Karlsruhe Institute of Technology

6-DoF Particle Filter-based Tracking of Arbitrarily Shaped Objects

Diploma Thesis
of

David Münch

At the Faculty of Informatics
Institute for Anthropomatics

First reviewer:	Prof. Dr.-Ing. Rüdiger Dillmann
Second reviewer:	Prof. Dr.-Ing. Jürgen Beyerer
Advisor:	Dr.-Ing. Pedram Azad

31. October 2009 – 30. April 2010

I declare that I have developed and written the enclosed Diploma Thesis completely by myself, and have not used sources or means without declaration in the text.
Karlsruhe, 30.04.2010

David Münch

Acknowledgments

Firstly, I want to thank my advisor, Pedram Azad, for a very comfortable working environment which allowed me to work independently but also for many explanations and the introduction into the research field of computer vision.

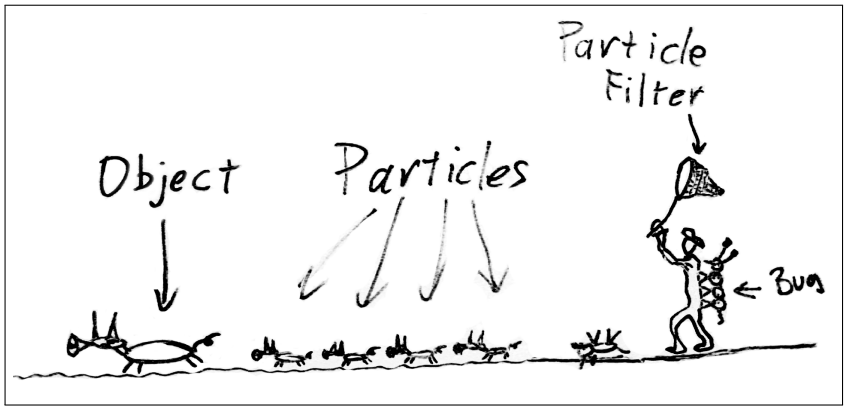
Thanks go to my colleagues in the laboratory for many fruitful discussions and inspirations; we also had fun playing flying disk. Thanks to the colleagues who always supported me with energy drinks. Thanks to the author of the template [1].

Thanks go to Matthias Huber and Susanne Münch, who proofread my thesis for several times. Thank you for your patience and your time.

I want to thank my family for their support and for their help.

Finally, I thank my wonderful wife for her continual support and love.

...on that note let the particles catch the object.



Abstract

6-DoF tracking of arbitrarily shaped objects is a challenging task concerning accurate pose estimation as well as real-time performance. Additionally, occlusions, rapid movements and a lack of usable textural information on the surface of the object can complicate the estimation of its correct pose or make it even impossible. In this work, a novel monocular model-based approach in 6-DoF tracking of arbitrarily shaped objects by using an annealed particle filter will be presented and evaluated – covering nontrivial single-colored as well as textured objects. For each particle representing a pose the corresponding object is rendered with OpenGL. Its edge image is evaluated pixel-by-pixel against the input edge image of the scene in reality. Subsequently, the actual mean pose of the object can be estimated. Thus, the results achieved are comparable and the pose estimations – both in simulation and in reality – are nearly faultless if the edge image of the object contains sufficient information. Real-time performance is not achieved on conventional hardware as will be discussed in this thesis. Speeding up the approach with NVIDIA's CUDA is limited due to hardware related issues on the GPU. In terms of universal applicability further speedup is required. The approach is a significant improvement compared to existing methods as it can deal with objects of arbitrary shape with few or even no textural information and as it can handle partial occlusions.

Zusammenfassung

In dieser Arbeit wurde ein Objektverfolgungsverfahren für einfarbige und texturierte beliebig geformte Objekte mit sechs Freiheitsgraden (6-DoF) entwickelt. Das zugrunde liegende Rahmenwerk ist ein mehrschichtiger (annealed) Partikelfilter.

Im Sonderforschungsbereich 588 Humanoide Roboter wird daran geforscht, einen humanoiden Roboter in einer Küche und später im gesamten Alltag die verschiedensten Aufgaben ausführen zu lassen. Möchte ein humanoider Roboter ein beliebig geformtes Objekt greifen, muss er dazu zu jedem Zeitpunkt die genaue Lage des sich bewegenden Objektes kennen. Ist die initiale Lage bestimmt und das dreidimensionale Modell des Objekts a priori bekannt, kann anschließend das in dieser Arbeit vorgestellte Objektverfolgungsverfahren eingesetzt werden, um zu jedem Zeitpunkt die genaue Lage des Objektes zu kennen.

In dieser Arbeit galt es, die Lage von einfarbigen oder schwach texturierten, beliebig geformten Objekten zuverlässig und exakt zu bestimmen; als Grundlage für komplexe Manipulationsaufgaben. Für diese Art von Objekten schlagen bisher bekannte Ansätze fehl. Aus zweidimensionalen Bildern dieser Objekte kann – bis auf die Farbe, die bei den hier untersuchten Objekten aber meist nur einfach ist – nicht mehr Information als die über deren Kanten gewonnen werden. Bisher bekannte modellbasierte und kantenbasierte Verfahren wie zum Beispiel [2, 3, 4, 5, 6, 7] arbeiten auf lokalen Kantenmerkmalen und minimieren Fehler zwischen geschätzter Lage des Modells und detektierten Kanten des Originalbildes. Partikelfilterbasierte Verfahren wie [8, 9, 10] arbeiten mit globalen Kantenvergleichen bei relativ einfach geformten Modellen. Das Problem bei beliebig geformten Objekten ist, dass nicht jede Kante des Modells auch eine reale Kante des Objekts repräsentiert, sondern oft nur eine Kante der durch Dreiecke erzeugten Objektoberfläche ist.

Der hier entwickelte neuartige Ansatz basiert auf einem mehrschichtigen Partikelfilter, welcher aus mehreren Anwendungen eines normalen Partikelfilters mit dynamischer Anpassung verschiedener Parameter bei gleichem Eingabebild besteht. Die Einzelbilder einer Bildfolge werden nacheinander wie folgt bearbeitet: Zuerst werden in dem Bild die Kanten detektiert und anschließend einer Dilatation unterzogen. Der Partikelfilter verwendet dieses Bild als Eingabe und vergleicht es mit allen von ihm erzeugten Bildern, die durch das Rendern des Objektes in verschiedenen Lagen und anschliessende Kantendetektion generiert werden. Der Vergleich der Bilder erfolgt pixelweise durch ein binäres Und mit Schwellwert. Je besser die Lage des realen Objektes mit der geschätzten Lage des vom Partikelfilter generierten Objektes übereinstimmt, desto mehr Kantenpixel werden übereinstimmen und desto besser wird diese Lage bewertet. Die Ausgabe des Partikelfilters für ein Objekt ist somit die gemittelte geschätzte Lage in Abhängigkeit ihrer Bewertung.

Die Ergebnisse der experimentellen Evaluation zeigen, dass der Ansatz sehr robust ist und zwar mit Fehlern von weniger als $1\,mm$ in horizontaler und vertikaler Richtung, $5\,mm$ in der Tiefe und $0.4°$ um alle Achsen. Voraussetzung für die Anwendbarkeit ist, dass qualitativ gute Kantenbilder bzw. Teile der Textur sichtbar sind. Auf derzeit verfügbarer Rechner-Hardware ist der Ansatz allerdings trotz der Ausschöpfung aller verfügbaren Optimierungsmöglichkeiten – sowohl auf der CPU als auch auf der GPU – nicht echtzeitfähig. Dies liegt an Beschränkungen der Architektur durch die Grafikkarte. Es ist jedoch davon auszugehen, dass in naher Zukunft diese Beschränkungen aufgehoben sein werden.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

In the past few years research and development in the field of computer vision has increased rapidly due to faster computers and the demand for sophisticated imaging sensors in artificial systems and video surveillance. An example of such a complex artificial system is a humanoid robot.

A humanoid robot is mainly designed to work in a human household, especially in the kitchen. In the German Collaborative Research Center SFB 588[1], a kitchen environment serves as the goal scenario, in which the humanoid robot ARMAR-III (see [11]) has to perform various manipulation tasks. The work performed in this thesis targets application on the humanoid robot ARMAR-III. A humanoid robot can be an assistant for those who need help due to reduced mobility and for those enjoying the convenience of an independent assistant doing the daily housework such as setting and clearing the table, putting the food from the fridge and the larder onto the table and back, filling the dishwasher and cleaning the house.

A human household – even if tidied up – is an extremely complex environment. There are many obstacles such as displaced chairs, moving persons or a pair of shoes laying on the floor. These dynamic changes of the environment cannot be captured and considered in the robot's static environment model of the household. Apart from the environment the actions the robot is assumed to take are complex, too:

The entire process of putting a can of beer into the fridge, for example, is organized in different layers and modules of different complexity. First, the robot needs to understand the action of putting a can of beer into the fridge, followed by planning the sequence of tasks satisfying the action. One task is to open the fridge consisting of further subtasks like moving the arm of the robot from its current position to the handle of the fridge and grasping it. Another subtask is to grasp the can of beer from the person ordering it. The robot has to determine the actual pose of the can and if the person is moving, it has to track the can and to grasp it dynamically.

Tracking the pose of an object in this environment is not an easy task as there are problems such as illumination changes, occlusions of the object and clutter in the background. Thus,

[1] `http://www.sfb588.uni-karlsruhe.de/`

the approach developed, evaluated and discussed in this work deals with tracking arbitrarily shaped objects – both single-colored and with textural information.

Tracking objects in a model-based and edge-based manner is straight forward for simple object primitives such as cuboids, cones or spheres. There, every edge in the two-dimensional (2D) wireframe image of the model represents a real edge of the object. Consequently, the edges of the wireframe model match the extracted edges of a real world image of the corresponding object. Dealing with more complex objects their corresponding models consist of several edges. For them it cannot be decided efficiently if a certain edge belongs to the outline of the model or if it is part of the surface as it can be seen in Figure 1.1. As recent approaches fail for such complex objects a new approach is developed which can deal with these difficulties.

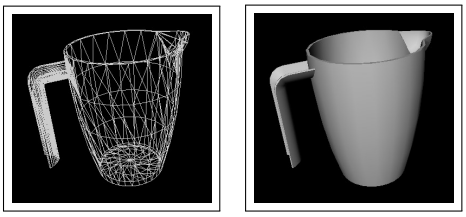


Figure 1.1: On the left side a wireframe model of a measuring cup can be seen. On the right side there is a surface model of the same object.

1.2 Aim and Contribution

The aim of this work is not to improve existing methods but to develop a novel approach for tracking arbitrarily shaped rigid objects as mentioned above and which is robust under translational and rotational movement as well as for partial occlusions of the object. Thus, a model-based approach for accurate six degrees of freedom (6-DoF) tracking is developed based on monocular vision. The objects to be tracked, which can be of arbitrary shape must be available as three-dimensional (3D) models – with or without texture. As the underlying algorithm is not restricted by any basic 3D primitives, it can track all kinds of objects. Thus, this approach is a major improvement to existing model-based 6-DoF tracking methods.

The contribution of this work is a tracking approach based on an annealed particle filtering framework and depending on edge images using existing 3D models of the objects to be tracked.

The annealed particle filtering framework is explained in the Sections 3.2 and 3.3. The first challenge to overcome in particle filtering is the initial condition: The initial pose of the object to be tracked has to be estimated, which can be accomplished by an already existing approach (see Section 2.6), and which is consequently not subject of this work. The annealed particle filter consists of several particles each describing the pose of the object, a six-dimensional (6D) configuration. To evaluate a particle, its configuration is used to render the object in its corresponding pose, in order to compare it to the current preprocessed input image from the camera and to rate it. The pose estimate is computed on the basis of the probability density function modeled by the current state of the particle filter. As the processing of the particles is independent and data parallel the approach is optimized with the support of NVIDIA's CUDA.

1.3 Structure and Overview

Chapter 2 provides an overview of the state of the art in model-based and edge-based object tracking. In Chapter 3 the basic components like the camera model, the CONDENSATION algorithm, the annealed particle filter and the Open Graphics Library are explained. The developed approach is described in detail in Chapter 4. Before evaluating the accuracy and performance in Chapter 6 the applied hard- and software is described in Chapter 5. Finally, Chapter 7 concludes with a summary of the results and an outlook is given.

Chapter 2

State of the Art

In this chapter classical approaches for model-based and edge-based object tracking will be presented, compared and evaluated. Due to the popularity of object tracking and its consequently broad field of scientific research – especially in the context of particle filtering – this chapter can only be seen as an excerpt without making claims of being complete. For this reason only the main fundamental and most important approaches are taken into account. In the survey "Monocular model-based 3D tracking of rigid objects" [12] Lepetit and Fua investigate the current state of the art in model-based rigid object tracking. The focus of this work is laid on tracking approaches based on edges as also described in [13].

2.1 Perceptual Organization

In [2], Lowe presents an edge-based approach for 3D object recognition. The input data he works with is a single gray-scale 2D image. The major improvement to former methods is the grouping of edge information called 2D perceptual organization. In detail the three basic considerations for grouping are:

- the proximity of endpoints,
- parallelism and
- collinearity of line segments.

The advantage of perceptual organization is its invariance over the most 3D viewpoint changes. However, two or more edges can be combined if they accomplish the grouping conditions in the 2D image even if being far away in 3D space. This step of perceptual organization reduces the search space as only groupings of more than three lines are considered and matched against a precomputed set of the viewpoints of the object. After matching, the resulting model is verified and the viewpoint is estimated. By applying the underlying approach to every frame a tracking mechanism is constructed. In [3], Lowe presents a possible improvement combining matching and measurement errors.

2.2 RAPID – A Video Rate Object Tracker

The popularity of RAPID [4] is based on its low computational complexity. It is one of the first real-time trackers on conventional hardware.

The object to be tracked needs to be known and it must have "high contrast edges". Control points are set on the object to allow fast and efficient tracking. In order to determine the motion between two frames for the control points $\boldsymbol{m}$ their projection $\boldsymbol{m'}$ in the new frame is searched. As visualized in Figure 2.1 only perpendicular distances l from $\boldsymbol{m}$ are calculated with an one-dimensional (1D) search in direction $\vec{n}$. This can be done only with the assumption of a low orientation change of the object. To speed up the 1D search, its direction $\vec{n}$ can be limited to horizontal, vertical and diagonal directions. With enough control points, the resulting distances $l = \vec{n}^\top(\boldsymbol{m} - \boldsymbol{m'})$ can be used to calculate the corresponding pose in minimizing the pose correction $\boldsymbol{q}$ in the equation $l' = l - \boldsymbol{qc}$ with a least-squares minimization approach.

A Kalman filter extension is applied to the RAPID tracker in [5]. Further robust improvements are performed in [6] describing an object at multiple levels and groupings of related geometric primitives.

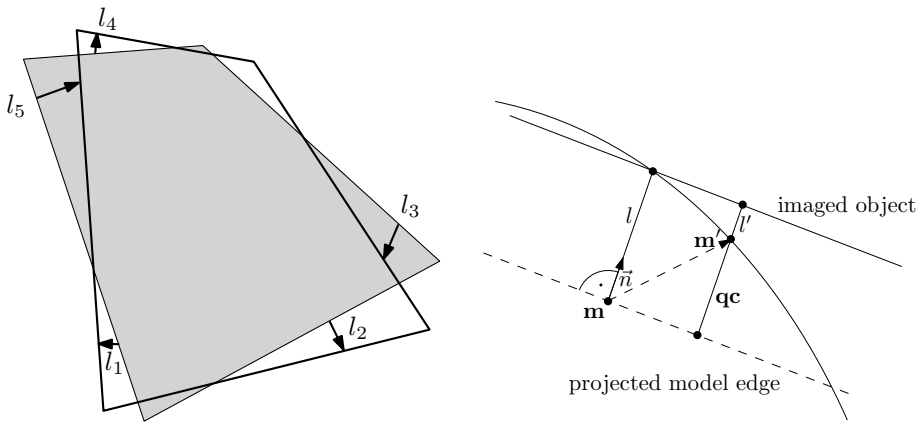


Figure 2.1: Visualization of the RAPID tracker. In the left picture the input image and the projected model edges can be seen. The distance between the control points $\boldsymbol{m}$ and their projection $\boldsymbol{m'}$ is visualized as l_i. In the right picture the interaction between the pose correction $\boldsymbol{q}$ and the distances l is visualized.

2.3 Integration of Model-Based and Model-Free Cues

In [14], Kyrki et al. present an approach combining model-based features and model-free cues based on an iterated extended Kalman filter. They extract the model-based features from a wireframe image of their object with hidden line removal. The model-free cues are extracted using a Harris corner detector [15]. The pose estimation performed by the iterated extended Kalman filter is compared within three different motion models:

- the object rotates around its own origin,

- the object rotates around the origin of the camera coordinate system or
- a constant velocity model is applied.

Kyrki et al. suggest to choose the motion model which is best fitting in a particular application. Additionally, they mention a particle filter but they also argue against it due to slow performance and worse results as only few samples and edge measurements can be performed in one time step.

2.4 2D-3D Tracking

Marchand et. al. present in [7] a two-step tracking approach. In the first step a 2D affine model of the object is fitted to the image. The 2D displacement – in contrast to other approaches even with large movements – can be computed iteratively. They use point correspondences between the edge image and the edges of the model with an 1D search combined with the Moving Edges algorithm [16] followed by the M-estimator [17]. In the second step the POSIT algorithm [18] and a non-linear minimization method similar to the one described in [3] are iteratively used to compute the pose on the basis of the point features already having been computed in the first step.

2.5 Particle Filtering Approaches

Due to the advantages to the Kalman filter [19] or other classical minimization approaches the particle filter can deal with non-Gaussian distributed noise, with non-linear system models and multiple hypotheses. A detailed explanation is offered in Section 3.2. The annealed particle filter is presented in Section 3.3. In the following, some particle filter-based approaches will be explained.

2.5.1 Real-Time Camera Tracking Using Known 3D Models

Pupilli and Calway develop in [8] a real-time camera tracking approach which is based on an annealed particle filter. In their approach they do not track the pose of the object. Instead they track the pose of the camera. In [9], they apply the inverse approach and track the pose of the object whereas the pose of the camera is fixed. This approach is also based on an annealed particle filter but additionally it requires the model of the object of interest.

As edge-based approaches are time consuming Pupilli's and Calway's approach is based on 3D edge junctions instead of edges only. They argue that even in dense clutter there are few edge junctions to allow fast tracking. A processing rate of 15 frames per second with 500 particles and images with a resolution of 320×240 is achieved. With this approach only a limited amount of objects can be tracked as sufficient edges are needed to form edge junctions. Additionally, no hidden line removal is implemented and consequently it is almost impossible to track complex objects.

2.5.2 Real-Time Visual Tracker by Stream Processing

Lozano et. al. present in [20] a real-time visual tracker for faces. Their approach is based on particle filtering speeded up with CUDA. The approach consists of three steps beginning with the initialization phase based on the Viola and Jones' detector [21, 22] and extracting the detected new faces as templates to extract relevant points decreasing the tracking complexity. The model of the face to be tracked is simulated as a planar surface projected on a generic 3D human face and is matched to the image with the Active Appearance Model algorithm.

In the second step the actual particle filtering is performed. The selection of particles and its diffusion is completed on the host whereas the weighting of each particle is calculated on the GPU as it can run independently and as it is the most time consuming part. This weighting is divided into two steps: In the first, the error contribution of each particle and feature point is computed, in the second a reduction of the results of the first step is performed.

In the last step the best particles are averaged and the resulting pose can be visualized as a 3D projection of the wireframe model into the image. This approach runs in real-time at 30 frames per second with four faces to be tracked in parallel.

2.5.3 Full-3D Edge Tracking on GPU

Klein and Murray implement in [10] a 6-DoF tracking method on the GPU. They use an annealed particle filter and their approach is based on a wireframe model with edge-based computations.

Each particle represents one pose estimate of the object to be tracked. The object is rendered with OpenGL on the GPU. To overcome the disadvantage of hidden lines they implement a hidden line removal with the depth buffer of OpenGL on subsampled images with a resolution of 160×120.

Due to slow readback from the GPU to the host they decided to implement the rating of each particle on the GPU with the aid of shaders. The processing pipeline begins with an undistortion of the subsampled images of the rendered object with a resolution of 320×240, followed by a thresholded sobel and thinning edge method. Finally, a distance transform for every pixel is applied.

The weighting of this edge image with the original image stored on the GPU is achieved in relation of matching and almost matching edge pixels in both images to all pixels of the edge image with the weighting function $\exp(const \cdot \frac{matchingPixels}{allPixels})$.

The annealed particle filter consists of two layers with a reduced amount of particles in the second step. With simple objects that can be rendered fast enough – compare Table 6.3 – good results at 30 frames per second and images with a resolution of 640×480 can be achieved although the core computations are based on subsampled images.

2.6 Accurate Shape-Based 6-DoF Pose Estimation of Single-Colored Objects

Azad et. al. present an appearance-based object recognition approach in [23]. An extension with pose correction is presented in [24]. It is a model-based extension working with views even allowing simulation environments for view acquisition or visualization.

The main idea is to divide the pose estimation in two steps:

- Orientation and
- position estimation.

The problem in this divided pose estimation process is that the object with the same rotation has different views for different positions, (see Figure 2.2 on the left and in the middle). The relation of both objects is visualized in the right image. A trained object view is available for position $\boldsymbol{t_l}$ and the object is located at position $\boldsymbol{t}$. With the learned view the object can be described with a corrective rotation R_c

$$\boldsymbol{t} = R_c \cdot \boldsymbol{t_l}.$$

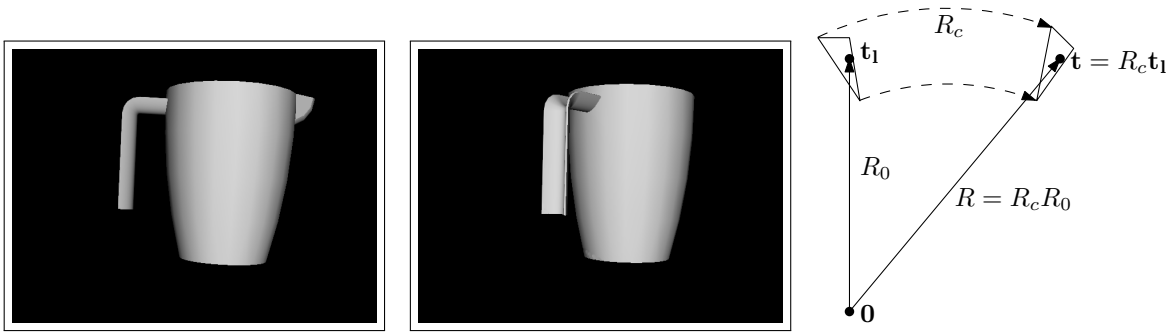


Figure 2.2: Different views in different positions. The object on the left has the same orientation as the object in the middle. Only the position of the object is different resulting in completely different views. The mathematical relation of this aspect is visualized on the right.

In the second part the position is estimated with stereo-triangulation of the objects describing their position with their triangulated centroid. The pose estimation $\boldsymbol{t'}$ is used to calculate the result $\boldsymbol{t} = f(R, \boldsymbol{t'})$. The function f matches the wireframe model into the image.

As the position and the orientation are affecting each other the whole process is iterative. Experimental results showed that at most two iterations of orientation and position correction are sufficient and that this approach can run in real-time. Unfortunately, the approach is error prone to partial occlusions as it requires global segmentation of the objects.

