 1 of the object being the target. It is then embedded into an exponential function that is used as a likelihood function when applying the observation model in the particle filter algorithm.

2.6.3 The Static Feature Pool Initialisation

The pool contains the features that remain constant throughout the tracking process. They are defined in relative coordinates with respect to the width/height ratio of the area of the first positive training example. Features of different dimensions are equally distributed over this area.

2.6.4 The General Algorithm

For the initialisation of the tracking process the user provides the starting position and dimensions of the target in the image; line 1 of Algorithm 4. This becomes the

first positive example; negative examples are randomly drawn from the image, line 4, and the classifier is trained, line 6. The particle cloud is initialised in line 11. The same particle is replicated with the initialisation values and zero velocities.

The same iterative procedure now starts for every incoming frame. In line 16, the motion model step of the particle filter is applied. The particles' positions, dimensions and velocities are updated according to equations 2.35 to 2.42. The observation model is then applied and the particles are rated according to the classifier learnt in the previous iteration, line 20. The rate is exponentiated to become the likelihood function $p(Z_k|X_k)$ of the observation model.

If the confidence of the best particle, and the weighted overlap is above some predefined thresholds, then a new positive example is added to $posExamples$ and the classifier is retrained, line 29. The computation of the weighted overlap is done in 5; given the area defined by the best particle, the overlap percentage is computed with respect to every other particle and weighted with their weights.

Finally, a re-sampling step is executed; this produces the effect that the particles with lower confidence will die out, while those with higher confidence values will last and replicate themselves; thus keeping more particles where the target is more likely to be.

Algorithm 4 Adaptive Tracker

Require: • Initial state of the target $x_0 = \{u_0, v_0, w_0, h_0\}$
• The training set of positive examples $posExamples$
• The training set of negative examples $negExamples$

$posExamples \leftarrow \{u_0, v_0, w_0, h_0\}$
while not enough negative examples **do**
  $\{u_{rand}, v_{rand}, w_{rand}, h_{rand}\} \leftarrow generateNegExample()$
  $negExamples \leftarrow \{u_{rand}, v_{rand}, w_{rand}, h_{rand}\}$
end while
$C_0.train(posExamples, negExamples)$
for $(j = 0; j \leq J; j{+}{+})$ **do**
  $\pi_0^j = 1/J$
  $x_0^j = \{u_0, v_0, \dot{u} = 0, \dot{v} = 0, w_0, h_0, \dot{w} = 0, \dot{h} = 0, C_0\}$
  $s_0^j = \{pi_0^j, x_0^j\}$
  $S_0 \leftarrow S_0 + s_0^j$
end for
while frame k is available **do**
  {applies the motion model of the particles}
  for $(j = 0; j \leq J; j{+}{+})$ **do**
    apply the motion model to particle $s_k^j \in S_k$
  end for
  $max\_conf \leftarrow 0$
  {rates the particles}
  for $(j = 0; j \leq J; j{+}{+})$ **do**
    $conf_k^j = C_{k-1}(x_k^j)$
    if $conf_k^j > max$ **then**
      $s_k^{best} \leftarrow s_k^j$
    end if
    $rate_k^j = \exp(\lambda \times conf_k^j)$
  end for
  {updates the classifier}
  $a^{best} \leftarrow \{u_k^{best}, v_k^{best}, w_k^{best}, h_k^{best}\}$
  if $getSupport(S_k, a^{best}) > suppThreshold$ AND $conf_k^{best} > confThreshold$ **then**
    $posExamples \leftarrow posExamples + \{u^{best}, v^{best}, w^{best}, h^{best}\}$
    $C_k \leftarrow C_{k-1}.train(posExamples, negExamples)$
  end if
  {resampling stage}
  $step \leftarrow 1/J$
  $w \leftarrow random() \times step$
  $S'_k \leftarrow \{\}$
  for $(j = 0; j \leq J; j{+}{+})$ **do**
    while $w_k^{p \mod n} < w$ **do**
      $w \leftarrow w - w_k^{p \mod n}$
      $p \leftarrow p + 1$
    end while
    $S'_k \leftarrow S'_k + s_k^{p \mod n}$
    $w \leftarrow w + step$
  end for
end while

Algorithm 5 getSupport

Require: • Set of particles $S_k = \{s_k^j\}$
• Area $a = \{u_a, v_a, w_a, h_a\}$

1: $supp = 0.0$, $sum = 0.0$
2: **for** $(j = 0; j \leq J; j{+}{+})$ **do**
3: $a_k^j \leftarrow u_k^j, v_k^j, w_k^j, h_k^j$
4: $supp \leftarrow supp + w_k^j * area\_overlap(a, a_k^j)$
5: $sum \leftarrow sum + w_k^j$
6: **end for**
7: **return** $supp/sum$

Chapter 3

Adaptive Tracking Using Multiple Appearance Modalities

3.1 3D Particle State

The former approach to defining the state of the particles can be found in section 2.6.1. This model worked reasonably well as one can see in the evaluation chapters of [Dominik A. Klein and Cremers, 2010] and [Klein and Cremers, 2011], but it had to deal with all the nonlinearities that arise from real world objects when projected to the image plane in an indirect way. This motivated developing a 3D state space and motion model for the particles that would deal with this nonlinearities in a more precise manner.

3.1.1 The 3D State Space and Motion Model

Rather than a 2D or image plane state space we shall now define a 3D state space and motion model for the particles. We refer to it for convenience as 2D for the two dimensions of the image plane, and 3D for the three dimensions of our world, but note that the dimensionality of the two spaces is higher. The state space now looks like the following:

$$x = \{x, y, z, \dot{x}, \dot{y}, \dot{z}, W, H\} \tag{3.1}$$

Thus, we have 3 spatial coordinates x,y,z, their respective first derivatives $\dot{x}$,$\dot{y}$,$\dot{z}$, and the dimensions of the bounding box around the object W and H. We will refer to the dimensions of the bounding box in the 3D world with capital

letters and with small letters when talking about its projections onto the image plane.

The motion model becomes the following:

$$\dot{x_k}' = \dot{x_{k-1}} + \mathcal{N}(0, \sigma_{xy}^2), \tag{3.2}$$

$$\dot{y_k}' = \dot{y_{k-1}} + \mathcal{N}(0, \sigma_{xy}^2), \tag{3.3}$$

$$\dot{z_k}' = \dot{z_{k-1}} + \mathcal{N}(0, \sigma_{z}^2), \tag{3.4}$$

$$x'_k = x_{k-1} + \Delta t \times \dot{x_k}', \tag{3.5}$$

$$y'_k = y_{k-1} + \Delta t \times \dot{y_k}', \tag{3.6}$$

$$z'_k = z_{k-1} + \Delta t \times \dot{z_k}'. \tag{3.7}$$

In the first three equations the particle's speeds are updated by adding white noise. We let the variance in the z axis be different than that of the x and y axes. Equations 3.5 to 3.7 use the velocities to update the positions of the particles. We let the positions and speeds have a prime symbol, since there will be a further update to them when the observation model is applied.

3.1.2 Restriction of the Particles' Positions

The RGB-D sensor provides spatial information attached to every pixel in the image [1]. We use the information provided by the point cloud to restrict the allowed positions of the particles. This has the effect of reducing the dimensionality of the state state space of particles by one: from the three degrees of freedom x,y,z we are constrained to the surface that is visible from the camera.

In Figure 3.1 we have a visualization of the point cloud and the particles. While most of the particles are concentrated on the object, some of them moved to the sides of the object where there is empty space; this has the effect that the particles are pushed in 3D space to points that are occupied by the point

[1] As explained in section 4.1 not every pixel in the image has a depth value due to hardware constraints

cloud. Equations 3.8 to 3.13 show how the positions and velocities are updated. Although these updates affect the motion of the particles, we do not include them in the motion model since they require the sensed data at the same time step.

$$x_k = restrict\_to\_cloud(x'_k, z_k), \tag{3.8}$$

$$y_k = restrict\_to\_cloud(y'_k, z_k), \tag{3.9}$$

$$z_k = restrict\_to\_cloud(z'_k, z_k), \tag{3.10}$$

$$\dot{x_k} = \dot{x_k} + \frac{x_k - x'_k}{\Delta t}, \tag{3.11}$$

$$\dot{y_k} = \dot{y_k} + \frac{y_k - y'_k}{\Delta t}, \tag{3.12}$$

$$\dot{z_k} = \dot{z_k} + \frac{z_k - z'_k}{\Delta t}. \tag{3.13}$$

Restriction of the Particle's Position Function

This function restricts the positions of the particles to actual points in the point cloud. For each particle, its projection on the image frame is found, and its the (u_p, v_p) coordinates are obtained. Then, given the image coordinates, the corresponding point $P(x_p, y_p, z_p)$ in the cloud is found and the 3D position of the particle is corrected.

3.1.3 Projection of the Particles onto the Image Plane

To project the particles to the image plane we perform the inverse process of the point cloud generation discussed in 4.1.2.

We can use a simple proportional law to derive the image coordinates from the 3D ones. For the top-down case, figure 3.3, the horizontal offset of the object in the image with respect to the center is given by:

$$p_u = Py \cdot \frac{f}{P_x} \tag{3.14}$$

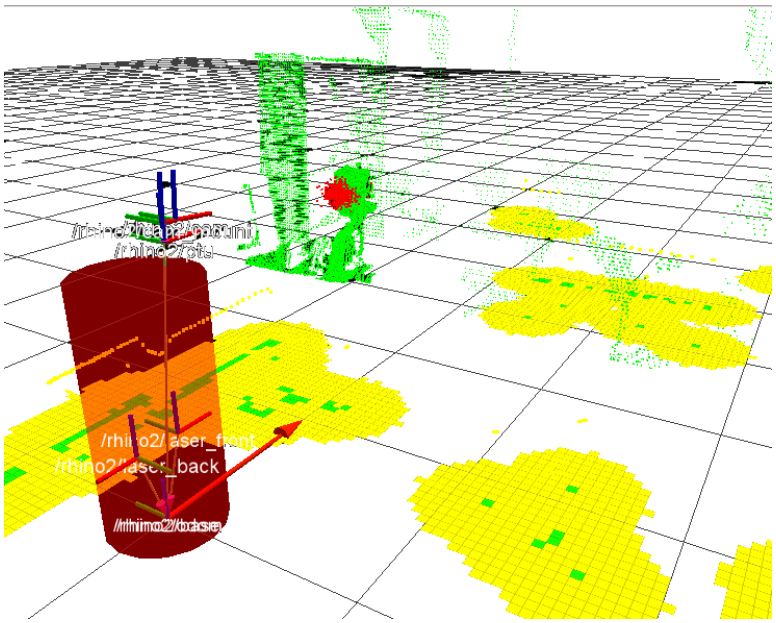

Figure 3.1: The point cloud (in green) and the particles (in red)

For the side case, Figure 3.2, the vertical offset of the object in the image with respect to the center is given by:

$$p_v = Pz \cdot \frac{f}{P_x} \tag{3.15}$$

Where f is the focal length of the camera, $\{Px, P_y, P_z\}$ are the 3D coordinates of the object, and $\{center|_u + p_u, center|_v + p_v\}$ are the image coordinates of the object.

This concludes all that relates to the motion model of the particles. To sum up the steps:

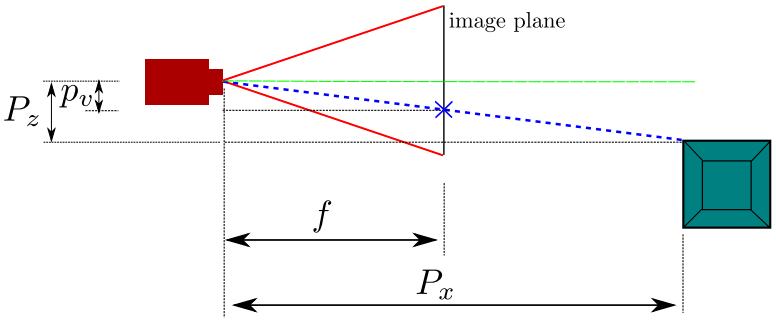

Figure 3.2: Side view of the projection of a point onto the image plane

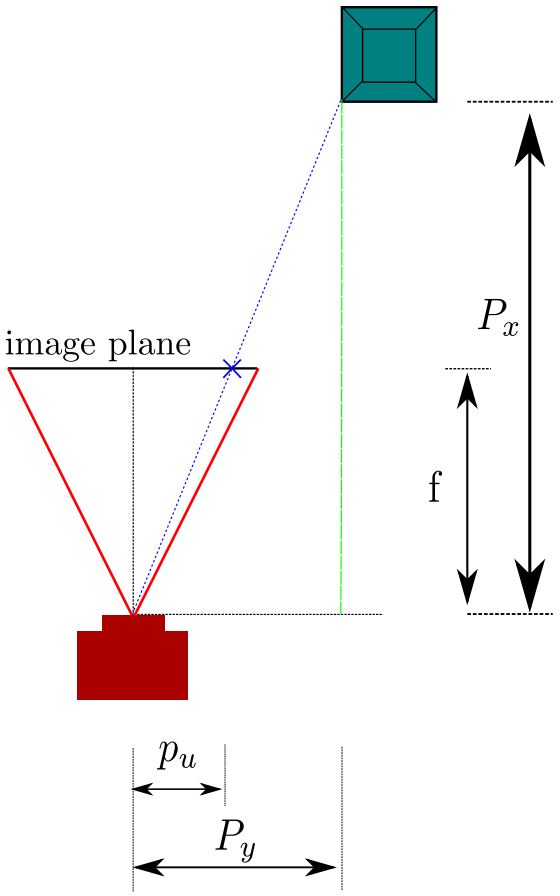

Figure 3.3: Top view of the projection of a point onto the image plane

- the motion model is applied in free space (equations 3.2 to 3.7),
- positions are updated according to the point in the cloud to which the projection of the free space positions correspond,
- the particles' velocities are updated according to the position adjustment,
- the dimensions of the bounding box are constant in the 3D world and only vary in the image plane.

3.2 Scalable Depth Gradient Features

The gradient features explained in Section 2.3 are also applied in the depth layer. In this way, we homogeneously incorporate a new source of information into the

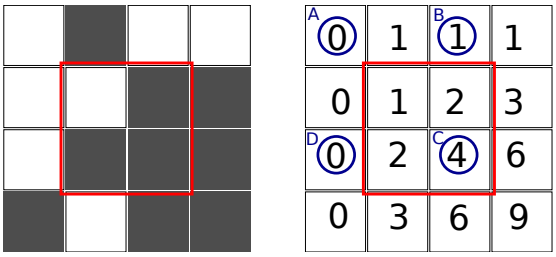

Figure 3.4: On the left a representation of the image of valid pixels; the red rectangle queries the number of valid pixels in its area. On the right the integral image of valid pixels

framework. It is homogeneous in the sense that the computational complexity of the features and the classifier is the same. Three aspects need to be considered:

- the nature of the depth values
- not every pixel comes with a valid depth value
- the thresholds of the classifier need to be adjusted

3.2.1 The Depth Image

The Kinect driver gives a depth image whose values correspond to the x component of the points in the cloud, as used in equation 4.3. In practise, the depth layer is used as if it were a grayscale image.

A significant number of pixels can turn up with invalid depth information. This phenomenon can arise due to the registration process (see section 4.1.2), but also to elements such as glasses that do not reflect the IR pattern back to the sensor. Therefore, a depth pixel validity integral image is computed. See Figure 3.4 for a representation.

3.2.2 Feature computation

In Section 2.3.2 the feature computation process is given for the gradient features. There, to compute the average value one must divide by the number of pixels in each of the regions. In the depth image the integral image of pixel validity ensures the averages are correctly computed by using only the number of valid pixels in the region.

3.3 Colour Features

In this section we introduce a new feature that is meant to capture the colour properties of the image. It captures average colour characteristics over regions, and its classifier is based on the scalable gradient features of section 2.3.

3.3.1 The Colour Space

The RGB pixel values of the image are transformed into the HSV colour space. The feature makes use of the hue and saturation, which are α and β in the cartesian representation. These two can be approximated from the RGB values as follows:

$$\alpha = \frac{1}{2}(2R - G - B) \tag{3.16}$$

$$\beta = \frac{\sqrt{3}}{2}(G - B) \tag{3.17}$$

We discard the V value, also known as brightness, since this information is already used by the gradient features of the grayscale layer.

3.3.2 Feature Computation

An integral image is computed using the α and β values of each pixel. To compute the feature value, the average α and β are computed over the whole region. The two cartesian averages are transformed to polar coordinates, namely the H_2 and C_2 approximations of the HSV colour space:

$$H_2 = atan2(\beta, \alpha) \tag{3.18}$$

$$C_2 = \sqrt{\alpha^2 + \beta^2} \tag{3.19}$$

The tuple $\{C_2, H_2\}$ is the feature value. The C_2 term is equivalent to the gradient magnitude in the grayscale and depth layers. The H_2 is analogous to the gradient angle. The main difference is that the colour feature gradient represents a two dimensional colour average and not a difference over adjacent regions. The H_2 and C_2 approximations are used for speeding up the computation of the features. It can be shown that error made by this approximation is of about 1.12° [wikipedia.org,].

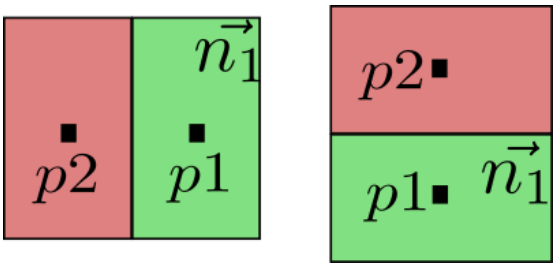

Figure 3.5: The vertical and horizontal surface normal feature regions.

3.3.3 The Classifier

Despite the different nature of the colour features, the same classifier as in the gradient features in used. It is explained in detail in section 2.3.3. The difference lays on some parameter adjustments that need to be performed and on the scale of the values.

In the grayscale and depth features, the magnitudes of the gradients are in logarithmic scale. In the colour features case this does not make sense anymore, since they represent colour averages. Therefore the natural scale is used. The thresholds of the center and outer bins have been adjusted so that the values corresponding to weak colour responses fall into the center bin.

3.4 Surface Curvature Features

In section 3.2 we present a way to integrate the spatial information into the object classifier. Whereas in 3.2 only one of the three spatial dimensions is used, the purpose of this section is to show a mechanism that utilises the three of them. For this purpose, the feature takes pairs of surface normals as an input and calculates three angular properties between them. In this way we attempt to capture the shape properties of the object.

3.4.1 Feature Description

The features specify two regions in the image for which the surface normals are computed. The regions can be arbitrarily defined within the space of the feature. The two basic forms are those of Figure 3.5, where the region is divided either horizontally or vertically.

The feature values are obtained from the first three dimensions of the surflet feature described in 2.5. The conditions to chose the reference vector are shown in 2.5. We skip this step and decide to choose n_1 as the reference vector u in every feature computation. In the case of horizontal features we take the right-hand side vector; for vertical features the lower one is chosen (see Figure 3.5).

The vectors $\vec{u}$, $\vec{v}$ and $\vec{w}$ as in equations 2.27 to 2.29. Then we chose the feature values as illustrated in the following equations

$$\alpha = atan2(\vec{w} \cdot \vec{n_2}) \tag{3.20}$$

$$\beta = \arccos(\vec{v} \cdot \vec{n_2}) \tag{3.21}$$

$$\gamma = \arccos(\vec{u} \cdot p_2\vec{p}_1) \tag{3.22}$$

3.4.2 The Grid Classifier

Our approach to a weak classifier discretises the possible feature values; see figure 3.6. The learning stage of the classifier consists in distributing the weights of the positive and negative examples in their corresponding cells of the grid. When querying the classifier on an unknown example, the corresponding cell of the grid is looked up and the ratio of positive over the sum of positive and negative values is returned.

The learning algorithm distributes the weights of the training examples into the cells. Each weight is distributed into seven cells as follows:

1. a cell is found such that the value of the training example feature is closest in the grid

2. the weight of the training example is distributed to the set of cells proportionally to the distance to each of the chosen bin centres

3. the fraction of the weight assigned to the cell is given by a Gaussian centred on the bin centre and evaluated at the corresponding feature value dimension

Learning Stage

We denote the feature values as $\Upsilon = \{\alpha, \arccos\beta, \arccos\gamma\}$ and the training set $\mathbf{T} = \{(\Upsilon_1, w_1, y_1), \ldots, (\Upsilon_N, w_N, y_N)\}$ where Υ_k is the feature value of the k^{th} training example, w_k the weight and $y_k = \pm 1$ its class label. The grid is divided into B bins of width bw; the k^{th} edge be_k is at $k\frac{\pi}{8}$. The k^{th} bin centre bc_k is at:

$$bw \equiv \frac{\pi}{B} \tag{3.23}$$

$$be_k \equiv \frac{k\pi}{B} \tag{3.24}$$

$$bc_k \equiv \frac{\frac{k\pi}{B} + \frac{(k-1)\pi}{B}}{2} = \frac{k\pi + k\pi - \pi}{B/2} = \frac{4k\pi}{B} - \frac{2\pi}{B} \tag{3.25}$$

The set of bin centres is denoted as $\{(\alpha_{c1}, \beta_{c1}, \gamma_{c1}), (\alpha_{c2}, \beta_{c1}, \gamma_{c1}), (\alpha_{c3}, \beta_{c1}, \gamma_{c1})...\}$. The grid is denoted as $\mathcal{G}^y_{\alpha,\beta,\gamma}$, with $\{\alpha, \beta, \gamma\} \in \{[0..B], [0..B], [0..B]\}$

The closest bin b_i for a given training example $\Upsilon_i = (\alpha_i, \beta_i, \gamma_i)$ is given by:

$$b_i \equiv (\alpha_{ci}, \beta_{ci}, \gamma_{ci}) = \Upsilon_i \cdot \frac{1}{bw} = (\lfloor \alpha_i \frac{B}{\pi} \rfloor, \lfloor \beta_i \frac{B}{\pi} \rfloor, \lfloor \gamma_i \frac{B}{\pi} \rfloor) \tag{3.26}$$

The weight w_i of the training example is distributed among the set of seven bins $\{(\alpha_{ci}, \beta_{ci}, \gamma_{ci})\} \cup \{(\alpha_{ci\pm1 \mod B}, \beta_{ci\pm1 \mod B}, \gamma_{ci\pm1 \mod B})\}$. The weighting factor of each bin is given by a product of three Gaussians; each centred on the bin centre of the corresponding feature dimension, with a standard deviation of bw and evaluated at the training example corresponding feature value. The weighting factor:

$$w_{fi} = w_{\alpha_{ki}, \beta_{ki}, \gamma_{ki}} \tag{3.27}$$

with

$$k \in \{ci, ci \pm 1 mod B\} \tag{3.28}$$

The individual weighting factors are then:

$$w_{\alpha_{ki}} = \mathcal{N}(0; \alpha_i - \alpha_k, bw^2) \tag{3.29}$$

$$w_{\beta_{ki}} = \mathcal{N}(0; \beta_i - \beta_k, bw^2) \tag{3.30}$$

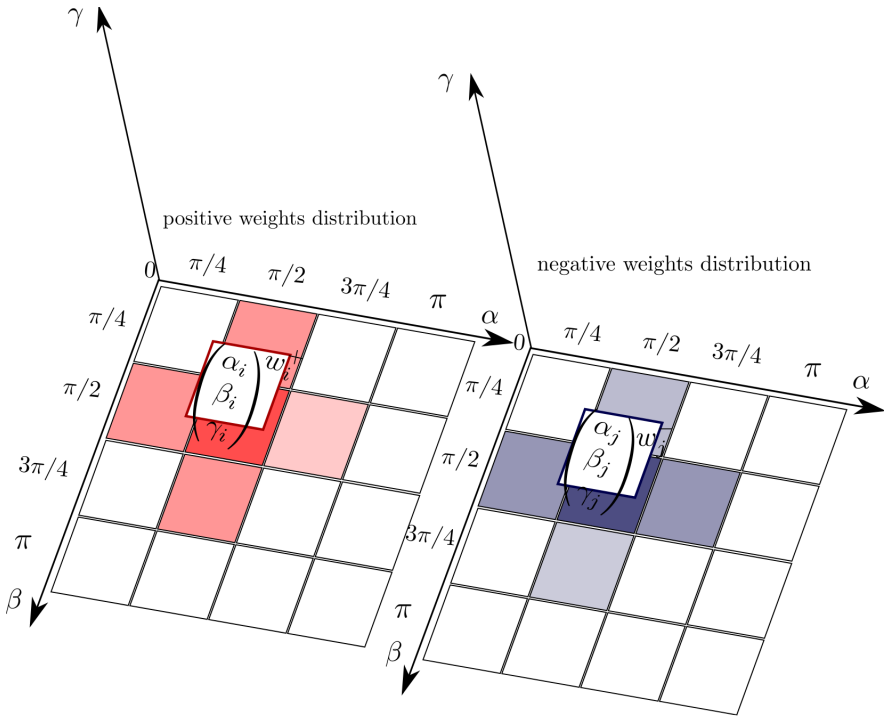

Figure 3.6: The axis of the grid are the feature values α , $\arccos\beta$ and $\arccos\gamma$

$$w_{\gamma_{ki}} = \mathcal{N}(0; \gamma_i - \gamma_k, bw^2) \tag{3.31}$$

The portion of the weight that goes to each bin is then:

$$w_{fi} = w_{\alpha_{ki}}\ w_{\beta_{ki}}\ w_{\gamma_{ki}} \tag{3.32}$$

$$\mathcal{G}^{yi}_{\alpha_k,\beta_k,\gamma_k} = w_i\ w_{fi} \tag{3.33}$$

Classifying unknown examples

For the classification of an unknown example $\Upsilon_u \equiv (\alpha_u, \beta_u, \gamma_u)$ the bin of the grid closest to the queried example is obtained as in equation 3.26:

$$u_\alpha = \lfloor \alpha_u \frac{B}{\pi} \rfloor \tag{3.34}$$

$$u_\beta = \lfloor \beta_u \frac{B}{\pi} \rfloor \tag{3.35}$$

$$u_\gamma = \lfloor \gamma_u \frac{B}{\pi} \rfloor \tag{3.36}$$

Let the positive value at the corresponding bin be $p \equiv \mathcal{G}^{y=+1}_{u_\alpha,u_\beta,u_\gamma}$ and the negative value be $n \equiv \mathcal{G}^{y=-1}_{u_\alpha,u_\beta,u_\gamma}$; the classifier prediction for the unknown example is the ratio of positive over the sum of positive and negative values:

$$C(\Upsilon_i) = \frac{p}{p+n} \tag{3.37}$$

3.5 Adaptive Feature Pool

A tracker for arbitrary objects has to deal with very different situations. In some scenarios, one feature kind might be more useful than in others. For example, when tracking a white object that is moving before a white background, colour features can become of very little help; while depth features will be decisive.

In this section a mechanism is introduced that makes the features in the pool adapt to the type of object and background in which it is being tracked. The aim is to keep the number of features in the pool constant but better specialised.

The following sections cover the mechanisms of initialising the pool of features, measuring the feature errors, and updating the pool.

3.5.1 The Dynamic Feature Pool Initialisation

The features that are initially added to this pool are chosen randomly in position and size. They can be as wide or high as the example on which they are evaluated. The number of dynamic features can be parametrisable.

3.5.2 Feature Error Measurement

The question that is tried to be answered here is, how should the error made by the features be measured? A first approach would be to use every training error measure obtained when calling the AdaBoost algorithm. Another possibility is to utilise only the error made on the first AdaBoost call, when the weights of positive and negative examples are evenly distributed, each set summing up 0.5.

In any of the two cases, after each iteration of the tracking algorithm, every feature has a training error value.

3.5.3 Pool Update

The objective is to make the features specialise on the object characteristics. Having a measure of the training error made by the features, the next step is to use this error to update the content of the pool. For this reason, the average error made by the features of each kind (colour, depth,...) is computed.

Feature Removal

For the selection of the feature kind that will be removed a probabilistic policy is used. The chances of selecting each kind for removal from the dynamic pool are proportional to its normalised average error. First, a random number is sampled; the kind with the smallest error that is bigger than this number is chosen. If none is bigger than the sampled number, then the kind with the highest average error is chosen. Out of the kind that was chosen, the feature that made the biggest training error is removed from the pool.

New Feature Sampled

If the step above succeeded a new feature is added to the pool. To select the kind to be added to the pool, a probabilistic policy is also used. The chances a feature kind has of being chosen are inversely proportional to their average errors. A random number is sampled; with the normalised average errors, the feature kind with the highest average error lower than the random number is chosen to be added. If none is lower than the random number, the kind with the lowest average error is chosen.

Algorithm 6 Adapt Dynamic Pool

Require: • Sets of errors of each feature kind $E_k = \{e_k^1, .., e_k^{M_k}\}$
• Average errors of each feature kind $\{c_k\}_{k \in kinds}$

```
if removeFeature({c_k}_{k∈kinds},{E_k}_{k∈kinds}) then
    addFeature({c_k}_{k∈kinds},{E_k}_{k∈kinds})
end if
```

The pseudocode for the update strategy can be seen in algorithm 6. Algorithm 7 shows the policy for feature removal. Algorithm 8 contains the policy for feature addition. In the next section we show the statistics used to analyse the behaviour of this mechanism.

