GPU-ACCELERATED PARTICLE FILTERING FOR MODEL-BASED TRACKING

GPU-ACCELERATED PARTICLE FILTERING FOR 3D MODEL-BASED VISUAL TRACKING

BY

J. ANTHONY BROWN, B.SC.ENG.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2010) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	GPU-Accelerated Particle Filtering for 3D Model-Based Visual Tracking
AUTHOR:	J. Anthony Brown B.Sc.Eng. (Computer Engineering) University of New Brunswick, Fredericton, Canada
SUPERVISOR:	Dr. David Capson
NUMBER OF PAGES:	xxi, 125

For my family

Abstract

Model-based approaches to 3D object tracking and pose estimation that employ a particle filter are effective and robust, but computational complexity limits their efficacy in real-time scenarios. This thesis describes a novel framework for acceleration of particle filtering approaches to 3D model-based, markerless visual tracking in monocular video using a graphics processing unit (GPU). Specifically, NVIDIA compute unified device architecture (CUDA) and Direct3D are employed to harness the single-instruction multiple-thread (SIMT) programming model used by the GPU's massively parallel streaming multiprocessors (SMs) for simulation (3D model rendering) and evaluation (segmentation, feature extraction, and weight calculation) of hundreds of particles at high speeds. The proposed framework addresses the computational intensity that is intrinsic to all particle filter approaches, including those with modifications and extensions that strive to reduce the number of required particles while maintaining tracking quality.

The sampling importance resampling (SIR) particle filter and its utility in 3D model-based tracking is reviewed and a detailed overview of relevant GPU-programming concepts is presented. The proposed framework is formulated as a series of interconnected steps and the functionality and implementation of each is described in detail. Rigid and articulated tracking examples are presented in the context of human-computer interaction (HCI) and augmented reality (AR) applications, with a focus on bare hand tracking. Performance and tracking quality results demonstrate markerless, model-based visual tracking on consumer-grade hardware with pixel-level accuracy up to 95 percent at 30+ frames per second. The framework accelerates particle evaluation up to 25 times over a comparable CPU-only implementation, providing an increased particle count while maintaining real-time frame rates.

Acknowledgments

I would like to express my sincere gratitude to everyone who has supported me over the past two years. First and foremost, to Dr. David Capson, who has not only offered support and encouragement throughout my tenure at McMaster, but is largely responsible for bringing me to the school in the first place. Somehow, even though he was constantly juggling countless demands on his time, Dr. Capson was always available for a meeting or demo and never failed to insight motivation and pride in my work. He has been a true inspiration and a constant reminder of what an engineer from New Brunswick can go on to achieve.

I would also like to thank Mike Kinsner and Peter Kuchnio for welcoming me to the Computer Vision lab and introducing me to CUDA programming, which has been a major focus of my research. Thanks goes out to the other members of the lab as well, for always making me feel welcome even when I was only coming in once or twice a month and for coming with me to get my "fancy coffee," even though it was a little farther away. Thanks also to the faculty and staff of the ECE department for your help and guidance, particularly Dr. Kiruba, Dr. Doyle, Cheryl Gies, Alexa Huang, Terry Greenlay, and Cosmin Coroiu.

I would not be where I am today without the constant support of my family and friends. Thanks especially to my grandmother, Dena Brown, for her love, encouragement, and lemon squares, and to my grandfather, Jim Brown, who taught me everything I know, including the importance of bringing your own pencil to a test. Thanks also to Amanda Dyer, for so much support when I needed it the most and for somehow tolerating my completely unnatural sleeping pattern. Last, but most definitely not least, thanks to my incredible mother, Renee Brown, whose love and pride has driven me to succeed more than anything else. Her phone calls and visits over the past two years meant more than she can possibly know.

I would also like to thank NSERC Canada and McMaster University for financially supporting this research.

Notation and Acronyms

General Notation	
x	Scalar
$\mathbf{x}$	Vector
$\mathbf{X}$	Matrix
$\mathbf{I}^n$	Identity Matrix of dimension $n \times n$
$\|x\|$	Absolute value of x
$y \propto x$	y is proportional to x
$y \approx x$	y is approximately equal to x
$y \sim f(x)$	y distributed according to f(x)
$Pr(y = x)$	Probability y is equal to x
$\mathcal{N}(\mu, \Sigma)$	Normal distribution with mean μ and covariance Σ
$\mathcal{U}[a, b]$	Uniform distribution between a and b
$\delta(\cdot)$	Dirac delta function
$\mathbb{N}$	Set of natural numbers $\{0, 1, 2, ...\}$
$A \cup B$	Union of A and B
$A \cap B$	Intersection of A and B

Bayesian Estimation Notation	
$p(x\|z)$	Probability of x given z
$p(\mathbf{x}_t\|\mathbf{x}_{t-1})$	Prior PDF
$p(\mathbf{z}_t\|\mathbf{x}_t)$	Likelihood PDF
$p(\mathbf{x}_t\|\mathbf{z}_{1:t})$	Posterior PDF

Particle Filter Notation	
$\mathbf{x}_t$	System state at time t
$\mathbf{z}_t$	Observation at time t
$\hat{\mathbf{x}}_t$	Estimate of $\mathbf{x}$ at time t
N	Number of particles
$\{\mathbf{x}_t^i\}_{i=1}^N$	Set of N particles at time t
$\{w_t^i\}_{i=1}^N$	Set of N particle weights at time t
$q(\mathbf{x}_t\|\mathbf{x}_{t-1}^i, \mathbf{z}_t)$	Proposal/Importance PDF

Acronyms	
API	Application programming interface
AR	Augmented reality
CDF	Cumulative density function
CPU	Central processing unit
CUDA	Compute unified device architecture
DIP	Distal interphalangeal
DOF	Degree of freedom
FLOPS	Floating point operations per second
FN	False negative
FP	False positive
fps	Frames per second
GPGPU	General purpose computing on graphics processing units
GPU	Graphics processing unit
HCI	Human-computer interaction
HLSL	High-level shading language
HPC	High-performance computing
i.i.d.	Independent and identically distributed
IP	Interphalangeal
MAE	Mean absolute error
MP	Metacarpophalangeal
NVCC	NVIDIA C compiler
PCI-E	Peripheral component interconnect express
PDF	Probability density function
PIP	Proximal interphalangeal
SDK	Software developer's kit
SIMD	Single-instruction multiple-data
SIMT	Single-instruction multiple-thread
SIR	Sampling importance resampling
SIS	Sequential importance sampling
SM	Streaming multiprocessor
SMC	Sequential Monte Carlo
SP	Streaming processor
TM	Trapeziometacarpal
TN	True negative
TP	True positive
TPC	Texture processing cluster

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Estimating the position and orientation of a rigid or articulated 3D object as it moves through a video sequence is a central problem in computer vision, with applications including visual servoing, surveillance, human-computer interaction (HCI), teleconferencing, performance-driven animation, medical imaging, and augmented reality (AR). For example, visual servoing involves estimation of the position and configuration of a device, such as a robotic arm, for input to the control system governing its movement. AR involves overlaying a real scene with computer-generated images, often "attaching" a virtual object to a real object that is moving through the scene. Vision-based HCI aims to provide a user with a more natural or intuitive platform for interacting with a computer, without the use of a mouse, keyboard, joystick, or other traditional input device.

Approaches to these applications are often simplified by relying on range sensors (e.g., ultrasonic, infrared, magnetic), point or planar fiducial markers (e.g., LEDs, reflectors), position encoders, (e.g., rotary encoders, optical encoders, data gloves in hand tracking applications), or multiple-camera rigs. However, in many applications, these engineered solutions prove intrusive, unreliable, expensive, cumbersome, or impractical and a purely vision-based method is preferred.

While practical, visual approaches to 3D object tracking are challenging (Lepetit and Fua, 2005). Six degrees of freedom (DOFs) are needed to describe an object's location with respect to a camera and each internal joint in the object can add one to three more. These DOFs must be continuously estimated based solely on a video sequence's pixel data. Because the pixel data represents a single camera's 2D projection of a 3D object, there can be a great deal of ambiguity. Other challenges include motion blur from fast movements, variations in lighting, shading, shadows and texture, and total or partial occlusion of the object.

1.1 3D Model-Based Tracking

Visual tracking techniques can be coarsely divided into *appearance-based* and *model-based* categories. Appearance-based methods, also known as view-based or single-frame approaches, use cues from the video sequence (colour, contours, etc.) whereas model-based methods project a 3D model of a *tracking target* with known geometry onto a 2D surface (plane) for comparison with the video sequence. In either case, the goal is to estimate the set of parameters that best describes the object's pose. Model-based approaches, sometimes referred to as *analysis-by-synthesis* or *generative methods*, often generate superior results and are algorithmically simpler because the parameters being estimated are the same DOFs being manipulated to control the model. Unfortunately, such approaches require many 3D renderings of the object to be produced and compared to each video frame, making them computationally intensive and consequentially less popular than appearance-based techniques.

A brute force approach to model-based tracking would involve configuring a 3D model in every possible pose, projecting each configuration onto a plane, and comparing each plane to the current video frame to identify the configuration that best represents the true pose of the tracking target. This would involve a massive search space that grows exponentially larger with each additional DOF that needs to be estimated. To limit the search space, and consequently the computational intensity, *statistical estimation* is often employed to ensure that only model configurations with a high likelihood of accurately representing the true pose of the tracking target are considered.

1.2 The Particle Filter

Model-based tracking can be framed as a *Bayesian state estimation* problem, facilitating the application of Bayesian filters, such as the Kalman filter, grid-based methods, and the particle filter (Candy, 2009) to identify statistically likely model configurations. The *particle filter* is a particularly attractive option, as it is a simulation-based *sequential Monte Carlo (SMC)* technique capable of estimating the evolution of non-linear, non-Gaussian stochastic processes (Ristic *et al.*, 2004), such as the unconstrained motion of a tracking target. Where most state estimation techniques attempt to find an ideal solution with an approximate model, the particle filter aims for an approximate solution using an ideal model.

Particle filters have been applied to a wide variety of applications, including finance theory, audio recognition, robot localization, and numerous tracking applications. Also known as the bootstrapping filter (Gordon *et al.*, 1993), survival-of-the-fittest (Kanazawa *et al.*, 1995), and the CONDENSATION algorithm (Isard and Blake, 1998), the particle filter has proven particularly applicable to 3D model-based

tracking applications, such as face, hand, and vehicle tracking (Zhou *et al.*, 2004). Unfortunately, even with the reduced search space provided by the particle filter, hundreds or thousands of model configurations are still required for robust and accurate tracking. Since each configuration must be rendered, projected and evaluated against each video frame, usually at 30 frames per second (fps) or faster, real-time performance is rarely achieved, particularly when tracking articulated objects.

1.3 GPU Computing and NVIDIA's CUDA

In recent years, the *graphics processing unit (GPU)* has emerged as a powerful, affordable, and adaptable parallel computing platform suitable for many high-performance computing (HPC) tasks. Programming models, such as ATI Stream, Brook+, and NVIDIA *compute unified device architecture (CUDA)*, allow programmers to exploit the GPU's massively parallel *streaming multiprocessors (SMs)* for general purpose computing instead of the 3D graphics processing for which they were originally designed. Serving as a coprocessor to the central processing unit (CPU), the GPU has been reported to accelerate suitable applications (Hwu *et al.*, 2009; Coutinho *et al.*, 2009) 10 to 1000 times over CPU-only implementations (Tan *et al.*, 2009; Pock *et al.*, 2008).

Introduced in 2006 to provide GPU computing on modern NVIDIA GeForce, Quadro, and Tesla products, millions of which are already deployed in PCs and workstations around the world (Halfhill, 2008), CUDA represents a large, collaborative online community, an ever-expanding number of university courses and textbooks (Kirk and mei W. Hwu, 2010), and a massive user-base. With a focus on scalability, a properly-written CUDA application will maximally utilize all available SMs as efficiently as possible, with minimal effort from the programmer. Because modern GPUs offer significantly more floating point operations per second (FLOPS) and a much higher memory bandwidth than similarly-priced CPUs, they are an ideal solution to arithmetically intense applications, such as image processing, computer vision, and video encoding. Computer vision, which involves the analysis of images, can be considered the inverse of computer graphics (Fung and Mann, 2008), which involves the synthesis of images, making GPU hardware particularly amenable to 3D model-based tracking and similar tasks that involve analysis of pixel data.

1.4 Problem Statement

This thesis demonstrates how NVIDIA's CUDA was used in conjunction with Microsoft Direct3D, an application programming interface (API) for optimized 3D graphics rendering on the GPU, to significantly accelerate particle filtering approaches

to 3D model-based visual tracking of rigid and articulated objects. Specifically, a framework for partitioning and mapping a particle filter's computationally intensive weight-update stage to a GPU is presented. This parallelizes the task of rendering and evaluating model configurations, facilitating significantly higher frame rates. The framework can be adapted to support whichever variation of the particle filter and evaluation methodology are best-suited to a particular application. Using rigid object tracking (6-DOF) and articulated bare hand tracking (10-DOF) examples in AR and HCI applications, robust and accurate, markerless, visual tracking with pixel-level likelihood at 30+ fps on consumer-grade hardware is achieved through GPU-acceleration of traditional particle filter approaches.