2.7 Comparison

In the sections above the state of the art in model-based and edge-based 6-DoF tracking has been presented. In Table 2.1 the approaches are directly compared and the type of

the required model as well as the extent to which edges are used is described. Either the approaches are edge-based on a wireframe model or they are based on the surface model of an object. The first six approaches use more local based edge computations whereas the remaining three approaches consider the edges as a whole. Only the last three approaches can deal with more complex single-colored objects. Table 2.1 depicts that textural information can be used to support the edge-based information.

It can be seen that none of the approaches can deal neither with arbitrarily shaped complex single-colored or few textured objects nor with the handling of occlusions, except the one developed in Chapter 4.

Approach	**Model-based**	**Edge-based**	**Model-free**	**Single-color.**	**Occlusions**	**Textured**	**Arb. shap.**
Perceptual Org. (2.1)	edge	grouping	-	+	+	+[3]	-
RAPID (2.2)	edge	distance	-	+	-	+[3]	-
Integration (2.3)	sur./ed.	distance	+	+	+	+	-
2D-3D (2.4)	edge	point cor.	-	+	-	-	-
RT known 3D (2.5.1)	edge	junctions	-	+	+	-	-
RT stream (2.5.2)	+[1]	o	points	-	-	+	-
Full-3D GPU (2.5.3)	edge	matching	-	+[2]	+	-	+
Pose estimation (2.6)	surface	matching	-	+	-	-	+
This Approach (4)	surface	matching	-	+	+	+[3]	+

Table 2.1: Comparison of the different approaches.

[1] deals with faces, uses (Active Appearance Model).
[2] wireframe model with hidden line removal.
[3] if enough edge information in texture available.

Chapter 3

Fundamentals

> "Success is neither magical nor mysterious. Success is the natural consequence of consistently applying the basic fundamentals." by Jim Rohn.

According to Rohn a profound understanding of the fundamentals is essential. The fundamentals of the developed approach in this work are covered in this chapter and will be described in detail. First, the extended camera model, second, the particle filter and its improvement the annealed particle filter and finally the Open Graphics Library (OpenGL) will be described.

3.1 Camera Model

In computer vision commonly the reality is sampled with a 2D sensor – a camera. Its output – an image – is used for further computations. As algorithms are working on these 2D images representing the real 3D world it is often necessary to know the transformation from the world coordinate system to the image coordinate system. The relation is visualized in Figure 3.1, for a detailed explanation see [25].

The principal axis runs perpendicularly through the image plane with its intersection (c_x, c_y) being called principal point. The camera coordinate system on the principal axis is pointing towards the scene. While x and y are parallel to the 2D image coordinate system on the upper left of the image plane the world coordinate system can be specified by the user arbitrarily.

The mapping function from the camera coordinate system to the image coordinate system is defined as

$$\begin{pmatrix} u \cdot z \\ v \cdot z \\ z \end{pmatrix} = K \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} \text{ with } K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}.$$

The camera constants f_x and f_y perform the conversion from $[mm]$ to $[pixel]$. The values of the matrix K are called intrinsic camera parameters.

The extrinsic camera parameters define the mapping from the world coordinate system to the camera coordinate system. They consist of a rotation R and a translation $\boldsymbol{t}$ resulting in

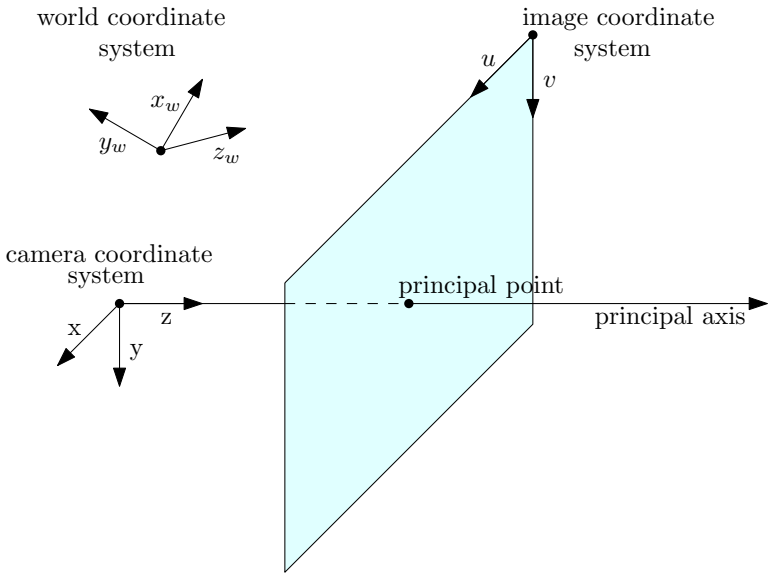


Figure 3.1: The extended camera model. The combination of the world, the camera and the image coordinate system as well as the principal axis and the principal point are visualized.

the transformation

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = R \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} + \boldsymbol{t}.$$

The intrinsic and extrinsic parameters can be combined in a projection matrix $P = K(R|\boldsymbol{t})$ to define the mapping from the world coordinate system to the image coordinate system using homogeneous coordinates:

$$\begin{pmatrix} u \cdot s \\ v \cdot s \\ s \end{pmatrix} = P \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

3.2 Particle Filter

In 1960, Kalman presented a recursive filter to estimate the current state of a system in [19], e.g. to eliminate noise in measurement data. The Kalman filter can fail if the system model is non-linear, if the noise is not Gaussian distributed or if the probability distribution has several maxima. The Kalman filter may fail by not reaching the global maximum and remaining in a local maximum, compare Figure 3.2. In this example it is possible to remain in the local maxima on the very right or left.

In order to overcome these limitations Gordon and Kitagawa presented a novel approach estimating the probability density function in a stochastic way in [26] resp. [27]. Later, in 1996, Blake et. al. applied the approaches mentioned above to visual tracking in [28]. Their framework is known as the Conditional Density Propagation for Visual Tracking (CONDENSATION).

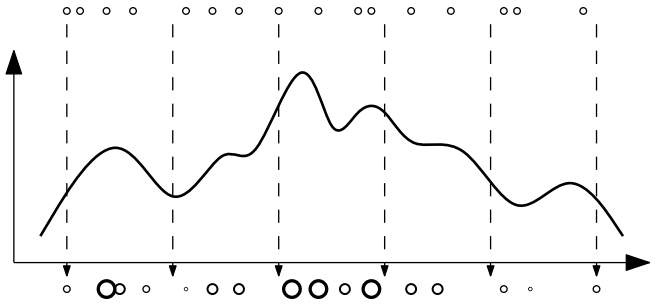


Figure 3.2: Risk of local maxima in particle filtering.

The advantages of CONDENSATION include dealing with non-linear systems, non-Gaussian distributions and multiple cues. Thus, several maxima can be estimated at once, not necessarily remaining in a local maximum as in Figure 3.2 in which every circle represents one particle.

Figure 3.3 depicts one time-step of the CONDENSATION algorithm. In the upper row the former particles are visualized with their probability being proportional to their diameter. In the first step, particles are picked proportionally to their probability and they undergo a deterministic "drift". Particles with high probability can be picked several times, those with low probability might not get picked at all. In the following step each particle undergoes some random "diffusion" movements. Finally, all the particles are "weighted" and their probability gets normalized.

In this work the CONDENSATION algorithm is implemented as presented in Section 4.1.2 in Algorithm 2. The outcome at time-step t of the CONDENSATION algorithm is estimated as

$$\mathbb{E}[f(x_t)] = \sum_{n=0}^{|particles|-1} \pi[n]_t \cdot f(s[n]_t).$$

In this case the weighted mean can be calculated with $f(x) \mapsto x$.

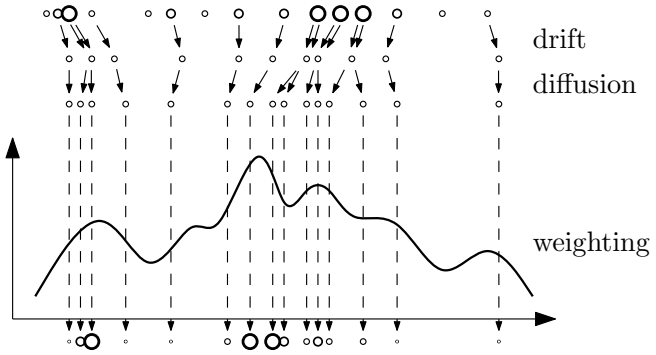


Figure 3.3: Visualization of the CONDENSATION algorithm.

3.3 Annealed Particle Filter

The CONDENSATION algorithm is applicable for various scenarios, however, if the search space is high-dimensional it cannot be applied in a performant way as the search space increases exponentially. An example of a high-dimensional search space is the human body consisting of at least 25-DoF when using a simplified model. In [29], Deutscher et. al. combine the CONDENSATION algorithm with simulated annealing resulting in the annealed particle filter.

Kirkpatrick et. al. describe simulated annealing in [30]: It is a probabilistic optimization method which can escape local extrema. Especially in large search spaces it produces good approximative results in a short time rather than the optimal solution.

Deutscher et. al. combine this idea with particle filtering. Their approach is outlined in Figure 3.4. Simplified, the annealed particle filter executes the CONDENSATION algorithm k – called layers – times with an adapting annealing rate affecting the variance of the pose estimations and an adapting number of particles.

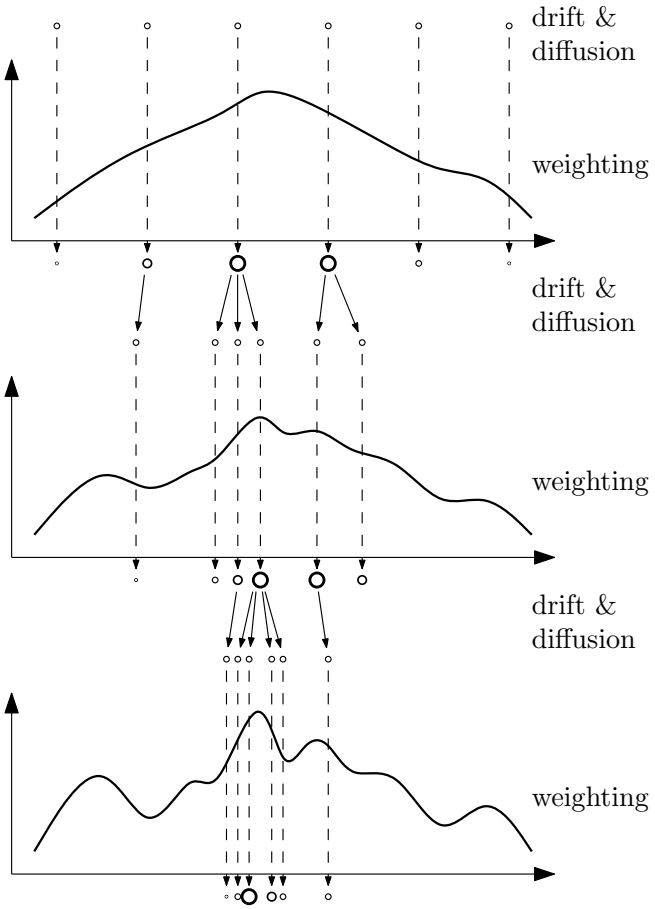


Figure 3.4: Annealed particle filtering. Due to smoothing at the beginning, the particles tend to reach the global maximum instead of a local maxima.

3.4 Open Graphics Library

The Open Graphics Library (OpenGL) is a platform-independent software interface for graphic cards usable for various 2D and 3D computer graphics. It has been developed by the Khronos Group[1] and has been realized as a state machine in which several parameters can be set and unset with `glEnable()` and `glDisable()`. All computations are based on several basic primitives like points, lines, triangles as well as more complex ones. In Figure 3.5 an overview of the pipeline of OpenGL is given as presented in [31].

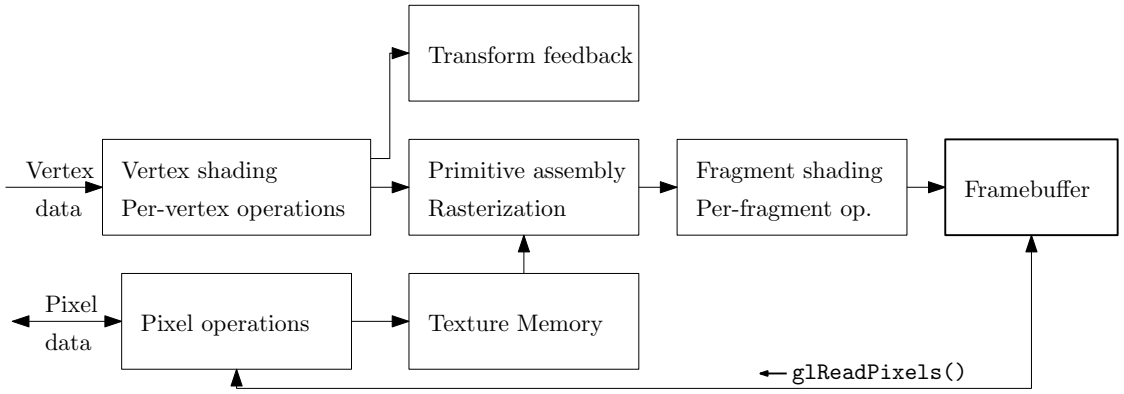


Figure 3.5: Simplified OpenGL pipeline.

In the first step – the vertex shading and per-vertex operation – the transformation from object coordinates to eye coordinates as well as the illumination model is applied. At the same time, some pixel operations are performed resulting in an image stored in the texture memory with the aim of being projected on the surface of the rendered object. In the following primitive assembly and rasterization step vertex and texture information are matched and a culling test is performed. Additionally, the transformation from eye coordinates to image coordinates is calculated. Subsequently, the geometric and pixel data are rasterized into fragments corresponding to pixels in the framebuffer. Before writing the result into the framebuffer some fragment operations are accomplished such as the conversion of the fragments in pixels, a scissor, alpha, stencil and depth test. The result in the framebuffer is ready to be displayed on the screen and can be read back to main memory with `glReadPixels()`.

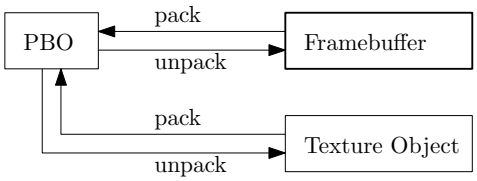


Figure 3.6: OpenGL pixel buffer objects (PBO) combine the use of pixel and vertex data allowing fast memory transfers with asynchronous direct memory access (DMA).

In order to use vertex and pixel data in combination in a performant manner there exist pixel buffer objects (PBO), see Figure 3.6. Their major advantage is the asynchronous fast transfer using direct memory access (DMA). Therefore, the texture image is directly transferred from

[1] `http://www.khronos.org/`

its source to the GPU without resting in the controlled memory space of the host. To speed up the use of PBOs multiple objects are simultaneously used in an asynchronous way.

Chapter 4

Developed Approach

In this work, any kind of a 3D model-based and edge-based 6-DoF tracking approach is provided. With this approach rigid objects that can be rendered by OpenGL can be tracked. The focus are kitchenlike objects which are in direct contact with the humanoid robot ARMAR-III (see [11]). As the developed approach can deal with single-colored objects it can also be applied to video surveillance especially in the dark where no color information is available.

In the following sections the different steps of the approach will be explained in detail. First, the general idea of this novel tracking approach is presented and second, the challenges arising are analyzed. Finally, the approach is optimized with general purpose computing on GPUs with CUDA.

4.1 Implementation of 6-DoF Tracking

The approach is explained in the following paragraphs. In Section 5.2.4 the UML class diagram is visualized and in Appendix C the C++ source code can be found.

4.1.1 Preprocessing the Image Sequence

In [24], Azad et. al. developed a fast and robust pose estimation approach for single-colored rigid objects. Error-prone to partial occlusions this approach is designed to improve it. For further details see Section 2.6.

In order to provide some input data an image sequence source is required. This can be a video camera, a precaptured image sequence or a simulated artificial scene. In the following, a sample input image of a blue single-colored measuring cup (see Figure 4.1) is used to explain and visualize the approach. The object was chosen precisely as it cannot be represented with basic primitives or straight lines and as it is not rotationally invariant and as it does not consist of many different edges which would allow applying an already existing approach, see Chapter 2.

As different kinds of objects have different needs a flexible algorithm is developed. The motivation to use a Sobel or Prewitt operator (see Appendix A) instead of the Canny edge detector is due to their better performance and an easier implementation in CUDA in a

further step (see Section 4.3). However, one kind of objects perform better with Sobel or Prewitt operator and other objects perform better with the Canny edge detector, compare Chapter 6. In the guiding example the Canny edge detector is applied to the input image, the outcome is visualized in Figure 4.2.

The rating of every single particle is calculated with the help of this image in a further step. Because of that and to make the tracking with an appropriate number of particles efficient, the edges in Figure 4.2 become stretched with a morphological dilation operation. With this step visualized in Figure 4.3, on the one hand, a small error in the correct pose estimation is allowed. However, on the other hand, the amount of particles can be reduced significantly (see Section 6.1.1). This step is necessary to make the algorithm more robust as the simulation edge image is never exactly the same as the input edge image.

Achieving good results with the Canny edge detector is even more ambitious than with the Sobel or Prewitt operator. This is due to two parameters to be set manually for the Canny edge detection algorithm which are a low and a high hysteresis threshold not being able to be set automatically.

This preprocessing of the input image is performed for every single frame, therefore, it is worth to do it precisely as every particle is rated in relation to this image (see Figure 4.3).

Figure 4.1: Captured image of a single-colored measuring cup.

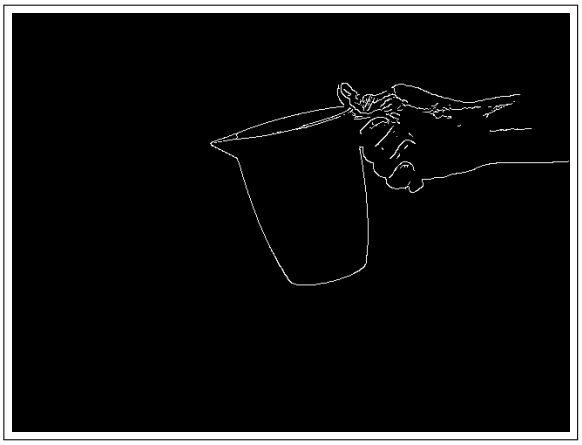


Figure 4.2: The Canny edge detector applied on the image of Figure 4.1.

4.1.2 Annealed Particle Filtering and Rating the Different Poses

The initial pose of the object to be tracked is known and the particle filter can be initialized with this configuration and equally distributed particles (see Algorithm 1). A particle $(s[i]_t, \pi[i]_t)$ consists of its 6D pose $s[i]_t$ and its probability $\pi[i]_t$ at the discrete time step t.

Algorithm 1: Initialize particles

Input: Initial pose $(x_0, y_0, z_0, \alpha_0, \beta_0, \gamma_0)$ of the object.
Output: Set of particles $\mathcal{M}_0 = \{(s, \pi)\}$.

1 **for** $i \longleftarrow 1$ **to** $|\mathcal{M}_0|$ **do**
2 $\quad s[i] \longleftarrow (x_0, y_0, z_0, \alpha_0, \beta_0, \gamma_0)$;
3 $\quad \pi[i] \longleftarrow \frac{1}{|\mathcal{M}_0|}$;
4 **return** $\{(s, \pi)\}$

As described in Section 3.2 and using the initialized particles from Algorithm 1 in the first step the particle filtering framework outlined in Algorithm 2 computes a new set of particles and their corresponding probability.

Algorithm 2: Particle filter framework

Input: Set $\mathcal{M}_{t-1} = \{(s_{t-1}, \pi_{t-1})\}$ of particles and $inputImage_t$.
Output: Updated set $\mathcal{M}_t = \{(s_t, \pi_t)\}$ of particles.

1 **for** $i \longleftarrow 1$ **to** $|\mathcal{M}_{t-1}|$ **do**
2 $\quad$ *Draw* $s[i]_{t-1} \propto \pi[i]_{t-1}$; `//drift`
3 $\quad s[i]_t \longleftarrow UpdatePose(s[i]_{t-1})$; `//diffuse`
4 $\quad \pi[i]_t \longleftarrow CalculateProbability(s[i]_t, inputImage_t)$; `//weighting`
5 $\pi_t \longleftarrow NormalizeProbability(\pi_t)$;
6 **return** $\{(s_t, \pi_t)\}$

Annealed Particle Filtering

Mentioned as the annealed particle filter in Section 3.3 Algorithm 2 can be performed for some times without changing the original input image. Every run of Algorithm 2 is called a layer. Thus, calling the particle filtering framework five times after updating the input image, corresponds to five layers in the annealed particle filter.

The aim of the particle filtering in each layer without updating the input image is to improve the correct pose. As a consequence of updating the pose of each particle in line 3 in Algorithm 2 the variance of "drift" should decrease in order to converge to a better pose estimation. Experimental results have shown that adapting the $variance_t$ in each layer t with

$$variance_t \longleftarrow variance_0 \cdot (1 - \pi[result]_t)$$

shows best results[1].

Algorithm 3 samples the current pose s_t of a particle in adding some random noise for every component of the pose vector. Figure 4.4 depicts the particle space of 20 particles with the edge image of every model. A broad variety of poses can be seen underlining the power of a particle filter.

[1]For example the cup in simulation mode with five layers and 500 particles: 0. layer: $\pi_0 = 0.12$, $variance_0 = 1.00$; 1. layer: $\pi_1 = 0.34$, $variance_1 = 0.88$; 2. layer: $\pi_2 = 0.57$, $variance_2 = 0.66$; 3. layer: $\pi_3 = 0.58$, $variance_3 = 0.43$; 4. layer: $\pi_4 = 0.61$, $variance_4 = 0.47$.

Algorithm 3: Update pose

Input: Pose s_{t-1} of a particle
Output: New pose s_t of a particle

1 $s_t \longleftarrow s_{t-1} \cdot (meanPose - lastPose) + variance_t \cdot random();$
2 **return** s_t

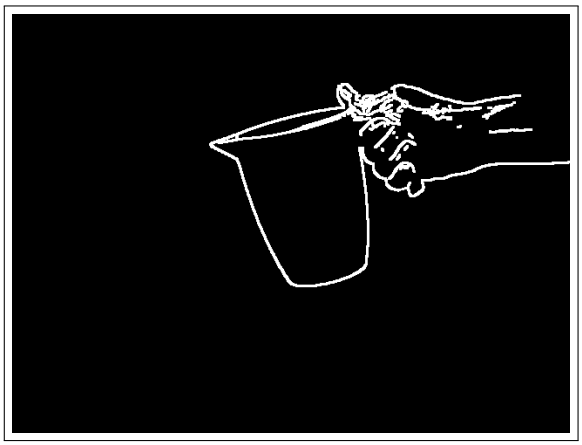


Figure 4.3: Dilation operation applied on the edge image of Figure 4.2.

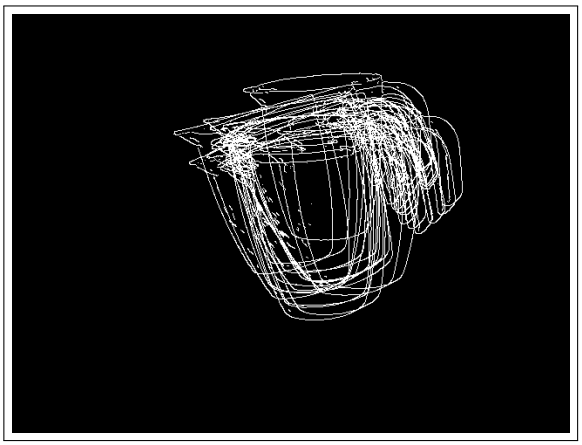


Figure 4.4: Edge image particle space of 20 particles.

Every particle $s[i]$ has a probability $\pi[i]$, its rating. The steps to be executed for each particle in order to get the rating are described in Algorithm 4. It starts with rendering the object in its current pose $s[i]_t$ and saving the rendered scene in a 2D image *image* (compare Figure 4.5).

