Motion Analysis

Model Based Head Pose Estimation of Infants

Roald Fernandez Cuesta

Master of Science in Engineering Cybernetics
Submission date: July 2010
Supervisor: Øyvind Stavdahl, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Analysis of infant movements has proved to be relevant for early diagnosis of disorders such as cerebral palsy. There are indications that head movements are particularly relevant for this kind of analysis. There are approximately 100 recordings available of infants' fidgety movements. In a previous student project, relevant methods for head pose estimation have been studied.

1. Identify methods for combining physics based constraints (equations of motion, kinematics etc.) with video based head pose estimation in order to improve and robustify the overall head pose tracking. The work can be based on methods from the literature and/or methods developed for this particular project by the student.

2. Implement a suitable subset of the methods studied in the previous student project. The implementation should be tailored to the task of estimating head pose of infants based on the available video footage, and preferably be able to perform in real time.

3. Integrate results from point 1 with those of point 2 above.

4. Assess the properties of the resulting system by comparison with available kinematic sensor data, graphical overlaying of tracking data on the video image and/or similar. The assessment should preferably include quantitative measures.

Assignment given: 25. January 2010
Supervisor: Øyvind Stavdahl, ITK

Abstract

This thesis presents a method for performing tracking and estimation of head position and orientation by means of template based particle filtering. The implementation is designed to withstand high levels of occlusion and noise, and allow for system dynamics to be accounted for. To accelerate the computation, GPGPU techniques are used to enable the GPU to function a co-processor, resulting in real-time performance. A method is devised to allow for dynamic creation of feature points used in the particle filter. Furthermore, the graphics pipeline is used to overlay and visualize the tracking, as well as play a key role in the dynamic template functionality. Finally, a benchmarking system is suggested and developed for carrying out controlled evaluation of tracking methods in general.

Preface

This thesis is written as part of my degree in Engineering Cybernetics at the Norwegian University for Science and Technology (NTNU), Trondheim. It follows as a continuation to the term project, [1], submitted in December 2009, which was mainly a literature study on face pose estimation methods in general terms.

The thesis focuses on a particular method for performing head pose estimation, and its implementation. In a broader sense, it is related to a project on on automatic detection of Cerebral Palsy (CP). In which, and briefly told, the goal is to create an automatic system capable of assessing the likelihood of an infant suffering from CP, based on the movement of the infant. Extracting this movement data from video has been the focus of several theses. The main distinction to these theses, is that tracking has been done in the image plane, while here it is done in 6 dimensions. In space (x,y,z) as well as orientation.

I've thoroughly enjoyed working on this project, as it has been very open, allowing me to approach the problem according to my own interests. I would like to extend my sincere gratitude to my advisor Øyvind Stavdahl (A/prof., NTNU) for the enjoyable meetings, encouraging my ideas, and being a source of good advice and motivation.

Trondheim, July 2010
Roald Fernández Cuesta

Contents

Nomenclature

CP	Cerebral Palsy
PCA	Principal Component Analysis
CIMA	Computer-assisted Infant Movement Analysis
OpenCL	Open Computing Language
OpenCV	Open Computer Vision Library
CUDA	Compute Unified Device Architecture
GPGPU	General-Purpose Computation on Graphics Processing Units
GPU Computing	Same as GPGPU
OpenGL	Open Graphics Library
GLSL	OpenGL Shading Language
pose	Collective term for position and orientation
AAM	Active Appearance Model
RGB	Additive color space of red, green and blue
GPU	Graphics Processing Unit
CPU	Central Processing Unit
OS	Operating System
API	Application program interface
MTM	Model Transform Matrix
VTM	View Transform Matrix
World Coordinates	Position in three dimensions $[x^W, y^W, z^W]$
Image Coordinates	Position in the image plane $[p_x^W, p_y^W]$ or $[q_x^W, q_y^W]$
$\boldsymbol{\tau}$	Feature Point
T	Template
$\boldsymbol{\xi}$	Particle
Ξ	Particle Cloud
p.d.f.	Probability distribution function

Chapter 1

Introduction

1.1 CIMA Project and Background

In the mid 1990s, Lars Adde of the Physiotherapy Section in St.Olav's Hospital in Trondheim started a project where the object was to measure movement in infants in order to estimate the likelihood of developing the neuromuscular disease Cerebral Palsy (CP). This project (henceforth referred to as the *CIMA–project*, short for Computer-assisted Infant Movement Assessment) has been an ongoing collaboration between St. Olav's Hospital and NTNU/SINTEF with several master theses being written on the subject. The main effort has been to robustly track movement of the infants' limbs, as well as machine learning methods for automatic classification. For more information on the project and the work done on tracking see, [2, 3, 4].

The setup used for recording the infants is detailed in [3, 4]. In short, the setup consists of a monocular camera, and six colored passive[1] markers on the infants' ankles, wrists, chest and forehead. Earlier recordings also subsumed additional sensors in these locations, directly measuring position $[x, y, z]$ and rotation $[\phi, \theta, \psi]$. These direct sensors were, mainly due to their weight and wiring, deemed obtrusive to the natural movements of the infants – motivating the research for non-intrusive measurement systems based on video recordings.

[1]Passive markers do not emit any lighting, as opposed to active markers such as LED lights.

1.2 Focus of Thesis

Prior work focused on image space tracking of the passive colored markers attached to the infants. It is a reasonable assumption that added dimensionality to the measurements has the potential of improving the accuracy of correctly classifying the presence of CP. More particularly, going from the $[x, y]$ coordinate in the image space, to the $[x, y, z]$-position in world space, including orientation $[\phi, \theta, \psi]$. Position and orientation will throughout this report, collectively be referred to as the *pose*. The rotation angles, $[\phi, \theta, \psi]$, are also termed *roll, pitch* and *yaw* respectively, as depicted in figure 1.1.

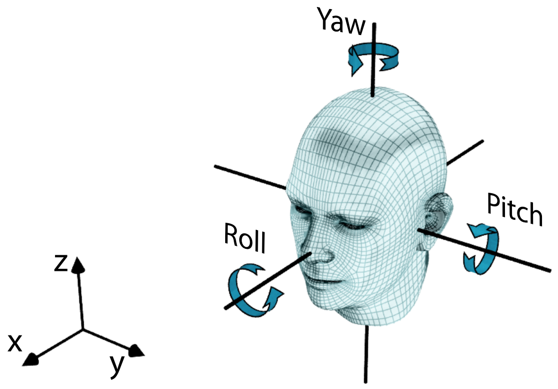

Figure 1.1: Head pose. [3d head model from sharecg.com in *human head pack*]

The main interest of this project is the estimation and tracking of head-position as well as its orientation. The head has several facilitating properties with respect to tracking: its rich distinctive features, lack of rapid movement, bounded movement and fairly rigid form. Section 1.5 gives additional motivation for head tracking. The desired characteristics of the tracking system is robustness, non-obtrusiveness and real-time execution. A more comprehensive list is given in the next page.
The last decade has seen remarkable advances in computation power. This has cleared the way for using methods deemed too computationally expensive in the past. One such method is particle filtering, which searches through a state space by simulating and evaluating a large number of states.

1.3 Desired Characteristics of Algorithm

Given the application for which the head-pose estimation algorithm is to be used, some desired characteristics are:

Autonomous
: Requires little or no user input, and if so, only for initialization

Marker-less
: Does not rely on active/passive markers for tracking. Completely non-intrusive.

Robustness
: Robust, in the sense that it does not lose track of the object.

Monocular
: Functional using a single camera as sensor.

Real-time
: Fast enough for real-time applications, using a regular consumer laptop.

Invariance to Image-Changing factors
: Perform reasonably well in spite of factors such as lens camera distortion, changing lighting conditions, difference in appearance of subject (including facial expressions)

Continuous Measurements
: Able to discern orientation and position with fine granularity, as opposed to a set of discrete orientations.

Noise Resistant
: Perform well under reasonable amounts of noise.

Occlusion Resistant
: As the head might be partially occluded by arms and feet (infants are very flexible), the algorithm should demonstrate some resistance to occlusion

1.4 CIMA Video Data

The videos from the CIMA-project containing the infant subjects are protected by patient confidentiality concerns, required by the Norwegian Data Inspectorate. Frames from these videos used in this report will have the eyes redacted to hide the identity of the subject. This thesis is concerned with head pose estimation, and although it is possible, redation becomes problematic for giving demonstrations. For this reason, an image of an infant not part of the CIMA project was used for illustrative purposes. The demo and test videos were created with this non-protected image, such that they can be accompanied in the CD..

1.5 Motivation

1.5.1 Biological & Anthropological

Humans have an innate ability to rapidly and effortlessly assert the orientation of a human head. This allows for inference of social intentions and comprehension of nonverbal communication. Studies have shown that by 6 months of age, infants begin to take notice of the gaze direction when it is signaled by both eyes and head turning [5].

Head orientation conveys much information relevant to social situations. It is used to indicate the intended target of a conversation, simply by the direction. During the conversation, head gestures can intentionally be used to indicate understanding (nods) or disaccord (shakes). Somewhat exaggerated head movements can also be used as a pointing mechanism (e.g. as seen in movies where the protagonist is in a predicament disallowing verbal communication, and vigorously jerks his head repeatedly to point the direction of impending danger, etc.). Similarly, inadvertent rapid head gestures, can communicate surprise or alarm, and will instinctively trigger reflexive responses in nearby onlookers.

Gaze estimation, that is, determining the percieved direction in which a person sees, is intrinsically linked to the head pose. Head pose estimation is intrinsically linked with gaze estimation, by which is meant the perceived direction in which a person sees. In itself, the head pose serves as a coarse indication (due to limited field of vision, and normalcy of looking straight ahead). In some cases head pose is the only indication. For instance; images with low resolution, occluded eyes (shades, welding helmets, superhero masks), rear view of head, etc. When two people focus their visual attention on each other, there is an immediate understanding of mutual awareness. Typically, pedestrians do this with approaching drivers before stepping onto a crosswalk.

Even when the eyes are clearly visible, the head orientation still plays an important role in correctly estimating gaze direction. According to [6], perceived gaze direction is surmised from a combination of both head pose and eye direction (more accurately, visible sclera, the white part of the eye). An example is given in the figure below, and it is suggested that the reader attempt to discern the gaze direction in each image before reading on.

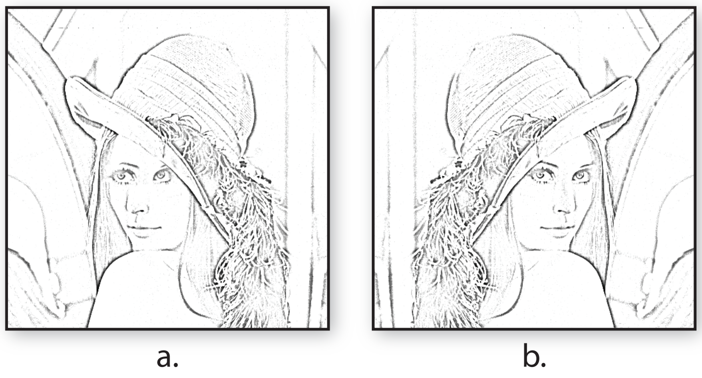

Figure 1.2: Wollaston Illusion: Head pose affects the perceived gaze direction.

The image has been mirrored, except for the eyes and eyebrows, which are exactly the same in both images. Regardless, she seems to look left in **a.** and towards the camera in **b.**

The article [6] more precisely argues that the interpretation of gaze is skewed in the direction of the head pose. This coincides with the assumption that humans are more comfortable looking straight ahead, and as a consequence, most often will do so. For a computer algorithm to correctly estimate the gaze direction, gaining knowledge of the head pose is therefore arguably a necessary preliminary step.

Head pose is a significant element in human–to–human communication as it conveys a lot of subtle, yet important, information. With society constantly more intertwined with technology, human–to–computer interaction beyond that of the standard keyboard & mouse configuration will be fairly relevant in the not so distant future. In fact, already, the major gaming console platforms have developed systems for video-based user input. In particular Sony's *PlayStation Motion Move* and Microsoft's *Xbox Kinect.* The next section discusses several specific applications that rely on the detection of head pose, based on image sequences.

1.5.2 Applications

Detecting and tracking heads autonomously has a wide range of applications beyond that of medical use. It forms the basis for many more complicated applications (such as facial expression detection, eye gaze estimation, etc.). Assuming a working algorithm with the characteristics listed in section 1.3, some suggestions for suitable applications are:

Driver Awareness System

Using a classifier for facial expressions, a camera monitoring a driver could be used to detect the alertness of the driver. This system could then caution the driver through the speakers.

Video Surveillance / Data mining.

Creating a database of visitors' faces, all transformed to being captured from the same viewpoint using head pose information. Feature extraction can be combined with this approach, giving (e.g. police) the option of searching by features like, hair color, accessories, etc.

Research

A system of head and pose tracking could be used in research kinesiological research, i.e. the science of human motion.

Motion Capture

Computer Generated Imaging (CGI) often relies on capturing movement of human actors. A system capable of robustly tracking all body parts from a single camera would make such techniques available for amateurs and low-budget studios.

Conference monitor

In a conference with participants present in the same room around a table, an extension could be made to record dialogue, and register information on who spoke (by audio direction), and to whom (gaze/head-pose) [7].

Human-Computer Interface/Visualization

Head gestures can be used to give commands to a computer (nods, shakes, etc). Moreover, the head position of the user can be used to displace the viewpoint on 3D-generated graphics (which in combination with stereoscopic displays is bound to induce a feeling of awe).

Augmented Reality (AR)

One of the most important elements for believable AR is accurately determining the correct viewpoint of the user. This requires pose estimation methods of high robustness and accuracy, pointing towards statistical methods.

Model Based Encoding

Video in teleconferencing can be encoded using pose knowledge and exploiting human similarity. Similarly to speech compression principals.

We make note that several, if not all, have been researched and in some cases possibly implemented commercially.

1.6 Own Contributions

The underlying principle of the method used in this thesis came from the paper [7]. That is, the combination of a sparse template with a particle filtering system to perform pose estimation with the help of the graphics card. However, there are several significant differences in the approach, which are either considered general improvements, or simplifications suited for the purpose of tracking based on the CIMA videos. Among the author's contributions are:

Dynamic Template
A particular contribution is the *dynamic template*, which allows for initialization to occur in any pose, as well as continued performance and functionality regardless of how much the subject turns his head. The dynamic template implementation uses the traditional graphics pipeline with custom shaders for extracting data, based on a 3D model. The 3D model itself was created with vague features and infant-like proportions particularly suited for the CIMA project. This also includes the implementation of an overlay system for visualizing the tracking, as well as facilitating manual pose initialization.

Particle Filtering Implementation
The implementation of the complete particle filtering system should in itself be considered a significant contribution, and is solely the work of the student (with the exception of random number generation, which comes from [8]). There are many details in the implementation which should be considered contributions, such as the suggested evaluation function (section 3.4), as well as an alternative based on hyperbolic tangent and its corresponding Padé approxmation, or for instance the usage of normals to perform a hit-test (section 3.3.6).

Testing System
A system was created to generate test videos based on user specified movement data. This provides a method for evaluating the performance of the particle filtering system in a controlled environment. A method was also devised to extract the noise components from the CIMA videos, allowing to replicate this effect in the generated videos. This benchmarking system can be used for future theses to test performance against standardized test-cases. Details in section 4.1.

1.7 Structure of Thesis

A brief overview of the thesis:

Chapter 1 - Introduction:
: Gives the reader an introduction to the thesis, rudimentary background information as well as motivation for head pose estimation in general.The background and motivation sections are largely based on the preceding term project [1].

Chapter 2 - Background:
: Covers the main technologies used in the system implementation. An introduction to computer graphics, and using the graphics cards' GPU for doing parallel computation. Both are tailored for the work in this thesis.

Chapter 3 - Method:
: Presents the complete system used for solving the problem of pose estimation. It is structured in an attempt to give the reader an intuitive understanding before delving into the details in the following sections.

Chapter 4 - Results:
: Details the benchmarking system which was developed and used for creating test videos. The chapter also demonstrates the method's robustness to noise and occlusion, and accuracy in tracking and overall performance, using the test videos. Analysis is provided alongside the results.

Chapter 5 - Concluding Remarks:
: The final chapter discusses the performance and merit of the implementation. It gives suggestions for interesting future extensions to the implementation, as well as a finalizing conclusion on the thesis.

Appendix:
: The appendix contains the plots pertaining to chapter 4, and an overview of the data files provided alongside this thesis.

Chapter 2

Background

2.1 Graphics Pipeline, OpenGL & GLSL

As the software implementation in this thesis is intrinsically connected to the openGL rendering pipeline, a brief overview will be given of the traditional rendering pipeline, as well as its transition to shader programming, and consequently development of GPGPU advancements such as openCL.

2.1.1 OpenGL

OpenGL (Open Graphics Library) is a software interface (API) to graphics-specific hardware. OpenGL is a *specification*, in other words, hardware vendors, such as nVidia and ATI, need to create their own drivers which conform to the specifications, making openGL hardware-independent and OS-independent. As such, any openGL code written for a specific hardware is guaranteed to function on all hardware, granted that the hardware is openGL compliant.

An alternative to consider is Microsoft's Direct3D; a similar graphics API. The main difference between openGL and Direct3D is that the latter is a proprietary API, only supported by the Windows platform. Direct3D has also been criticized for being difficult to use in comparison. For this reason, the OpenGL was the preferred choice for use in this thesis.

For the sake of brevity, the following description will err on the side of simplicity. For a detailed introduction, the reader is directed to the OpenGL Reference Manual [9], also known as the OpenGL Blue Book.

2.1.2 Using OpenGL for non-graphic purposes

Using openGL will allow for a neat feedback of the tracking, as the head pose can be visualized by a 3D head model, overlaid on top of the video stream. However, and more importantly, combining the tracking with a 3D model of a head allows for dynamic generation of feature points (which are introduced in chapter 3. This comes particularly in handy if the subject rotates its head such that the initial points no longer face the camera, and become poor for tracking.

2.1.3 Graphics Terminology

Some common terms used in graphics are introduced to make the subsequent sections more approachable.

Vertex

Three-dimensional geometry consists mostly of triangles. The three points which make up the triangle corners are called vertices. The term also applies to the corners of all geometry in general.

Viewing Frustum

A viewing frustum is an enclosed region in the scene, defining the viewable space. Only geometry inside the frustum can potentially appear on the final rendered scene. The shape of this region varies depending on the projection method used, but is typically a frustum of a rectangular pyramid (as shown in figure 2.1).

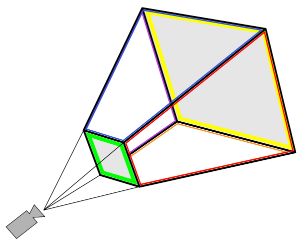

Figure 2.1: Frustum.

Fragment

Similar to a pixel, but contains information beyond color and position, most importantly for this thesis: depth information.

Rasterization

Process in which vector graphics (lines and shapes connected by coordinates and mathematic functions with infinite resolution) are converted to pixels.

Rendering

In short, the complete process of generating an image from a model, by means of computer graphics.

Shader

Set of software instructions which perform rendering effects.

2.1.4 Rendering Pipeline

The process of generating a 2D pixel image, based on a model description of 3D objects (viewpoint, geometry, texture, etc.) is called *rendering*. The series of steps that are required to render a scene is collectively called the *rendering pipeline*, or *graphics pipeline*. The term is somewhat flexible in what it includes, however, the following introduction should suffice for the purpose of this thesis.

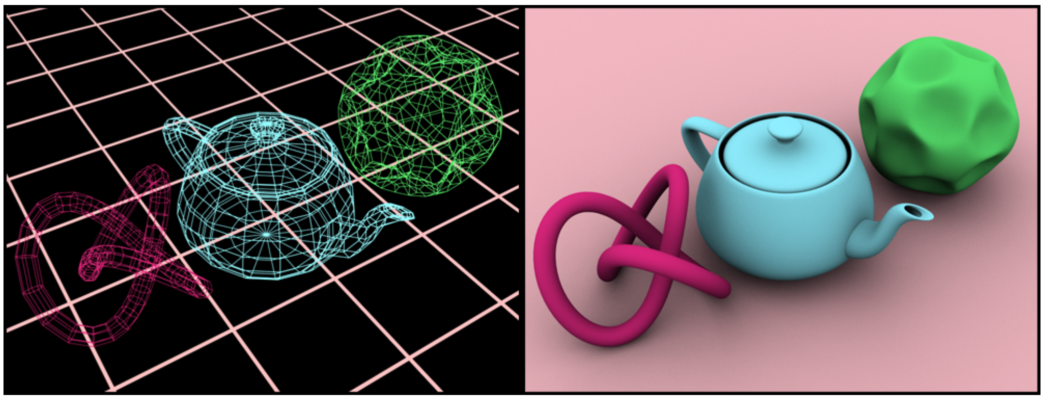
Figure 2.2: Example of rendering

The three main stages of the pipeline are Application, Geometry and Rasterization:

Application Stage

The application state is often considered to be outside the rendering pipeline, as it is done completely on the CPU. It is where the content of the scene is decided, and all underlying logic pertaining to the scene is determined. For example, in the case of a scene with balls bouncing around, the application stage would be in charge of collision detection, and calculating the positioning and rotation of all the geometry building up each ball. The developer has complete control of everything done in the application stage. The output of this stage to the geometry stage is a set of geometry (i.e. triangles, lines, points), as well as matrices for the object as well as camera view transformation, and other shader related values (texture, lights, etc.).

Geometry Stage

The geometry stage operates on a per-vertex basis, which means that they are processed independently from each other. It is capable of repositioning the vertices, as well as calculating a corresponding color value. As the vertices passed on by the application stage are usually in object space with an associated *model transform matrix* (MTM), the vertices are repositioned according to this matrix. After the model transform, a similar transform is done with respect to the *view transform matrix* (VTM), associated with point of view. If an object (for instance a teapot) is rotated, the only necessary change would be in the MTM. Similarly, if only the point of view changed, only the VTM would have to change.

The color of each vertex can be determined by a function based on texture coordinates, surface normals, material properties, light position, etc, referred to as the *shading equation.* The choice of this equation was limited to a few set choices up until the introduction of programmable shaders in early 2001 [1], more on this in section 2.1.6. In particular the shader in this stage is called *vertex shading.*

Following the shader computation, several other steps are performed, most notably geometry projection to transform the 3D scene to a "2.5D" scene. The term 2.5D is used to denote that the screen coordinate of the vertices correspond to their final image output, however the depth information is still preserved. The two most common projection models are *orthographic* and *perspective* projection, described in section 2.1.5.

Rasterization

In short, the purpose of the rasterization stage is to compute and set the pixel color values which cover the geometry, based on the data from the geometry stage. The three vertices of the triangles passed on from the geometry stage serve as interpolation corners for the pixels who's center are within the triangle area.

Figure 2.3 shows a visualization of the interpolation of color values over the surface of a triangle. However the same is done for all parameters pertaining to each vertex. This could for instance be vertices' surface normal, texture coordinate, etc. The final stage of the rasterizer is a new shading operation similar to the vertex shader, here called *pixel shader.* The main difference; instead of computing values for each vertex, it computes it for each pixel (A better term is *fragment*, as it contains information such as raster position (pixel coordinate), color, depth and alpha value, as well as other interpolated attributes). The output of the pixel shader is a handful of bitmaps, commonly called buffers in this context. The two most relevant are

[1]Nvidia's GeForce 3 was the first graphics card with a GPU capable of programmable shaders

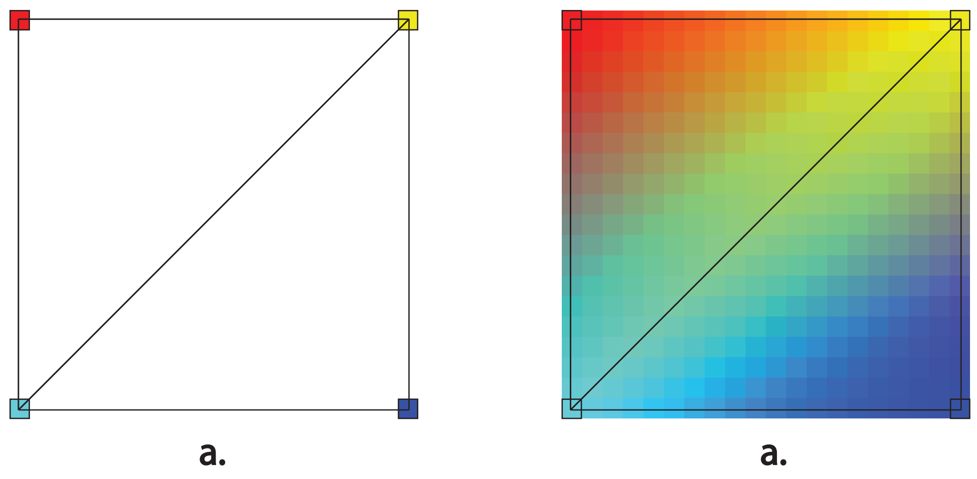

Figure 2.3: **a.** Triangles with vertex colors **b.** Interpolation and rasterization over triangles

the *color buffer* which contains the final output usually sent to the display adapter. The second relevant buffer is the *depth buffer*, also called *Z-buffer* containing a grayscale image representing the distance from the camera to the visible objects.

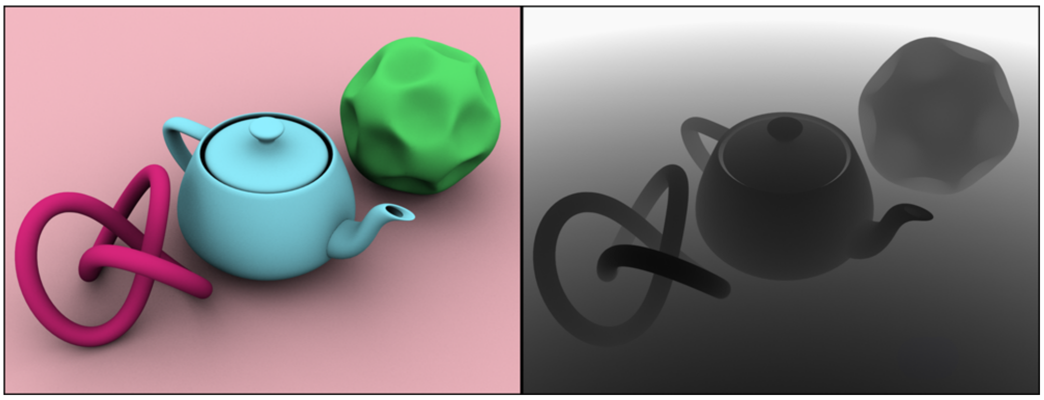

Figure 2.4: Example of the color buffer and depth buffer of a rendered scene.

2.1.5 Projection Models

The two most common projection models are *orthographic* and *perspective*. In the following discussion, it assumed that the scene has the Z-axis pointing towards the viewer, Y-axis down and X-axis to the right. Projection consists of transforming the camera frustum to a unit cube.

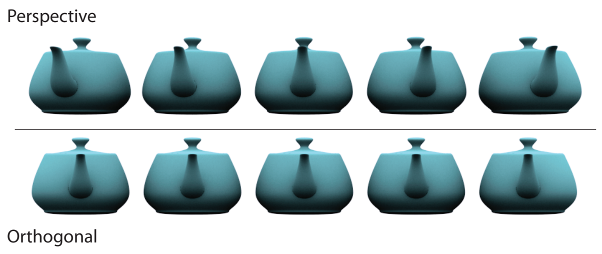

Figure 2.5: Example difference between perspective (top) and orthogonal (bottom) projection.

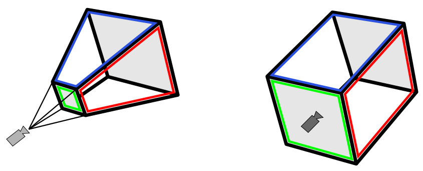

Figure 2.6: The camera frustum for pinhole perspective (left) and orthographic (right) projection.

Orthographic

Orthographic projection can be thought of as simply omitting the depth component. Objects remain the same size regardless of distance to camera, and also do not change shape based on the screen position. In the openGL pipeline, orthographic projection is used by choosing a box shaped frustum. The sides of the box are denoted *(l, r, b, t, n, f)*, corresponding to the left, right, bottom, top, near and far coordinates. Translating this box so that its center is the origin can be done with the transformation matrix **T**, and scaling it to a unit cube is done by the scaling matrix **S**:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{S} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

$$\mathbf{P}_{orth} = \mathbf{S} \cdot \mathbf{T} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{l+r}{2} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{2} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

$\mathbf{P}_{orth}$ in equation (2.2) is equivalent to the orthogonal projection matrix generated by the OpenGL function `glOrtho`. The important thing to note from the orthogonal projection matrix, $\mathbf{P}_{orth}$, is that it preserves the homogeneous coordinate. To demonstrate: given a point in homogeneous coordinates, $\mathbf{q} = [x_0, y_0, z_0, w_0]^T$, the orthogonal projection of this point results in:

$$\hat{\mathbf{q}} = \mathbf{P}_{orth}\ \mathbf{q} = \mathbf{P}_{orth}[x, y, z, w]^T = [\hat{x}, \hat{y}, \hat{z}, w] \tag{2.3}$$

In other words, the whole process of rotating, translating, and orthogonally projecting a point, $\mathbf{q}$, from object space to screen space, $[p_x, p_y]^T$, can equivalently be computed by equation (2.4).

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = s \cdot \mathbf{R}_{2\times3} \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \tag{2.4}$$

where p_x, p_y are the projected screen coordinates, s is a scaling parameter, $\mathbf{R}_{2\times3}$ the upper part of a rotation matrix, T_x, T_y translation values. The importance of this function as well as its direct equivalence to the openGL orthographic projection will become apparent in chapter 3.

Perspective

The perspective projection is a more accurate model of how cameras perceive the world, and represents a perfect pin-hole camera model. The benefit of using perspective projection was not considered sufficient to warrant the more expensive computation. There are several reasons for this assessment, first and foremost, the infant in the scene is resting his head on the table, making the screen displacement highly limited. A comparison of the two projection models is given in figure 2.5.

2.1.6 GLSL and Rendering Normals

The graphics pipeline described earlier had historically a very limited flexibility. The shader equations were limited to a very few choices, all hard-coded, with only a set adjustable parameters. In 2001 the first GPU capable of *programmable shaders* was introduced, and the trend ever since has been increased configurability and flexibility. To program shaders for the GPU, a *shading language* similar to `C` is used. The most common choices are `HLSL`, `Cg` and `GLSL`, which are all very similar to each other. The decision fell on `GLSL` (OpenGL Shading Language) as it was (as the name implies) designed to be used with OpenGL. `HLSL` (High-Level Shader Language) is only supported by Direct3D, and `Cg` (C for graphics) is in the author's opinion a bit more tedious to synchronize with the openGL rendering scene.