1.5 Thesis Organization

This thesis is based on research conducted between September 2008 and July 2010 at the Department of Electrical and Computer Engineering, Computer Vision Laboratory at McMaster University. Significant portions of the research have been submitted for publication in the IEEE Transactions on Visualization and Computer Graphics in July, 2010 under the title *A framework for 3D model-based visual tracking using a GPU-accelerated particle filter*. Additionally, early results were presented at the High Performance Computing Symposium (HPCS2010) in Toronto, Ontario, and will appear in the Journal of Physics: Conference Series, published by IOP Publishing, under the title *GPU-accelerated 3-D model-based tracking*.

The remainder of this thesis is organized as follows. Chapter 2 contains a description of Bayesian state estimation, the details of the sequential importance sampling (SIS) and sampling importance resampling (SIR) particle filters (supplemented by a derivation of the SIS filter in Appendix A), an overview of alternatives and extensions to the particle filter, and a description of the particle filter's utility in 3D model-based tracking. Chapter 3 describes the NVIDIA GT200 GPU architecture, with a focus on memory and threading models, and how it can be exploited with CUDA and Direct3D for 3D model rendering and evaluation. This section also provides a literature review of relevant GPU-computing research. The functionality and implementation of the proposed GPU-accelerated 3D model based tracking framework is detailed in Chapter 4. Section 5 describes how the framework was tested and presents results for five separate tracking experiments, with supplemental results included in Appendix B and Appendix C. Finally, Chapter 6 concludes the thesis with a discussion and summary of future work.

Chapter 2

Particle Filters and 3D Model-Based Tracking

3D model-based tracking can be framed as a Bayesian state estimation problem to facilitate the application of Bayesian filters. This chapter begins by formalizing the tracking problem in Section 2.1, then describes how the problem can be solved from a Bayesian state estimation perspective in Section 2.2. The SIR particle filter is described in Section 2.3 and the variations and alternatives to the particle filter are presented in Section 2.4. Finally, Section 2.5 describes how the particle filter is used in 3D model-based tracking applications, with a focus on articulated hand tracking. Note that this chapter is based on material presented in (Brown and Capson, 2010a).

2.1 Formulation of the Tracking Problem

The pose of a non-rigid object as it moves through a video sequence can be described by the evolution of a state sequence $\{\mathbf{x}_t, t \in \mathbb{N}\}$ comprised of six global parameters and m local (i.e., joint) parameters

$$\mathbf{x}_t = (t_x, t_y, t_z, r_x, r_y, r_z, j_0, \ldots, j_{m-1}), \tag{2.1}$$

where (t_x, t_y, t_z) are the coordinates of the object's origin (global translation), (r_x, r_y, r_z) are the global rotations around the x-, y-, and z-axis, respectively, and $(j_0, \ldots, j_{m-1})$ are the angles of m joints relative to the global object pose, configured as one or more kinematic chains. The object can be tracked by estimating the parameters of $\mathbf{x}_t$ at discrete time points (e.g., once per frame in a video sequence) as the sequence evolves according to a system dynamics model

$$\mathbf{x}_t = f_t(\mathbf{x}_{t-1}, \mathbf{v}_{t-1}), \tag{2.2}$$

where f_t is a (possibly nonlinear) function of the previous state that varies with time and describes the evolution of the state sequence, and $\mathbf{v}_{t-1}$ is an i.i.d. noise sequence. It is not generally possibly to observe $\mathbf{x}_t$ directly; therefore, measurements are taken from a noisy observation model

$$\mathbf{z}_t = h_t(\mathbf{x}_t, \mathbf{n}_t), \tag{2.3}$$

where h_t is a (possibly nonlinear) function of the current system state and $\mathbf{n}_t$ is another i.i.d. noise sequence. Equation (2.2) and equation (2.3) constitute a dynamic state space model for the system where estimates of $\mathbf{x}_t$ are based on all available observations $\mathbf{z}_{1:t}$ up to time t.

2.2 Bayesian State Estimation

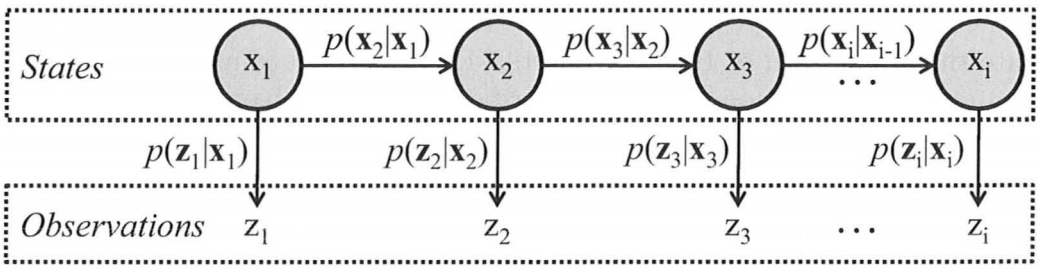

Figure 2.1: Dynamic state space model as a first-order Markov chain

To frame object tracking as a stochastic state estimation problem, the dynamic system is represented by a first-order Markov chain (Fig. 2.1) where equation (2.2) is described by a state transition probability density function (PDF) $p(\mathbf{x}_t|\mathbf{x}_{t-1})$, also called the *prior*, and equation (2.3) is described by a *likelihood* PDF $p(\mathbf{z}_t|\mathbf{x}_t)$. The probability of belief that the system is currently in state $\mathbf{x}_t$, given a series of independent observations $\mathbf{z}_{1:t}$ up to time t, can then be represented by a *posterior* PDF $p(\mathbf{x}_t|\mathbf{z}_{1:t})$, assuming the initial posterior $p(\mathbf{x}_0|\mathbf{z}_0) \equiv p(\mathbf{x}_0)$ is known. From a Bayesian perspective, knowledge of the posterior constitutes an optimal solution to the estimation problem (Arulampalam *et al.*, 2002).

Recursive filtering is a popular approach to estimating the posterior of a dynamic system, as it produces a new estimate with the arrival of each measurement, while considering all previous measurements without the need to reprocess them. This is generally facilitated through two stages:

1. **Prediction:** The prior PDF (based on the system dynamics defined by equation (2.2)) is used to propagate the posterior from the previous time step to the

current time t according to

$$p(\mathbf{x}_t|\mathbf{z}_{1:t-1}) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1})p(\mathbf{x}_{t-1}|\mathbf{z}_{1:t-1})d\mathbf{x}_{t-1}. \tag{2.4}$$

2. **Update:** The likelihood PDF (based on the observation model defined by equation (2.3)) is used to modify the posterior in (2.4) based on a new measurement according to

$$p(\mathbf{x}_t|\mathbf{z}_{1:t}) = \frac{p(\mathbf{z}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{z}_{1:t-1})}{p(\mathbf{z}_t|\mathbf{z}_{1:t-1})}. \tag{2.5}$$

Note that (2.4) uses the Chapman-Kolmogorov equation (Papoulis, 1984), and (2.5) uses Bayes rule and contains a normalization PDF $p(\mathbf{z}_t|\mathbf{z}_{1:t-1})$ defined as

$$p(\mathbf{z}_t|\mathbf{z}_{1:t-1}) = \int p(\mathbf{z}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{z}_{1:t-1})d\mathbf{x}_t. \tag{2.6}$$

While (2.4) and (2.5) conceptually represent the optimal Bayesian filtering solution, their integrals are computationally intractable and an analytic solution is not generally available. If the dynamic state space is linear and Gaussian, a tractable solution can be obtained using the Kalman filter. Similarly, if the state space is discrete and finite, grid-based methods can provide an optimal estimate. Unfortunately, these conditions rarely hold in the application of visual tracking and a sub-optimal solution is sought using alternative methods, such as the extended Kalman filter, unscented Kalman filter, approximate grid-based methods, or the *particle filter* (Doucet *et al.*, 2000).

2.3 The Particle Filter

An optimal Bayesian estimate is intractable because it is not possible to evaluate all potential parameter values of $\mathbf{x}_t$ in the state space. Particle filtering is an SMC technique (Doucet *et al.*, 2001) that implements the recursive Bayesian filtering approach described above, but circumvents the intractable integrals by selecting only a statistically relevant subset of possible state values for prediction and updating. Specifically, the posterior at time t is approximated by a set of N *discrete random samples* $\{\mathbf{x}_t^i\}_{i=1}^N$ and their corresponding *weights* $\{w_t^i\}_{i=1}^N$ (Fig. 2.2) where each sample-weight pair is referred to as a *particle* and the weight is a measure of quality that has been normalized such that $\sum_{i=1}^N w_t^i = 1$. Formally, the posterior is approximated by point masses according to

$$p(\mathbf{x}_t|\mathbf{z}_{1:t}) \approx \sum_{i=1}^{N} w_t^i \delta(\mathbf{x}_t - \mathbf{x}_t^i) \tag{2.7}$$

where $\delta(\cdot)$ is the Dirac delta function and as $N \rightarrow \infty$ the right side of (2.7) approaches the left side. In other words, the higher the number of particles in the set, the closer the approximation of the true posterior.

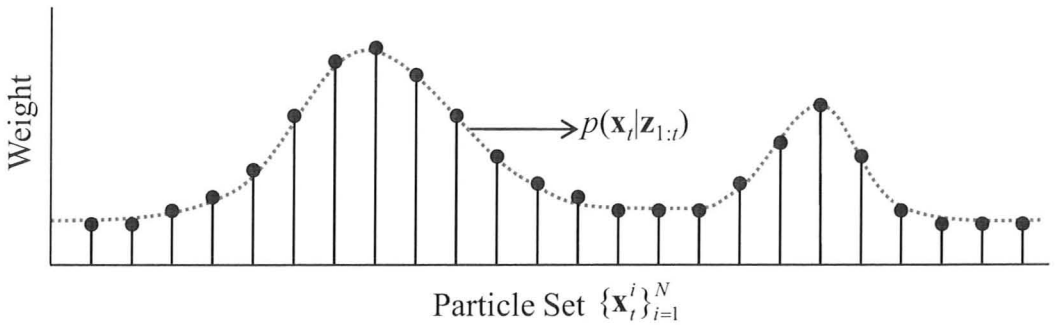

Figure 2.2: Posterior PDF approximated by uniformly distributed particles

The particle filter is applicable to a variety of dynamic state space models defined by equation (2.2) and equation (2.3), including those with nonlinear system dynamics models f_t and non-Gaussian noise $\mathbf{v}_{t-1}$, as well as a nonlinear observation model h_t, which can also contain non-Gaussian noise $\mathbf{n}_t$. A key advantage of the particle filter over other estimation techniques is its ability to simulate and track multiple state hypotheses and generates intelligent estimates of $\mathbf{x}_t$ based on the particle weights.

The remainder of this section is presented as follows. First, the various methods through which the particle set and weights can be used to generate an estimate for the state of a system are outlined. Next, the sequential importance sampling particle filter, which forms the basis for most SMC estimation techniques, is introduced. Finally, the concept of resampling is used to derive the sampling importance resampling particle filter, which is used in this work.

2.3.1 State Estimation

The particle set $\{\mathbf{x}_t^i\}_{i=1}^N$ and weights $\{w_t^i\}_{i=1}^N$ can be used to generate an estimate of the current system state $\hat{\mathbf{x}}_t$ using a number of methods. For example, the state of the highest-weighted particle

$$\hat{\mathbf{x}}_t = \mathbf{x}_t^1, \tag{2.8}$$

the average of the M highest-weighted particles

$$\hat{\mathbf{x}}_t = M^{-1} \sum_{i=1}^{M} \mathbf{x}_t^i, \tag{2.9}$$

or, the weighted average of the M highest-weighted particles

$$\hat{\mathbf{x}}_t = \sum_{i=1}^{M} w_t^i \mathbf{x}_t^i, \tag{2.10}$$

where $M \leq N$ and the particles have been sorted from highest-weighted to lowest-weighted and weights have been normalized. This work exclusively uses equation (2.10) for state estimation. Once particles and weights are available, state estimation is trivial. The challenge, however, is determining the ideal location of the particles in the state space and the optimal approach to generating their weights, as described below.

2.3.2 The SIS Filter

In the example shown in Fig. 2.2, the particles are distributed evenly across the state space. While this would certainly yield the most complete approximation of the posterior, significant computational resources would be wasted in simulation and measurement of many particles with extremely low weights (and consequently low probability of accurately estimating the system state). A more efficient approach is to concentrate particles in statistically likely areas of the state space, as shown in Fig. 2.3, sacrificing some robustness for accuracy and speed. To achieve this, the particle set must be periodically *propagated* such that the concentration of particles in the state space is proportional to the posterior.

Ideally, the particles would be *drawn* (i.e., selected) directly from the posterior:

$$\mathbf{x}_t^i \sim p(\mathbf{x}_t|\mathbf{z}_{1:t}). \tag{2.11}$$

Unfortunately, the posterior is the PDF being estimated and therefore cannot be sampled from. Instead, an alternative, easily-sampled function $q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{z}_t)$ called the *proposal* or *importance* PDF provides an approximation of the true posterior:

$$\mathbf{x}_t^i \sim q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{z}_t). \tag{2.12}$$

This is demonstrated in Fig. 2.4, where the particles are distributed in the state space according to the proposal PDF, but are still used to approximate the posterior.