According to the edge detection method chosen in the beginning either the Sobel or Prewitt operator or the Canny edge detector are used. In the case of Sobel operator the image is convolved as

$$imageSobel_x \longleftarrow Sobel_x * image$$

and

$$imageSobel_y \longleftarrow Sobel_y * image$$

resulting pixel-by-pixel in

$$edgeImage \longleftarrow \min\{\max\{|imageSobel_x|, |imageSobel_y|\}, 255\}.$$

Finally the *edgeImage* is binarized with a $threshold \in [0..255]$. For the case of the Canny edge detector see [32].

Algorithm 4: Calculate Probability

Input: $inputImage_t$ and pose $s[i]_t$
Output: Probability $\pi[i]_t$ of particle i

$image \longleftarrow renderObject(object, s[i]_t)$
switch *typeOfEdgeDetection* **do**
  case *Sobel*
    $edgeImage \longleftarrow CalculateGradientImageSobel(image)$;
    $edgeImage \longleftarrow ThresholdBinarize(image, threshold)$;
    break;
  case *Prewitt*
    $edgeImage \longleftarrow CalculateGradientImagePrewitt(image)$;
    $edgeImage \longleftarrow ThresholdBinarize(image, threshold)$;
    break;
  case *Canny*
    $edgeImage \longleftarrow Canny(image, lowThreshold, highThreshold)$;
    $edgeImage \longleftarrow Dilate3x3(image)$;
    break;
$pixelCount = \frac{PixelSum(edgeImage)}{255}$;
$andImage \longleftarrow AND(edgeImage, inputImage)$;
$pixelCountAND = \frac{PixelSum(andImage)}{255}$;
return $e^{-c\cdot(1-\frac{pixelCountAND}{pixelCount})}$

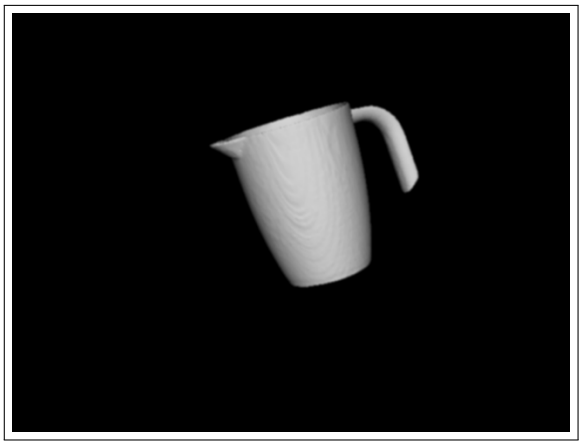


Figure 4.5: The best rated particle from Figure 4.4.

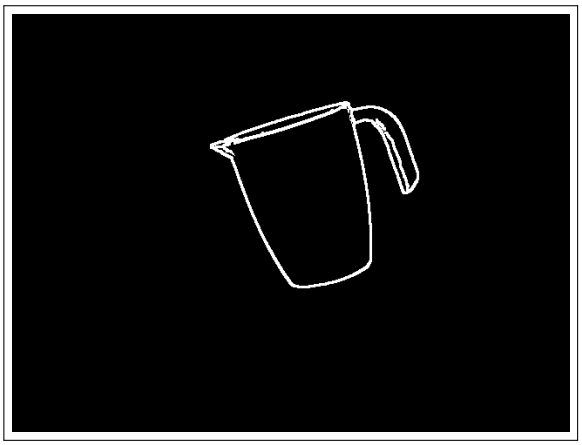


Figure 4.6: Application of Canny edge detector and dilation operation on Figure 4.5.

For the same reason as mentioned in the paragraph above the edge images of the rendered objects are postprocessed by a morphological dilation operation. Thus, there are more pixels which could match the input image accepting a small error in the correct pose estimation. An example of a postprocessed image is shown in Figure 4.6.

An important issue in this approach is the rating of every particle. A fast and simple but effective method is used: The pixel-by-pixel computed binary AND of the *inputImage* and *edgeImage*. Figure 4.7 depicts the binary AND of Figure 4.3 and 4.6.

The first approach developed to rate the current pose in Algorithm 4 line 18 was a simple relation of white pixels in the *edgeImage* and the resulting image *andImage*. An example of rating with an exponential function can be seen in Figure 4.9. There cannot be more white pixels in *andImage* than in *edgeImage*. An exponential function satisfies the needs perfectly in rating a broad variety of bad poses poor with less matches of pixels and in rating the few best poses well with the most matching pixels.

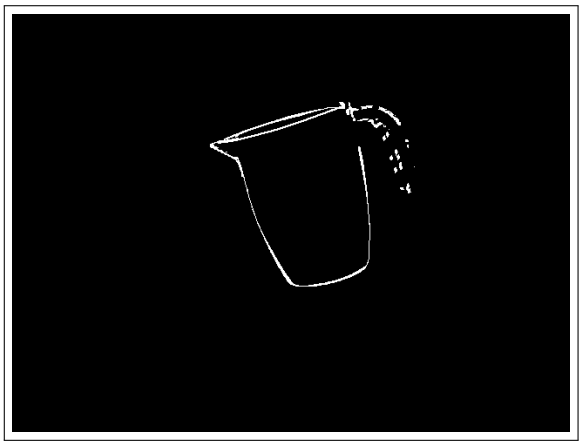


Figure 4.7: Pixel-by-pixel binary AND operation with figures 4.3 and 4.6.

Figure 4.8: Visualization of the estimated pose of the measuring cup in Figure 4.1.

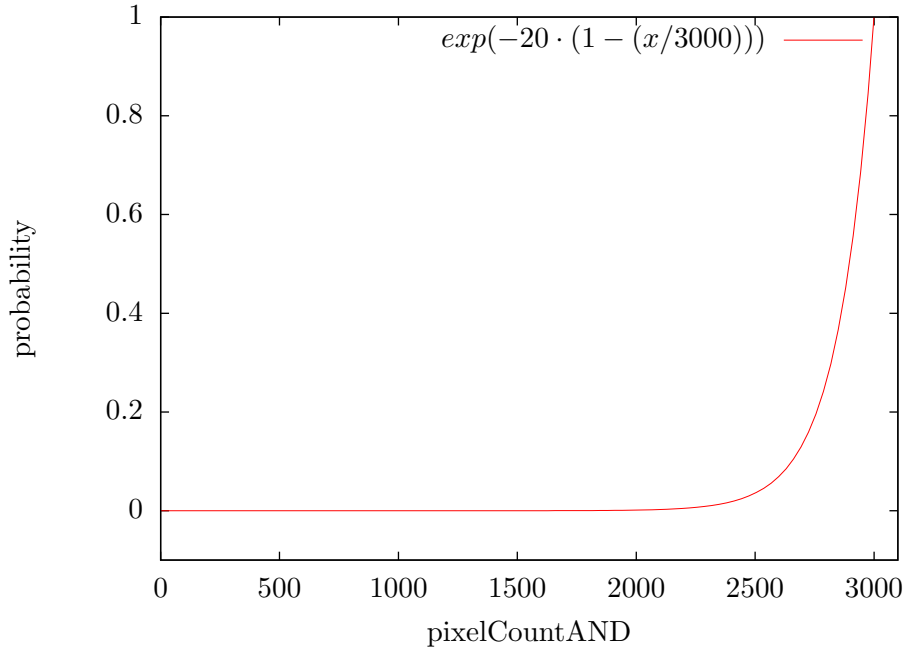


Figure 4.9: Rating function with 3000 projected white pixels (compare Figure 4.3).

4.1.3 Visualization of the Result

Having received the result of the annealed particle filter framework, either the mean pose or the best pose, it is desirable to visualize the resulting pose. On the one hand, the visualization of a tracked object is required to observe the correct pose, on the other hand, a visualized simulation environment is a benefit for the observer. In Figure 4.8 the original image is overlaid with the edge image of the resulting pose of the object to be tracked. All examples evaluated in Chapter 6 are visualized in this manner. Thus, it is even easier to verify the result of the algorithm visually.

4.2 Challenges

Throughout developing the approach above there were several challenges to overcome. In analyzing the critical steps and anticipating the evaluation results in Section 6.1.1 some steps can be improved.

4.2.1 Quality of the Input Image

An important preconditions of the approach is an edge image of the object to be tracked. On one side, the edge image can be adjusted by the low and high threshold of the Canny edge detector or by the threshold of binarization after the Sobel filtering. The focus of this work has not been laid on developing an automatic dynamic threshold estimation algorithm. This is why the thresholds are set manually after having inspected the intermediate results of the edge images. On the other side, the quality of the image should be good to allow a precise edge detection.

Many factors play a major role in this step: First, the illumination is important to avoid reflections on the objects that would result in edge clutter. Additionally, sparse illuminated areas in which no edge can be seen are a disadvantage, too. In general, the whole object should be illuminated sufficiently.

Second, the image should have a characteristic edge image, not only a contour and not too much edges. An example of an image with too less information may be a ball resulting in only one circle as edge image. In that case the rotational parameters of the pose cannot be determined – in this example they are even out of interest. Another example of an image with too much information may be an extremely crumpled piece of paper resulting in an edge image full of edges.

Finally, the background plays a major role in rating the particles. If there are lots of edges in the background not belonging to the object, but being matched to the model, the resulting pose can be wrong. Thus, the background should be removed resulting in an input image without clutter (see Figure 4.10).

To sum up, a good input edge image is required to allow a precise and efficient rating and consequently a good tracking.

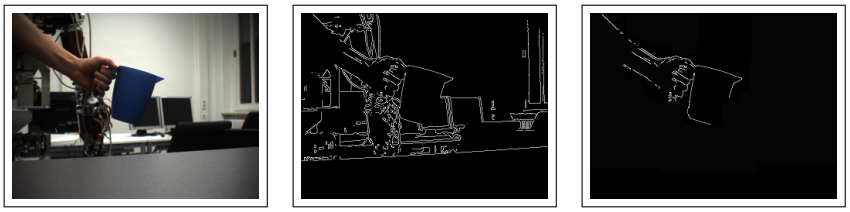


Figure 4.10: An edge image without background clutter.

4.2.2 Object Model

After having dealt with the input image the object to be tracked contributes with its quality to the outcome of tracking. A 3D model consisting of triangles does not have a continuous surface as the triangles form only an approximation of that. Thus, if the model has low quality, meaning only few triangles, the edge detector may detect edges on the surface where in reality no edges exist. Otherwise, if the object is scanned with e.g. a low resolution 3D scanner there will be surface irregularities resulting in a edge image with clutter (see for example Figure 6.18). Even if there are more than enough triangles (e.g. 20,000 or more) describing the measuring cup the problem of this object is a low scan quality.

A solution is postprocessing the rendered object with a Gaussian filter. In practice this means additional $0.1\,ms$ per rating of each particle. Receiving the rendered image from the GPU seems to be a simple task. Surprisingly, reading back in graystyle mode the image looks overexposed (see Figure 4.11). In contrast, reading the rendered image back as color image and convert it later into a grayscale image, the result is the expected grayscale image.

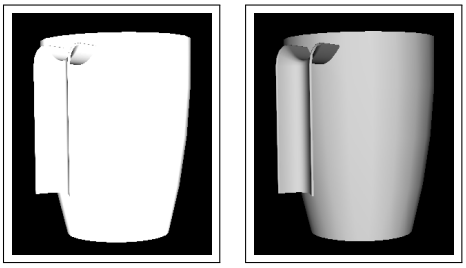


Figure 4.11: On the left side there is a grayscale image of a rendered measuring cup. On the right side there is the same object, apart from reading the image back as a color image and convert it later into grayscale.

4.2.3 Rating

A major point in this novel approach is the simple rating of the pose of the object. However, because of that simplicity the rating function has to be determined carefully. Simply rating the pixel matches like in Algorithm 4 line 18 often is sufficient, however, reaches its limit in certain cases. In Figure 4.12 such a challenging case is illustrated. This example will be discussed in detail.

As the rating function mentioned in line 18 in Algorithm 4 only calculates the relation between the white pixels of the input edge image and all the white pixels in the edge image of the

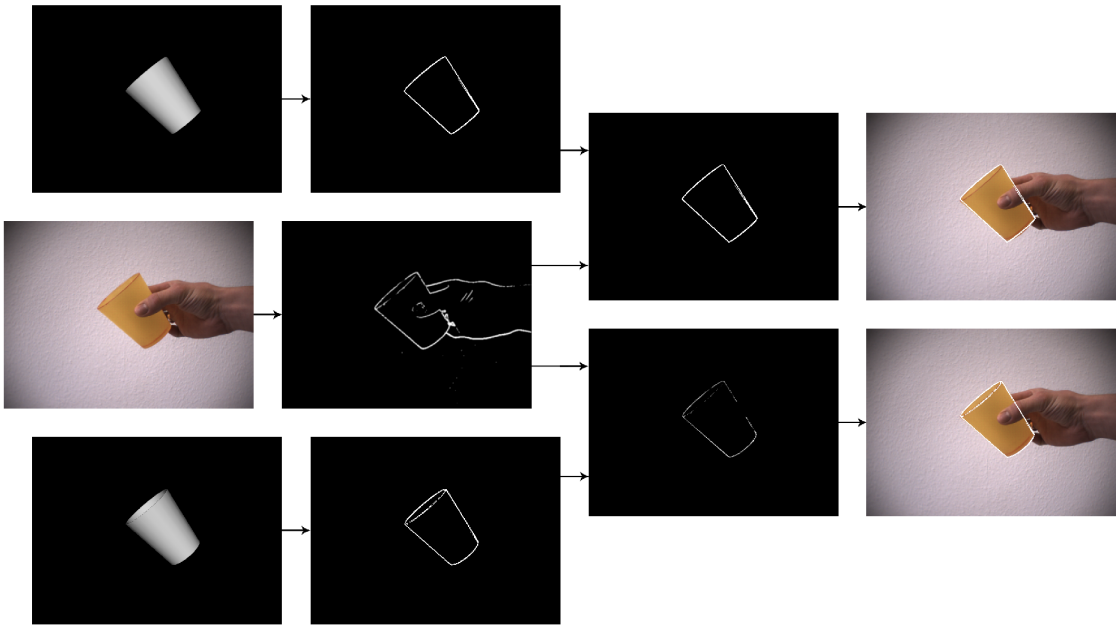


Figure 4.12: Challenge of the rating function. In this case not the good configuration (visualization at the bottom right) but the worse configuration (visualization at the top right) was the result of the algorithm. The problem occurs due to rating challenges.

object there can occur some problem poses. On the upper left in Figure 4.12 can be seen that the cup is slightly tilted to the back. Its edge image consists of one certain border of white pixels, exactly 715. In a better estimated pose of the object (e.g. in Figure 4.12 on the bottom left) the edge image seems to have more white pixels, summed up to the value of 739. After performing the binary AND with the input edge image in the middle there are 307 matching pixels in the upper and 310 matching pixels in the lower example. With the rating approach the lower pose has an amount of 42% matching pixels, the upper pose has 43% matching pixels. Thus, the pose on the bottom left – which is quite better – receives a worse rating.

An important step in decreasing the problem of rating is not to calculate the probability of each pose directly but to normalize the probability after processing all the particles and then applying problem-specific ratings. Algorithm 4 gets slightly modified. Instead of resulting the probability it returns nothing but stores the variables $pixelCount$ and $pixelCountAND$ for each particle. After having finished the loop in Algorithm 2 the probability of each particle is calculated by calling Algorithm 5. It normalizes the variables $pixelCount$ and $pixelCountAND$ followed by ratings $\pi_1, \pi_2, \cdots, \pi_k$ and their weighting with $c_1, c_2, \cdots, c_k$ in the final rating function. The rating functions for π_i can be chosen arbitrarily.

To come back to the example in Figure 4.12 with function $\pi_2 = (pixelCountAND)^2$ and $\pi_3 = (pixelCount)^2$ the result is for $c_1 = 0.4$, $c_2 = 2$ and $c_3 = 4$ a probability of 0.13 for the lower and 0.10 for the upper pose. The normalization and definition of arbitrarily rating functions in Algorithm 5 contribute to a flexible approach. In this case the upper pose achieves a better rating. Of course, the normalization does not solve the problem described, however, make the problem-specific definition of rating functions easier.

Another example is a disappearing pose estimation in z-direction resulting in only few white

pixels far away – up to one single white pixel – which completely match the input edge image resulting in a good rating but bad pose estimation. The solution is to penalize a low amount of white pixels and to reward many white pixels.

Algorithm 5: Normalization improvement

Input: $pixelCount$ and $pixelCountAND$ of every particle
Output: Probability π

1 **for** $i \longleftarrow 1$ **to** $|particles|$ **do**
2 $\quad pixelCountNormalized[i] \longleftarrow \frac{pixelCount[i]-pixelCountMin}{pixelCountMax-pixelCountMin}$;
3 $\quad pixelCountANDNormalized[i] \longleftarrow \frac{pixelCountAND[i]-pixelCountANDMin}{pixelCountANDMax-pixelCountANDMin}$;
4 $\quad \pi[i]_1 = \frac{pixelCountAND[i]}{pixelCount[i]}$;
5 $\quad \pi[i]_2 = \cdots$;
6 $\quad \vdots$
7 $\quad \pi[i]_k = \cdots$;
8 $\quad \pi[i] \longleftarrow e^{-c\cdot(1-(c_1\cdot\pi[i]_1+c_2\cdot\pi[i]_2+\cdots+c_k\cdot\pi[i]_k))}$;
9 **return** π

4.3 Optimizations

As seen in Section 6.2 the overall performance of this approach has to be improved. As already having rendered the model of every pose on the GPU the decision is to continue processing there with CUDA, see Section 5.2.3. Thus, the most time-intensive part of the approach – Algorithm 4, the rating of every pose – is moved to the GPU. The imaging operations like the Sobel operator, the binarization, the computation of the sum of the pixels or the binary AND perfectly fit into CUDA as the operations and the image are 2D data parallel and can be executed as a single program with multiple data.

Figure 4.13 depicts the optimization process. The calculated pose of all particles is transferred to the GPU which renders the corresponding object to each pose. After having rendered it, the result is mapped from the memory space of OpenGL into CUDA. To adapt the approach to the hardware, CUDA is running on the image that is divided into several parts of same size. Each part can be computed autonomously, thus, the approach is scalable.

Each part is computed by a group of threads called a thread block. To avoid concurrency errors synchronization is required among all the thread blocks. Finally, the calculated probability of each pose is transferred back from the GPU. The measured runtime can be seen in Section 6.3.

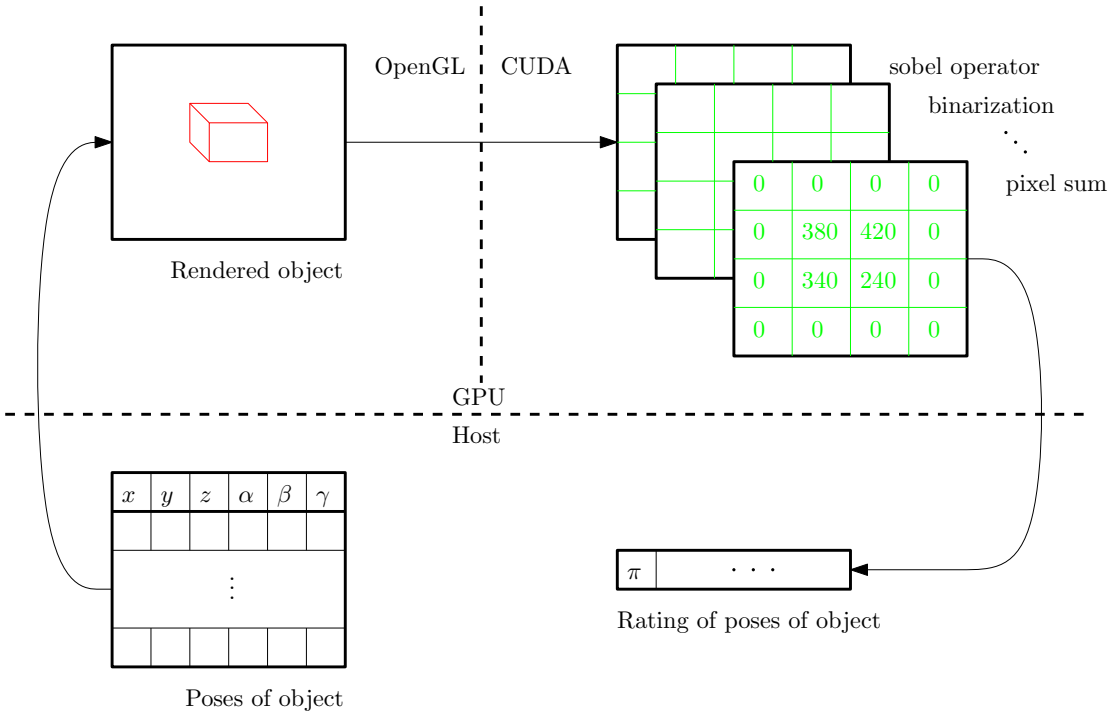


Figure 4.13: From the pose of the object to its probability. The objects belonging to the calculated poses are rendered with OpenGL on the GPU. Then, the image of this rendered scene is mapped into the memory space of CUDA. The image is divided into different blocks of the same size. On each block several threads apply the Sobel operator, the binarization, the binary AND and the computation of the sum of the pixels. Finally, the calculated probability is transfered to the host.

Chapter 5

Software and Interfaces

The following chapter describes the environment taken into account for all developed and evaluated algorithms above. First, the hardware and software being used will be reviewed, second, the Integrating Vision Toolkit, its optimizations and the Compute Unified Device Architecture will be presented. Third, the class diagram and some details of the implementation of the approach above will be explained and finally, the KIT ObjectModels Web Database will be introduced.

5.1 Hardware

The evaluations in Chapter 6 have been performed on a workstation equipped with an Intel Core 2 Duo E8400 running at 3,00 GHz, with 4 GB Random Access Memory (RAM) and a Nvidia Geforce GTX 280 GPU.

5.2 Software

Microsoft Windows Vista Business 32 Bit with Service Pack 2 has been used as the operating system. The decision for using the 32 Bit operating system instead of 64 Bit is due to more stable software libraries and existing well working camera drivers.

The Microsoft Visual C++ 2008 Express Edition has been used as development environment and compiler.

As the version 2.0.3.12 of the FlyCapture 2 developed by Point Gray Research does not work with Windows Vista the FlyCapture 1.8.2.0016 libraries have been used, see Appendix D.

5.2.1 Integrating Vision Toolkit

The Integrating Vision Toolkit[1] (IVT) is a flexible object-oriented C++ computer vision library. By now, the IVT has come to be a multi purpose platform independent framework

[1] `http://ivt.sourceforge.net`

in some areas of computer vision. Algorithms not yet supported can be integrated by an OpenCV wrapper offered by the IVT. The main features of the IVT are

- support for various cameras,
- monocular and stereo vision,
- different filters,
- color segmentation,
- Hough transform,
- point operations,
- SIFT features,
- Harris corner detector,
- undistortion,
- rectification,
- linear least squares,
- particle filtering framework,
- data structures for ease of handling and an
- own GUI toolkit.

The IVT offers an object-oriented architecture and it is easy to use. Additionally, a documentation is available for some classes as well as a variety of example applications. An impressive performance boost for the IVT is offered by the company Keyetech UG.

5.2.2 Keyetech Performance Primitives

The Keyetech Performance Primitives[2] (KPP) have been available since 2010 being a high-performance library for image processing routines to be used by the IVT or independently. The major advantage of the KPP is that they are platform and operating system independent on Intel CPUs offering MMX and SSE2 (Intel Pentium 4, Intel Core 2 Duo, Intel Core 2 Quad, Intel Atom, or compatible) and their impressive speedup[2] compared to other existing imaging routines.