The purpose of using programmable shaders in this thesis is for rendering the surface normals of the geometry. As far as shaders go, this is one of the simplest to perform. The basic idea is to pass on the normal information to the vertex shader, and set the RGB color of the fragment to the normal coordinates. The vertex and pixel shader code is shown in below.

Code 2.1: Vertex Shader code

```
varying vec3 normal;
void main()
{
  normal = normalize(gl_NormalMatrix * gl_Normal);
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

The `gl_ModelViewProjectionMatrix` is equivalent to `gl_ProjectionMatrix * gl_ModelViewMatrix`, and are the matrices described in the previous sections. Even though they are equivalent, the vertex shader operates on a per-vertex basis, while matrices are uniform throughout the render-pass. In other words, it would be wasted effort to compute the matrix-matrix multiplication for each vertex; thus, gl_ModelViewProjectionMatrix is a better choice.

By multiplying this matrix with the vertex, (`gl_Vertex`), it is positioned correctly. Storing it in the spatial variable, `gl_Position`, sends it on to the pixel shader. A similar transformation is done to the corresponding normal, `gl_Normal`, as it needs to be transformed to camera space. This is done with the `gl_NormalMatrix` matrix, which is the inverse-transpose of the ModelView matrix. To pass the normal on to the pixel shader, a variable `normal`, is declared with the special attribute of `varying`.

Code 2.2: Pixel Shader code

```
varying vec3 normal;
void main()
{
  vec3 n = normalize(normal);
  gl_FragColor = vec4(n/2 + vec3(0.5,0.5,0.5),1.0);
}
```

The interpolated normals from the geometry stage are re-normalized (to correct for the interpolation). The color buffer stores values between $[0, 1]$, while the normal coordinates range from $[-1, 1]$. The normals are thus rescaled appropriately before being sent to the color buffer, which is done by storing the RGBA-value in `gl_FragColor`. An example output of this shader is shown in figure 2.7. Decomposition of each color channel is shown in figure 2.8, making it easy to see how each component of the normal is represented.

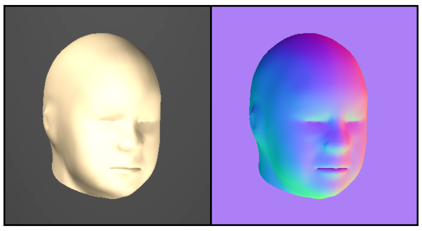

Figure 2.7: Regular rendering (left). GLSL shader outputting normals (right)

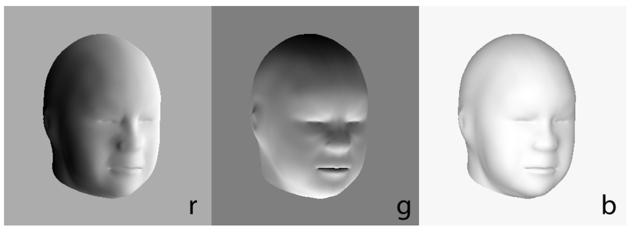

Figure 2.8: Channel decomposition for the normal render.

2.2 GPGPU & OpenCL

"GPUs have evolved to the point where many real-world applications are easily implemented on them and run significantly faster than on multi-core systems. Future computing architectures will be hybrid systems with parallel-core GPUs working in tandem with multi-core CPUs."

- Jack Dongarra [2]

2.2.1 GPGPU

As mentioned in the previous section, graphics hardware started out with a very fixed rendering pipeline. The continued trend was to increase flexibility, more settings to configure; and later on to the advancement of programmable shaders. Due to the nature of graphics computation, the hardware was designed with emphasis on processing large numbers of floating point operations in parallel, as well as provide a large memory bandwidth. Figure 2.9 shows the main difference between a typical CPU and GPU. Just the difference in number of computing cores (around 4 for CPUs and >300 for GPUs at the time of this thesis) alone reflect the CPUs' sequential focus and the GPU's focus on parallelism.

Following the flexibility and the ease of programming shaders, the scientific community began to use the GPU to perform non-graphics computation. This was termed GPGPU, short for General-purpose Computing on Graphics Processing Units. The process would consist of baking computation data into textures, which in turn were accessed by the shaders using texture lookups. The shaders would use the data and compute results, finally passing it on to the CPU-accessible buffers (e.g. color buffer). The OpenGL function `glReadPixels` can be used to access these buffers.The process was constrained to the graphics pipeline, and required considerable effort and "hacking" tricks to make the general algorithms look like graphics applications.

The desire to use the GPUs for GPGPU led to the development of a new GPU architecture allowing even more ease of control than the traditional graphics pipeline would permit, or what would be necessary for pure graphics purposes. They have become fully programmable, allowing use of high-level languages similar to C and C++. It has reached the point of abstracting away the traditional function of the GPU, from a graphics pipeline, to a

[2]Director of the Innovative Computing Laboratory, University of Tennessee

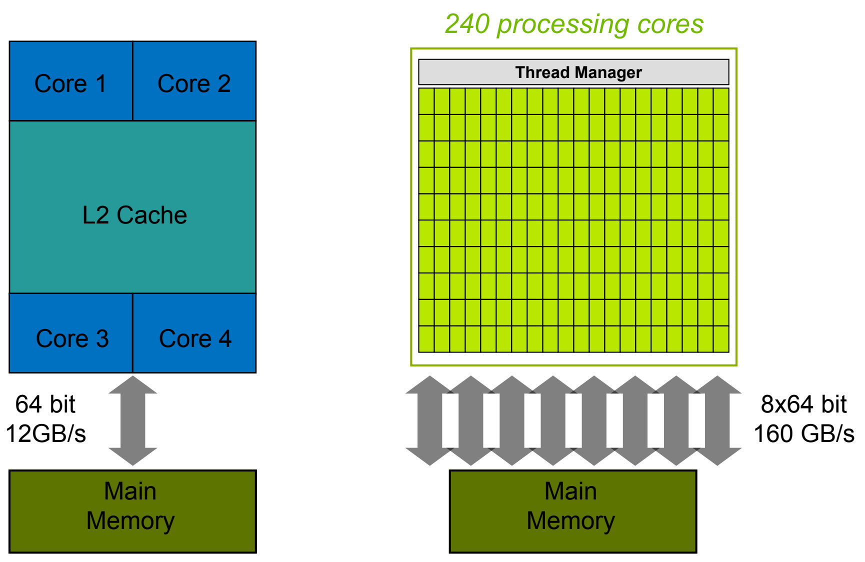

Figure 2.9: Example visualization of computing cores of CPU (left) and GPU (right). Image from Nvidia.

dedicated hardware highly optimized for parallel algorithms requiring large floating-point computational throughput. The GPU architectures in question are NVIDIA's `CUDA` (Compute Unified Device Architecture), and ATI's `AMD FireStream`.

The following sections will give an introduction to this architecture and programming model from an OpenCL (Open Computing Language) point of view. There is a lot to be said about the details in hardware and ways to exploit this for better performance, etc. However, emphasis will be put on implementation, and general benefits of using the GPU as a co-processor.

2.2.2 Comparison of GPGPU APIs

There are several programming APIs for doing GPGPU. Among the most notable are `OpenCL`, `CUDA C` and `DirectCompute`.

Table 2.1: Overview of common GPGPU programming APIs

	Proprietary	**OS specific**	**Hardware specific**
`DirectCompute`	Yes (Microsoft)	Yes (Windows)	**No**
`CUDA`	Yes (Nvidia)	**No**	Yes (Nvidia)
`OpenCL`	**No**	**No**	**No**

DirectCompute

DirectCompute is part of Microsoft's DirectX11+, as such it is limited to the Windows Platform, thereby making it a less appealing choice.

CUDA C

NVIDIA advanced the field considerably with the introduction of CUDA in 2005, and the following C-like programming language CUDA C, or "C for CUDA". The main restrictions here is limitation to NVIDIA hardware, as well as being a proprietary API.

OpenCL

OpenCL is an open standard, royalty free API. Any vendor can follow the specifications, and make their hardware OpenCL compliant. It is thus not limited to any specific hardware vendor, or operating system. The API is also an open standard, welcoming a broader and quicker acceptance. Because of these benefits, OpenCL is considered by the author to be the most relevant choice, and future of GPU computing. This thesis will therefore focus on OpenCL for use in the software implementation.

2.2.3 OpenCL

OpenCL, short for Open Computing Language was initially proposed by Apple (then under the name *Grand Central*) and later passed on to the Khronos Group, which is an industry consortium dedicated to royalty-free, open standard APIs. Khronos Group maintains several well-known APIs, such as the previously discussed OpenGL. The specification was created based on a collaboration from all the big names (Nvidia, ATI, Intel, AMD, Apple, IBM, and several others). Any vendor can adopt the specification and write their own implementation, as it is an open standard. In other words, no single company holds control of the specification.

Device Agnosticism

The most attractive feature of OpenCL, is what is called *device agnosticism*, or *device heterogeneity*. Since OpenCL is only a specification, hardware vendors need only create drivers conforming to the specification for it to be used. In other words, OpenCL is not limited to GPUs, but can just as easily take advantage of multi-core CPUs. An OpenCL code written for a graphics card GPU can, without modification, run on any OpenCL compliant hardware, be it Nvidia or ATI GPUs, AMD or Intel CPUs, IBMs Cell Processor on the Playstation3 or any of an increasing number of hand-held devices. Since the main benefit at the time is usage on GPUs, it will remain the focus of the following discussion.

2.2.4 GPGPU Processing Flow

A program typically starts out on the CPU, with data in Main Memory. The GPU does not have access to this data, so the first step is to transfer the data to the Device Memory. Typically, data needs to be gathered and stored in C-compliant data-structures before being passed on to the GPU. In other words, passing on C++ classes would be a bad idea, as OpenCL does not support it as of yet. The next step is to instruct the GPU to execute a program which processes the data, and calculates results. This program, which runs in parallel is commonly called a *kernel*. Upon completion, the CPU again initiates transfer of memory, this time to get access to the results generated by the GPU.

1. Transfer data from CPU-accessible RAM to GPU accessible RAM.
2. Kernel Execution
3. Transfer back data from GPU-accessible RAM to CPU-accessible RAM.

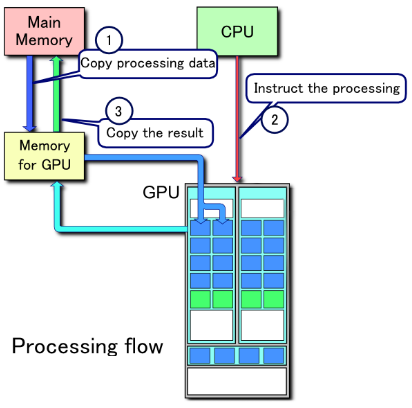

Figure 2.10: GPGPU Processing Flow. Image from Wikimedia Commons repocitory, CUDA page

2.2.5 Speed Comparison w/Example

For small computational tasks, transferring the data from the CPU to the GPU can take many orders of magnitude longer than the computation itself, if kept and done on the CPU. In other words, there is a bottleneck to consider when transferring large chunks of data back and forth from the host memory to device memory.

The Algorithm

Assume a core element of a computationally expensive program was the main bottleneck in the system. One such could be the following evaluation function:

$$V(x,y) = \frac{1.25 \cdot \kappa(x,y)^2}{\kappa(x,y)^2 + 1} \text{ where } \kappa = \frac{x-y}{x+y+1} \tag{2.5}$$

A sequential `C++` code computing this function on the CPU is shown in Code 2.3. The equivalent OpenCL code is shown in Code 2.4.

Code 2.3: Cost Function V(x,y), CPU

```
float costFnc(float x, float y){
  float k = (x-y)/(y+x+1.0f);
  float k2 = k*k;
  return (1.25f*k2)/(k2+1.0f);
}
```

Code 2.4: Cost Function V(x,y), GPU

```
float costFnc(float x, float y){
  float k = native_divide(x-y,y+x+1.0f);
  float k2 = k*k;
  return native_divide(1.25f*k2 , k2+1.0f);
}
```

The differences so far are not significant, and in fact the CPU code would be completely valid. The `native_divide()` function is the same as the usual floating-point division, except that it uses highly optimized device specific routines, if available.

GPU vs. CPU

Consider an application needing to evaluate the function n times. Intuitively, if n is a small number, the cost of transferring memory between devices will greatly overshadow the execution time if it were performed locally on the CPU. Code listing 2.5 shows the obvious CPU implementation. Code listing 2.6 shows the OpenCL implementation. This is where the difference in programming models become apparent.

Code 2.5: CPU program

```
void runHst(float *hst_A, float *hst_B,
            float *hst_results, int n){
  for (int indx = 0; indx < n ; indx++)
  {
    hst_results[indx] = costFnc(hst_A[indx],hst_B[indx]);
  }
}
```

Code 2.6: GPU kernel

```
__kernel void runGPU(__global float *dev_A,
                     __global float *dev_B,
                     __global float *dev_results){
  int gid = get_global_id(0);
  dev_results[gid] = costFnc(dev_A[gid],dev_B[gid]);
}
```

The kernel code is executed in parallel through all available cores on the GPU. In order for the kernel to know exactly which element to process, and where to store the result, it can get its index through the function `get_global_id(0)`. This index is functionally equivalent to the, `indx`, in the for-loop of the CPU. When the CPU tells the GPU to execute the kernel, the number of *work-items* to use (n, in this case), is specified.

Speed Comparison

A comparison was made by measuring the execution times for the CPU and GPU code, over different values of n. Figure 2.11 shows, as expected, that for small values of n, the CPU outperforms the GPU. This is mainly due to the delays in transferring data between the main memory and device memory. However, the GPU quickly outperforms the CPU by a factor 7. This factor depends on the complexity of the kernel, as well the degree in which the kernel is optimized to take advantage of the memory hierarchy. For the implementation in chapter 3 a factor of up to 21 was obtained.

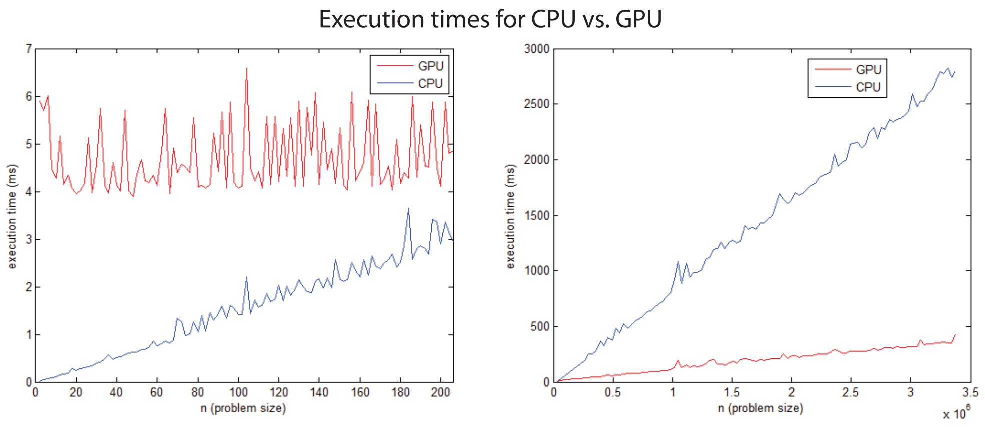

Figure 2.11: Computation time for equivalent code on GPU and CPU.

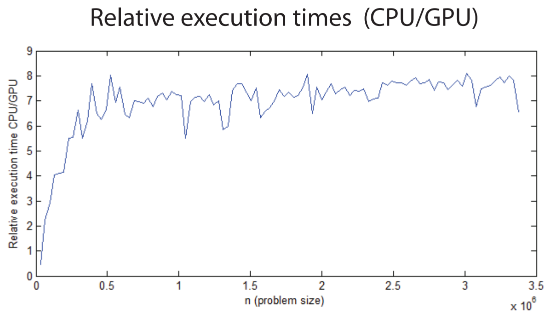

Figure 2.12: Relative computation time

2.3 OpenCV

Open Source Computer Vision (OpenCV) is a cross-platform library initially developed by Intel for doing real-time image processing. The first version was released in 2000, and has grown through contributions by individuals as well as corporations (amongst, Willow Garage, Intel and Google) ever since. The latest version (2.1) was released in April 2010. Its free use under the BSD-license, as well as good performance makes it very attractive for the scientific community, and has been used in numerous research papers. A list of some of the major functionality areas covered by openCV:

- **Data Persistence**: Methods for storing and loading image and video-files.

- **Image Processing**: Standard CV-algorithms such as convolution, FFT, edge detection, thresholding, filtering, etc.

- **Image Transforms and Sampling**: Color space conversion; resizing, image pyramids, warping, rotation, sub-image extraction, etc.

- **Higher Level Modules**: Optical flow, face detection, point tracking, template matching, stereo vision, Kalman-filter, etc.

- **Machine Learning**: Support for many ML methods. Artificial Neural Networks, Bayes Classifiers, Support Vector Machines, Decision Trees, etc.

- **Camera Support**: Easy to interface with camera devices, as well as advanced lens calibration tools.

- **User Interface**: Keyboard and Mouse callback functionality, as well window management.

- **Graphics Drawing**: Support for drawing text and shapes on image (e.g. for overlaying information on image).

- **Data Structures & Operations**: Vectors, matrices with corresponding mathematical arithmetic support. Higher-level data structures such as graphs and trees.

Chapter 3

Method

3.1 Overview

The system implemented for performing head pose estimation of infants is presented in this chapter. The underlying principle is *Particle Filtering*, which will introduced in the process of covering this implementation. For a general introduction to particle filtering, refer to [10], [11] and [12]. It is desirable to give the reader an overview of the system and its functionality before delving into details pertaining to each step. It can be benefitial for the reader to look at the demonstration videos in the accompanied CD.

3.1.1 Terminology

Pose

The head's particular position in world space $[x, y, z]$, as well as its orientation $[\phi, \delta, \psi]$ is collectively referred to as the *pose*.

3D Model

A *3D Model*, or just simply *mesh*, is used to refer to the geometrical representation of a head with smooth features. Figure 3.1 shows the mesh used by the system.

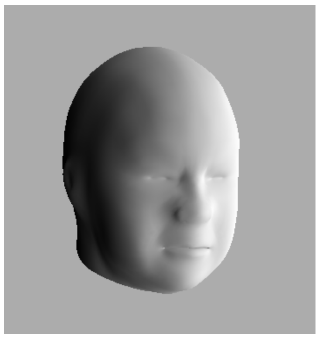

Figure 3.1: Shows the 3D model used by the pose estimation system.

State

A state contains pose information, and can either refer to the actual state, $\mathbf{x}$, or a hypothesized state, $\tilde{\mathbf{x}}$. The terms will sometimes be used interchangeably. Note that in some cases, the term *particle* (see below) is preferred over *state* to make it clear that the state in question is a hypothesis; which is a small abuse of terminology.

Particle

A particle can be thought of as a pose suggestion. In the general sense it contains a hypothesized state, $\tilde{\mathbf{x}}$, and an associated weight corresponding to its fitness, π, i.e. likelihood of being a correct hypothesis of the actual state, given the observational data. The symbol used for a particle is ξ.

$$\xi = [\tilde{\mathbf{x}}, \pi] \tag{3.1}$$

Particle Cloud

A collection of particles is referred to as a *particle cloud.*

$$\Xi = \{\xi^1, \xi^2, \ldots, \xi^M\} = \{[\tilde{\mathbf{x}}^1, \pi^1], [\tilde{\mathbf{x}}^2, \pi^2], \ldots, [\tilde{\mathbf{x}}^M, \pi^M]\} \tag{3.2}$$

Template

In order to evaluate the correctness of a particle, there needs to be a way to relate a pose to the actual image. This is done through *template matching.* The template is comprised of a set of *feature points*, in the number of hundreds. The template is mapped onto the underlying image based on a particle. In the case of this thesis, the template is rotated, scaled and positioned, such that a direct similarity comparison can be made between the template and the underlying video frame.

Feature Point

The feature points are locally defined, and contain world coordinates, $\mathbf{q}$, surface normal, $\mathbf{n}$, and texture value, g. By locally defined, it is meant that all feature points are defined in the same coordinate space. However, when discussing the FPs normals and points, it is implied that the template has been transposed to match the current output, even though values, strictly speaking, never change.

3.1.2 Overview of the Algorithm

An overview of program flow is given in figure 3.2.

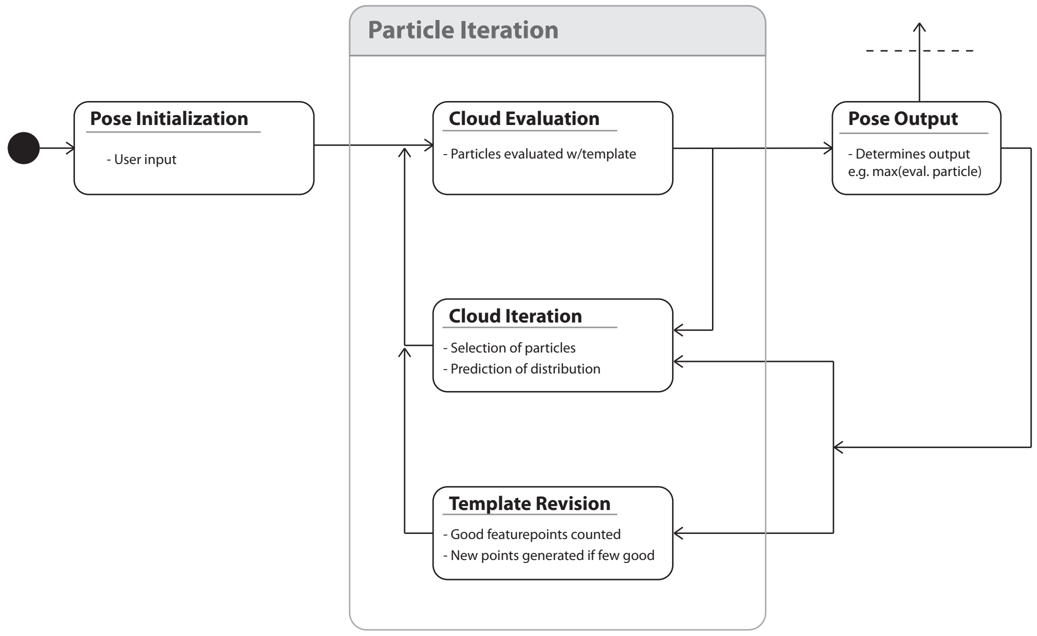

Figure 3.2: Shows the sequence of the major steps the system is comprised of.

Pose Initialization

The program starts out showing the first frame of the video. It is paused at this frame, waiting the user to manually position and rotate a 3D head model to match the subject in the frame. The window continuously reads the user input, and updates an overlay of the 3D model until the user is satisfied with the positioning. This is a quick and easy task using the mouse for input. An example is given in figure 3.3. This model is used as the basis for creating the template. –(section 3.2)–

Particle Cloud Evaluation

A particular positioning and rotation of the 3D model is called a *particle*. A set of particles is referred to a *particle cloud*. The process of determining a

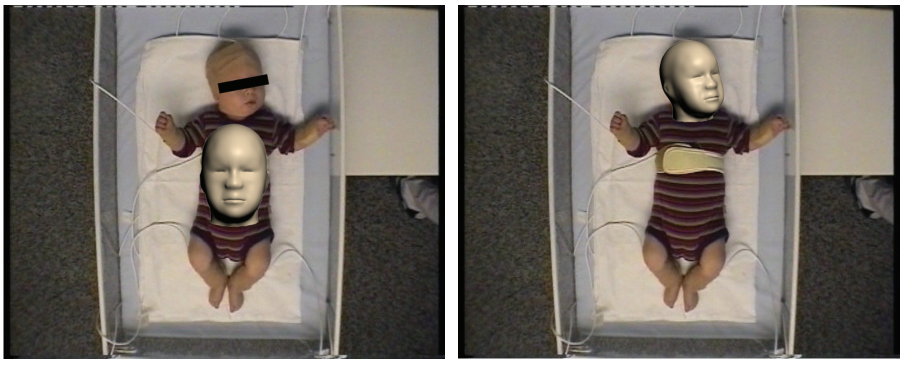

Figure 3.3: **a.** Shows the initial frame with default 3D model position.
b. The model is correctly placed

particle's correspondence with the underlying image is called *particle evaluation*. So in short, this step consists of determining the correctness of each suggested pose, given the current frame. – (section 3.4) –

Output

In each frame, the system outputs a single particle, corresponding to the best suggestion given the particle evaluation in the prior step. This can for instance be an average of the M highest evaluated particles for the current frame.

Particle Cloud Iteration

As mentioned, each particle functions as a pose suggestion. I.e. *"check to see if this particular pose is a good choice"*. As the subject head moves around, it is beneficial to update the particle cloud to improve the likelihood of finding a good correspondence. E.g.: a simple choice would be to search near the output of the current frame. – (section 3.5) –

Template Revision

When the subject rotates his head, the initial feature points begin to become occluded by the face itself, and serve as poor choices. This part of the algorithm checks to see how many feature points remain usable, and creates new ones, based on the 3D model and current frame output particle. – (section 3.3) –

3.1.3 State & Particle Description

State

In terms of *Particle Filtering*, the state functions as the search parameters. Since the system should estimate the pose, a natural choice is to make this part of the state description. The state is defined as

$$\mathbf{x} = (T_x, T_y, S, R_x, R_y, R_z, \alpha) \tag{3.3}$$

T_x and T_y are scalar translation coordinates, and R_x, R_y, R_z are scalar angles denoting rotation about the x, y and z axis, respectively. There is no metric by which to determine the distance to the head. For instance, a small head near the camera can have the same visual size as a larger head further away from the camera. Since the projection model used is orthogonal, the depth coordinate (T_z) can more intuitively be represented by a scaling constant, S. It can be considered as the reciprocal of the depth (distance from the camera to the head). There is also an implementational reason for this choice, which is mentioned in section 3.3.5.

Lastly, and not related to the pose is the α parameter, which is used to compensate for change in lighting intensity throughout the tracking sequence. That is, if there are subtle changes in lighting intensity, the α-value, instead of the pose, can be used to compensate.

Particle

A *particle* consists of a hypothesized state, $\tilde{\mathbf{x}}$, defined as above, and a corresponding value, π, in the range $[0, 1]$ corresponding to that individual particle likelihood of being the correct hypothesis for the actual state. The complete set of particles is called the particle cloud, represented by $\Xi = \{\xi^i\}$.

Time Discretization

Time is discretized in terms of video frames. For instance the notation used for a particle cloud with M particles at frame k:

$$\Xi_k = \{\xi_k^{1:M}\} = \{\xi_k^1 \ldots, \xi_k^M\} = \{[\tilde{\mathbf{x}}_k^1, \pi_k^1], \ldots, [\tilde{\mathbf{x}}_k^M, \pi_k^M]\} \tag{3.4}$$

3.1.4 Problem Description

This section gives a brief mathematical description of the tracking problem. The tracking problem this system is meant to solve can be considered the evolution of the state sequence $\mathbf{x}_k$, which is to say, determine the corresponding pose to each frame throughout the image sequence.

The state sequence is constrained to be a *Markov Process*, meaning that the process retains no memory, and thus only the prior state has any influence on the next. This is represented by function shown in equation (3.5).

$$\mathbf{x}_k = \mathbf{f}_k(\mathbf{x}_{k-1}, \mathbf{v}_{k-1}) \tag{3.5}$$

where $\mathbf{v}$ is a process noise sequence. This function is often called the state *state transition model.*

Since we only have a video footage of the subject, direct knowledge of $\mathbf{x}_k$ is not available, and it thus becomes a *Hidden Markov Process.* The tracking goal is to recursively estimate $\mathbf{x}_k$ from available measurements, $\mathbf{z}_k$. That is, we seek estimates of $\mathbf{x}_k$ based on the current and prior measurements, $\mathbf{z}_{1:k} \equiv \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_k\}$. The measurement model is defined in equation (3.6) and is evaluated based on the image pixels from the video stream.

$$\mathbf{z}_k = \mathbf{h}_k(\mathbf{x}_k, \mathbf{w}_k) \tag{3.6}$$

To summarize; we wish to attain the probability of a particular state being the correct representation of reality, given all available measurements. In other words, we wish to determine the probability density function (pdf) $p(\mathbf{x}_k|\mathbf{z}_{1:k})$.

The pdf for the state in the initial frame is provided by the user (through the pose initialization step), and as such is considered known, and independent of the measurement. I.e. $p(\mathbf{x}_0|\mathbf{z}_0) = p(\mathbf{x}_0)$, which is a necessary basis for the particle filtering.

3.1.5 Feature Point & Template Description

Feature Point Definition

A feature point (FP) is a vector comprising of position, $\mathbf{q}$, surface normal, $\mathbf{n}$, and a grayscale texture value, g. τ is used to represent this vector.

$$\boldsymbol{\tau} = [\mathbf{q}, \mathbf{n}, g] = [q_x, q_y, q_z, n_x, n_y, n_z, g] \tag{3.7}$$

Feature points are locally defined, which is to say, the positions and normals are all defined in the same coordinate system, regardless of the frame in which the FPs are created.

Template Definition

A template is built up of a finite number, N, of feature points (FP), as described by equation (3.8).

$$\mathbf{T} = \{\boldsymbol{\tau}^{1:N}\} = [\boldsymbol{\tau}^1, \boldsymbol{\tau}^2, \ldots, \boldsymbol{\tau}^N] \tag{3.8}$$

Say for instance the template represented a two-dimensional flat plane. In this simple case, the template would be a sparse representation of the image it is based on. That is, $\boldsymbol{\tau} = [q_x, q_y, 0, 0, 0, 1, I(q_x, q_y)]$, where q_x and q_y is the pixel coordinate, and $I(q_x, q_y)$ the pixel's intensity value. This simplification could function reasonably well, granted that the subject does not rotate about the x- or y-axis. Such a rotation would alter the normals' value, and thus affect the FP's representational capability.

The implemented solution employs the FPs' full expressability, and is described in detail in section 3.3. An accurate representation takes advantage of a FP's surface normal, as it can be used to determine if the FP contains valuable information. If not, the FP can be disregarded.

Dynamic vs. Static Feature Points

When discussing FPs, it is worth mentioning that upon creation, a FP remains unaltered throughout the whole tracking sequence, i.e. *static.* The main benefit is robustness to occlusion, and resistance to drifting. Consider the subject partially covering his face with an arm, if the FPs where updated to match the texture, the tracking might decide to follow the hand when it moves away from the face. A related issue is the concept of *drifting*, where it gradually loses track of the original template, and begins to track the surroundings, etc.

Dynamic vs. Static Template

Upon initialization of the tracking system, a template consisting of N FPs is generated before continuing past the first frame. As mentioned, FPs are static, and remain unchanged. However, the template itself could either admit new FPs during the tracking sequence (*dynamic template*), or keep the number constant at initial N (*static template*).

Creating new FPs present the same issues as described above for dynamic FPs, however, the downsides of a static template are too significant to ignore. Consider the simplistic flat template structure mentioned earlier. If for instance the subject rotates more than 90 degrees about either x or y-axis, and the template is rotated to match, then none of the normals point towards the camera[1]. Thus, the template has lost all representational capability, and continued tracking is made pointless.

A caveat of using a dynamic template on the other hand, is that its benefit rests solely on the system's ability to accurately represent the correct pose upon the creation of new FPs. Furthermore, implementation of the dynamic template system is a fairly complicated procedure. Regardless, it has been considered worth the effort to support larger deviation in pose angles.

[1]By which is meant that the normal's z-coordinate is negative

3.2 Pose Initialization Alternatives

Although only the first method listed has been implemented, it is worth mentioning a few alternatives to pose initialization for the sake of completeness. The basic methods here are limited in some way, making them suitable for initializing the pose, but not for performing the full tracking. This goes without saying; if it were not the case, these methods themselves could instead be used for tracking the head pose.

3.2.1 User Input Model Placement (UIMP)

This is the implemented method, and was described in section 3.1. In short: manual initialization.

3.2.2 Tuned Face Detection (TFD)

A face detector, such as the Viola-Jones method (see [1] for an overview) can be tuned to a particular orientation. The method outputs a region containing the face, thus location and scale is acquired. The orientation can be assumed to correspond to the tuned detector, as it will only detect a face given a particular pose. The problem with this method is that it requires a particular pose in order to be initialized, and consequently might have to skip several frames.

3.2.3 TFD + Active Appearance Model (AAM)

Active Appearance Model (AAM) is a method for fitting statistical shapes and textures to images. Having learned the variation of a face shape, and how it correlates to texture, the AAM framework iteratively fits the statistical model to match the underlying image. This method is very powerful, but relies on a sub-region containing the face in a somewhat known pose. The main benefit comes from having a 3D model specifically fitted to the actual subject, making the template more accurate, and potentially improve the tracking. The downside is the complexity of such a system. For more information on AAM, refer to [13].

3.2.4 Thoughts on Choice

Since the 3D model's accuracy in representing the subject is not of critical importance, advanced methods such as AAM was not considered cost effective for this thesis. As for TFD, which is already implemented in OpenCV, it can help the user along in some cases (when it detects a face on the first frame) by suggesting a pose that the user can further improve upon. However, the effort of doing it manually is so minute (taking about 3 seconds) that focus was instead put on the tracking itself, leaving the initialization up to the user.

3.3 Template & Feature Points

To reiterate, a *template* is comprised of a set of *feature points* (FPs), which in turn are vectors containing position, surface normal and texture value. Sections 3.1.1 and 3.1.5 describe the principle and functionality of FPs and template, which is important to understand before continuing. FPs can be thought of as points on the surface of the 3D model with pertaining surface normal, as well as texture value corresponding to the underlying pixel from the video frame. Figure 3.4 illustrates this concept. This section covers the details regarding how such feature points are chosen, and how they are initialized.

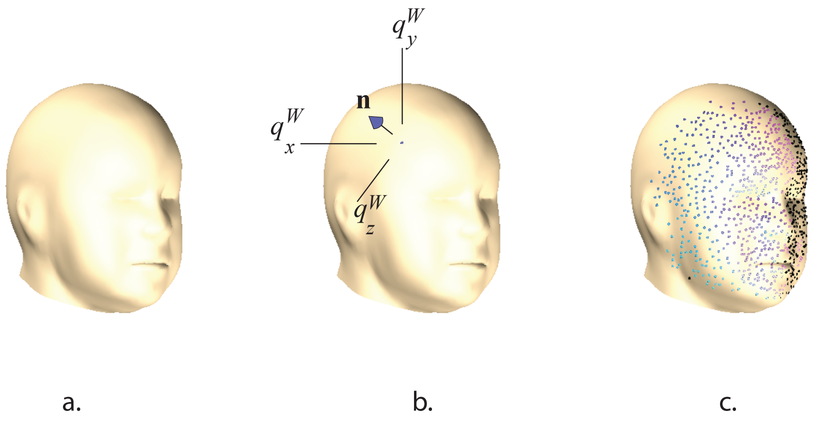

Figure 3.4: **a.** 3D mesh model on which template is based. **b.** One feature point is shown with denoted values of $q_x^W, q_y^W, q_z^W, n_x, n_y, n_z$. Overlaying this on top of a grayscale image, intensity value, g, is also extracted. **c.** Shows a template when fully generated. Black FPs are bad, as they do not adequately face the camera.

The 3D head model overlays the image with the subject's head beneath. A FP is chosen based on the underlying image pixel coordinate. This coordinate itself ends up becoming FP's two first elements of the FP vector, q_x and q_y. The last element, g, corresponds to the intensity of the pixel coordinate, all remaining elements (surface normal and depth) are based on the overlaying 3D model. Figure 3.5 illustrates this.

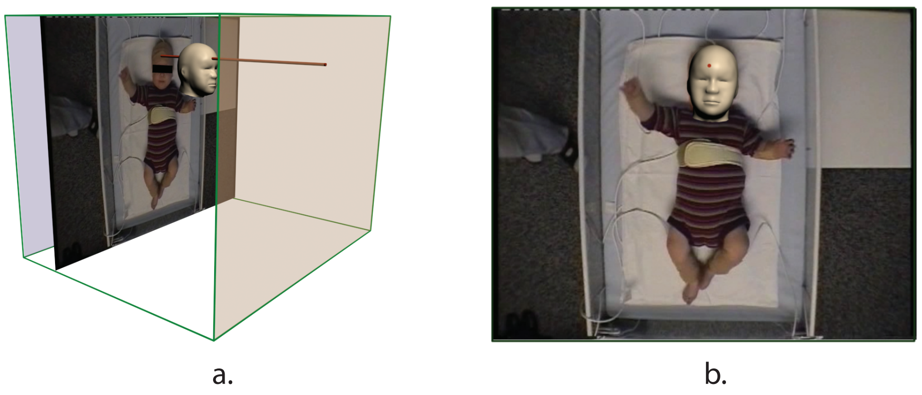

Figure 3.5: a. Shows a conceptualization of the orthogonal rendering frustum (green) with video frame rendered on top of a square polygon in the background, with the head model overlaid. The line going straight through the model from frame to camera represents the pixel selection from the video frame. **b.** Shows how that particular render looks.

3.3.1 Projecting FP onto Image Plane

As mentioned in earlier sections, the FPs are specified in the template coordinate system. Each FP has a grayscale intensity value which corresponds to a pixel on the image frame. This correspondence is determined through orthogonally projecting the FP onto the image plane through a state vector, $\tilde{\mathbf{x}}$. In other words, when evaluating a particle, the template is translated according to the particle's state, and each FP's stored grayscale value is compared to that of the corresponding pixel in the current frame. Translation of the template is shown in figure 3.6, while the projection is illustrated in 3.5.

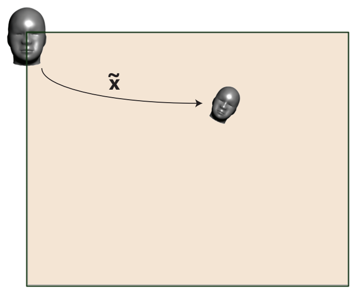

Figure 3.6: a. Template transposed to match $\tilde{\mathbf{x}}$

A FP position in the template, $\mathbf{q} = [q_x, q_y, q_z]^{\mathrm{T}}$, is projected to a point

$\mathbf{p} = [p_x, p_y]^{\mathrm{T}}$ on the image plane as given by:

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = S \cdot \mathbf{R}_{2\times 3} \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \tag{3.9}$$

Rotation and scaling is done prior to translation as it necessary to pivot about the template's center. $\mathbf{R}_{2\times 3}$ is the 2×3 upper sub-matrix of the rotation matrix $\mathbf{R}$ given by:

$$\mathbf{R} = \begin{bmatrix} & \mathbf{R}_{2\times 3} & \\ \omega_1 & \omega_2 & \omega_3 \end{bmatrix} = \mathbf{R}_z \mathbf{R}_y \mathbf{R}_x \tag{3.10}$$

where $\mathbf{R}_x$, $\mathbf{R}_y$ and $\mathbf{R}_z$ are fixed axis rotation matrices which rotates respectively about the x, y and z-axis, with angles R_x, R_y and R_z.

3.3.2 Frame Feature Consideration

When determining which FPs to create, a good idea is to consider the underlying frame when choosing which points to select. Regions in the image with prominent features are more likely better choices than areas with no contrast. There are several ways of determining which pixels are good for tracking, one such is *edge detection*, in particular the Sobel edge detection. Figure 3.7. Although edge detection is the only implemented approach, other suggestions are color– and texture segmentation.

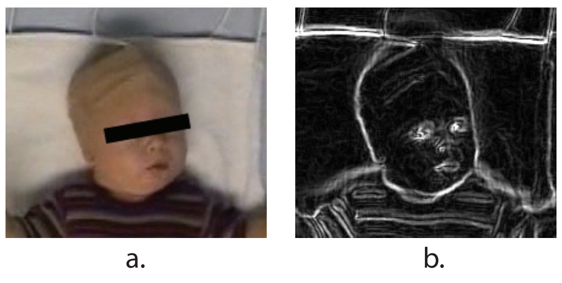
[redacted]

Figure 3.7: Shows the Sobel operator on a frame, highlighting good choices for creating FPs

3.3.3 Local vs. Regional Property

A property to consider when determining which FPs to create, is whether the creation process of a single FP requires knowledge of the surrounding pixels. This becomes an important consideration when implementing FP creation.

Regional

The creation process is based on an image region. Feature consideration requires regional knowledge, which is apparent as an edge cannot be defined by a single pixel. Color segmentation for instance, does not require regional knowledge, while texture segmentation on the other hand, does. The main benefit of a regional approach is that it allows for finding good tracking points.

Local

The creation process is only based on the particular pixel at hand. The benefit of a local method is that it does not require retrieving and processing a whole image region from the graphics card in order to create a particle, which is potentially faster.

Template Window

In order to speed up regional methods, a smaller sub-image containing just the subject's head and overlaid 3D model is used. This is extracted using the output state. Because of the orthogonal projection, this process is easy.

3.3.4 Determining the Number to Create

When the subject rotates about the x- or y-axis, the original FPs' surface normals become less directed towards the camera[2]. When the number of FPs in the template which are facing the camera drop under a certain threshold, the template becomes ill-conditioned, and new FPs are generated.

Note on Translating Normals

Normally, when geometry is rotated scaled and projected, some special concern needs to be made for the normal. Consider the non-uniform scaling case

[2]As mentioned earlier, the FPs themselves do not change in numerical values, but it is implied that the template has been rotated to match the current pose output.

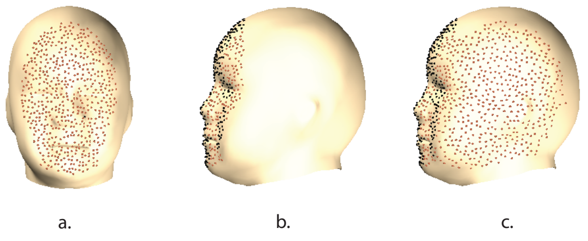

Figure 3.8: Good FPs shown in red, bad ones shown in black. **a.** Well conditioned template. **b.** Badly conditioned template. New FPs needed. **c.** New FPs created. Template now well conditioned.

in figure 3.9. The resulting "normal", $\mathbf{n}'$, is clearly no longer perpendicular to the surface, $\mathbf{s}'$.

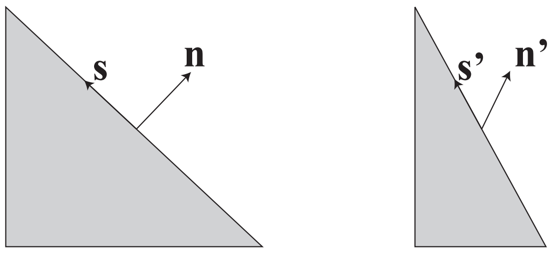

Figure 3.9: Issue with preservation of surface normal after surface transformation.

In the general case, the transformation needs to be the transpose of the inverse of the surface transform. This is shown in equation (3.11). $\mathbf{G}$ and $\mathbf{F}$ correspond to the transform on the normal and surface, respectively, such that perpendicularity is preserved. k is an arbitrary scaling constant.

$$(\mathbf{Gn})^{\mathrm{T}}\mathbf{Fs} = \mathbf{n}^{\mathrm{T}}\mathbf{G}^{\mathrm{T}}\mathbf{Fs} = 0 \overset{\mathbf{n}^{\mathrm{T}}\mathbf{s}=0}{\Rightarrow} \mathbf{G}^{\mathrm{T}}\mathbf{F} = k\mathbf{I} \Rightarrow \mathbf{G} = k(\mathbf{F}^{-1})^{\mathrm{T}} \tag{3.11}$$

If the surface transform consists of rotation, translation and/or uniform scaling, there is no need to compute this inverse transpose. Translation does not affect the surface normal. As for pure rotation, the matrix is orthogonal, and its inverse is also its transpose. I.e. $\mathbf{F^{-1}} = \mathbf{F}^{\mathrm{T}}$ and $\mathbf{G} = k(\mathbf{F}^{-1})^{\mathrm{T}} = k\mathbf{F}$.

In the case of scaling, only the length of the normal is changed, and orthogonality is still preserved. The projection model used orthogonal projection and uniform scaling. In other words, the pose itself can directly be used to calculate the normal, if subsequently normalized.

Counting Good Normals

For each FP in the template, its corresponding normal is transformed to match the current pose output, $\mathbf{x}_k$. If the z-component is above a threshold, t_z, the FP is considered good. If the number of good FPs, c_{good}, drops below a threshold, t_f, a new batch of n FPs is created. This is shown in Algorithm 3.1.

Algorithm 3.1: Determining how many new FPs to create