The *sequential importance sampling* (SIS) particle filter is so named because of its ability to utilize the importance function to sequentially (i.e., recursively) propagate the particle set and weights with the arrival of each new measurement. The SIS weight-update equation is derived in Appendix A, and describes how particles sampled

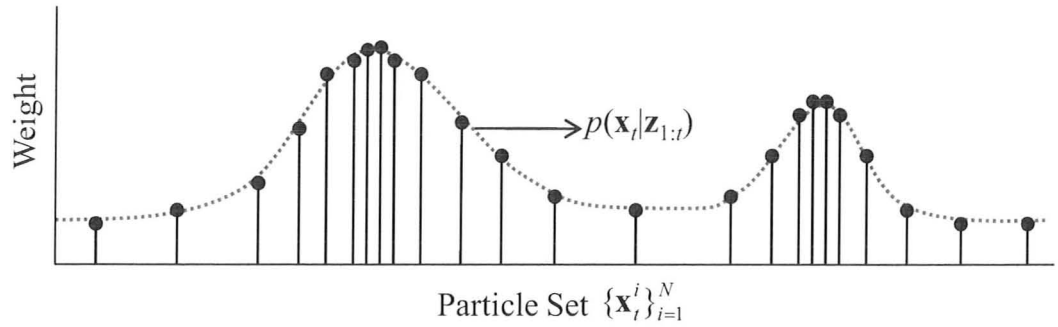

Figure 2.3: Posterior PDF approximated by particles drawn from the posterior PDF

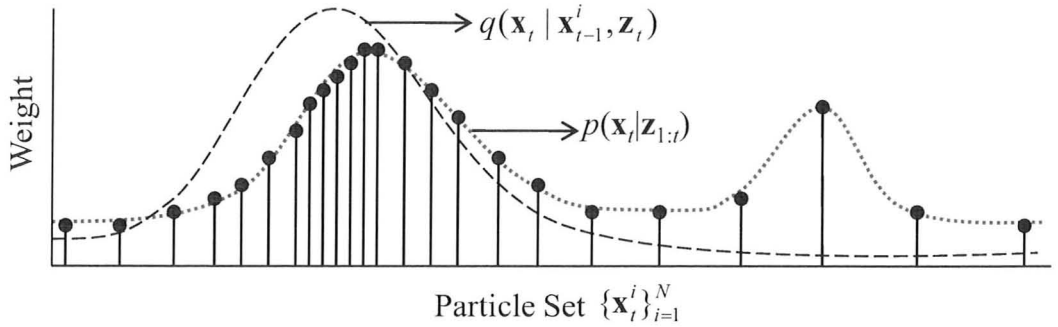

Figure 2.4: Posterior PDF approximated by particles drawn from the proposal PDF

from $q(\cdot)$ are recursively updated:

$$w_t^i \propto w_{t-1}^i \frac{p(\mathbf{z}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{z}_t)}. \tag{2.13}$$

Note that equation (2.13) relies solely on the current observation and previous state, meaning prior weights, observations, and states do not need to be stored or processed, and the memory and computational requirements of the SIS particle filter do not increase with time. Additionally, (2.13) can be directly used with equation (2.7) to approximate the posterior and yield an estimate for the current system state. One iteration of the SIS particle filter algorithm is shown below.

The choice of proposal function $q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{z}_t)$ significantly affects the efficacy of the particle filter and many have been proposed in the literature. For example, Doucet (1998) states that the optimal importance function minimizes the variance of the particle weights and can be shown to be $q_{opt}(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{z}_t) = p(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{z}_t)$. This proposal, however, is difficult to sample and leads to generally intractable integrals during the weight-update stage. As a result, the optimal importance function must

SIS Particle Filter Algorithm

procedure $[\hat{\mathbf{x}}_t, \{\mathbf{x}_t^i, w_t^i\}_{i=1}^N] = \text{SIS}(\{\mathbf{x}_{t-1}^i, w_{t-1}^i\}_{i=1}^N, \mathbf{z}_t)$

for $i = 1 : N$ **do**

- Propagate particle: $\mathbf{x}_t^i \sim q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{z}_t)$ *//Prediction*
- Simulate and measure weight: $w_t^i \propto w_{t-1}^i \frac{p(\mathbf{z}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{z}_t)}$ *//Update*

end for

- Estimate state $\hat{\mathbf{x}}_t$ using (2.8), (2.9), or (2.10) *//Estimation*

end procedure

be approximated (van der Merwe *et al.*, 2001) or an alternative, easy-to-sample function is employed. For example, motion features are introduced to the proposal by Odobez *et al.* (2006), and Rui and Chen (2001) describe an unscented particle filter (UPF), which integrates the unscented Kalman filter (UKF) into the proposal PDF. An appearance-guided particle filter (AGPF) is presented by Chang *et al.* (2008), which introduces several attractors into the state space to augment the proposal density with current visual cues and minimize particle drift.

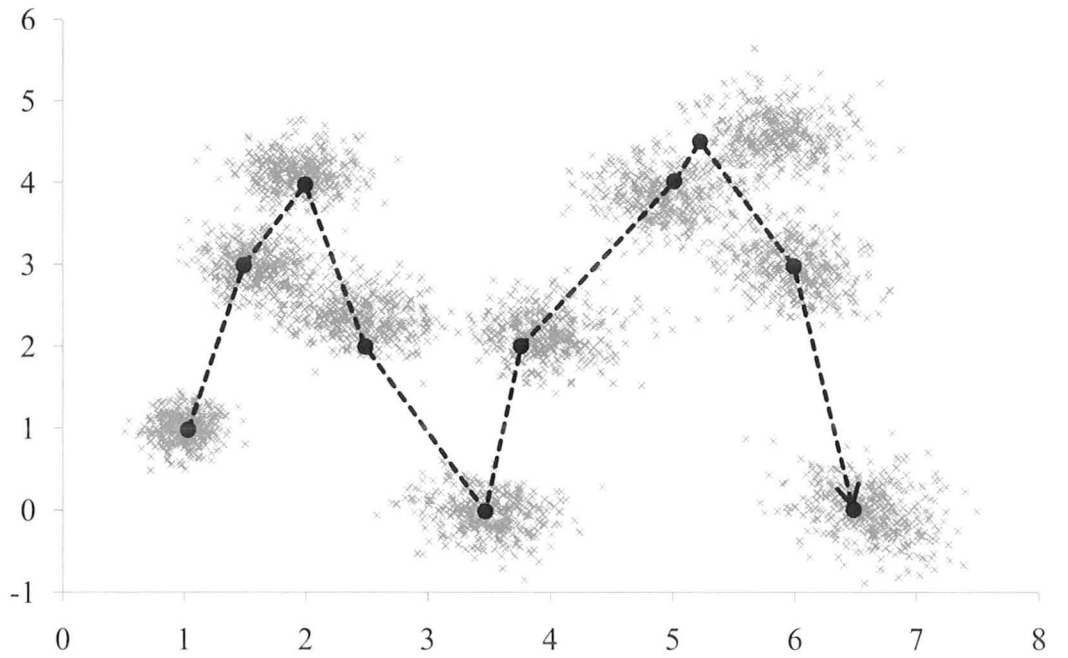

Figure 2.5: Particle propagation demonstration
As an object moves from left to right in a zigzag pattern, the particle set (shown at 10-frame intervals) propagates by clustering around predicted object locations.

The most common and convenient choice of importance function is the prior PDF $q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{z}_t) = p(\mathbf{x}_t|\mathbf{x}_{t-1}^i)$, which is easy to sample, provides a reasonable approximation of the posterior if it accurately reflects the system dynamics, and simplifies (2.13) to

$$w_t^i \propto w_{t-1}^i p(\mathbf{z}_t|\mathbf{x}_t^i). \tag{2.14}$$

Although it does not incorporate the most recent observation, this choice of proposal can be effective for visual tracking and is used throughout the remainder of this work. Particle propagation using the prior is demonstrated in Fig. 2.5, which shows the displacement of a particle set as it tracks an object with two DOFs using a model-based approach. As the tracking target moves through the state space, particles cluster around the object's path, with higher concentrations of particles overlapping with the most likely object locations according to the system dynamics.

2.3.3 Particle Degeneracy and Resampling

The SIS particle filter is prone to a phenomenon called *particle degeneracy*, wherein poorly-weighted particles become effectively useless as their weights approach zero after a few filter iterations (i.e., the variance of particle weights increases with time). This effect was quantified by Liu and Chen (1998) by introducing a measure of *effective sample size* N_{eff} that is proportional to the variance of the particle weights and can be approximated according to

$$N_{eff} \approx \frac{1}{\sum_{i=1}^{N}(w_t^i)^2}. \tag{2.15}$$

Because each particle represents a possible system state that is going to be simulated and measured, if N_{eff} becomes too small, significant computational resources would be wasted and the robustness of tracking would be impacted. Instead of a brute force approach that simply increases the particle count, a technique called *resampling* is employed to counteract particle degeneracy.

Conceptually, resampling discards poorly weighted particles and duplicates highly weighted particles while maintaining a fixed particle count. Formally, resampling generates a new set of particles $\{\mathbf{x}_t^{j*}\}_{i=1}^N$ by resampling (with replacement) N times from the existing particle set $\{\mathbf{x}_t^i\}_{i=1}^N$ such that $Pr(\mathbf{x}_t^{j*} = \mathbf{x}_t^i) = w_t^i$. This effectively results in an i.i.d. sample from the approximation of the posterior PDF defined by equation (2.7). Each time the particle set is resampled, all particle weights are reset so that $w_t^i = 1/N$. A number of resampling methodologies have been proposed, including stratified sampling, residual sampling (Liu and Chen, 1998), and *systematic resampling* (Kitagawa, 1996), which is used in this work and can be implemented for *O(N)* time complexity (Arulampalam *et al.*, 2002; Ripley, 1987).

Systematic resampling begins by constructing a cumulative density function (CDF) from the normalized particle weights $\{w_t^i\}_{i=1}^N$ then picking an random initial starting point $u_1 \sim \mathcal{U}[0, N^{-1}]$ where $\mathcal{U}[a, b]$ is a uniform distribution between a and b. The algorithm then steps through the CDF, starting from u_1, in steps of size $1/N$. The corresponding particle associated with each selected point on the CDF is added to the new particle set $\{\mathbf{x}_t^{j*}\}_{i=1}^N$. With this approach, highly-weighted particles can be selected multiple times whereas low-weighted particles may not be selected at all. Assuming the particle weights consistently represent the posterior PDF of the system, the end result is a particle set that probabilistically propagates over time such that the majority of particles fall around the peaks of the posterior and few computational resources are wasted on the simulation of unlikely hypotheses.

2.3.4 The SIR Filter

This work utilizes the *sampling importance resampling* (SIR) particle filter introduced by Gordon *et al.* (1993), which applies systematic resampling on every filter iteration and uses the prior $p(\mathbf{x}_t|\mathbf{x}_{t-1}^i)$ as the proposal PDF. Resampling is performed after the prediction, update, and $\hat{\mathbf{x}}_t$ estimation stages. Additionally, since a new set of particles is generated at each time point, there is no need to consider previous weights w_{t-1}^i and equation (2.14) simplifies to

$$w_t^i \propto p(\mathbf{z}_t|\mathbf{x}_t^i). \tag{2.16}$$

The SIR particle filter is demonstrated visually in Fig. 2.6 and described algorithmically below with the four stages (prediction, update, estimation, and systematic resampling) indicated as comments.

Although the SIR particle filter has been shown to be a robust and effective approach to dynamic state estimation, it suffers from some shortcomings. *Sample impoverishment*, a condition wherein many particles converge to represent the same location in the state space, can occur when (2.2) contains very little noise. Furthermore, as described above, the SIR filter does not include the most recent observation when propagating particles during the prediction step, meaning the proposal PDF may not accurately reflect the true posterior and state estimation can be sensitive to *outliers*. Finally, when dealing with state estimation of systems described by a kinematic chain, an incorrectly estimated parameter at the root of the chain can propagate and perturb all dependent parameter estimates; a problem that is often circumvented through the use of multiple or hierarchical particle filters (Brandao *et al.*, 2006; Stenger *et al.*, 2006; Qu and Schonfeld, 2007).