5.2.3 Compute Unified Device Architecture

Over the last few years parallel programming has turned into a popular area in computer science. Theoretical basics of parallel programming have been developed since the 1950s [33][34], but no affordable, parallel hardware was available for the consumer market. Times changed in 2005[3] when Intel released its first mainstream multi-core CPU, which was the advent of main street's parallel programming. Considering that GPUs already are many-core

[2] `http://www.keyetech.de/en/products/keyetech-performance-primitives.html`
[3] `http://download.intel.com/pressroom/kits/IntelProcessorHistory.pdf`

processors, in 2007 NVIDIA introduced their architecture *Compute Unified Device Architecture* (CUDA). There are three reasons why parallel programming with CUDA is getting more and more popular: the hardware is now available, it is comparably cheap and a great number of consumer computers have a CUDA-capable NVIDIA GPU.

A modern GPU is no longer only a memory controller and display generator as it used to be in the 1990s. Instead, it is a highly parallel and multithreaded multiprocessor. Being both a programmable graphics and a scalable programming platform, a modern GPU is breaking the mould concerning the variety of capabilities. To take advantage, it was necessary to add some processor instructions and memory hardware to the GPU and provide a more general API. With these modifications the data-parallel GPU can be used as a general-purpose, programmable many-core processor with its own benefits and limitations. The modern GPU is characterized by its large amount of floating-point processing power, which can be exploited for non-graphical problems. This was the birth of the programming model CUDA which bypasses the graphics API of the GPU and allows simple implementations of programs in C. Single-Program, Multiple Data (SPMD) is the underlying abstraction to achieve high parallelism on thread level. In the SPMD style of parallel programming all the threads execute the same code on different portions of data, see [35]. The coordination is done with a barrier synchronization method. In summary, the three key abstractions of the CUDA programming model are:

- hierarchy of thread groups,
- shared memory and
- barrier synchronization.

The two components of the programming system are the *host* (=CPU) and at least one *device* (=GPU).

$$host \quad \xrightarrow{\text{uses as a coprocessor}} \quad device$$

The host calls and controls functions running massively parallel on the device. The host code has a few extensions of a programming language or API to specify the execution parameters for device functions, to control the device, memory and context management and more. Currently, the functions callable on the device are limited by those provided by the high or low-level CUDA APIs. They comprise some mathematical, texture and memory functions as well as barrier synchronization.

NVIDIA's marketing department is successfully advertising their CUDA-capable GPUs and promising an easy and instantly learnable programming model resulting in speedups of factor 10 to 1500[4]. However, a more detailed and a closer inspection reveals a programming model with many limitations compared to standard programming, such as recursion and the IEEE 754 standard.

The applied key components of CUDA 3.0 public beta – used in this work – are:

- NVIDIA Driver for Microsoft Windows Vista with CUDA Support (196.34),
- the CUDA Toolkit version 3.0 for Windows Vista 32bit,
- the CUDA SDK 3.0 for Windows Vista

[4] `http://www.nvidia.com/object/cuda_showcase_html.html`

- and the CUDA Visual Profiler 3.0.

The Profiler is only fully compatible with Windows XP, under Windows Vista the memory transaction counters do not work. The CUDA resources are free to download from the web[5]. A good introduction to CUDA can be found in [36].

5.2.4 Class Diagram

Figure 5.1 depicts the class diagram of the implemented classes. The existing interface `CRigidObjectTrackingInterface` and skeleton `CParticleFilterFramework` of the IVT have been extended by two classes: The class `CParticleFilter6D` implements the particle filter for 6-DoF and the class `CParticleFilterUniversalTracker` using the `CParticleFilter6D` organizes all the methods required for tracking. According to [37] the applied software design pattern is called "template method".

5.2.5 User Interface

The aim of the developed approach is the application of arbitrarily shaped objects as well as an universal use. This is achieved by an object-oriented design and few different parameters. As it is not necessary to view the result of the algorithm online in a GUI the initialization of `GLContext` is sufficient. The parameters necessary for the constructor of the tracking class are:

- an object,
- the texture mapping of the object,
- the corresponding texture bitmap,
- the number of particles,
- the particle filters variance,
- the flag of using a textured object,
- the initial pose,
- the edge detection mode,
- the low threshold for Canny edge detector,
- the high threshold for Canny edge detector and
- the threshold for sobel and prewitt operator.

After calling the constructor, the `Init` function has to be called together with a camera calibration file. In the main step, capturing the scene and calling the `Track` function returning the estimated pose of the object is performed in a loop.

[5] `http://developer.nvidia.com/object/cuda_3_0_downloads.html`

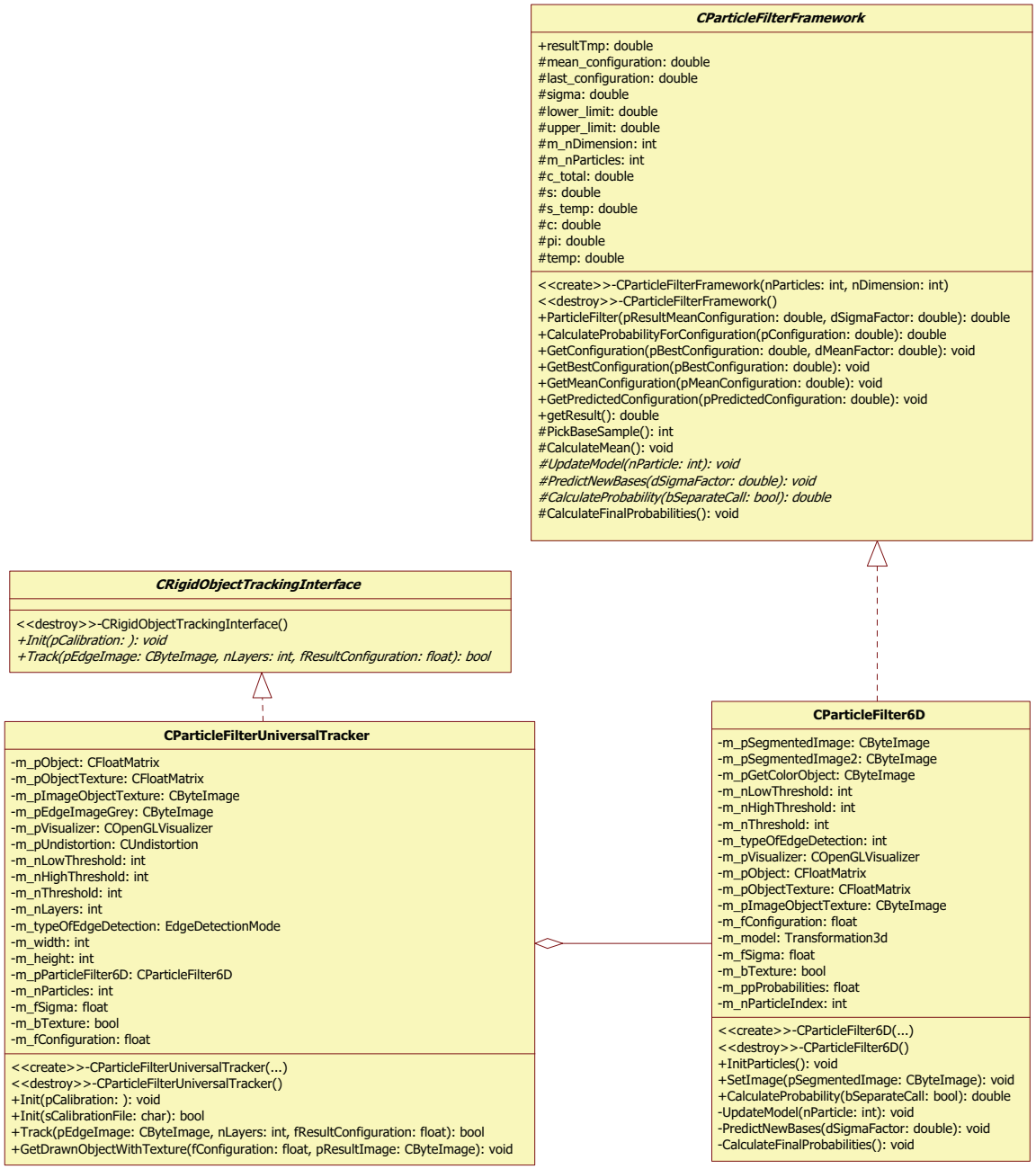


Figure 5.1: UML class diagram of the implemented approach.

5.3 KIT ObjectModels Web Database

The KIT ObjectModels Web Database[6] has been founded at the Institute for Anthropomatics[7] with the aim to provide a large variety of accurate 3D models of objects which can be found in kitchens that can be used for the interaction with a humanoid robot as mentioned

[6] http://i61p109.itec.uni-karlsruhe.de/ObjectModelsWebUI/
[7] http://www.informatik.kit.edu/1323.php

above (see [38, 39, 40]). By now, there are 100 3D models available. Each object is available as a mesh model based on triangles in different file formats with 800, 5,000, 25,000 or 200,000 faces. Additionally, the textural information of each object is available as a TIFF and a PNG image. A XML file contains several information on the scanning process and in the near future it will also contain some properties such as weight and material. The objects are scanned in 3D with a high-accuracy Minolta "VI-900" laser scanner with active triangulation and an error below $0.2\,mm$. The images containing the textural information are taken with a "Marlin 145C2" stereo color camera system from Allied Vision Technologies.

Chapter 6

Evaluation

In this chapter the evaluation of the developed approach will be presented. Due to missing ground truth in real world data the correct pose of the object can only be measured manually frame-by-frame. Though, an accurate plot of errors is not created and the result is observed manually. In simulation mode, however, the correct pose of the object is available and the errors can be determined correctly: First, the accuracy with different parameters and sample data sets is evaluated and second, the velocity performance is investigated. Finally, the results of the algorithm using CUDA are determined.

6.1 Accuracy

Apart from the performance of an algorithm it is necessary to know if the realized approach is working and to what degree it is working correctly. Thus, the first step in simulation mode is to vary different parameters and to extract the best ones – assuming their existence. Additionally, the approach is carried to its limits. Then, the error of accuracy is determined for different objects in simulation mode. Finally, it is applied to some real world instances with major success.

6.1.1 Comparison of Different Parameters

One of the tasks in the developed approach is to adjust a set of different parameters: The amount of particles, the number of layers and the edge detection operators. In this section an explorative step-by-step comparison of the different parameters is given. Due to the different results an optimal set of parameters is established.

In the following experiments the cup of Figure 6.1 was used in simulation mode. As the images are generated from rendered objects the exact pose of the model in each frame is known. Thus, the error of the tracking approach can be easily determined. As the cup has a rotational symmetry axis the focus is laid on the translational errors in the following charts. Figure 6.3 shows the errors of the x, y and z axis using the Canny edge detector and applying a Gaussian filter in the preprocessing of the input edge image. 100 particles and one layer were used. There was no translational movement of the cup. According to the chart the z-

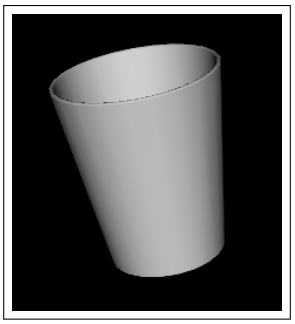


Figure 6.1: Sample object cup.

Figure 6.2: Static image sequence of measuring cup.

axis is considerably more error-prone than the x- and y-axis. This observation was expected due to fewer change of pixels while moving the object in z-direction.

Figure 6.4 and 6.5 depict the results with Sobel and Prewitt operators instead of the Canny edge detector. In this case the error is halved compared to the Canny edge detector.

To compare the three different methods above Figure 6.6 shows the calculated error $\sqrt{x^2 + y^2 + z^2}$. In this instance the Sobel operator shows the best results. These results were not expected and more investigations using the Canny edge detector showed thin edges, being difficult to match exactly with only few particles. In contrast, the Sobel operator generates broader edges which are easier to be met by the model. The idea to make use of the accuracy of the Canny edge detector results in the application of a dilation operation on the input edge image. Figure 6.7 depicts the error of the pose of the object applying a dilation operation or a Gaussian filter on the input edge image. In average applying a dilation operation is more effective than applying a Gaussian filter.

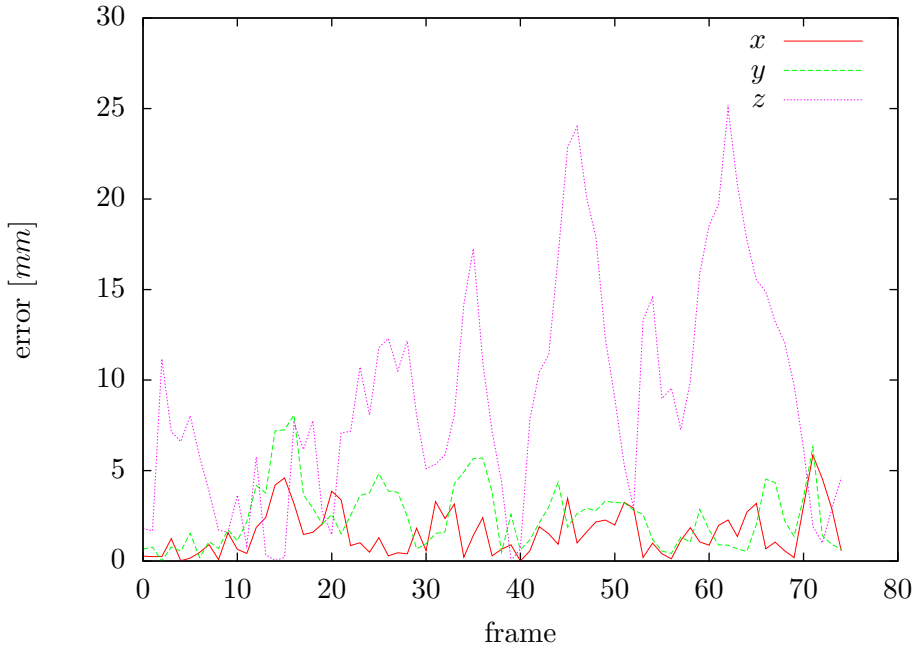


Figure 6.3: Absolute error of translational axis using Canny edge detector. *Parameters:* 100 particles, one layer, variance of the particle filter 5 mm resp. 2°, no translational movement, object cup (see Figure 6.1), initial pose $15.9, 14, 494, 91°, 15°, -9°$.

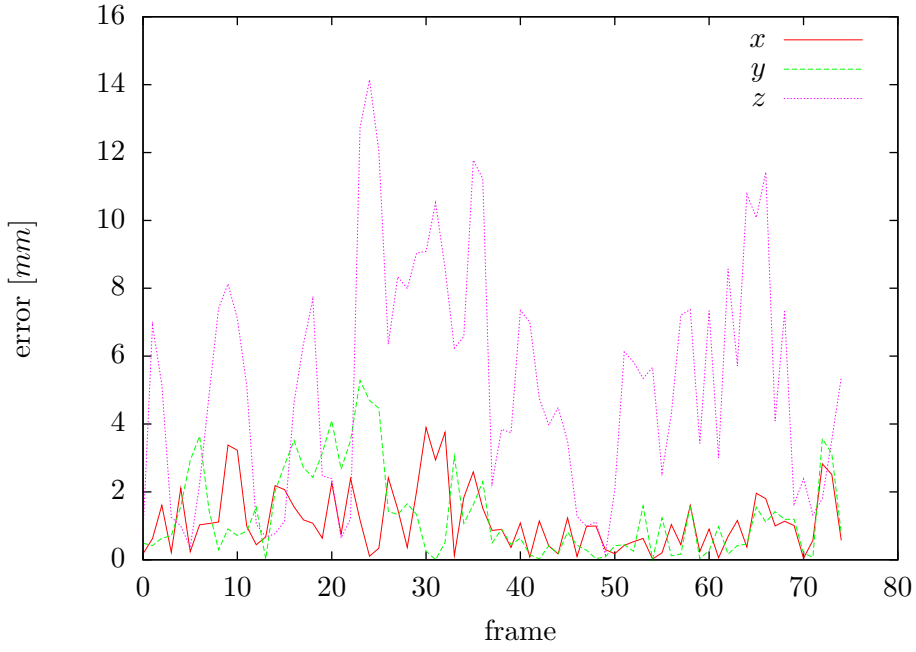


Figure 6.4: Absolute error of translational axis using Prewitt operator. *Parameters:* 100 particles, one layer, variance of the particle filter 5 mm resp. 2°, no translational movement, object cup (see Figure 6.1), initial pose $15.9, 14, 494, 91°, 15°, -9°$.

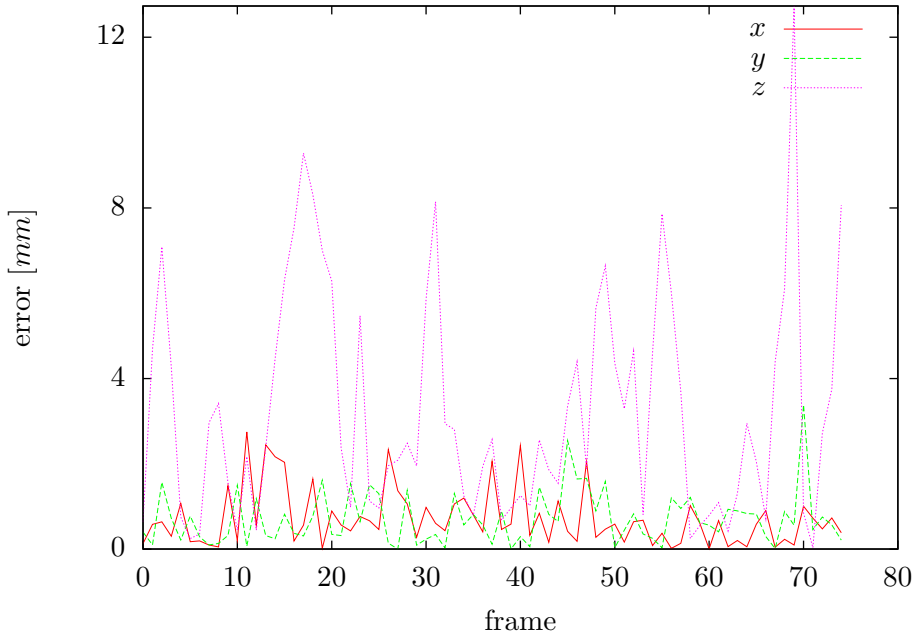


Figure 6.5: Absolute error of translational axis using Sobel operator. *Parameters:* 100 particles, one layer, variance of the particle filter $5\,mm$ resp. $2°$, no translational movement, object cup (see Figure 6.1), initial pose $15.9, 14, 494, 91°, 15°, -9°$.

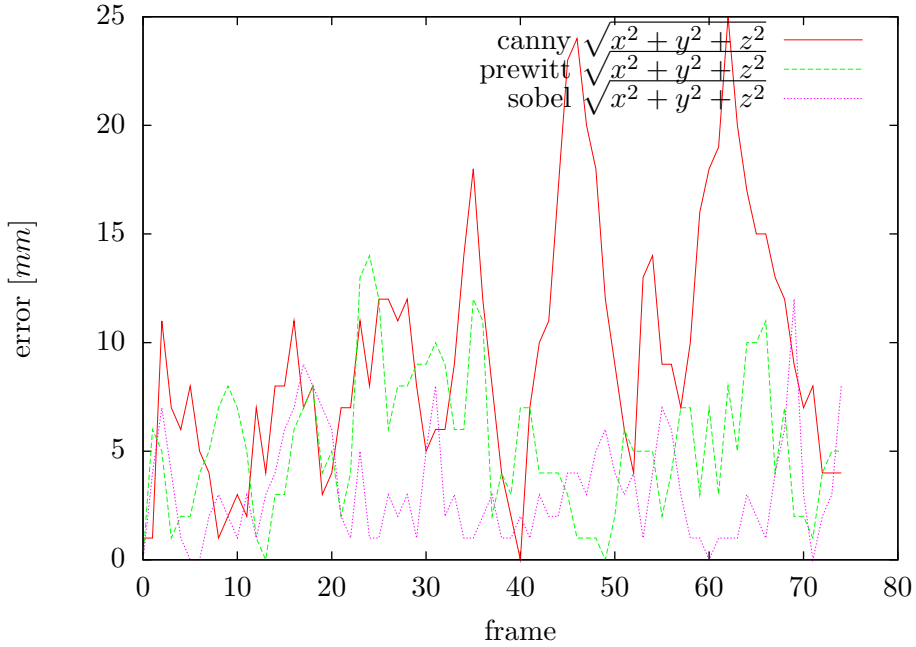


Figure 6.6: Comparison of Canny, Prewitt and Sobel. *Parameters:* 100 particles, one layer, variance of the particle filter $5\,mm$ resp. $2°$, no translational movement, object cup (see Figure 6.1), initial pose $15.9, 14, 494, 91°, 15°, -9°$.

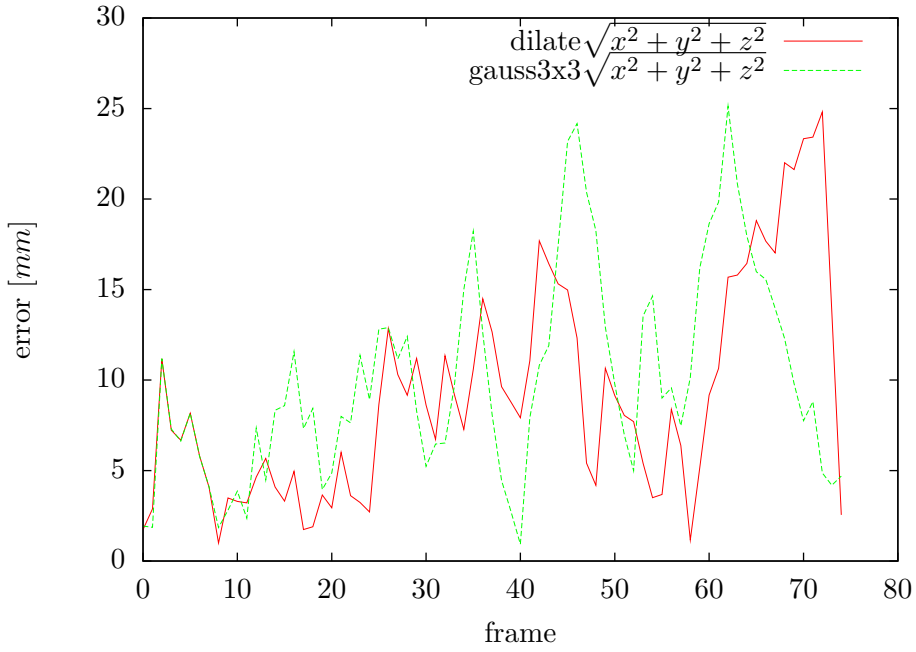


Figure 6.7: Comparison of dilation operation and Gaussian preprocessing on input image. *Parameters:* 100 particles, 1 layer, variance of the particle filter $5\,mm$ resp. $2°$, no translational movement, object cup (see Figure 6.1), initial pose $15.9, 14, 494, 91°, 15°, -9°$, Canny edge detector.