```
$c_{\text{good}} \leftarrow 0$
for each FP $\in$ T do
    $n_z$ = recalcNormalGetZ(FP, $\mathbf{x}_k$)
    if $n_z > t_z$
        $c_{\text{good}} \leftarrow c_{\text{good}} + 1$
if $c_{\text{good}} < t_f$
    createNewBatch($n$)
```

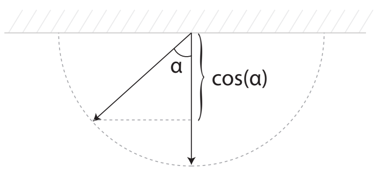

Figure 3.10: Shows the view direction, surface normal, and angle between

Since the surface normal has been transformed to screen space, its depth component is directly related to its direction towards the camera. A value of 1 corresponds to pointing directly towards the camera, -1 directly away.

Assuming no self-occlusion in the model geometry, any positive value of the depth component indicates visibility. Self occlusion is not a large concern as the model is fairly convex. For example: in order to only count FPs with normals within 60° deviation from view angle, the threshold would be $t_z = \cos(60°) = 0.5$, as shown in figure 3.10.

It is beneficiary to overshoot in the number of FPs that are created; such that the total amount becomes larger than the threshold by a margin. Which is a consequence of it being less costly per FP to create a batch instead of individual FPs. This is more so the case for regional methods mentioned in section 3.3.3.

3.3.5 Getting Model Data

Given the frame image, and an overlaid model. Creating a FP for the highlighted example coordinate might seem like a trivial matter. The pixel coordinate and intensity is based on the frame itself, while depth and normal values are determined using the 3D model. Figure 3.11 illustrates this. Superscript notation is used to distinguish between FPs in world coordinate system and FPs template coordinate system, shown in equations (3.12a) and (3.12b) respectively.

$$\boldsymbol{\tau}^W = [q_x^W, q_y^W, q_z^W, n_x^W, n_y^W, n_z^W, g^W] \tag{3.12a}$$

$$\boldsymbol{\tau} = [q_x, q_y, q_z, n_x, n_y, n_z, g] \tag{3.12b}$$

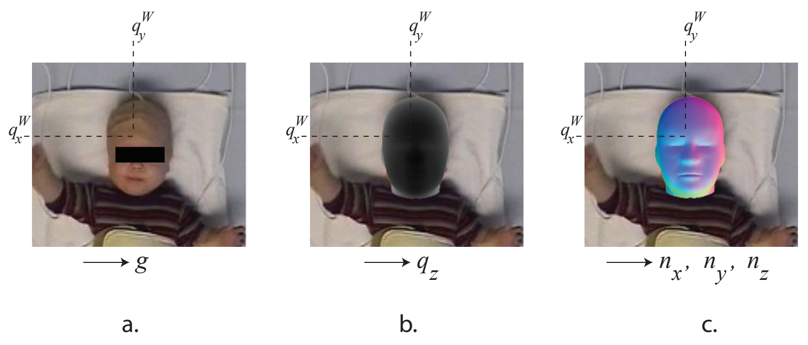

Figure 3.11: **a.** Frame with head model matching pose. Marked coordinate. x,y is known, g is gathered from the image, the rest come from the 3D model

Position (q_x, q_y)

The $[x, y]$ pixel position is assumed to be known. It can be the result of a random choice, or through a feature sensitive choice (ref. section 3.3.2).

$$[q_x^W, q_y^W] \leftarrow [x, y] \tag{3.13}$$

Pixel intensity (g)

Every frame is converted to a grayscale image. The pixel intensity is a simple lookup, in the sense of `getPixelValue(frame,x,y)`. The value is independent of coordinate system. Thus

$$q_z \leftarrow \texttt{getPixelValue(frame,x,y)} \tag{3.14}$$

Position (q_z) – Depth

By rendering the 3D model, the z-buffer in the graphics card contains the depth information. See section 2.1.4 for more information. The rendering frustum used is a box with top–left–near corner at `(0,0,0)` and lower–right–back corner at `(height, width, 255)`. The `width` and `height` correspond to the frame dimensions, such that there is a $1:1$ correspondence. The depth values range from 0 (near) to 255 (far), which are chosen for simplicity to match 8–bit *Z–buffer*.

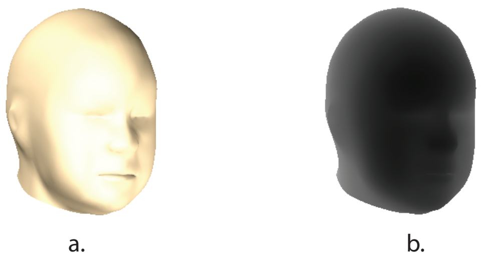

Figure 3.12: **a.** Rendered Scene **b.** Depth Buffer

The state description uses a scaling value rather than z-coordinate. This simplifies matters as the 3D model's center's z-distance can be set constant at 128 from the camera (i.e. half way deep in the frustum). This allows to set the new FP's z-coordinate directly as:

$$q_z^W \leftarrow 128.0 - \texttt{getPixelValue(openGLZBuffer, x, y)} \tag{3.15}$$

Normals (n_x, n_y, n_z)

In section 2.1.6 it was described how to render the 3D model, such that the normals are displayed as color vectors. Figure 3.13 shows and example of a regular scene, its corresponding normal render, and a decomposition of RGB-channels. Each color channel contains 8bit information. That is, the normal values from the color buffer range from $[0, 255] \in \mathbb{N}_0$. Equation (3.16) takes this into account when setting the values:

$$\begin{aligned}[\hat{n}_x, \hat{n}_y, \hat{n}_y] &\leftarrow \texttt{getPixelValue(glslNormalRender, x, y)} \\ [n_x^W, n_y^W, n_z^W] &\leftarrow \frac{[\hat{n}_x, \hat{n}_y, \hat{n}_y]}{127.5} - [1,1,1]\end{aligned} \tag{3.16}$$

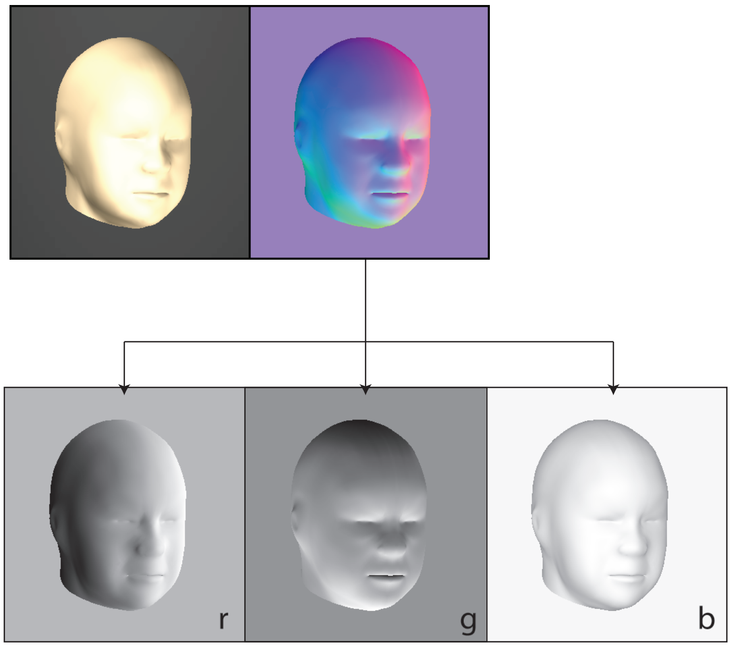

Figure 3.13: Shows the regular render used in overlay (top left). Normal render (top right), and its decomposition in each RGB color channel in the lower row. This shows that the x-component (red-channel) increases for values towards the right, downwards for the y-component, and towards camera for the z-component.

Compensating for Model State

Thus far, the following has been aquired: $[q_x^W, q_y^W, q_z^W, n_x^W, n_y^W, n_z^W, g]$. The current state, $\mathbf{x}_k$, does not affect color value g, thus $g = g^W$. As for the rest, they need to be made independent of the state used when placing the model.

$$\mathbf{x} = [T_x, T_y, S, R_x, R_y, R_z, \alpha] \tag{3.17}$$

The process of compensation for model state is almost similar to reversing the projection described in equation (3.9) in section 3.3.1.

$$\begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} = S^{-1} \cdot \mathbf{R}^{\mathrm{T}} \left(\begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix}^W - \begin{bmatrix} T_x \\ T_y \\ 0 \end{bmatrix} + \begin{bmatrix} C_x \\ C_y \\ 0 \end{bmatrix} \right) \tag{3.18a}$$

$$\begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} = S^{-1} \cdot \mathbf{R}^{\mathrm{T}} \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}^W \tag{3.18b}$$

C_x and C_y are the center coordinate of the template window (from section 3.3.3). I.e. $C_x = 0.5 \cdot \texttt{width}$ and $C_y = 0.5 \cdot \texttt{height}$. Rotation matrices are orthogonal, such that the transpose $\mathbf{R}^{\mathrm{T}}$ is used instead of its inverse $\mathbf{R}^{-1}$, which are equivalent.

3.3.6 Hit Test

Determining whether a pixel is overlapped by the model is referred to as a *hit test*, and are used to assure that a selected coordinate is likely to pertain to the head in the frame. It is also desirable to have a margin distance from the edges of the model. Two methods are proposed, the former is implemented.

Using Alpha Map (regional)

The depth–map [3] used earlier can be thresholded to produce a *binary image* (also called a *binary alpha map*), as shown in figure 3.14.

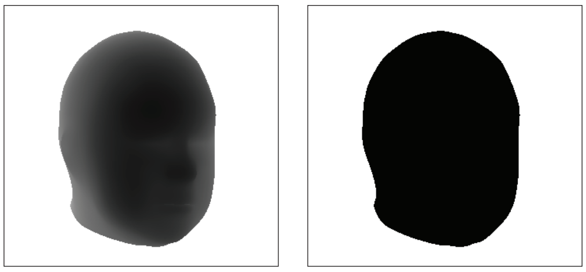

Figure 3.14: Shows the depth map, and thresholding

By blurring this image and subsequently re-applying thresholding, a new binary image is produced with the desired characteristics. There are several common blurring methods to choose from. *Gaussian blur* gives a mathematically perfect shrinkage, however, also very computationally expensive. A more suited method is called *box blur*, and is much faster. A comparison of these two methods is shown in figure 3.15.
It is assumed that this is a common and well known method for performing area reduction, however no literature study has been made to verify. It is also worth noting that doing multiple box blur passes approximates the result to that of Gaussian blur, such that more accuracy can easily be acquired if desirable, and at little extra cost. However, as can be seen on the figure, the difference between the two methods is negligible, thus favoring the box blur.

[3]It should be noted that the alpha buffer is accessible from the graphics card, similar to the depth buffer.

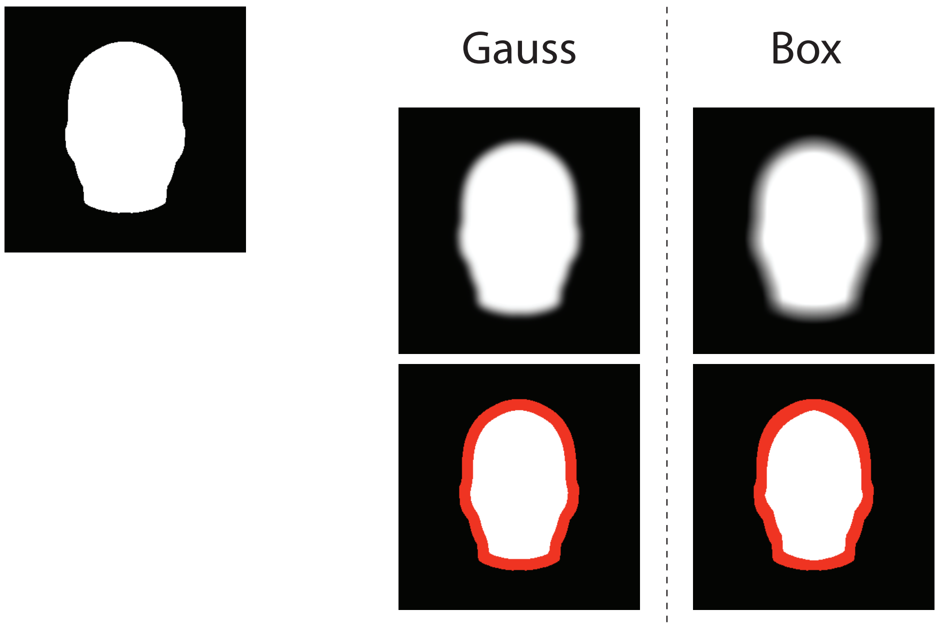

Figure 3.15: Shows a comparison between Gaussian and box blur to perform area reduction. Top left shows the alpha map. The reduced area is shown in red for the respective Gaussian blur and box blur.

Performing this kind of hit test is considered a regional method, as it relies on kernel filtering.

Using Normal Information (local)

Surface normal information can be used to achieve a similar result as the previous method, with the added property of being a local method (i.e. can be computed solely from the information of a single pixel coordinate). Another benefit is that should it pass this hit test, it is also guaranteed for it to be considered a good normal[4]. As such, this method also performs a thresholding, on angle between the surface normal and view normal. The result of this method is shown in figure 3.16.

[4]A good normal is considered one that passes the visibility test from section 3.3.3.

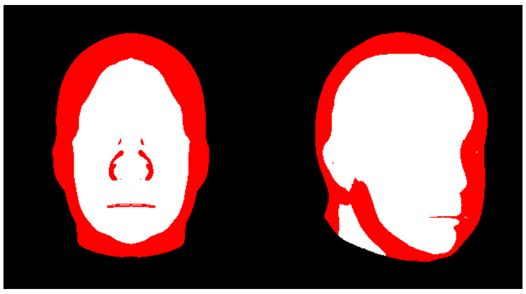

Figure 3.16: Result of using normal thresholding for computing hit test. Example shown for two angles. The reduced area is marked is red. Compare with figure 3.15

3.3.7 Proximity Test

In order to achieve a good spread in FPs, a minimum distance between any two FPs is required. Determining whether a suggested FP coordinate is valid in this regard, is called a *proximity test*. As with the hit test, two methods are described, one local and another regional. The former in this case has been implemented.

Local

Local proximity test is straight forward. It is a simple match between current suggested FP and all other existing ones in the template. This requires transforming the suggested coordinate to template space, according to equation (3.18a).

Algorithm 3.2: localProximityTest(FP,T,$\mathbf{x}$)

```
1 $\mathbf{q} \leftarrow$ transformToTemplateSpace($\mathrm{FP}_{\mathbf{q}}, \mathbf{x}$)
2 for each $\widehat{\mathrm{FP}}$ in T do
3   $\hat{\mathbf{q}} \leftarrow \widehat{\mathrm{FP}}_{\mathbf{q}}$
4   if distance($\hat{\mathbf{q}}, \mathbf{q}$) $< t_d$
5     return false
6 return true
```

The distance thresholding in line 4 should for correctness be the Euclidean distance between the two points. However, for speed, this line is implemented by checking whether $\hat{\mathbf{q}}$ is within a bounding box around $\mathbf{q}$ with distance t_d. The approximation is considered to be reasonable.

Regional

A regional method was not implemented, and unless the batch size is large, it is considered to be less effective than the local method. This regional method starts out creating a white grayscale *weight image*. It traverses through all good[5] FPs in the template, transforms them from template to image space, and reduces the values of the surrounding the pixels in the weight image. The hit test is now reduced to checking the value of the corresponding pixel in the weight image.

Algorithm 3.3: setUpWeightImage(T,$\mathbf{x}$)

```
weightImg $\leftarrow$ createWhiteImage()
for each $\boldsymbol{\tau}$ in T do
  $\boldsymbol{\tau}^W \leftarrow$ transformToImageSpace($\boldsymbol{\tau}$, x)
  if isGood($\boldsymbol{\tau}^W$)
    $[q_x^W, q_y^W] \hookleftarrow \boldsymbol{\tau^W}$
    reduceSurroundingPixels(weightImg, $q_x^W$, $q_y^W$)
return weightImg
```

Algorithm 3.4: regionalProximityTest(weightImg, q_x^w, q_y^w)

```
return getPixelValue(weightImg, $x, y$) $> t_g$
```

[5]A "good FP" is a FP with a normal pointing within an threshold angle towards the camera

3.3.8 Creation Process

Algorithm 3.5 below shows the creation process of a batch of n FPs, using the methods described earlier.

Algorithm 3.5: createNewFPs(n)