Algorithm 7 removeFeature

Require: • The normalised average errors of each kind $\{c_k\}_{k \in kinds}$
• Sets of errors of each feature kind $E_k = \{e_k^1, .., e_k^{M_k}\}$
• Selection function $f(x, rand) = max(x - rand, 0)$

$n \leftarrow$ random()
$kind \leftarrow \underset{k}{\text{argmin}}(f(c_k, n))$
if $kind == 0$ **then**
  $kind \leftarrow k | c_k = max\{c_k\}_{k \in kinds}$
end if
if numberOfFeatures($kind$)>0 **then**
  $r_k \leftarrow \underset{i}{\text{argmax}}(max(\{e_k^i\}_{i \in 1..M_k}))$
  pool.remove(k,r_k)
  return true
else
  return false
end if

Algorithm 8 addFeature

Require: • The normalised average errors of each kind $\{c_k\}_{k \in kinds}$
• Selection function $f(x, rand) = max(rand - x, 0)$

$n \leftarrow$ random()
$kind \leftarrow \underset{k}{\text{argmin}}(f(c_k, n))$
if $kind == 0$ **then**
  $kind \leftarrow k | c_k = min\{c_k\}_{k \in kinds}$
end if
pool.addNewSampledFeature(k)

Feature Statistics

We compute the following statistics for each feature type and kind:

- first quartile, median and third quartile
- minimum and maximum error
- average error

The first two sets of measurements are used for the analysis of the behaviour of the pool. The average error is used in order to decide which feature kinds will be removed and added to the pool.

3.6 Robot Control

The information provided by the tracking algorithm about the target position is used to control the behaviour of a robot. The robot will have a pan-tilt unit on which the camera is mounted. A circular base will let the robot move around.

Algorithm 9 Pan-tilt unit target update

Require: • The target position at time t $P_t = (p_x, p_y, p_z)$
• The confidence on the estimate $conf$
• The physical limits on the pan-tilt unit positions $(PAN\_LIMIT, TILT\_LIMIT)$

if $conf > threshold$ **then**
  targetpan $\leftarrow atan(p_y/p_x)$+ current pan
  targettilt $\leftarrow atan(p_z/p_x)$+ current tilt
  targetpan $\leftarrow max(-PAN\_LIMIT, min(PAN\_LIMIT, targetpan))$
  targettilt $\leftarrow max(-TILT\_LIMIT, min(TILT\_LIMIT, targetilt))$
else
  targetpan $\leftarrow 0$
  targettilt $\leftarrow 0$
end if

The target position of the pan tilt unit is updated given the relative position of the target. If there is not enough confidence on the estimate, the target position of the pan tilt unit is set to the origin. The steps are contained in Algorithm 9.

To control the base the robot the relative coordinates of the target are used. The initial distance to the target is stored so that the robot never gets any closer to it. The control scheme rotates the base with the current angle of the target, and if the distance to it is greater than the initial one, moves the base forward.

The angle to the target and the distance are incorporated in a single movement command. The steps are summarised in Algorithm 10.

Algorithm 10 Base control

Require: • The target position at time t $P_t = (p_x, p_y, p_z)$
- The confidence on the estimate $conf_t$
- The initial distance d_0

1: $\alpha \leftarrow atan(p_y/p_x)$
2: **if** $conf_t > movingConfidence$ **then**
3: $r \leftarrow \sqrt{p_x^2 + p_y^2}$
4: **if** $r > safetyDistance$ **then**
5: move(α,$r - d_0$)
6: **else**
7: move(α,0)
8: **end if**
9: **end if**

Chapter 4

Hardware and Software Environment

This chapter covers the hardware and software environment on which the tracker runs. The nature and characteristics of the Kinect sensor, and those of the mobile platform on which the tracker was tested. A brief overview on how the system was designed is also given. A deeper view on the system design and the software environment can be found in the appendix appendix.

4.1 The Kinect RGB-Depth Sensor

RGB-D sensors are a powerful tool in the fields of robotics and computer vision. They provide both RGB and depth information in a way that can be easily exploited by the programmer. In the last years there has been a great improvement in terms of cost; especially after the introduction of Microsoft Kinect, a low-cost RGB-D sensor sold commercially for the company's gaming platforms.

The Kinect sensor [Microsoft,] consists of an RGB camera, an infra-red pro-

Figure 4.1: The Kinect sensor

jector and a CMOS sensor. To generate the depth data, the projector emits a certain pattern of light that is reflected in the objects and received in the IR sensor. The distortion of the pattern is used to estimate the depth pixel wise. Further information about the depth obtainment mechanism can be found at [Kurt Konolige,].

4.1.1 Characteristics of the sensor

The available frame rates are 30Hz and 15Hz. The RGB and depth available resolutions are 640 x 480, 320 x 240 and 160 x 120 pixels. The horizontal field of view of the RGB sensor is 62 degrees. For the depth sensor this value is 58 degrees. The operation range is in the interval $[0.8m, 3.5m]$; the depth resolution at $2m$ from the sensor is $1mm$. These values were reversed engineered at [Openni,].

Precision and Accuracy

The precision and accuracy were calibrated in [ROS, a]. The error err grows quadratically with the distance of the measurement $dist$. It follows the curve $err = dist^2 * 0.0075$ in milimeters. Thus, at a distance of two meters, the error is in the range of $\pm 3mm$. The accuracy of the sensor is of $\pm 1mm$.

4.1.2 The OpenNI Driver

OpenNI is an open source driver for the Kinect. In our system we will be using the available version for the ROS environment [Radu Bogdan Rusu,].

Depth Registration

Depth registration is the process of assigning a depth value to each pixel in the image. Since the focal length of the RGB sensor is smaller than that of the IR sensor, its field of view is therefore greater; this means that the area projected on the depth image is smaller than the RGB one. The problem to be solved is that of pixel correspondence between a pair of stereo images (one RGB and one depth image) where one image is contained in the other.

The depth image must be transformed using the information of the physical location of the depth sensor with respect to the RGB one. Thus, after transforming the depth values to make the two images match, the one provided by the depth sensor is contained in the RGB one; but not every pixel in the RGB image

has a corresponding depth value. Those pixels lying on the borders are the ones that have a missing depth value. This can be seen in the registered depth image in 4.2 and its corresponding RGB frame 4.3.

This process is handled by the OpenNI driver.

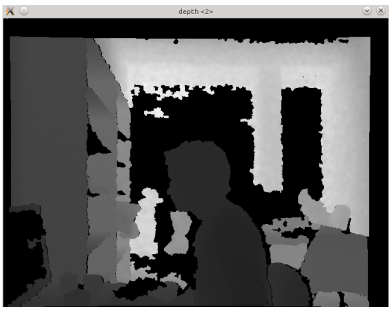

Figure 4.2: The registered depth image

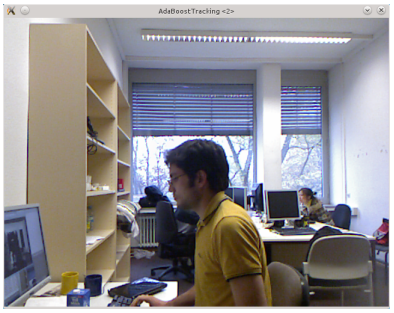

Figure 4.3: The RGB image

Image format

The image matrix *img* can be accessed by its pixel absolute coordinates as $img(u, v)$ returning a 24 bit value containing the RGB colour values.

The depth image matrix *depth_img* can be accessed in the same way as $depth\_img(u, v)$ to return the depth associated to that pixel. In this case it is an 11 bit value. It comes in meters and is interpreted as follows: if the image plane were parallel to the YOZ plane in 3 dimensions, the X axis orthogonal to this plane, the depth value is the distance from each point in space to the YOZ plane.

Point cloud generation

The OpenNI driver can generate an RGB point cloud. That is an array of 3D points in space and their associated RGB values. Given the depth X of each pixel (u, v) in the RGB image, its 3D coordinates (x, y, z) are computed as:

$$y = (u - centerCloudY) * X * constant \tag{4.1}$$

$$z = (v - centerCloudX) * X * constant \tag{4.2}$$

$$x = X \tag{4.3}$$

Where the term *constant* is the inverse of the RGB camera focal length.

4.1.3 Modification of the OpenNI kinect driver

To fit our needs the OpenNI Kinect driver was modified. A new type of ROS sensor message, *ImageWithDepth.msg*, was created to hold the RGB-D data and send it to the tracker node. The RGB data is stored in an 8 bit integer matrix resulting in 24 bits per pixel and 640x480 pixels at full resolution. The depth values are stored in a 16 bit integer array. The function that normally produces the RGB point cloud was modified to simply publish the registered depth data together with the RGB data. The point cloud computation was removed from the driver to save computation power.

4.2 Rhino

Rhino is an autonomous mobile plattform. Some of the experiments were carried out on it. It consists of a circular base that can rotate and move in arbitrary directions. 40 centimeters from the wheels there are two laser range sensors. The Kinect sensor is on the top of the robot; it is mounted on a pan base that works together with the Kinect tilt unit. An onboard computer lets the robot be fully autonomous.

4.2.1 Coordinate frames

When the tracker is run on a mobile platform (see Section A.2.2 of the appendices) several coordinate frames have to be maintained. We call *world* the global coordinate frame that has its origin at the starting position of the robot. The robot's frame of reference is called *base* and corresponds to the position and orientation of the base of the robot. The camera has also its own frame of reference and is denoted as *camera*. The homogeneous transformation matrices that transform the coordinates from one frame to the other are denoted with the letter H for homogeneous matrix, and have a sub and a super index denoting from which (sub) frame of reference to which (super) the transformation of coordinates is done. For example, the matrix T_{base}^{camera} is used to change the coordinates from the base link to the camera link. The particle's state is defined in the *world* frame of reference. In order to evaluate the observation model of the particles, they first

have to be transformed from the *world* frame of reference to that relative to the camera.

4.3 System Design

The system is implemented as a collection of ROS nodes in the C++ language. The *tracker* node implementing the tracking algorithm; the *ctrl* node that controls the pan tilt unit on which the Kinect sensor is mounted; and the *driver* node that sends movement commands to the base.

The architecture of the system is designed so that heterogeneous features could be homogeneously integrated into a common boosting framework. It also has to cope with new features being added into the pool at any point in time. The main difficulty is to make the system fast enough to run in real-time. This section gives a general view of the system.

4.4 General Structure of the System

An overview of the system is shown in diagram 4.4. The tracker node implements the tracking algorithm. It subscribes to the node where the modified version of the *OpenNI Kinect* driver is running. The latter publishes the RGB-D data in the format already mentioned in 4.1.3.

The tracker node processes every incoming frame inside of a callback function and produces several output information:

- displays an overlay of the bounding box estimate over the current frame
- the projection of the particles on the image
- the bounding box estimate position and dimensions in image coordinates are written to a text file
- a message containing the 3D coordinates of the target and the confidence on this estimate is published

The last element is the *targetPos* message, received by the *Ctrl* and *Driver* nodes. The *ctrl* node uses this information to control the pan tilt unit of the robot and keep the target centred in the image. The *driver* node controls the base of the robot; it tries to follow the target's movements.

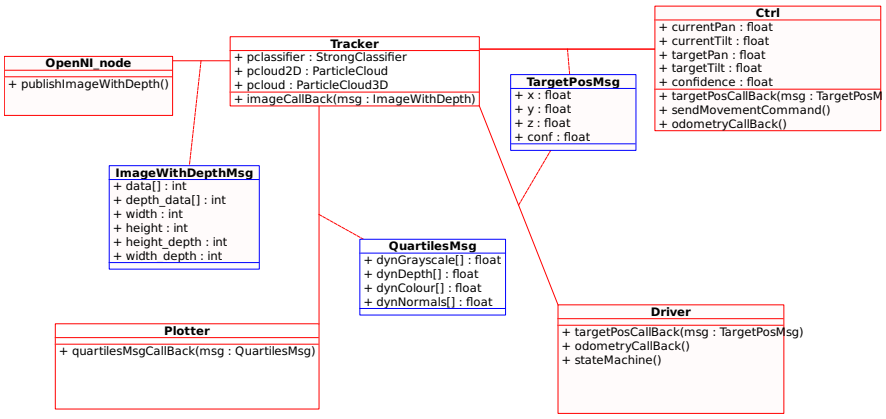

Figure 4.4: UML overview of the system

The classes relations are depicted in the UML diagram of Figure 4.5. The logic of the system is that the tracker class contains an object that implements the particle filter algorithm. That is the *ParticleCloud3D* class, which at the same time contains particles. The *ParticleCloud3D* class must have an observation model. This is implemented in the *StrongClassifier* class and all the others that are related to it. The weak classifiers defined for the system must inherit from the *WeakClassifier* abstract class. The system makes use of past examples to compute the new features as they are added by the dynamic feature pool. The Example class stores this information.

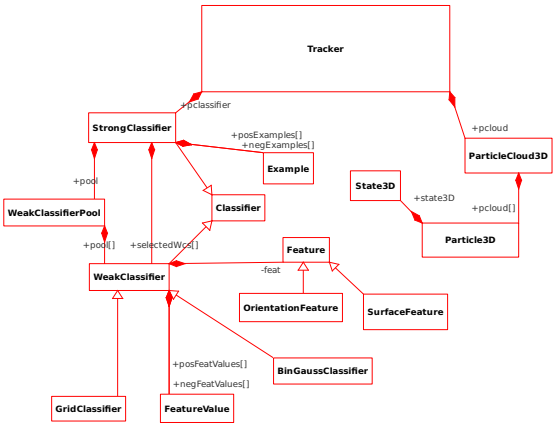

Figure 4.5: UML class diagram of the tracker node

Further details about the implementation can be found in appendix B.

Chapter 5

Evaluation

5.1 Measurements

Two magnitudes are used to measure the performance of the tracking algorithm: the overlap and hit rates. The first gives an idea of how much of the target estimate actually matches the ground truth. Figure 5.1 depicts two cases where a 33% overlap is achieved. The hit rate measurement states on what percentage of frames the overlap was bigger than 33%; or in other words, what was the percentage of frames on which the target was on track. The two measurements are evaluated on a per-frame basis and averaged over the number of frames. The tool for evaluating the overlap can be found in the BoBoT web site [Klein,].

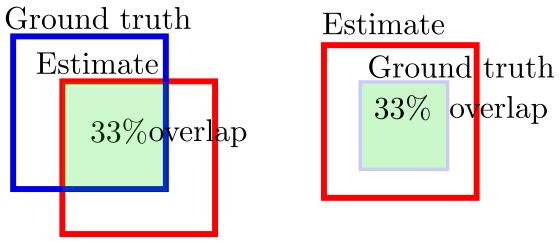

Figure 5.1: Overlap measurement

5.2 Hardware Setup

For the execution of the tests a PC was used with the following characteristics: an *Intel(R) Xeon(R) X5650 (Hexcore)* CPU with 6 cores and 2.66 Ghz clock frequency; and *48 GB DDR3 (1066 MHz)* of memory. In Table 5.1 and 5.2

Figure 5.2: Sequence M. RGB and depth images

Figure 5.3: Sequence N. RGB and depth images

we show the average timing results obtained in two main configurations for the tracker.

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
false	true	true	true
particleState2D	**numParticles**	**numWC**	**numPosEx**
true	1500	23	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
125	0.56	false	false
boost_max	**boost_min**	**std_dev**	**Time per callback function**
6	1	550	39.6 ms

Table 5.1: Timing with base configuration

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
true	true	true	true
particleState2D	**numParticles**	**numWC**	**numPosEx**
true	1500	23	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
125	0.56	false	false
boost_max	**boost_min**	**std_dev**	**Time per callback function**
6	1	550	89 ms

Table 5.2: Timing with normal features

5.3 *BoBoT-D* Benchmark

The *BoBoT-D* benchmark was recorded in the faculty environment. In order to evaluate the results, the ground truth was manually labelled. It consists of five RGB-D video sequences and their corresponding ground truth data.

- The milk sequence (Sequence M): a milk tetra-pack is standing on a breakfast table. It is grabbed and moved to pour some milk on a cup of coffee.

Figure 5.4: Sequence O. RGB and depth images

Figure 5.5: Sequence P. RGB and depth images

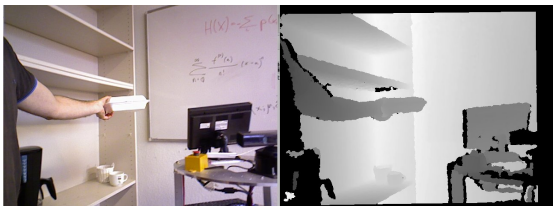

Figure 5.6: Sequence Q. RGB and depth images

Then it is returned to the original position. The difficulty arises when the tetra-pack is rotated to pour the milk and its appearance changes.

- The ball sequence (Sequence N): a ball is being passed between two persons. The ball is red with white pentagons over the surface. The difficulty is the change of appearance and the fast movement of the ball; as well as some partial occlusions produced by the hands of the persons.

- The tank sequence (Sequence O): a radio control tank is moving; a bridge has been improvised and some batteries populate the environment. Fast moving object as well as fast changes of appearance are the characteristics of this video.

- The person sequence (Sequence P): it was recorded with the camera mounted on the Rhino robot. One person is walking down one office corridor; other people cross on the way. Some partial and total occlusions occur on this video.

- The lunch box sequence (Sequence Q): a white lunch box is placed together with other white objects such as cups. It is grabbed and moved close over a white board. Towards the end the light changes dramatically.

5.3.1 Test Set 1: Colour, Depth and Grayscale Features

This test set was performed with the following parameters:

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
false	true	true	true
particleState2D	**numParticles**	**numWC**	**numPosEx**
false	1500	23	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
125	0.56	false	false
boost_max	**boost_min**	**std_dev**	
6	1	550	

The maximum number of newly boosted classifiers is relatively high: almost one fourth of the total number of classifiers. All the feature types are available except for the surface normals. Every test was run ten times. In the following sections we will analyse the results in each of the tests, by picking one of the experiments at random and showing the average and individual results.

Sequence M (Milk)

The results obtained in this sequence were the following:

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
74,77	75,14	76,31	74,48	72,61	75,42	74,60	73,94	74,99	75,65	74,54

Table 5.3: Sequence M: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
99.52	100.00	100.00	99.38	96.75	100.00	100.00	100.00	100.00	99.07	100.00

Table 5.4: Sequence M: Hit rate over runs and average

The test on the milk sequence gave as a result a 74.77% overlap average and a 99.52% hit rate. The main intention of this sequence is to evaluate the performance of the tracker with an object that rotates around a horizontal axis. This is the most challenging kind of rotation for the tracking algorithm.

To analyse the events of this sequence, one of the executions is chosen at random.

Figures 5.8, 5.10 and **??** contain the error evolution of the dynamic grayscale, colour and depth features. Picking the last, one can see the effect of the dynamic

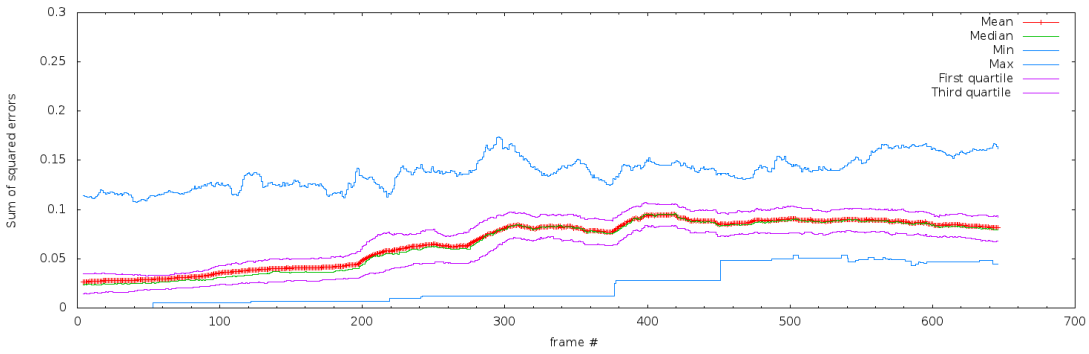

Figure 5.7: Sequence M: static gray features error quartiles distribution

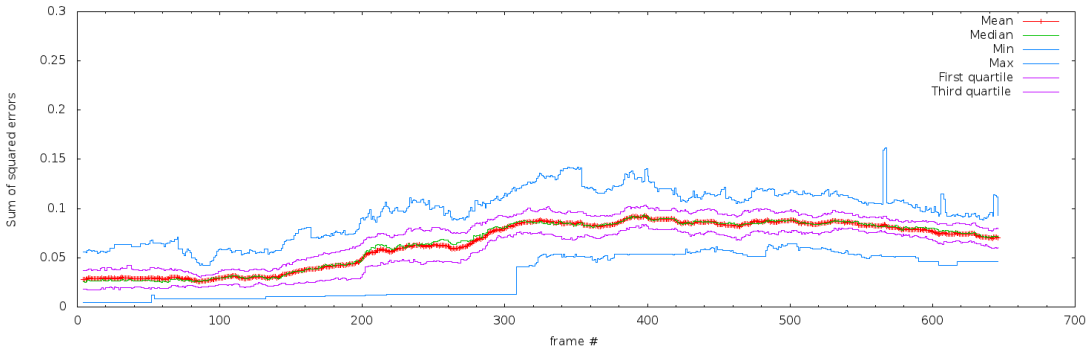

Figure 5.8: Sequence M: dynamic gray features error quartiles distribution

pool on the error distribution evolution: as the depth features are being removed from the pool, see Figure 5.11, the error distribution shrinks; when the number of features is close to the minimum (around frame 200) the values start varying greatly from frame to frame. In the case of the dynamic colour features, their average error stays below that of the static. Compare figures 5.10 and 5.9.

Average values	Min	First quartile	Median	Third quartile	Max	Mean
Static grays	0.0225	0.0537	0.0649	0.0777	0.1381	0.0671
Dynamic grays	0.0322	0.0539	0.0650	0.0751	0.0964	0.0646
Static depths	0.0265	0.1957	0.2116	0.2219	0.3170	0.1997
Dynamic depths	0.0963	0.1196	0.1332	0.1512	0.1624	0.1323
Static colours	0.0516	0.1321	0.1870	0.2166	0.2387	0.1735
Dynamic colours	0.0295	0.0571	0.0671	0.0781	0.1201	0.0683

Table 5.5: Sequence M: averaged error statistics

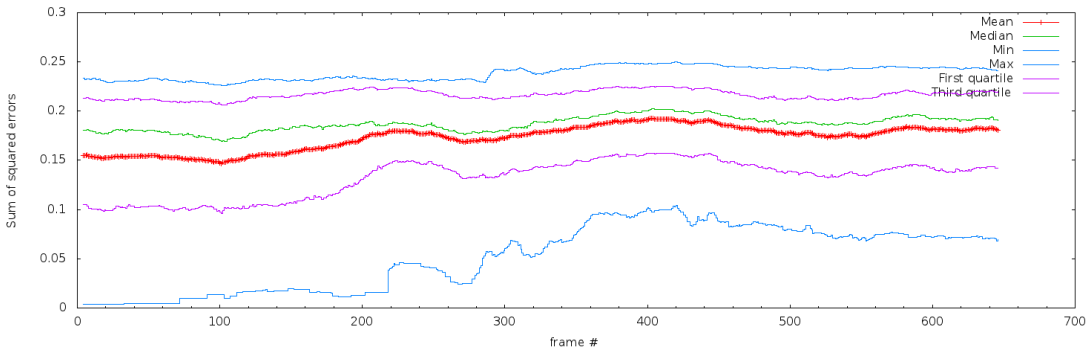

Figure 5.9: Sequence M: static colour features error quartiles distribution

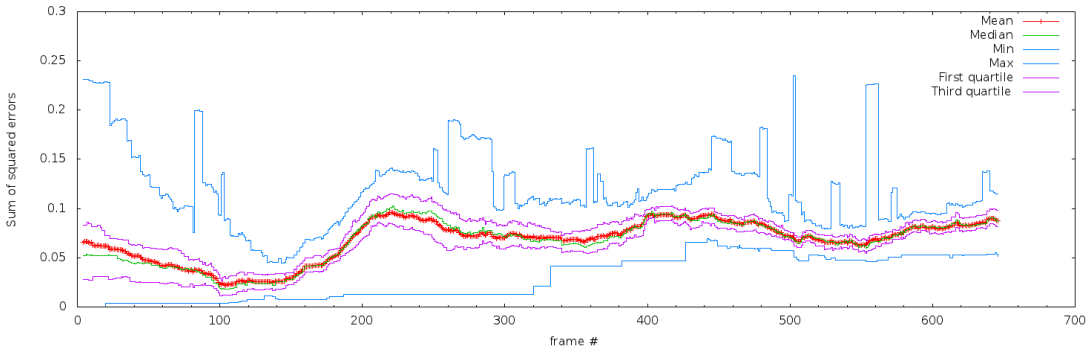

Figure 5.10: Sequence M: dynamic colour features error quartiles distribution

The number of features in the pool can be seen in Figure 5.11. The selected kinds in the classifier are depicted in figure 5.12. How the feature kinds are distributed within the static and dynamic types in the classifier is shown in Figures 5.14 and 5.13 respectively.

Figure 5.15 contains four relevant frames of the sequence. They are named with letters A,B,C, and D, which are also marked on the confidence plot . The sequence starts with a breakfast table with some objects on top. At frame 190 the light intensity drops and the viewpoint of the camera slightly changes. This causes a decay on the confidence of the classifier. After being picked up from the table, around frame 290, the tetra-pack starts to rotate causing a drop in the confidence. The classifier re-learns its new aspect until frame 375, when the object is rotated a bit more to pour some milk. This part was critical for the algorithm, since rotations of this kind have to be handled indirectly by the classifier. Finally, at frame 550 the camera moves away from the scene, and the gain of the sensor

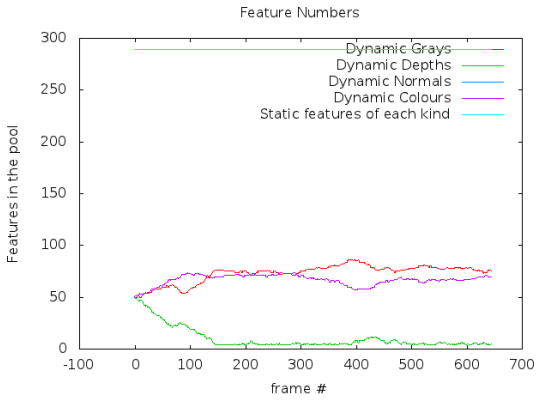

Figure 5.11: Sequence M: static and dynamic features in the pool

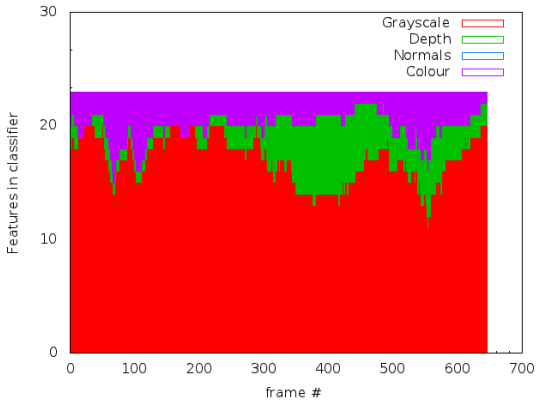

Figure 5.12: Sequence M: total selected features

is adjusted to the lighting conditions of the beginning of the sequence.