Many of these problems can be avoided altogether by utilizing a sufficiently large particle set. However, this exposes the primary drawback of particle filtering approaches: the computational complexity associated with simulating and weighing

SIR Particle Filter Algorithm

procedure $[\hat{\mathbf{x}}_t, \{\mathbf{x}_t^i\}_{i=1}^N] = \text{SIR}(\{\mathbf{x}_{t-1}^i\}_{i=1}^N, \mathbf{z}_t)$
  for $i = 1 : N$ **do**
    • Propagate particle: $\mathbf{x}_t^i \sim p(\mathbf{x}_t|\mathbf{x}_{t-1}^i)$ *//Prediction*
    • Simulate and measure weight: $w_t^i = p(\mathbf{z}_t|\mathbf{x}_t^i)$ *//Update*
  end for
  • Estimate state $\hat{\mathbf{x}}_t$ using (2.8), (2.9), or (2.10) *//Estimation*
  • Normalize weights such that $\sum_{i=1}^N w_t^i = 1$
  • Initialize new particle set: $\{\mathbf{x}_t^{j*}\}_{j=1}^N$
  • Initialize CDF: $c_1 = 0$
  for $i = 2 : N$ **do**
    • Build CDF: $c_i = c_{i-1} + w_t^i$
  end for
  • Choose random point $u_1 \sim \mathcal{U}[0, N^{-1}]$
  for $j = 1 : N$ **do** *//Resampling*
    • Initialize CDF index: $i = 0$
    • Select point on CDF: $u_j = u_1 + N^{-1}(j-1)$
    while $u_j > c_i$ **do**
      • Find corresponding particle: $i = i + 1$
    end while
    • Add particle to new set: $\mathbf{x}_t^{j*} = \mathbf{x}_t^i$
  end for
  • Replace old particle set: $\{\mathbf{x}_t^i\}_{i=1}^N = \{\mathbf{x}_t^{j*}\}_{j=1}^N$
end procedure

multiple hypotheses - a task that becomes exponentially more demanding as the dimensionality of the state vector increases. Existing solutions aim to lower the dimensionality of the problem (Wu *et al.*, 2001) or reduce the particle count (Bray *et al.*, 2007) while maintaining tracking quality, but the computational demands remain. This framework addresses the inherent computational complexity by parallelizing particle simulation and measurement on a GPU.

2.4 Particle Filter Variations and Alternatives

Entire books have been written on Bayesian tracking and estimation techniques (Candy, 2009; Ristic *et al.*, 2004) and extensive research has been done toward modification and optimization of the particle filter. Here, a brief introduction of particularly popular approaches is provided with the goal of highlighting the strengths and weaknesses of the SIR filter, used in this work.

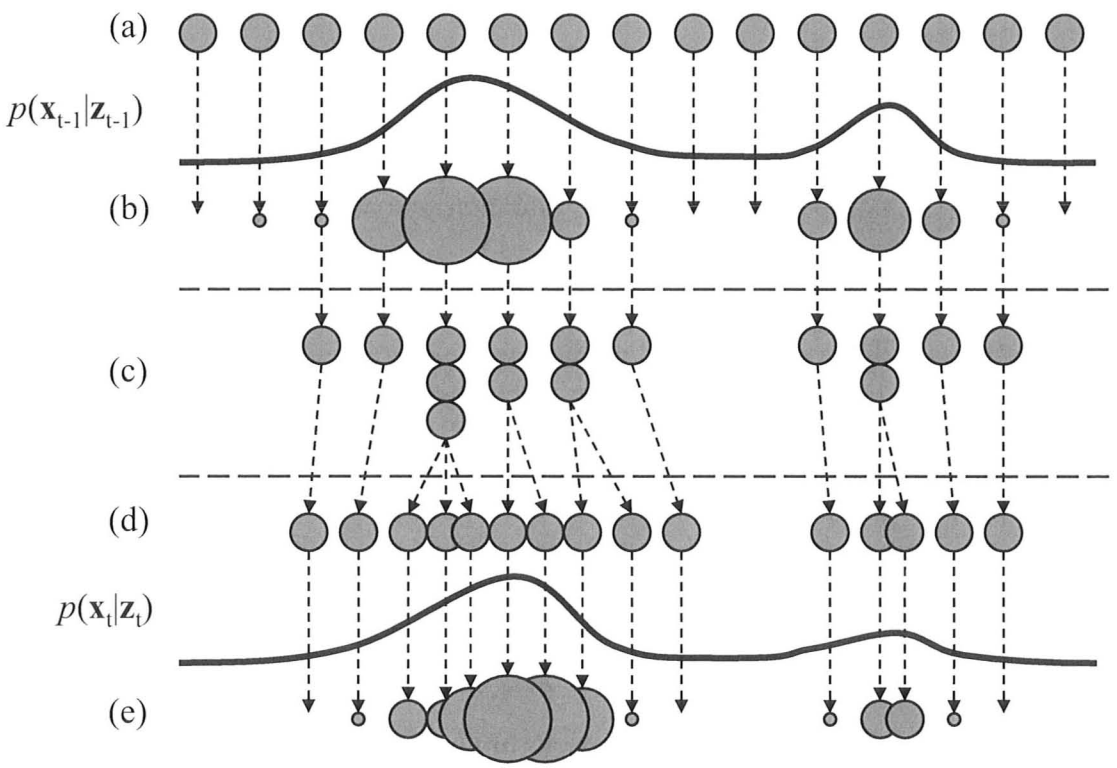

Figure 2.6: Sampling importance resampling demonstration
Single-parameter-state particle resampling with $N = 15$, $i = 1 \ldots N$, size of particle indicates weight. (a) $\{\mathbf{x}_{t-1}^i\}$, $\{w_{t-1}^i = N^{-1}\}$: particles uniform across the state space. (b) $\{\mathbf{x}_{t-1}^i\}$, $\{w_{t-1}^i = p(\mathbf{z}_{t-1}|\mathbf{x}_{t-1}^i)\}$: particle weights measured. (c) $\{\mathbf{x}_{t-1}^i\}$, $\{w_{t-1}^i = N^{-1}\}$: particles resampled to better represent the state space. (d) $\{\mathbf{x}_t^i\}$, $\{w_{t-1}^i = N^{-1}\}$: particles propagated based on $p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)$ to predict current posterior. (e) $\{\mathbf{x}_t^i\}$, $\{w_t^i = p(\mathbf{z}_t|\mathbf{x}_t^i)\}$: particle weights updated based on new measurements.

2.4.1 Optimal Estimation Algorithms

The class of optimal algorithms are so named due to their ability to generate an optimal solution to a Bayesian estimation problem (i.e., an exact representation of the posterior PDF) under certain restrictive conditions. Two popular examples, the Kalman filter and grid-based methods, are introduced below.

Kalman Filter

The recursive Kalman filter (Ho and Lee, 1964) provides an optimal solution in cases where a *linear Gaussian* assumption applies to the dynamic state space model. In other words, h_t and f_t in equations (2.2) and (2.3), respectively, must be known linear

functions, and noise parameters $\mathbf{v}_{t-1}$ and $\mathbf{n}_t$ must be Gaussian distributions with known parameters. These assumptions imply that the posterior is always Gaussian (and is therefore parameterized by a mean and covariance), which is often not the case in high-dimensional model-based tracking applications. Nonetheless, for linear Gaussian dynamic state spaces, no algorithm can produced superior results to the Kalman filter (Arulampalam *et al.*, 2002), making them a popular target for research.

Grid-Based Methods

Grid-based methods formulate the Bayesian tracking problem in a manner similar to the particle filter, but are applicable only when the state space is discrete and has a finite number of states (Arulampalam *et al.*, 2002). Like the Kalman filter, as long as these conditions hold and the prior and likelihood PDFs are known, grid-based methods provide an optimal solution to the tracking problem; however, most visual tracking applications deal with a continuous state space, thus limiting the utility of these approaches.

2.4.2 Suboptimal Estimation Algorithms

Suboptimal algorithms, such as the particle filter, are employed when dealing with applications that do not adhere to the restrictive optimal assumptions above. Here, two alternatives to the particle filter are presented, based on the Kalman filter and grid-based methods, respectively.

Extended Kalman Filter

The extended Kalman filter (EKF) applies the principle of the Kalman filter to nonlinear functions through local linearization; specifically, by expressing a nonlinear function as one or more terms of its Taylor expansion. The result approximates the posterior PDF as Gaussian, which can be effective in cases where nonlinearities are weak, but in cases of bi-modal or highly skewed distributions, the filter can diverge, and a particle filter generally provides a superior estimate.

Approximate Grid-Based Methods

Approximate grid-based methods apply standard grid-based techniques to continuous, infinite dynamic state spaces by truncating and discretizing the space into "cells." If a sufficiently dense grid is utilized, and the continuous state space is properly approximated, this approach can lead to accurate estimations of the posterior. Unfortunately, unless the nature of the state space is known in advance, the entire space must be evenly partitioned and significant computational resources can be wasted (i.e., it is

not possible to increase grid resolution for likely areas of the state space), unlike the particle filter which dynamically assigns particles to high-probability regions of the posterior PDF.

2.4.3 Variations of the Particles Filter

The particle filter has been adapted to fit a wide array of applications through a variety of modifications and augmentations that aim to increase efficiency, accuracy, or robustness. In most cases, variations of the particle filter can be derived from the SIS filter by specifying a particular importance function and resampling technique. Chen (2003) describes many such modifications, such as the auxiliary, rejection, regularized, Rao-Blackwellization, MCMC, and mixture particle filters. Below, the auxiliary and regularized particle filters are introduced, followed by a description of the classes of adaptive and hierarchical particle filters that can be well-suited to high-dimensional tracking.

Auxiliary Particle Filter

Particle filters that use the prior PDF as an importance function, such as the SIR particle filter, do not incorporate the most recent observation during the prediction stage, which can lead to poor sampling of the posterior. The auxiliary particle filter, introduced by Pitt and Shephard (1999), circumvents this deficiency by introducing an *auxiliary variable* that increases the dimensionality of the filter, but can be used to aid in simulation by acting as an index for particles. The auxiliary variable is integrated into the proposal PDF and facilitates simulation of likely particles based on the current measurement, meaning particles are more likely to approximate the current posterior. The auxiliary particle filter has been shown to perform better than the SIR filter when process noise is small, but can degrade accuracy when noise is large. Furthermore, the additional complexity of the auxiliary filter makes it computationally slower than the SIR filter.

Regularized Particle Filter

Another drawback of the SIR filter outlined above is the possibility of sample impoverishment, wherein all particles in the set converge to a single location in state space in processes with little noise. This loss of diversity occurs because particles are resampled from the existing, discrete particle set, as opposed to a continuous distribution. The regularized particle filter, proposed by Musso *et al.* (2001), uses a modified resampling stage to avoid impoverishment by resampling from a continuous distribution based on the discrete particle set and a variable symmetric PDF. Although the

regularized particle filter performs with similar computational complexity to the SIR filter, it is only practical in scenarios with extensive sample impoverishment.

Adaptive Particle Filters

The broad class of adaptive particle filters describes algorithms that utilize dynamic particle allocation or state space distributions to achieve superior results. For example, Fox (2003) and Ben and Jiantong (2010) vary the size of the particle set based on uncertainty, thus decreasing sample size and limiting the computational complexity during times when tracking is satisfactorily reliable. Junxia *et al.* (2008) use a fixed number of particles in a full body pose tracking application, but adaptively allocate particles to various body parts as needed. In (Wang *et al.*, 2009; Huang and Llach, 2008; Park *et al.*, 2008), an autoregressive system dynamics model is used that can automatically select an appropriate model order and covariance for optimal tracking. Of course, filters that adaptively change the size of the particle set are prone to inconsistent computation time, which is not suitable in many applications. Algorithms with intelligently adaptive prior or likelihood distributions, however, can provide increased tracking quality at minimal computational cost.

Hierarchical Particle Filters

Hierarchical particle filters are often employed for high-dimensional tracking tasks, where there are dependencies between the parameters being estimated, such as body or hand tracking. In general, hierarchical filters partition the tracking task into two or more stages to exploit natural correlations in the tracking target's architecture and increase the quality of tracking. For example, Brandao *et al.* (2006) partition the parameter space into a tree of subspaces based on the nature of the tracking target and estimate the states of the resulting groups separately, whereas Dong and DeSouza (2009) use a two-level hierarchy of particle filters to distinguish between coarse and fine grained tracking tasks. While hierarchical filters require multiple iterations for state estimation, they can lower the number of particles required for applicable tracking tasks.

2.5 Model-Based Tracking With a Particle Filter

There are particular requirements and challenges associated with the use of a particle filter in the application of 3D model-based tracking, specifically in the weight-update stage. With the state of the tracking target represented by (2.1), simulating a particle set $\{\mathbf{x}_t^i\}_{i=1}^N$ involves configuring a 3D model of the tracking target in the pose described by each of the N particles and projecting the configurations onto planes referred to

as *particle images*. Measuring particle weights requires a quantifiable comparison between each of the particle images and the current frame of the video sequence, referred to as *evaluation*. In the context of 3D model-based tracking, model simulation and evaluation comprise the weight update stage of the particle filter.

2.5.1 Choosing a Motion Dynamics Model

In a tracking application, the prior (system dynamics) PDF $p(\mathbf{x}_t|\mathbf{x}_{t-1})$ describes the motion of an object. In other words, it is a model of the object's motion that can predict the current pose of a tracking target based on its previous pose and movement. In the SIR particle filter, the prior is used to propagate the particle set $\{\mathbf{x}_{t-1}^i\}_{i=1}^N$ to $\{\mathbf{x}_t^i\}_{i=1}^N$, with the goal of making each particle a realistic estimate of the tracking target's next position.