```
for $i \leftarrow$ to $n$ do:
    do:
        do:
            $[q_x^W, q_y^W] \leftarrow$ randomPixel()
        while !passLocalHitTest($q_x^W, q_y^W$)                ▷ section 3.3.6
        $[q_x, q_y, q_z] \leftarrow$ getPosition($q_x^W, q_y^W$)          ▷ section 3.3.5
    while !localProximityTest($[q_x, q_y, q_z], \mathrm{T}, \mathbf{x}$)     ▷ section 3.3.7
    ▷ Valid point found. Now create FP:
    $[n_x, n_y, n_z] \leftarrow$ getNormal($q_x^W, q_y^W$)              ▷ section 3.3.5
    $g \leftarrow$ getColorValue($q_x^W, q_y^W$)                       ▷ section 3.3.5
    $\boldsymbol{\tau}_{\mathrm{new}}[i] \leftarrow [q_x, q_y, q_z, n_x, n_y, n_z, g]$
```

3.4 Evaluation of Particles

Each particle contains a hypothesized state, $\tilde{\mathbf{x}}$, which can be evaluated through template matching. In section 3.3.1, it was described how the FPs in a template could be projected onto the image plane through a state description. The projection equation (3.9) is repeated below as algorithm 3.6.

Algorithm 3.6: fp2ImageProjection($\boldsymbol{\tau}, \mathbf{x}$)

```
$[q_x, q_y, q_z] \hookleftarrow \boldsymbol{\tau}$
$[T_x, T_y, S, R_x, R_y, R_z] \hookleftarrow \mathbf{x}$
$\mathbf{R}_{2\times3} \leftarrow$ createSubRotationMatrix($R_x, R_y, R_z$)
$[p_x, p_y]^{\mathrm{T}} \leftarrow S \cdot \mathbf{R}_{2\times3}[q_x, q_y, q_z]^{\mathrm{T}} + [T_x, T_y]^{\mathrm{T}}$
return $[p_x, p_y]^{\mathrm{T}}$
```

By projecting the FP onto the image plane, a comparison can be made between a FP's stored color value, and the corresponding color value on the current video frame. This comparison forms the basis for evaluating a particle, the higher the matching, the more likely the particle is a correct representation of the actual state. Algorithm 3.7 shows the pseudocode for how the weight of a single particle is evaluated.

Algorithm 3.7: evaluateParticle($\tilde{\mathbf{x}}$,T)

```
$\varepsilon \leftarrow 0$
$n_{\text{good}} \leftarrow 0$
for each $\boldsymbol{\tau}$ in T do:
    $g \hookleftarrow \boldsymbol{\tau}$
    $\alpha \hookleftarrow \tilde{\mathbf{x}}$
    if isGood($\boldsymbol{\tau}, \tilde{\mathbf{x}}$) do:                      ▷ $\boldsymbol{\tau}$ is visible
        $n_{\text{good}} \leftarrow n_{\text{good}} + 1$                  ▷ Count good FPs
        $[p_x, p_y]^{\mathrm{T}} \leftarrow$ fp2ImageProjection($\boldsymbol{\tau}, \tilde{\mathbf{x}}$)
        $\hat{g} \leftarrow$ getPixelValue(frame, $p_x, p_y$)
        $\varepsilon \leftarrow \varepsilon +$ costFunction($g, \hat{g}, \alpha$)
return $n_{\text{good}}/\varepsilon$                                ▷ $\varepsilon \neq 0$ is guaranteed
```

The `costFunction` computes a matching error between the two pixel values; $\hat{g}$ from the current image frame, and g from the FP in the template. In a sense, the particle cloud can be regarded as the search points in the state-space. The α-value allows for accounting for constant lighting difference between the template and frame (see section 3.4.1). The matching error, ε, is accumulated over all good FPs in the template, and the particle's weight is the reciprocal of this accumulated error.

The particle filtering system has a large amount of particles, of which all should be weighted according to algorithm 3.7. This process is listed in algorithm 3.8 below. In short, it evaluates the weight of each particle and subsequently normalizes the weights throughout the cloud.

Algorithm 3.8: evaluateParticleCloud(Ξ,T)

```
$i \leftarrow 0$
$\boldsymbol{\pi}[1\,..\,\text{length}(\Xi)] \leftarrow \mathbf{0}$
for each $\xi$ in $\Xi$ do:
    $\tilde{\mathbf{x}} \hookleftarrow \xi$
    $i \leftarrow i + 1$
    $\boldsymbol{\pi}[i] \leftarrow$ evaluateParticle($\tilde{\mathbf{x}}, T$)
▷ Normalize weights:
sum $\leftarrow$ sumOver($\boldsymbol{\pi}$)
for $k \leftarrow 1$ to length($\Xi$) do:
    $\boldsymbol{\pi}[k] \leftarrow \boldsymbol{\pi}[k]/\text{sum}$
return $\boldsymbol{\pi}$
```

In normalizing the weights, the particle cloud functions as a numerical approximation of the probability density function $p(\tilde{\mathbf{x}}_k|\mathbf{z}_k)$, where $\mathbf{z}_k$ are the measurements.

3.4.1 Cost Function

The cost function evaluates the error associated to the distance between the color value of to pixels. There are several good ways to implement this function. It is very desirable to reduce the computation cost of evaluating this function, as it is in the core functionality of the particle filtering system. Given M particles and N FPs, the cost function is evaluated $M \cdot N$ times each frame.

A suggested cost function, V, is given in equation (3.19):

$$V(\mathrm{g}, \hat{\mathrm{g}}, \alpha) = \rho(\kappa(\mathrm{g}, \hat{\mathrm{g}}, \alpha)) \tag{3.19}$$

where κ and ρ are defined as:

$$\kappa(\mathrm{g}, \hat{\mathrm{g}}, \alpha) = 2\frac{(1-\alpha)\mathrm{g} - (1+\alpha)\hat{\mathrm{g}}}{\mathrm{g} + \hat{\mathrm{g}} + 1} \qquad \rho(\kappa) = 0.313\frac{\kappa^2}{\kappa^2 + 1} \tag{3.20}$$

Parameters g and $\hat{g}$ correspond to the grayscale color value of the FP and image frame respectively. α is used to remedy color discrepancy due to change in lighting. A plot of this function for varying values of α is shown in figure 3.17.

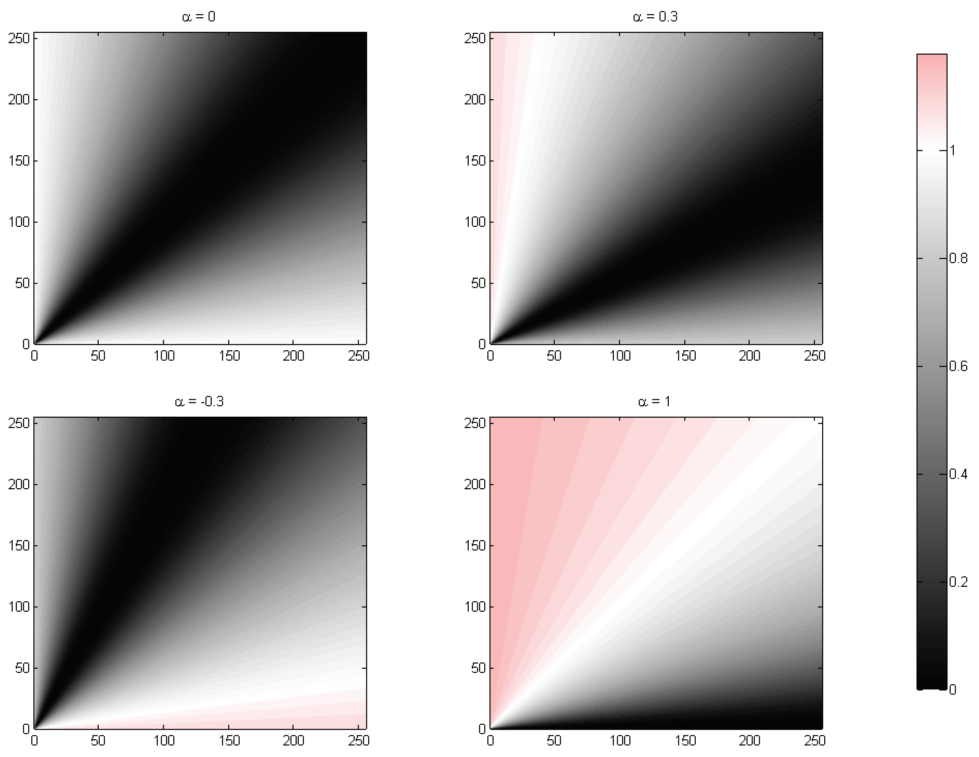

Figure 3.17: Cost function from equation (3.19) plotted for values of $\alpha = \{0.0, 0.3, -0.3, 1.0\}$.

The x and y axes correspond to g and $\hat{g}$ respectively, and range from $[0\,..255]$ covering the complete function space. A desired property of the cost function is to return values in the same range $[0, 1]$, regardless of values of α. In the current cost function, this is only strictly the case for $\alpha = 0$, however, it is fairly accurate for values near 0. It is expected that the actual α-values in the particle cloud are close to 0, such that this is not a major concern. However, the next section discusses a method to improve this.

3.4.2 Correcting for α

In order to determine the consequence of the cost function returning values outside the range [0,1], a modification is made to equation (3.20), such that it compensates for different values of α. This way, a performance comparison can be made, an it can be determined if the correction is considered worth the extra computation.

$$\rho(\kappa, \alpha) = \frac{\kappa^2}{(\nu(\alpha)(\kappa^2 + 1))} \tag{3.21}$$

The plot in fig. 3.18 shows the max value of the original cost function, plotted against ranging values of α, together with a quadratic approximation. Using

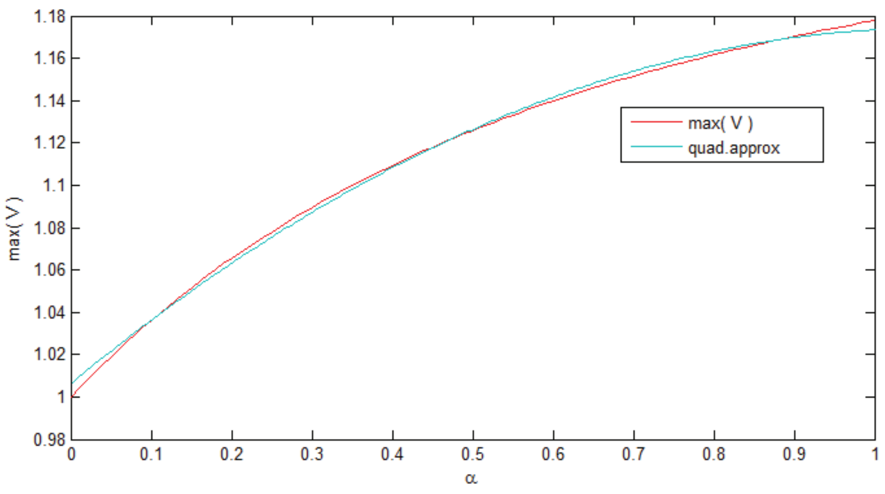

Figure 3.18: Max vale of cost function from equation (3.19) plotted against ranging values of *alpha*. A quadratic approximation is shown in green.

the quadratic approximation, $\tilde{\nu}(\alpha)$, can be rewritten as:

$$\tilde{\nu}(\alpha) = -0.11645\alpha^2 + 0.25|\alpha| + 0.80; \tag{3.22}$$

Note that $\tilde{\nu}(\alpha)$ has been made symmetric about the y-axis, such that the correction is good for all α in the valid range $[-1, 1]$.

The modified cost function is shown in figure 3.19, and shows the desired properties:

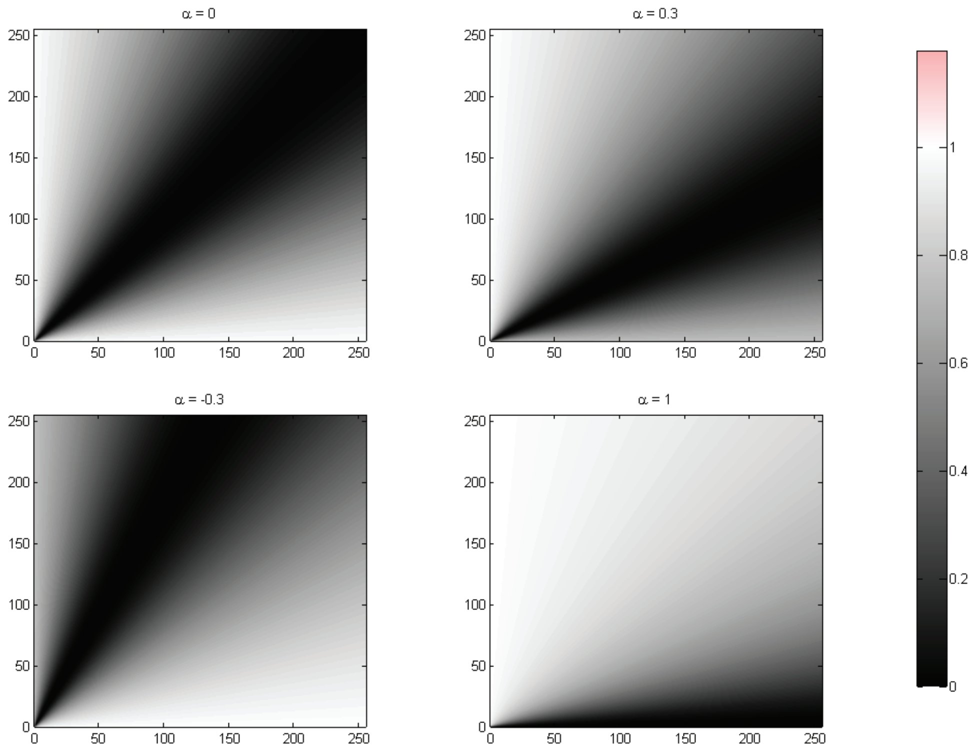

Figure 3.19: Modified cost function using eqn. (3.21) and (3.22)

3.4.3 Other Approaches Worth Considering

Some effort was made to find good cost functions with similar properties to the one already covered. The hyperbolic tangent function has properties which make it of interest to cost functions.

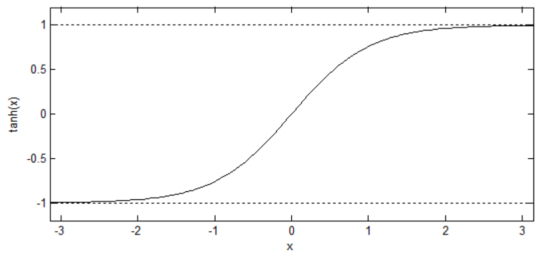

Figure 3.20: tanh(x)

The new cost function given below, using t as subscript to distinguish from the previous cost function. The inner function κ is kept unaltered.

$$V_t(\mathrm{g}, \hat{\mathrm{g}}, \alpha) = \rho_t(\kappa(\mathrm{g}, \hat{\mathrm{g}}, \alpha)) \tag{3.23}$$

where κ and ρ are defined as:

$$\kappa(\mathrm{g}, \hat{\mathrm{g}}, \alpha) = \frac{(1-\alpha)\mathrm{g} - (1+\alpha)\hat{\mathrm{g}}}{\mathrm{g} + \hat{\mathrm{g}} + 1} \qquad \rho_t(\kappa, \alpha) = \frac{d \cdot \tanh(a \cdot |\kappa| + b) - c}{\tilde{\nu}_t(\alpha)} \tag{3.24}$$

The scalars a, b, c, d are used to scale the functions, and can be modified for different properties. $\tilde{\nu}(\alpha)$ is a symmetric linear function used to normalize the cost function over values of α, the same as before:

$$\tilde{\nu}_t(\alpha) = e \cdot |\alpha| + f \tag{3.25}$$

The constants e and f are found through regression in the same way as was done for equation (3.22). Due to the nature of the cost function, ν is in this case almost perfectly approximated by a linear function.
Using values:

$$a = 0.01848 \qquad b = -2.0 \qquad c = 0.48201 \tag{3.26a}$$

$$d = 0.5 \qquad e = 0.0006858 \qquad f = 0.00065615 \tag{3.26b}$$

a new plot of the cost function is shown in figure 3.21, which shows to be similar to the previous method. A difference to the previous method is the configurability of the function – the four constants a, b, c, d control many features of the cost function.

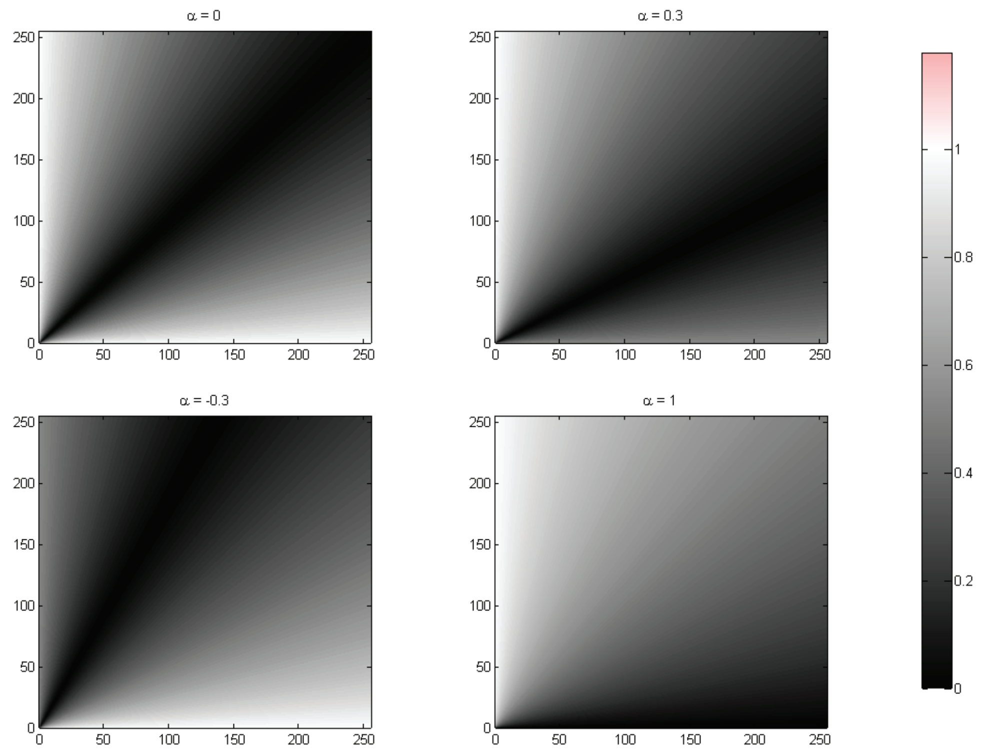

Figure 3.21

Pade approximation of hyperbolic tangent

Hyperbolic tangent is defined as:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.27}$$

Which is considered to be fairly expensive to compute. For that reason, it is desirable to find an approximation to hyperbolic tangent. Padé approximation showed to yield far better results than Taylor series expansion. The Padé approximant of order $(3, 4)$ of tanh is given in equation (3.28).

$$\tanh_p(x) = \frac{10x^3 + 105x}{x^4 + 45x^2 + 105} \tag{3.28}$$

The remarkable thing about this approximation is that for the interesting ranges of tanh, i.e. [-π, π], the maximum error is $6.4977 \cdot 10^-3$. An implementation in `C++`-code computed the approximation in 53% of the time of `tanh()` from the `cmath` library.

Directly comparing the two in a plot would be meaningless, as they would completely overlap. Instead a residue plot, $|\tanh(x) - \tanh_p(x)|$, is shown in figure 3.22.

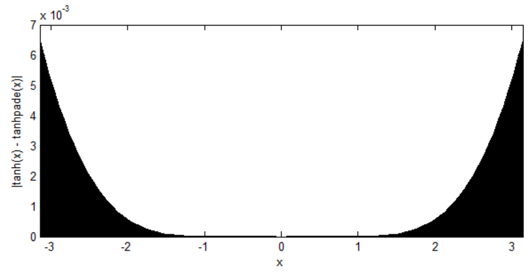

Figure 3.22: Residue plot, $|\tanh(x) - \tanh_p(x)|$

3.5 Particle Filter Iteration

After evaluating the particle cloud, in a sense one has gained a numerical representation of the probability density function of the current state of the system, $p(\mathbf{x}_k|\mathbf{z}_{1:k})$. Based on this, one should create a new particle cloud matching the prediction for the next step, $p(\mathbf{x}_{k+1}|\mathbf{z}_{1:k})$. The intuition behind this is that the particle cloud functions as the search points in the state space, and it is desirable to concentrate the search in areas more likely to contain the correct representation. Consequently, this allows for using physical attributes of the tracking system, to improve the prediction. There are thus many ways to approach this – many ways to simplify and model the dynamics of the system.

The main iteration steps of the particle filtering method consists of *selection*, *prediction* and *evaluation*, of which the latter has been discussed in depth in section 3.4. A brief textual description of the stages is given, with an illustration in figure 3.23.

Selection

Given particle cloud $\Xi_{k-1} = \{[\tilde{\mathbf{x}}^1_{k-1}, \pi^1_{k-1}], \ldots, [\tilde{\mathbf{x}}^M_{k-1}, \pi^M_{k-1}]\}$, the selection creates a new cloud $\hat{\Xi}_k$ by a weighted resampling of the previous distribution. That is, the weaker particles are replaced by copies of stronger ones.

Prediction

Prediction does not take into account the particle's weight. This step consists in relocating the particle by resampling the probability of state transition — i.e. assuming the previous state, $\mathbf{x}_{k-1}$, corresponds to the particle at hand, $\hat{\tilde{\mathbf{x}}}_k$, choose the new location of $\tilde{\mathbf{x}}_k$ by resampling

$$\tilde{\mathbf{x}}_k \sim p(\mathbf{x}_k|\mathbf{x}_{k-1} = \hat{\tilde{\mathbf{x}}}_k) \tag{3.29}$$

where $\hat{\tilde{\mathbf{x}}}_k$ comes from $\hat{\Xi}_k$ in the selection stage.

In other words, the prediction stage relocates particles based on the dynamics of the system.

Evaluation

The new particle cloud, Ξ_k, from the prediction stage is evaluated using the template, as described in section 3.4.

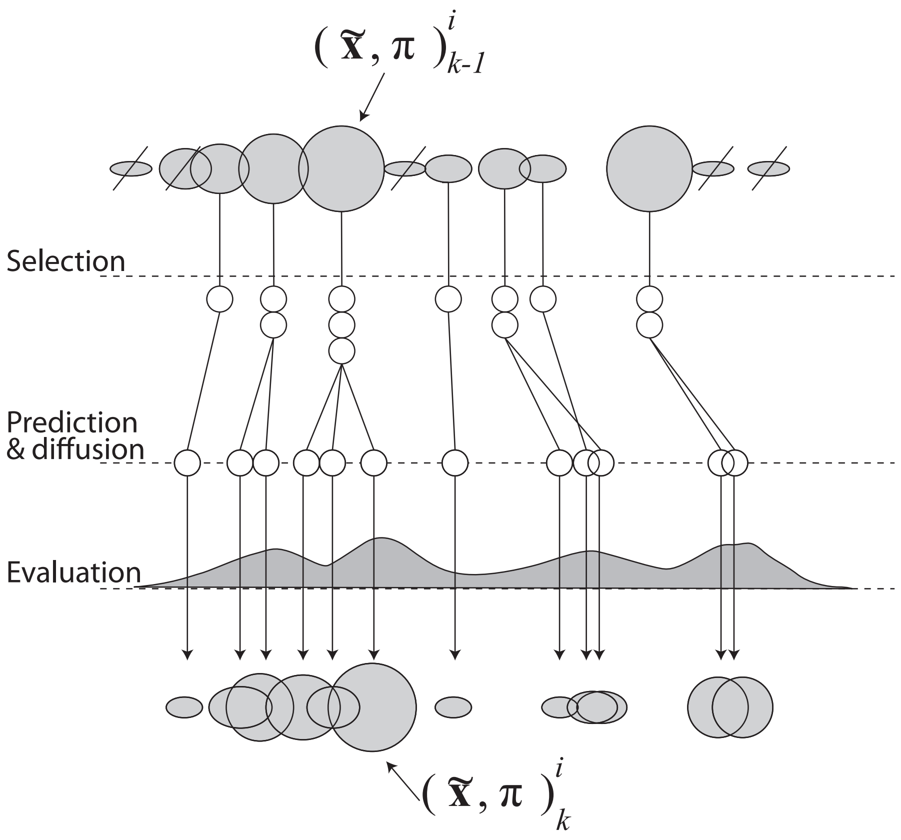
$(\tilde{\mathbf{x}}, \pi)^i_{k-1}$
Selection
Prediction & diffusion
Evaluation
$(\tilde{\mathbf{x}}, \pi)^i_k$

Figure 3.23: Overview of particle filtering iteration, showing the main steps: *selection*, *prediction* and *evaluation* of particles. The top row circles repersent the particle cloud, the radius denoting its fitness. Depending on their performance, a selected few are represented multiple times, while lesser ones are discarded. A prediction step relocates the particles, resulting in a new particle cloud, which is subsequently re-evaluted.64

3.5.1 Selection

Selection is the process of disregarding the particles which are considered to have failed, and replacing them with copies of the more promising ones. This is done through resampling, described in algorithm 3.9

Reshape of weights

The weights from the evaluation step need to be normalized such that

$$\sum_{i=1}^{N} \pi^i = 1 \tag{3.30}$$

It can be worth investigating what properties can be gained from the initial evaluated weights, prior to normalization. For instance if a connection can be determined between drop in weight values and the tracking failing, etc.

Resampling Algorithm

The algorithm presented is an adaptation of the one suggested in [14].

Algorithm 3.9: $\hat{\Xi}$ = resample(Ξ)