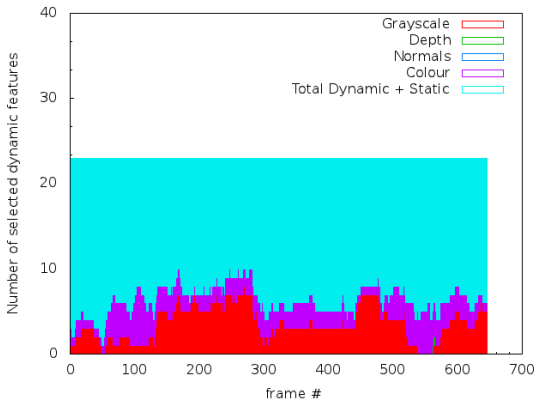

Figure 5.13: Sequence M: selected dynamic features

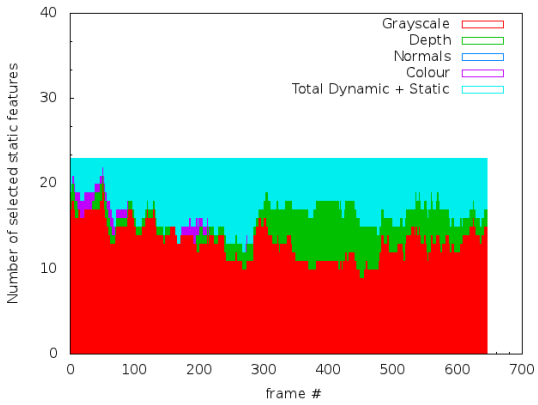

Figure 5.14: Sequence M: selected static features

Figure 5.15: A: frame 190, the light intensity drops. B: frame 290, the tetra-pack is rotated around the view-point axis. Frame 375: the object is greatly rotated. Frame 550: lightning conditions are restored as the camera moves back

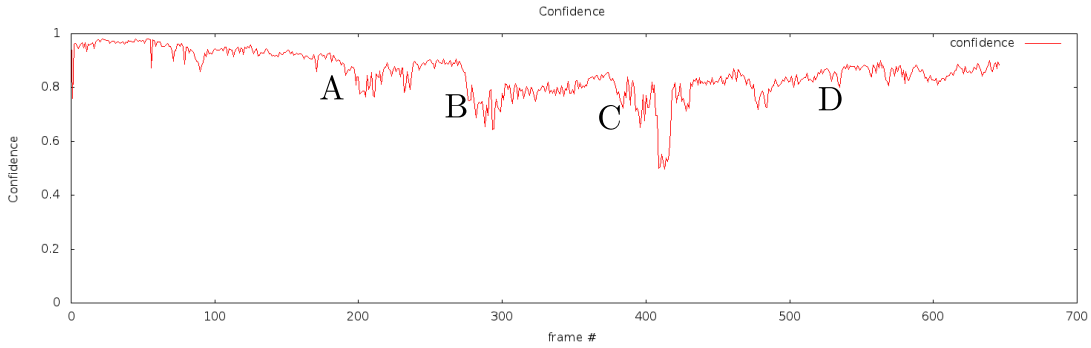

Figure 5.16: Sequence M: confidence

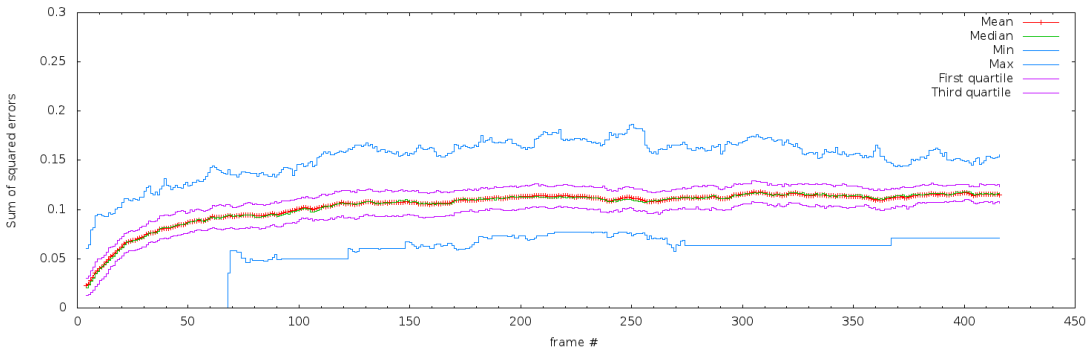

Figure 5.17: Sequence N: static gray features error quartiles distribution

Sequence N(Ball)

The results obtained in the ball sequence were the following:

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
67,78	61,13	72,92	70,34	70,25	66,13	68,24	66,84	64,82	70,35	66,80

Table 5.6: Sequence N: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
96.53	89.47	99.52	98.56	97.13	97.13	99.52	95.69	93.78	97.13	97.37

Table 5.7: Sequence N: Hit rate over runs and average

The test on the ball sequence gave as a result an average of 67.78% overlap and a 96.53% hit rate. The difficulty on this sequence resides on the high speed at which the ball moves. Because of this movement some shade artefacts appear in the image. The colour features play an important role here since they are able to "resist" the appearance changes of the rolling ball.

Figure 5.22 shows the feature kinds that were part of the classifier over time. The colour features are about two thirds of the total in the first half, and no less than one half in the second. It is also remarkable that throughout the test, about one third of the selected features are dynamic, most of them being of the colour kind, see Figure 5.23.

The evolution of the static types errors goes along what has been commented. In the case of the colour features, Figure 5.19, the first quartile of the distribution stays close to zero throughout the test, and the mean is also kept under 0.1.

The effect of the dynamic pool can be seen in the changes in the error distributions. For example, in the dynamic grayscale features, Figure 5.18, one can see

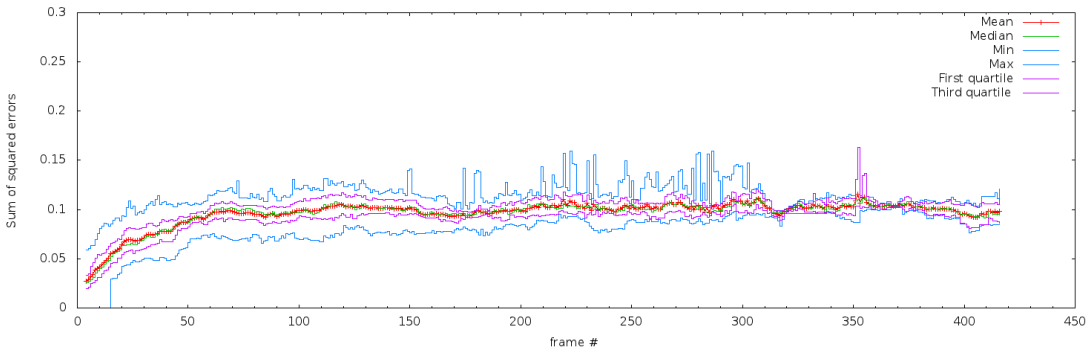

Figure 5.18: Sequence N: dynamic gray features error quartiles distribution

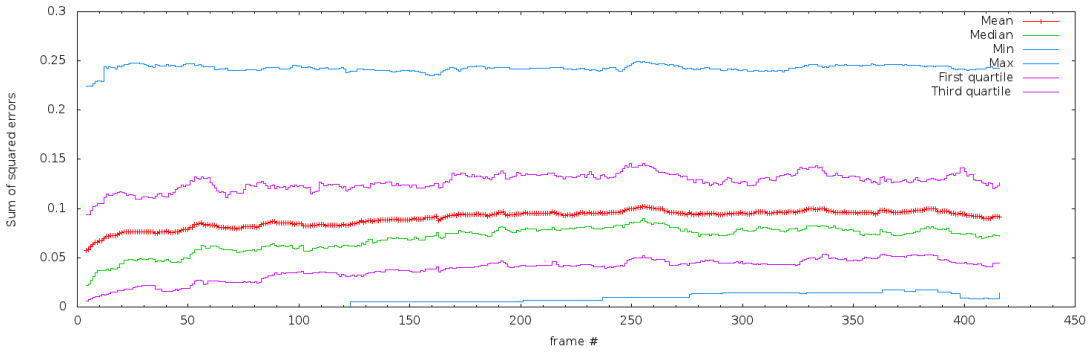

Figure 5.19: Sequence N: static colour features error quartiles distribution

that the maximum value is constantly changing due to the removal of features from the pool. Also in the case of the dynamic colour features, there are significant peaks in the maximum error graph that appear from time to time. This is due to new features being added to the pool that are worse than the existing ones. These peaks disappear less often since the colour features are less often removed from the pool as compared with the grayscales or depth features. Note towards the end of the dynamic feature numbers plot, Figure 5.21, that there is a straight line; the reason is that the chosen kinds for removal were already at its minimum number and therefore no changes in the pool occur.

Figure 5.25 shows four key frames of the sequence. Frames A and C contain moments at which a sudden acceleration of the ball was occurring. One can trace this event in the confidence graph of figure 5.26 and see a down peak in the

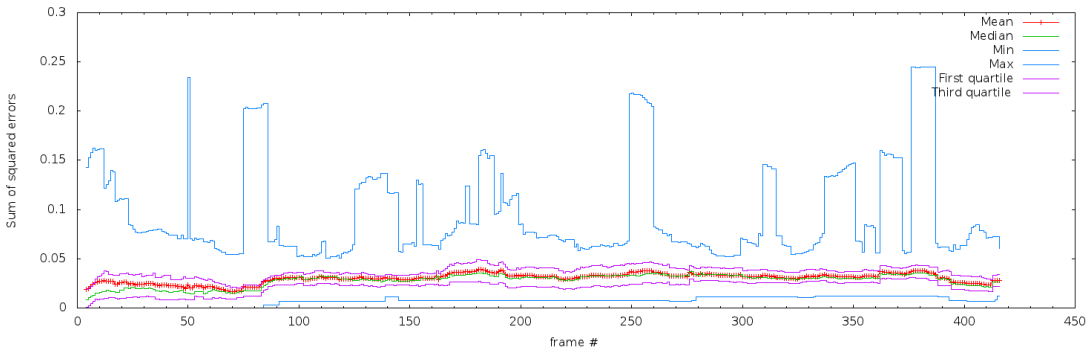

Figure 5.20: Sequence N: dynamic colour features error quartiles distribution

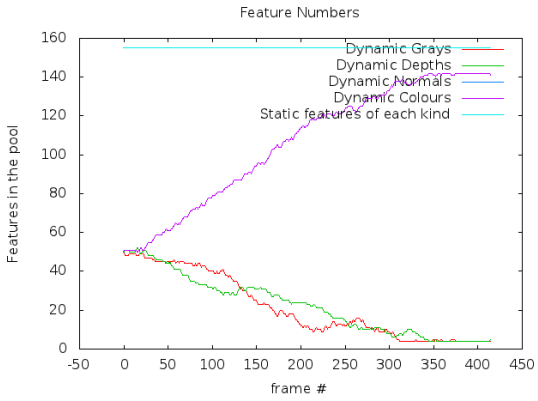

Figure 5.21: Sequence N: static and dynamic features in the pool

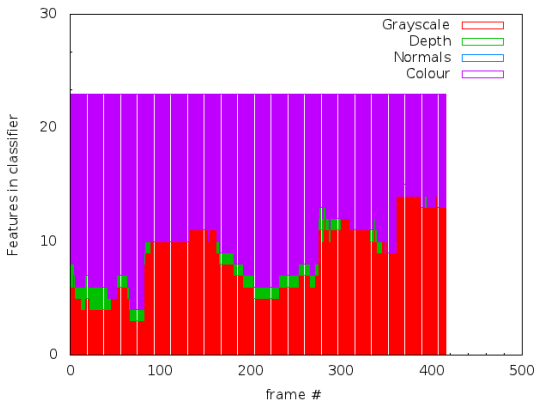

Figure 5.22: Sequence N: total selected features

line. In frames B and D, a sudden stop was occurring, and a similar down in the confidence can be seen in the plot.

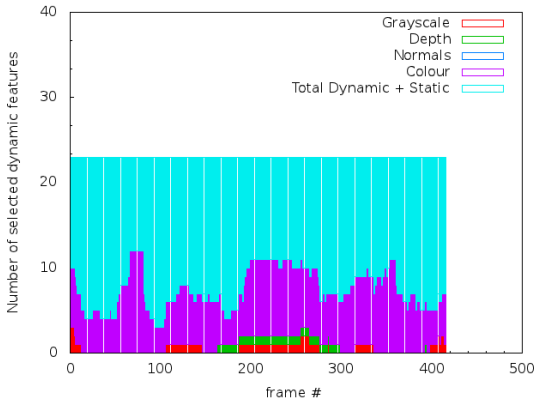

Figure 5.23: Sequence N: selected dynamic features

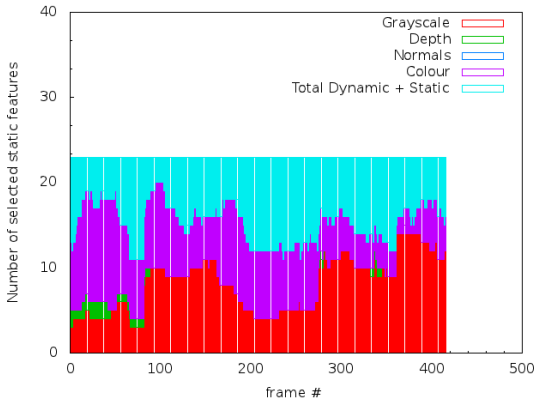

Figure 5.24: Sequence N: selected static features

Average values	Min	First quartile	Mean	Third quartile	Max	Mean
Static grays	0.0543	0.0922	0.1033	0.1141	0.1521	0.1035
Dynamic grays	0.0783	0.0889	0.0958	0.1035	0.1147	0.0962
Static depths	0.0638	0.1146	0.1443	0.1639	0.2035	0.1402
Dynamic depths	0.0754	0.0854	0.0932	0.1038	0.1168	0.0950
Static colours	0.0073	0.0382	0.0695	0.1272	0.2425	0.0904
Dynamic colours	0.0071	0.0212	0.0279	0.0367	0.0948	0.0303

Table 5.8: Sequence N: averaged error statistics

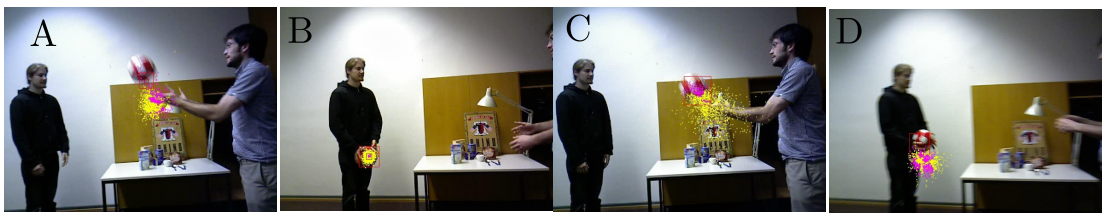

Figure 5.25: A: frame 45, sudden acceleration. B: frame 80, sudden stop. C: frame 125, sudden acceleration. D: frame 335, sudden stop

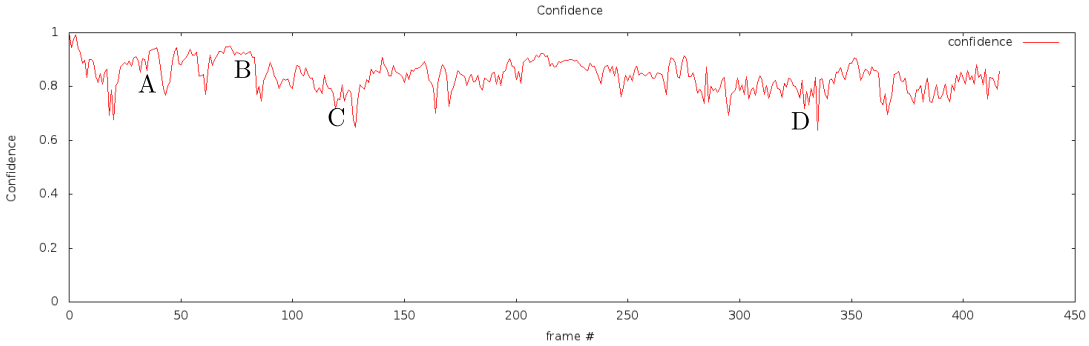

Figure 5.26: Confidence

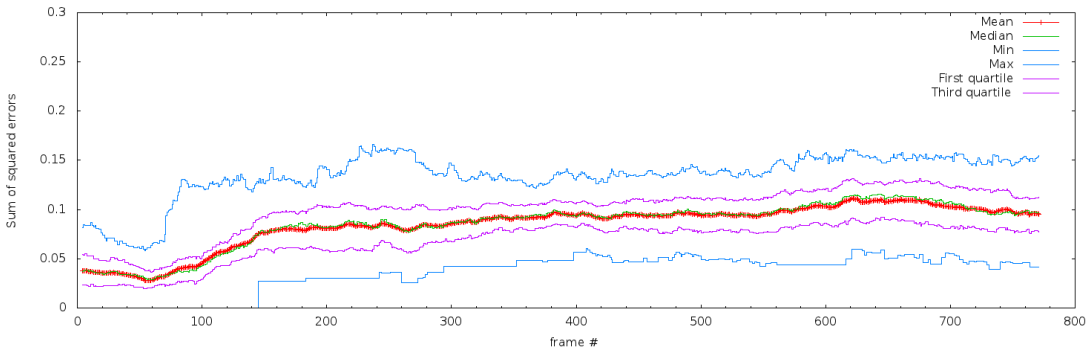

Figure 5.27: Sequence O: static gray features error quartiles distribution

Sequence O (Tank)

The results obtained in this sequence were the following:

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
54,39	55,94	55,51	54,47	55,14	46,92	55,31	53,97	54,69	55,35	56,64

Table 5.9: Sequence O: Overlap percentages over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
91.23	94.30	93.52	90.54	94.82	72.80	92.10	93.26	92.62	93.78	94.56

Table 5.10: Sequence O: Hit rates over runs and average

The test on the tank sequence gave as a result an average of $54,39\%$ overlap and a 91.23% hit rate. This is a relatively low overlap indicator, while the hit rate indicates that the tank is kept on track for most of the time. The explanation has to do with the bounding box that is learned in the first frame: its dimensions in 3D are kept constant throughout the whole process. In the first frame the tank is learned from a side point of view, and when it turns, the bounding box includes greater parts from the background. When evaluating it with the ground truth, the overlap percentage is smaller than in the other sequences.

Looking at the feature kinds that form the classifier, figure 5.34, the grayscale features turned out to be the more successful kind. Their error, in the dynamic and static types (Figures 5.27 and 5.28), is kept below 0.1 throughout the test, and their number is increased in the dynamic pool, Figure 5.35. An interesting effect can be observed in the selected classifiers' plots, Figures 5.36 and 5.35. Comparing it to the other sequences there are more frequent "ups and downs",

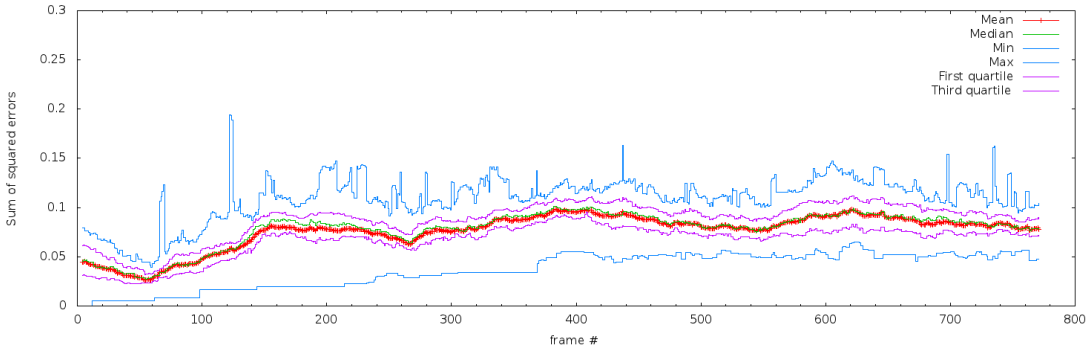

Figure 5.28: Sequence O: dynamic gray features error quartiles distribution

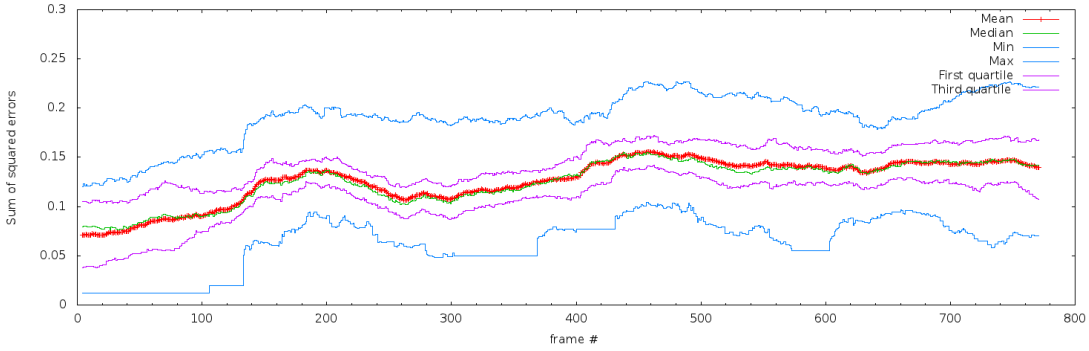

Figure 5.29: Sequence O: static depth features error quartiles distribution

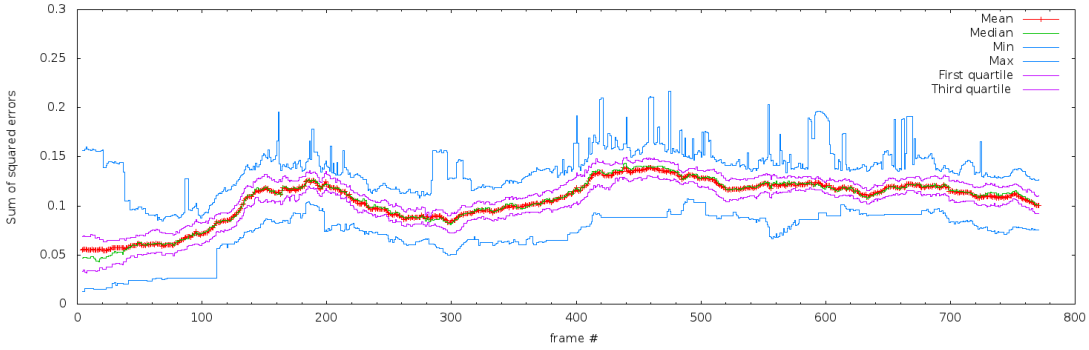

Figure 5.30: Sequence O: dynamic depth features error quartiles distribution

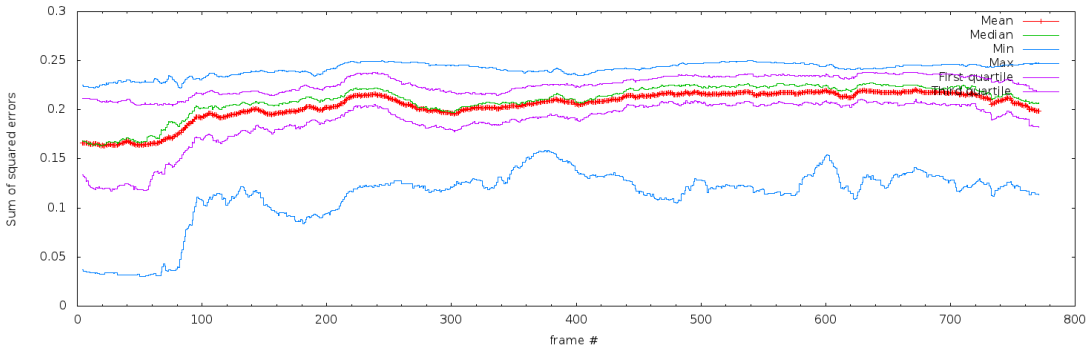

Figure 5.31: Sequence O: static colour features error quartiles distribution

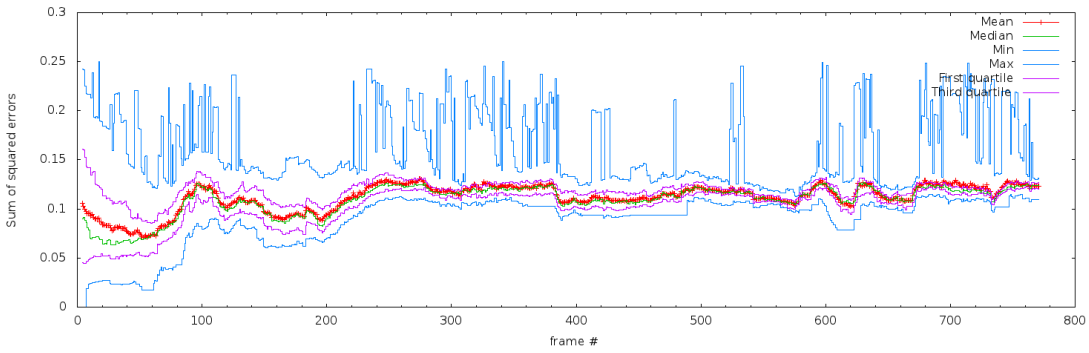

Figure 5.32: Sequence O: dynamic colour features error quartiles distribution

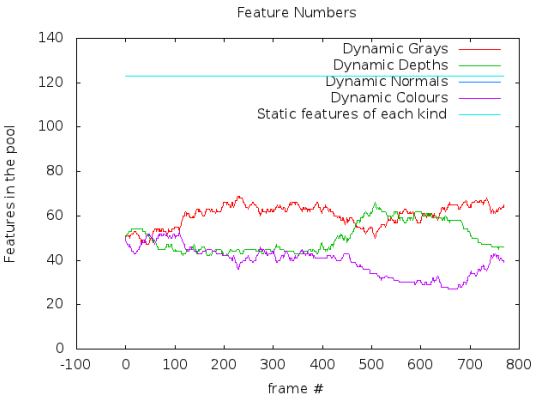

Figure 5.33: Sequence O: static and dynamic features in the pool

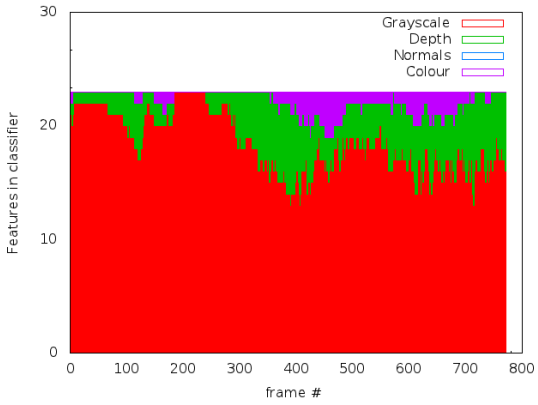

Figure 5.34: Sequence O: selected features

even on a per-frame basis. One explanation is that the tank is changing its pose on a frame to frame basis, and each time new features are included in the classifier. Another possible reason, is that the error distributions are all in similar lower bounds.

Average values	Min	First quartile	Mean	Third quartile	Max	Mean
Static grays	0.0357	0.0675	0.0853	0.1003	0.1341	0.0844
Dynamic grays	0.0368	0.0666	0.0782	0.0877	0.1091	0.0762
Static depths	0.0634	0.1068	0.1248	0.1449	0.1902	0.1264
Dynamic depths	0.0716	0.0962	0.1058	0.1148	0.1386	0.1057
Static colours	0.1121	0.1878	0.2104	0.2257	0.2416	0.2050
Dynamic colours	0.0891	0.1017	0.1091	0.1193	0.1625	0.1122

Table 5.11: Sequence Q: averaged error statistics

In Figure 5.37 and 5.38 we show some selected frames of the sequence together with a graph of the confidence of the classifier on the target. The sequence consists of 771 frames. The sequence starts with the tank running over the bridge. There we see the first two drop in the confidence (point A in the confidence plot) at frames 120 and 140; this is due to the new textured background found on the bridge. The next difficulty arises around frame 210 (point B), when the tank turns around, shows its front side, and its new appearance makes the classifier doubtful, with a confidence of about 0.5 on the target. When the turn is completed, the high symmetry of the object lets the classifier recover its confidence (frame 250).

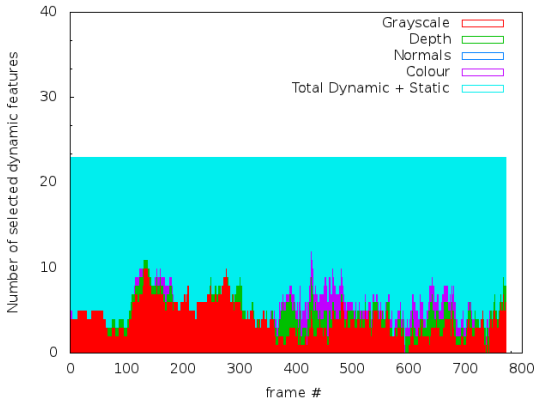

Figure 5.35: Sequence O: selected dynamic features

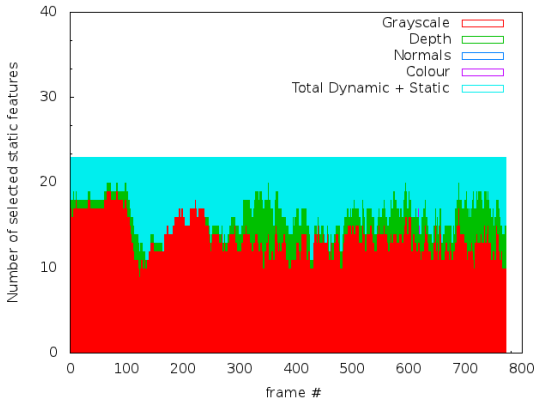

Figure 5.36: Sequence O: selected static features

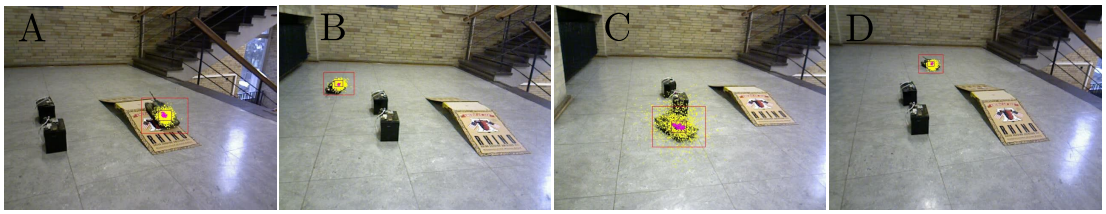

Figure 5.37: A: the tank goes over the bridge. B: it turns around. C: goes before a battery. D: turns to frontal view

Another drop happens around frame 280 when the tank passes before a battery that enters the bounding box of the target; point C in the confidence plot and in image 5.37. Between frames 470 and 490, point D, we see the confidence slightly decaying. This is caused by the tank in a position that has the least distinguishable features and includes more of the background: the front side view. This happens because the first estimate has been learned with a side view of an object that is longer than wide, when the object turns to a front view, the bounding box still maintains its dimensions but the appearance of the object actually changed; now including more parts of the background into the estimate. The next significant drop happens between frames 570 and 580, point E, when the tank is slowly moving over the bridge.