In Figure 6.8 the same measurement is performed except of using five layers instead of one. The resulting error of the pose estimation is reduced significantly. In Figure 6.9 the correlation of errors and layers is investigated. A slightly better result can be seen using more layers. In Table 6.1 the standard deviations of the pose estimation in the static real world scenario of Figure 6.2 are summarized.

particles	$\sigma(x)\,[mm]$	$\sigma(y)\,[mm]$	$\sigma(z)\,[mm]$	$\sigma(\alpha)\,[°]$	$\sigma(\beta)\,[°]$	$\sigma(\gamma)\,[°]$
10	9.504163176	2.939950779	8.697365146	11.39972943	6.931195345	4.749352007
50	2.113246738	1.419224743	7.013441555	4.068941372	4.352667504	4.182579826
100	1.634655337	1.169513526	5.217948338	3.679418742	3.860799455	3.657213207
500	0.784534588	0.354221639	1.357680855	1.759365106	1.611197696	2.970091903
1000	0.55611226	0.226365405	0.978019227	1.220483623	1.300958196	2.086690242
2000	0.415776187	0.159751087	0.707134304	0.804579698	0.874325139	1.542158627

Table 6.1: Standard deviation of pose estimation of static object (see Figure 6.2)

In this section the parameters for the developed approach were investigated. In general the combination of several hundreds of particles, some layers, a dilation operation on the edge images and the Canny edge detector produce good results.

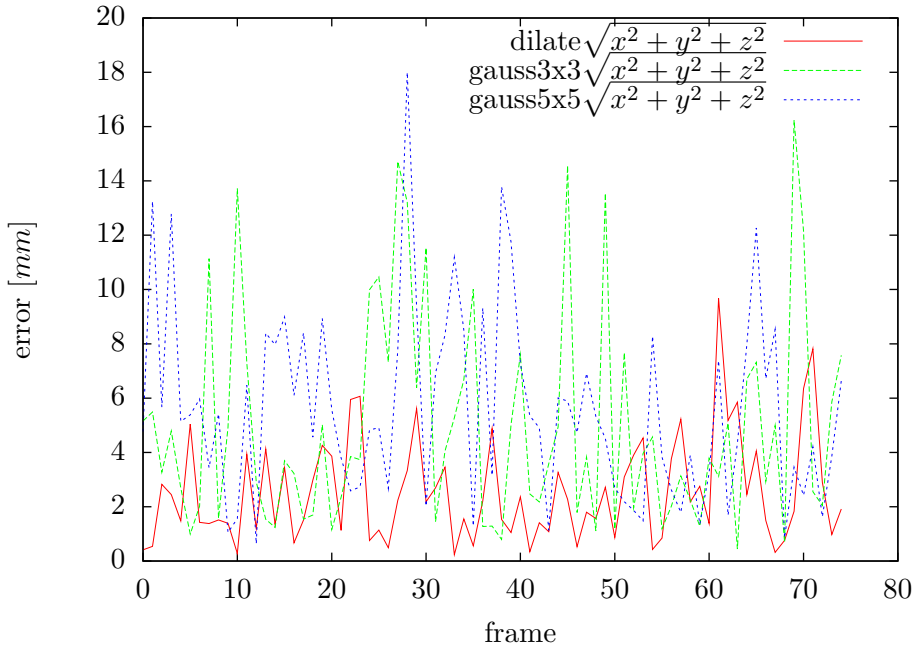


Figure 6.8: Comparison of 3×3 dilation and Gaussian and 5×5 Gaussian preprocessing on input image. *Parameters:* 100 particles, 5 layers, variance of the particle filter $5\,mm$ resp. $2°$, no translational movement, object cup (see Figure 6.1), initial pose $15.9, 14, 494, 91°, 15°, -9°$, Sobel operator.

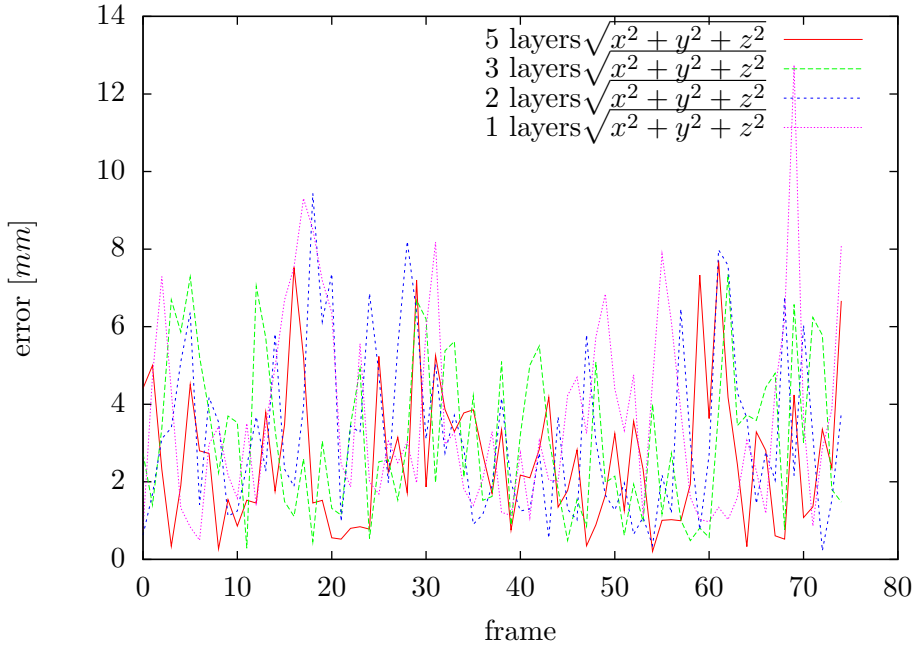


Figure 6.9: Comparison of different number of layers. *Parameters:* 100 particles, variance of the particle filter $5\,mm$ resp. $2°$, no translational movement, object cup (see Figure 6.1), initial pose $15.9, 14, 494, 91°, 15°, -9°$, Sobel operator.

6.1.2 6-DoF Tracking in Simulation Mode

In this section some image sequences generated with the corresponding 3D model are investigated. In Figure 6.11 a random walk of a cooking oil can can be seen. The variance of the movement was $5\,mm$ per frame for each translation axis and $2°$ per frame for each rotation angle. Assuming 30 frames per second, this corresponds to a movement of about $150\,mm$ and $60°$ per second.

The images in Figure 6.11 are the original input images with the result of the tracking projected onto them as white edge-models. As in this case all six dimensions have to be considered, a slightly higher number of 1000 particles and 10 layers is used. Figure 6.10 and 6.12 visualize the translational and rotational error for each axis. The average error is $0.225\,mm$ for the x-axis and $0.394\,mm$ for the y-axis. Compared to these results the average error of $5.385\,mm$ for the z-axis seems to be quite high. As the z-axis is the depth axis the movement of the model in this direction is less distinct caused by less change of pixels compared to the movement in x and y direction. According to Figure 6.12, no rotational axis has outstanding errors. The combination of the error of all three rotations is visualized in Figure 6.13. As expected, the result has mostly a small rotational error of less than three degrees. It can be seen that textured objects with expressive edges such as the cooking oil can perform well.

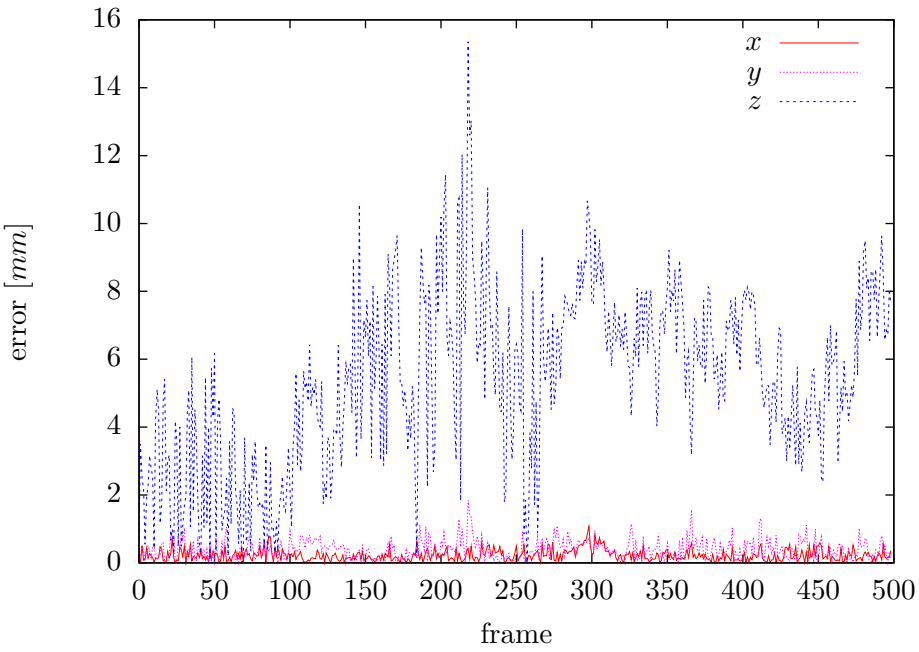


Figure 6.10: Absolute error of translational axis with the cooking oil object.

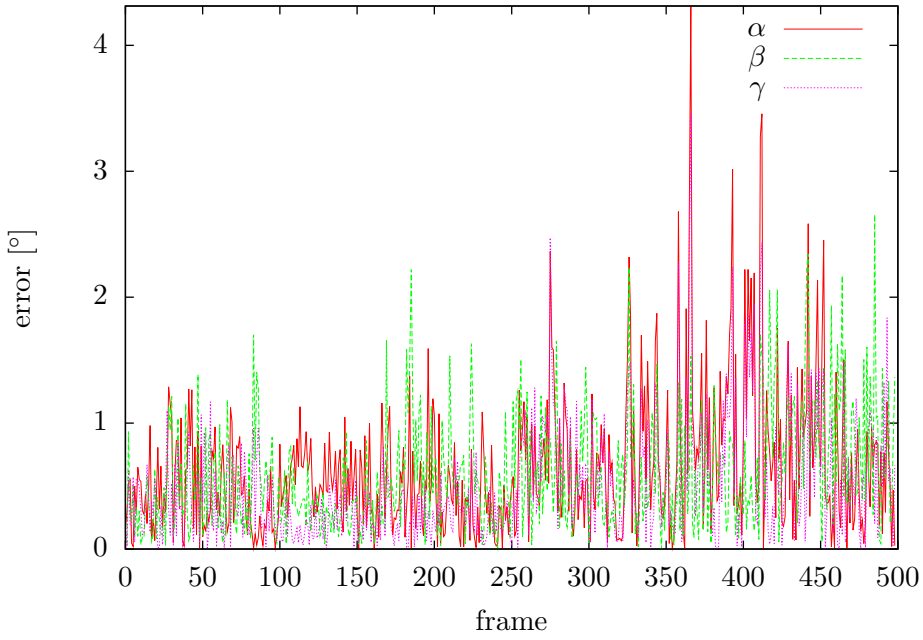


Figure 6.12: Absolute error of rotational angles with the cooking oil object.

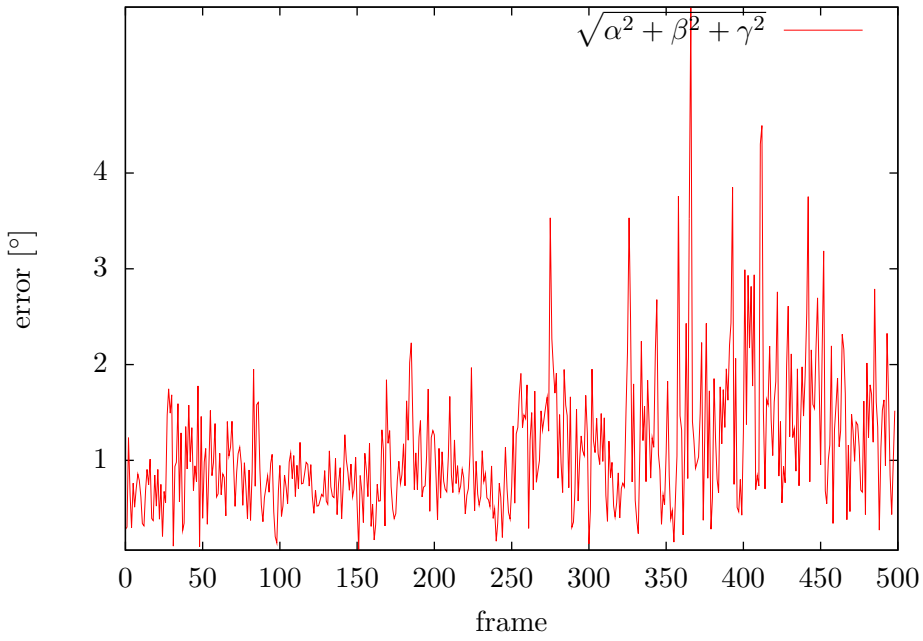


Figure 6.13: Absolute error of rotational angles with the cooking oil object.

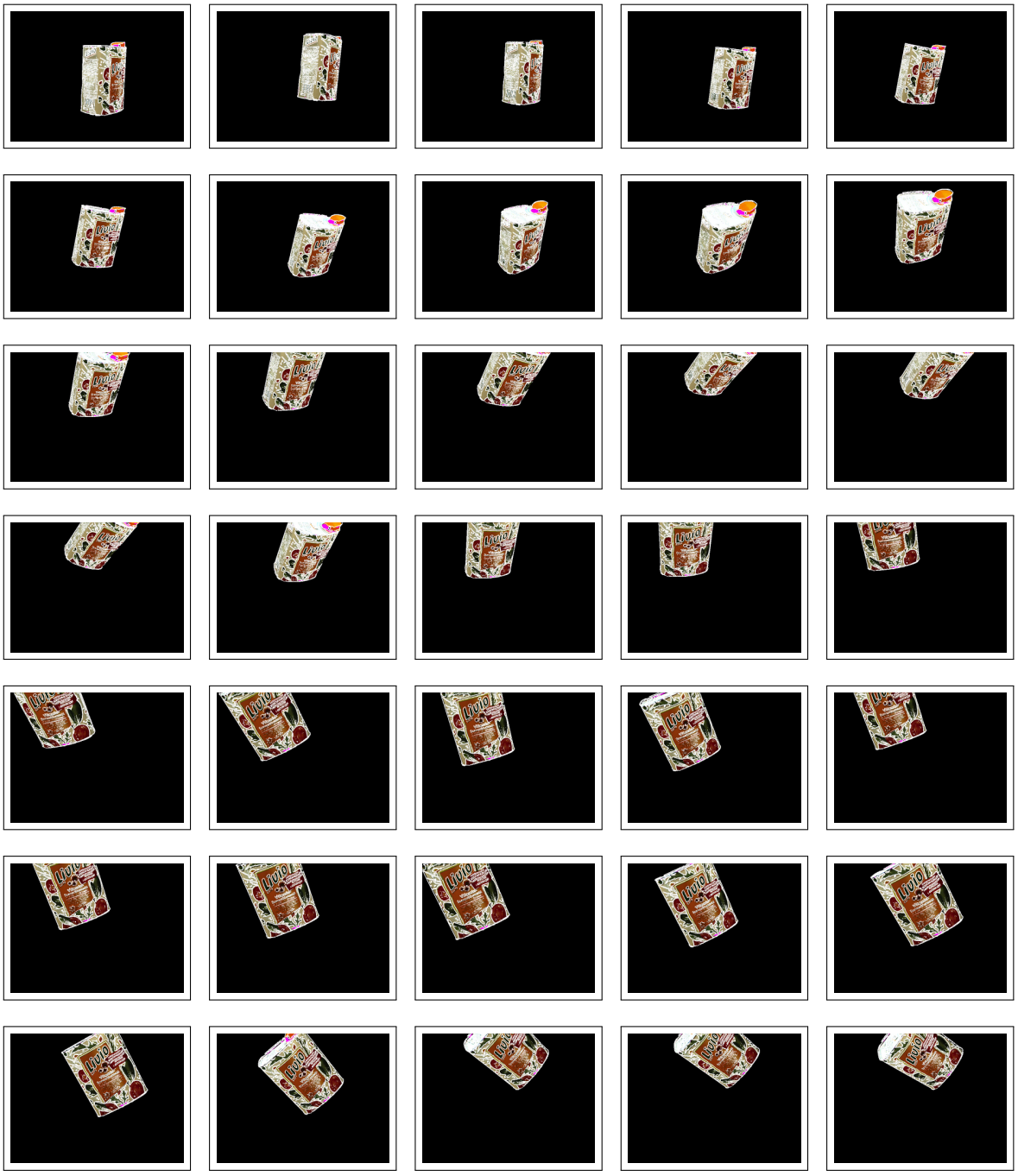


Figure 6.11: Tracking of a colorful textured can of cooking oil in simulation mode. *Parameters:* The image sequence above is an extract, meaning that every fifteenth frame is shown. The Sobel operator was used and an amount of 1000 particles per layer, with a total of 10 layers. The initial configuration is $x = 16\,mm, y = 14\,mm, z = 494\,mm, \alpha = 91°, \beta = 15°$ and $\gamma = -9°$. The variance is $5\,mm$ for translation and $2°$ for rotation.

Figure 6.14 depicts a random walk of a measuring cup. Using 600 particles and 5 layers, which is quite a common configuration, the approach results in an almost perfect pose estimation. Figures 6.15 and 6.16 visualize the translational and rotational error for each axis. The average error is $0.366\,mm$ for the x-axis, $0.282\,mm$ for the y-axis and $4.80\,mm$ for the z-axis. The average error of the rotation around the x-axis, the y-axis and the z-axis is $0.39°$, $0.33°$ and $0.24°$. The combination of the errors of all three rotations is visualized in Figure 6.17. As expected the result has mostly a small rotational error of less than one degree.

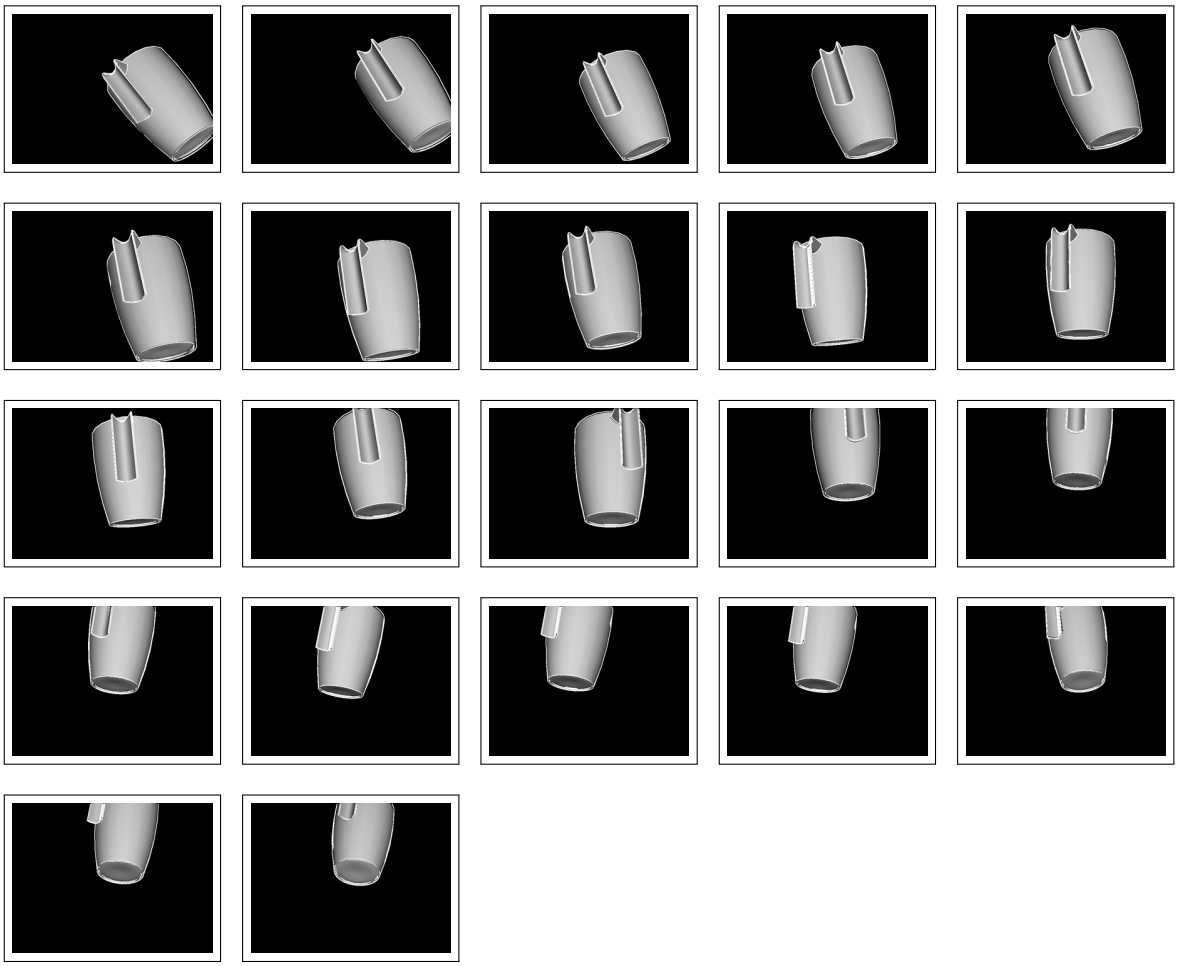


Figure 6.14: Tracking of a blue single-colored measuring cup in simulation mode. *Parameters:* The image sequence above is an extract, meaning that every tenth frame is shown. The Sobel operator was used and an amount of 600 particles per layer, with a total of 5 layers. The initial configuration is $x = 58\,mm, y = -3\,mm, z = 434\,mm, \alpha = -22°, \beta = 32°$ and $\gamma = -40°$. The variance is $5\,mm$ for translation and $2°$ for rotation.

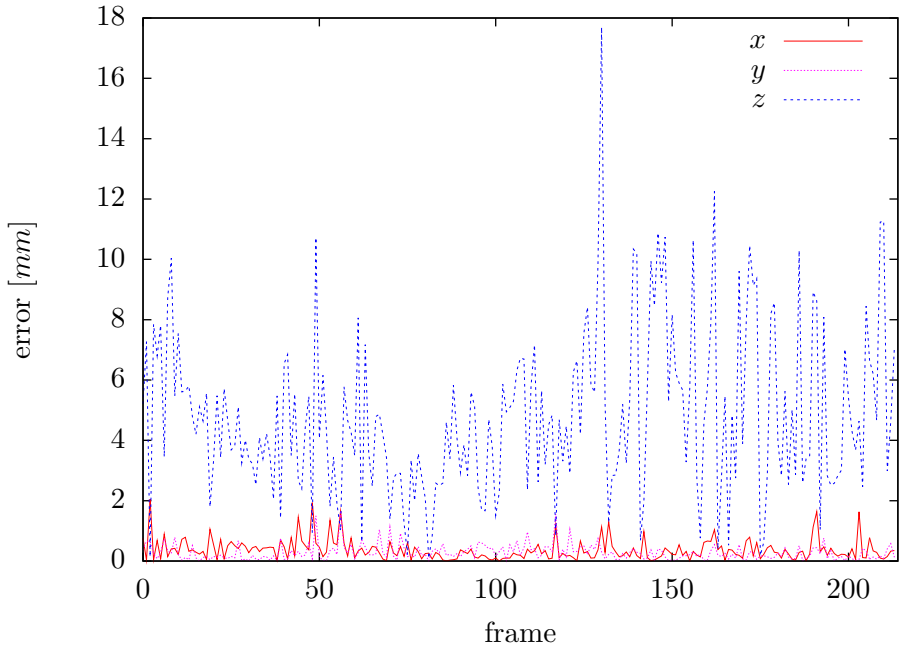


Figure 6.15: Absolute error of translational axis with the measuring cup object.