```
▷ Initialize comulative density function (CDF) of $\Xi$
$C[0\,..M] \leftarrow \mathbf{0}$
$C[0] \leftarrow \pi^0$
for $i \leftarrow 1$ to $M$ do:
    $C[i] \leftarrow C[i-1] + \pi^i$
▷ Traversing through CDF:
$u_1 \leftarrow$ randomValue($0, M^{-1}$) ▷ Random starting point from range $[0, M^{-1}]$
$i \leftarrow 0$
for $j \leftarrow 0$ to $N$ do:
    $u_j \leftarrow u_1 + jN^{-1}$
    while $u_j > C[i]$ and $i < N-1$ do:
        $i \leftarrow i + 1$
    $\hat{\tilde{\mathbf{x}}}^j \leftarrow \tilde{\mathbf{x}}^i$
return $\hat{\Xi} \leftarrow \{[\hat{\tilde{\mathbf{x}}}^1, 0], \ldots, [\hat{\tilde{\mathbf{x}}}^M, 0]\}$
```

3.5.2 Prediction

As mentioned, the prediction stage relocates particles based on the dynamics of the system. To give an example, assume the system conforms to a linear AR-process [6]. The new particle state could then be generated as

$$\tilde{\mathbf{x}}_k^i = \mathbf{A}\hat{\tilde{\mathbf{x}}}_k^i + (\mathbf{I} - \mathbf{A})\overline{\mathbf{x}}_{1:k-1} + \mathbf{B}\mathbf{w}_k^i \tag{3.31}$$

where $\mathbf{w}_k^i$ is a vector with normally distributed random values, and $\mathbf{B}\mathbf{B}^{\mathrm{T}}$ is the process noise covariance matrix [12]. It would be very interesting to create a model of the kinematics of an infant's head and joint movement in order to create a similar model description. However, due to time-constraints, a simpler method was implemented.

The Simplest Case

The simplest functional prediction is given as:

$$\tilde{\mathbf{x}}_k^i = \mathrm{diag}(\boldsymbol{\sigma})\mathbf{w}_k^i + \overline{\mathbf{x}}_{k-1} \tag{3.32}$$

Which is a diffusion around the previous state output, making the preceding selection stage obsolete ($\hat{\tilde{\mathbf{x}}}$ was not used). $\mathbf{w}_k^i$ is a vector containing normally distributed random values, which are scaled according to the elements in $\boldsymbol{\sigma}$. These values determine the standard deviation of the added noise, which is added to each state parameter of the previous output.

$$\boldsymbol{\sigma} = [\sigma_x, \sigma_y, \sigma_z, \sigma_{R_x}, \sigma_{R_y}, \sigma_{R_z}, \sigma_{R_x}] \tag{3.33}$$

The standard distribution vector, $\boldsymbol{\sigma}$, can be estimated through measurements in the video stream, or using data from the recordings as mentioned in section 1.4. Although this is a crude way of performing prediction (if "probably close" can be called a prediction at all), it has demonstrated to work reasonably well. This prediction will be refered to as *no prediction.* Better methods will be discussed next.

The General Case

In the general case, an arbitrary function, $\mathbf{f}$, generates a new hypothesis, $\tilde{\mathbf{x}}_k^i$, based on knowledge of the previous output history, $\overline{\mathbf{x}}_{1:k-1}$, and current selected particle $\hat{\tilde{\mathbf{x}}}_k^i$. It can be written as:

$$\tilde{\mathbf{x}}_k^i = \mathbf{f}(\hat{\tilde{\mathbf{x}}}_k^i, \overline{\mathbf{x}}_{1:k-1}) + \mathrm{diag}(\boldsymbol{\sigma})\mathbf{w}_k^i \tag{3.34}$$

[6]A linear autoregressive (AR) model can be used to model many systems, and is defined as $X_t = c + \sum_{i=1}^{p} a_i X_{t-i} + \epsilon_t$

By numerically calculating differentials on the outputs $\overline{\mathbf{x}}$, Runge-Kutta methods can be used to perform prediction. For instance, a first-order prediction would be:

$$\tilde{\mathbf{x}}_k^i = 2\overline{\mathbf{x}}_{k-1} - \hat{\hat{\mathbf{x}}}_k^i + \mathrm{diag}(\boldsymbol{\sigma})\mathbf{w}_k^i \tag{3.35}$$

Which in a sense is the assumption of linear displacement of the system state. It will be refered to as *linear prediction.*

3.5.3 Implemented Approach

After the selection stage, particles are considered equal. This allows for easy simultaneous implementation of multiple prediction methods. For instance, 40% of the particles could be relocated using no prediction, while the remaining particles could can perform linear prediction.

Chapter 4

Results

4.1 Testing Setup

4.1.1 Implementation Bug

In the current version of the implementation, a bug has remained unresolved which causes system to perform inadequately when allowing for rotation in R_x and R_y. That is when the subject rotates in such a way that the face no longer is directed towards the camera. It is the authors firm belief that this bug can be resolved with a very few lines of code, as all required elements, such as *dynamic template*, *model based template generation*, and *template rotation* are implemented and functional. However, because of limitations in time, it was deemed more important to show the functionality of the system, focusing on the performance in tracking in dimensions T_x, T_y, S and R_z, which do not rely on the parts of the system where the bug is present. Tracking in these four dimensions should also not be a task taken too lightly.

In order to test the individual capabilities and performance of the tracking system, controlled test cases needed to be created. This involves creating a video with a subject head conforming to a pre-determined sequence of values for T_x, T_y, S and R_z. This would allow a direct comparison between true values, and tracked values. An attempt was made at making the test cases somewhat similar to the video data of the infants (mentioned in section 1.4), and as realistic as possible.

To improve readability, data plots are mostly gathered in appendix A.1, and will be referenced through this chapter.

4.1.2 Motion Data Creation

To generate a sequence of motion data (position, scale or z-rotation), a program was created to record the mouse movement. This way a more natural flow of data was achieved, with different properties along the way. For instance, data created for position, $[T_x, T_y]$ is shown below in figure 4.1.
Similarly, data was created for scaling, S, and z-rotation, R_z.

4.1.3 Video Creation

Using the generatad data, an image of a head[1] was moved around, scaled and rotated to match the data accordingly. Using alpha-channel information, the

[1]The image used is not from the CIMA database, to avoid redaction, and thus allow easier demonstration

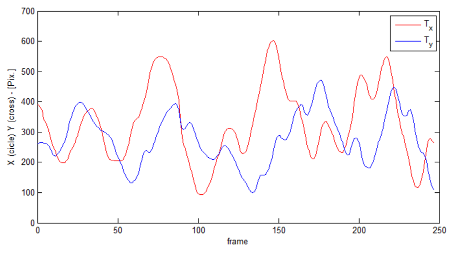

Figure 4.1: Generated movement data for position, $[T_x, T_y]$

head makes an immediate transition to the background. This is shown in figure 4.2.

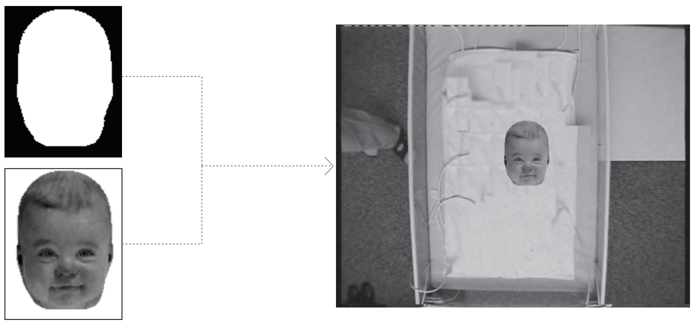

Figure 4.2: Alpha channel is used to properly overlay the head image onto the background

The background can be set as desired, and is used to test the effect of different backgrounds. The fact that this background is static is not as big a simplification as one might think. For instance, if the tracking was done with optical flow, it would be an enormous simplification to keep the background static. However, since this method is based on template matching alone, this is considered good enough.

4.1.4 Occlusion

Using a foreground image with accompanied alpha channel, it is possible to simulate the effects of occlusion. The intensity value of the occlusion is set to gray clutter, similar in intensity as skin (as hands and feet are the main source of occlusion).
The different kinds of occlusions are grouped by three key characteristics: *partial-*, *passing-* and *complete occlusion.* A description of each follows.

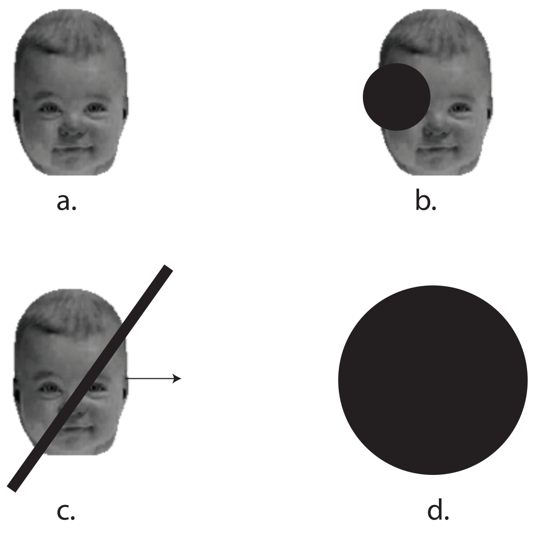

Figure 4.3: Illustration of the different kinds of occlusion considered. **a.** No occlusion **b.** Partical occlusion **c.** Passing Occlusion **d.** Complete Occlusion

Partial Occlusion

Partial occlusion, as the term indicates, occurs when part of the head is hidden. See figure 4.3b. At no point is the face completely occluded, and furthermore, it is implied that passing occlusion does not occur (see next point). The foreground image passed through the alpha channel is shown in figure 4.4.

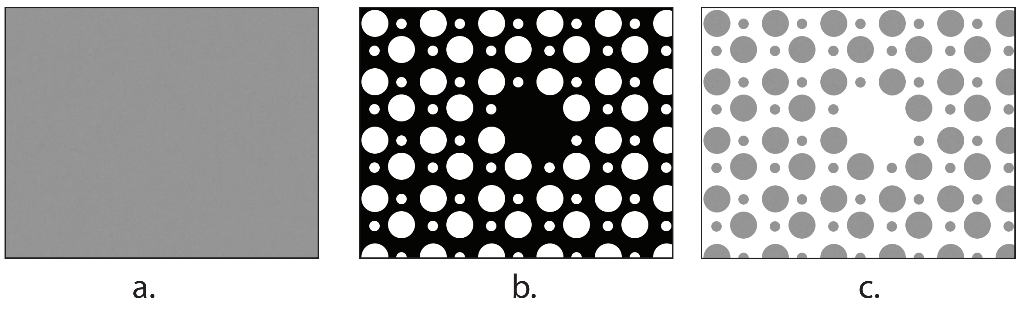

Figure 4.4: **a.** Foreground image used for in partial occlusion. **b.** Alpha map. **c.** Combination of the two. White pixels here represent transparency

Passing Occlusion

Passing occlusion is a form of partial occlusion, however the key difference is that the face passes by an occlusion such that the whole face has been occluded, however not fully at any point in time (see figure 4.3c). Many point based tracking systems fail at this test. Foreground image used shown in figure 4.5.

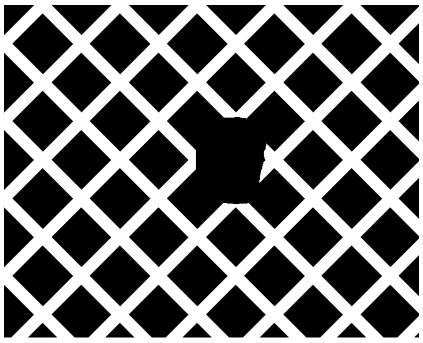

Figure 4.5: Alpha map used in passing occlusion. White pixels denote transparency

Complete Occlusion

The last type of occlusion is called *complete occlusion.* At some point in time, the face is completely occluded, as illustrated in figure 4.3d. The foreground image used is shown in figure 4.6 and was created such that the head movement resulted in complete occlusion.

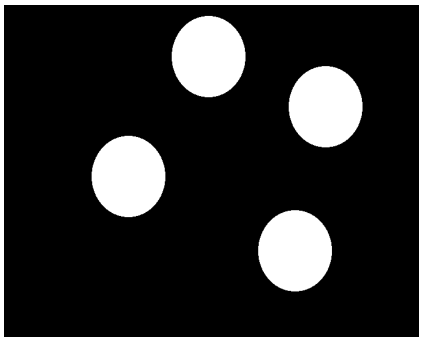

Figure 4.6: Alpha map used in complete occlusion. White pixels denote transparency

4.1.5 Noise Creation

It is desirable to test the robustness towards noise in the video stream. This noise was extracted from the original CIMA videos. By using a region in the video which contained static scenery, the change in pixels from one frame to the next corresponded purely in noise elements. The noise was extracted by first acquiring the static scene by averaging over a long period of time; creating a noiseless sub-image. By subtracting this noiseless static image, only noise elements remain.

4.2 Tracking in 2D

Initial testing starts out limiting the search space to two dimensions, $[T_x, T_y]$. Different properties will become apparent and potential weaknesses will be made highlighted. It starts out by determining the effects of the background, the addition of noise, and different levels of occlusion.

4.2.1 Simple Background

To test the potential accuracy given near-perfect conditions, we start out with a very simple case of constant monotone background (see figure 4.7). Regarding the prediction, the most rudimental method is used (equation (3.32)). The initialization starts with the user placing the 3D head model on top of the face to track. After this, the system tracks until the completion of the video, and writes the output pose for each frame to a file, which is compared with the original motion data.

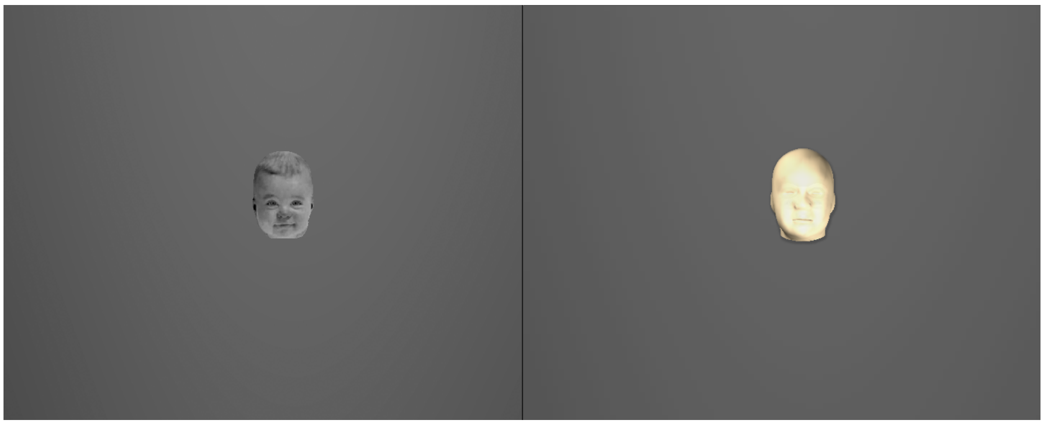

Figure 4.7: The first frame from the *simple background* test video, and the model initialized shown in the right

Because the initial position the user considers (when initializing model position) to be the center of the head might differ from the actual motion data, calibration is done by subtracting the difference from the first output. In other words

$$\overline{[T_x, T_y]}^{\text{calib}}_{1:k} = \overline{[T_x, T_y]}_{1:k} - [\text{Offset}] = \overline{[T_x, T_y]}_{1:k} - (\overline{[T_x, T_y]}_1 - [T_x, T_y]_1) \quad (4.1)$$

Figure 4.8 shows the comparison between the true state $\mathbf{x}$ and calibrated tracking output $\overline{\mathbf{x}}$.

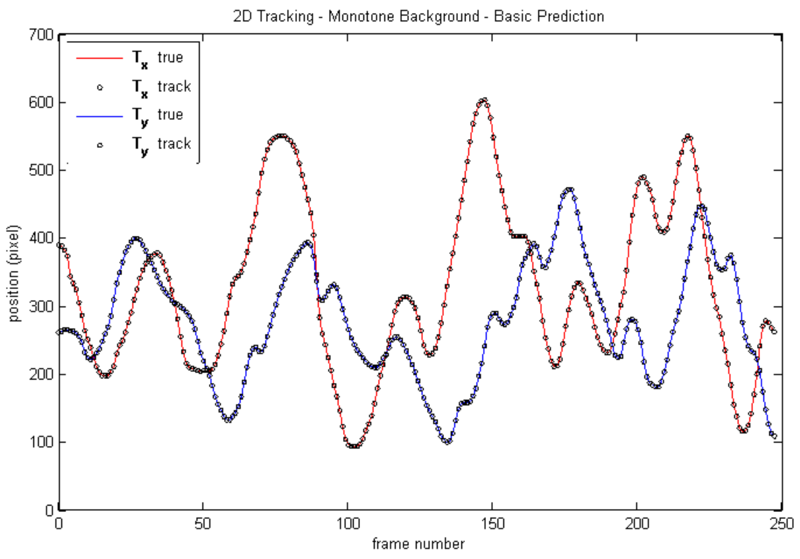

Figure 4.8: Plot of 2D position tracking sequence

It was found to perform very well. The difference plot, $\overline{\mathbf{x}} - \mathbf{x}$, shows that the tracking is for the most part within the correct pixel, i.e. higher accuracy cannot be attained. There are about 20 frames where the tracking erred with a few pixels.

By using the slightly more advanced prediction method (equation (3.34)) for 60% of the particle cloud, an even lower error-rate can be achieved as showed in figure 4.10.

Since the movement data is given in integers, anything within a sub-pixel accuracy (below the dotted line, $y = 0.5$) is considered to be a perfect match. This can be demonstrated by rounding the output data to the nearest integer. The corresponding difference plot shown in figure 4.11.

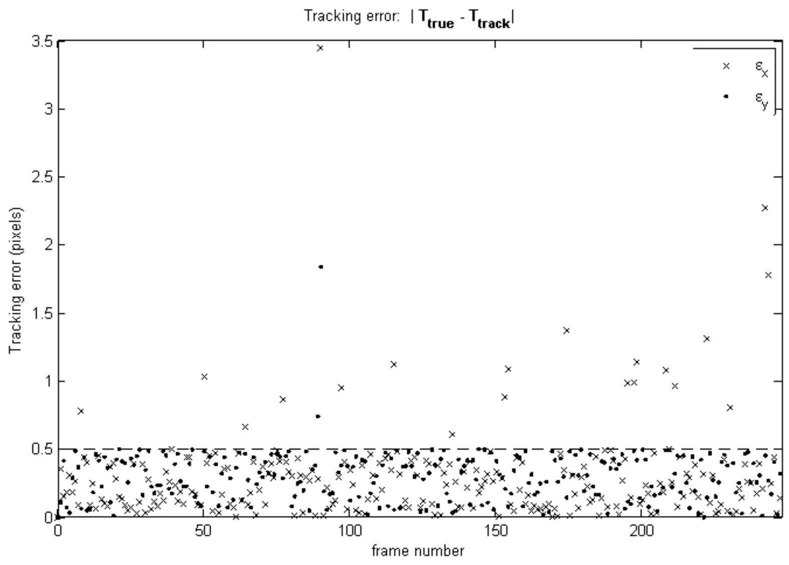

Figure 4.9: Difference plot $\overline{\mathbf{x}} - \mathbf{x}$, monotone background.

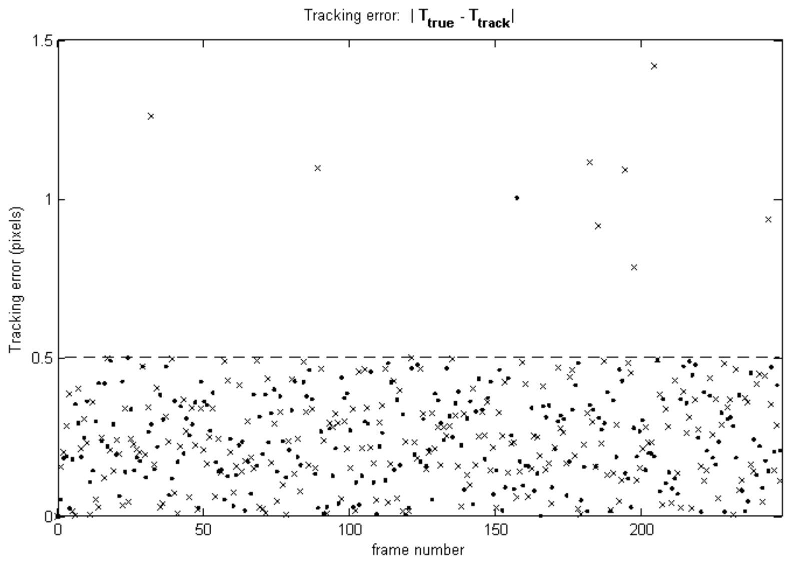

Figure 4.10: Difference plot $\overline{\mathbf{x}} - \mathbf{x}$, monotone background. Linear prediction method for 60% of cloud

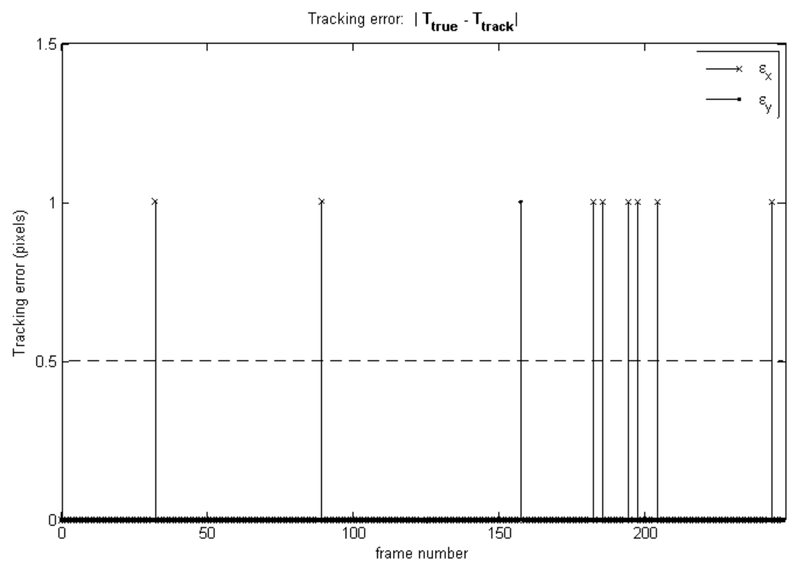

Figure 4.11: Difference plot `round(x̄)` − **x**, monotone background. Linear prediction method for 60% of cloud