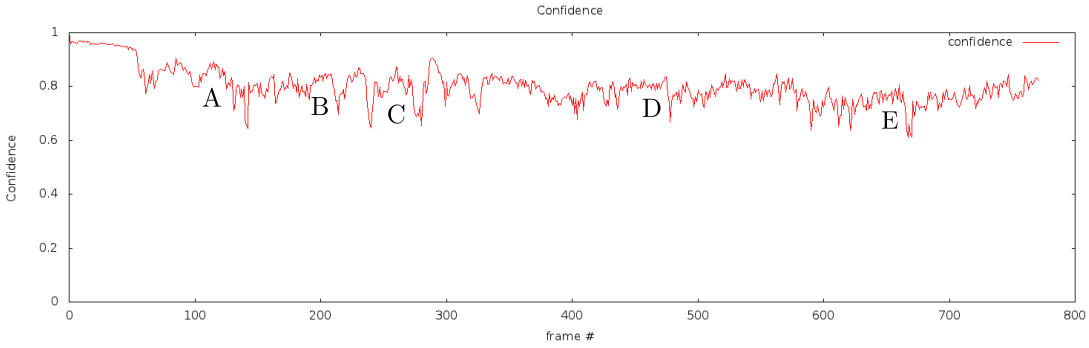

Figure 5.38: Confidence

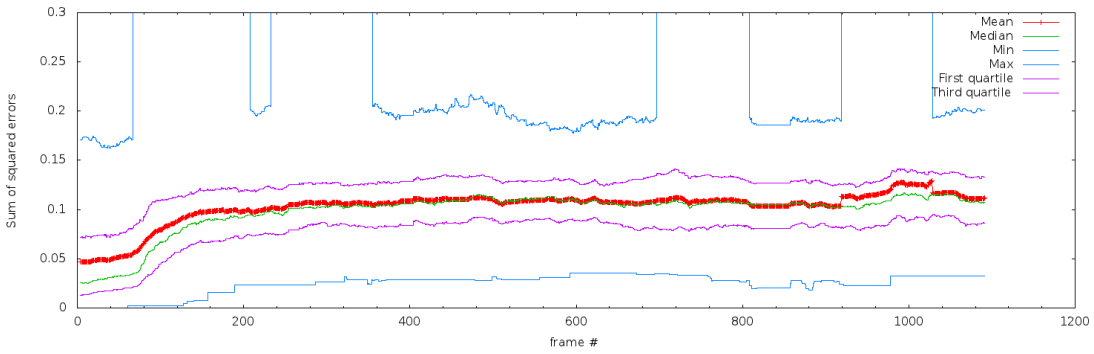

Figure 5.39: Sequence P: static gray features error quartiles distribution

Sequence P (Person)

The results obtained in this sequence were the following:

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
71,80	72,35	72,49	72,19	72,33	71,55	71,19	71,77	72,49	72,69	68,96

Table 5.12: Sequence P: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
95.38	95.42	95.42	95.24	95.51	95.42	95.33	95.33	95.33	95.24	95.60

Table 5.13: Sequence P: Hit rate over runs and average

The test on the chasing sequence gave as a result an average of 71.80% overlap and a 95.38% hit rate. The scene contains the difficulty of partial and total occlusions of the target.

Among the training error graphs, the static colour features pops out, Figure 5.43. It remains almost flat and piled upon 0.2 for the whole sequence; this means that the static colour features were not able to classify the training set much better than with a random decision. It can also be seen in Figure 5.48 that no static colour features ever belonged to the classifier. Instead, the dynamic colour features, after frame 100 start to be more successful and end up being about one fourth of the total features in the classifier.

Why the depth features are so unsuccessful is explained by the fact that, although the person is full of valid depth measurements, all its surrounding is not.

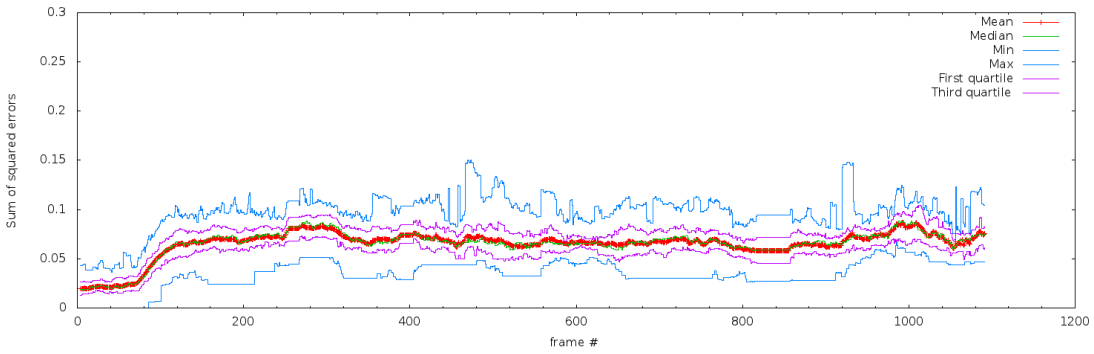

Figure 5.40: Sequence P: dynamic gray features error quartiles distribution

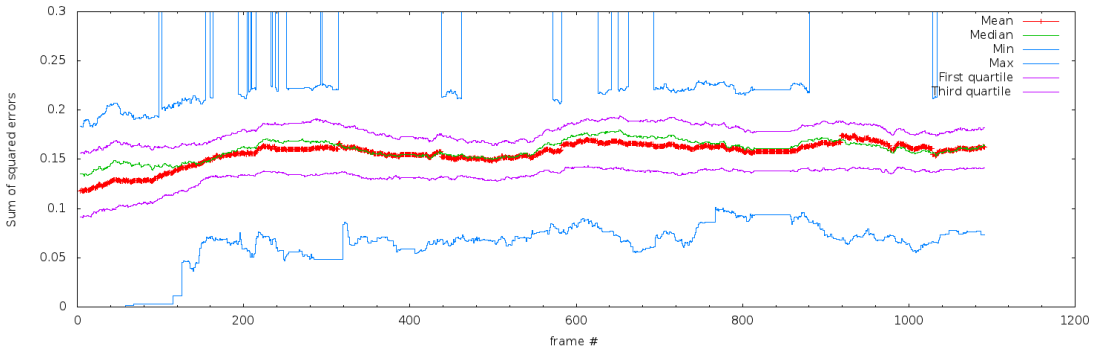

Figure 5.41: Sequence P: static depth features error quartiles distribution

To see this, one can refer back to Figure 5.5, where Sequence P was presented with the RGB and depth images. All the black pixels around the person are invalid measurements. This makes all the features in the contour useless.

In figures 5.49 and 5.50 we show some selected frames of the sequence together with a graph of the confidence of the classifier on the target. The sequence consists of 1196 frames. It was recorded with the camera mounted on a mobile platform moving after the target down a corridor; at different points in time three persons cross on the way. At the beginning, the confidence of the classifier stays very stable while the target is still. After frame 80 approximately, the target starts to walk and the confidence drops close to 0.8. The first occlusion appears around frame 240, point A in images 5.49 and 5.50; the confidence of the classifier descends until around frame 270 where the target is on track again. The next occlusion starts around frame 380 and continues until frame 420, and corresponds to point B in images 5.49 and 5.50.

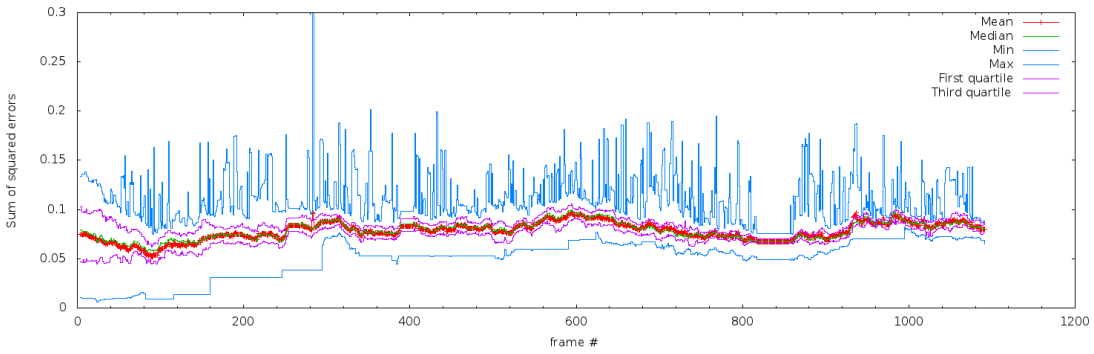

Figure 5.42: Sequence P: dynamic depth features error quartiles distribution

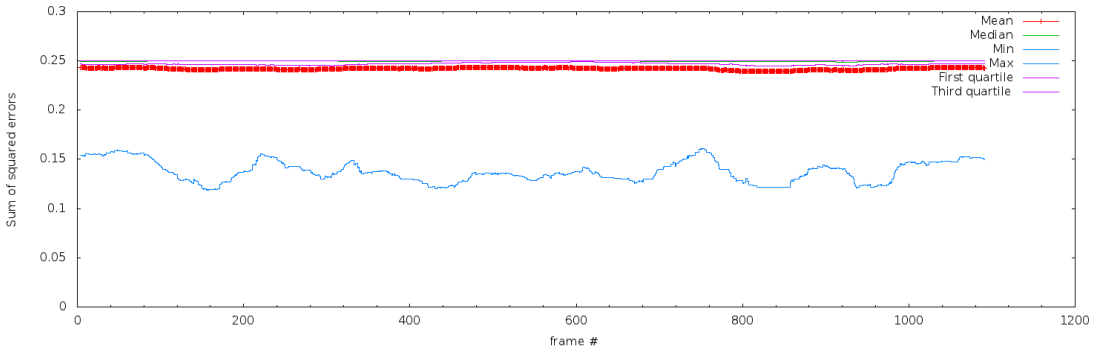

Figure 5.43: Sequence P: static colour features error quartiles distribution

Between frames 710 and 720 (point C in the sequence) there is fast descent on the confidence. It is caused by a bump on the floor that made the camera shake intensely for some frames. The next event to happen is a total occlusion that lasts for several frames. This happens in point D in Figure 5.50 , where the confidence of the classifier drops below 0.5, until the particles are able to reach the target again. It is interesting to comment here an effect that is directly related to the correction of the particles' positions (Section 3.1.2). Right after the target is occluded the particles are displaced to the right hand side of the scene, while the target keeps in the center and the occluding person keeps walking in between them at a shorter distance from the camera. This acts as a sort of potential barrier: the particles that happen to move to the space occupied by the occluding person, see that their velocities are corrected in the position adjustment process, and in the next frame they "fly away" of the scene. Only after some frames this effect ceases and the target is on track again.

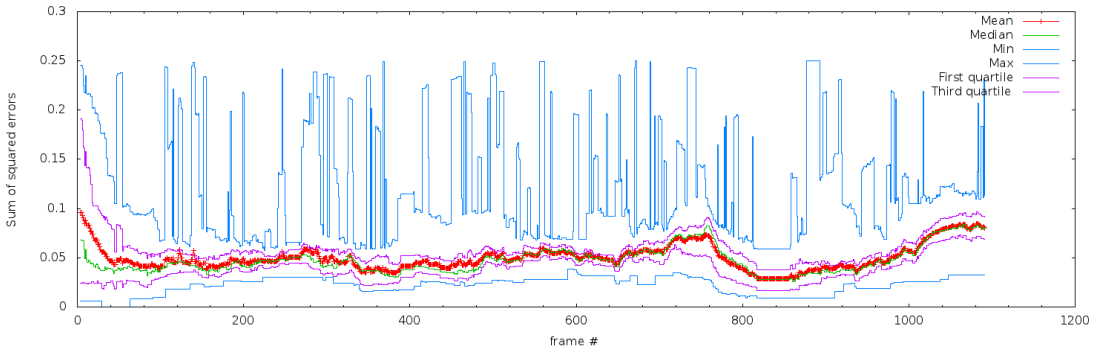

Figure 5.44: Sequence P: dynamic colour features error quartiles distribution

Average values	Min	First quartile	Mean	Third quartile	Max	Mean
Static grays	0.0245	0.0769	0.0981	0.1244	0.5502	0.1031
Dynamic grays	0.0345	0.0545	0.0653	0.0757	0.0969	0.0652
Static depths	0.0629	0.1322	0.1601	0.1780	0.6442	0.1564
Dynamic depths	0.0501	0.0711	0.0786	0.0856	0.1134	0.0783
Static colours	0.1375	0.2468	0.2495	0.2499	0.2500	0.2426
Dynamic colours	0.0216	0.0376	0.0474	0.0605	0.1221	0.0505

Table 5.14: Sequence P: averaged error statistics

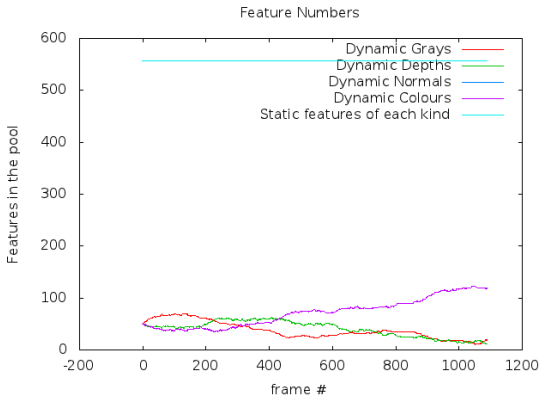

Figure 5.45: chase sequence: static and dynamic features in the pool

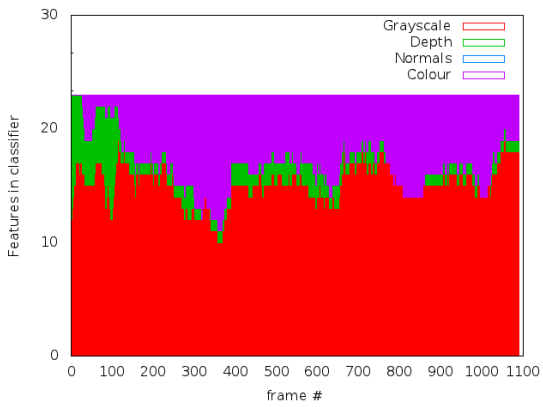

Figure 5.46: chase sequence: selected features

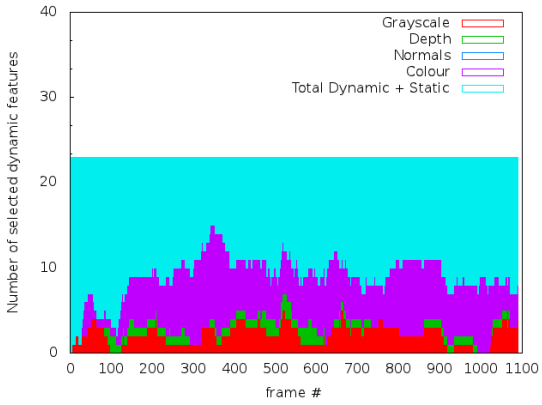

Figure 5.47: Box sequence: selected dynamic features

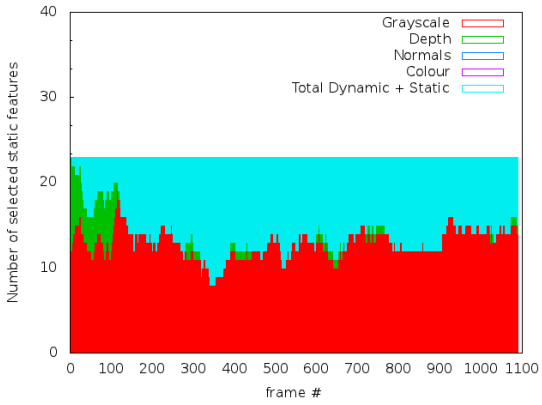

Figure 5.48: chase sequence: selected static features

Figure 5.49: A: first occlusion. B: second occlusion. C: the recording platform shakes and the image becomes blurry. D: a full occlusion

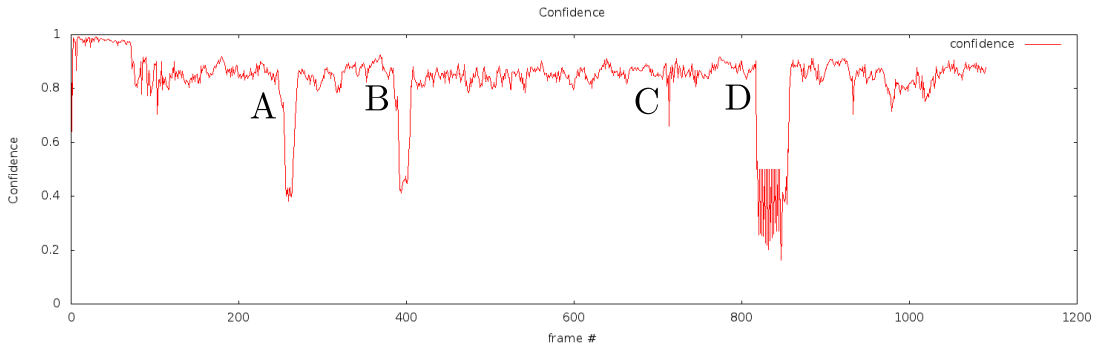

Figure 5.50: Chase sequence: confidence

Sequence Q (Lunch Box)

The results obtained in this sequence were the following:

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
75,21	83,09	71,23	71,24	78,01	62,93	80,22	80,70	78,55	76,47	69,55

Table 5.15: Sequence Q: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
99.98	100.00	100.00	100.00	99.83	100.00	100.00	100.00	100.00	100.00	100.00

Table 5.16: Sequence Q: Hit rate over runs and average

The test on the lunch box sequence gave as a result an average of 75.21% overlap and a 99.98% hit. The intention of this sequence is to test the utility of the depth features. Figure 5.52 shows the selected feature kinds in the classifier. It can be seen that grayscale and depth features are the most prominent ones.

Figure 5.55 contains four relevant frames of the sequence. It consists of 570 frames. The confidence graph, Figure 5.56, contains the corresponding frames notated on it. The sequence starts with the object laying on a shelf. After frame 120 the object is grabbed from the shelf and lifted, this coincides with an increase in the number of chosen depth features. Their number, however, becomes less significant as the object approaches the board: frame 220 and point B in the image sequences of Figure 5.55. The reason is that the gradient values of features on the contour of the object become less strong the closer the object to the background is. While the object is being carried before the wall, the hand has been incorporated by the classifier, which gives a hint on why by the end of the sequence there are more selected grayscales than depth features. Towards the end, around frame 470 a lighting change occurs but the classifier is able to cope well with it. The colour features don't play a significant role here and are rarely included in the classifier as can be seen in Figure 5.52.

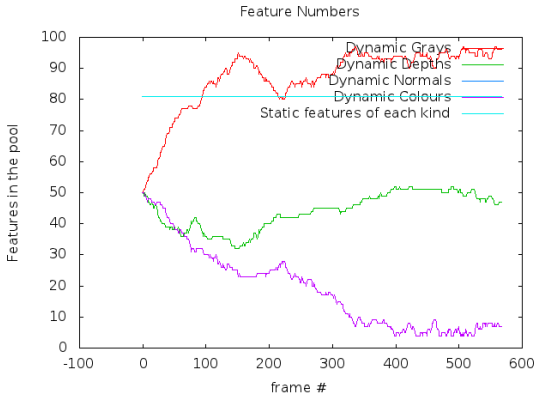

Figure 5.51: Sequence Q: static and dynamic features in the pool

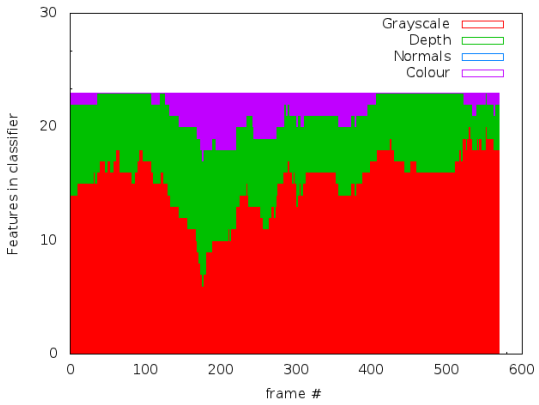

Figure 5.52: Sequence Q: selected features

Average values	Min	First quartile	Mean	Third quartile	Max	Mean
Static grays	0.0322	0.0661	0.0818	0.0963	0.1287	0.0810
Dynamic grays	0.0277	0.0635	0.0774	0.0901	0.1148	0.0765
Static depths	0.0244	0.0916	0.1198	0.1877	0.2181	0.1313
Dynamic depths	0.0153	0.0737	0.1038	0.1215	0.1739	0.0979
Static colours	0.0701	0.1190	0.1526	0.1747	0.2240	0.1477
Dynamic colours	0.0587	0.0842	0.1036	0.1334	0.1759	0.1085

Table 5.17: Sequence Q: averaged error statistics

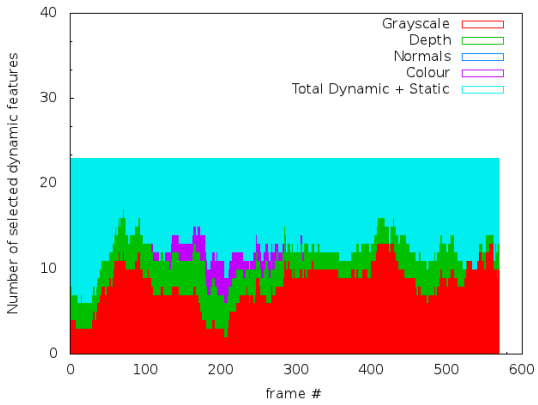

Figure 5.53: Sequence Q: selected dynamic features

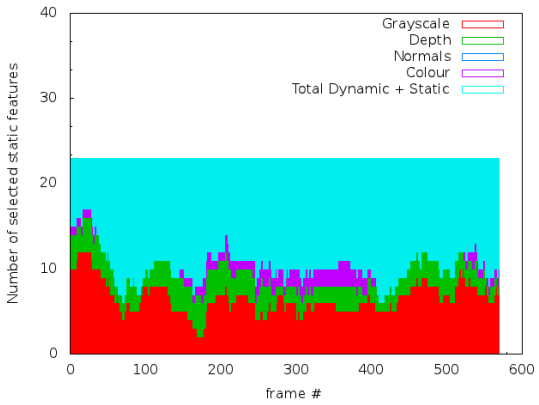

Figure 5.54: Sequence Q: selected static features

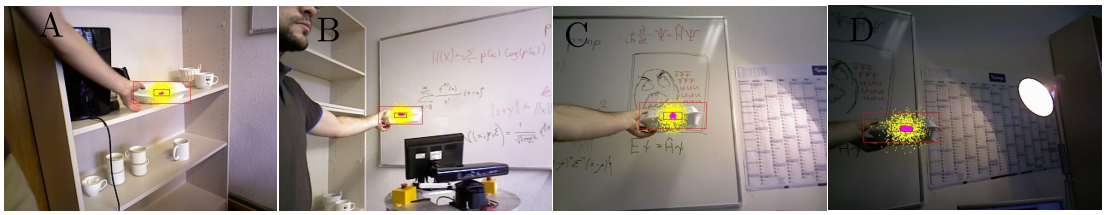

Figure 5.55: A: frame 75, the hand enters the target estimate. B: frame 225, the object is moved before the wall. C: frame 470, the lightning conditions change. D: frame 520, the illumination becomes very poor

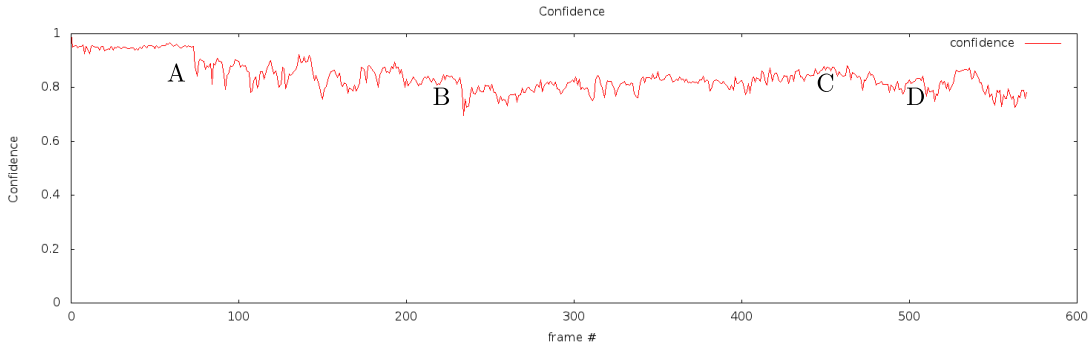

Figure 5.56: Confidence

5.3.2 Test Set 2: Colour, Depth, Grayscale and Surface Normal Features

In this experiment, the performance of the surface normals features is to be tested. It was performed with the following parameters:

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
true	true	true	true
particleState2D	**numParticles**	**numWC**	**numPosEx**
false	1500	23	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
125	0.56	false	false
boost_max	**boost_min**	**std_dev**	
6	1	550	

With these parameters, real time could not be achieved. In fact, the introduction of the surface normal features almost halves the speed. On average, every callback function took 89 ms. Therefore, real time was simulated by providing frames at a speed lower than 25 Hz. The purpose of this is to evaluate the performance of the surface normal features in the same conditions of frames availability as the others.

The following sections will show the hit average obtained in each sequence, together with the feature types that formed the classifier:

Sequence M (Milk)

The test on the milk sequence gave as a result a 73.98% overlap average and a 99.81% hit rate. Table 5.18 and 5.19 show the overlap and hit rates in the ten executions. The error distributions of the normals features can be seen in Figure 5.59 and 5.60.

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
73,98	75,85	75,38	70,41	75,44	74,44	73,69	74,70	72,16	74,24	73,46

Table 5.18: Sequence M: Overlap percentages over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
99.81	100.00	100.00	98.77	100.00	100.00	100.00	99.54	99.85	100.00	100.00

Table 5.19: Sequence M: Hit rates over runs and average

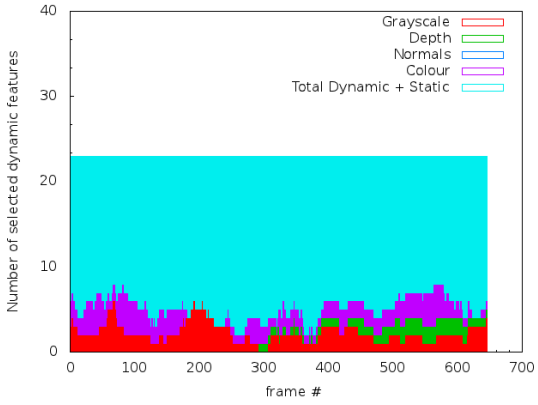

Figure 5.57: Sequence M: selected dynamic features

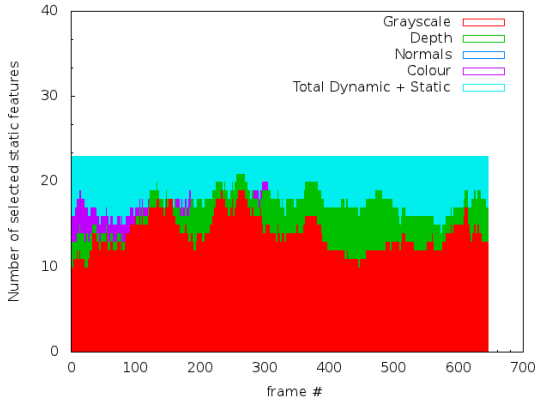

Figure 5.58: Sequence M: selected static features

Sequence N (Ball)

The test on the ball sequence gave as a result a 67.94% overlap average and a 97.42% hit rate. Table 5.21 and 5.20 show the overlap and hit rates in the ten executions. The selected features in the classifier are shown in Figure 5.78. The error distributions of the normals features can be seen in Figure 5.63 and 5.64.