The choice of motion model (2.2) therefore has significant influence on tracking quality. In this work, three motion models with the form $\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{v}_t$ are considered:

- a random walk model with $\mathbf{v}_t \sim \mathcal{N}(0, \Sigma_{t-1})$;
- a first-order motion model with $\mathbf{v}_t \sim \mathcal{N}(\Delta\hat{\mathbf{x}}_{t-1}, \Sigma_{t-1})$; or,
- a second-order motion model with $\mathbf{v}_t \sim \mathcal{N}(\Delta\hat{\mathbf{x}}_{t-1} + \frac{1}{2}(\Delta\hat{\mathbf{x}}_{t-1} - \Delta\hat{\mathbf{x}}_{t-2}), \Sigma_{t-1})$

where $\mathcal{N}(\mu, \Sigma)$ is a normal distribution with mean μ and covariance Σ, and $\Delta\hat{\mathbf{x}}_i = \hat{\mathbf{x}}_i - \hat{\mathbf{x}}_{i-1}$. The application of a first- or second-order motion model to the particle set is sometimes referred to particle *drift and diffusion*, where drift refers to displacement of the particle according to the mean of the distribution (deterministic component based on motion), and diffusion refers to the scattering of particles according to the covariance of the distribution (random component to account for uncertainty in predicted motion). In all motion models, Σ_{t-1} can be adjusted so that all state parameters have a common transition variance or so that each parameter has a unique transition variance. Additionally, all parameters need not use the same motion model; for example, if tracking an object that is expected to move linearly and rarely change its internal joint flexions, it may be desirable to use a second-order motion model with a large variance for global state parameters and a random walk model with low variance for local parameters.

2.5.2 Particle Image Evaluation

If particle images accurately reflect the color, texture, lighting, and shading of the actual tracking target, a direct pixel-to-pixel comparison is feasible; however, this is not generally the case and a number of other approaches have been proposed in the literature. For example, Maggio *et al.* (2007) extract color and orientation histograms and Wang *et al.* (2007) use a similarity measure based on a spatial-color

mixture of Gaussians (SMOG) appearance model that considers the spatial layout of colors, augmented by a shape similarity measurement. In most cases, some degree of image processing is performed to extract features (edges, corners, silhouettes, etc.) from the particle images and current frame, which can be compared to generate a more meaningful weight than raw pixel data. Although this comes with the price of additional computational complexity, these types of image processing tasks are ideal candidates for parallel processing on a GPU. The type of feature extraction performed varies depending on the nature of the tracking application. In this work, image segmentation, silhouette detection, and edge detection are employed and are further described below.

Image Segmentation

Image segmentation involves partitioning an image into two or more regions, generally to facilitate further processing (Shapiro *et al.*, 2001). In the context of model-based tracking, segmentation is used to partition a video frame into foreground and background components, where the foreground contains only the tracking target and the background is suppressed, providing a more meaningful comparison with the particle images. Segmentation generally involves identifying regions using colour, texture, edge, contour, or motion information.

This work focuses solely on tracking objects moving in front of a stationary camera with a static background; therefore, the tracking target can be segmented from the background using a simple *background subtraction* technique. The frame background is initially "learned" by observing one or more frames of unobstructed background and recording the range of colour intensities that each pixel location achieves. During tracking, each pixel is compared to the corresponding intensity ranges in the learned background. Pixels that closely resemble the learned background are suppressed, whereas all other pixels are preserved, effectively segmenting the tracking target. Formally,

$$p^{seg}_{(i,j)} = \begin{cases} p^{in}_{(i,j)}, & p^{in}_{(i,j)} > b^{high}_{(i,j)} + T_{seg} \\ p^{in}_{(i,j)}, & p^{in}_{(i,j)} < b^{low}_{(i,j)} - T_{seg} \\ 0, & \text{otherwise} \end{cases}$$

where $p^{in}_{(i,j)}$ is the input pixel from the video sequence at location (i,j), $b^{high}_{(i,j)}$ is the upper bound of the range of colour intensities for the corresponding pixel location in the learned background, $b^{low}_{(i,j)}$ is the lower bound, T_{seg} is a threshold, and $p^{seg}_{(i,j)}$ is the resulting segmented pixel.

Silhouette Detection

Directly comparing the pixels of a segmented tracking target to particle images is only effective if the colour, lighting, shading, and texture of both are very similar. This is often not the case, requiring these features to be suppressed while preserving the geometry of the object. This can be done by reducing all particle images and the segmented tracking target to their silhouettes. Using the same notation as above:

$$p^{sil}_{(i,j)} = \begin{cases} S, & p^{seg}_{(i,j)} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

where $p^{sil}_{(i,j)}$ is a pixel in the resulting silhouette map and S is an arbitrary silhouette colour intensity.

Edge Detection

Reducing the projection of an object to its silhouette can discard valuable information about its orientation, particularly when the object is prone to self-occlusion or has discontinuities in depth, such as the object shown in Fig. 2.7. To preserve as much orientation information as possible, edge detection is applied to the segmented tracking target and particle images, and resultant edges are overlaid on the silhouette.

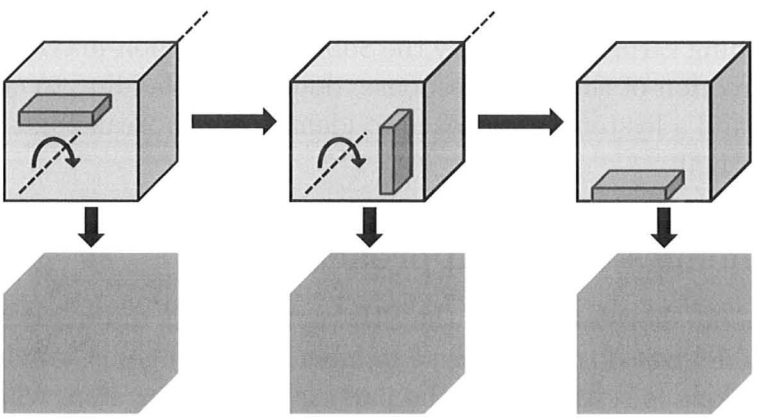

Figure 2.7: Orientation ambiguity due to silhouette extraction
The orientation of the object at three different rotations is indiscernible due to the loss of edge information

Many approaches to edge detection have been proposed over the years, most of which involve the convolution of a *filter* or *mask* with pixel data, such as the Prewitt detector, the Roberts detector, or the *Sobel detector*. The Sobel detector (Gonzalez

and Woods, 1992), used in this work, involves the convolution of two separate filters

$$\mathbf{S}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \mathbf{S}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

that generate a response from edges in the x- and y-direction, respectively. Formally, the edge response in the x-direction $s^x_{(i,j)}$ and y-direction $s^y_{(i,j)}$ can be expressed as:

$$s^x_{(i,j)} = -p^{seg}_{(i-1,j-1)} + p^{seg}_{(i+1,j-1)} - 2 \cdot p^{seg}_{(i-1,j)} + 2 \cdot p^{seg}_{(i+1,j)} - p^{seg}_{(i-1,j+1)} + p^{seg}_{(i+1,j+1)} \quad (2.17)$$

$$s^y_{(i,j)} = p^{seg}_{(i-1,j-1)} + 2 \cdot p^{seg}_{(i,j-1)} + p^{seg}_{(i+1,j-1)} - p^{seg}_{(i-1,j+1)} - 2 \cdot p^{seg}_{(i,j+1)} - p^{seg}_{(i+1,j+1)} \quad (2.18)$$

An edge pixel is then identified if the sum of the absolute values of the x and y edge responses exceeds a threshold. Edge pixels are overlaid on the silhouette map according to:

$$p^{out}_{(i,j)} = \begin{cases} E, & |s^x_{(i,j)}| + |s^y_{(i,j)}| > T_{edge} \\ p^{sil}_{(i,j)}, & \text{otherwise} \end{cases} \quad (2.19)$$

where $p^{out}_{(i,j)}$ is a pixel in the resulting edge and silhouette map and E is an arbitrary edge colour intensity.

Supplemental results, presented in Appendix B, utilize the more computationally demanding *Canny edge detector* (Canny, 1986), which involves the convolution of a Gaussian smoothing kernel, followed by the Sobel detector, non-maximum suppression based on the direction of the Sobel response, double thresholding to classify edges as *weak* or *strong*, and a hysteresis approach to identify which weak edges should remain in the final edge map.

2.5.3 Challenges of the Approach

Beyond the general shortcomings of the SIR particle filter discussed above, this approach to 3D model-based tracking has a number of challenges. Complete parallelization of the technique is restricted by the particle resampling step, which requires all weights to be available, thus introducing a bottleneck. The issue of parallelizing the particle filter has been addressed by Sankaranarayanan *et al.* (2008); Medeiros *et al.* (2008); Gumpp *et al.* (2006). In the GPU-accelerated tracking system described in this work, only the weight-update stage, by far the most computationally intensive aspect of the technique, is parallelized. The particle filter sacrifices some accuracy for robustness, and is prone to a *jitter* effect due to the constant propagation of particles, even when the tracking target is not moving. This can be addressed by *smoothing* (i.e., averaging) estimates.

Accurate evaluation of the particle images is dependent on a number of factors. Consistent lighting is essential for successful background extraction, and self-occlusion should be avoided, particularly for articulated objects. Self-occlusion is a challenge in any articulated model-based tracking task and is often addressed by relying on multiple-camera integration (Sundaresan and Chellappa, 2009). Although the proposed GPU-accelerated approach could be adapted to include additional views, this work focuses solely on monocular tracking.

Of course, model-based tracking is only viable when a 3D model of the tracking target is available, which may not always be the case. Furthermore, the model must accurately reflect the object's geometry and the motion model must adequately represent its motion. Finally, the pixel-by-pixel approach to weight calculation is most responsive when small changes in particle state parameters lead to significant changes in particle images; for example, rotation or translation in the same plane as the image. Out-of-plane rotation and translation can cause almost insignificant changes to resulting particle images, meaning more ambiguity in estimation.

Perhaps the most significant drawback of particle filtering approaches to 3D model-based tracking is its computational complexity. Each particle in the set requires a 3D model of the tracking target to be rendered to a particle image, feature extraction must be performed on that image, and the resulting feature map must be compared, pixel by pixel, to the feature map of the current frame to generate a weight. Furthermore, the accuracy of tracking increases with the particle count, meaning a large particle set is highly desirable. Overcoming this computational burden through parallelization of particle evaluation is the key tenet of this work.

2.5.4 Case Study: Model-Based Hand Tracking

Articulated hand tracking is used as an example application in Chapter 5 and is therefore introduced here as a case study in 3D model-based tracking. There are countless applications for hand tracking, primarily in the domains of HCI or gesture recognition. For example, vision-based sign language recognition could greatly improve accessibility for the deaf. Accurate hand pose estimation can provide a natural interface to interact with a virtual 3D objects, perhaps in a computer-aided design (CAD) application, or give an operator direct control over a robotic arm, for example, in a remote surgery application. An excellent summary of hand tracking applications and approaches is provided by Wu and Huang (2001).

Most models of the human hand (Fig. 2.8) utilize 27 DOFs (Albrecht *et al.*, 2003): six in the wrist (global rotation and translation), two in the trapeziometacarpal (TM) joint at the base of the thumb, one in the interphalangeal (IP) joint at the top of the thumb, two in the metacarpophalangeal (MP) joints at the base of each finger and middle of the thumb, one in each of the proximal interphalangeal (PIP) joints in the

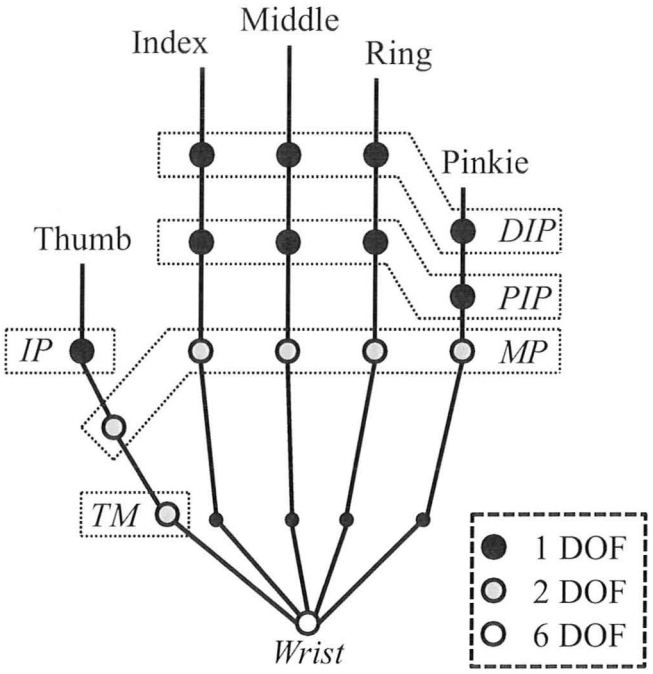

Figure 2.8: A 27-DOF skeletal hand model

middle of each finger and one in each of the distal interphalangeal (DIP) joints at the top of each finger.