It shows that in only 9 out of the 248 frames, the tracking erred by a single pixel. In other words, given these pristine conditions, the tracking method works almost perfectly. The performance of the tracker can be measured in terms of average pixel offset pr frame, which will be called the *tracker performance*, $\boldsymbol{\varepsilon}_{\mathrm{TP}}$. In other words

$$\boldsymbol{\varepsilon}_{\mathrm{TP}} = \tfrac{1}{k}\sum_{i=1}^{k} |\texttt{round}(\overline{\mathbf{x}})_i - \mathbf{x}_i| \tag{4.2}$$

For the case above, $\boldsymbol{\varepsilon}_{\mathrm{TP}} = 9/248 \approx 0.03629$.

4.2.2 Regular Background

To give the tracking a bit more of a challenge, the same test was done, but with a non-uniform background. A sample frame is shown in figure 4.12.

The tracking results are very similar to that of the monotone background test case. The performance is shown in figure A.2 in the appendix. Tracker performance is $\boldsymbol{\varepsilon}_{\mathrm{TP}} = 0.04435$.

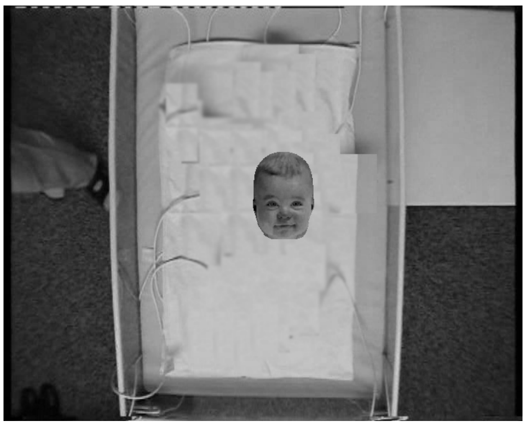

Figure 4.12: Example frame from none fixed background image, 2D tracking test

4.2.3 Similarity-Cluttered Background

To create a serious challenge for the tracking system. The following background image is used:

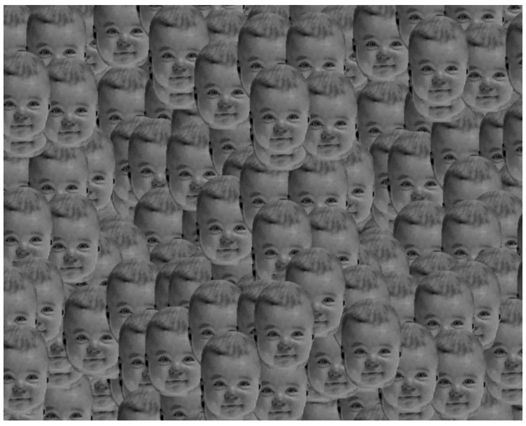

Figure 4.13: Example frame from cluttered background image, 2D tracking test. Severe degree of clutter.

This contains several copies of the template itself, creating several highly evaluated points. Using no prediction, the tracking fails, and latches onto the surrounding faces. It resumes tracking when the head approaches again. This is shown in figure 4.14.
If the same test was done with the basic linear prediction, it again works perfectly, regardless of background clutter. The two cases (no prediction and linear prediction) are shown in figure A.3 and A.4, respectively.

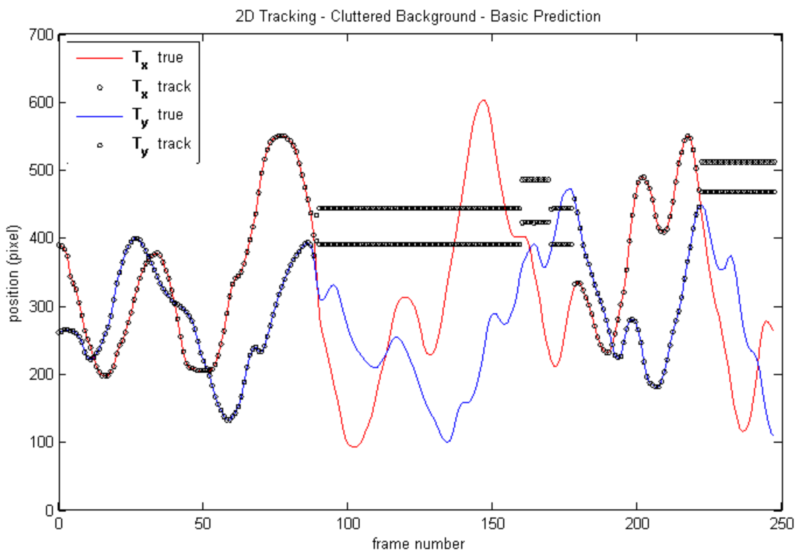

Figure 4.14: Plot of 2D position tracking sequence. No prediction used. It loses track of the correct face between frames 90 and 179, and again at frame 223. $\varepsilon_{\mathrm{TP}} = 164.21$

4.2.4 With Noise

The examples thus far have been very ideal, in that they contained no noise elements. For these tests, we will continue with the more realistic scenario of a fixed background. Noise was extracted from the CIMA video files, such that it matches the statistical properties identically. Noise intensity is measured relative to the CIMA videos. 1X denotes the same intensity. The tracking was tested against multiples of this noise. Although it is easier to see noise when animated, figure 4.15 tries to show the effect of different noise levels added onto a image.

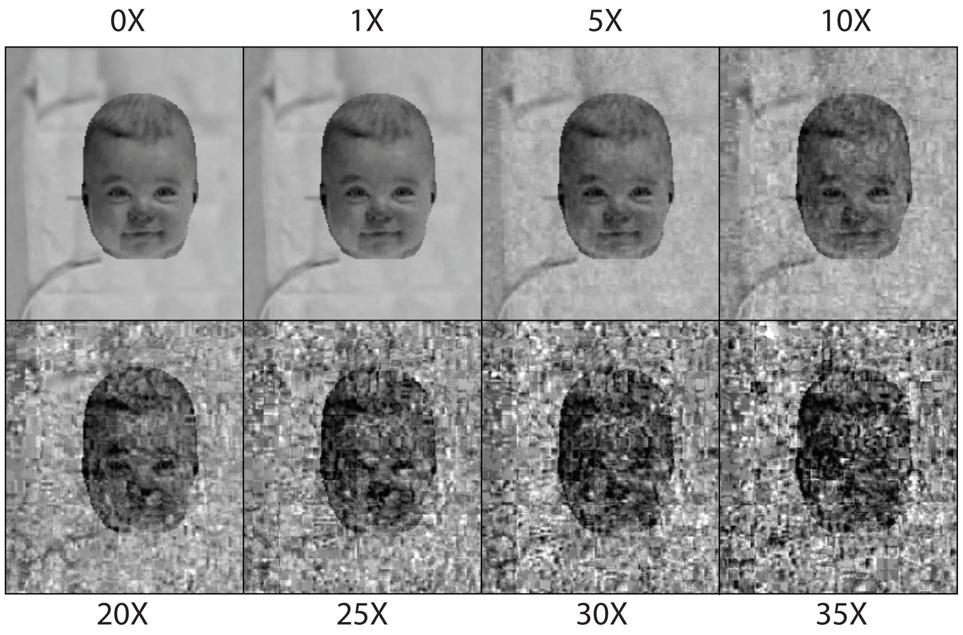

Figure 4.15: Demonstrates the effect of noise at different factors.

Noise Factor	**1X**	**10X**	**20X**	**25X**	**30X**	**35X**
$\varepsilon_{\mathrm{TP}}$	0.04839	0.44758	1.6532	4.6774	19.290	144.28
$\texttt{max}(\varepsilon)$	1	2	8	44	156	508

The individual tracking cases are shown in figures A.5, A.6, A.7, A.8 and A.9 for 1X, 10X, 20X, 25X, 30X and 35X respectively. Noise levels above 30X cause the tracking to fail, as the noise to signal ratio is simply too great for the system to overcome. However, tracking the subject at noise levels of up 30X is fairly impressive. Very few tracking algorithms would perform under such conditions. To get a perspecive of how significant this noise is, the reader is encouraged to view the demo videos in the accompanied CD. This is also an indication of robustness towards change in facial expressions. Next follows a series of occlusion robustness tests.

4.2.5 Partial Occlusion

By setting a foreground image with regions of transparency, it is possible to assess how robust the system is to different kinds of occlusion. First off, partial occlusion will be tested. The foreground image is chosen as neutral values (127), to make it more realistic as it is close to skin color. Furthermore, because the particle system is based on template matching, neutral values are more problematic than say completely black values. The alpha mask and resulting foreground occluded frame is shown in figure 4.16. There is a deliberately made hole in the foreground, as the initialization starts with the assumption of no occlusion. To make it even more of a challenge, 5X noise has been used.

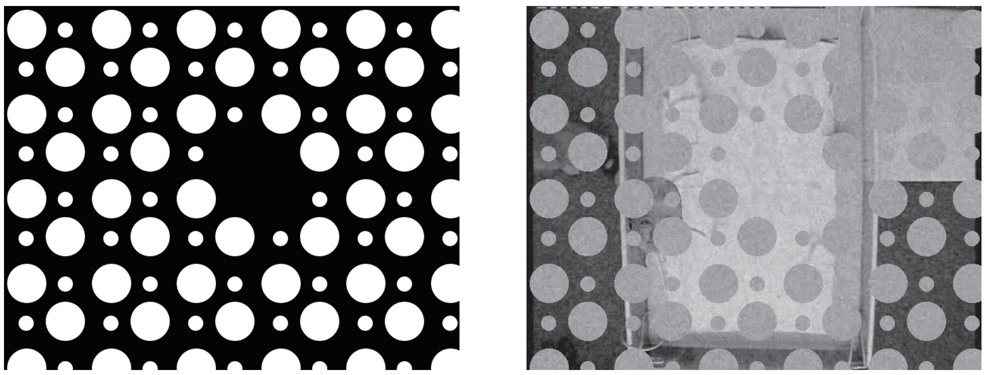
Figure 4.16: Partial occlusion map shown on the left. Sample frame on the right

The degree of occlusion and noise is significant, however, as can be seen on from the result in figure A.11, this had very little effect on the outcome. Average error was $\varepsilon_{\mathrm{TP}} = 0.68548$ pixels.

4.2.6 Passing Occlusion

A traditionally harder type occlusion is what was termed *passing occlusion.* The example alpha map used for the foreground passing occlusion is shown below in figure 4.17

Tracking sequence shown in figure A.12. The higher average error $\varepsilon_{\mathrm{TP}} = 1.9596$ is due to a few frames where the tracker lost track. As can be seen

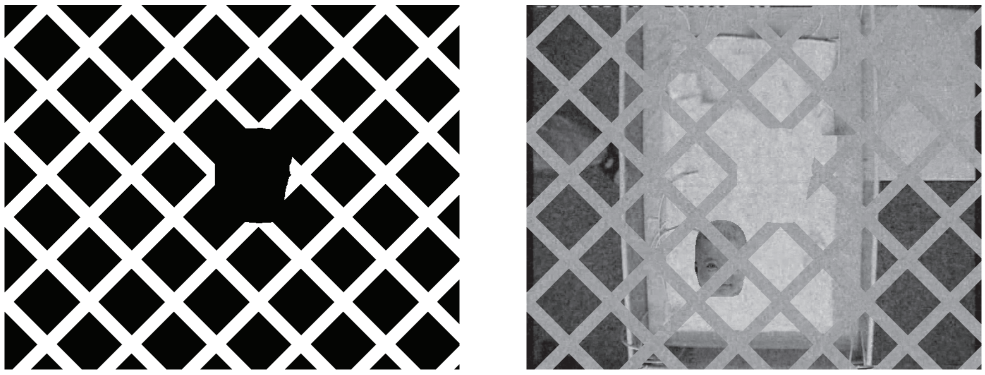
Figure 4.17: Passing occlusion map shown on the left. Sample frame on the right

from the sequence, in only 19 frames was the tracking error greater than 1 pixel. In these cases, the tracking resumes high performance within the next frames.

4.2.7 Complete Occlusion

Under the circumstance of complete occlusion, tracking is by definition impossible (as it cannot be seen). However, it is of interest to see how the system acts under such conditions, and whether it can recuperate. The alpha map used in this instance is shown in figure 4.18, which in certain frames, covers the head completely.

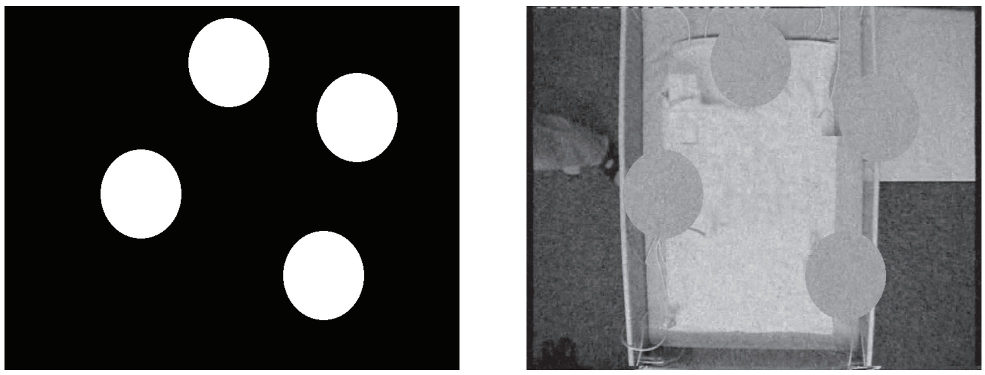
Figure 4.18: Complete occlusion map shown on the left. Sample frame on the right

Tracking sequence is shown in figure A.13. The failure occurs when the face becomes fully occluded, and emerges again at a point too far from where it lost track. The only way it can manage to resume good tracking, is for the state to approach the point where it lost track, in a state of no occlusion. This can be seen happen at around frame 50, and again later at around frame 125. It is thus apparent that to account for complete occlusion, some additional logic needs to be added to the system – detecting the occurrence of failed tracking, and counter measures to facilitate the resuming of good tracking. For instance, spreading out the particle cloud upon losing track, can help to improve this issue.

4.3 Tracking in 2D + S

In the previous section, the search space was two dimensional, as only template position varied. In this section, the added dimension of scaling will be tested. As mentioned in previous sections, the scaling can be seen as the reciprocal of the distance to the camera. For perspective projection, this is a fairly good simplification, while in orthogonal projection, the two are exactly equivalent. Testing will not cover all the test cases that were shown for pure 2D tracking.

4.3.1 Basic Test

To start out with a simple case, only 1X noise is used and no occlusion. Motion data was generated for scaling values, in the same way as for the tracking. The generated values range from 0.42 to 1.60, and is shown here in figure 4.19.

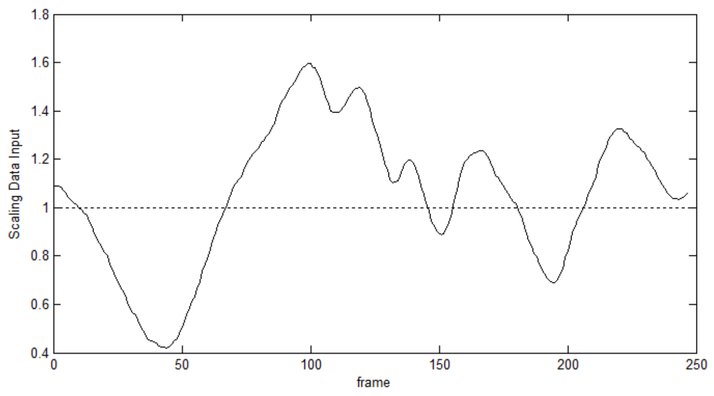

Figure 4.19: Plot of motion data for scaling used in testing

It is necessary to calibrate the scaling output of the tracker, as the user decides the initial scale, which might not be accurate in terms of data matching. This follows the same logic as in equation (4.1), used for calibrating scale:

$$\overline{S}_{1:k}^{\text{calib}} = [\text{Scaling Offset}] \cdot \overline{S}_{1:k} = \frac{\overline{S}_1}{S_1} \cdot \overline{S}_{1:k} \tag{4.3}$$

As with the 2D tracking cases covered until now, video was generated based on the motion data. This was inputted into the tracking system, which now allows for variance in scalem i.e. the search space has been extended to three dimensions, one of which represents scale. The first tracking case uses a 1X noise intensity, and no occlusion. A sample frame containing the low scaling value is shown in figure 4.20.

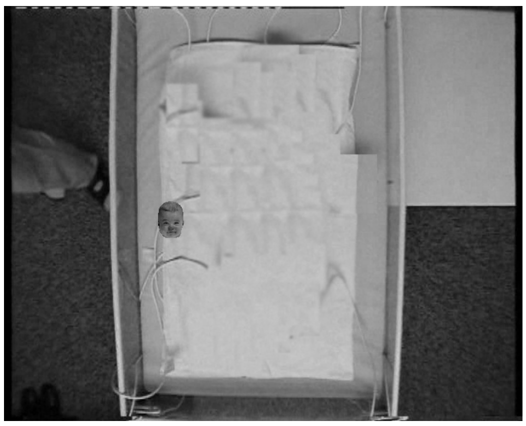

Figure 4.20: Sample frame of test video with varying scale.

As with the previous tests, a plot of the tracking sequence is given in the appendix, in figure A.14. The system performs well, with an average pixel position error of $\varepsilon_{\text{TP}} = 1.3387$.

4.3.2 With Noise and Occlusion

Next, 5X noise is added, and a combination of partial and passing occlusion. The new alpha map used is shown in figure 4.21.
Tracking sequence shown in figure A.15. The system still performs well under these circumstances. $\varepsilon_{\text{TP}} = 1.4919$.

4.4 Tracking in 2D + S + R$_z$

The performance of tracking rotation is tested in this section. The sample data sequence was created as before using the mouse as input, and is shown in

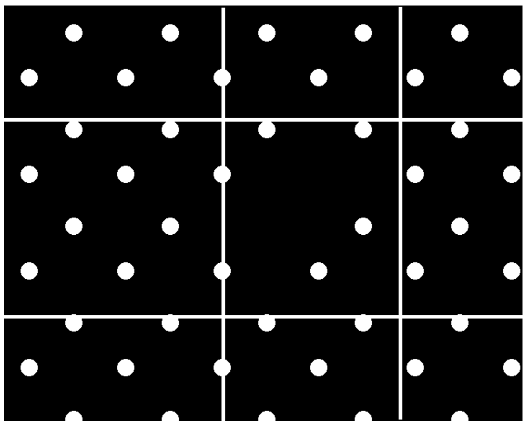

Figure 4.21: Masking image used for foreground occlusion in the subsequent tests.

figure 4.22. To avoid issues with calibration, the initial angle was deliberately set at 0 degrees. The values range from ±90 degrees.

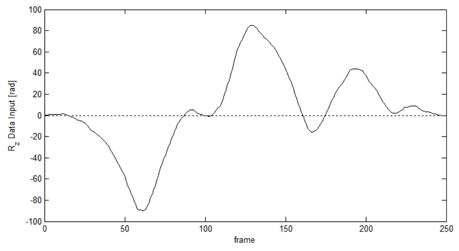

Figure 4.22: Plot of motion data for R$_z$ rotation used in testing.

4.4.1 Basic Test: 2D + R$_z$

Starting off with a simple case of 1X noise, no occlusion, and no scaling. The tracking can be seen in figure A.16. It should be noted that the position error is occurring mostly in the horizontal x-direction, and highly correlated to the rotation angle. This should not be the case, and is believed to be a symptom of the unresolved bug mentioned in section 4.1.1. When angles are present in tracking, there is a seeming discrepancy between the internally calculated particle position, and what is shown on the output. There is in other words an added error due to the angle itself, which although present in the output, does not severely compromise the internal functionality of the tracking (this cannot be said for R_x and R_z, which compromise tracking significantly).

Although the exact consequences of the bug remains undetermined, a rudimentary correction can be made by redefining the x-position output as:

$$\hat{\overline{T}}_x = \overline{T}_x + 110 \cdot \overline{R}_z \tag{4.4}$$

The corresponding plot with this correction is shown in figure A.17. Using this simple correction, the average error dropped from $\varepsilon_{\text{TP}} = 20.254$ to $\varepsilon_{\text{TP}} = 4.0161$. In either case, the error in rotation is consistently low, with an average error of 0.0165 radians, less than 1 degree. The maximum error was 4.0 degrees.

4.4.2 Full dimension: 2D + S + R$_z$

Adding variation of scaling to the mix, the result is shown in figures A.18 and A.19. In the latter, the translation of x was corrected for using the method shown in equation (4.4), with a different constant. The angular correction is no longer sufficient to account for the error in horizontal tracking. This indicates that the scaling might also impart an adverse effect. Regardless of the tracked position, both scaling and rotation perform well.

4.4.3 With Noise and Occlusion

The final testing case adds 5X noise and foreground occlusion using a map shown in figure 4.23, which is similar to the one from the scaling test. Tracking sequence is shown in figure A.20.

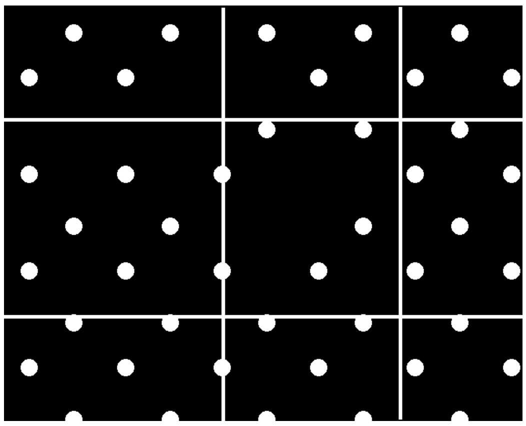

Figure 4.23: Alpha map used for the foreground occlusion in the test video

Tracking in four dimensions, with 5 times regular noise levels and foreground occlusion, the system performs very well. As mentioned, the error in horizontal positioning is mainly due to a bug, which although present in the output, does not affect the tracking itself. That is, if the particle cloud was internally evaluated about the suggested horizontal coordinate, the tracking would fail early, and certainly affect all other state outputs, and not just the horizontal positioning. This suggests that the horizontal tracking error paints an unjust picture as to the potential of the system, as it is simply a matter of calibration, rather than poor tracking. The expected case would be a similar performance for both horizontal and vertical position.