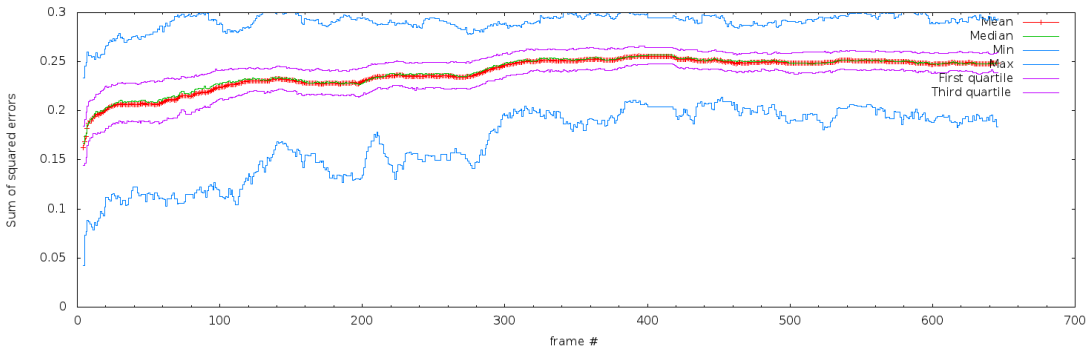

Figure 5.59: Sequence M: static normal features error quartiles distribution

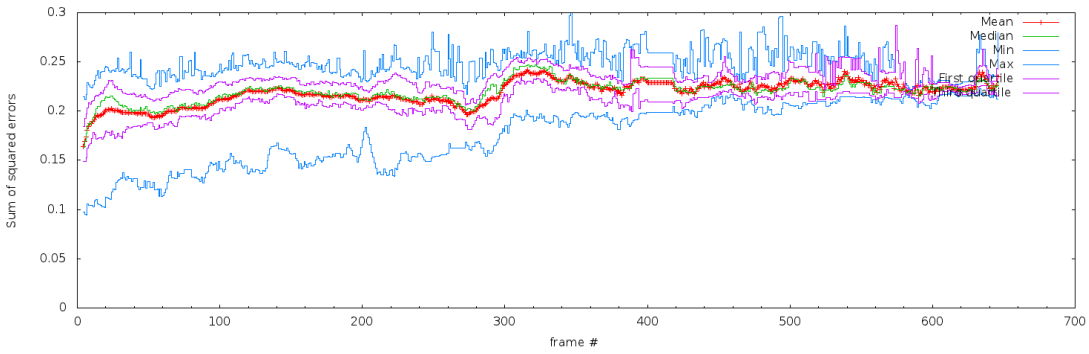

Figure 5.60: Sequence M: dynamic normal features error quartiles distribution

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
67,94	61,91	66,83	69,36	71,77	66,78	72,90	67,37	66,52	71,43	64,56

Table 5.20: Sequence N: Overlap rate over runs and average

Sequence O (Tank)

The test on the tank sequence gave as a result a 53.94% overlap average and a 91.37% hit rate. Table 5.23 and 5.22 show the overlap and hit rates in the ten executions. The selected features in the classifier are shown in Figure **??**.

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
97.42	95.69	97.85	97.37	99.76	92.58	99.76	97.61	97.13	98.80	97.61

Table 5.21: Sequence N: Hit rate over runs and average

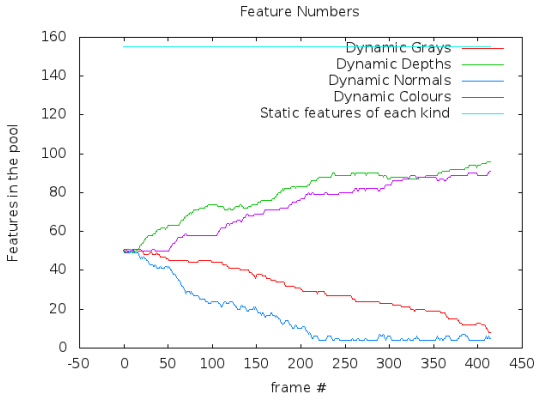

Figure 5.61: Sequence N: static and dynamic features in the pool

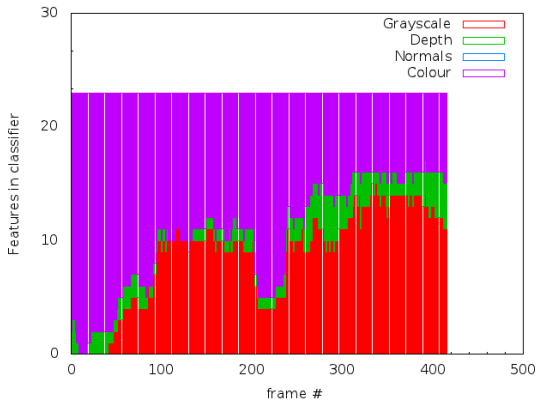

Figure 5.62: Sequence N: selected features

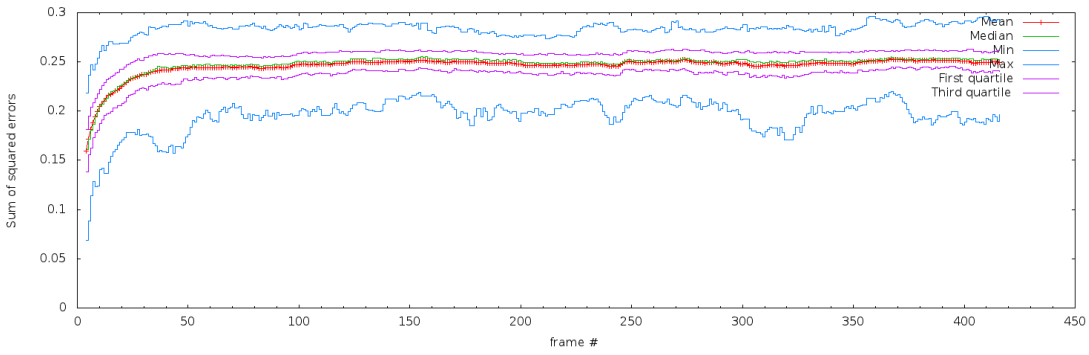

Figure 5.63: Sequence N: static normal features error quartiles distribution

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
53,94	55,05	56,04	55,48	55,97	53,48	46,34	55,59	53,06	52,85	55,57

Table 5.22: Sequence O: Overlap percentages over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
91.37	94.05	94.57	93.66	94.44	93.92	74.13	90.94	91.33	92.24	94.44

Table 5.23: Sequence O: Hit rates over runs and average

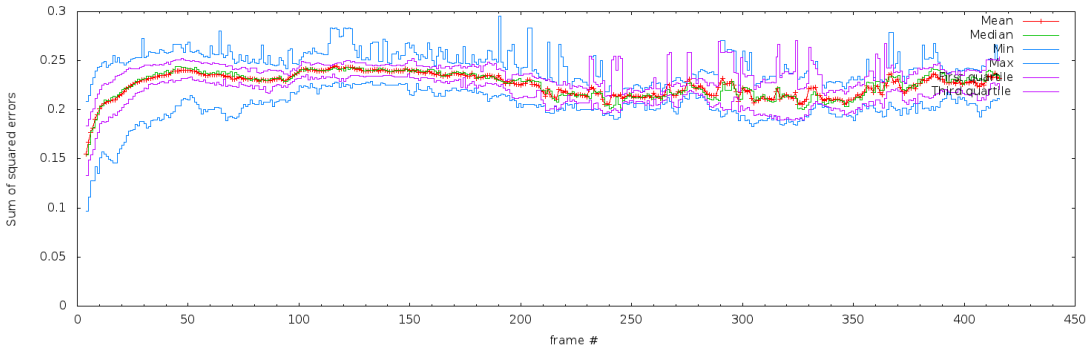

Figure 5.64: Sequence N: dynamic normal features error quartiles distribution

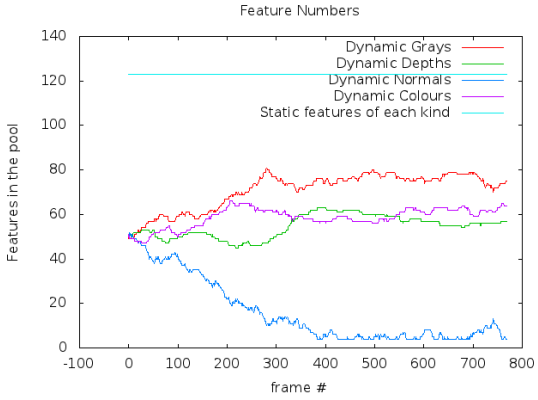

Figure 5.65: Sequence O: static and dynamic features in the pool

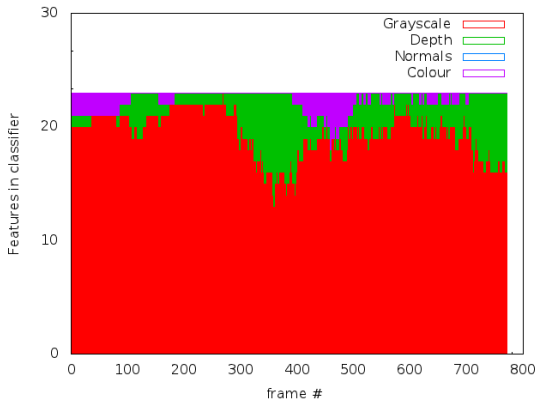

Figure 5.66: Sequence O: selected features

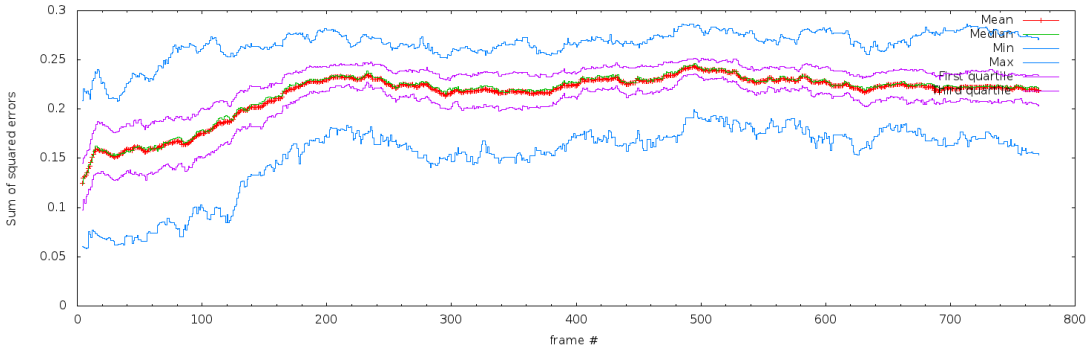

Figure 5.67: Sequence O: static normal features error quartiles distribution

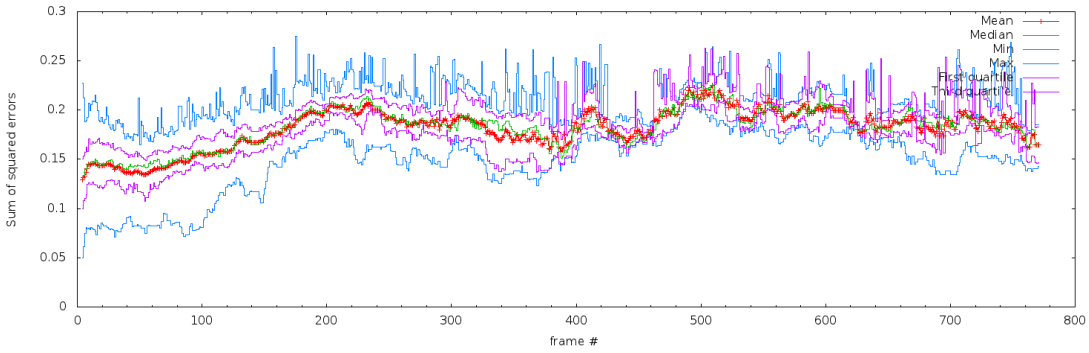

Figure 5.68: Sequence O: dynamic normal features error quartiles distribution

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
72,33	72,85	72,69	72,31	72,03	72,53	72,65	72,52	70,72	72,43	72,61

Table 5.24: Sequence P: Overlap percentages over runs and average

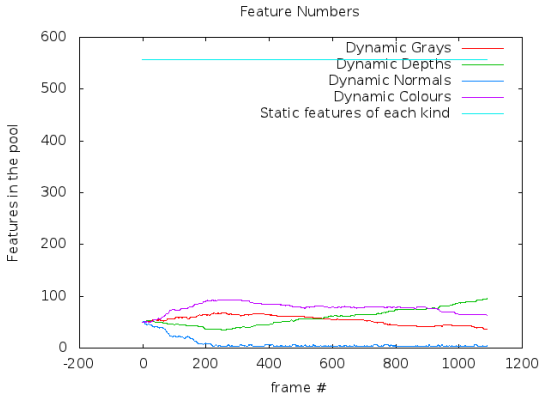

Figure 5.69: Sequence P: static and dynamic features in the pool

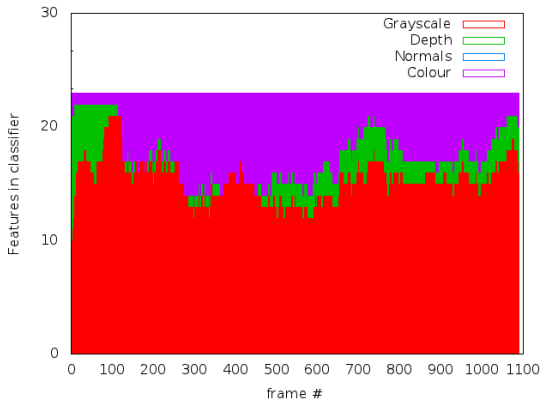

Figure 5.70: Sequence P: selected features

Sequence P (Person)

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
95.41	95.43	95.52	95.24	95.43	95.43	95.33	95.43	95.43	95.43	95.43

Table 5.25: Sequence P: Hit rates over runs and average

The test on the person sequence gave as a result a 72.33% overlap average and a 95.41% hit rate. Table 5.25 and 5.24 show the overlap and hit rates in the ten executions. The error distributions of the normals features can be seen in Figure 5.71 and 5.72.

Sequence Q (Box)

The test on the ball sequence gave as a result a 74.47% overlap average and a 100.00% hit rate. Table 5.21 and 5.26 show the overlap and hit rates in the ten executions. The selected features in the classifier are shown in Figure 5.81. The error distributions of the normals features can be seen in Figure 5.75 and 5.76.

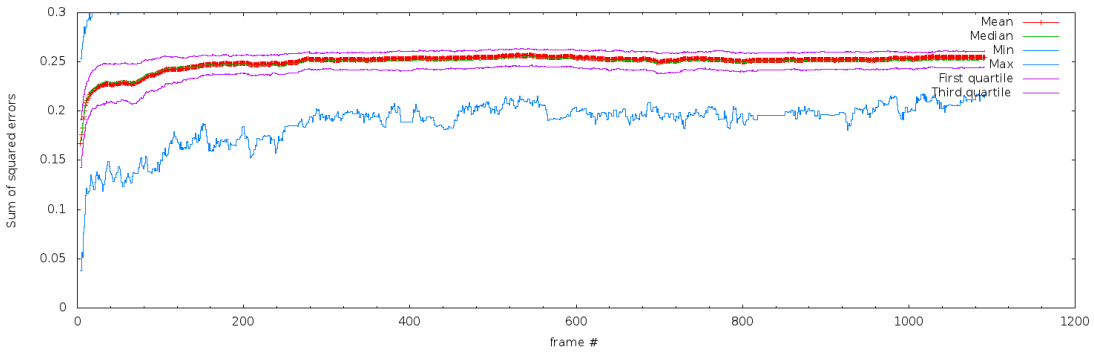

Figure 5.71: Sequence P: static normal features error quartiles distribution

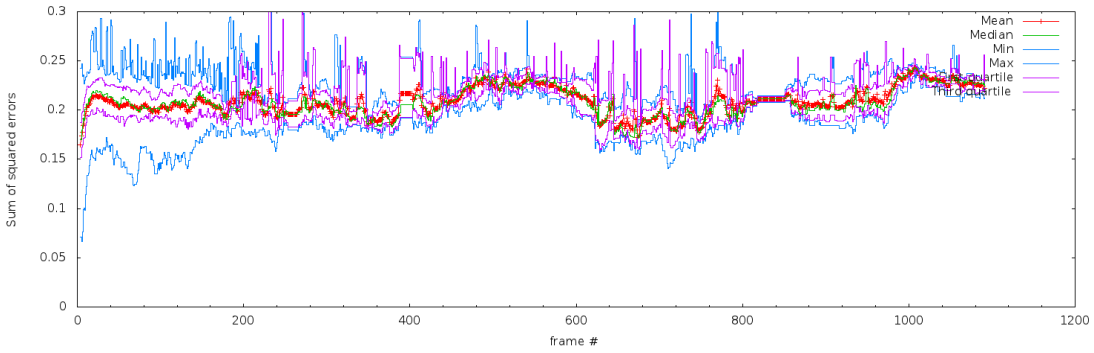

Figure 5.72: Sequence P: dynamic normal features error quartiles distribution

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
74,47	63,89	80,35	81,52	68,04	74,96	74,51	78,39	78,63	76,28	68,17

Table 5.26: Sequence Q: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 5.27: Sequence Q: Hit rate over runs and average

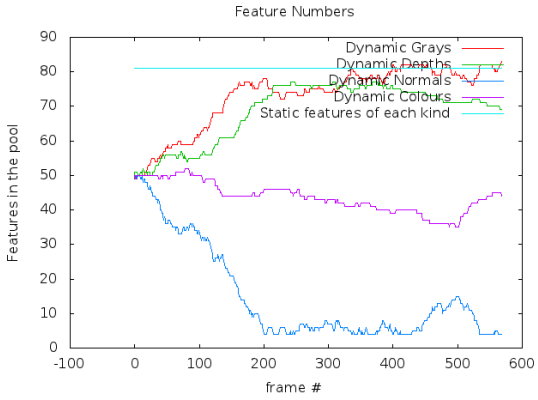

Figure 5.73: Sequence Q: static and dynamic features in the pool

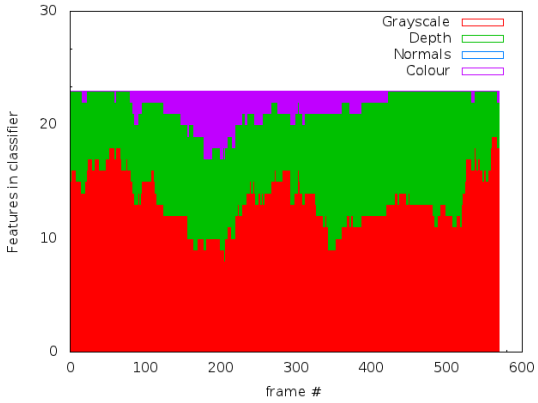

Figure 5.74: Sequence Q: total selected features

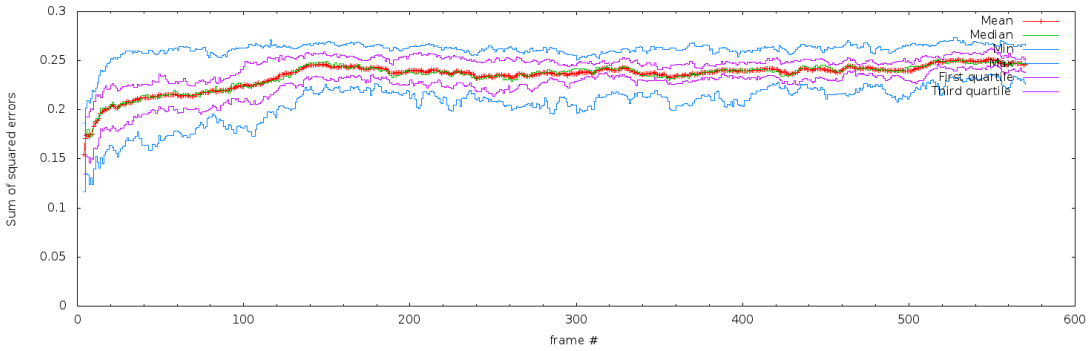

Figure 5.75: Sequence Q: static normal features error quartiles distribution

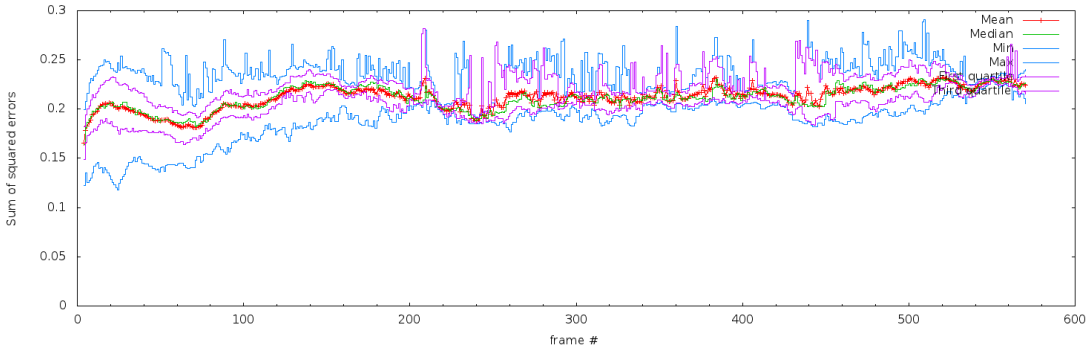

Figure 5.76: Sequence Q: dynamic normal features error quartiles distribution

The surface normals did not succeed to be chosen by the classifier. There are several possible reasons. The first, having to do with the chosen number of grids in the classifier, Section 3.4.2. Eight bins are chosen in each of the three grid dimensions, which makes a total of 512 bins. A lower number of bins, three or four, might improve the classifier performance; as well as to help to deal with the noisy nature of the Kinect data. Another strong reason has to do with the negative and positive priors. Referring again to section 3.4.2, there are two distributions: those of the positive and the negative examples. The positive examples distribution is set to 0.0 everywhere. The negative examples distribution is initialised with a value greater than 0.0, in this test set it was set to 0.15. This was done to prevent the classifier from over-fitting to the training set. Setting equal priors to both distributions led the classifier to over-fit to the training data.

A different way on how to exploit the surface normal features would be to use histograms over regions. This would be a more robust approach compared to the pair-wise comparison of features, but would also require some effort on how to integrate it with the current framework.

5.3.3 Test Set 3: Dynamic Feature Pool Behaviour Disabled with Colour, Depth and Grayscale Features

In this experiment, the dynamic pool behaviour is disabled. To be able to compare with previous results, the same 50 dynamic features per kind are initially added to the pool, but no pool update mechanism is executed. The following parameters were used:

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
false	true	true	true
particleState2D	**numParticles**	**numWC**	**numPosEx**
false	1500	23	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
125	0.56	false	**true**
boost_max	**boost_min**	**std_dev**	
6	1	550	

Sequence M (Milk)

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
74,61	73,73	75,34	74,13	75,78	73,96	74,75	74,39	74,75	74,31	74,95

Table 5.28: Sequence M: Overlap percentages over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
99.78	100.00	100.00	100.00	100.00	97.84	100.00	100.00	100.00	100.00	100.00

Table 5.29: Sequence M: Hit rates over runs and average

The test on the milk sequence gave as a result a 74.61% overlap average and a 99.78% hit rate. Table 5.28 and 5.29 show the overlap and hit rates in the ten executions. The selected features in the classifier are shown in Figure 5.77.

Sequence N (Ball)

The test on the ball sequence gave as a result a 69.29% overlap average and a 97.56% hit rate. These results can be compared to those obtained when the

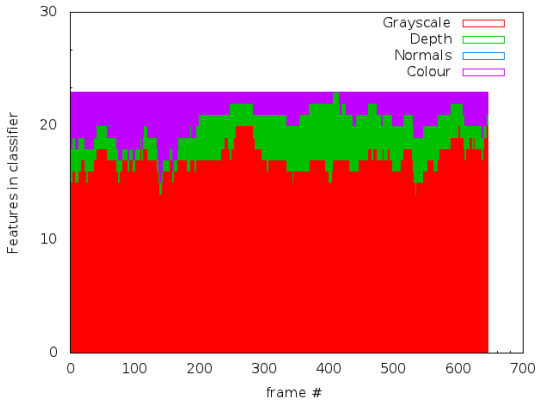

Figure 5.77: Sequence M: selected features

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
69,29	74,30	70,27	66,47	69,58	72,25	64,16	70,88	66,34	73,49	65,13

Table 5.30: Sequence N: Overlap percentages over runs and average

dynamic pool was enabled having a 67.78% overlap average and a 96.53% hit rate, see Section 5.3.1. Table 5.31 and 5.30 show the overlap and hit rates in the ten executions. The selected features in the classifier are shown in Figure 5.78.

Sequence O (Tank)

The test on the tank sequence gave as a result a 54.85% overlap average and a 94.11% hit rate. These results can be compared to those obtained when the dynamic pool was enabled having a 54.39% overlap average and a 91.23% hit rate, see Section 5.3.1. In this case, while the overlap averages are almost coincident, the hit rate is slightly higher for the static pool configuration. Table 5.33 and 5.32 show the overlap and hit rates in the ten executions. The selected features in the classifier are shown in Figure 5.79.

Sequence P (Person)

The test on the person sequence gave as a result a 71.62% overlap average and a 95.22% hit rate. These results can be compared to those obtained when the dynamic pool was enabled having a 71.80% overlap average and a 95.38% hit rate, see Section 5.3.1. Table 5.35 and 5.34 show the overlap and hit rates in the ten executions. The selected features in the classifier are shown in Figure 5.80.

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
97.56	99.52	98.09	94.74	97.37	99.04	98.09	98.80	97.13	97.13	95.69

Table 5.31: Sequence N: Hit rates over runs and average

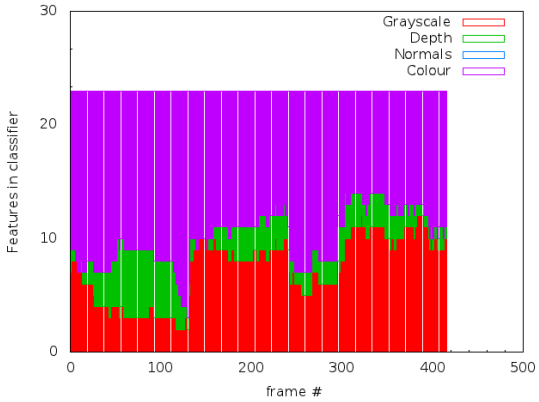

Figure 5.78: Sequence N: selected features

Sequence Q (Box)

The test on the box sequence gave as a result a 76.26% overlap average and a 100.00% hit rate. These results can be compared to the ones obtained when the dynamic pool was enabled having a 75.21% overlap average and a 99.98% hit rate, see Section 5.3.1. Table 5.37 and 5.36 show the overlap and hit rates in the ten executions. The selected features in the classifier are shown in Figure 5.81.

The results obtained in the five sequences, and the comparison to the experiments where the dynamic pool operates does not show any strong evidence in favour or against the use of the dynamic pool. In all cases, the overlap averages and hit rates were on very similar levels.

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
54,85	55,05	55,08	55,03	55,58	52,58	55,35	54,69	55,59	54,47	55,04

Table 5.32: Sequence O: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
94.11	93.92	94.57	93.53	94.31	94.31	94.44	93.27	94.44	94.05	94.3

Table 5.33: Sequence O: Hit rate over runs and average

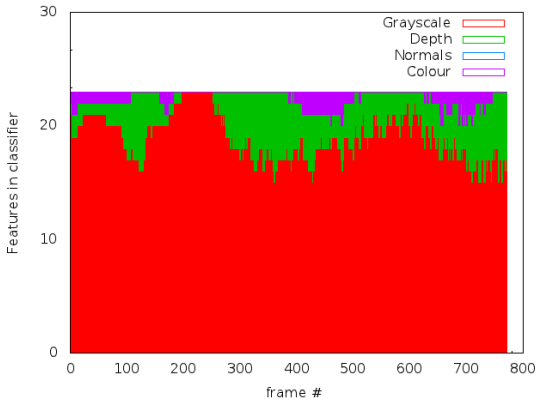

Figure 5.79: Sequence O: total selected features

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
71,62 72,52	72,30	71,62	71,35	72,66	72,01	72,12	71,69	67,50	72,45	

Table 5.34: Sequence P: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
95.22	95.15	95.24	95.24	94.42	95.43	95.24	95.52	95.43	95.33	95.24

Table 5.35: Sequence P: Hit rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
76,26	74,96	78,95	72,83	74,98	77,92	77,71	75,12	75,00	79,52	75,81

Table 5.36: Sequence Q: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 5.37: Sequence Q: Hit rate over runs and average

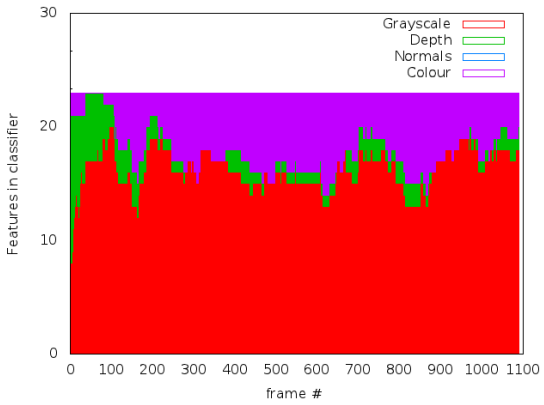

Figure 5.80: Sequence P: selected features

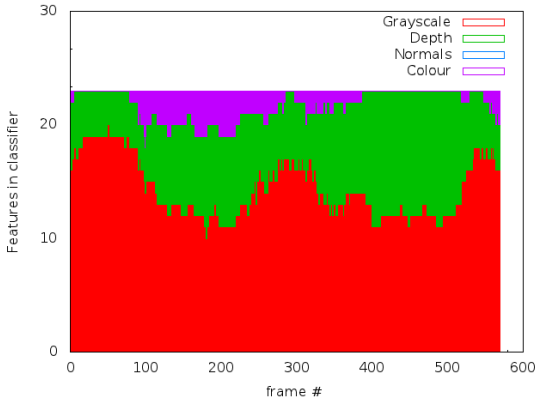

Figure 5.81: Sequence Q: total selected features

5.3.4 Test Set 4: Depth and Grayscale Features

In this experiment, the colour features were removed and the test was run on sequence N, where their presence turned out to be more relevant. This relevance to the tracking process is what is pretended to be shown in this test. The following parameters were used:

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
false	true	false	true
particleState2D	**numParticles**	**numWC**	**numPosEx**
false	1500	23	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
125	0.56	false	false
boost_max	**boost_min**	**std_dev**	
6	1	550	

Sequence N (Ball)

The average overlap over the ten runs was 24.26%. The hit rate was 34.19%. This results can be compared to the ones obtained with the colour features, Section 5.3.1, where the hit rate was 96.53% and the overlap average 67.78%. This difference gives an idea of the importance of the colour features for this specific test. The confidence plot is shown in Figure 5.82. There is a drop in the confidence as soon as the ball starts to move. The classifier plot does not vary over frames because the overlap and confidence conditions never hold, and thus is never updated. See Figure 5.83.