The natural angular range for each joint can be integrated into the model to reduce the search space during estimation, for example, $0° \leq \theta_{MP} \leq 90°$. Additionally, dependencies between angles can be used to reduce the dimensionality of the problem. For example, it is a natural constraint of human hand motion that $\theta_{DIP} \approx \frac{2}{3}\theta_{PIP}$, meaning only the angle of the PIP needs to be estimated. The joints are connected by finger segments called phalanxes, forming a kinematic chain, as shown in Fig. 2.9. In other words, each joint angle is measured relative to the orientation of its parent phalanx, not relative to a global reference. Skin is attached to phalanxes and defined by geometric primitives, such as splines, cylinders, or polygon meshes.

The challenge of model-based hand tracking or pose estimation is finding an accurate and computationally efficient method to "fit" the model to images in a video sequence, while dealing with issues such as severe self-occlusion, depth ambiguity, and complex textures and shading. A wide variety of approaches have been proposed in the literature, most of which involve off-line generation of a 3D model and on-line tracking that aims to find the set of state parameters that minimizes the matching error between model features and extracted features from images in a video sequence (Erol *et al.*, 2005). A common single-hypothesis approach is to define an objective

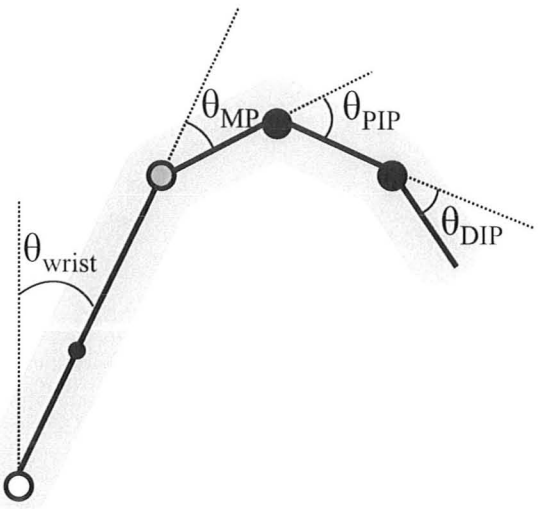

Figure 2.9: Wrist and index finger as a kinematic chain

function that can be minimized to estimate the hand pose using optimization techniques (de La Gorce *et al.*, 2008; Bray *et al.*, 2007; Heap and Hogg, 1996). A wide variety of multiple hypothesis approaches have been proposed as well, often involving the application of a Bayesian filter (Stenger *et al.*, 2006), such as the unscented Kalman filter (Stenger *et al.*, 2001), and the particle filter (Bretzner *et al.*, 2002; Gumpp *et al.*, 2006; Brandao *et al.*, 2006; Cui and Sun, 2004; Wu *et al.*, 2001; Chang *et al.*, 2008; Weng *et al.*, 2008). In some cases, the dimensionality of the problem is reduced using principle component analysis (PCA) (Kato *et al.*, 2006) or nonparametric belief propagation (NBP) (Sudderth *et al.*, 2004). Additionally, depth ambiguity can be resolved by integrating two or more cameras (Delamarre and Faugeras, 1998; Ueda *et al.*, 2003) during tracking.

Chapter 3

GPU Computing

As the GPU has evolved from a fixed-function graphics processor into a fully programmable, highly-parallel, manycore device, its popularity as a general-purpose platform for computationally intensive tasks has escalated considerably. A complete overview of the broad field of GPU computing is beyond the scope of this thesis; instead, the focus is limited primarily to material that will clarify and justify the proposed approach to model-based visual tracking described in Chapter 4. Specifically, the hardware architecture of the NVIDIA GT200-series GPU and how it can be exploited for high-performance parallel computing using NVIDIA CUDA and Microsoft Direct3D is detailed. Note that portions of this chapter appear in (Brown and Capson, 2010b).

The chapter is organized as followed. An introduction and brief history of GPGPU and its evolution to modern GPU computing is given in Section 3.1, followed by a review of similar applications in Section 3.2. The NVIDIA GT200 GPU architecture is described in Section 3.3 and the fundamentals of GPU programming with CUDA are introduced in Section 3.4. Section 3.5 presents Direct3D concepts and terminology, and Section 3.6 describes how these topics are relevant to 3D model-based tracking.

3.1 Introduction to GPU Computing

Traditionally, the microprocessor industry has improved performance by increasing clock frequencies and consequently raising power consumption with each architecture generation. Within the last decade, however, the industry has acknowledged the existence of a *power wall*, meaning further increase in power consumption is providing diminishing returns. This, among other factors, has triggered a paradigm shift and moved the industry in a new direction: parallel processors. Nowhere is this more evident than the architecture of modern manycore GPUs (generally 128 to 480 cores), which have outpaced multicore CPUs (generally two to six cores) in terms of peak

FLOPS and memory bandwidth. For example, NVIDIA's current flagship consumer-level GPU, the GTX 480, has 480 parallel processor cores, theoretical single-precision peak performance of 1.3 TFLOPS and memory bandwidth of 177.4 Gbyte/s, whereas the Intel Core i7-960 (similarly priced at the time of writing), offers four cores, 51.2 GFLOPS, and a memory bandwidth of 25.6 Gbyte/s.

This performance differential is largely due to a difference in fundamental design principles of the GPU and CPU. The majority of a GPU's transistors are dedicated to optimizing multi-threaded performance, with an emphasis on arithmetically intense, high-throughput numerical computation, whereas the strength of a multicore CPU lies in optimization of single-thread performance, through branch prediction, flow control, caching, and other mechanisms. Driven by the rapidly increasing demands of the video gaming market, the majority of GPUs forgo double-precision floating point accuracy and low-latency instruction execution to provide superior single-precision, high-throughput computation. The GPU is therefore best-suited to serve as a *co-processor* to the CPU, utilized only when needed for parallelizable, computationally demanding portions of an application.

Current GPUs are the culmination of more than a decade of hardware and software evolution. Early GPUs, which started to emerge around 1996 at the consumer level, were fixed-function pipelines (Fig. 3.1), designed solely to convert geometric primitives (vertex data) into pixels in a frame buffer through a series of steps, including vertex processing, rasterization, and fragment processing (Owens *et al.*, 2008). Each stage of the pipeline could be done in parallel, providing a level of *task parallelism*, and multiple processor cores at each stage provided an additional level of *data parallelism*. Originally, the stages could be configured (e.g., the location and intensity of a light source could be configured during shading) but their function remained fixed (e.g., the manner in which a light source affected a vertex could not be modified), greatly limiting their utility in non-graphics applications. Additionally, because each stage was executed by dedicated hardware, load-balancing was a challenge in applications that were dominated by a single stage.

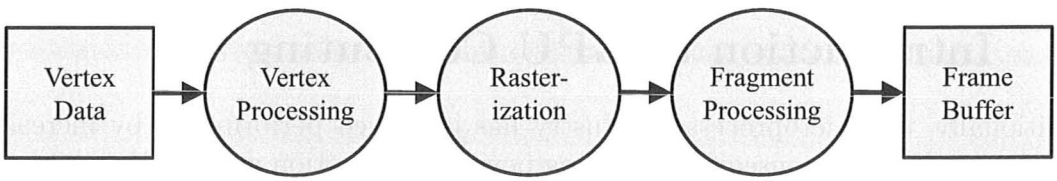

Figure 3.1: Graphics pipeline

As the pipeline architecture evolved, the stages became increasingly programmable. Minor programming capabilities in the vertex stage in 2001 matured into fully programmable vertex and shader processors by 2002. By 2006, a unified shader model

had emerged that provided a common instruction set for all stages of the pipeline. The benefit is programmable hardware, able to partition resources between all stages of the graphics pipeline and avoid becoming overburdened by a single stage. Furthermore, within the last four years, this programmability has matured to a point where it can be harnessed by application developers for tasks far beyond the capabilities of the earlier pipeline, while preserving the data- and task-parallelism that is fundamental to a GPU's operation. This paradigm is often described as single-program multiple-data (SPMD), as each programmable unit executes the same program simultaneously on multiple pieces of data.

Developing effective parallel hardware is futile without software that is able to exploit it. This is becoming increasingly evident as the vast majority of legacy code, written for sequential execution on a single-core CPU, is no longer scaling with new iterations of CPU architectures that introduce additional cores instead of accelerating clock speeds. In the early days of GPU programming (Owens *et al.*, 2007), a great deal of creativity was required as programmers were forced to frame their algorithms as graphics applications (using elements such as vertices, shading operations and pixels) to achieve *general purpose computation on the GPU (GPGPU)*. For example, in the very early days of desktop graphics processors, Lengyel *et al.* (1990) used standard graphics hardware for robot motion planning by packaging data as bitmaps that could be fed directly to a graphics processor's frame buffer for parallel rasterization. In (Hopf and Ertl, 1999), wavelet transformations are implemented on the fixed-function graphics pipeline using OpenGL's texture scaling and filtering operations.

As the programmability of graphics hardware increased, its potential for non-graphics applications became more apparent. With the release of Microsoft's Pixel Shader 1.0 as part of Direct3D 8.0 in 2001, the GPU became much easier to exploit using shading languages, such as Cg, high-level shading language (HLSL), and OpenGL shading language. The GPU soon became a target for a variety of computationally demanding applications, including matrix multiplication (Larsen and McAllister, 2001), physics simulations (Harris *et al.*, 2003), data mining (Sun *et al.*, 2003), and numerous image processing (Rumpf and Strzodka, 2001) and computer vision tasks (Fung *et al.*, 2002). Once the graphics pipeline stages were unified in 2006 by Pixel Shader 4.0 in Direct3D 10.0 and single-precision floating point arithmetic was introduced on the GPU, NVIDIA CUDA and similar platforms emerged as a solution beyond GPGPU, providing complete programmability of the entire graphics pipeline. This new era has become known as *GPU computing*, representing the GPU's maturation into a fully-programmable, general purpose, massively parallel computational resource capable of competing with tradition HPC platforms, such as CPU clusters, supercomputers, and field-programmable gate arrays (FPGAs) Cope *et al.* (2010).

3.2 Similar Applications and Literature Review

CUDA and GPU computing is well-established in the scientific community, having been successfully applied to an incredibly broad range of applications including optical flow (Kuchnio and Capson, 2009; Pauwels and Van Hulle, 2008), image processing (Yang *et al.*, 2008), video encoding (Datla and Gidijala, 2009), motion estimation (Colic *et al.*, 2010; Yu and Medioni, 2008), feature detection (Kinsner *et al.*, 2008), face recognition (Poli *et al.*, 2008), graph cuts (Bhusnurmath and Taylor, 2008), biomedical image analysis (Hartley *et al.*, 2008), object detection (Mussi *et al.*, 2009; Xu *et al.*, 2009), molecular dynamics simulation (Yang *et al.*, 2009), and financial engineering (Gaikwad and Toke, 2010).

CUDA has been a popular platform for object tracking and pose estimation applications, specifically face, body, and hand tracking (Breitenstein *et al.*, 2008; Lehment *et al.*, 2010; Lenz *et al.*, 2008). For example, Schoenemann and Cremers (2010) achieved real-time results with a GPU-accelerated template-matching approach that fits contours to images for object segmentation and tracking. Breitenstein *et al.* (2008) use GPUs to compare range images to prerendered pose hypotheses at 55 fps for robust face pose estimation. Lee *et al.* (2009) demonstrated a speedup up to six times over the CPU in an augmented reality application that involved tracking visually attended objects in virtual environments using both preattentive features and contextual information. Because many Bayesian tracking techniques involve the simulation of multiple hypotheses, and each hypothesis produces independent, localized data, they are ideal candidates for parallel implementation on the GPU; for example, Ferreira *et al.* (2010) describe a single-GPU implementation of a multi-modal perception system that utilizes a Bayesian volumetric map (BVM) for robotic exploration.

Model-based tracking using a particle filter is particularly well-suited to GPU-acceleration, yet there is relatively little literature available describing the integration of the two. Real-time, appearance-based particle filtering approaches to face tracking are described by Lozano and Otsuka (2008) and Liu *et al.* (2009), with the former using a sparse template matching algorithm and the latter integrating several visual cues, such as colour and edges. Both report speedups of at least 10 times over CPU implementations. A hybrid model-based and appearance-based tracker is described Cabido *et al.* (2009) for 2D articulated object pose estimation using the GPU for segmentation, rendering and particle weighting. GPU filtering, projection and weighting are also used by Pezzementi *et al.* (2009) for a divide-and-conquer, machine learning approach to 3D model-based tracking of articulated objects in a surgical setting with both mono and stereo image sequences. Finally, Lenz *et al.* (2008) describe a rigid object tracker that utilizes silhouettes to evaluate particles, achieving a speedup up to 10 times over a CPU implementation in face and hand tracking experiments.

3.3 The GT200 GPU

A fundamental goal of NVIDIA CUDA is to provide a level of abstraction between a software developer and the GPU hardware to eliminate the historical prerequisite for an expertise in GPU architecture when writing applications for the GPU. Nonetheless, at this stage in the GPU's evolution, an elementary understanding of its architecture still proves valuable. For example, the justification for several of the optimization concerns discussed in Section 3.4.4 is rooted in the GPU architecture, particularly in the domain of thread scheduling. Therefore, this section presents an overview of the hardware and functionality of NVIDIA GT200-series GPUs, the architecture on which this research was developed and tested.