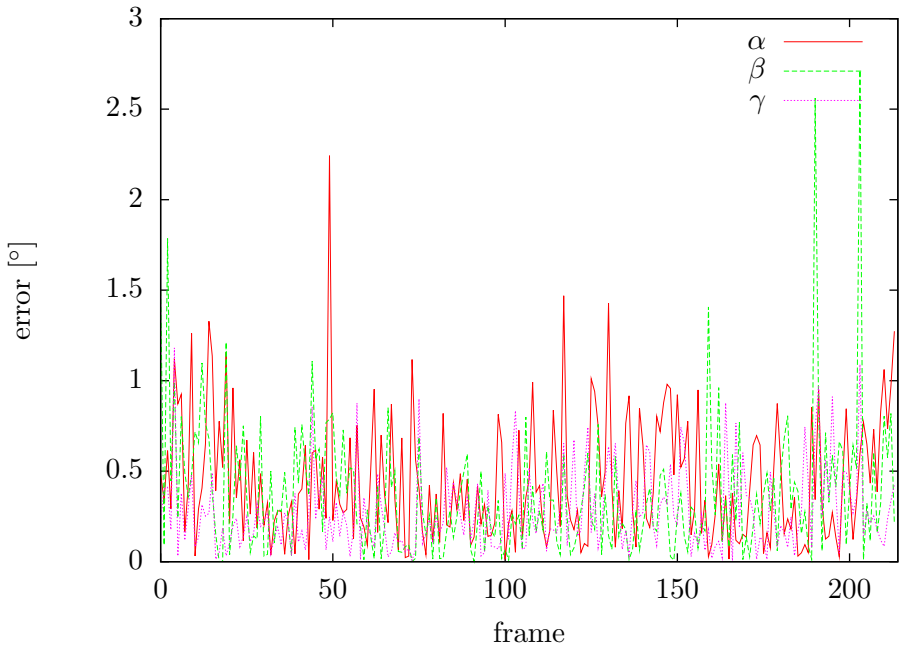


Figure 6.16: Absolute error of rotational axis with the measuring cup object.

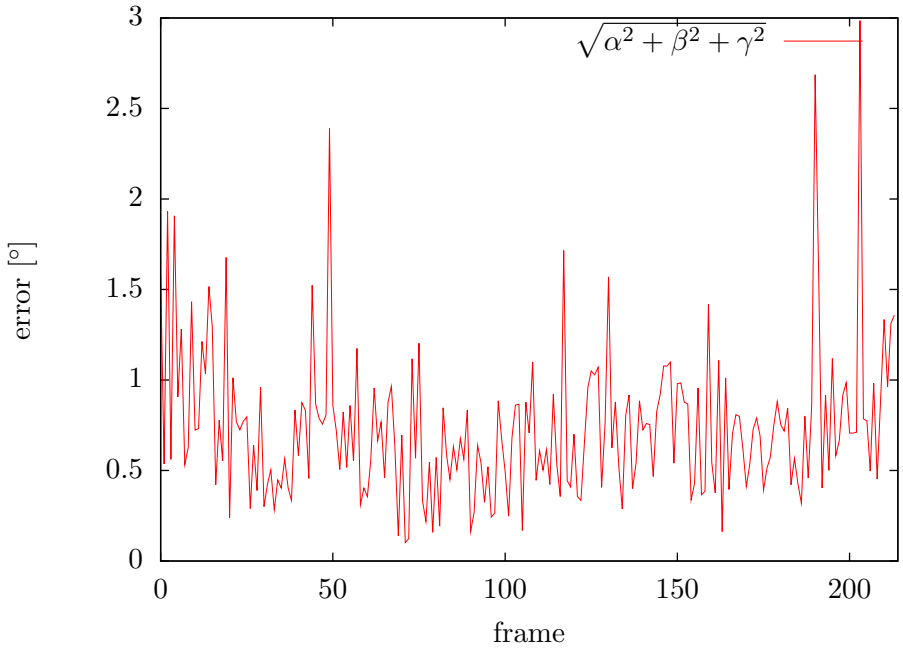


Figure 6.17: Absolute error of rotational axis with the measuring cup object.

6.1.3 Real World Experiments

The developed approach works well in simulation mode (see Section 6.1.2). As there are always constant conditions, meaning no noise and no illumination changes the edge image is throughout perfect. However, in reality the edge images are often of low quality. The question to be answered in this section is whether the approach can deal with real world data.

Therefore, a sequence of images of a moving measuring cup was recorded first, then this data was used as input for the approach. This instance is a common situation in the kitchen as not only the measuring cup is moved translationally but as it is also emptied which is a rotational movement.

To detect the edges the Canny edge detector is used in Figure 6.18. It is an excerpt of the whole image sequence, every tenth image is shown. As it is difficult to determine the correct error the estimated pose is projected in the original input image to compare it visually. The results are good, no major discrepancies can be seen, the trajectory of the object is quite fluent.

As a second object a cup is chosen. The result can be seen in Figure 6.19. Although it is invariant to rotations around the y-axis and the result seems slightly better than in Figure 6.18 another problem arises. In the images 17 to 21 the estimated pose of the cup is wrong, exactly while the cup tilts. This problem is described precisely in Section 4.2.3.

In Figure 6.20 the focus was laid on occlusions of the object. The approach performs well even if more than half of the object is occluded. Additionally, the plate has individual characteristics in its edge model supporting the good results and consequently allowing the use of fewer particles.

Although there already exist performant approaches for simple standard primitives to track (see Chapter 2) the tracking of a cuboid is evaluated. Tracking results of the cuboid are good as it can be seen in Figure 6.21.

In Figure 6.22 a blue small bowl was tried to track. Even if the result is useful there can be seen a rotational error. To investigate the cause of error of this pose estimation the input edge images and the binary AND processed result images are presented, too. It can be seen that only few edge pixels in the input image belong to the bowl and the AND images have a lot of matching pixels due to edge clutter of the hand. The causes of the problem are

- the bowl is small,
- it has only few edge-based information,
- there are illumination irregularities and
- similarity in the edges of the hand compared to the edges of the bowl.

Thus, a small bowl is not suitable for the proposed tracking approach as it cannot be grasped without occluding too much edge information and adding wrong edge information with the grasping hand.

Finally, a textured object, a box of soup, is evaluated (see Figure 6.23). Due to the texture of the soup box much more information can be extracted from the edge images. If there are many edges and if they are close together the correct pose of the object will blur and the accuracy will decrease.

6.2 Runtime

Tracking objects is mostly a real-time task. In this paragraph the performance of the approach above is evaluated. In Figure 6.24 the performance of the approach in dependence of the number of particles used is shown. The software and hardware environment is described in detail in Chapter 5. The parameters of the instance are one layer, the Sobel operator as well as the cup object. The processing time of the algorithm includes capturing the image, setting and preprocessing the image and applying the particle filter.

For one particle the algorithm needs about $16.4\,ms$ which is equivalent to 60.9 frames per second, thus, this is the lower bound of the approach with the parameters mentioned above. One can see that with an increasing number of particles the algorithm scales linearly. Compared to real-time performance the developed approach is currently too slow. With up to ten particles a performance of more than 30 frames per second is achieved, however, ten particles is much too less. The investigation of the whole algorithm will be presented below in detail.

Deutscher et. al. propose in [29] an annealed particle filter. Their idea is to use fewer particles and the division of the particle filter into several layers. The correlation of the number of layers and the runtime is shown in Figure 6.25. The runtime scales linearly with the number of layers, thus, the runtime is not affected if there are ten layers of 100 particles or one layer with 1000 particles.

Figure 6.18: Tracking of a blue single-colored measuring cup. *Parameters:* Every tenth frame of a sequence of 350 frames is shown. The Canny edge detector was used and an amount of 400 particles per layer, with a total of 6 layers. The initial configuration is $x = 20\,mm, y = 138\,mm, z = 888\,mm, \alpha = 28°, \beta = -126°$ and $\gamma = 109°$. The variance is $5\,mm$ for translation and $2°$ for rotation.

Figure 6.19: Tracking of a yellow single-colored cup. *Parameters:* Every fifteenth frame of a sequence of 525 frames is shown. The Sobel operator was used and an amount of 500 particles per layer, with a total of 5 layers. The initial configuration is $x = 57\,mm, y = 61\,mm, z = 480\,mm, \alpha = 0°, \beta = 1°$ and $\gamma = -1°$. The variance is $5\,mm$ for translation and $4°$ for rotation.

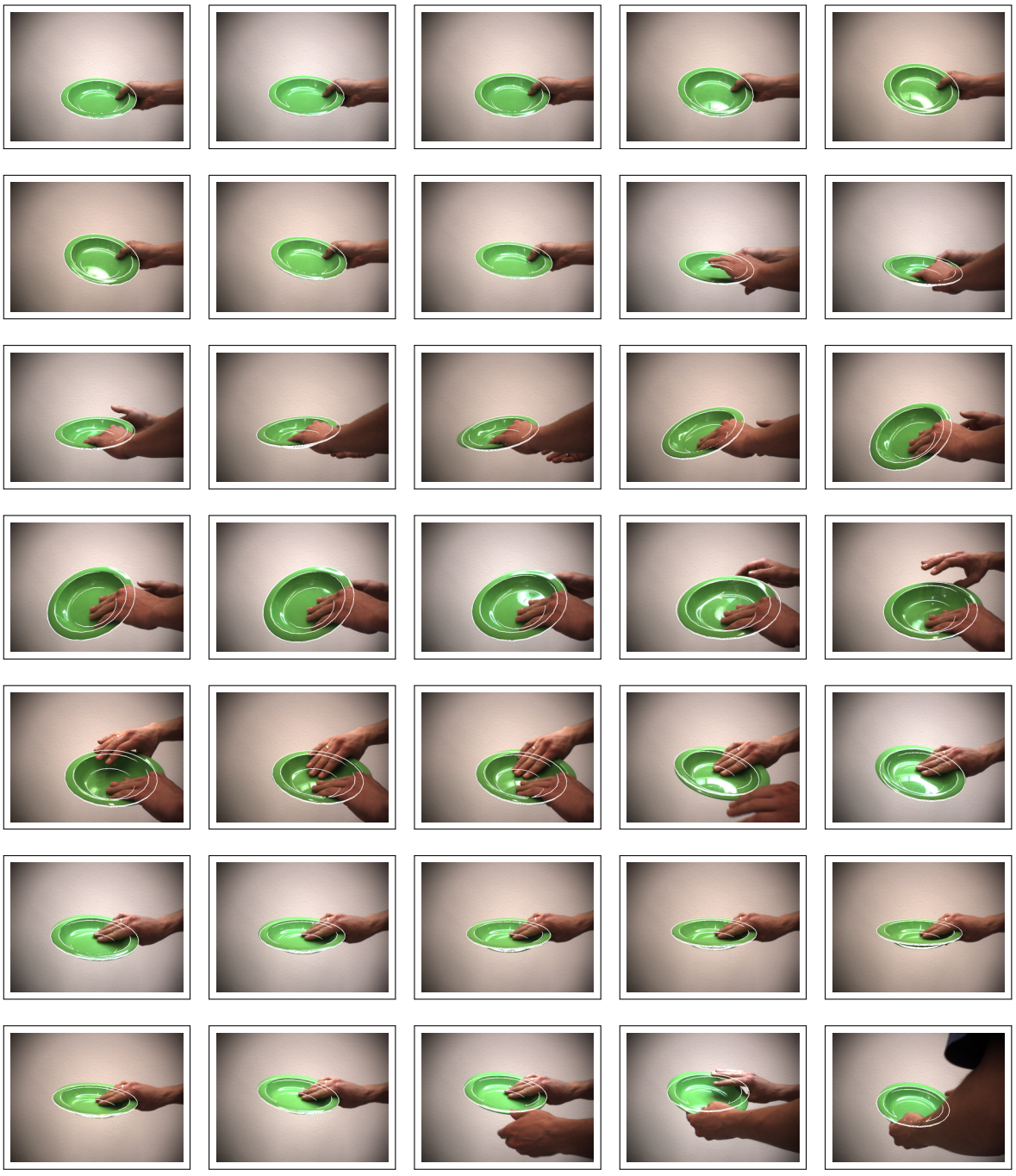


Figure 6.20: Tracking of a green single-colored plate. *Parameters:* Every tenth frame of a sequence with 350 frames is shown. The Sobel operator was used and an amount of 250 particles per layer, with a total of 6 layers. The initial configuration is $x = 5\,mm, y = 40\,mm, z = 750\,mm, \alpha = 29°, \beta = 15°$ and $\gamma = 6°$. The variance is $5\,mm$ for translation and $2°$ for rotation.

Figure 6.21: Tracking of an orange single-colored cuboid. *Parameters:* Every tenth frame of a sequence of 230 frames is shown. The Sobel operator was used and an amount of 250 particles per layer, with a total of 6 layers. The initial configuration is $x = 36\,mm, y = 37\,mm, z = 676\,mm, \alpha = 60°, \beta = -23°$ and $\gamma = 37°$. The variance is $5\,mm$ for translation and $2°$ for rotation.

Figure 6.22: Tracking of a blue single-colored bowl. *Parameters:* Every tenth frame of a sequence of 100 frames is shown. The Sobel operator was used and an amount of 250 particles per layer, with a total of 3 layers. The initial configuration is $x = 52\,mm, y = -2\,mm, z = 700\,mm, \alpha = 17°, \beta = 0°$ and $\gamma = 6°$. The variance is $5\,mm$ for translation and $2°$ for rotation.

Figure 6.23: Tracking of a colorful textured box of soup. *Parameters:*Every tenth frame of a sequence of 130 frames is shown. The Sobel operator was used and an amount of 1000 particles per layer, with a total of 10 layers. The initial configuration is $x = 21\,mm, y = 32\,mm, z = 492\,mm, \alpha = 80°, \beta = 54°$ and $\gamma = -57°$. The variance is $5\,mm$ for translation and 2° for rotation.

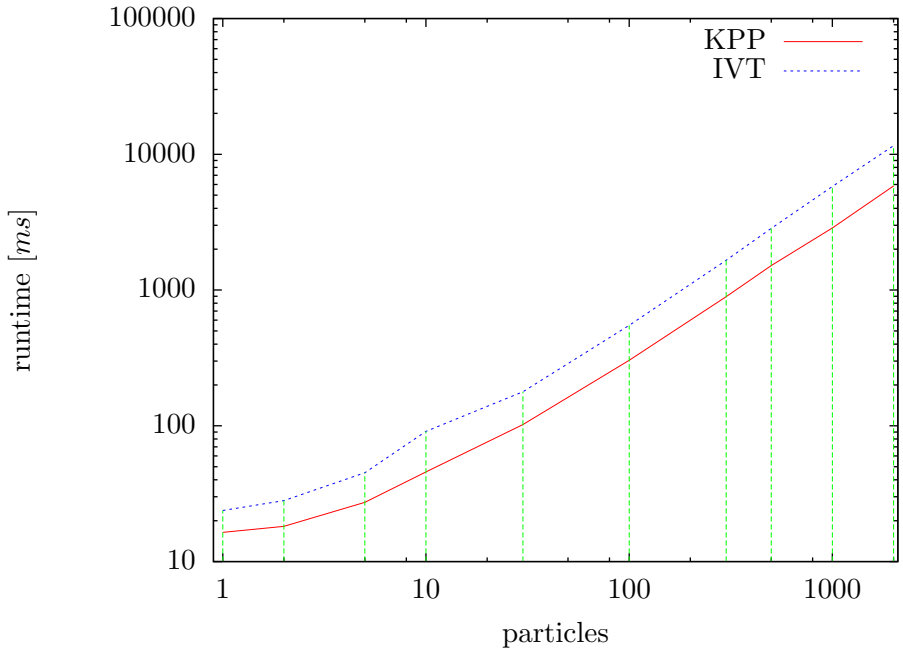


Figure 6.24: Runtime vs. number of particles. *Parameters:* The cup (see Figure 6.1) and the Sobel operator were used with one layer.

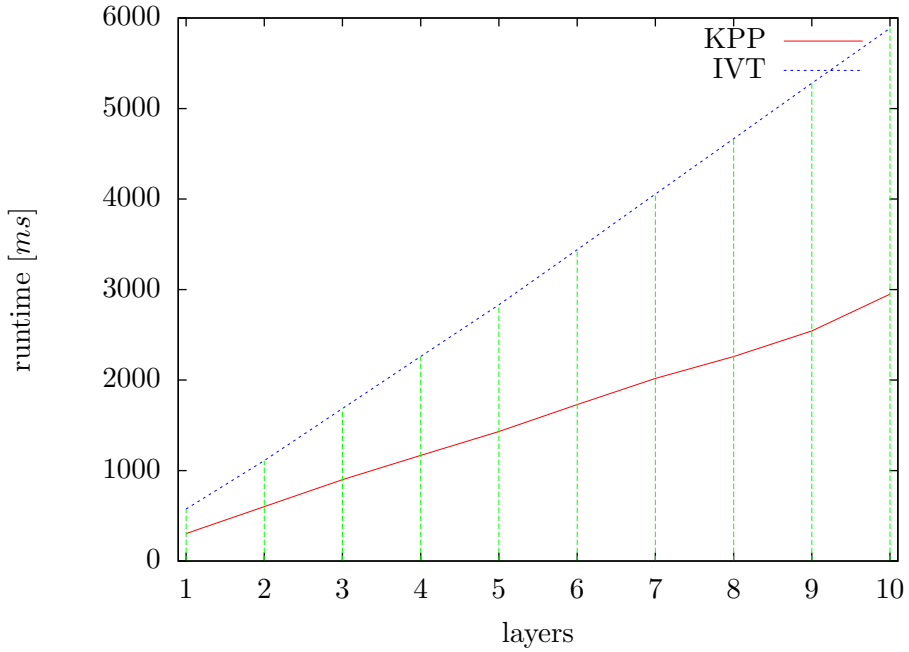


Figure 6.25: Runtime vs. number of layers. *Parameters:* The cup (see Figure 6.1) and the Sobel operator were used with an amount of 100 particles.

In Table 6.2 the runtime of the developed algorithm is compared to the Sobel operator and the Canny edge detector. The Canny edge detector is slower, but as seen in Section 6.1.1 it is sometimes worth using the Canny edge detector to gain better results.

Rendering and transferring the objects to main memory with OpenGL is shown in Table 6.3. The first six objects are without texture, the remaining ones are with texture. One constant factor is: the time of transferring the rendered image from the framebuffer to main memory. As the images have a resolution of 640×480 pixels and as then have $24\,bit$ color depth this corresponds to $900\,kb$ per image. With an readbacktime of `glReadPixels` of about $2.02\,ms$ the upper bandwidth of the frambuffer is 435 $\frac{MB}{s}$ in this case. This bandwidth is a bottleneck and limits the algorithms.

Further, there is an almost linear relation between the objects triangles and the rendering time. The objects with texture are slower as the bitmap file for the texture has to be transferred and mapped, too. Object "Amicelli" is slower due to a larger bitmap file.

Altogether, the whole rendering and readback steps from the GPU are too slow. Skipping the readback to the host was the aim when using CUDA to speed the algorithm up. In the following paragraph the results with CUDA support are presented.

Edge detection	**IVT**		**KPP**		**KPP vs. IVT**
Sobel operator	$550.0ms$	$1.81fps$	$304ms$	$3.29fps$	1.80
Canny edge detector	$881.8ms$	$1.13fps$	$408ms$	$2.45fps$	2.16

Table 6.2: Runtime with different edge detection modes. *Parameters:* The cup (see Figure 6.1) was used with an amount of 100 particles and one layer.

Object file	Triangles	Transfer & Rendering [ms]	Transfer [ms]	Rendering [ms]
cuboid.dat	12	2.05	2.02	0.03
cup.dat	664	2.16	2.02	0.14
bowl_small.dat	860	2.2	2.02	0.18
measuring_cup.dat	1493	2.3	2.02	0.28
plate_deep.dat	1832	2.57	2.02	0.55
measuring_cup_rescan.dat	24532	7.56	2.02	5.54
livio.dat	4946	6.9	2.02	4.88
soup.dat	4986	6.7	2.02	4.68
amicelli.dat	4986	7.6	2.02	5.58

Table 6.3: Rendering time with OpenGL for different objects.

6.3 Optimized with CUDA

One aim of this work was to use CUDA to speed up the algorithm, especially the imaging functions, as they are best candidates for data parallel computations. The OpenGL rendered object is already stored on the GPU and thus, the expensive transfer from the framebuffer of the GPU to the main memory on the host can be skipped. Instead, the rendered object is directly mapped into the memory space of CUDA on the GPU.

In Table 6.4 the runtime of the key steps of the approach is shown. The core components such as the Sobel operator and the binary AND perform $2-3$ times better than the KPP implementations. The expection for using CUDA was an extremely reduced readback time. Unfortunately, the readback is even slower than without CUDA. Additionally, the mapping from the memory space of OpenGL into the memory space of CUDA is time consuming, too. The reason is the framebuffer which is a bottleneck due to slow bandwidth.

Another disadvantage of CUDA is its overhead. Initialization of buffers, the transfers from main memory to the GPU and the memory mapping and unmapping cost time, in this case about $1.2\,ms$. On the one hand, the use of CUDA is a success as the imaging steps of the algorithm perform very well on the GPU. On the other hand, the mapping step is slow and it is not possible to track in real-time.

operation	KPP [ms]	CUDA [ms]
glReadPixels	2.02	**2.12**
cudaGLMapBufferObject	–	**1.01**
Sobel fused	0.36	0.12
Thresholding	0.025	0.02
Pixel sum	0.13	0.13
And	0.08	0.03
CUDA related overhead	–	aprox. 1.2
Other	0.42	0.42
Total	**3.04**	**5.05**

Table 6.4: “Benefits” of using CUDA.

Chapter 7

Conclusion

7.1 Summary and Results

The aim of this work was the development of an algorithm tracking arbitrarily shaped and complex objects with an annealed particle filter. The approach is model-based and edge-based using known surface models of the objects rendered with OpenGL.

Robustness in 6-DoF tracking as well as partial occlusions of objects with sufficient edge information can be handled successfully as shown in Chapter 6. Speeding up the approach with CUDA has not shown the expected real-time performance results due to hardware related bandwidth limitations, as discussed in Section 6.3.

In Section 2, the state of the art in edge-based and model-based 6-DoF tracking was analyzed and an overview of the different underlying approaches was given. To ensure a profound understanding of the fundamentals, the camera model, the annealed particle filter framework and the OpenGL were explained.

In the main part, the tracking approach was developed step-by-step with a guiding example application. The input image sequence is preprocessed with edge detectors. Every object gets rendered with OpenGL and an edge detector is used. This image is applied to an annealed particle filter and is compared pixel-by-pixel to the input image to measure the amount of matching edge pixels. The overall rating can be designed flexibly as different parameters can be adjusted. It is a challenge to define the best rating function as it is not universally valid. After rating, the pose estimate is computed as the weighted mean over all particles.

The implementation for developing a specific scenario for object tracking was described afterwards. Additionally, an optimization with CUDA was presented, motivated by the independent and data-parallel weighting process of the particles. Its processing performance is faster than the CPU-based algorithms apart from transferring and mapping data.

Closely related to the development was the evaluation which resulted in an increasing of the robustness, performance and accuracy of the approach. The evaluation of seven different objects illustrates the feasibility of the developed approach in real world scenarios.

As expected, the depth of the estimated poses was more error-prone than translations in horizontal and vertical direction. The computational complexity of the developed approach

– both the CPU-based and the CUDA-based approach – does not yet allow real-time 6-DoF tracking at 30 frames per second for images of size 640×480.

The main challenge in this approach is a rating function satisfying the needs of a correct pose estimation as fair ratings such as the ones described in Section 4.2.3 result in non-optimal pose estimations.

7.2 Future Work

The developed approach in this diploma thesis is not efficient enough to be used on current conventional hardware. An interesting question is whether it can be optimized to run in real-time by now or if this approach has to wait for faster hardware due to performance bottlenecks. Additionally, there are some challenges in the context of the rating function to be overcome such as wrong pose estimations due to "wrong" pixel matches. In the remainder of this section possible future work is presented which will probably lead to a fast and robust approach.