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
24,26	17,57	23,90	16,58	21,08	45,47	36,77	6,90	50,22	12,42	11,67

Table 5.38: Sequence N: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
34.19	23.44	33.49	23.44	28.23	69.14	50.48	9.81	72.73	16.03	15.07

Table 5.39: Sequence N: Hit rate over runs and average

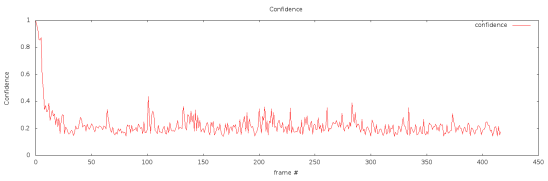

Figure 5.82: Sequence N: confidence

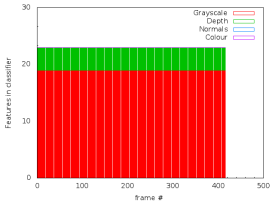

Figure 5.83: Sequence N: selected features

5.3.5 Test Set 5: Colour and Grayscale Features

In this experiment, the depth features were removed and the test was run on sequence Q. The following parameters were used:

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
false	true	true	false
particleState2D	**numParticles**	**numWC**	**numPosEx**
false	1500	23	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
125	0.56	false	false
boost_max	**boost_min**	**std_dev**	
6	1	550	

Sequence Q (Box)

The average overlap over the ten runs was 68.62%. The hit rate was 100.00%. This results can be compared to the ones obtained with the depth features enabled, Section 5.3.1. The hit rate then was 99.98% and there is thus no significant difference between the two. The overlap average was then 75.21%. Removing the depth features caused a drop in the precision of the algorithm.

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
68,62	66,50	75,73	63,42	75,68	75,61	64,96	67,07	68,33	65,28	63,86

Table 5.40: Sequence Q: Overlap percentages over runs and average

5.3.6 Test Set 6: Colour and Depth Features

In this experiment, the grayscale features were removed and the test was run on sequences Q and O. The following parameters were used:

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 5.41: Sequence Q: Hit rates over runs and average

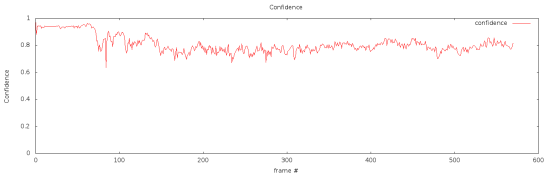

Figure 5.84: Sequence Q: confidence

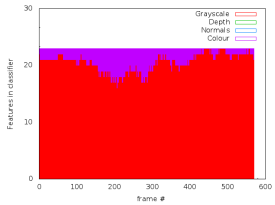

Figure 5.85: Sequence Q: total selected features

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
false	false	true	true
particleState2D	**numParticles**	**numWC**	**numPosEx**
false	1500	23	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
125	0.56	false	false
boost_max	**boost_min**	**std_dev**	
6	1	550	

Sequence Q (Box)

The average overlap over the ten runs was 74.56%. The hit rate was 98.71%. This results can be compared to the ones obtained when the depth features were disabled, Section 5.3.5. The hit rate then was 100.00% and there is thus no significant difference between the two. The overlap average was then 68.62%. The accuracy results in the overlap average are thus in favour of the depth features.

Sequence O (Tank)

The average overlap over the ten runs was 37.68%. The hit rate was 61.22%. When the grayscale features were present the average overlap was 54.39% and the hit rate 91.23%. Figure 5.89 shows how the depth feature kind replaced the grayscale features as the most prominent kind, but with worse results.

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
74,56	70,87	75,26	75,62	75,60	73,96	73,42	77,77	75,60	70,91	76,56

Table 5.42: Sequence Q: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
98.71	95.98	100.00	96.50	100.00	98.25	96.33	100.00	100.00	100.00	100.00

Table 5.43: Sequence Q: Hit rate over runs and average

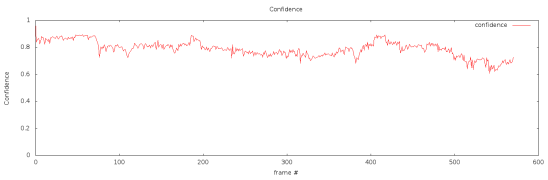

Figure 5.86: Sequence Q: confidence

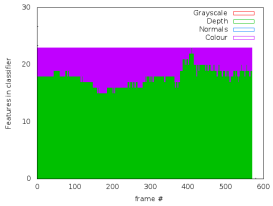

Figure 5.87: Sequence Q: total selected features

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
37,68	16,52	46,42	42,46	50,41	51,50	43,17	47,63	20,47	22,11	36,12

Table 5.44: Sequence O: Overlap rate over runs and average

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
61.22	19.02	79.69	74.39	86.03	91.33	71.54	82.28	23.80	28.33	55.76

Table 5.45: Sequence O: Hit rate over runs and average

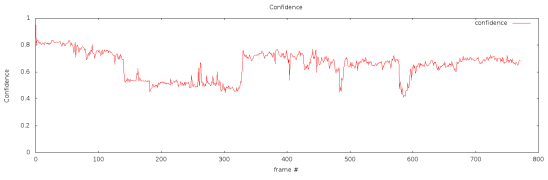

Figure 5.88: Sequence O: confidence plot in run 3

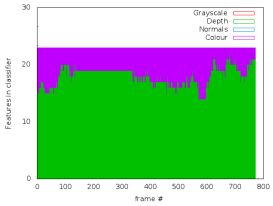

Figure 5.89: Sequence O: selected features in run 3

5.3.7 Test Set 7: 2D State Space and Grayscale Features

In this experiment we want to show the performance of the tracker without any of the improvements proposed in this thesis. Thus, the old 2D motion model was used, the dynamic feature pool disabled, and only the grayscale features were present. The rest of the parameters used were the same as in the other tests:

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
false	true	false	false
particleState2D	**numParticles**	**numWC**	**numPosEx**
true	1500	23	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
125	0.56	false	true
boost_max	**boost_min**	**std_dev**	
6	1	550	

The experiment was run ten times as in the other test cases. The average overlap and hit rates per sequence are displayed in Table 5.46 and 5.47 respectively. Some of the sequences show particularly bad results. This is the case of the ball and the tank sequence. In the first, the absence of colour features already proved in this section to be crucial. In the second, the changing dimensions of the bounding box and the low re-training threshold, made the classifier to be easily adapted to objects from the background. In the person and the box sequences it is remarkable the lower overlap rates obtained, when compared to previous experiments in this section.

M (Milk)	N(Ball)	O (Tank)	P (Person)	Q (Box)
69.88	18.25	24.78	55.26	54.66

Table 5.46: Sequence overlap averages

M (Milk)	N(Ball)	O (Tank)	P (Person)	Q (Box)
95.26	26.05	27.36	76.62	84.34

Table 5.47: Sequence hit rate averages

5.4 *BoBoT* Benchmark

The *BoBoT* benchmark was introduced in [Dominik A. Klein and Cremers, 2010]. It consists of thirteen video sequences where different objects have to be tracked. Figure 5.90 shows a preview of each of the first nine sequences. To evaluate the performance of the tracking algorithm every sequence was run ten times. Each of them was compared to the ground truth and the overlap between the two evaluated; then the average over the ten repetitions is calculated. For this set of experiments we used the old 2D motion model of the particles, since no depth information is available. We run the two complete sets with the Dynamic Feature Pool behaviour enabled and disabled, and then we compare the results.

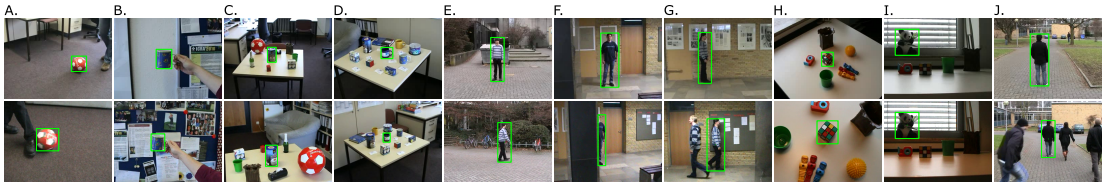

Figure 5.90: BoBoT Benchmark preview

5.4.1 Test case 1: Dynamic Feature Pool Disabled

In this set of tests we evaluate the results of our tracking algorithm on the BoBoT set with the following parameters:

Table 5.48: Static Feature Pool Parameters

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
false	true	true	false
particleState2D	**numParticles**	**numWC**	**numPosEx**
true	1200	24	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
150	0.6	false	true
boost_max	**boost_min**		
4	1		

The average overlap in each of the sequences is displayed in Table 5.49.

Table 5.49: Overlap Rate. Static Feature Pool Test

Seq	A	B	C	D	E	F	G	H	I	J1	J2	K	L	Avg
Avg	77.88	79.86	93.06	66.44	85.75	49.64	77.29	94.69	76.05	82.13	79.42	84.69	42.80	**76.13**

5.4.2 Test case 2: Dynamic Feature Pool Enabled

This set of tests is meant to test the behaviour of the dynamic pool on the *BoBot* benchmark. To do so we will run the tests with the same parameters as in test case 5.4.1, except for the one that concerns the dynamic feature pool.

Table 5.50: Dynamic Feature Pool Parameters

normalsAvailable	**visualsAvailable**	**colourAvailable**	**depthAvailable**
false	true	true	false
particleState2D	**numParticles**	**numWC**	**numPosEx**
true	1200	24	25
numNegEx	**learnThreshold**	**movingAvgMethod**	**onlyStaticPoolBhvr**
150	0.6	false	false
boost_max	**boost_min**		
4	1		

Table 5.51: Overlap Rate. Dynamic Feature Pool Test

Seq	A	B	C	D	E	F	G	H	I	J1	J2	K	L	Avg
Avg	79.84	79.43	92.40	80.42	86.78	65.01	78.12	94.21	83.83	77.57	75.21	84.68	52.84	**79.26**

In 8 out of 13 cases the dynamic pool showed better results. In average, the dynamic pool got a 79.26% overlap, compared to a 76.28% without it. It must be noted, that only two feature kinds were present in the system: the grayscale and colour, since no depth information is available on this benchmark. Therefore, the scope of the dynamic pool behaviour is limited.

However, the D, F and L sequences show particularly high differences between the two approaches. The D sequence shows a person wearing a blue striped jersey that moves before the camera. The dynamic feature pool test showed that about one third of the classifiers in the pool were dynamic features. And many of those were of the colour kind. The difference between the two approaches was 20% higher for the dynamic pool case.

In sequence F a person is walking while several partial and total occlusions occur. In the static pool case, the dynamic features are added to the pool in the first frame but no update mechanism is executed. In the dynamic pool case, the presence of the dynamic features in the classifier is not more than 10% of the total. However, this difference proves to be determinant in the overlap rate. being about 15% higher for the dynamic pool case.

In the L sequence the camera is moving around a table and some objects lay on top of it. The target is a round coffee box. Here, the dynamic features were significantly present in the classifier, and were about 50% of the total number in the classifier. A difference between the two approaches of about 10% in the overlap rate was achieved in favour of the dynamic pool approach.

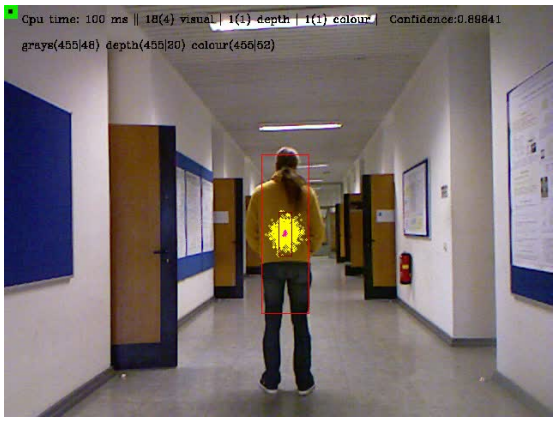

Figure 5.91: On-board tracking frame 1

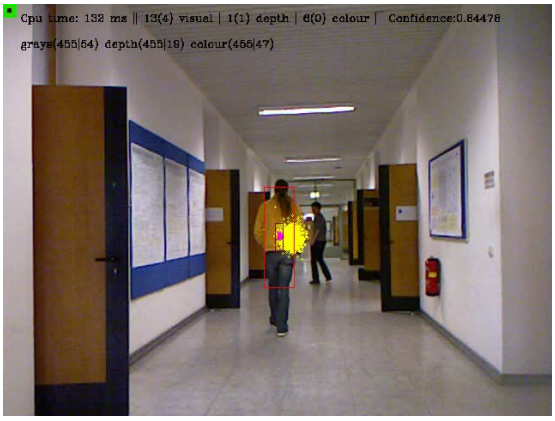

Figure 5.92: On-board tracking frame 2

5.5 On-board Tracking Experiment

In this test we evaluated the performance of the tracking algorithm on board of a mobile platform. For this purpose we used Rhino (4.2). The applied control strategy is the one described in Section 3.6. The experiment comprised following a person down a corridor. The starting and end positions were marked on the ground. We considered an experiment successful if the robot was able to follow the person from the starting position to the end without any human help. We repeated the experiment ten times having the following result in each of them:

Avg	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10
80,00%	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗

We were successful in eight of the ten experiments. In the two experiments that failed, the control scheme was not good enough to keep the robot at a close distance and at some point the target was lost and too far away to be tracked again. We include a set of pictures in figures extracted from the video of an extra eleventh experiment.

Figure 5.93: On-board tracking frame 3

Figure 5.94: On-board tracking frame 4

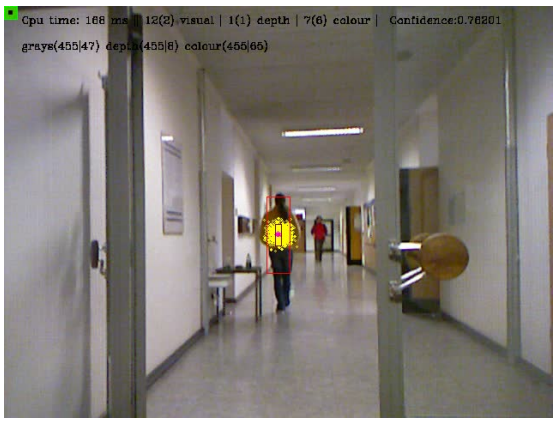

Figure 5.95: On-board tracking frame 5

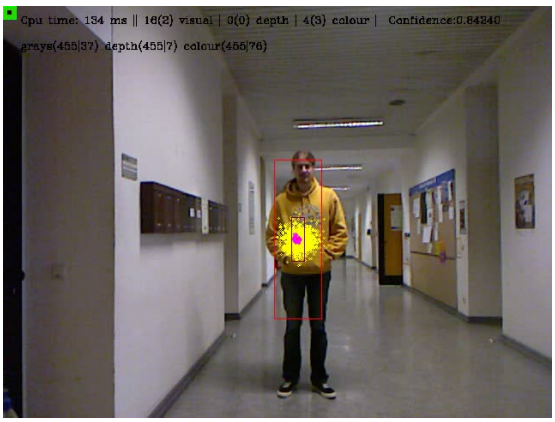

Figure 5.96: On-board tracking frame 6

Chapter 6

Summary

The goal of this thesis was to extend the tracker of [Dominik A. Klein and Cremers, 2010] and [Klein and Cremers, 2011]. For this purpose, several contributions have been proposed. The first of them was to integrate the depth information into the tracker. This new source of information was useful as a new feature kind, and as a way of extending the state space of the particles to the real 3D world. Using depth information allowed for a better estimation of the size that the object will have in the image and a more accurate generation of negative examples.

Two other feature kinds were also proposed. The objective was to capture all the modalities of the appearance of the object: light intensity, colour, depth variations and shape. Thus, the second contribution of this work were the colour features, which estimate average colours over regions. A shape feature was proposed to capture the curvature of the object.

Having a common pool of different feature kinds, the next step was to try to make it specialize to the object modalities that were more prominent. A mechanism was defined to remove those features that were less accurate, and to sample new features of the kind with the least average error over the training set. This strategy had the benefit of improving the quality of the features in the pool while their number is kept constant.

One last contribution involved the control of a robot. A mobile platform was able to follow the movements of a person down a corridor, by means of the target positions reported by the tracking algorithm.

For the evaluation of the system a benchmark was developed consisting of five sequences. Metrics were defined to measure the performance of the algorithm in

quantitative terms. Experiments were performed to show the usefulness of the individual contributions of this thesis, and the results justified the use of the mentioned improvements. A significant performance improvement was achieved when comparing the results of the new approach to the ones obtained by the old one.

6.1 Future Work

The results were encouraging to follow this line of research. An interesting point would be to extend the system to be able to cope with multiple targets. At a first thought, computation time could be a constraint, but there are some resources that could be shared among the different targets, e.g. the negative examples. The classifiers in the pool would need to be extended to be able to tell the feature values of which target they are learning. A benefit of multi-target tracking is that one can deal with occlusions between the targets. For example, if we are tracking two persons and one happens to move behind the other, we can still have some confidence on the position of the occluded person until it becomes visible again.

Another line of research could go in the direction of integrating the tracker for visual servoing. For example, a robotic arm that is able to be driven to the object of interest, grab it, and manipulate it with the feedback provided by the tracker. A multi-target tracker of arbitrary objects can be of a great utility in the field of robotics in general, because it provides the robot with a lot of information about its environment.

Finally, a word about the initialisation of the process. At the moment, it is the user that manually selects the object of interest and the tracking process begins. Object detection could be integrated into the system to trigger the tracker. A multi-target tracker triggered by an object detector could have great applications. How to share the information between the features used by the tracked and the object detector would be a matter of research as well. In fact the new positive examples provided by the tracker could be then integrated by the object detector. This was already mentioned in the papers that preceded this thesis, but is still to be investigated.

Appendix A

Software Environment

A.1 Robot Operating System (ROS)

ROS is a software framework used for robotics. It provides many of the functionalities needed on a robotic environment, such as low-level sensors/actuators abstraction, message passing, synchronization. It is structured in nodes; nodes solve particular tasks and communicate with each other. We quote the introduction page of the ROS website:

> *ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.*
> [ROS, b]

A.1.1 Program Structure in ROS

ROS is a centralised system. In a typical ROS setup we find several nodes running, each solving one particular task. There is a special set of nodes that must be present, in order that the system works; this is the *roscore*. It is the one that is firstly contacted when a node is started and the point through which all messages pass; it is also responsible for logging.

A.1.2 Communication Between Nodes

Interactions between nodes can be achieved via service calls (a sort of "Remote Procedure Call" in ROS), or via message passing. In our system we made use of

the latter.

Messages in ROS are sent over channels known as topics. The node that wants to provide certain information will announce to the roscore that it is able to publish messages of such topic. A node that is interested in having this information will subscribe to this topic. The roscore provides the connectivity information.

This scheme is very common for nodes that run the driver of some sensor. For example a camera. The driver will advertise an *image* topic to the core node. Other nodes will then subscribe to it and receive periodically, via a callback function, the messages that contain the images.

Message Format

Messages contain parameters of different data types. These are the usual types found in other programming languages: floats, integers, booleans, arrays, etc. The format for declaring a message is:

field-datatype1	field1
field-datatype2	field2
field-datatype3	field3
...	

A special parameter is the header. It is optionally included in the message. It contains information such as the time when the message was created *stamp*, what number in the sequence corresponds *seq* and the frame the data is associated with:

uint32 seq
time stamp
string frame_id

For further reading on the topic of ROS messages one can refer to [ROS, d].

A.1.3 Launching Nodes

Nodes can be ran by using the *rosrun* command or by setting up a launchfile with the *roslaunch* command. The syntax for the first case is the following: $rosrun < nodename >< param1 >< param2 > ....$ For the second case the syntax is simply *roslaunch ¡package name¿ ¡node launch file¿*. The launchfile is

an xml file that contains the name of the node and the parameters it requires. A description of the launchfile format is accessible at [ROS, c].

A.1.4 Rhino Description in ROS

Robots can have articulated joints and links between them. In ROS it is possible to define the structure of the robot, by specifying the links sizes and the types of joints between them. It is also a way for defining coordinate frames in the robot that will later on be useful.

A.1.5 Coordinate Frames and the *tf* Package

Sensor data, robot positions, or points in space in general can be expressed in different coordinate frames. The *tf* package lets the nodes seamlessly transform points or data expressed in one coordinate frame to another. For example, in the case of our Rhino platform; the pan-tilt unit node will publish periodically a *JointState* message containing the current state of the unit; so that the tf stack can then provide the transformation matrices/methods from the base of the robot to the camera coordinate system.

In our system we make use of three coordinate frames. The world or global coordinate frame, the one relative to the base of the robot, and that of the *Kinect* sensor. The *move base* node is responsible for publishing the *JointState* data so that tf can transform between relative to the base and global coordinates. The pan tilt unit node must publish its state so that *tf* can transform between coordinates relative to the camera and relative to the base.

A.2 Running the tracker

There are several configurations on which the tracker can be run. The first three parts of this section give a description of the configuration, state the hardware/software requirements and parameters that need to be set. The last one goes through all the parameters that can be included in the launchfile.

A.2.1 Free-Hand Camera Mode

In this configuration the tracker can be run without the need of a robot. The camera can be freely moved by the user. Internally, the 3D coordinates of the

target are permanently kept relative to the camera since there is no information available about the camera movement.

- Requirements:
  - the Kinect sensor
- Launchfile specific parameters:
  - freeHandCam set to *true*
  - simulatedTime set to *false*

A.2.2 Onboard Tracker

In this mode, the tracker runs on a mobile platform such as Rhino. The different coordinate frames are taken into account; thus, the target's 3D coordinates are global.

- Requirements:
  - Rhino mobile platform
  - the Kinect sensor
  - the configuration file with the characteristics of the robot for ROS
- Launchfile specific parameters:
  - freeHandCam set to *false*
  - simulatedTime set to *false*

The configuration file we used for Rhino has the following content:

A.2.3 Evaluation mode

This configuration was used to evaluate the performance of the tracker on the benchmark test set. It runs a node that reads the bag/video files and provides the frames as if the Kinect driver was running. In this way we are able to reproduce and evaluate experiments. There are no hardware requirements on this mode.

- Requirements:
  - bag/video files

- Launchfile specific parameters:
  - evaluationfalse set to *true*
  - simulatedTime set to *true*

A.2.4 Parameters

Table A.1: Tracker node parameters

Parameter	Type	Description
normalsAvailable	bool	Enables the surface normal features
visualsAvailable	bool	Enables the scalable gradient features on the grayscale layer
colourAvailable	bool	Enables the scalable gradient features on the colour layers
depthAvailable	bool	Enables the scalable gradient features on the depth layer
freeHandCam	bool	Set to true for the free-hand camera mode
particleState2D	bool	Set to true to enable the image state space of the particles
recordVideo	bool	Set to true to record a video with the result of the experiment
videoFile	string	Absolute path to the name of the video file that will be recorded if recordVideo is enabled
evaluation	bool	Set to true to run the evaluation mode
globalX	float	Initial x image relative coordinate of the target
globalY	float	Initial y image relative coordinate of the target
globalW	float	Initial width of the bounding box in image relative coordinates of the target
globalH	float	Initial height of the bounding box in image relative coordinates of the target
nodisplay	bool	If set to true disabled any visualization other than the console output
test	bool	If set to true a test callback function is used

Table A.1: Tracker node parameters

Parameter	Type	Description
mark	bool	Together with the test flag, runs a mode that lets the user mark the position of the target in the first frame
numParticles	int	Sets the number of particles of the particle filter
numWC	int	Sets the number of weak classifiers that form the strong classifier
numPosEx	int	Sets the number of positive examples that will be stored by the classifier
numNegEx	int	Sets the number of negative examples that will be stored by the classifier
learnThreshold	float	Sets the threshold above which the confidence on the target triggers the adaptation of the classifier. It is a value in the range $[0, 1]$
movingAvgMethod	bool	Enables the moving average method for the error calculations in the dynamic feature pool mechanism
onlyStaticPoolBhvr	bool	If set to true disables the feature pool adaptation mechanism
simulatedRealTime	bool	If set to true ignores any time interval calculation for the motion model of the particles
intervalRealTime	float	Specifies a time interval in miliseconds that is used for the motion model of the particles if the *simulatedRealTime* flag is set to true
resultsFileName	string	The relative path to the file that contains the estimated positions of the target frame by frame.

If the evaluation flag is set to true, then the bagger node needs to be run. The following table covers the parameters that can be set for this node:

Table A.2: Bagger node parameters

Parameter	Type	Description
bobot	bool	If enabled reads one video avi of the bobot test set. If disabled reads one bag file of the RGB-D benchmark
bobotFile	string	The absolute path to the bobot video file
bagFile	string	The absolute path to the bag file
waitTime	float	The number of seconds that the node will wait before sending the next frame

Appendix B

System Design

This appendix goes into the details of the implementation of the system. It is a continuation of the overview given in section 4.3.

B.1 The Tracker Node

The tracker node implements the tracking algorithm. The classes in this node have to deal with the pre-processing of the data as it is received from the openni node, training the object classifier, manage the particle filter and display the results. An overview of the class diagram is given in figure B.1.

B.1.1 Pre-processing stage: the Examples class

The *Example* class processes the incoming *ImageWithDepth* message data and stores it in a way that is usable by the different features. Each of the layers, grayscale, depth, colour and the point cloud is only processed and stored if the corresponding feature is enabled in the system. Every *Example* object is uniquely identified by the *exampleID* attribute. The uml diagram of this class is shown in B.2.