4.5 Overview of Tracker Performance

The performance of the system in each of the discussed tests is summarized in table 4.1.

Fig.	Track.	Noise	OCC	TX.	BG.	Pd.	$\varepsilon_{\mathrm{TP}}$	*aex*	*aey*	*mex*	*mey*	*aes*	*mes*	*aer*	*mer*
A.1	2D	0X			M	L	0.036290	0.032258	0.0040323	1	1	-	-	-	-
A.2	2D	0X			F	L	0.044355	0.000000	0.0443548	0	1	-	-	-	-
A.3	2D	0X			C	N	164.2096	78.89112	85.318548	395	358	-	-	-	-
A.4	2D	0X			C	L	0.012096	0.008064	0.0040322	1	1	-	-	-	-
A.5	2D	1X			F	L	0.048387	0.044354	0.0040322	1	1	-	-	-	-
A.6	2D	10X			F	L	0.447580	0.391129	0.0564516	2	1	-	-	-	-
A.7	2D	20X			F	L	1.653226	1.201613	0.4516129	8	3	-	-	-	-
A.8	2D	25X			F	L	4.677419	3.419355	1.2580645	30	44	-	-	-	-
A.9	2D	30X			F	L	19.29032	10.22581	9.0645161	108	156	-	-	-	-
A.10	2D	35X			F	L	144.2863	82.17742	62.108871	508	322	-	-	-	-
A.11	2D	5X	PT		F	L	0.685484	0.463710	0.2217742	34	15	-	-	-	-
A.12	2D	5X	PS		F	L	1.959677	0.733871	1.2258064	34	63	-	-	-	-
A.13	2D	5X	C		F	L	270.9960	110.0081	160.98790	444	422	-	-	-	-
A.14	2D S	1X			F	L	1.338710	0.612903	0.7258065	2	2	1.116875	4.014505	-	-
A.15	2D S	5X	PS PT		F	L	1.491935	0.616935	0.8750000	3	3	1.424960	6.719985	-	-
A.16	2D R	1X			F	L	20.25403	19.05645	1.1975806	48	4	-	-	0.947638	3.961955
A.17	2D R	1X		Y	F	L	4.016129	2.818548	1.1975806	9	4	-	-	0.947638	3.961955
A.18	2D S R	1X			F	L	21.94758	20.83065	1.1169355	78	6	3.106650	20.04628	1.700285	7.763289
A.19	2D S R	1X		Y	F	L	6.895162	5.778226	1.1169355	23	6	3.106650	20.04628	1.700285	7.763289
A.20	2D S R	5X	PS PT		F	L	21.06855	19.84274	1.2258064	71	9	2.363958	12.26053	1.548891	9.453386
A.21	2D S R	5X	PS PT	Y	F	L	6.665326	5.439516	1.2258065	33	9	2.363958	12.26053	1.548891	9.453386

Table 4.1: Overview of tracking performance in each test case.
Fig: Reference to figure number.
Track. Dimensions of tracking. *2D* (x,y). *S* (scale). *R* (R_z rotation)
Noise. Factor of noise used. 1X denotes same noise as in CIMA videos. 2X is twice the intensity, etc.
OCC Occlusion. *PT* (Partial Occlusion). *PS* (Passing Occlusion). *C* (Complete Occlusion)
TX. Partial correction for T_x error introduced by system bug. *Y* (yes). *N* (no). Only applicable for *R*.
BG. Background used. *M* (monotone). *F* (fixed regular image). *C* (high-clutter image).
Pd. Prediction method used. *N* (none). *L* (linear prediction).
aex / *aey*: Average error in horizontal/vertical tracking
mex / *mey*: Maximum error in horizontal/vertical tracking
aes / *mes*: Average and maximum error in scale. Numbers given in percentage (i.e. 100 times true value).
aex / *mex*: Average and maximum angle, given in degrees.

4.6 Speed Performance

The performance in terms of accuracy has been covered in the previous sections. However, no mention has been made of the speed in which the tracking is computed. The implemented method allows for easy modulation of tracking accuracy by altering the number of particles, or the number of feature points. In other words, it is a highly customizable method, allowing for adjustable trade-off in tracking accuracy vs. computational speed. That said, the method requires high level of computation, which has in large been transferred to the graphics card's GPU, functioning as a co-processor.

Throughout the testing examples discussed, the number of particles was kept constant at 11 000. For each iteration in the tracking, each of these particles performed a 3-axis rotation, scaling and translation of 550 feature points. With this high number of particles and FPs, the system performed at a consistent pace, with an average of 280 ms pr. frame.

Even though tracking was performed for a subset of the state space, the implementation performed the computation as if the full state-space was being explored. To be precise, only the standard deviation for the diffusion stage for the different parameters in the state space was set to 0. This causes the particle to remain in the same position along the corresponding dimension.

This might seem like a waste of computation for the special cases of selective search, which, would be a correct assessment. However, the particular task for which this method was developed employs the full range of the state space, and as such cannot take advantage of selective search. Nevertheless, it would not be problematic to modify the implementation such that it takes advantage of the special cases.

4.6.1 GPU Contribution (vs. CPU)

The main computational effort with particle filters is the evaluation of the particle cloud. Through use of GPGPU methods and OpenCL, this was outsourced to the GPU. Since the evaluation for each particle is done independently, this task becomes highly parallelizable, and thus well suited for the GPU hardware architecture. To demonstrate the significance of the GPU contribution, a separate implementation was created for evaluating particles in the CPU, and not relying on the GPU at all.

Using 2k particles, 550 FPs, the CPU implementation spent an average of 1060 ms per frame. Performing the exact same tracking using the GPU took an average of only 66 ms, which is 16 times the performance. The advantage of the GPU over the CPU lies in parallelization. Repeating the same test but with 5k particles, the speed is 123ms to 2817ms – a factor of approximately 23. Consider a simulation task which takes one week to complete on a GPU implementation of a particle filter of this size. The equivalent CPU solution would take almost half a year!

It is important to note that the system on which these tests are made, is a soon two year old personal consumer laptop. It runs on an Intel Core2Duo 2.26Ghz processor and an NVidia GeForce 9600M Gs graphics card. This graphics card was one of the earliest models to be released with CUDA support. That said, the GPGPU has matured considerably in the past two years, such that the difference between the GPU and CPU performance is now only bound to be greater.

4.6.2 Consistency in Computation Time

A good quality of the method is consistency in computation time. The task for each frame is very much predetermined throughout the sequence, regardless of the accuracy of tracking, or variations through the video stream. The timing consistency was shown to be significantly higher for the GPU implementation than the CPU implementation. Since the main work load is transferred to the GPU, while the CPU waits for the evaluation to be complete, it can perform other duties. For the CPU only implementation, it is much more affected by other processes running. Figure 4.24 shows the timings for processing one of the earlier test cases.
The standard deviation for each case is 42 ms for the CPU and 1.64 ms for the GPU. If the GPU times were scaled to match the average of the CPU, the standard deviation of the GPU would still be half of that of the CPU.

Performing the same test with 5k particles resulted in standard deviation times of 80 ms and 5.89 ms for the CPU and GPU respectively.

4.6.3 Memory Consumption

A quick test of the program showed consistent memory usage, indicating no memory leakage. The memory consumption for the program using 11k particles and video stream of dimensions 720x520, was kept constant at 52 MiBs.

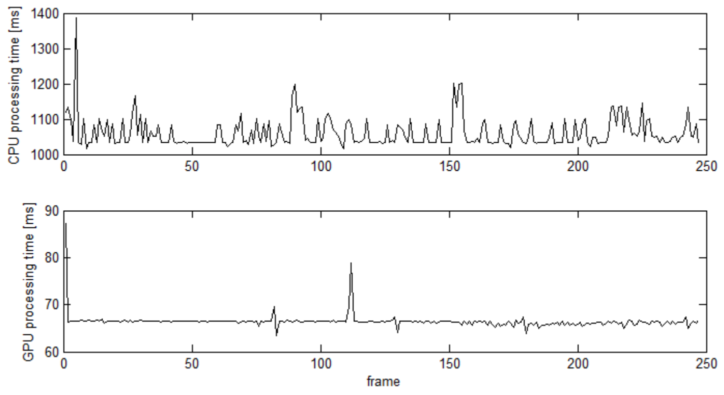

Figure 4.24: Computation time pr frame for CPU and GPU, 2k particles 550 FPs.

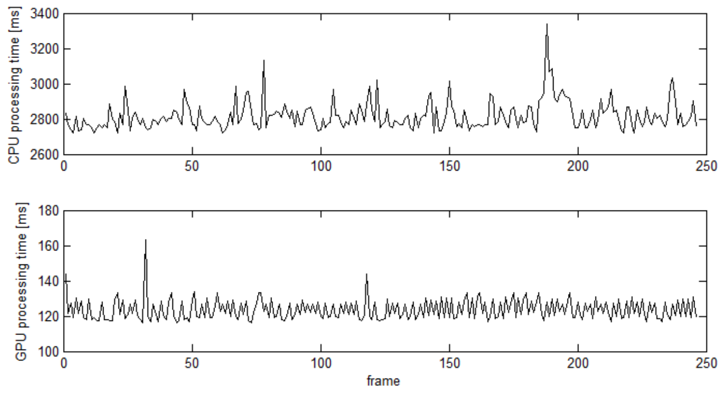

Figure 4.25: Computation time pr frame for CPU and GPU, 5k particles 550 FPs.

4.7 Dynamic Template

Since the dynamic template is mostly used for the case of change in R_x and R_y, showcasing the performance in terms of quantitative measurements becomes difficult. However, a visualization of the template and dynamic creation of FPs is provided in the CD accompanying this thesis. A selected number of frames during this process is shown in figure 4.26.

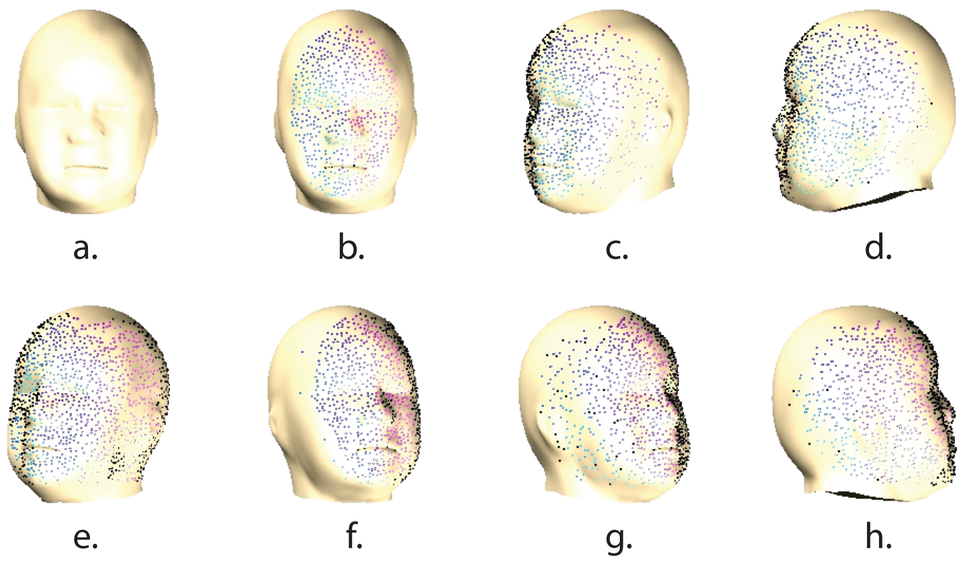

Figure 4.26: Selected frames during the demonstration of automatic generation of FPs. **a.** No points initialized. **b.** User input of initial pose complete. First batch of FPs created. **c.** Rotation of template. Black FPs are bad. New FPs created and shown. **d.** Almost all of the initial FPs are considered bad. If not for dynamic template, tracking would most likely fail. **e-h.** Rotation back shows few FPs generated when enough good FPs are present.

Chapter 5

Concluding Remarks

5.1 Conclusion

This section is devoted to summarize and freely discuss the results and potential of the implemented tracking system.

5.1.1 Accuracy & Robustness

Accuracy

The test cases showed that the tracking system is capable of accuracy down to the pixel level, achieving performances of under 0.04 average pixel error when tracking in two dimensions. When tracking in four dimensions, with added noise 5 times that of regular strength, and adding considerable foreground occlusion, it performed within 7 pixels average position error, 1.6 degree average angle error and 2.4% average scaling error.

When dealing with rotation, an unaccounted error in the system caused the output estimation of horizontal position to err considerably. Although this gives the impression that the system performed very bad for estimating horizontal position when the state contained rotation, it is shown that this error is directly related to the rotation angle. Since this can significantly be corrected for without knowledge of the true rotation values, it implies that this error paints an inaccurate image of the potential performance.

Robustness to Noise

Testing was done to evaluate the system performance given varying degrees of noise. The noise was generated by extracting the noise components form the CIMA videos, and adding this on top if the generated videos, with varying levels of strength. The tracking algorithm showed considerable level of robustness towards noise, achieving successful tracking for noise levels up to 30 times the strength of the levels in the CIMA videos.

Robustness to Occlusion

Similarly, the system managed to perform successful tracking in spite of high levels of both partial and passing occlusion. However, upon encountering complete occlusion the tracking easily failed. Recovery from the failure was achieved only if the head re-approached the state point where the tracking was lost, and of course being non-occluded. For the case of the CIMA videos, the main source of occlusion comes from limbs passing over the face. This kind of occlusion is dynamic as opposed to the static occlusion in the test videos. It is suggested that the system can more easily recover from dynamic occlusion, as the occlusion itself will eventually disappear. If the particle cloud remains around the point where it lost track, the true pose is likely to be nearby.

5.1.2 Speed

The OpenCL implementation showed a 21X speed increase over the pure CPU version. This allows for adequate particles quantities to be processed in real time. For instance, using 5k particles and a template with 550 feature points, the system can process each frame on an average of 123ms, or 8.13 frames per second. For 2k particles, it reaches an average of 15 frames per second. It should be mentioned that very little effort has been made in optimizing the different routines.

As has been shown, the performance in frames per second is very consistent throughout the tracking sequence. It is therefore possible to adjust the particle count and template complexity to match a desired frame rate.

The sparse template representation makes the system scale very well in terms of video dimension. The number of operations is mostly related to the particle and feature point count. If using local based methods for feature point creation (see section 3.3.3), the system performs almost independently of image dimension.

5.1.3 Accounting for Dynamics in System

The particle filtering system can easily take into account the dynamics of the tracked subject in the prediction stage. If done well, this can greatly improve the system performance by reducing the necessary particle count. For instance, in the case of cluttered background, the method failed using no prediction logic, while performed without fail using the very simple linear prediction.

5.1.4 Versatility and Customizability