One idea to increase the performance of the developed approach is to use multiple high-end GPUs. In this case more objects can be rendered with OpenGL per time step as the rendering time scales linearly to the amount of available GPUs. One possibility is to use shaders as mentioned in Section 2.5.3. However, in the case of using CUDA the bottleneck of the slow readback from the framebuffer and the slow mapping from the memory space of OpenGL into CUDA should be eliminated.

The computational effort needed by the developed approach is too high for the actual ARMAR-III humanoid robot. With respect to this performance lack ongoing work lies on developing some flexible FPGA-based implementations of several image processing routines. The approach can be realized in hardware probably resulting in real-time performance.

Apart from the performance further work to improve the robustness of the approach has to be done. As some objects are not suitable for tracking with the developed approach because only the edge information is not sufficient (compare Section 6.1.3) as far as other information is available it has to be used for tracking – probably some additional cues such as combining edge and texture information as proposed in [41]. The ongoing and open question is how to extract more information than edges from single-colored objects.

An important influence on edge images of good quality is a constant and adequate illumination of the scene. One solution might be a light source mounted on the camera spotting towards the scene. This simple setup should result in images whose edge images contain most information. Additionally, the illumination conditions in OpenGL can be adjusted to decrease the difference of the real world and the rendered models of the objects.

Further improvement can be reached with the use of a stereo camera system to increase the accuracy of the depth information. Also, a motion model like the one mentioned in Section 2.3 should be investigated. The developed approach is well prepared to implement this step.

Appendix A

Mathematics

In this section some mathematical theory used in this diploma thesis is presented. As there is no need for complex numbers in this context only real-valued matrices and vectors are assumed. First, the definition of the rotation matrices with a given axis and angle is explained.

Rotation matrix given an axis and an angle

A vector $\vec{v} = (x, y, z)^\intercal \in \mathbb{R}^3$ can be rotated with a rotation matrix $R \in \mathrm{SO}(3)$, resulting in a new vector $\vec{v}' = R \cdot \vec{v}$.
According to Euler's theorem [42] any rigid object can be arbitrarily rotated in 3D space with one single rotation around a fixed axis (see Figure A.1). If there is an axis $\vec{a}$ as well as an angle α the rotation matrix R can be determined as follows:

1. Normalize the axis: $(u, v, w)^\intercal := \frac{\vec{a}}{|\vec{a}|}$
2. $$R = \begin{pmatrix} tu^2 + c & tuv - sw & tuw + sv \\ tuv + sw & tv^2 + c & tvw - su \\ tuw - sv & tvw + su & tw^2 + c \end{pmatrix} \tag{A.1}$$

   with $s := \sin\alpha$, $c := \cos\alpha$ and $t := 1 - c$.

Remark:
The same rotation can be performed with three consecutive rotations using Euler angles (see Figure A.2).

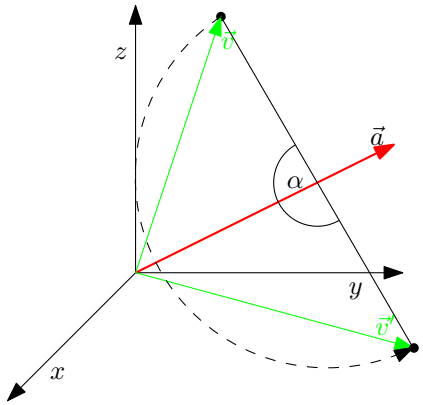


Figure A.1: Rotation around a given axis. *Example:* Vector $\vec{v} = \begin{pmatrix} 0.1 \\ 0.4 \\ 1.1 \end{pmatrix}$ is rotated around the axis $\vec{a} = \begin{pmatrix} 0.143 \\ 0.858 \\ 0.493 \end{pmatrix}$ with angle $\alpha = 180°$. The result of this rotation is $\vec{v}' = \begin{pmatrix} 0.157 \\ 1.144 \\ -0.212 \end{pmatrix}$. The rotation matrix R calculated with Equation A.1 results in $R = \begin{pmatrix} -0.959 & 0.245 & 0.141 \\ 0.245 & 0.472 & 0.846 \\ 0.141 & 0.846 & -0.514 \end{pmatrix}$.

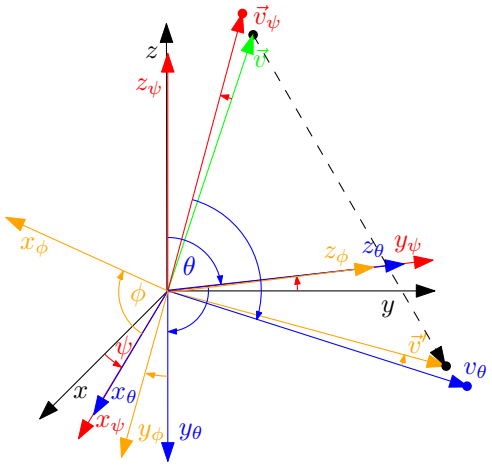


Figure A.2: Rotation using Euler angles. *Example:* The same displacement as in Figure A.1 except with Euler angles. Vector $\vec{v} = (0.1, 0.4, 1.1)^\intercal$ is rotated around the z axis with angle $\psi = 10°$. Then with $\theta = -90°$ around the x_ψ axis. Finally, the result is $\vec{v}'$ after rotating with $\phi = -45°$ around the z_θ axis. $\vec{v}'$ is the same as in Figure A.1, of course.

Sobel and Prewitt operator

The filter matrices for the Sobel operator are

$$Sobel_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \text{ and } Sobel_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

The prewitt operator can be used with the filter matrices

$$Prewitt_x = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} \text{ and } Prewitt_y = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}.$$

Appendix B

Structure of Parameter Files

The parameter files of the 3D objects are of the type CFloatMatrix, see the IVT[1]. They contain the necessary information that is required to render the objects with OpenGL.

Object model, File `object.dat`

The file is organized in lines of 18 values of the type float. Each line represents one triangle. Elements zero to five, six to eleven and twelve to 18 represent one edge of the triangle. The first three elements are the normals of the point defined in the following three elements:

Example:

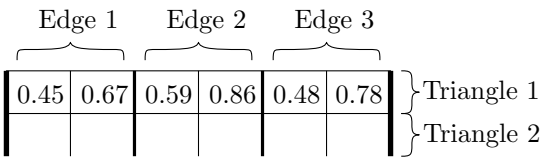
Normal 1			Vertex 1			Normal 2			Vertex 2			Normal 3			Vertex 3			
0.45	0.67	0.59	17.1	34.5	456.7	0.45	0.67	0.59	17.1	−4.5	456.7	0.45	0.67	0.59	57.1	34.5	456.7	Triangle 1
																		Triangle 2

Texture of object model, File `objectTex.dat`

If the object is not single-colored some color information – the texture – has to be mapped onto the 3D object model. The texture is stored as an 24*bit* bitmap image in `object.bmp`. The file `objectTex.dat` contains the mapping information for the texture and is of type CFloatMatrix. It is organized in lines of six values. The first two, second two and as well the third two represent one edge of a 2D triangle of the texture file. The number of triangles of the object model must be the same as the number of triangles of the texture of the object model.

Example:

[1] `http://ivt.sourceforge.net`


Edge 1
Edge 2
Edge 3
0.45
0.67
0.59
0.86
0.48
0.78
Triangle 1
Triangle 2

Appendix C

Source Code

Sources of the cores of the developed approach are listed in the following. On page 67 and 69 the header and the source file of the general tracking approach are printed. The particle filter with its details is shown on page 71. The printed source code together with the UML diagram in Section 5.2.4 allows to get a profound understanding of the implementation.

```cpp
// ********************************************************************************
// This file is part of the Integrating Vision Toolkit (IVT).
//
// The IVT is maintained by the Karlsruhe Institute of Technology (KIT)
// (www.kit.edu) in cooperation with the company Keyetech (www.keyetech.de).
//
// Copyright (C) 2009 Karlsruhe Institute of Technology (KIT).
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
// 1. Redistributions of source code must retain the above copyright
//    notice, this list of conditions and the following disclaimer.
//
// 2. Redistributions in binary form must reproduce the above copyright
//    notice, this list of conditions and the following disclaimer in the
//    documentation and/or other materials provided with the distribution.
//
// 3. Neither the name of the KIT nor the names of its contributors may be
//    used to endorse or promote products derived from this software
//    without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE KIT AND CONTRIBUTORS "AS IS" AND ANY
// EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
// WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
// DISCLAIMED. IN NO EVENT SHALL THE KIT OR CONTRIBUTORS BE LIABLE FOR ANY
// DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
// (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
// LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
// ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
// THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
// ********************************************************************************
// ********************************************************************************
// Filename:  ParticleFilterUniversalTracker.h
// Author:    David
// Date:      30.04.2010
// ********************************************************************************


#ifndef _PARTICLE_FILTER_UNIVERSAL_TRACKER_H_
#define _PARTICLE_FILTER_UNIVERSAL_TRACKER_H_

// ********************************************************************************
// Necessary includes
// ********************************************************************************

#include "Interfaces/RigidObjectTrackingInterface.h"
#include "ParticleFilter6D.h"

// ********************************************************************************
// Forward declarations
// ********************************************************************************

class CFloatMatrix;
class COpenGLVisualizer;
class CUndistortion;
struct Transformation3d;

// ********************************************************************************
// CParticleFilterUniversalTracker
// ********************************************************************************


class CParticleFilterUniversalTracker: public CRigidObjectTrackingInterface
{
public:

    enum EdgeDetectionMode{
        Sobel,
        Prewitt,
        Canny
    };

    //constructor
    CParticleFilterUniversalTracker(CFloatMatrix *pObject, CFloatMatrix *
    pObjectTexture, CByteImage *pImageObjectTexture, int nParticles, float *fSigma,
    bool bTexture, float *fConfiguration, CParticleFilterUniversalTracker::
    EdgeDetectionMode typeOfEdgeDetection = CParticleFilterUniversalTracker::Sobel,
    int nLowThreshold = 15, int nHighThreshold = 80, int nThreshold = 200);

    //destructor
    ~CParticleFilterUniversalTracker();

    //public methods
    void Init(const CCalibration *pCalibration);
    bool Init(const char* sCalibrationFile);
    bool Track(const CByteImage *pEdgeImage, int nLayers, float *fResultConfiguration)
    ;
    bool Track(const CByteImage *pEdgeImage, Vec3d *pOutlinePoints, int nOutlinePoints
    , Mat3d &rotation, Vec3d &translation);
    void GetDrawnObjectWithTexture(float *fConfiguration, CByteImage *pResultImage);


private:
    // private attributes
    CFloatMatrix *m_pObject;
    CFloatMatrix *m_pObjectTexture;
    CByteImage *m_pImageObjectTexture;

    CByteImage *m_pEdgeImageGrey;

    COpenGLVisualizer *m_pVisualizer;

    CUndistortion *m_pUndistortion;

    int m_nLowThreshold;
    int m_nHighThreshold;
    int m_nThreshold;
    int m_nLayers;

    EdgeDetectionMode m_typeOfEdgeDetection;

    int m_width;
    int m_height;
    CParticleFilter6D *m_pParticleFilter6D;

    int m_nParticles;

    float *m_fSigma;

    bool m_bTexture;

    float *m_fConfiguration;
};


#endif /* _PARTICLE_FILTER_UNIVERSAL_TRACKER_H_ */

```

```cpp
// ********************************************************************************
// This file is part of the Integrating Vision Toolkit (IVT).
//
// The IVT is maintained by the Karlsruhe Institute of Technology (KIT)
// (www.kit.edu) in cooperation with the company Keyetech (www.keyetech.de).
//
// Copyright (C) 2009 Karlsruhe Institute of Technology (KIT).
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
// 1. Redistributions of source code must retain the above copyright
//    notice, this list of conditions and the following disclaimer.
//
// 2. Redistributions in binary form must reproduce the above copyright
//    notice, this list of conditions and the following disclaimer in the
//    documentation and/or other materials provided with the distribution.
//
// 3. Neither the name of the KIT nor the names of its contributors may be
//    used to endorse or promote products derived from this software
//    without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE KIT AND CONTRIBUTORS “AS IS” AND ANY
// EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
// WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
// DISCLAIMED. IN NO EVENT SHALL THE KIT OR CONTRIBUTORS BE LIABLE FOR ANY
// DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
// (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
// LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
// ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
// THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
// ********************************************************************************
// ********************************************************************************
// Filename:  ParticleFilterUniversalTracker.cpp
// Author:    David
// Date:      30.04.2010
// ********************************************************************************


// ********************************************************************************
// Includes
// ********************************************************************************

#include "ParticleFilterUniversalTracker.h"
#include "ParticleFilter6D.h"
#include "Calibration/Calibration.h"
#include "Visualizer/OpenGLVisualizer.h"
#include "Calibration/Undistortion.h"
#include "Image/ByteImage.h"
#include "Image/ImageProcessor.h"
#include <math.h>
#include "Helpers/helpers.h"



// ********************************************************************************
// Constructor / Destructor
// ********************************************************************************

CParticleFilterUniversalTracker::CParticleFilterUniversalTracker(CFloatMatrix *pObject
    , CFloatMatrix *pObjectTexture, CByteImage *pImageObjectTexture, int nParticles,
    float *fSigma, bool bTexture, float *fConfiguration, EdgeDetectionMode
    typeOfEdgeDetection, int nLowThreshold, int nHighThreshold, int nThreshold){
    m_pObject = pObject;
    m_pObjectTexture = pObjectTexture;
    m_pImageObjectTexture = pImageObjectTexture;

    m_nLowThreshold = nLowThreshold;
    m_nHighThreshold = nHighThreshold;
    m_nThreshold = nThreshold;
    m_typeOfEdgeDetection = typeOfEdgeDetection;

    m_pVisualizer = new COpenGLVisualizer();
    m_pUndistortion =  new CUndistortion();

    m_nParticles = nParticles;

    m_fSigma = new float[6];
    for(int i=0;i<6;i++) m_fSigma[i] = fSigma[i];

    m_bTexture = bTexture;
    m_fConfiguration = new float [6];
    for(int i=0;i<6;i++) m_fConfiguration[i] = fConfiguration[i];
}

CParticleFilterUniversalTracker::~CParticleFilterUniversalTracker(){
    if (m_pVisualizer) delete  m_pVisualizer;
    if (m_pUndistortion) delete m_pUndistortion;
    if (m_pParticleFilter6D) delete m_pParticleFilter6D;
    if (m_fSigma) delete m_fSigma;
    if (m_fConfiguration) delete m_fConfiguration;
    if (m_pEdgeImageGrey) delete m_pEdgeImageGrey;
}


// ********************************************************************************
// Methods
// ********************************************************************************

void CParticleFilterUniversalTracker::Init(const CCalibration *pCalibration){
    m_width = pCalibration->GetCameraWidth();
    m_height = pCalibration->GetCameraHeight();

    //init the visualizer
    if (!m_pVisualizer->InitByCalibration(pCalibration))
    {
        printf("error: could not init calibration\n");
    }

    m_pUndistortion->Init(pCalibration);

    m_pParticleFilter6D = new CParticleFilter6D(m_nParticles, m_fSigma, m_pVisualizer,
     m_pObject, m_pObjectTexture, m_pImageObjectTexture, m_bTexture, m_fConfiguration,
     m_width, m_height, m_nLowThreshold, m_nHighThreshold, m_nThreshold,
    m_typeOfEdgeDetection);
    m_pParticleFilter6D->InitParticles();

    m_pEdgeImageGrey = new CByteImage(m_width, m_height, CByteImage::eGrayScale);
}

bool CParticleFilterUniversalTracker::Init(const char* sCalibrationFile){
    CCalibration calibration;
    if (!calibration.LoadCameraParameters(sCalibrationFile))
    {
        printf("error: could not load calibration\n");
        return false;
    }
    calibration.SetRotation(Math3d::unit_mat);
    calibration.SetTranslation(Math3d::zero_vec);
    this->Init(&calibration);
    return true;
}

bool CParticleFilterUniversalTracker::Track(const CByteImage *pEdgeImage, int nLayers,
     float *fResultConfiguration){
    ImageProcessor::ConvertImage(pEdgeImage, m_pEdgeImageGrey, true);
    m_pUndistortion->Undistort(m_pEdgeImageGrey, m_pEdgeImageGrey);

    switch(m_typeOfEdgeDetection)
    {
        case Sobel:
            ImageProcessor::CalculateGradientImageSobel(m_pEdgeImageGrey,
    m_pEdgeImageGrey);
            ImageProcessor::ThresholdBinarize(m_pEdgeImageGrey,m_pEdgeImageGrey,
    m_nHighThreshold);
            break;
        case Prewitt:
            ImageProcessor::CalculateGradientImagePrewitt(m_pEdgeImageGrey,
    m_pEdgeImageGrey);
            ImageProcessor::ThresholdBinarize(m_pEdgeImageGrey,m_pEdgeImageGrey,
```

```
    m_nHighThreshold);
143             break;
144         case Canny:
145             ImageProcessor::Canny(m_pEdgeImageGrey, m_pEdgeImageGrey, m_nLowThreshold,
     m_nHighThreshold);
146             break;
147         default:
148             printf("error: typeOfEdgeDetection\n");
149             return false;
150     }
151     m_pParticleFilter6D->SetImage(m_pEdgeImageGrey);
152 
153     double *result_configuration = new double[6];
154 
155     float sigma = 1;
156     for(int i=0;i<nLayers;i++){
157         m_pParticleFilter6D->ParticleFilter(result_configuration,sigma);
158         printf("SIGMA: %f\n", sigma);
159         sigma=1-(float)m_pParticleFilter6D->resultTmp;
160     }
161 
162     for(int i=0;i<6;i++) fResultConfiguration[i]=(float)result_configuration[i];
163     delete result_configuration;
164     return true;
165 }
166 
167 bool CParticleFilterUniversalTracker::Track(const CByteImage *pEdgeImage, Vec3d *
    pOutlinePoints, int nOutlinePoints, Mat3d &rotation, Vec3d &translation){
168     //empty
169     return true;
170 }
171 
172 void CParticleFilterUniversalTracker::GetDrawnObjectWithTexture(float *fConfiguration,
     CByteImage *pResultImage){
173     //Draw the calculated model
174     Transformation3d pose;
175     Math3d::SetVec(pose.translation, fConfiguration[0], fConfiguration[1],
    fConfiguration[2]);
176     float angle = 0;
177     Vec3d axis = {fConfiguration[3], fConfiguration[4], fConfiguration[5]};
178     angle = Math3d::Length(axis);
179     Math3d::NormalizeVec(axis);
180     if (fabsf(angle) < 0.0001f)
181         Math3d::SetMat(pose.rotation, Math3d::unit_mat);
182     else
183         Math3d::SetRotationMatAxis(pose.rotation, axis, angle);
184     m_pVisualizer->Clear();
185     if(m_bTexture)
186     m_pVisualizer->DrawObjectWithTexture(m_pObject, m_pObjectTexture,
    m_pImageObjectTexture, pose);
187     else
188     m_pVisualizer->DrawObject(m_pObject, pose);
189 
190     CByteImage edgeImageColor(640, 480, CByteImage::eRGB24);
191     CByteImage *pEdgeImageColor = &edgeImageColor;
192     m_pVisualizer->GetImage(pEdgeImageColor);
193     ImageProcessor::FlipY(pEdgeImageColor,pEdgeImageColor);
194     ImageProcessor::ConvertImage(pEdgeImageColor, pResultImage);
195 }
196 
```

```cpp
// ********************************************************************************
// This file is part of the Integrating Vision Toolkit (IVT).
//
// The IVT is maintained by the Karlsruhe Institute of Technology (KIT)
// (www.kit.edu) in cooperation with the company Keyetech (www.keyetech.de).
//
// Copyright (C) 2009 Karlsruhe Institute of Technology (KIT).
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
// 1. Redistributions of source code must retain the above copyright
//    notice, this list of conditions and the following disclaimer.
//
// 2. Redistributions in binary form must reproduce the above copyright
//    notice, this list of conditions and the following disclaimer in the
//    documentation and/or other materials provided with the distribution.
//
// 3. Neither the name of the KIT nor the names of its contributors may be
//    used to endorse or promote products derived from this software
//    without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE KIT AND CONTRIBUTORS “AS IS” AND ANY
// EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
// WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
// DISCLAIMED. IN NO EVENT SHALL THE KIT OR CONTRIBUTORS BE LIABLE FOR ANY
// DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
// (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
// LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
// ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
// THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
// ********************************************************************************
// ********************************************************************************
// Filename:  ParticleFilter6D.cpp
// Author:    David
// Date:      30.04.2010
// ********************************************************************************


// ********************************************************************************
// Includes
// ********************************************************************************

#include "ParticleFilter6D.h"
#include "Image/ImageProcessor.h"
#include "Image/ByteImage.h"
#include <math.h>
#include "Helpers/helpers.h"
#include "ParticleFilterUniversalTracker.h"
#include "Math/Constants.h"
#include <float.h>


// ********************************************************************************
// Constructor / Destructor
// ********************************************************************************

CParticleFilter6D::CParticleFilter6D(int nParticles, float *fSigma, COpenGLVisualizer *pVisualizer, CFloatMatrix *pObject, CFloatMatrix *pObjectTexture, CByteImage *pImageObjectTexture, bool bTexture, float *fConfiguration, int width, int height, int nLowThreshold, int nHighThreshold, int nThreshold, int typeOfEdgeDetection): CParticleFilterFramework(nParticles,6){
    m_pVisualizer = pVisualizer;
    m_pObject = pObject;
    m_pObjectTexture = pObjectTexture;
    m_pImageObjectTexture = pImageObjectTexture;
    m_fConfiguration = new float [6];
    for(int i=0;i<6;i++) m_fConfiguration[i] = fConfiguration[i];
    m_nLowThreshold = nLowThreshold;
    m_nHighThreshold = nHighThreshold;
    m_nThreshold = nThreshold;
    m_typeOfEdgeDetection = typeOfEdgeDetection;
    m_fSigma = new float [6];
    for(int i=0;i<6;i++)m_fSigma[i] = fSigma[i];
    m_bTexture = bTexture;
    m_pSegmentedImage = new CByteImage(width, height, CByteImage::eGrayScale);
    m_pSegmentedImage2 = new CByteImage(width, height, CByteImage::eGrayScale);
    m_ppProbabilities[0] = new float[m_nParticles];
    m_ppProbabilities[1] = new float[m_nParticles];
    m_nParticleIndex = 0;
    m_pGetColorObject = new CByteImage(640, 480, CByteImage::eRGB24);
}

CParticleFilter6D::~CParticleFilter6D(){
    if (m_pSegmentedImage) delete m_pSegmentedImage;
    if (m_pSegmentedImage2) delete m_pSegmentedImage2;
    if (m_fConfiguration) delete m_fConfiguration;
    if (m_fSigma) delete m_fSigma;
    if (m_pGetColorObject) delete m_pGetColorObject;
}


// ********************************************************************************
// Methods
// ********************************************************************************

void CParticleFilter6D::SetImage(const CByteImage *pSegmentedImage)
{
    ImageProcessor::CopyImage(pSegmentedImage,m_pSegmentedImage);
    ImageProcessor::Dilate(m_pSegmentedImage,m_pSegmentedImage);
}

void CParticleFilter6D::UpdateModel(int nParticle)
{
    Math3d::SetVec(m_model.translation, (float) s[nParticle][0], (float) s[nParticle][1], (float) s[nParticle][2]);
    float angle = 0;
    Vec3d axis = { (float)s[nParticle][3], (float)s[nParticle][4], (float)s[nParticle][5] };
    angle = Math3d::Length(axis);
    Math3d::NormalizeVec(axis);
    if (fabsf(angle) < 0.001f)
        Math3d::SetMat(m_model.rotation, Math3d::unit_mat);
    else
        Math3d::SetRotationMatAxis(m_model.rotation, axis, angle);
}


void CParticleFilter6D::PredictNewBases(double dSigmaFactor)
{
    //set it for dynamic motion modell
    const float VELOCITY_FACTOR = 0.0f;
    int nNewIndex = 0;

    for (nNewIndex = 0; nNewIndex < m_nParticles; nNewIndex++)
    {
        int nOldIndex = PickBaseSample();

        const double xx = s[nOldIndex][0] + VELOCITY_FACTOR * (mean_configuration[0] - last_configuration[0]) + dSigmaFactor * sigma[0] * gaussian_random();
        const double yy  = s[nOldIndex][1] + VELOCITY_FACTOR * (mean_configuration[1] - last_configuration[1]) + dSigmaFactor * sigma[1] * gaussian_random();
        const double zz  = s[nOldIndex][2] + VELOCITY_FACTOR * (mean_configuration[2] - last_configuration[2]) + dSigmaFactor * sigma[2] * gaussian_random();

        const double alpha  = s[nOldIndex][3] + VELOCITY_FACTOR * (mean_configuration[3] - last_configuration[3]) + dSigmaFactor * sigma[3] * (float) FLOAT_PI / 180 * gaussian_random();
        const double beta    = s[nOldIndex][4] + VELOCITY_FACTOR * (mean_configuration[4] - last_configuration[4]) + dSigmaFactor * sigma[4] * (float) FLOAT_PI / 180 * gaussian_random();
        const double gamma  = s[nOldIndex][5] + VELOCITY_FACTOR * (mean_configuration[5] - last_configuration[5]) + dSigmaFactor * sigma[5] * (float) FLOAT_PI / 180 * gaussian_random();

        s_temp[nNewIndex][0] = xx;
        s_temp[nNewIndex][1] = yy;
        s_temp[nNewIndex][2] = zz;

        s_temp[nNewIndex][3] = alpha;
```

```
        s_temp[nNewIndex][4] = beta;
        s_temp[nNewIndex][5] = gamma;
    }

    // switch old/new
    double **temp = s_temp;
    s_temp = s;
    s = temp;
}

double CParticleFilter6D::CalculateProbability(bool bSeparateCall)
{
    m_pVisualizer->Clear();
    if(m_bTexture)
    m_pVisualizer->DrawObjectWithTexture(m_pObject, m_pObjectTexture,
    m_pImageObjectTexture, m_model);
    else
    m_pVisualizer->DrawObject(m_pObject, m_model);
    m_pVisualizer->GetImage(m_pGetColorObject);

    ImageProcessor::ConvertImage(m_pGetColorObject, m_pSegmentedImage2);
    ImageProcessor::FlipY(m_pSegmentedImage2,m_pSegmentedImage2);

    switch(m_typeOfEdgeDetection)
    {
        case CParticleFilterUniversalTracker::Sobel:
            ImageProcessor::CalculateGradientImageSobel(m_pSegmentedImage2,
    m_pSegmentedImage2);
            ImageProcessor::ThresholdBinarize(m_pSegmentedImage2,m_pSegmentedImage2,
    m_nThreshold);
            break;
        case CParticleFilterUniversalTracker::Prewitt:
            ImageProcessor::CalculateGradientImagePrewitt(m_pSegmentedImage2,
    m_pSegmentedImage2);
            ImageProcessor::ThresholdBinarize(m_pSegmentedImage2,m_pSegmentedImage2,
    m_nThreshold);
            break;
        case CParticleFilterUniversalTracker::Canny:
            ImageProcessor::Canny(m_pSegmentedImage2, m_pSegmentedImage2,
    m_nLowThreshold, m_nHighThreshold);
            ImageProcessor::Dilate(m_pSegmentedImage2, m_pSegmentedImage2);
            break;
        default:
            printf("error: typeOfEdgeDetection\n");
    }
    int count = ImageProcessor::PixelSum(m_pSegmentedImage2)/255;

    ImageProcessor::And(m_pSegmentedImage, m_pSegmentedImage2, m_pSegmentedImage2);

    int matchingcount = ImageProcessor::PixelSum(m_pSegmentedImage2)/255;

    if (bSeparateCall) return expf(-20.0f * (1.0f - (matchingcount / (float)(count))))
    ;

    m_ppProbabilities[0][m_nParticleIndex] = (1.0f - (matchingcount / (float)(count)))
    ;
    //arbitrary weighting function
    m_ppProbabilities[1][m_nParticleIndex] = (float) count*count;

    m_nParticleIndex++;
    return 0;
}


void CParticleFilter6D::InitParticles()
{
    int i;

    // init particle related attributes
    for (i = 0; i < m_nParticles; i++)
    {
        // particle positions
        s[i][0] = m_fConfiguration[0];
        s[i][1] = m_fConfiguration[1];
        s[i][2] = m_fConfiguration[2];
        s[i][3] = m_fConfiguration[3];
        s[i][4] = m_fConfiguration[4];
        s[i][5] = m_fConfiguration[5];

        // probability for each particle
        pi[i] = 1.0 / m_nParticles;
    }

    // initialize configurations
    for (i = 0; i < 6; i++)
    {
        mean_configuration[i] = s[0][i];
        last_configuration[i] = s[0][i];
    }

    c_total = 1.0;

    // maximum offset for next configuration
    sigma[0] = (float) m_fSigma[0];
    sigma[1] = (float) m_fSigma[1];
    sigma[2] = (float) m_fSigma[2];
    sigma[3] = (float) m_fSigma[3];
    sigma[4] = (float) m_fSigma[4];
    sigma[5] = (float) m_fSigma[5];
}


void CParticleFilter6D::CalculateFinalProbabilities()
{
    m_nParticleIndex = 0;

    float minn = FLT_MAX;
    float maxx = FLT_MIN;
    int i;

    for (i = 0; i < 2; i++)
    {
        int j;

        for (j = 0; j < m_nParticles; j++)
        {
            if (m_ppProbabilities[i][j] < minn)
                minn = m_ppProbabilities[i][j];

            if (m_ppProbabilities[i][j] > maxx)
                maxx = m_ppProbabilities[i][j];
        }

        if (maxx != minn)
        {
            const float fFactor = 1.0f / (maxx - minn);

            for (j = 0; j < m_nParticles; j++)
                m_ppProbabilities[i][j] = (m_ppProbabilities[i][j] - minn) * fFactor;
        }
        else
        {
            for (j = 0; j < m_nParticles; j++)
                m_ppProbabilities[i][j] = 1.0f / m_nParticles;
        }
    }

    float result;
    for (i = 0; i < m_nParticles; i++)
    {
        //weight the different parameters
        result = expf(-20.0f * (1 * m_ppProbabilities[0][i] + 0 * m_ppProbabilities[1]
    [i]));
        pi[i] = result;
    }
}
```