Example Constructor

The constructor of this class takes the following parameters:

- cv::Mat imageMat: the RGB image
- cv::Mat depthMat: the depth image

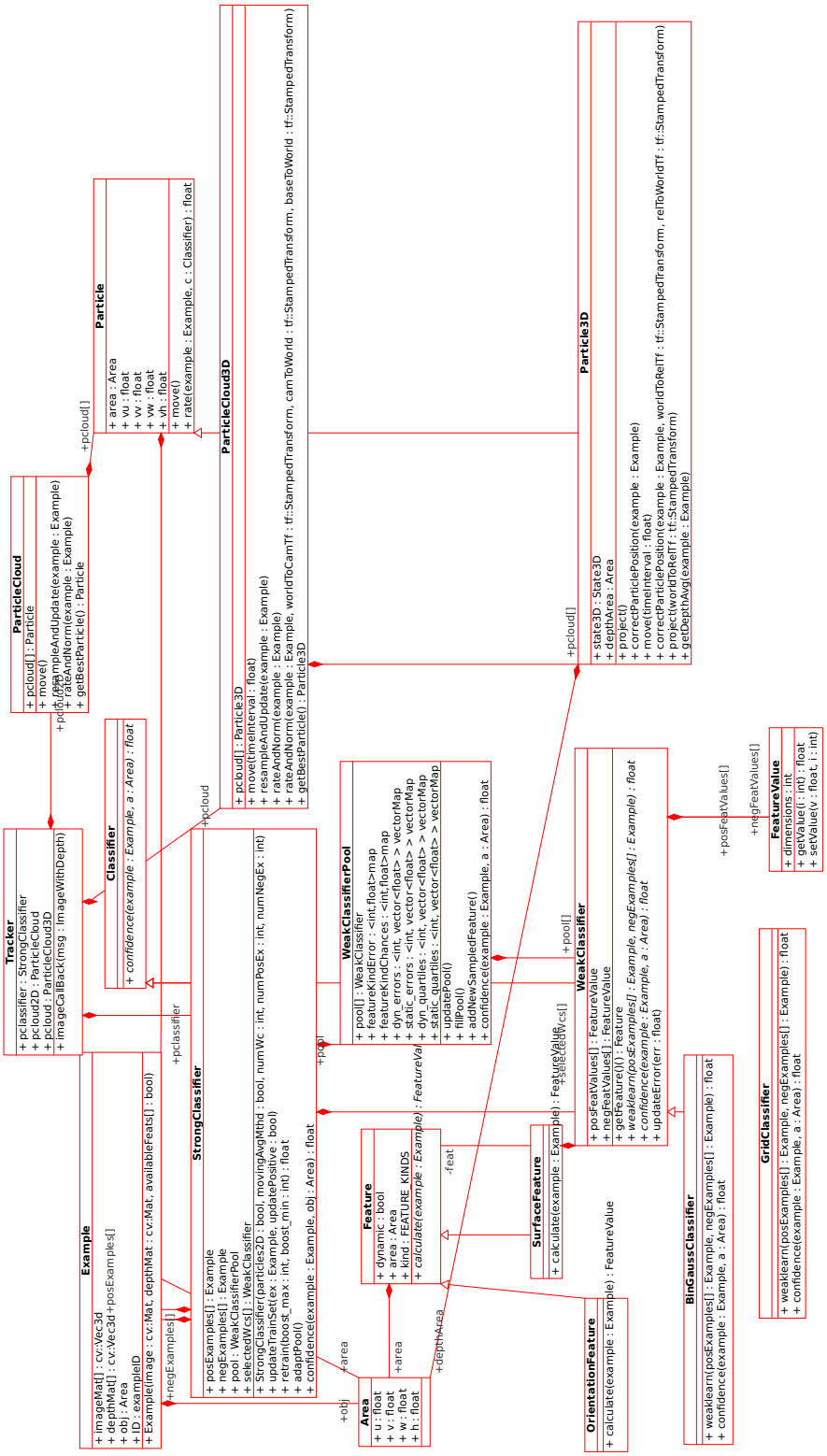

Figure B.1: UML class diagram of the tracker node

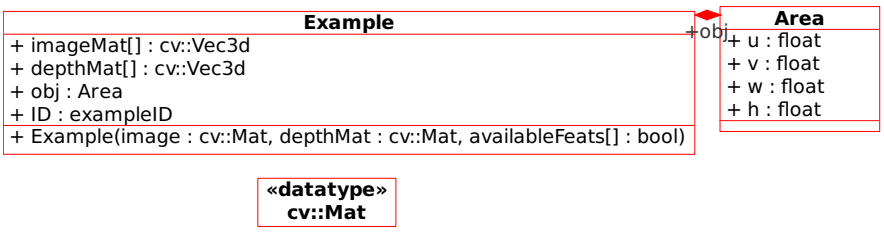

Figure B.2: UML class diagram of the example class

- bool grayEnabled: set to true if grayscale layer scalable gradient features are enabled
- bool colourEnabled: set to true if colour layer scalable gradient features are enabled
- bool depthEnabled: set to true if depth layer scalable gradient features are enabled
- bool normalsEnabled: set to true if surface normal features are enabled

There are two main loops in the creation of an Example object, lines 2 and 30 of algorithm 11. The first iterates over the RGB pixels, and transforms them to obtain the grayscales, as well as the α and β of the HSV colour space.

The second loop goes through the depth values, checking if they are valid. If so, they are stored in two layers in meters and millimetres. Then a 1.0 is stored in the layer of valid pixels. If the depth value is not valid, a 0.0 is stored in every layer.

The two kinds of information, visual and depth, are stored in separate multi layer matrices: imageMat and depthMat. The main reason behind this structure is to make use of vectorial instructions present in modern processors.

- Visual information
  - imageMat[0] : grayscale
  - imageMat[1] : α
  - imageMat[2] : β
- Depth information
  - depthMat[0] : number of valid pixels

Algorithm 11 Example::constructor

Require: • Image: mat
• Depth image: depth
• Available features boolean array availableFeats[] containing: grayEnabled,colourEnabled,depthEnabled,normalsEnabled

```
1: if grayEnabled OR colourEnabled then
2:    for (i = 0; i ≤ mat.cols; i++) do
3:       for (j = 0; j ≤ mat.rows; j++) do
4:          imageMat(i,j)[0] := 1/3 (mat(i,j)[0]+mat(i,j)[1]+mat(i,j)[2])
5:          imageMat(i,j)[1] := 0.5(2mat(i,j)[0]-mat(i,j)[1]-mat(i,j)[2])
6:          imageMat(i,j)[2] := √3/2 (mat(i,j)[1]-mat(i,j)[2])
7:       end for
8:    end for
9:    imageMat.integral()
10: end if
11: if cloudEnabled then
12:    for (i = 0; i ≤ depthMat.cols; i++) do
13:       for (j = 0; j ≤ depthMat.rows; j++) do
14:          depthValue := depth(i,j)
15:          if isValid(depthValue) then
16:             depthMat(i,j)[0] := 1.0
17:             depthMat(i,j)[1] := depthValue
18:             depthMat(i,j)[2] := depthValue * 1000
19:             cloud(i,j).pt.X := depthValue
20:             cloud(i,j).pt.Y := depthValue * constant * (j-depthMat.rows/2)
21:             cloud(i,j).pt.Z := depthValue * constant * (i-depthMat.cols/2)
22:          else
23:             depthMat(i,j)[0] := depthMat(i,j)[1] depthMat(i,j)[2] := 0.0
24:             cloud(i,j).pt.X := cloud(i,j).pt.Y := cloud(i,j).pt.Z := 0.0
25:          end if
26:       end for
27:    end for
28:    cloud.integral()
29: else
30:    for (i = 0; i ≤ depthMat.cols; i++) do
31:       for (j = 0; j ≤ depthMat.rows; j++) do
32:          if isValid(depth(i,j)) then
33:             depthMat(i,j)[0] := 1.0
34:             depthMat(i,j)[1] := depth(i,j)
35:             depthMat(i,j)[2] := depth(i,j) * 1000
36:          else
37:             depthMat(i,j)[0] := depthMat(i,j)[1] depthMat(i,j)[2] := 0.0
38:          end if
39:       end for
40:    end for
41: end if
42: depthMat.integral()
```

- depthMat[1] : depth in milimeters
- depthMat[2] : depth in meters

At the end of the loops the respective integral image is calculated. Each loop and its integral image call forms a section that is executed in parallel.

The Example Copy Constructor

This constructor is available for the creation of negative examples. It creates an Example object where all the cv::Mat attributes are kept as references. Therefore no time is spent in allocating memory or pre-processing the data. It takes two parameters:

- Example example: a reference to a positive example
- Area a: the area over which the negative example is defined

B.1.2 The Classifier Family of Classes

The classifier class is a base class, of which all the different classifiers inherit. It provides a confidence method that returns how likely it is that a certain Area region is the target. The StrongClassifier and the WeakClassifier inherit from this class.

B.1.3 The strong classifier

The *StrongClassifier* class must be able to learn the appearance of the target. It holds a reference to the pool of features, to the vectors of positive and negative examples, and keeps a vector of references to the selected features that form the ensemble classifier.

The constructor

The constructor parameters are:

1. *particle2D*: a flag that says whether the particle filter is using the image state space. This flag is later checked for the generation of negative examples.

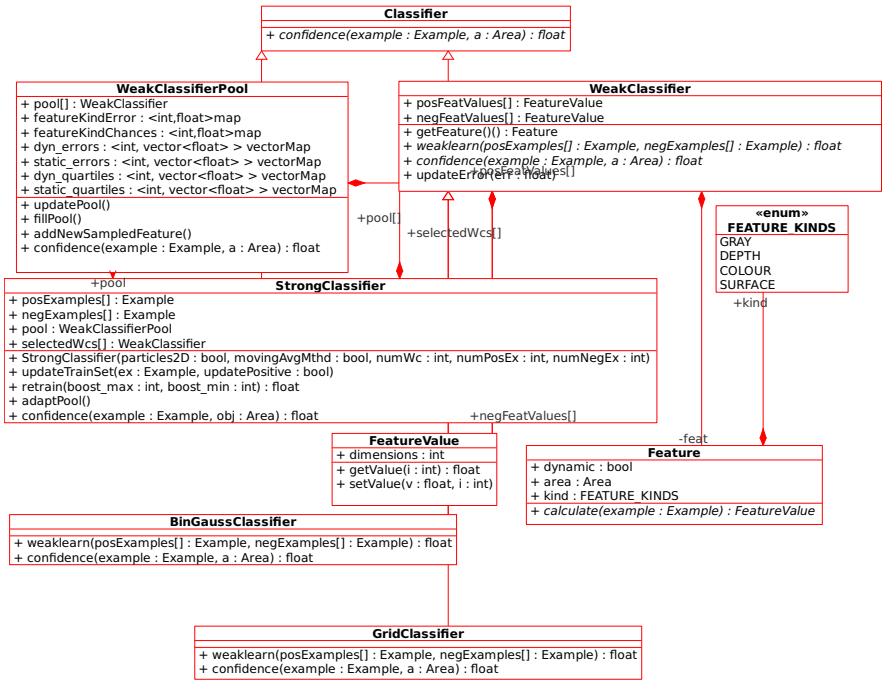

Figure B.3: UML class diagram of the classifier related classes

2. *movingAvgMthd*: a flag that says whether the error of the features in the dynamic pool class will be updated using a moving average technique.

3. *numWc*: the number of weak classifiers that will form the ensemble

4. numPosEx: the number of positive examples that will be stored and used for training

5. numNegEx: the number of negative examples that will be stored and used for training

The updateTrainSet Method

The updateTrainSet method takes the following parameters:

- example: a reference to a positive example
- *updatePositive*: a flag that says whether the positive examples should be updated

The example parameter will hold a reference to a positive example object. The strategies for updating positive and negative examples are described in algorithms

12 and 13. The main idea is that examples are stored if there is enough space in the corresponding vectors. Then the feature results are calculated for this new example. If there is not enough room for storing the new example, different policies apply in both cases.

Algorithm 12 StrongClassifier::updatePosExamples

Require: • Positive example: example

```
 1: if posExamples.size() < numPosEx then
 2:     posExamples.push(example)
 3: else
 4:     for (i = 0; i ≤ numPosEx; i++) do
 5:         res := confidence(posExamples[i])
 6:         if res > max then
 7:             best.clear()
 8:             best.push(i)
 9:             max := res
10:         else if res == max then
11:             best.push(i)
12:         end if
13:     end for
14:     if confidence(example) < max then
15:         randPos := random() * best.size()
16:         addResults(example)
17:         posExamples[best[randPos]] := example
18:     end if
19: end if
```

The generateNegExample Method

For the generation of new negative examples, it is checked whether there is a valid depth information at the randomly generated x and y positions. If so, its dimensions in the image are calculated according to the real dimensions of the target. This mechanism allows us to generate negative examples of the same projected dimensions the target would have at the chosen position in the image. If there is no depth available the dimensions of the rectangle are randomly generated. The whole algorithm can be checked at 14. The method receives two parameters:

- Example positiveExample: a reference to a positive example
- Example negativeExample: the returned generated negative example

Algorithm 13 StrongClassifier::updateNegExamples

Require: • Positive example: example
• Negative examples to be refreshed: negRefresh

```
 1: while negExamples.size() < numNegEx do
 2:     generateNegExample(example,negExample)
 3:     negExamples.push(negExample)
 4:     addResults(negExample)
 5: end while
 6: for (i = 0; i ≤ negRefresh; i++) do
 7:     a := (oldestNeg + i)  mod  negSamples.size()
 8:     generateNegExample(example,negExample)
 9:     negExamples[a] := negExample
10:     addResults(negExample)
11:     oldestNeg := (a + negRefresh)  mod  negSamples.size()
12: end for
```

- The 3D dimensions of the object W,H

The retrain Method

The retrain method updates the ensemble of classifiers. First, a new feature is chosen by calling the boost() method. Of the selected classifiers of the last iteration, a number of them will remain in the ensemble regardless of their current classification rate. A number of classifiers between a parametrisable minimum and maximum will be newly boosted. Algorithm 15 details the steps.

In line 1 the first call to boost is executed. In the next lines the weights of the training examples are updated and the pool updates the error statistics of the classifiers. In line 5 the main loop starts; up to $numWc - boostmax$ classifiers will remain in the ensemble; up to $boostmax$ classifiers will be newly boosted. Inside this loop, a previously selected classifier can be discarded if the error it makes on the training set is greater than the parameter eth. Finally in line 30, if parameter $boostmin$ is greater than 1, the last loop is executed until another $boostmin - 1$ new classifiers are boosted.

The boost Method

The boost method implements the Gentle AdaBoost algorithm defined in section 2.1. In our implementation, the algorithm is extended as detailed in section 3.5.2. It is detailed in algorithm **??**.

Algorithm 14 StrongClassifier::generateNegExamples

Require: • Positive Example: positiveExample
• Negative Example: negativeExample

```
1: for (i = 0; i ≤ 30; i++) do
2:    valid := false, trials := 0
3:    while !valid AND trials <10 do
4:       x := random(), y := random()
5:       if positiveExample.depthMat(y,x)[VALID]==1.0 then
6:          w := W/depth
7:          h := H/depth
8:       else
9:          w := random()
10:         h := random()
11:      end if
12:      Area temp(x-0.5*w,y-0.5*h,w,h)
13:      if sharedArea(temp,positiveExample.getArea()) < shThreshold then
14:         valid := true
15:      end if
16:   end while
17:   if valid then
18:      conf := confidence(positiveExample,temp)
19:      if conf > best then
20:         best = conf
21:         negativeExample(positiveExample,temp)
22:      end if
23:   end if
24: end for
```

Algorithm 15 StrongClassifier::retrain

Require: • int: boostmin,boostmax
- The number of selectable weak classifiers: numWc
- The array of selected classifiers: selected
- Threshold float: eth

```
1: temp[0] = boost(true)
2: updateWeights(temp[0])
3: boostmin–
4: pool.updateErrors()
5: for (i = 0; i < numWc; i++) do
6:     epsilon := 9999
7:     for (j = 0; j ≤ last; i++) do
8:         if sectionsToBoost[selected[j]] > 0 then
9:             eps := selected.weaklearn(posExamples,negExamples)
10:            if eps < epsilon then
11:                epsilon := eps, best := j
12:            end if
13:        end if
14:    end for
15:    if epsilon < eth OR boostmax == 0 then
16:        temp[i] := selected[best]
17:        selected[best] := selected[last]
18:        selected[last] := NULL
19:        last–
20:    else
21:        temp[i] = boost(false)
22:        boostmax–
23:        if boostmin>0 then
24:            boostmin–
25:        end if
26:    end if
27:    updateWeights(temp[i])
28:    sectionsToBoost(temp[i])–
29: end for
30: for (i = numWc − boostmin; i < numWc; i++) do
31:    temp[i] = boost(true)
32:    updateWeights(temp[i])
33: end for
34: swap(selected,temp)
```

Algorithm 16 StrongClassifier::boost

Require: • Boolean flag updateErrors

```
 1: if updateErrors then
 2:     for (i = 0; i ≤ pool.size(); i++) do
 3:         if sectionsToBoost(pool[i])>0 then
 4:             eps := pool[i].weaklearn(posExamples,negExamples)
 5:             pool[i].updateError(eps)
 6:             if eps<epsilon then
 7:                 epsilon := eps
 8:                 best = pool[i]
 9:             end if
10:         end if
11:     end for
12: else
13:     for (i = 0; i ≤ pool.size(); i++) do
14:         if sectionsToBoost(pool[i])>0 then
15:             eps := pool[i].weaklearn(posExamples,negExamples)
16:             if eps<epsilon then
17:                 epsilon := eps
18:                 best = pool[i]
19:             end if
20:         end if
21:     end for
22: end if
23: return best
```

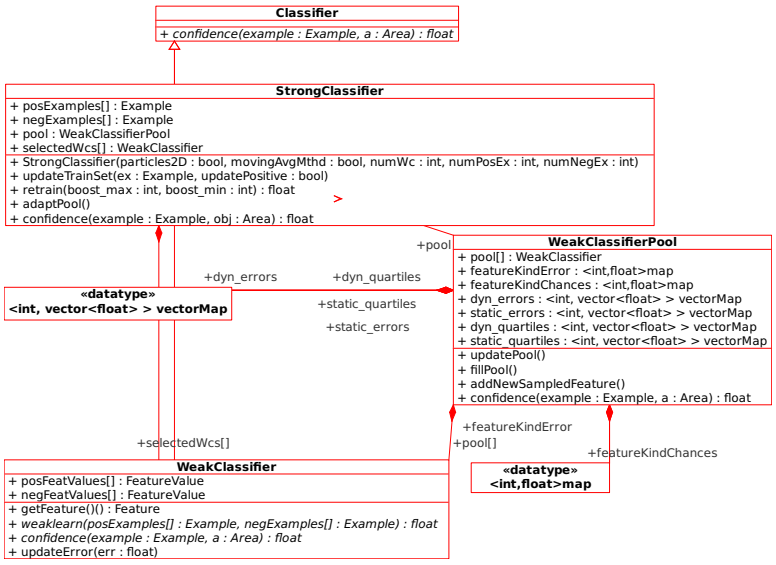

Figure B.4: UML class diagram of feature pool

In the current implementation, the loop that iterates over the classifiers is run in parallel. Before the method was able to finish if a classifier was found with an error of 0.0, now the loop has to go through the entire set of classifiers.

B.1.4 The WeakClassifierPool Class

The dynamic pool management is implemented in the *WeakClassifierPool* class. The UML diagram of this class is shown in figure B.4.

Feature Pool Unitialisation: the *fillWeakClassifierPool* Method

The feature pool is initialised in two steps. First, the static features are evenly distributed in positions and sizes over a rectangle with a given ratio. Then, a parametrisable number of dynamic features are added with random positions and sizes. The enabled feature kinds are boolean parameters of this method.

Error Update

The error measurement and pool update processes are executed when the strong classifier is confident enough on the target to be re-trained. The boost method now incorporates a boolean parameter as shown in algorithm 16. When enabled, the training error of the classifier is updated with the value returned by

the *weaklearn* method. This value is managed by each classifier object; the *updateError* contains updates the error value according to the moving average formula $avg_t \leftarrow factor * avg_{t-1} + (1 - factor) * err$. Tuning the $factor$ parameter to 0 is easy to turn it into a simple update function.

After the training step of the strong classifier has finished, the so computed training errors for this round are pushed into a vector. There is one error vector per feature kind and type. They are accessible via map data structures: 1 map per type of the form $< featType, vector < float >>$, so that the error vectors can be indexed by the feature kind identifier. The classifiers that made the lowest and highest errors in each category are also stored.

Random Seed Values

The dynamic feature pool mechanism can be disabled (see section A.2.4). If this is the case, the *fillWeakClassifierPool* method keeps generating the dynamic features but adds them to the pool as static features. No pool update mechanism is executed, except for the error updates and statistics generation.

The initial set of dynamic features is generated using a random number generator that is initialised using the a fixed seed. Therefore this set of features is initially always the same. Features added from there on are created with another random number generator, this one is initialised with the current time; so that different experiments produce different features.

B.1.5 The Feature Classes

The Feature class is the base class for all the feature kinds. The colour,depth and grayscale scalable gradient features are implemented inside the *OrientationFeature* class. The surface normals features are implemented in the *SurfaceFeature* class. The attributes inside the Feature class are:

- bool dynamic: it tells whether the feature belongs to the dynamic or the static pool
- Area area: the area that defines region with respect to the positive or negative example
- Kind kind: the kind of feature: {GRAY,DEPTH,COLOUR,SURFACE }

Figure B.5 shows the uml diagram of this class.

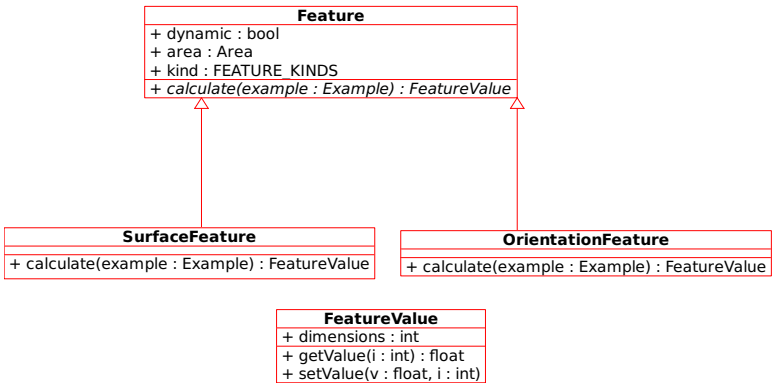

Figure B.5: UML class diagram of *Feature* class

The *calculate* method

The abstract method *calculate* is implemented by the specific feature classes. It takes an *Example* object and returns a FeatureValue object with the result of the feature on that particular example.

B.1.6 The WeakClassifier Class

The WeakClassifier class is the base class of which every specific weak classifier inherits. Among its attributes we find:

- Feature feat: a reference to a feature
- FeatureValue posFeatValues[]: a vector that stores the feature results for each positive example
- FeatureValue negFeatValues[]: a vector that stores the feature results for each negative example

The positive and negative feature values are indexed by the position of the example in the posExamples or negExamples vectors of the StrongClassifier class. This is the reason why the WeakClassifier needs to know the number of positive and negative examples present in the strong classifier.

A previous solution involved a map data structure, where the feature values were indexed by the *exampleID* of the example object. This was a clean solution but made the learning of weak classifiers much slower: the access time on a map structure is logarithmic ([cplusplus.com, a]) on the number of elements, whereas on a vector the access time is constant ([cplusplus.com, b]).

The weaklearn Method

The abstract weaklearn method that must be implemented by every specific classifier. This is convenient since the strong classifier can have a pool of heterogeneous features, that can be transparently handled.

When calling the weaklearn method, the weak classifier will iterate over the positive and negative examples, storing the feature values in the posFeatValues[] and negFeatValues[] arrays. The actual learning depends on the implementation of this class.

B.1.7 Particle Related Classes

In this and the following sections we cover the classes that implement the particle filter, namely *ParticleCloud3D* and *ParticleCloud*; and the particles themselves: *Particle* and *Particle3D*. They are presented in the uml diagram of figure B.8.

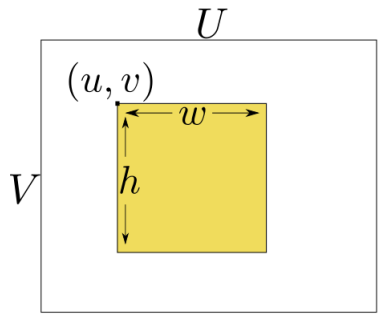

Figure B.6: The particle representation

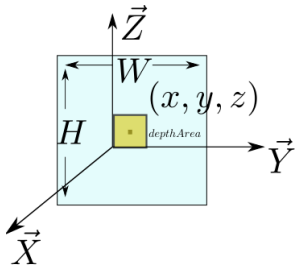

Figure B.7: The Particle3D representation

B.1.8 The *Particle* Class

The *Particle* class contains an *Area* object, with the position and dimensions of the bounding box in image coordinates. The position (u, v) is that of the upper left corner of the bounding box. The velocities at which the pixel positions and dimensions change are also stored. When the move method is called, the positions and dimensions are updated with the velocities at that moment; the speeds are also updated by adding white noise.

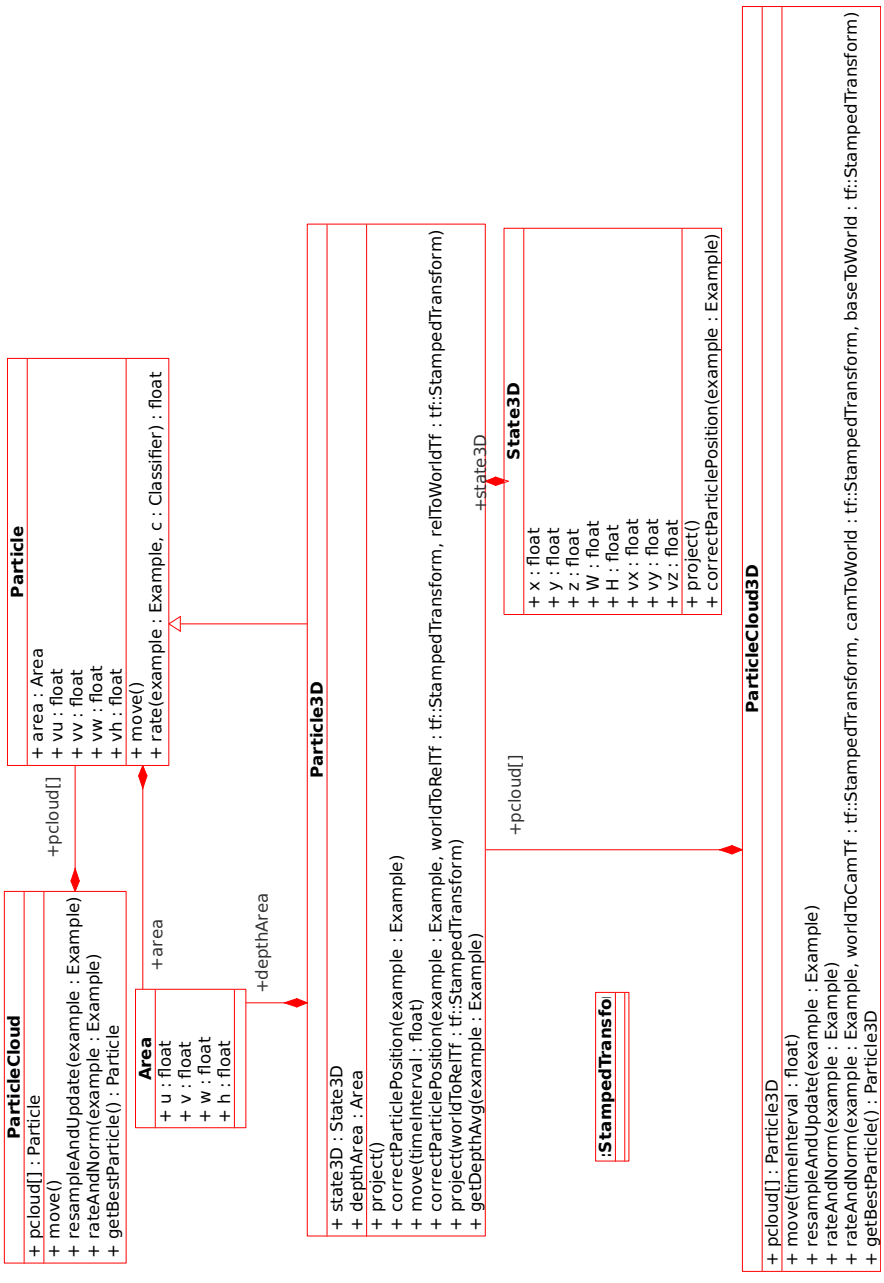

Figure B.8: UML class diagram of the particles family

B.1.9 The *Particle3D* Class

The *Particle3D* class inherits from the *Particle* class. It stores the 3D world position information, the velocities and the dimensions of the bounding box in 3D. Unlike the *Particle* class, the coordinates are those of the center of the bounding box and not the upper left corner. Figure B.1.7 shows a representation of this difference. The time interval with respect to the last iteration of the filter is internally stored.

The *move* Method

The move method applies the 3D motion model. It takes as parameter the *timeInteval*. The motion model is updated with this value and it is then stored to be used by the *correctParticlePosition* method.

The *project* Method

The *project* method projects the particles to the image plane, so that the observation model can be applied. It is overloaded in two methods:

- One for the free-hand camera mode (see section A.2.1). It takes no parameters.
- The other version is used for the onboard mode of the tracker (section A.2.2). It takes the as parameter the coordinate transformation matrix *worldToRelTf*; it gives the transformation from the world coordinate frame to that relative to the camera.

In both cases, the 3D coordinates of the particle relative to the camera are necessary. The inverse process to that described in section 3.1.3 is applied; it is described in algorithm 17. Note that the stored position of the projected rectangle is the upper left corner and not the center as in the 3D state of the particle.

The *correctParticlePosition* Method

The method *correctParticlePosition* corrects the position and speeds of the particles according to the current's frame point cloud. There are two overloaded versions of this method:

- One for the free-hand camera mode (see section A.2.1). It takes an *Example* object as parameter.

Algorithm 17 Particle3D::project

Require: • Particle 3D coordinates relative to the camera:x,y,z
- The 3D dimensions of the bounding box W and H
- The focal length of the RGB camera: focalLength
- The *width* and *height* of the RGB image

1: center_x := $\frac{width}{2}$
2: center_y := $\frac{height}{2}$
3: u := (center_x - ($\frac{y}{x} * focalLength$))/width
4: u2 := (center_x - ($\frac{y+W}{x} * focalLength$))/width
5: w := fabs(u2-u)
6: v := (center_y - ($\frac{z}{x} * focalLength$))/height
7: v2 := (center_y - ($\frac{z+H}{x} * focalLength$))/height
8: h := fabs(v2-v)
9: area.set(u-w/2,v-h/2,w,h)

- The second version is used for the onboard mode of the tracker (section A.2.2). It takes the following parameters:
  - an *Example* object
  - the coordinate transformation matrix *worldToRelTf*; it gives the transformation from the world coordinate frame to that relative to the camera
  - the coordinate transformation matrix *relToWorldTf*; it gives the transformation from relative to the camera to world coordinates

In both cases, the 3D coordinates of the particle relative to the camera are necessary. The procedure starts by projecting the particle to the image plane. With this projection, an average depth value is computed over an inner rectangle, defined as in figure B.1.7 by the same ratio as the particle's projection but smaller in size. This depth value is used to update the particle's 3D coordinates by the mechanism defined in section **??**. The speeds are also corrected taking in account the distance travelled by the particle with respect to the position before the adjustment, and the time interval stored in the *move* method.