The most attractive property of the implemented is customizability. The following is an overview of the most prominent customizable properties of the implementation.

Speed/Accuracy Trade-off

There is a direct trade-off between speed and accuracy when choosing the particle and feature point count. In other words, the tracking can be done to the desired level of accuracy, or conform to a desired throughput in frames per second.

Cost Function

The cost function itself has several parameters that can be set. The effect of different nuances in the cost function has not been explored to any significant extent. Upon doing so, this can also be tailored for optimum performance.

Subject Invariance

The system generates the template given the image itself and subject within the frame. This means that the system does not depend on color markers in order to function. More precisely, whether the markers are present or not, it is of little significance, as the creation of the template will be based on the subject itself.

Also, although the system has been designed for tracking head pose, there is no reason for it to not function similarly for any general tracking object, of somewhat rigid nature. With some modification to implementation, even rigidness is no longer a constraint, as the shape of the model can potentially be expressed in the state representation. For instance in the case of tracking a cylindrical object, a parameter could specify the cylinder's bend. Similarly, a leg can be somewhat modeled by two cylinders with the adjoining angle as a parameter.

System Dynamics

The prediction stage can take into account the dynamics of the tracked subject. As we've seen, only two rudimentary general prediction methods were used in the testing. There is great flexibility in how the prediction is performed. More importantly, any number of prediction methods can be used simultaneously. For instance use prediction method A for 20% of the particle cloud, and method B for the remaining particles. Additionally, the choice of prediction method can be changed dynamically throughout the tracking sequence. In other words, if one particular type of dynamic was believed to be dominating, the prediction best suited for representing this dynamic can be given more priority and thus determine prediction for a larger chunk of the particle cloud.

5.1.5 Final Words

The implemented method has been shown to perform remarkably well given the areas for which it has full functionality, showing high levels of resistance to occlusion and noise levels. It has taken use of GPGPU methods, allowing the system to perform in real time, where a pure CPU implementation would be significantly slower. Also, a benchmarking system has been created, allowing the controlled evaluation of the method. It is therefore in the author's humble opinion that the task this thesis was set to resolve, has been done so to satisfactory levels.

5.2 Future Work

At time of writing, the current system implementation suffers from a small bug, restricting functionality to tracking in the four dimensions – image position, scale and R_z rotation. The first priority would be to resolve this issue, which should not be a considerable effort, as the complete solution is fully implemented, it only needs to be corrected. This would allow to showcase the strength and full functionality of the developed dynamic template.

The following list is an overview of the additional different tasks which are suggested as a continuation on the work of this thesis:

- Test the fully functional tracking system directly on the CIMA videos.
- Invastigate the correlation between head movement and presence of CP in the CIMA project.
- Use the system a basis for recognizing facial expressions or gaze direction. Determine if correlation to CP.
- More accurately model the dynamics of the system, and subsequently incorporate said model into the prediction stage.
- Create logic for determining when tracking has failed, together with recovery modes/methods to improve search and chances for resuming tracking.
- Add functionality to dynamically adapt parameters based on the tracking itself. That is, methods for exploiting the high customizability of the method should be investigated to improve tracking based on a feedback system.

- Determine scalability of performance given more powerful hardware. OpenCL and other GPGPU methods are known for being highly scalable, it is interesting to investigate how so for this particular implementation.
- Test the system for different tracking subjects. For instance cars (traffic control), boats (offshore industry), other limbs like legs and arms (CIMA project),
- In general, use the GPGPU Particle System Framework for any number of state estimation problems relying on visual data.

Bibliography

[1] Roald Fernandez Cuesta. Model based head tracking and pose estimation in image sequences, 2009.

[2] Svein Arne Nesset. Estimering av spedbarns bevegelser ut fra videodata [nor]. Master's thesis, Norwegian University of Science and Technology, 2008.

[3] Odd Martin Staal. Robust video based motion tracking in young infants. Master's thesis, Norwegian Univsersity of Science and Technology, 2006.

[4] Parsa Rahmanpour. Features for movement based prediction of cerebral palsy. Master's thesis, Norwegian Univsersity of Science and Technology, 2009.

[5] George Butterworth and Edward Cochran. Towards a mechanism of joint visual attention in human infancy. *International Journal of Behavioral Development*, 3(3):253–272, September 1980.

[6] S. R. H. Langton, H. Honeyman, and E Tessler. The influence of head contour and nose angle on the perception of eye-gaze direction. *Perception & Psychophysics*, 2004.

[7] Oscar Mateo Lozano and Kazuhiro Otsuka. Real-time visual tracker by stream processing. *J. Signal Process. Syst.*, 57(2):285–295, 2009.

[8] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1992.

[9] Dave Shreiner. *OpenGL(R) 1.4 Reference Manual (4th Edition).* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[10] Arnaud Doucet, Nando De Freitas, and Neil Gordon, editors. *Sequential Monte Carlo methods in practice.* Springer, 2001.

[11] Arnaud, Doucet, and Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. Technical report, 2008.

[12] Michael Isard and Andrew Blake. Condensation - conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29:5–28, 1998.

[13] T. F. Cootes, G. J. Edwards, and C. J. Taylor. Active appearance models. *Lecture Notes in Computer Science*, 1407, 1998.

[14] Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for on-line non-linear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, 50:174–188, 2001.

Appendix A

A.1 Results - Plots

Plot A.1: 2D-tracking using linear prediciton, monotone background.

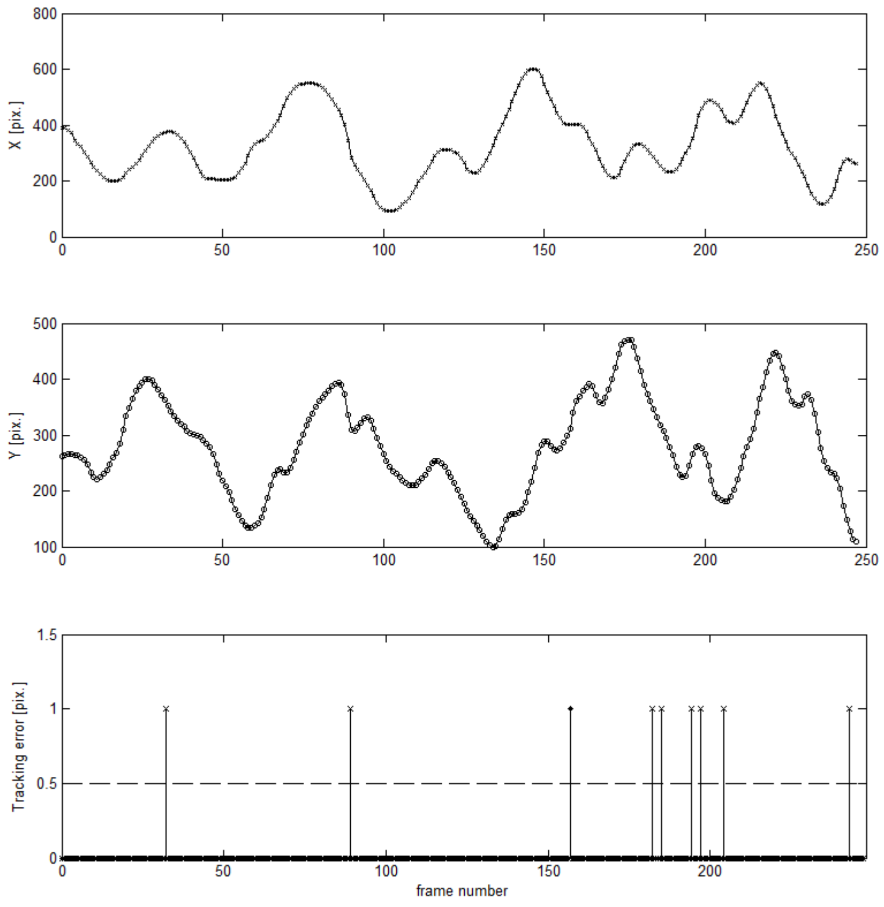

Figure A.1: 2D Tracking. Linear Prediction. Simplest of the test-cases. Main purpose to demonstrate maximum accuracy given pristine conditions. $\varepsilon_{\mathrm{TP}} = 0.03629$

Plot A.2: 2D-tracking using linear prediciton, fixed-image background.

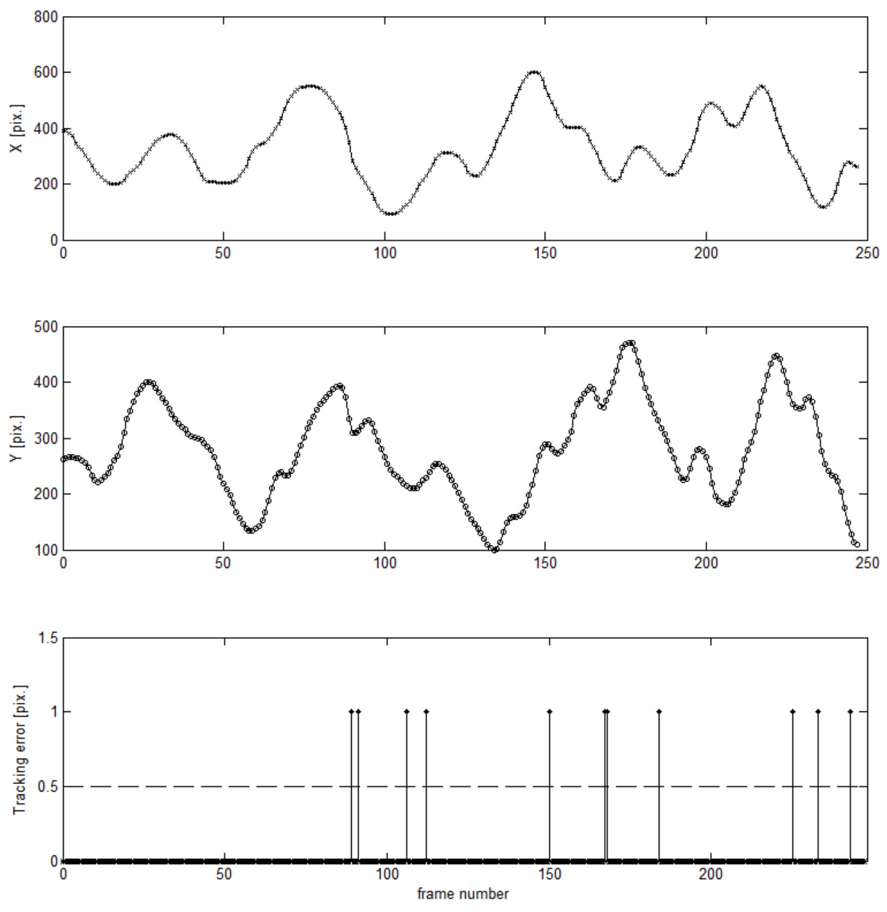

Figure A.2: 2D Tracking. Linear Prediction. Fixed image background. Purpose to see if background affects tracking. $\varepsilon_{\mathrm{TP}} = 0.044355$

Plot A.3: 2D-tracking using no prediciton, cluttered background.

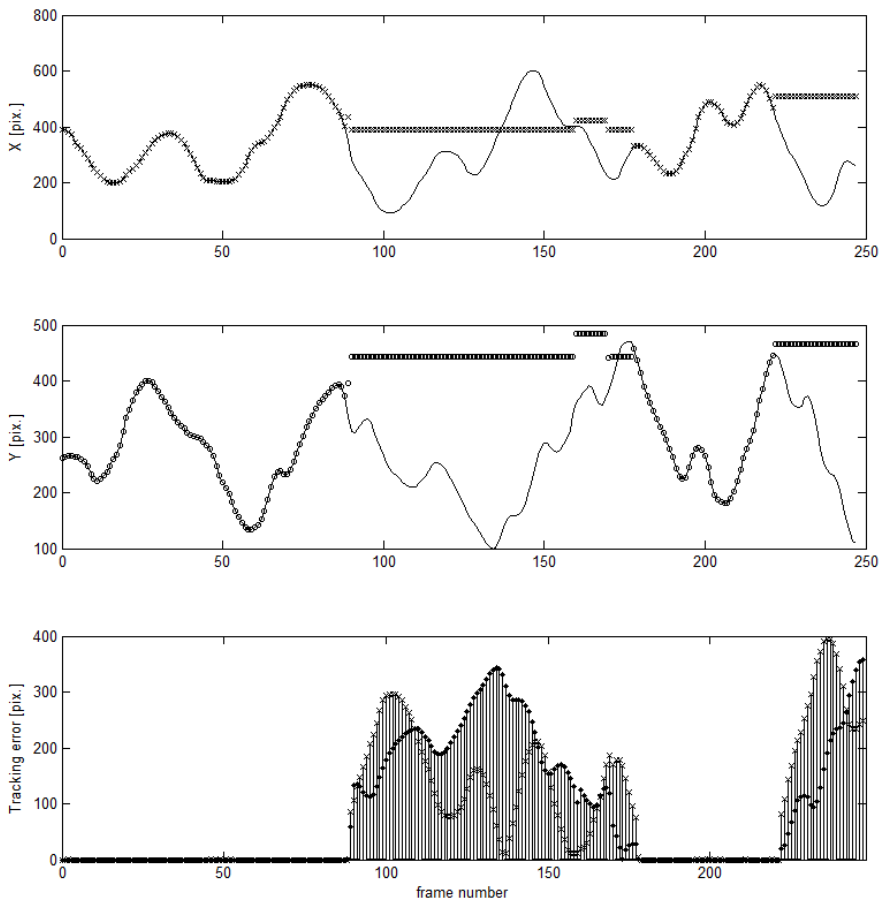

Figure A.3: 2D Tracking. No Prediction. Cluttered background. Tracking fails between frames 90 and 179, and from 223 and out.$\varepsilon_{TP} = 164.21$

Plot A.4: 2D-tracking using linear prediciton, cluttered background.

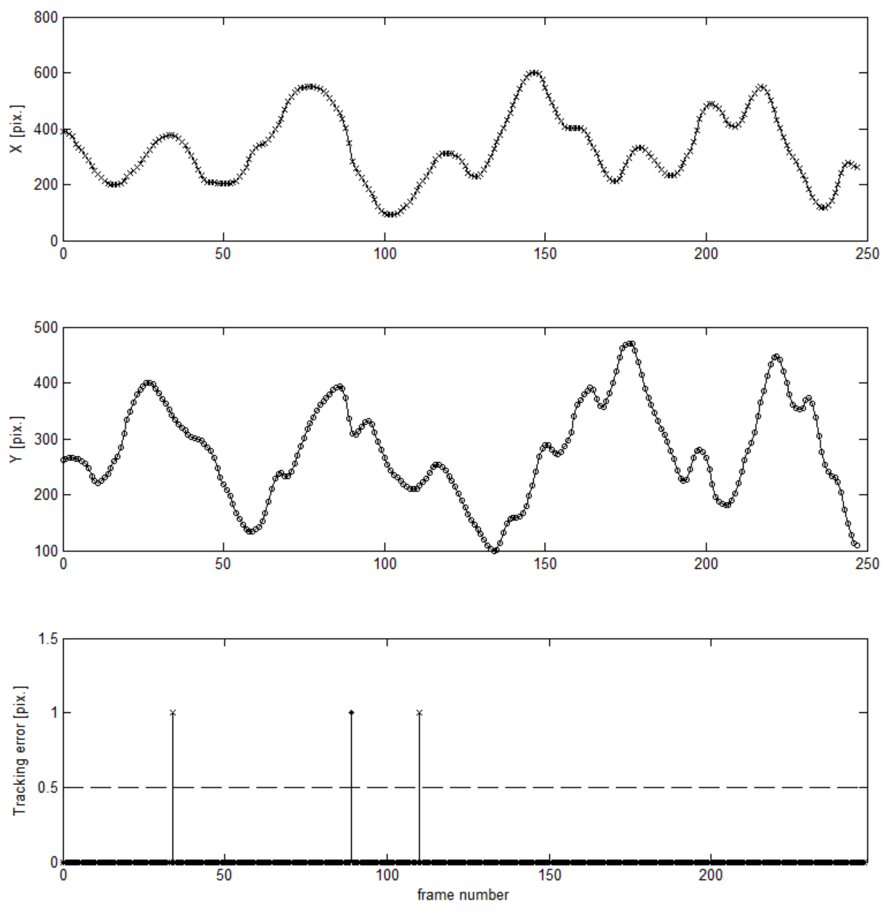

Figure A.4: 2D Tracking. Linear Prediction. Cluttered background. Tracking succedes again.$\varepsilon_{\mathrm{TP}} = 0.0120$

Plot A.5: 2D-tracking. Noise overlay 1X.

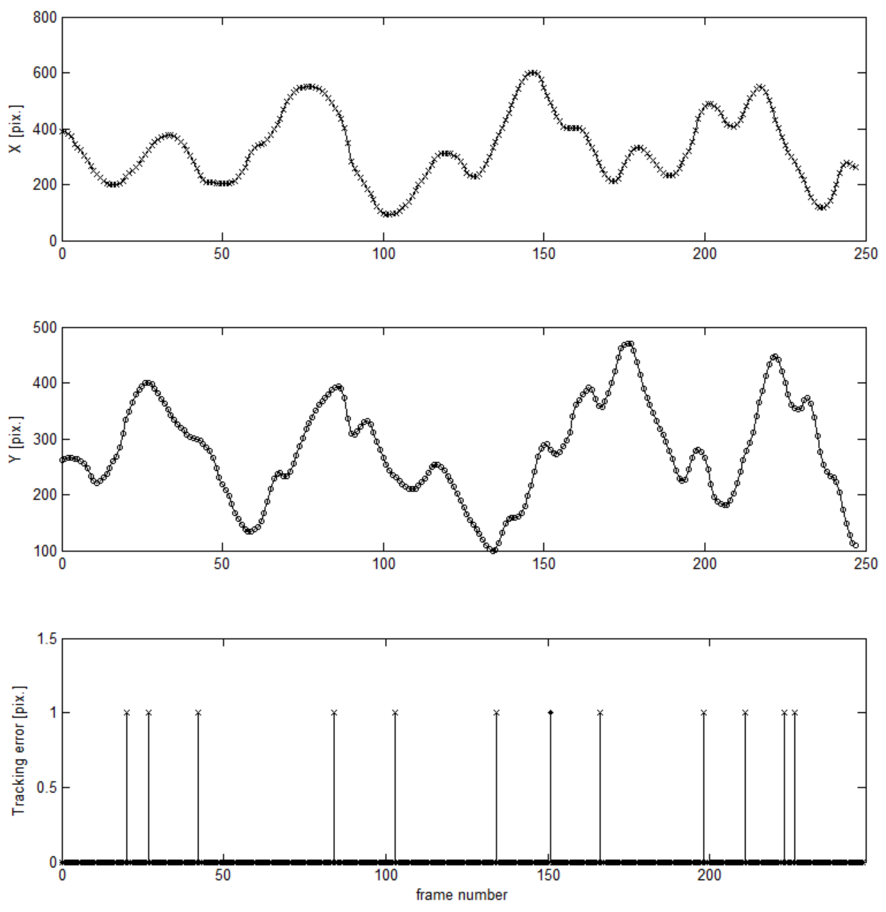

Figure A.5: 2D Tracking. Linear Prediction. Fixed background. Noise added with factor of 1X. $\varepsilon_{\mathrm{TP}} = 0.04839$

Plot A.6: 2D-tracking. Noise overlay 10X.

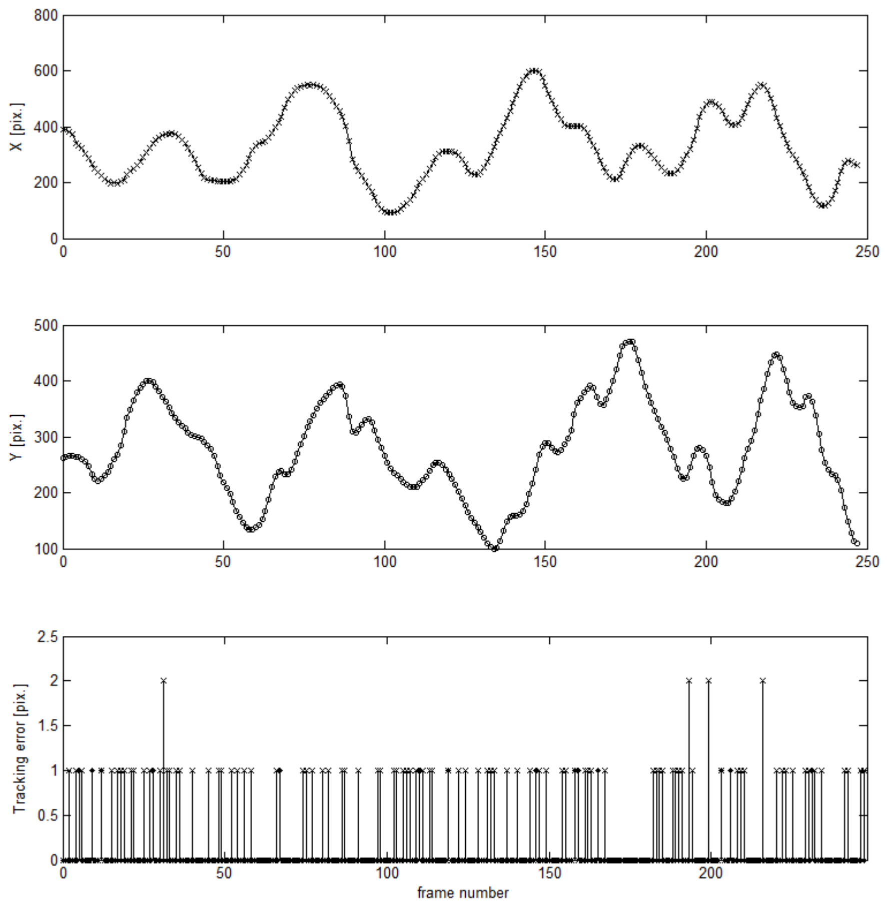

Figure A.6: 2D Tracking. Linear Prediction. Fixed background. Noise added with factor of 10X. $\varepsilon_{\mathrm{TP}} = 0.44758$

Plot A.7: 2D-tracking. Noise overlay 20X.

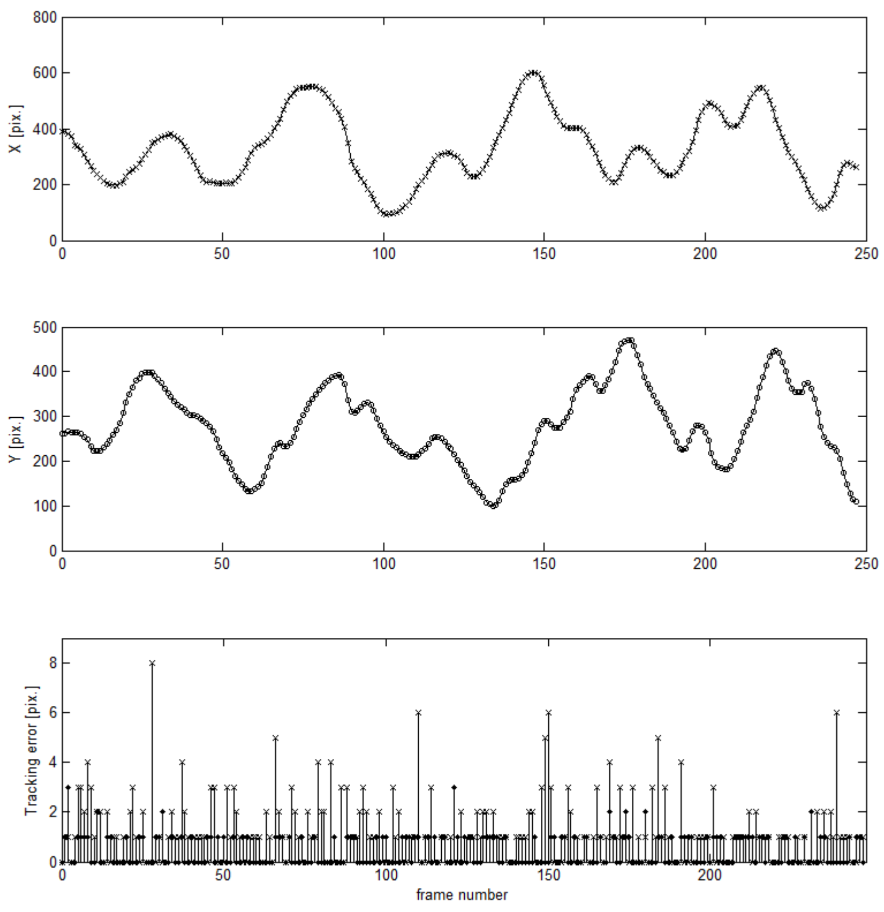

Figure A.7: 2D Tracking. Linear Prediction. Fixed background. Noise added with factor of 20X. $\varepsilon_{\mathrm{TP}} = 1.6532$

Plot A.8: 2D-tracking. Noise overlay 25X.

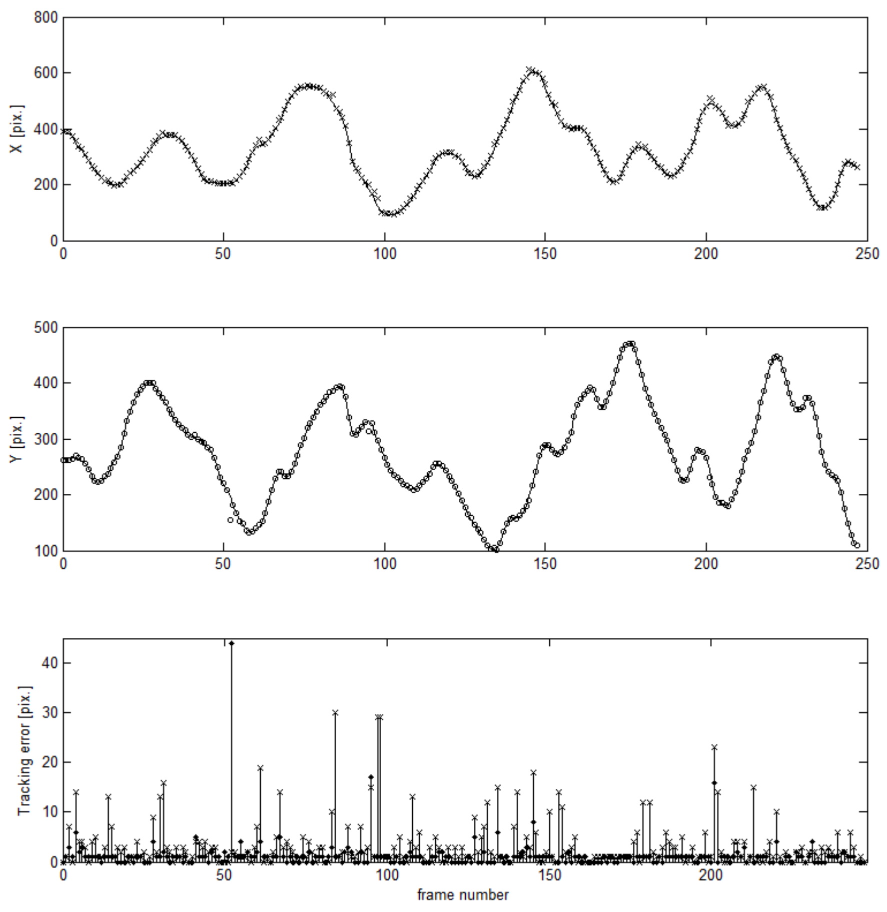

Figure A.8: 2D Tracking. Linear Prediction. Fixed background. Noise added with factor of 25X. $\varepsilon_{\mathrm{TP}} = 4.6774$

Plot A.9: 2D-tracking. Noise overlay 30X.

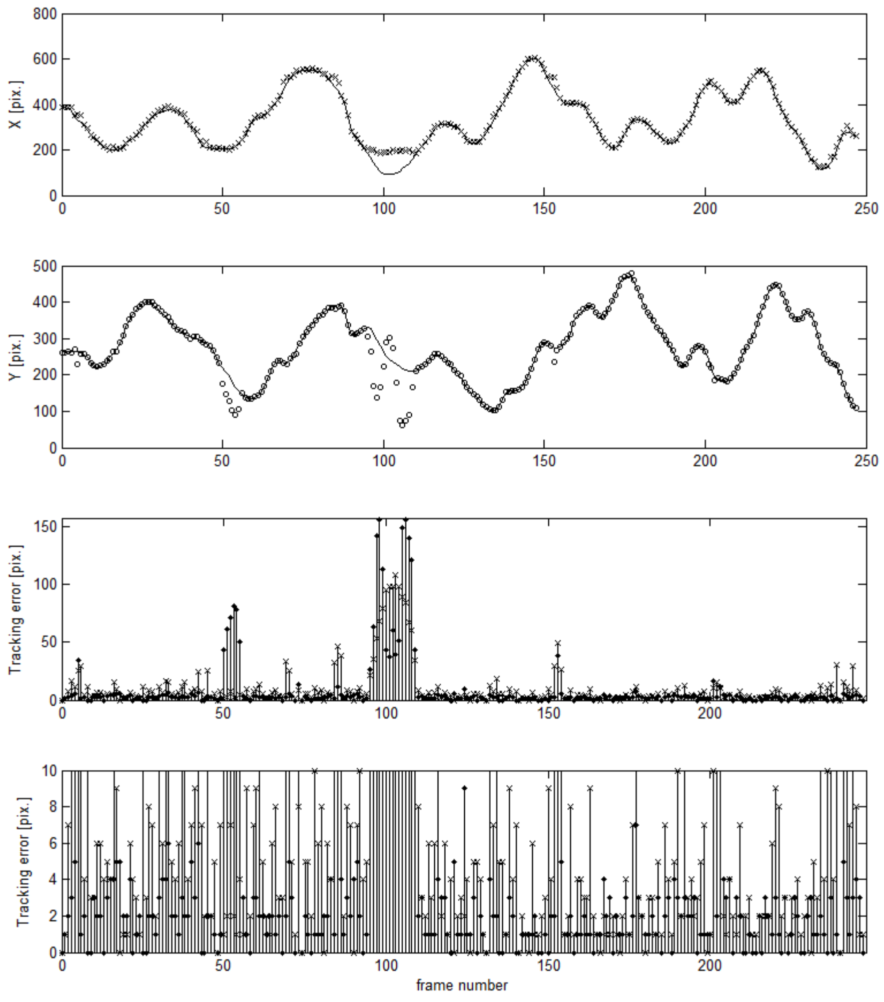

Figure A.9: 2D Tracking. Linear Prediction. Fixed background. Noise added with factor of 30X. $\varepsilon_{\mathrm{TP}} = 19.290$

Plot A.10: 2D-tracking. Noise overlay 35X.

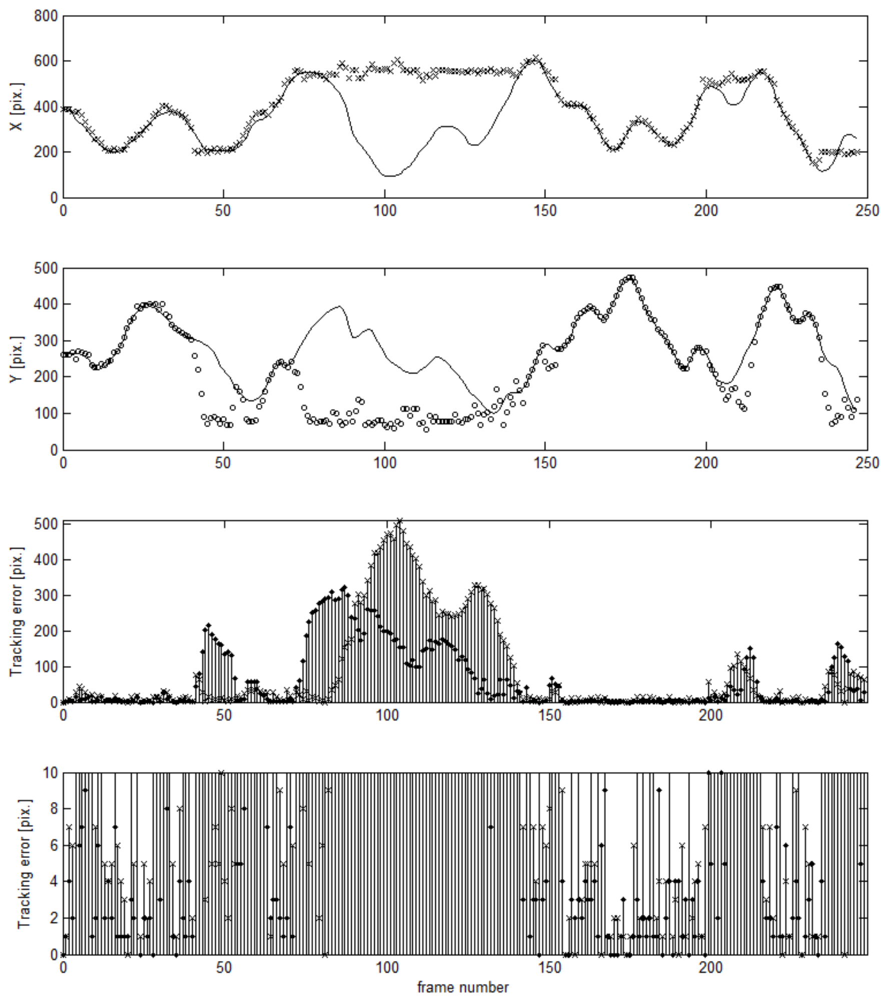

Figure A.10: 2D Tracking. Linear Prediction. Fixed background. Noise added with factor of 35X. $\varepsilon_{\mathrm{TP}} = 144.28$

Plot A.11: 2D-tracking. Partial Occlusion. 5X noise.

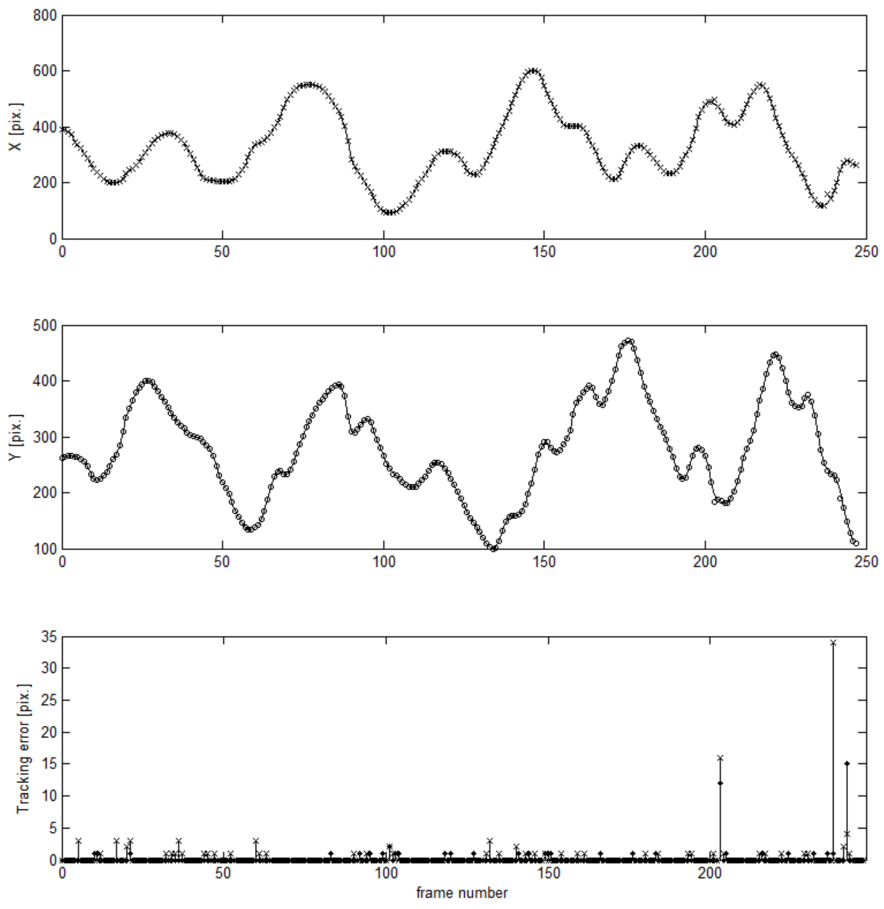

Figure A.11: 2D Tracking. 60% Linear Prediction. Fixed background. 5X Noise. High level of partial occlusion. $\varepsilon_{\mathrm{TP}} = 0.68548$

Plot A.12: 2D-tracking. Passing Occlusion. 5X noise.

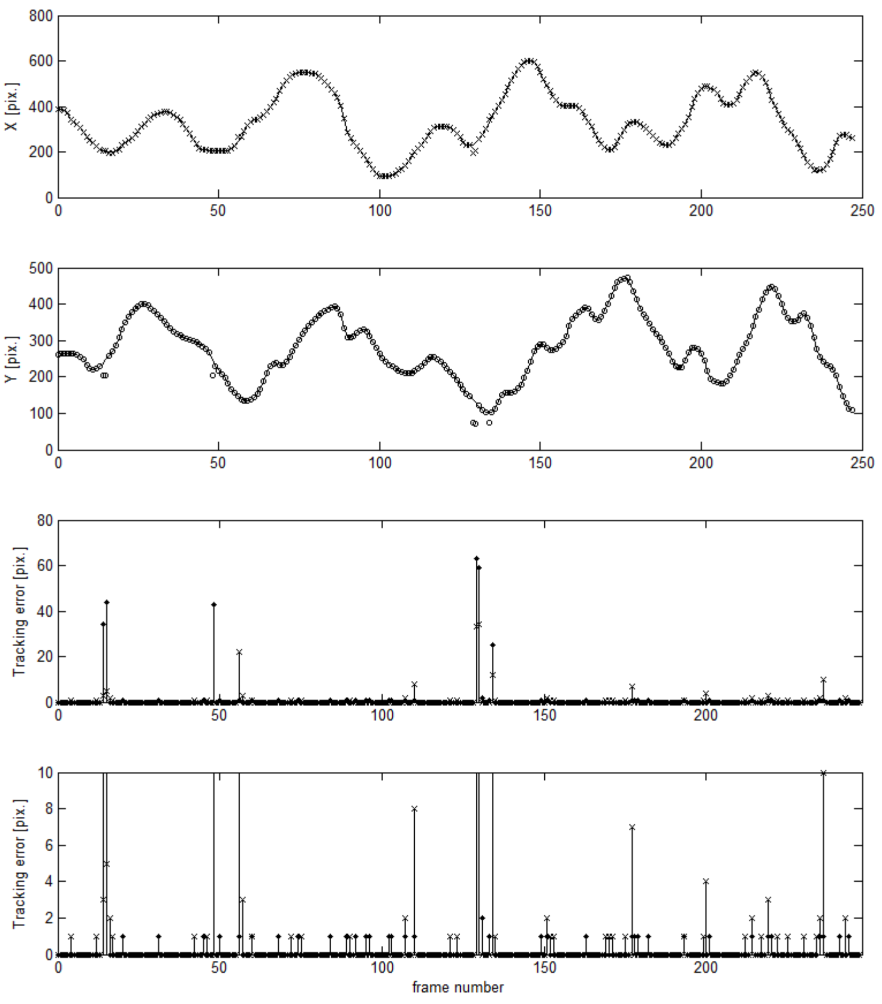

Figure A.12: 2D Tracking. 60% Linear Prediction. Fixed background. 5X Noise. High level of passing occlusion. $\varepsilon_{\mathrm{TP}} = 1.9596$. In only 19 frames was the tracking off by more than 1 pixel

Plot A.13: 2D-tracking. Complete Occlusion. 5X noise.

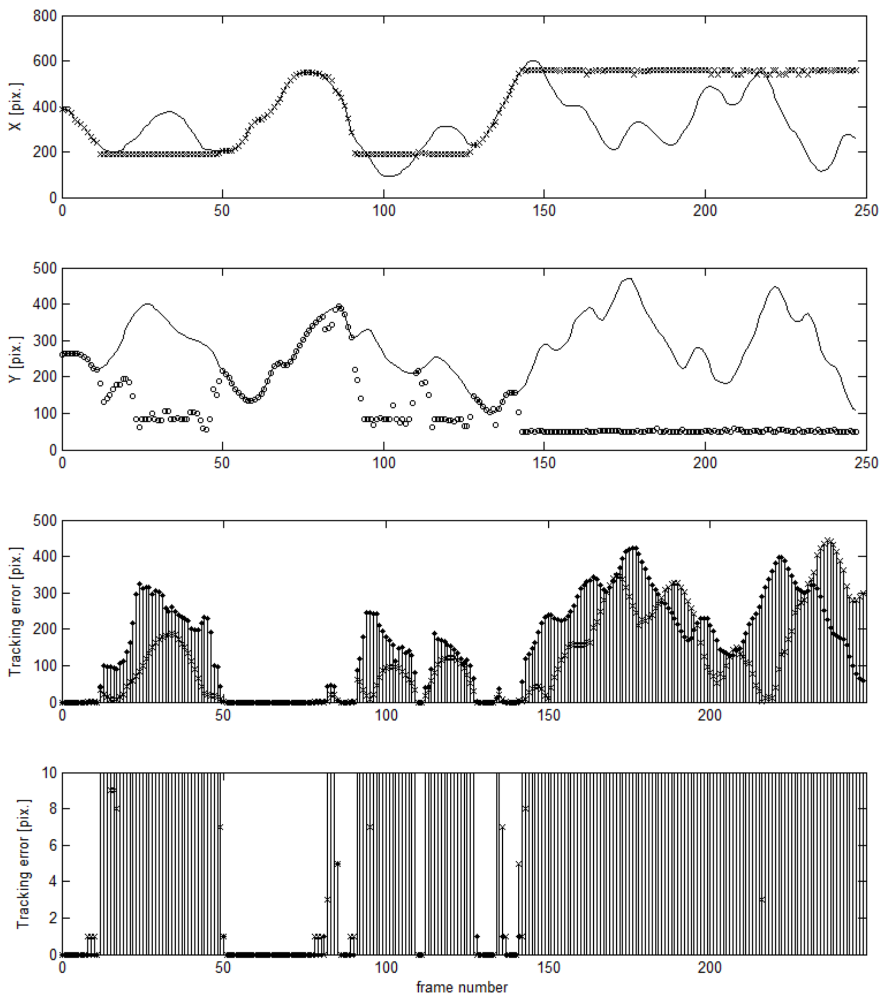

Figure A.13: 2D Tracking. 60% Linear Prediction. Fixed background. 5X Noise. Complete occlusion causes tracking to fail. It only resumes if the subject becomes close to the state space point where it lost track, and is not occluded. $\varepsilon_{\mathrm{TP}} = 271.00$.

Plot A.14: 2D+Scale tracking. 1X noise.

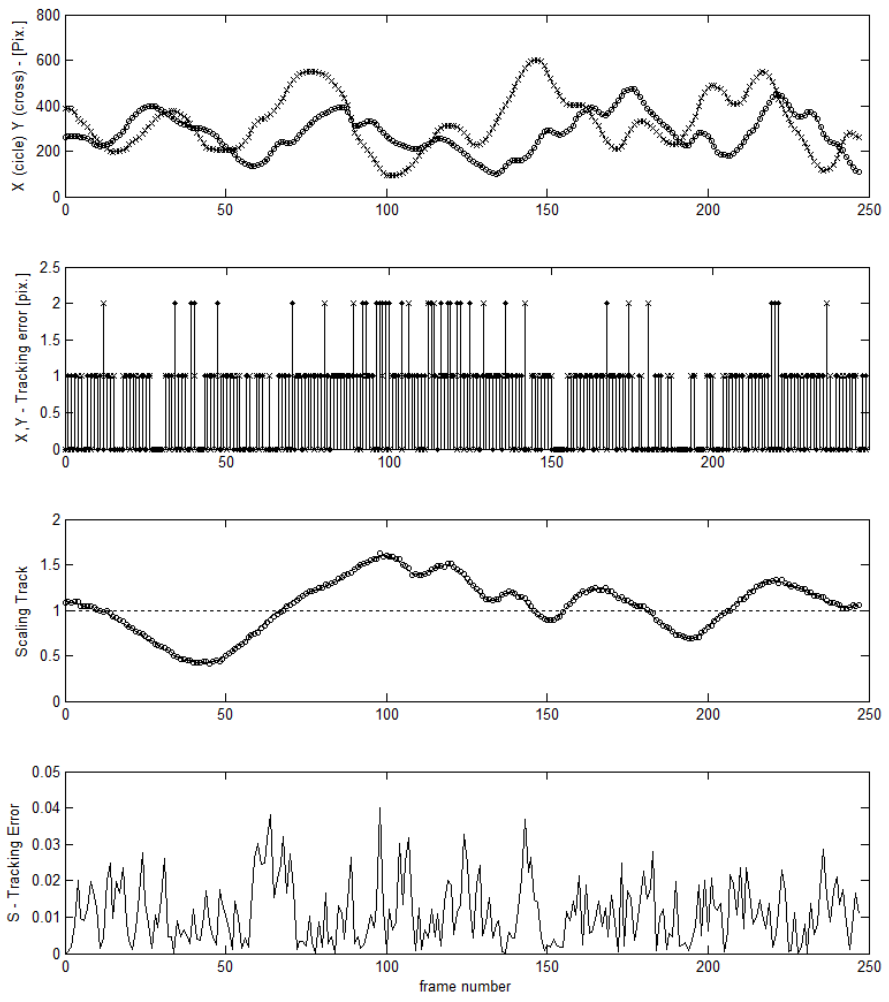

Figure A.14: 2D+Scale Tracking. Fixed background. 1X Noise. Tracking extended to the three-dimensional search space containing position and scale. Third plot from the top shows the true values (line) and the tracked values (circle). $\varepsilon_{\mathrm{TP}} = 1.3387$.

Plot A.15: 2D+Scale tracking. 5X noise. Partial Occlusion

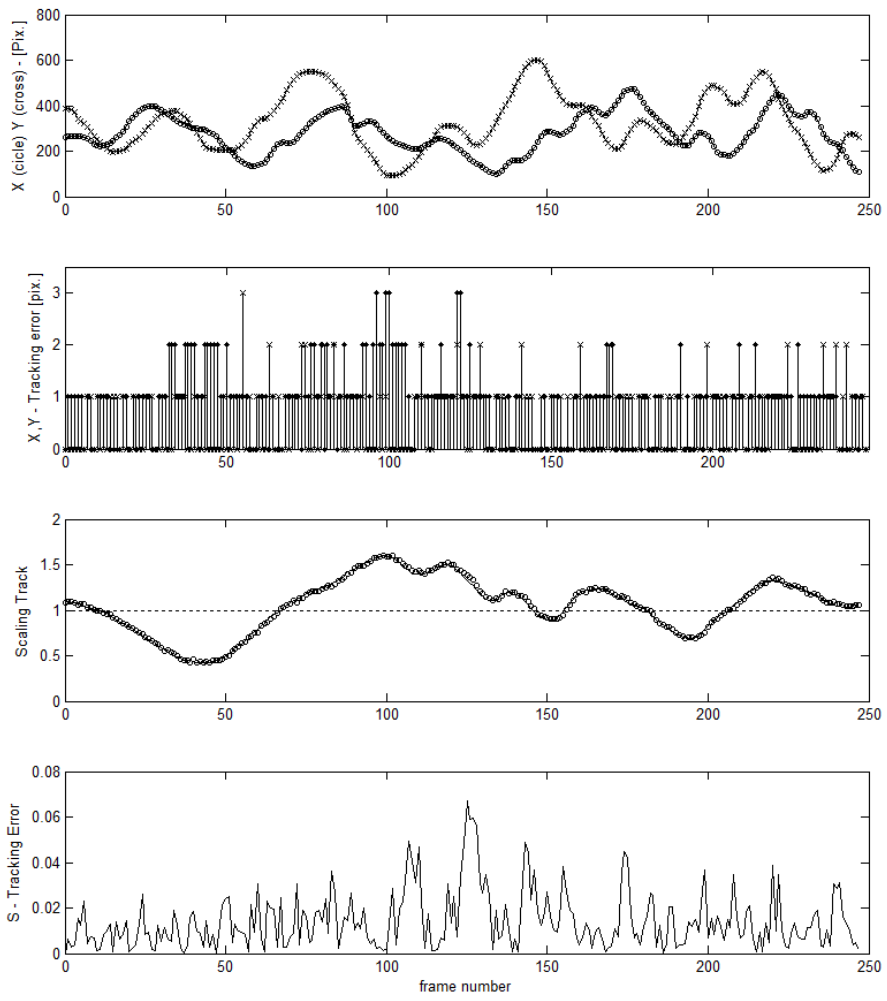

Figure A.15: 2D+Scale Tracking. Fixed background. 5X Noise. $\varepsilon_{\mathrm{TP}} = 1.4919$.

Plot A.16: 2D+R_z tracking. 1X noise.

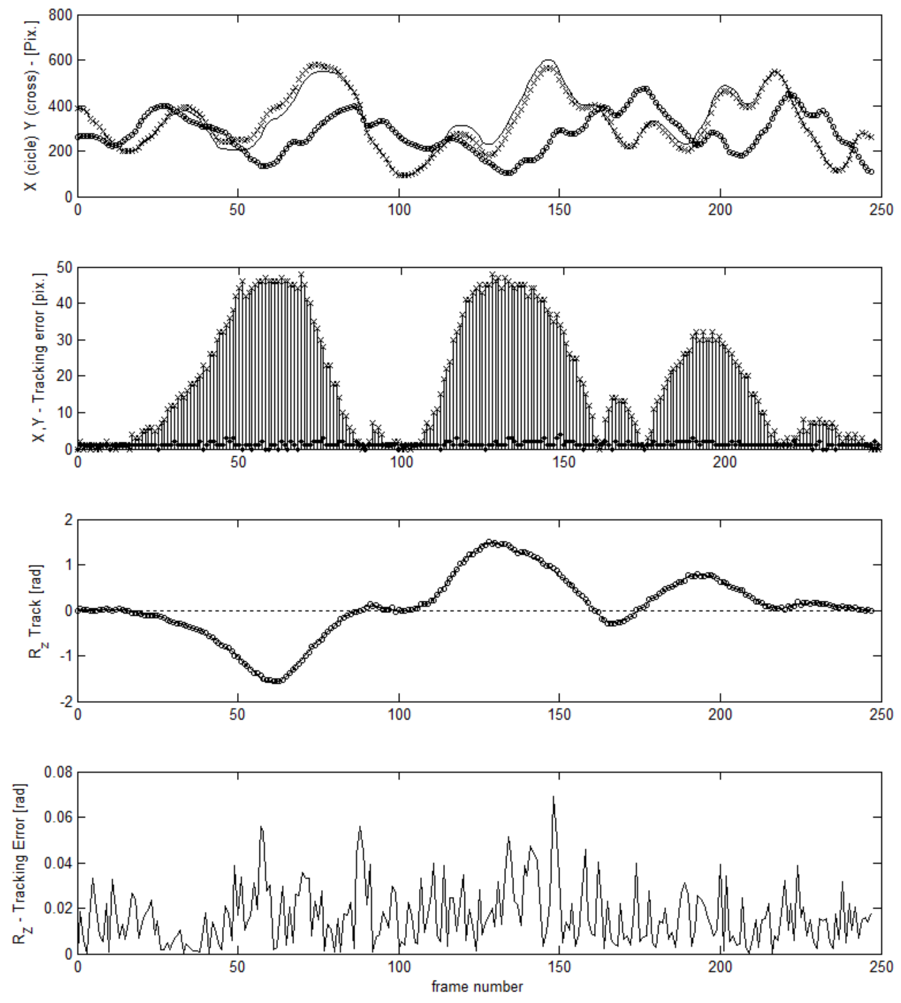

Figure A.16: 2D+R_z Tracking. Fixed background. 1X Noise. $\varepsilon_{\text{TP}} = 20.254$.

Plot A.17: 2D+R$_z$ tracking. 1X noise.

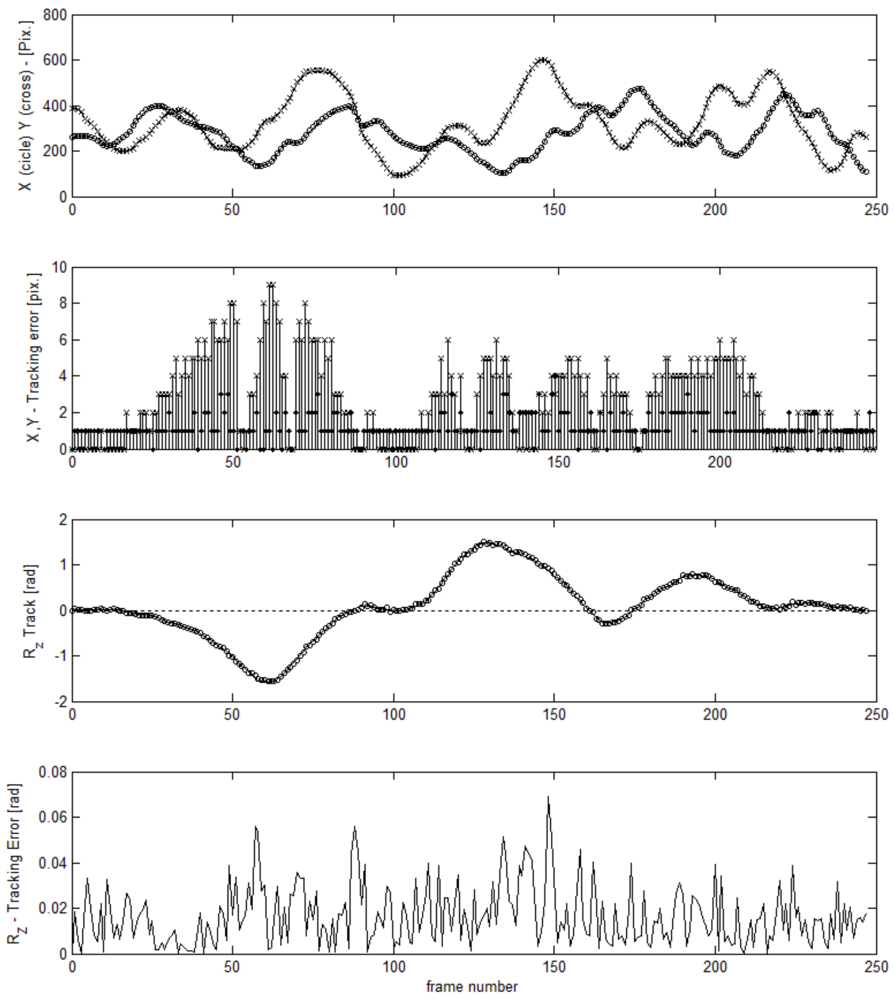

Figure A.17: 2D+R$_z$ Tracking. Fixed background. 1X Noise. The same tracking output as on the previous page, but T_x correction performed using tracked angle output. $\varepsilon_{\mathrm{TP}} = 4.0161$.

Plot A.18: 2D+Scale+R_z tracking. 1X noise.

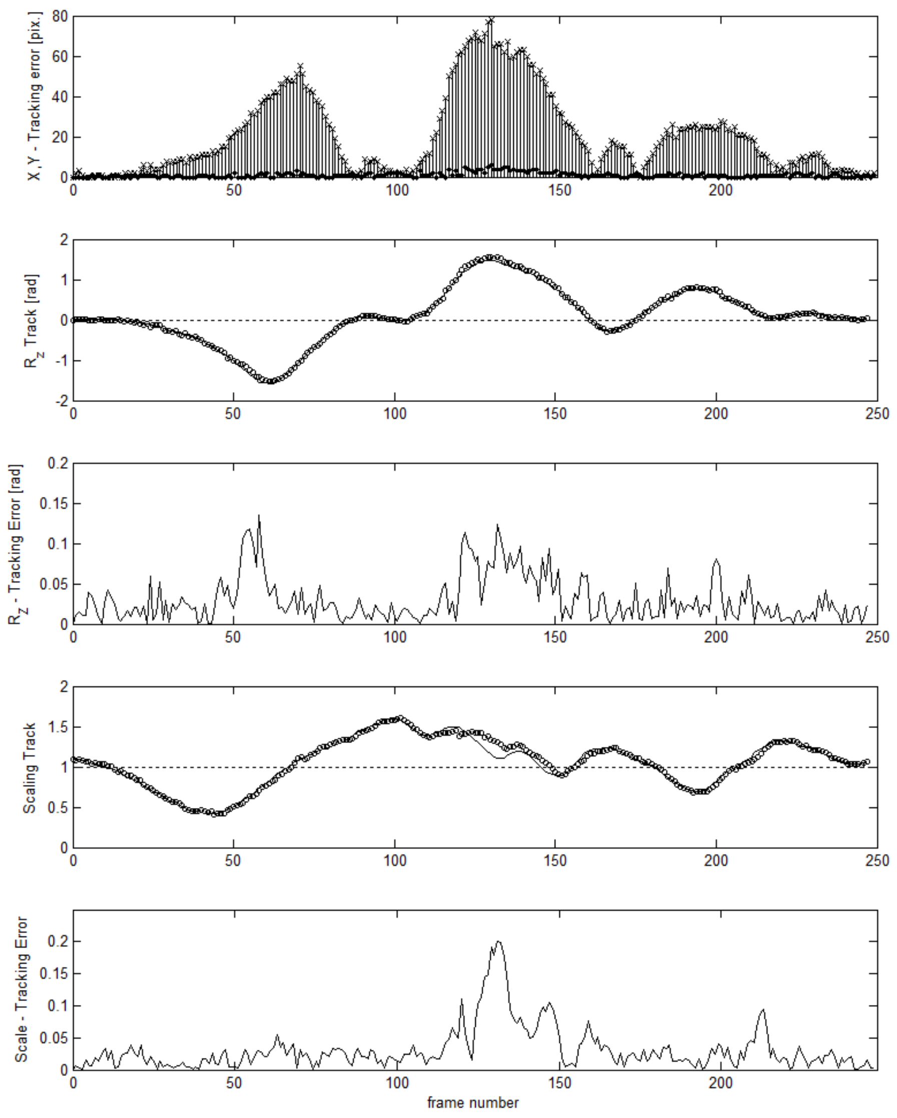

Figure A.18: 2D+Scale++R_z Tracking. Fixed background. 1X Noise. $\varepsilon_{\mathrm{TP}} = 6.7339$.

Plot A.19: 2D+Scale+R$_z$ tracking. 1X noise.

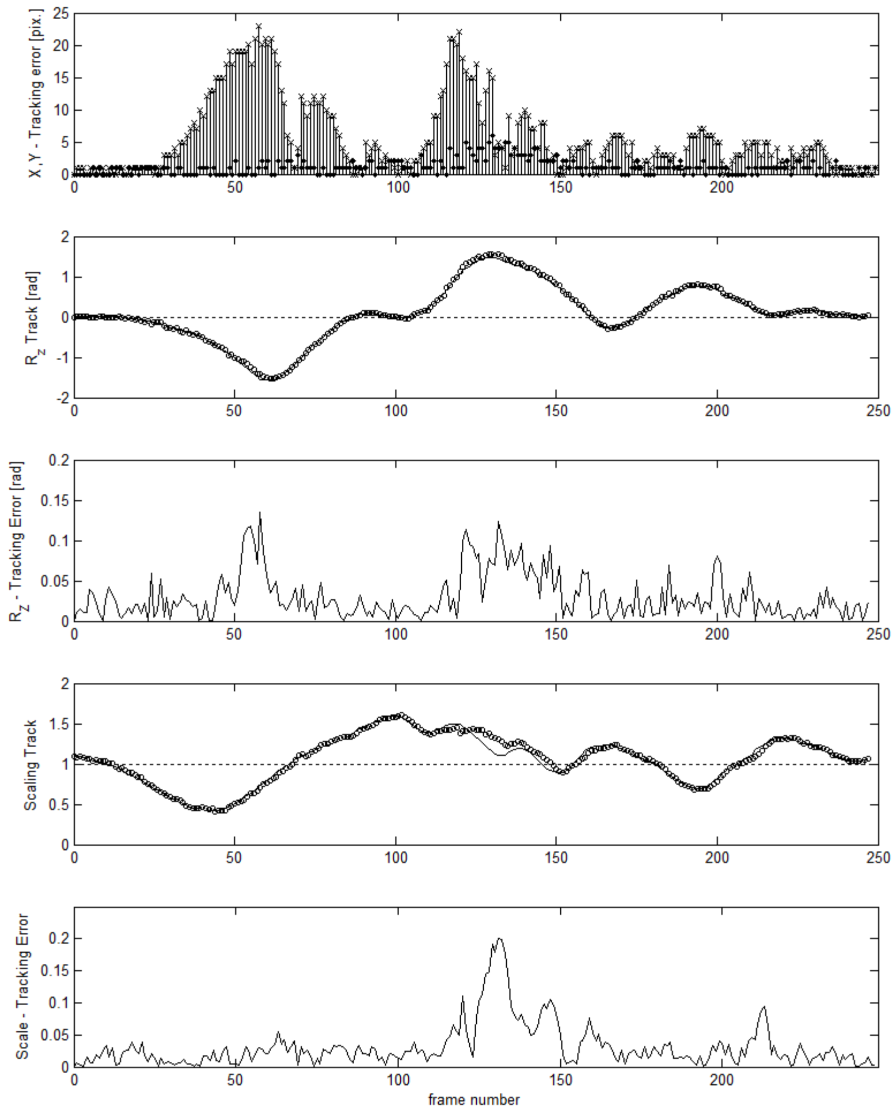

Figure A.19: 2D+Scale++R$_z$ Tracking. Fixed background. 1X Noise. $\varepsilon_{\mathrm{TP}} = 21.948$.

Plot A.20: 2D+Scale+R$_z$ tracking. 5X noise. Occlusion

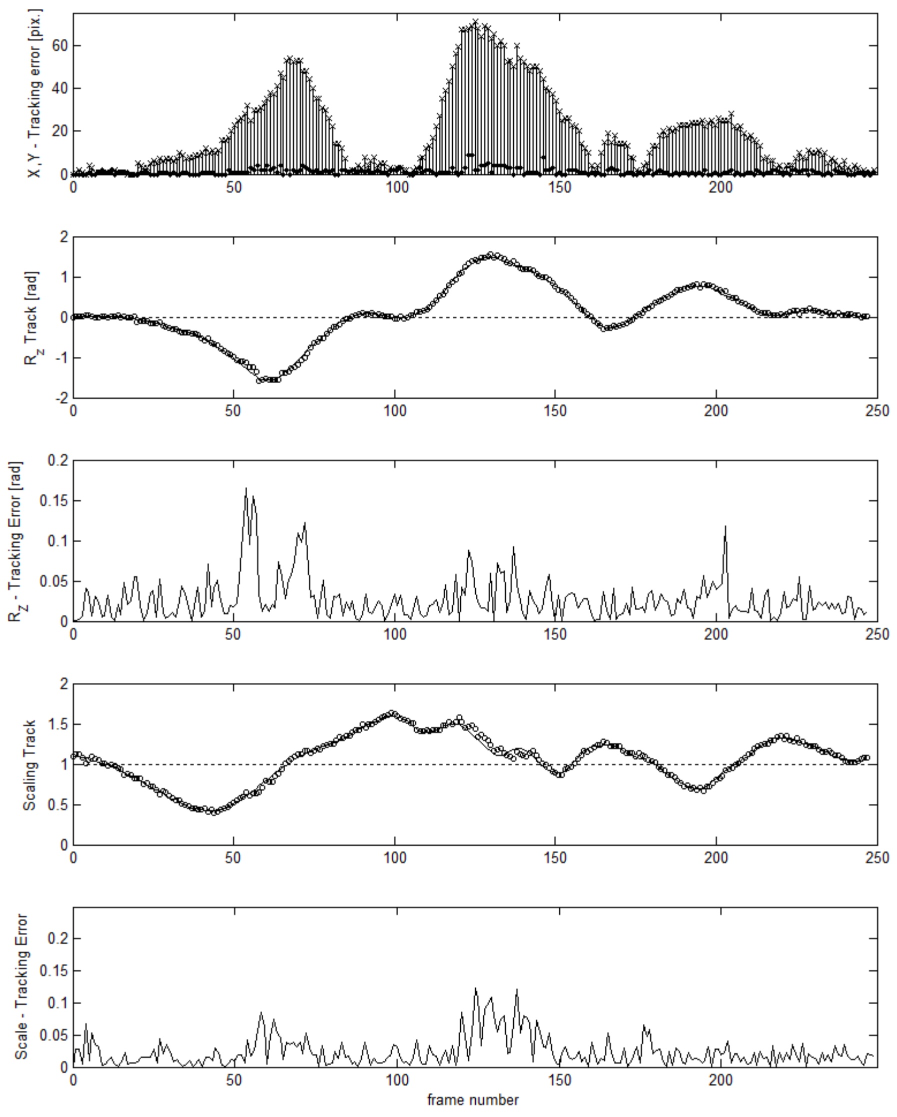

Figure A.20: 2D+Scale++R$_z$ Tracking. Fixed background. 5X Noise. $\varepsilon_{\mathrm{TP}} = 21.948$.

Plot A.21: 2D+Scale+R$_z$ tracking. 5X noise. Occlusion

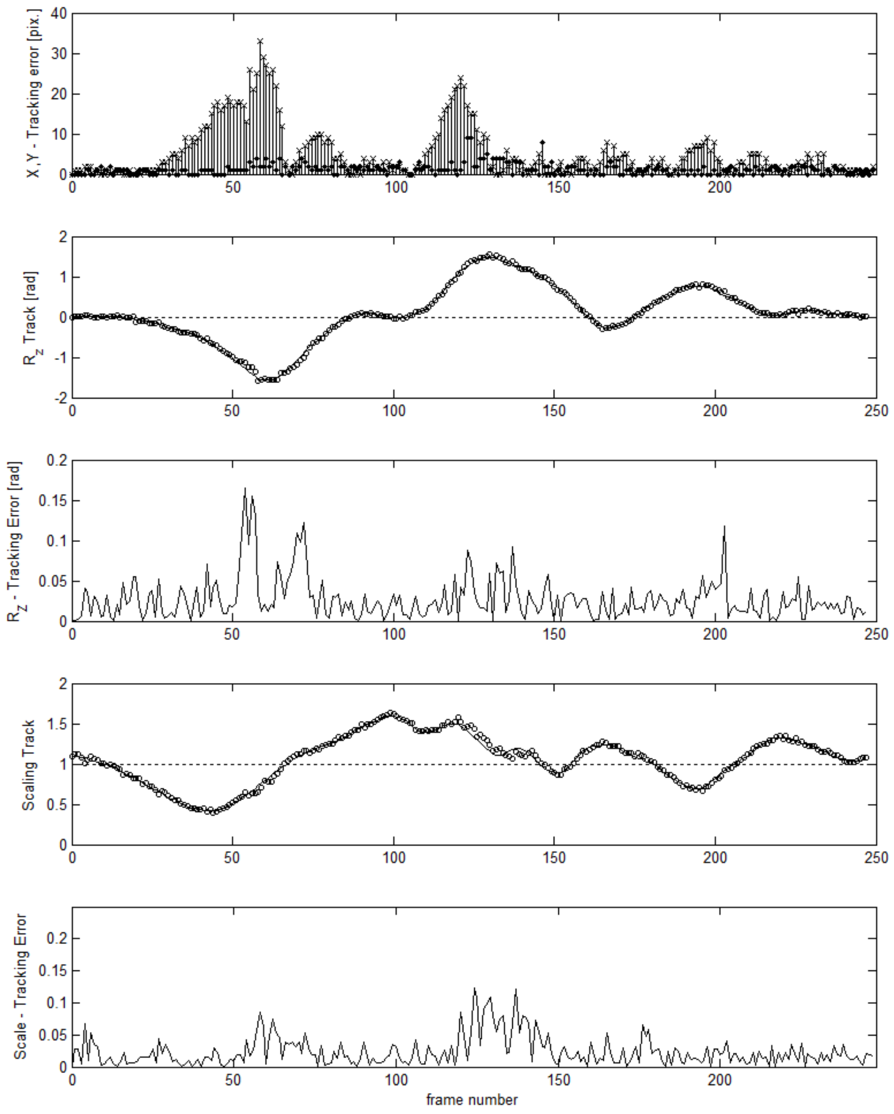

Figure A.21: 2D+Scale++R$_z$ Tracking. Fixed background. 5X Noise. $\varepsilon_{\mathrm{TP}} = 21.948$.

A.2 Content of Accompanied CD

Attached to the back of this thesis follows a CD. Documentation of particulars is provided in the CD. A rough overview of the content is given:

Software [Code and Binaries]

1. Implemented particle filtering method.
2. Benchmark-video creation tools.

Demonstration [Video]

1. Overall functionality of method
2. Dynamic Template
3. Particle cloud visualization

Test Cases [Video]

1. The test cases discussed in the thesis
2. Corresponding 3D model overlaid (covering head)
3. Tracking overlaid