3.3.1 Hardware Architecture

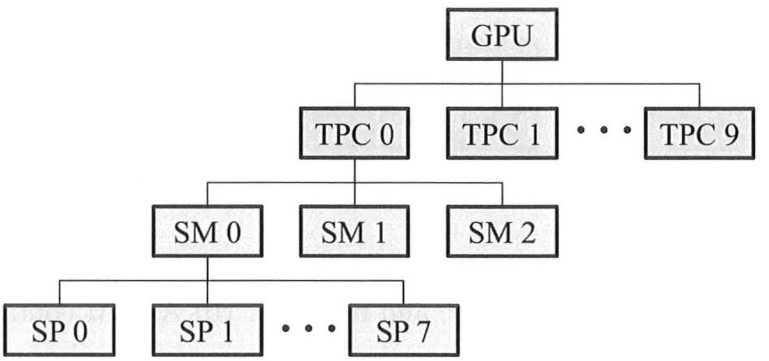

Figure 3.2: GT200 processing element hierarchy
A GT200 GPU has 10 TPCs, each containing three SMs, that contain eight SPs

Released in 2008, the GT200-series built on NVIDIA's previous G80/G92 architecture in several ways; for example, more streaming multiprocessors (SMs), additional texture cache memory, twice as many registers, increased memory bandwidth, support for the peripheral component interconnect express (PCI-E) 2.0 bus, and limited double precision floating point calculation capabilities. In a sense, the GT200 serves a precursor for NVIDIA's most recent GF100 architecture (NVIDIA Corporation, 2009b), also known as Fermi, which primarily targeted the GPU computing community by adding full double-precision floating point support, cache coherency, more cores per SM, and many other features. The trend is clearly toward more programmability and generality, rather than focusing exclusively on the addition of computational resources with each new generation.

The GT200 is structured as a hierarchy (Fig. 3.2) (Kanter, 2008). At the lowest level is the *streaming processor* (SP) (Fig. 3.3), a processing element that contains

two arithmetic logic units (ALUs), an instruction pointer, and a 2 Kbyte portion of a 32-bit-wide register file, but no logic for fetching or scheduling instructions. It therefore lacks the functionality of a true processor *core*, although it is often referred to as one.

At the second level of the hierarchy, the SM (Fig. 3.3) resembles a traditional multiprocessor, such as a multicore CPU. It is able to fetch and schedule instructions with dedicated scheduling hardware and execute them in parallel on an array of eight SPs, which can be thought of as an 8-element vector processor. An SM also has two special function units (SFUs) for advanced operations, such as sine, cosine, and square root, an 8 Kbyte read-only cache for constant memory and 16 Kbyte of 32-bit-wide *shared memory*. Shared memory is unique in that it is the only on-chip, read/write memory that is accessible to multiple SPs, providing a low-latency communication channel between them.

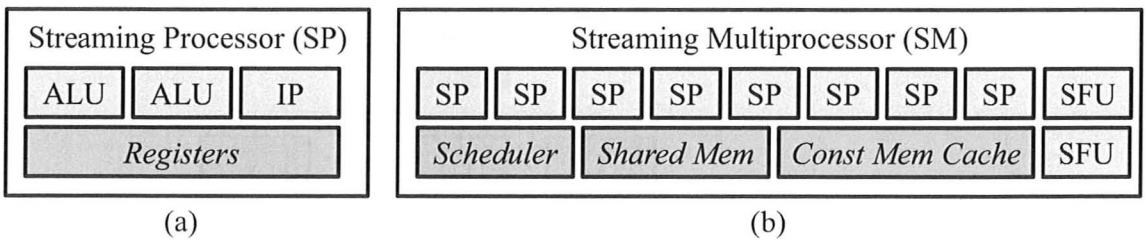

Figure 3.3: Streaming processor and streaming multiprocessor
(a) An SP contains two ALUs, an IP, and registers. (b) An SM contains eight SPs

SMs are organized in groups of three called *texture processing clusters* (TPCs) at the third level of the hierarchy. Each TPC contains a 24 Kbyte read-only L1 texture cache in addition to some control logic (Fig. 3.4). The division of SMs into TPCs is not relevant for CUDA programming, but is an important factor in how the GPU partitions graphics rendering tasks.

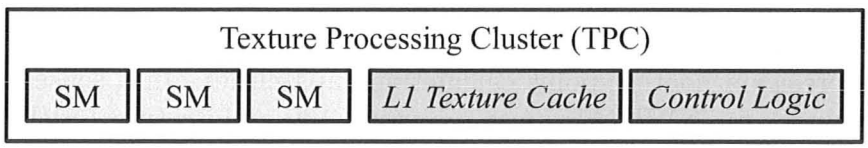

Figure 3.4: Texture processing cluster
A TPC contains three SMs

At the top of the hierarchy (Fig. 3.5), a GT200-series GPU consists of up to ten TPCs, a 256-Kbyte read-only L2 texture cache, a PCI-E 2.0 x16 bus to communicate with the CPU with a theoretical bandwidth of 8 Gbyte/s in each direction, another

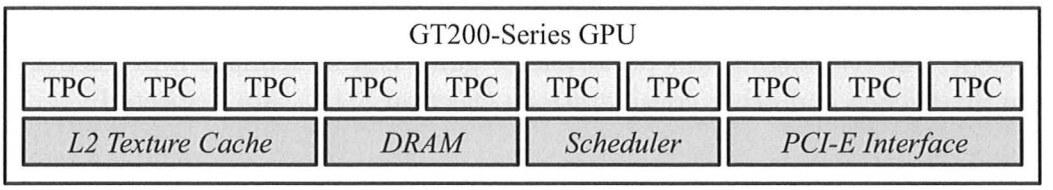

Figure 3.5: GT200-series GPU
A GT200-series GPU contains up to 10 TPCs

level of scheduling logic, and up to 2 Gbyte of GDDR3 DRAM device memory. DRAM contains a variety of GPU memory spaces, including constant, global, local, and texture memory, which are further discussed in Section 3.4.3. To understand the scale and capability of the GT200 series, consider that high-end consumer-grade GPUs built on this architecture are comprised of approximately 1.4 billion transistors, manufactured using a 65 or 55 nm process, are designed to consume up to 234 watts, have a memory bandwidth up to 159 Gbyte/s, and are theoretically capable of executing over one trillion parallel 32-bit FLOPS on 240 SPs.

3.3.2 Scheduling Paradigm

This degree of parallel computational power can only be exploited by highly parallel tasks; in fact, thousands of concurrent threads are recommended to maximize hardware utilization in most applications. Unlike multicore CPU threads, a GPU thread is extremely lightweight and can be created, scheduled, and launched almost instantly. Threads are dynamically partitioned by an SM's scheduler into groups of 32 called *warps*, a paradigm referred to as *single-instruction multiple-thread* (SIMT), a variation on the single-instruction multiple-data (SIMD) class of parallel computers.

Both SIMD and SIMT utilize parallel processing elements to simultaneously execute a common instruction on a vector of data; however, SIMD requires all threads to take the same execution path, whereas SIMT relaxes this constraint by allowing threads within a warp to diverge. *Warp divergence* forces all threads in a warp to execute every instruction branch taken by any thread in the warp, but unnecessary branches are suppressed. As a result, computation time increases for highly-divergent code and is minimized when all threads in a warp agree on a common path. Additionally, SIMD threads have access to all vector elements whereas each SIMT thread maintains its own registers and relies on shared memory (Section 3.4.3) to share data.

Depending on resource utilization, up to 32 warps (1,024 threads) can be active on each SM, totalling 30,720 threads on a GPU. With so many active threads, a new warp can be swapped in by the SM's scheduler while another is waiting on a high-latency instruction, such as a memory transfer. Because warp packaging and scheduling is

handled by low-level hardware, the programmer could choose to completely ignore the concept; however, significant performance gains can be realized if problems, such as warp divergence and other considerations (Section 3.4.4), are respected.

3.4 NVIDIA CUDA

The following section presents the fundamental concepts of NVIDIA CUDA, but is by no means a comprehensive guide. For a more detailed look, the reader is directed to the CUDA programming guide (NVIDIA Corporation, 2010b) or one of the recently released textbooks on the topic (Kirk and mei W. Hwu, 2010; Sanders and Kandrot, 2010). It is important to note that all material in this section refers exclusively to the GT200 architecture and may not apply to G80 or GF100 GPUs.

3.4.1 Overview

CUDA can be described in terms of the architecture it offers to developers to aide in the creation of GPU-based applications, or in terms of its ability to exploit the GPU in a variety of ways. Here, both perspectives are presented.

CUDA Architecture

At its core, CUDA is a driver that exposes the GPU as a parallel processor to application developers through a number of APIs, including OpenCL, DirectCompute, the low-level CUDA driver API, and the higher-level CUDA runtime API (NVIDIA Corporation, 2009a), which was used in this work and is the focus of the remainder of the discussion. The runtime API allows standard C functions to execute on the GPU through a simple set of extensions to the C language called *C for CUDA* (Fig. 3.6). These extensions can be integrated directly into standard C programs and compiled with the included NVIDIA C compiler (NVCC), using standard development environments, such as Microsoft Visual Studio. This simple, low-overhead approach has been a major factor in CUDA's adoption by developers.

In addition to the driver, API, and compiler, CUDA comes with a variety of useful tools and libraries to facilitate GPU software development. For example:

- A **debugging environment** for simultaneous CPU and GPU debugging.
- **Libraries** for fast-Fourier transforms (FFT) and basic linear algebra subprograms (BLAS) on the GPU
- An **occupancy calculator** and **visual profiler** for program optimization
- An extensive **SDK** with a variety of **code samples** exemplifying a broad spectrum of GPU computing applications

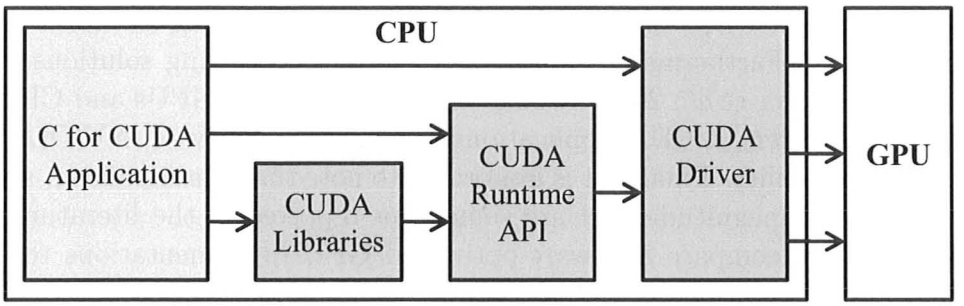

Figure 3.6: CUDA software stack
A CUDA application runs on the GPU driver, the runtime API, and/or libraries

These factors, combined with a massive installation base of over 100 million CUDA-capable GPUs already deployed around the world, has made CUDA the prominent solution for mass-market, massively parallel processing.

CUDA Capabilities

CUDA programs execute serial code on a *host* (CPU) and can asynchronously launch functions called *kernels* to be executed in parallel as threads on a *device* (GPU). This allows the GPU to serve as a coprocessor for data-intensive, parallelizable code (NVIDIA Corporation, 2010a). Scalability is a key tenet of CUDA, and is achieved primarily through a hierarchical execution model with two levels of barrier synchronization (Section 3.4.2) and high-speed shared memory (Section 3.4.3) for inter-thread communication. Beyond these key features, CUDA exposes a variety of additional GPU capabilities that can be extremely valuable in certain applications:

- Access to other **GPU memory spaces**, (global, texture, constant, etc.)
- **Concurrent execution** of host and device code
- **Graphics interoperability** with a variety of Direct3D and OpenGL resources
- Support **for multiple GPUs**
- Single or double-precision **floating point calculations**
- High-speed, lower-precision **math libraries** (sine, exponent, etc)
- Access to **GPU timers** for accurate benchmarking

There are, however, limitations to CUDA's utility. As discussed in Section 3.3.2, the GPU is best-suited for computation with minimal branching (i.e., conditional and looping structures), making it a poor choice for many user-driven programs, such as word processors. Additionally, parallel computation on the GPU is optimal when dealing with array-like data structures with a static size, and localized data. Ideally, memory accesses should always be consistent and coalesced, not scattered. Recursion

is not possible with CUDA, meaning many data structures, such as linked lists, are not viable options. Furthermore, unlike other parallel processing solutions, such as OpenCL (Tsuchiyama *et al.*, 2010), which targets a variety of GPUs and CPUs from different manufacturers, CUDA applications can execute solely on NVIDIA GPUs, limiting their portability. Finally, it is important to note that the significant speedups of several orders of magnitude that are sometimes reported in the literature can be misleading, as they compare hardware-optimized GPU implementations to unoptimized CPU implementations Lee *et al.* (2010).