Appendix D

Datasheets

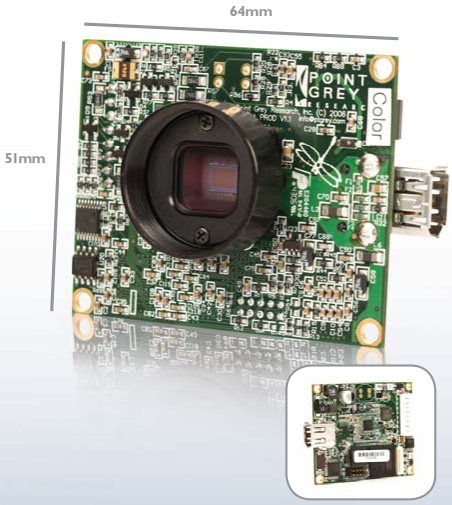
The following datasheets[1] illustrate the technical details of the Dragonfly® 2 cameras developed by Point Grey Research Inc. Throughout this work the DR2-COL-CSBOX version of these cameras has been used with a $6mm$ fixed-focus lens for all image sequences taken.

[1] `http://www.ptgrey.com/products/dragonfly2/dragonfly2.pdf`

Dragonfly®2

FLEXIBLE + FULL-FEATURED

POINT GREY



- **IEEE-1394a (FireWire) digital camera**
- **On-board color processing**
- **1/3" Sony® CCDs, BW or Color**
- **Enclosed or remote head options available***

The Dragonfly®2 is a flexible, full-featured IEEE-1394a (FireWire) camera designed for imaging product development.

Models	Lense Specification
DR2-BW/COL-XX	Sony 1/3" CCD, BW / Color, 648x488 at 60 FPS
DR2-HIBW/HICOL-XX	Sony 1/3" CCD, BW / Color, 1032x776 at 30 FPS
DR2-13S2M/C-CS	Sony 1/3" CCD, BW / Color, 1296x964 at 20 FPS
DR2-03S3M/C-EX-CS	Sony 1/3" CCD, BW / Color, 648x488 at 60 FPS
DR2-08S3M/C-EX-CS	Sony 1/3" CCD, BW / Color, 1032x766 at 30 FPS

Triggering and GPIO

The Dragonfly2 has an 8-pin GPIO connector located on the back of the camera and case. Inputs can be configured to accept an external trigger signal. Outputs can be configured to send an output signal, strobe or PWM signal and can drive most TTL devices at approximately 10mA. The Dragonfly2 has a logic level serial port with a bandwidth capacity of up to 115.2 Kbps.

Region of Interest (ROI) & Pixel Binning

The Dragonfly2 supports Format_7 custom image modes such as pixel binning and region of interest (ROI) to achieve faster frame rates and higher sensitivity. (Example below uses a DR2-BW)

Mode	Resolution	FPS	Description
0	648x488	60	Region of Interest (ROI)
1	320x240	100	2x2 pixel binning
2	640x240	60	1x2 pixel binning

Software

The FlyCapture® software development kit (SDK) is included with all Point Grey imaging products. The SDK is compatible with Microsoft Windows and includes a camera device driver, full software library with Application Programming Interface (API), demo programs and C/C++ example source code. The Dragonfly2 is also compatible with many third-party software packages from vendors such as National Instruments, Cognex, MVTec, A&B Software, Matrox, Mathworks, and Norpix.

Updatable Firmware

The field-programmable gate array (FPGA) chip controls all camera functionality, including exposure, resolution and frame rate, pixel binning, user memory channels and more. It can also be updated with new functionality in the field.

Automatic Synchronization

Multiple Dragonfly2 cameras networked on the same IEEE-1394 bus are automatically synchronized to each other. The maximum deviation between cameras is 125µs.

Color Processing

The color Dragonfly2 features on-camera color processing and auto white balance. Available outputs include YUV411, YUV422 and RGB. If a reduction in the bus bandwidth is required, users can access the raw Bayer pattern.

Auto Iris

In addition to auto-gain and auto-shutter/exposure controls, the Dragonfly2 has a DC auto-iris output. Using standard CCTV auto-iris lenses, users can physically control the amount of light that falls onto the CCD. This feature is particularly important in outdoor applications where the amount of light can vary greatly.

Gamma and Programmable LUT

The digitization of the images on the camera is achieved using a 12-bit analog to digital converter. Users can choose either an 8-bit or 16-bit output from the camera. Gamma can be applied to12-bit data when 8-bit output is used. Lookup table support (LUT) is also available for custom mapping of image values.

On-Board Memory Channels

The Dragonfly2 has the ability to save and restore camera settings and imaging parameters via on-board memory channels. This is useful for saving default power-up settings, such as gain, shutter, video format and frame rate, etc., that are different from the factory defaults.

*not available for DR2-13S2M/C-CS models

North America T +604.242.9937 E sales@ptgrey.com Europe T + 49 7141 488817-0 E eu-sales@ptgrey.com www.ptgrey.com

Dragonfly®2 Specifications

Specification	BW/COL/03S2	HIBW/HICOL/08S2	I3S2
Image Sensor Type	Sony® 1/3" progressive scan CCDs		
Image Sensor Model	ICX424	ICX204	ICX445
Sensor Pixel Size	7.4µm square pixels	4.65µm square pixels	3.75µm square pixels
Maximum Resolution	648x488	1032x776	1296x964
Maximum Frame Rate	648x488 at 60 FPS	1032x776 at 30 FPS	1296x964 at 20 FPS
Lens Mount	C/CS-mount, M12 microlens		C/CS-mount
A/D Converter	Analog Devices 12-bit analog-to-digital converter		
Video Data Output	8, 16 and 24-bit digital data		
Partial Image Modes	Pixel binning and region of interest modes via Format_7		
Interfaces	6-pin IEEE-1394 for camera control and video data transmission 8 general purpose digital input/output (GPIO) pins		
Power Requirements	8-30V, max 2W at 12V		max 2.2W at 12V
Gain	Automatic/Manual/One-Push Gain modes 0dB to 24dB		
Shutter	Automatic/Manual/One-Push/Extended Shutter modes 0.01ms to 66.63ms at 15 FPS, greater than 5s in extended mode		
Gamma	0.50 to 4.00		
Trigger Modes	DCAM v1.31 Modes 0, 1, 3, 4, 5 and 14		Modes 0, 1, 3, 14
Signal To Noise Ratio	Greater than 60dB at 0dB gain		
Dimensions	64mm x 51mm (bare board w/o case or lens holder)		
Mass	45 grams (bare board w/ lens holder and C-mount adapter)		
Camera Specification	IIDC 1394-based Digital Camera Specification v1.31		
Emissions Compliance	Complies with CE rules and Part 15 Class A of FCC Rules		
Operating Temp.	Commercial grade electronics rated from 0° to 45°C		
Storage Temperature	-30° to 60°C		
Remote Head Option	Available with 6-inch shielded ribbon cable		Not available
Case Enclosed Option	Available (except with remote head option)		Not available

Development Kit (DR2-DEVKIT) Includes:

- CS-mount lens with variable focus and auto iris
- GPIO connector for quick and easy external wiring
- 4.5 meter, 6-pin to 6-pin, IEEE-1394 cable w/ferrites
- IEEE-1394 OHCI PCI Host Adapter 3 port-400Mbps card
- FlyCapture® SDK (C/C++ API and device drivers) CD

OEM Kit (DR2-xxxx-OEMKIT) Includes:

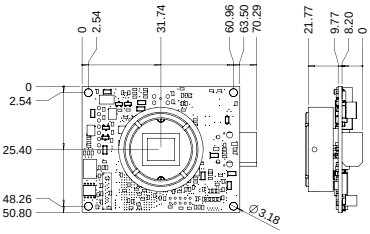
- M12 microlens with 6mm focal length, M12 lens holder†
- Tripod mounting bracket
- Wiring harness for the GPIO connector
- 4.5 meter, 6-pin to 6-pin, IEEE-1394 cable w/ferrites
- IEEE-1394 OHCI PCI Host Adapter 3 port-400Mbps card
- FlyCapture® SDK (C/C++ API and device drivers) CD

Recommended System Configuration:

- Windows® XP Service Pack 1
- 512MB of RAM
- Intel® Pentium 4 2.0GHZ or compatible processor
- AGP video card with 128MB video memory
- 32-bit PCI slot for IEEE-1394 PCI card
- Microsoft® Visual C++ 6.0 (to compile and run example code)

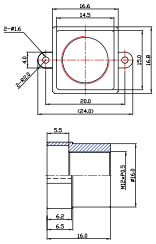
† Not compatible with DR2-13S2M/C-CS camera models

Dimensional Drawings - DR2-xxxx-CS

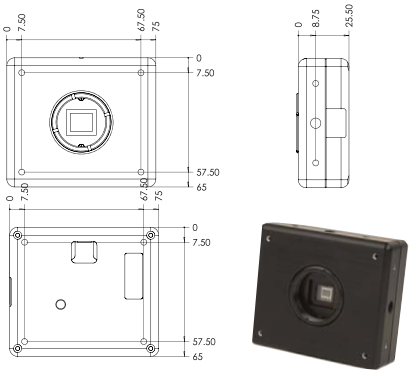


M12 Lens Mount Drawing

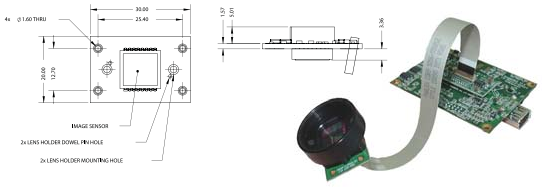
This mount is used for attaching an M12 microlens to the camera board.



Dimensional Drawings - DR2-xxxx-CSBOX



Dimensional Drawings - DR2-xxxx-EX-CS



FireWire

Sept 2009

North America T +604.242.9937 E sales@ptgrey.com Europe T + 49 7141 488817-0 E eu-sales@ptgrey.com www.ptgrey.com

Bibliography

[1] T. Gockel, "Latex-Template für Seminar-, Studien- und Diplomarbeiten und Dissertationen (V0.99)," 2007. [Online]. Available: http://wwwiaim.ira.uka.de/form-der-wissenschaftlichen-ausarbeitung

[2] D. G. Lowe, "Three-dimensional object recognition from single two-dimensional images," *Artif. Intell.*, vol. 31, no. 3, pp. 355–395, 1987.

[3] ——, "Robust model-based motion tracking through the integration of search and estimation," *Int. J. Comput. Vision*, vol. 8, no. 2, pp. 113–122, 1992.

[4] C. Harris and C. Stennett, "Rapid - a video rate object tracker," in *1st British Machine Vision Conference*, 1990.

[5] C. Harris, "Tracking with rigid models," in *Active Vision*, A. Blake, Ed. MIT Press, 1992, ch. 4, pp. 59–73.

[6] M. Armstrong and A. Zisserman, "Robust object tracking," in *Proceedings of the Asian Conference on Computer Vision*, vol. I, 1995, pp. 58–61. [Online]. Available: http://www.robots.ox.ac.uk/~vgg

[7] Éric Marchand, P. Bouthemy, and F. Chaumette, "A 2d-3d model-based approach to real-time visual tracking," *Image and Vision Computing*, vol. 19, no. 13, pp. 941 – 955, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/B6V09-44B23S3-B/2/0733bb46a9406e2768273a5611828a85

[8] M. Pupilli and A. Calway, "Real-time camera tracking using a particle filter," in *In Proc. British Machine Vision Conference*, 2005, pp. 519–528.

[9] ——, "Real-time camera tracking using known 3d models and a particle filter," in *ICPR '06: Proceedings of the 18th International Conference on Pattern Recognition.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 199–203.

[10] G. Klein and D. Murray, "Full-3d edge tracking with a particle filter," in *Proc. British Machine Vision Conference (BMVC'06)*, vol. 3. Edinburgh: BMVA, September 2006, pp. 1119–1128.

[11] T. Asfour, K. Regenstein, P. Azad, J. Schröder, and R. Dillmann, "Armar-iii: A humanoid platform for perception-action integration," in *2nd International Workshop on Human-Centered Robotic Systems (HCRS)*, 2006.

[12] V. Lepetit and P. Fua, "Monocular model-based 3d tracking of rigid objects," *Found. Trends. Comput. Graph. Vis.*, vol. 1, no. 1, pp. 1–89, 2005.

[13] P. Azad, *Visual Perception for Manipulation and Imitation in Humanoid Robots*. Berlin Heidelberg: Springer, 2008, vol. 4.

[14] V. Kyrki and D. Kragic, "Tracking rigid objects using integration of model-based and model-free cues," *Machine Vision and Applications*, 2009. [Online]. Available: http://www.springerlink.com/content/b623117p8566k503/

[15] C. Harris and M. Stephens, "A combined corner and edge detection," in *Proceedings of The Fourth Alvey Vision Conference*, 1988, pp. 147–151. [Online]. Available: http://www.bmva.org/bmvc/1988/avc-88-023.pdf

[16] P. Bouthemy, "A maximum likelihood framework for determining moving edges," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 11, no. 5, pp. 499–511, 1989.

[17] J. M. Odobez and P. Bouthemy, "Robust multiresolution estimation of parametric motion models," *Jal of Vis. Comm. and Image Representation*, 1995.

[18] D. F. Dementhon and L. S. Davis, "Model-based object pose in 25 lines of code," *Int. J. Comput. Vision*, vol. 15, no. 1-2, pp. 123–141, 1995.

[19] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME Journal of Basic Engineering*, no. 82 (Series D), pp. 35–45, 1960. [Online]. Available: http://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf

[20] O. Mateo Lozano and K. Otsuka, "Real-time visual tracker by stream processing," *J. Signal Process. Syst.*, vol. 57, no. 2, pp. 285–295, 2009.

[21] P. A. Viola and M. J. Jones, "Rapid object detection using a boosted cascade of simple features." in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*. IEEE Computer Society, 2001, pp. 511–518.

[22] P. Viola and M. J. Jones, "Robust real-time face detection," *Int. J. Comput. Vision*, vol. 57, no. 2, pp. 137–154, 2004.

[23] P. Azad, T. Asfour, and R. Dillmann, "Combining appearance-based and model-based methods for real-time object recognition and 6d localization," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006, pp. 5339–5344.

[24] ——, "Accurate shape-based 6-dof pose estimation of single-colored objects," in *Proceedings of 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, St. Louis, USA, 2009.

[25] P. Azad, T. Gockel, and R. Dillmann, *Computer Vision – Das Praxisbuch*. Aachen: Elektor-Verlag, 2007. [Online]. Available: http://wwwiaim.ira.uka.de/computer-vision

[26] N. Gordon, D. Salmond, and A. Smith, "Novel approach to nonlinear/non-gaussian bayesian state estimation," *Radar and Signal Processing, IEE Proceedings F*, vol. 140, no. 2, pp. 107 –113, apr 1993.

[27] G. Kitagawa, "Monte carlo filter and smoother for non-gaussian nonlinear state space models," *Journal of Computational and Graphical Statistics*, vol. 5, no. 1, pp. 1–25, 1996. [Online]. Available: http://www.jstor.org/stable/1390750

[28] A. Blake and M. Isard, "The condensation algorithm - conditional density propagation and applications to visual tracking," in *Advances in Neural Information Processing Systems*. The MIT Press, 1996, pp. 36–1.

[29] J. Deutscher, A. Blake, and I. Reid, "Articulated body motion capture by annealed particle filtering," in *IEEE Conference on Computer Vision and Pattern Recognition.*, vol. 2, 2000, pp. 126 –133 vol.2.

[30] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[31] M. Segal and K. Akeley, "The opengl graphics system: A specification," Mar. 2010. [Online]. Available: http://www.opengl.org/registry/doc/glspec33.core.20100311.pdf

[32] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 6, pp. 679–698, 1986.

[33] *Parallel Programming*, vol. 1, 1958.

[34] G. Wilson, 1994. [Online]. Available: http://ei.cs.vt.edu/~history/Parallel.html

[35] M. J. Atallah, *Algorithms and theory of computation handbook*. CRC Press, 1999.

[36] D. B. Kirk, *Programming massively parallel processors : a hands-on approach*, W.-m. W. Hwu, Ed. Amsterdam: Elsevier/Morgan Kaufmann, 2010.

[37] E. Gamma, *Design patterns : elements of reusable object-oriented software*, 28th ed., ser. Addison-Wesley professional computing series. Boston: Addison-Wesley, 2004.

[38] R. Becher, P. Steinhaus, and R. Dillmann, "Interactive object modelling for a humanoid service robot." in *Proceedings of the Conference on Humanoids.*, 2003.

[39] R. Becher, P. Steinhaus, R. Zöllner, and R. Dillmann, "Design and implementation of an interactive object modelling system," in *Proc. Robotik/IRS*, 2006.

[40] A. Kasper, R. Becher, P. Steinhaus, and R. Dillmann, "Developing and analyzing intuitive modes for interactive object modeling," in *International Conference on Multimodal Interfaces*, 2007.

[41] L. Vacchetti, V. Lepetit, and P. Fua, "Combining edge and texture information for real-time accurate 3d camera tracking," in *ISMAR '04: Proceedings of the 3rd IEEE/ACM International Symposium on Mixed and Augmented Reality*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 48–57.

[42] L. Euler, *Novi Commentarii academiae scientiarum Petropolitanae 20*. n.n., 1776.

Index