B.1.10 The *ParticleCloud3D* Class

The *ParticleCloud3D* class implements the particle filter. Among its attributes, it contains a vector of *Particle3D* objects.

The *ParticleCloud3D* constructor

The *ParticleCloud3D* constructor takes the following parameters:

- *Area area*: the rectangle in the image of the first estimate
- *State3D state*: the world coordinates of the first particle
- *StrongClassifier pclassifier*: a reference to the strong classifier
- *int numP*: the number of particles

The *move* Method

It iterates over the particles and calling for each its *move* method. It takes *timeInterval* as a parameter; this value is the time elapsed between frames and is used to update the motion of the particles.

The *rateAndNorm* Method

The *rateAndNorm* method first corrects the particles' positions by calling the *correctParticlePosition* method. Then the *project* method is called and the particles that were inside the field of view of the camera will have an *Area* object with the position of the particle in the image. Those particles out of the field of view are marked as invalid ones and are not taken in account anymore. Those that survived can be rated with the observation model. Finally the weights are normalised.

The resampleAndUpdate Method

The *resampleAndUpdate* method first checks whether it is possible to update the classifier of the object. Two conditions need be satisfied:

- the weighted overlap is above the overlap threshold
- the confidence of the best particle is above the confidence threshold

If the two conditions hold, a new positive example, given by the projected area of the most confident particle, is added to the *posExamples* vector, the strong classifier is retrained, and the dynamic pool update mechanism is called.

Finally the re-sampling stage is executed, letting particles with lower confidence die out and replicating those with higher confidence. At the end of this stage we end up again having *numP* valid particles.

Algorithm 18 ParticleCloud3D::rateAndNorm

Require: • Example *example*

```
 1: for (i = 0; i ≤ n; i++) do
 2:     particles[i].correctParticlePosition(example)
 3:     if particles[i].isValid() then
 4:         particles[i].project(example)
 5:         if particles[i].isValid() then
 6:             rate := particles[i].rate(example)
 7:         end if
 8:     end if
 9: end for
10: sum := 0
11: for (i = 0; i ≤ particles.size(); i++) do
12:     sum := sum + particles[i].accentuate()
13: end for
14: estimate := particles.getBestParticle().getArea()
15: supp := getSupport(estimate)
16: conf := pclassifier.confidence(example,estimate)
```

B.1.11 The *Tracker* Class

It is the main class of the tracker node. It subscribes to the *ImageWithDepth* topic that contains the image data. Every incoming RGB-D frame is treated in a callback function. The steps executed in this callback function are those shown in algorithm 20. The function first checks if the received frame is the first; if so, it initialises the first estimate of the target, the object classifier and the particle cloud. The initial estimate is increased in size by 10% so that the classifier can use the surrounding background around the object; this happens on line 4. When the estimate of the target is stored, it is again shrunken to a 90.91% of the size.

From this level of abstraction, the algorithm goes through the motion model of the particles, line 12. Then rates the particles with the object classifier, line 13; and finally updates the observation model and does the resampling step in line 14. Finally, the 2D estimate of the current frame is resized and stored. The 3D estimate of the target is sent as a *ROS* topic.

B.2 Pan-tilt Control Node

The Ctrl node is in charge of controlling the pan tilt unit that holds the camera. A simple control scheme receives the target position in 3D and sends movement

Algorithm 19 ParticleCloud3D::resampleAndUpdate

Require: • Example *example*
- Confidence threshold $th1$
- Overlap threshold $th2$
- Last estimate support *supp*
- Last estimate confidence *conf*
- Maximum number of newly boosted classifiers $boost_{max}$
- Minimum number of newly boosted classifiers $boost_{min}$

if $conf > th1$ AND $supp > th2$ **then**
  pclassifier.updateTrainSet(*example*,*true*)
  pclassifier.retrain($boost_{min}$,$boost_{max}$)
  pclassifier.adaptPool()
end if
n := numParticles
p := 0
w := random() / n
for $(i = 0; i \leq n; i{+}{+})$ **do**
  while particles[$p \mod n$].getWeight() $< w$ **do**
    w := particles[$p \mod n$].getWeight()
    p++
  end while
  temp[i] := particles[$p \mod n$]
  $w := w + \frac{1}{numParticles}$
end for
swap(particles,temp)

Algorithm 20 Tracker::imageCallBack

Require: • Number of selectable weak classifiers wc
- Number of positive examples p
- Number of negative examples n
- Number of particles $nparticles$
- Maximum number of newly boosted classifiers $boost_{max}$
- Minimum number of newly boosted classifiers $boost_{min}$
- Incoming RGB-D frame msg
- Particle cloud $pcloud$
- Strong classifier $pclassifier$
- The initial estimate of the target $\{u_0, v_0, w_0, h_0\}$

```
1: Example example(msg)
2: if firstFrame then
3:     Area a({u_0, v_0, w_0, h_0})
4:     a.resize(1.1)
5:     State3D state3d := example.getCoordinates(a)
6:     pclassifier(wc,p,n,boost_max,boost_min)
7:     pcloud(a,state3d,pclassifier)
8: end if
9: t0 = msg.header.time
10: timeInterval = t0 - lastTime
11: lastTime = t0
12: pcloud.move(timeInterval)
13: pcloud.rateAndNorm(example)
14: pcloud.resampleAndUpdate(example)
15: Area a := pcloud.getBestParticle().getArea()
16: a.resize(-0.9091)
17: pushTargetEstimate(a,msg.header.seq)
18: sendTargetEstimate(pcloud.getBestParticle())
```

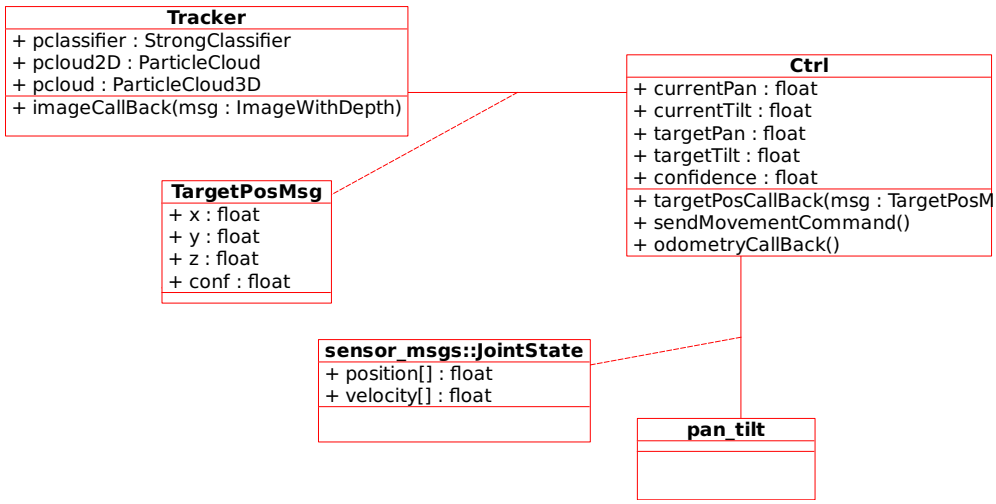

Figure B.9: UML class diagram of the control node

commands to the pan tilt driver so that the target is kept centred in the image. Diagram B.9 shows an UML representation of the classes involved.

B.2.1 The *ctrl* Class

The *ctrl* class is subscribed to the TargetPos topic. The target positions and confidence on the estimate are updated in the control node at a frequency of approximately $25Hz$. The behaviour is the following: the target positions are being constantly updated as they are sent by the *tracker* node (algorithm 21). The general control scheme is shown in algorithm 22.

Algorithm 21 Ctrl:targetPosCallBack

Require: • The *TargetPos* msg containing the 3D position of the target

1: targetPan := atan(msg.y/msg.x) + curPan
2: targetTilt := atan(msg.z/msg.x) + curTilt
3: targetPan := max(-0.94, min(0.94,targetPan))
4: targetTilt := max(-0.54, min(0.54,targetTilt))

B.3 Driver Node

The Driver node is in charge of sending movement commands to the base of the robot. It is subscribed to the TargetPos topic sent by the tracker, that contains the 3D estimated position of the target together with the confidence. Upon the

Algorithm 22 Ctrl:pantiltReadingsCallBack

Require: • sensor msgs::JointState msg
- Allowed difference between requested and current tilt angle $allowedTiltDiff \in [0, 1]$
- Allowed difference between requested and current pan angle $allowedPanDiff \in [0, 1]$
- The time the unit must stay still after reaching a requested position: $timeThreshold$

```
1: difPan := targetPan - curPan
2: difTilt := targetTilt - curTilt
3: if difPan < allowedPanDiff AND difTilt < allowedTiltDiff then
4:     if !firstTimeReached then
5:         timeReached := time.now()
6:         firstTimeReached := true
7:     end if
8:     targetReached := time.now() - timeReached > timeThreshold
9: else
10:    targetReached := false
11: end if
12: if !targetReached then
13:    sendPanTiltCommand()
14: end if
```

arrival of the first target position message, the distance to the target is stored as a safety distance. The state machine permanently checks if the target is at a distance greater than the one initially received, if so, if drives the base to the position where the target is, trying to keep the base aligned with the target. If not, it will only rotate the base to align with the target.

B.4 Plotter Node

The plotter node is responsible for storing all the information related to the features error, in a way that can later be used for displaying. It is subscribed to a topic where the Tracker node sends QuartilesMsg information. The message contains information about the quartile distribution of the error made by each feature kind (grayscale, depth, colour, normals) and type (static or dynamic).

B.4.1 Files Generated and Format

The plotter produces as output one file per type and kind, containing in each row the minimum, first quartile, median, third quartile and maximum error. Every row corresponds to a frame in the sequence. If the classifier was not updated,

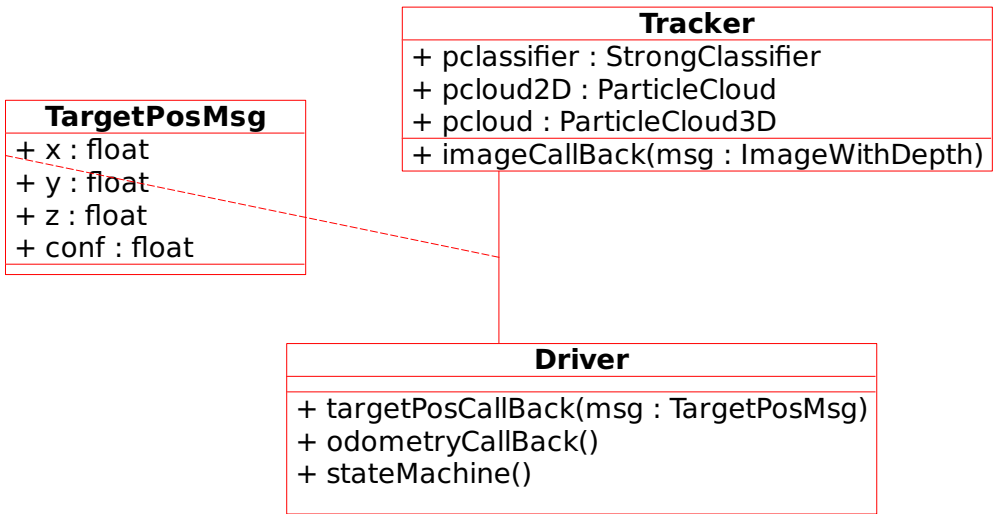

Figure B.10: UML class diagram of the driver class

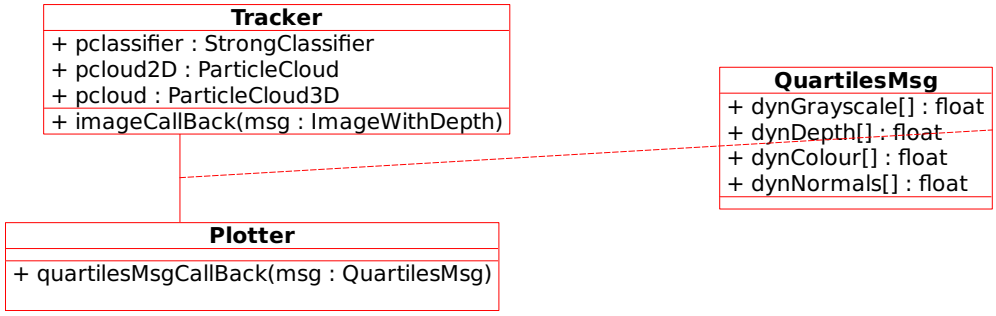

Figure B.11: UML class diagram of the plotter class

and therefore the errors were not computed, we will find zeros in the row.

The number of static and dynamic features of each type that belonged to the pool at every frame are also printed in a separate file. The same applies for the number of selected features of each kind and type.

Algorithm 23 Driver::statemachine

Require: • boolean $safetyDistance$
- boolean $rotatingAllowed$
- boolean $advancingAllowed$
- boolean $movingConfidence$

```
 1: while true do
 2:   if targetConfidence > movingConfidence then
 3:     r ← √(targetRelX² + targetRelY²)
 4:     if r > safetyDistance then
 5:       if advancingAllowed then
 6:         chase(targetRelAngle,r − safetyDistance)
 7:       else if rotatingAllowed then
 8:         rotate(targetRelAngle)
 9:       end if
10:     end if
11:   end if
12: end while
```

List of Figures

2.1 A graphical representation of the state estimation problem 7
2.2 Derivations of image regions in x and y dimensions are computed using differences of the means of scalable regions. Obtained from [Klein and Cremers, 2011] 10
2.3 Layout of the image plane, positive/negative examples, and features 11
2.4 The gradient features classifier 12
2.5 The steps of the algorithm for the computation of surface normals as detailed in [Dirk Holz and Behnke, 2011] 15
2.6 Each crossing represent a pixel in the image (u_i, v_i) which has some 3D coordinates $[x_p, y_p, z_p]$. the green vector are the tangents computed in each of the image axis directions. The surface normal (in blue) is the cross product of the two tangents $\vec{t_{ui}}$ and $\vec{t_{vj}}$. . . 17
2.7 The feature angles. The figure was taken from [Wahl et al., 2003]. The n_2' vector is the projection of n_2 on the vw plane 18

3.1 The point cloud (in green) and the particles (in red) 27
3.2 Side view of the projection of a point onto the image plane 27
3.3 Top view of the projection of a point onto the image plane 28
3.4 On the left a representation of the image of valid pixels; the red rectangle queries the number of valid pixels in its area. On the right the integral image of valid pixels 29
3.5 The vertical and horizontal surface normal feature regions. 31
3.6 The axis of the grid are the feature values α , $\arccos\beta$ and $\arccos\gamma$ 34

4.1 Picture by James Pfaff (litheon). CC-BY-2.0 (www.creativecommons.org/licenses/by/2.0) 40
4.2 The registered depth image 42
4.3 The RGB image 42

4.4 UML overview of the system 45
4.5 UML class diagram of the tracker node 45

5.1 Overlap measurement 46
5.2 Sequence M. RGB and depth images 47
5.3 Sequence N. RGB and depth images 47
5.4 Sequence O. RGB and depth images 48
5.5 Sequence P. RGB and depth images 48
5.6 Sequence Q. RGB and depth images 48
5.7 Sequence M: static gray features error quartiles distribution . . . 50
5.8 Sequence M: dynamic gray features error quartiles distribution . . 50
5.9 Sequence M: static colour features error quartiles distribution . . 51
5.10 Sequence M: dynamic colour features error quartiles distribution . 51
5.11 Sequence M: static and dynamic features in the pool 52
5.12 Sequence M: total selected features 52
5.13 Sequence M: selected dynamic features 53
5.14 Sequence M: selected static features 53
5.15 A: frame 190, the light intensity drops. B: frame 290, the tetra-pack is rotated around the view-point axis. Frame 375: the object is greatly rotated. Frame 550: lightning conditions are restored as the camera moves back 53
5.16 Sequence M: confidence 53
5.17 Sequence N: static gray features error quartiles distribution 54
5.18 Sequence N: dynamic gray features error quartiles distribution . . 55
5.19 Sequence N: static colour features error quartiles distribution . . . 55
5.20 Sequence N: dynamic colour features error quartiles distribution . 56
5.21 Sequence N: static and dynamic features in the pool 56
5.22 Sequence N: total selected features 56
5.23 Sequence N: selected dynamic features 57
5.24 Sequence N: selected static features 57
5.25 A: frame 45, sudden acceleration. B: frame 80, sudden stop. C: frame 125, sudden acceleration. D: frame 335, sudden stop 57
5.26 Confidence 58
5.27 Sequence O: static gray features error quartiles distribution 59
5.28 Sequence O: dynamic gray features error quartiles distribution . . 60
5.29 Sequence O: static depth features error quartiles distribution . . . 60

5.30 Sequence O: dynamic depth features error quartiles distribution . 60
5.31 Sequence O: static colour features error quartiles distribution . . . 61
5.32 Sequence O: dynamic colour features error quartiles distribution . 61
5.33 Sequence O: static and dynamic features in the pool 62
5.34 Sequence O: selected features 62
5.35 Sequence O: selected dynamic features 63
5.36 Sequence O: selected static features 63
5.37 A: the tank goes over the bridge. B: it turns around. C: goes before a battery. D: turns to frontal view 63
5.38 Confidence 64
5.39 Sequence P: static gray features error quartiles distribution 65
5.40 Sequence P: dynamic gray features error quartiles distribution . . 66
5.41 Sequence P: static depth features error quartiles distribution . . . 66
5.42 Sequence P: dynamic depth features error quartiles distribution . 67
5.43 Sequence P: static colour features error quartiles distribution . . . 67
5.44 Sequence P: dynamic colour features error quartiles distribution . 68
5.45 chase sequence: static and dynamic features in the pool 68
5.46 chase sequence: selected features 68
5.47 Box sequence: selected dynamic features 69
5.48 chase sequence: selected static features 69
5.49 A: first occlusion. B: second occlusion. C: the recording platform shakes and the image becomes blurry. D: a full occlusion 69
5.50 Chase sequence: confidence 69
5.51 Sequence Q: static and dynamic features in the pool 71
5.52 Sequence Q: selected features 71
5.53 Sequence Q: selected dynamic features 71
5.54 Sequence Q: selected static features 71
5.55 A: frame 75, the hand enters the target estimate. B: frame 225, the object is moved before the wall. C: frame 470, the lightning conditions change. D: frame 520, the illumination becomes very poor 72
5.56 Confidence 72
5.57 Sequence M: selected dynamic features 74
5.58 Sequence M: selected static features 74
5.59 Sequence M: static normal features error quartiles distribution . . 74
5.60 Sequence M: dynamic normal features error quartiles distribution 75

5.61 Sequence N: static and dynamic features in the pool 76
5.62 Sequence N: selected features 76
5.63 Sequence N: static normal features error quartiles distribution . . 76
5.64 Sequence N: dynamic normal features error quartiles distribution . 77
5.65 Sequence O: static and dynamic features in the pool 77
5.66 Sequence O: selected features 77
5.67 Sequence O: static normal features error quartiles distribution . . 77
5.68 Sequence O: dynamic normal features error quartiles distribution . 78
5.69 Sequence P: static and dynamic features in the pool 79
5.70 Sequence P: selected features 79
5.71 Sequence P: static normal features error quartiles distribution . . 80
5.72 Sequence P: dynamic normal features error quartiles distribution . 80
5.73 Sequence Q: static and dynamic features in the pool 81
5.74 Sequence Q: total selected features 81
5.75 Sequence Q: static normal features error quartiles distribution . . 81
5.76 Sequence Q: dynamic normal features error quartiles distribution . 81
5.77 Sequence M: selected features 84
5.78 Sequence N: selected features 85
5.79 Sequence O: total selected features 86
5.80 Sequence P: selected features 87
5.81 Sequence Q: total selected features 87
5.82 Sequence N: confidence 89
5.83 Sequence N: selected features 89
5.84 Sequence Q: confidence 90
5.85 Sequence Q: total selected features 90
5.86 Sequence Q: confidence 91
5.87 Sequence Q: total selected features 91
5.88 Sequence O: confidence plot in run 3 91
5.89 Sequence O: selected features in run 3 91
5.90 BoBoT Benchmark preview 93
5.91 On-board tracking frame 1 96
5.92 On-board tracking frame 2 96
5.93 On-board tracking frame 3 97
5.94 On-board tracking frame 4 97
5.95 On-board tracking frame 5 97
5.96 On-board tracking frame 6 97

B.1 UML class diagram of the tracker node 108
B.2 UML class diagram of the example class 109
B.3 UML class diagram of the classifier related classes 112
B.4 UML class diagram of feature pool 118
B.5 UML class diagram of *Feature* class 120
B.6 The particle representation 121
B.7 The Particle3D representation 121
B.8 UML class diagram of the particles family 122
B.9 UML class diagram of the control node 129
B.10 UML class diagram of the driver class 131
B.11 UML class diagram of the plotter class 131

List of Tables

5.1 Timing with base configuration 47
5.2 Timing with normal features 47
5.3 Sequence M: Overlap rate over runs and average 49
5.4 Sequence M: Hit rate over runs and average 49
5.5 Sequence M: averaged error statistics 50
5.6 Sequence N: Overlap rate over runs and average 54
5.7 Sequence N: Hit rate over runs and average 54
5.8 Sequence N: averaged error statistics 57
5.9 Sequence O: Overlap percentages over runs and average 59
5.10 Sequence O: Hit rates over runs and average 59
5.11 Sequence Q: averaged error statistics 62
5.12 Sequence P: Overlap rate over runs and average 65
5.13 Sequence P: Hit rate over runs and average 65
5.14 Sequence P: averaged error statistics 68
5.15 Sequence Q: Overlap rate over runs and average 70
5.16 Sequence Q: Hit rate over runs and average 70
5.17 Sequence Q: averaged error statistics 71
5.18 Sequence M: Overlap percentages over runs and average 73
5.19 Sequence M: Hit rates over runs and average 74
5.20 Sequence N: Overlap rate over runs and average 75
5.21 Sequence N: Hit rate over runs and average 76
5.22 Sequence O: Overlap percentages over runs and average 76
5.23 Sequence O: Hit rates over runs and average 76
5.24 Sequence P: Overlap percentages over runs and average 79
5.25 Sequence P: Hit rates over runs and average 79
5.26 Sequence Q: Overlap rate over runs and average 80
5.27 Sequence Q: Hit rate over runs and average 80

5.28 Sequence M: Overlap percentages over runs and average 83
5.29 Sequence M: Hit rates over runs and average 83
5.30 Sequence N: Overlap percentages over runs and average 84
5.31 Sequence N: Hit rates over runs and average 85
5.32 Sequence O: Overlap rate over runs and average 86
5.33 Sequence O: Hit rate over runs and average 86
5.34 Sequence P: Overlap rate over runs and average 86
5.35 Sequence P: Hit rate over runs and average 86
5.36 Sequence Q: Overlap rate over runs and average 86
5.37 Sequence Q: Hit rate over runs and average 86
5.38 Sequence N: Overlap rate over runs and average 88
5.39 Sequence N: Hit rate over runs and average 88
5.40 Sequence Q: Overlap percentages over runs and average 89
5.41 Sequence Q: Hit rates over runs and average 90
5.42 Sequence Q: Overlap rate over runs and average 91
5.43 Sequence Q: Hit rate over runs and average 91
5.44 Sequence O: Overlap rate over runs and average 91
5.45 Sequence O: Hit rate over runs and average 91
5.46 Sequence overlap averages 92
5.47 Sequence hit rate averages 92
5.48 Static Feature Pool Parameters 93
5.49 Overlap Rate. Static Feature Pool Test 94
5.50 Dynamic Feature Pool Parameters 94
5.51 Overlap Rate. Dynamic Feature Pool Test 94

A.1 Tracker node parameters 104
A.1 Tracker node parameters 105
A.2 Bagger node parameters 106

Bibliography

[Avidan, 2007] Avidan, S. (2007). Ensemble tracking. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29:261–271.

[cplusplus.com, a] cplusplus.com. *C++ reference: map data structure, [] operator*. `http://www.cplusplus.com/reference/stl/map/operator%5B%5D/`.

[cplusplus.com, b] cplusplus.com. *C++ reference: vector data structure, [] operator*. `http://www.cplusplus.com/reference/stl/vector/operator%5B%5D/`.

[Dirk Holz and Behnke, 2011] Dirk Holz, Stefan Holzer, R. B. R. and Behnke, S. (2011). Real-time plane segmentation using rgb-d cameras. In *Proceedings of the 15th RoboCup International Symposium 2011*.

[Dominik A. Klein and Cremers, 2010] Dominik A. Klein, Dirk Schulz, S. F. and Cremers, A. B. (2010). Adaptive real-time tracking for arbitrary objects. In *Proc. of the Int. Conf. on Robots and Systems (IROS)*.

[Freund and Schapire, 1995] Freund, Y. and Schapire, R. E. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. In *European Conference on Computational Learning Theory*, pages 23–37.

[Freund and Schapire, 1999] Freund, Y. and Schapire, R. E. (1999). A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780.

[Fukunaga and Hostetler, 1975] Fukunaga, K. and Hostetler, L. (1975). The estimation of the gradient of a density function, with applications in pattern recognition. *Information Theory, IEEE Transactions on*, 21(1):32 – 40.

[Giebel et al., 2004] Giebel, J., Gavrila, D., and Schnoerr, C. (2004). A bayesian framework for multi-cue 3d object tracking. In Pajdla, T. and Matas, J., editors, *Computer Vision - ECCV 2004*, volume 3024 of *Lecture Notes in Computer Science*, pages 241–252. Springer Berlin / Heidelberg.

[Gordon et al., 1993] Gordon, N., Salmond, D., and Smith, A. (1993). Novel approach to nonlinear/non-gaussian bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F*, 140(2):107 –113.

[Isard and Blake, 1998a] Isard, M. and Blake, A. (1998a). Condensation-conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29:5–28.

[Isard and Blake, 1998b] Isard, M. and Blake, A. (1998b). Icondensation: Unifying low-level and high-level tracking in a stochastic framework. In *Proceedings of the 5th European Conference on Computer Vision-Volume I - Volume I*, ECCV '98, pages 893–908, London, UK. Springer-Verlag.

[Kalman, 1960] Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45.

[Klein,] Klein, D. A. *BoBoT - Bonn Benchmark on Tracking*. `http://www.iai.uni-bonn.de/~kleind/tracking/index.htm`.

[Klein and Cremers, 2011] Klein, D. A. and Cremers, A. B. (2011). Boosting scalable gradient features for adaptive real-time tracking. In *Proc. of the Int. Conf. on Robotics and Automation (ICRA)*.

[Kurt Konolige,] Kurt Konolige, P. M. *Kinect calibration*. `http://www.ros.org/wiki/kinect_calibration/technical`.

[Microsoft,] Microsoft. *kinect*. `http://es.wikipedia.org/wiki/Kinect`.

[Openni,] Openni. *Openni: kinect imaging information*. `http://openkinect.org/wiki/Imaging_Information`.

[Radu Bogdan Rusu,] Radu Bogdan Rusu, Ken Conley, T. F. *Openni kinect driver for ROS*. `http://www.ros.org/wiki/openni_kinect`.

[ROS, a] ROS. *Kinect error calibration.* `http://www.ros.org/wiki/openni_kinect/kinect_accuracy`.

[ROS, b] ROS. *ROS:Introduction.* `http://www.ros.org/wiki/ROS/Introduction`.

[ROS, c] ROS. *ROS:launch files format.* `http://www.ros.org/wiki/roslaunch/XML`.

[ROS, d] ROS. *ROS:Messages.* `http://www.ros.org/wiki/ROS/Introduction`.

[Simone Frintrop and Schulz, 2010] Simone Frintrop, Achim Koenigs, F. H. and Schulz, D. (2010). A component-based approach to visual person tracking from a mobile platform. *International Journal of Social Robotics*, 2:53–62.

[Wahl et al., 2003] Wahl, E., Hillenbrand, U., and Hirzinger, G. (2003). Surflet-pair-relation histograms: A statistical 3d-shape representation for rapid classification.

[wikipedia.org,] wikipedia.org. *HSV and HSV: Hue and Chroma.* `http://en.wikipedia.org/wiki/HSL_and_HSV#Hue_and_chroma`.

[Wu and Nevatia, 2007] Wu, B. and Nevatia, R. (2007). Detection and tracking of multiple, partially occluded humans by bayesian combination of edgelet based part detectors. *Int. J. Comput. Vision*, 75:247–266.

[Yilmaz et al., 2006] Yilmaz, A., Javed, O., and Shah, M. (2006). Object tracking: A survey. *ACM Comput. Surv.*, 38(4):13.

Statement of Originality

I hereby declare that this thesis was composed independently and that no resources other than the declared ones have been used and that citations are indicated clearly.

Bonn,

Germán Martín García