Floating point precision has been a major limiting factor for many HPC tasks that could benefit from GPU-acceleration but demand extreme accuracy in large-scale calculations. Prior to the GT200 architecture, double-precision arithmetic was unavailable on GPUs, and, although The GT200 architecture supports double-precision operations, it is not fully IEEE-compliant and performance is comparable to modern CPUs. The GF100 architecture introduced fast, IEEE-compliant double precision arithmetic on professional-grade GPUs, but consumer-grade performance is scaled back. Nonetheless, many applications are unaffected by this limitation and can realize significant performance increases with virtually no decrease in functionality.

3.4.2 Execution Model

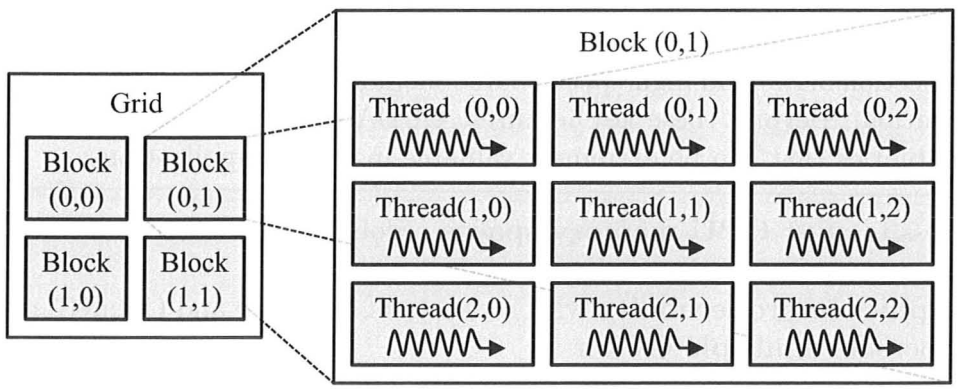

Figure 3.7: GT200 thread hierarchy
The hierarchy consists of a grid of blocks containing threads.

The CUDA execution model is a hierarchical, scalable paradigm able to dynamically partition and execute appropriate algorithms across thousands of GPU threads with great efficiency. The following describes the nature of the thread hierarchy, how it scales to maximally utilize available hardware, and how threads are launched and coordinated.

Thread Hierarchy

The kernel code is executed by fine-grained parallel threads, which are grouped into 1D, 2D, or 3D *blocks* of dimension (B_x, B_y, B_z). Blocks are further arranged in a 1D or 2D *grid* of dimension (G_x, G_y) providing second, coarser level of parallelism. The thread hierarchy is illustrated in Fig. 3.7. Grid and block dimensions can be determined at runtime or compile time according to the application requirements. On GT200-series GPUs, there can be no more than 512 threads in a block; however, there can be up to 65,535 blocks in a grid. Additionally, the total number of threads in a block should always be a multiple of 32 on GT200 GPUs to ensure even partitioning into warps. A unique identifier for each thread can be calculated within a kernel according to

$$tid = b_i + b_j B_x + b_k B_x B_y \tag{3.1}$$

where (b_i, b_j, b_k) are the x, y, and z thread indices, respectively. A similar approach can be taken to derive a unique block identifier within the grid. This identifier is often used as an index for a data structure, ensuring each thread accesses a unique piece of data.

Blocks and grids exhibit different levels of parallelism. When a kernel is launched, individual blocks within the grid are assigned to a single SM by the GPU's scheduling hardware. Although multiple blocks can be active on an SM at once (depending on number of threads in the block and resource utilization), there is no guarantee that there will be enough SMs available for all blocks. Therefore, while blocks *may* execute concurrently, some blocks will often be serialized, with larger grid sizes exhibiting more serialization. The disadvantage is that no inter-block coordination, synchronization, or data sharing is possible since there is no guarantee of concurrency. If an application requires global synchronization of all blocks in a grid, a new kernel must be launched. The benefit of having multiple blocks per SM is that the scheduling hardware is able to swap out a block that is waiting on a high-latency instruction and replace it with a block that contains threads ready to execute.

Threads within a block, however, are guaranteed to reside on a single SM simultaneously. Threads are packed into warps, as described in Section 3.3.2, and executed concurrently on an SM's array of streaming processors. While there is no predictability to the exact order that threads within a block will be scheduled, the guarantee of concurrency facilitates intra-block *barrier synchronization* and communication through shared memory (Section 3.4.3). Furthermore, the second level of scheduling provides increased efficiency through fine-grained warp swapping.

When determining block and grid dimensions, many factors must be considered, most of which are discussed in Section 3.4.4. In general, a thread block should be considered a fixed-size unit of work to be executed by a cooperative set of threads, whereas a grid's size should be dynamic and reflect the size of the overall data set,

divided into completely independent units of work. At this point, it may seem beneficial to always utilize the largest block dimensions possible; however, this is rarely the case, as described in Section 3.4.3 and Section 3.4.4.

Scalable Computation

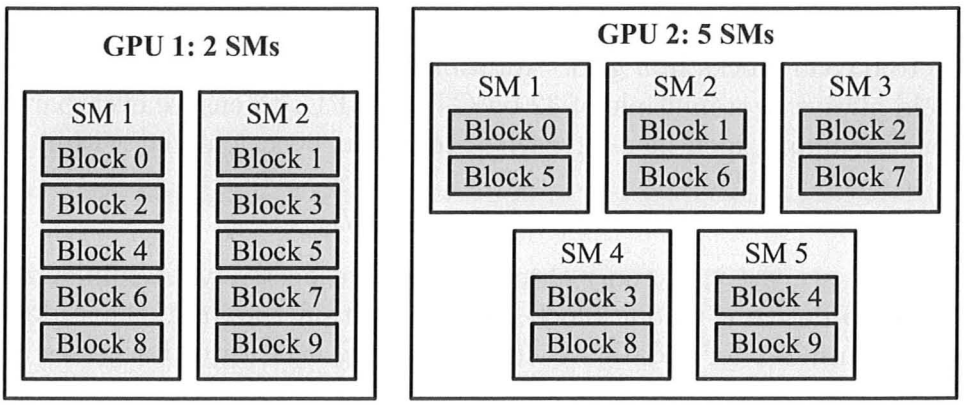

Figure 3.8: CUDA automatic scalability
CUDA blocks will automatically be scheduled to efficiently use all available SMs

A benefit of the thread hierarchy is automatic scalability; as long as a task can be partitioned into computationally independent sub-problems that do not need to communicate with each other, that task can be efficiently scheduled across all SMs of any capable NVIDIA GPU. This means a well-written CUDA application is able to scale to maximally utilize a suitable GPU, whether that GPU is, for example, the GeForce 210 with two SMs (16 SPs) or the GeForce GTX 285, which has 30 SMs (240 SPs). This is very appealing to programmers who wish to target as broad an audience as possible without having to generate multiple versions of an application. Fig. 3.8 demonstrates how a grid of 10 blocks would scale to utilize GPUs with two or five SMs.

Kernels and Memory Transfers

Although the device (GPU) and host (CPU) code may share the same C file, their execution and memory spaces are completely decoupled, requiring explicit memory copies to transfer data across the PCI-E bus for data exchange. A traditional CUDA application (Fig. 3.9) is roughly structured as follows:

1. Allocate memory on the host and device
2. Prepare data in the host memory space

3. Transfer data from the host to device with a synchronous memory copy function
4. Asynchronously launch a kernel on the GPU
5. Continue execution of serial code on the CPU concurrently with the GPU kernel
6. Transfer results from device to the host with a synchronous memory copy. This causes CPU execution to be blocked until the kernel has finished executing
7. Continue execution of serial code on the CPU

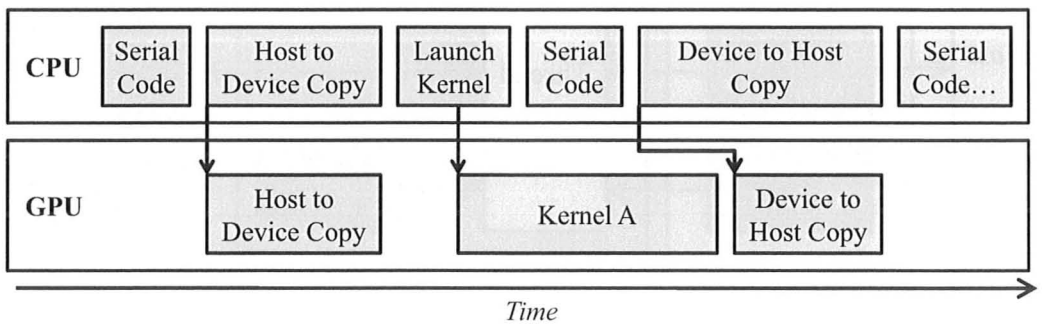

Figure 3.9: Blocking and non-blocking GPU function calls
Kernels can be launched asynchronously, but memory transfers cannot

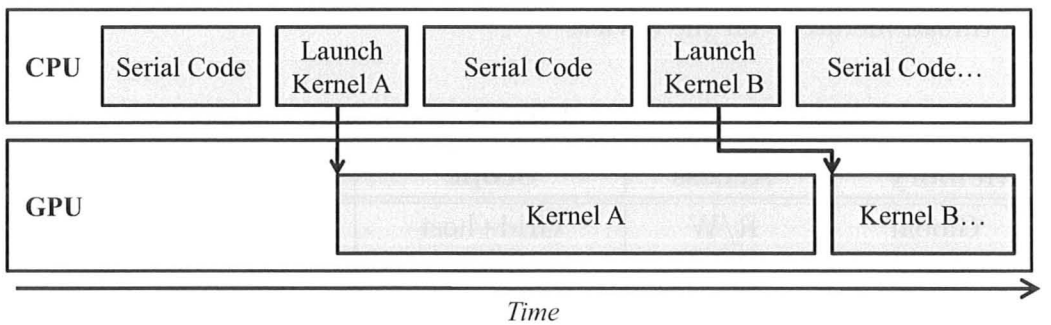

Figure 3.10: Asynchronous kernel launching
The GPU and CPU can execute code concurrently, but only one kernel can be active on the GPU

It is important to note is that concurrent execution of GPU and CPU code is possible, further exemplifying the GPU as a coprocessor. While some programs may deviate from this structure, the concept is almost always the same. A limitation of the GT200 architecture that has been eliminated in the GF100-series is that only one kernel can be active of a GPU at any given time. Attempts to launch multiple kernels will simply serialize the kernels' execution, as demonstrated in Fig. 3.10.

3.4.3 Memory Model

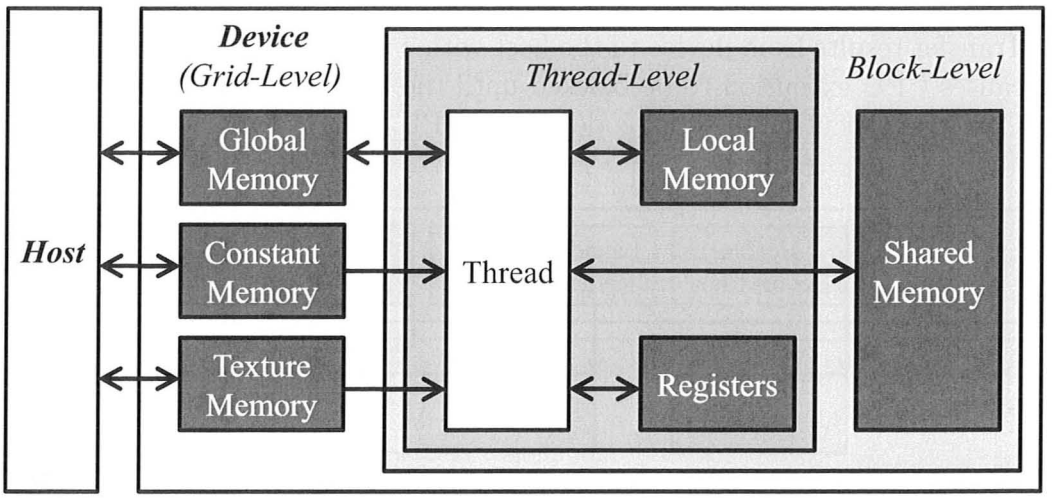

Figure 3.11: CUDA memory model

The GPU has several unique memory spaces that are exposed by CUDA for use in a variety of scenarios. Each is described below and summarized in Table 3.1. Additionally, Fig. 3.11 shows how the different memory spaces interact with the host and the thread hierarchy on the device.

Table 3.1: Summary of the GT200 memory model

Memory	Access	Scope	Cached	Latency
Global	R/W	Grid+host	No	High
Shared	R/W	Block	No	Low[1]
Register	R/W	Thread	No	Low
Local	R/W	Thread	No	High
Constant	R	Grid+host	Yes	Low[2]
Texture	R	Grid+host	Yes	Low[2]

[1]Assuming no bank conflicts
[2]Cached reads only

Global memory

As the largest memory space available on the GPU, global memory acts as the primary data transfer channel between the host and device in most applications; however, it is

also the highest-latency memory (400 to 800 clock cycles) because it is found off-chip (i.e., separate from the SMs in DRAM). The host and all threads in the grid can read and write anywhere in global memory, but it is not cached and should therefore be accessed sparingly to avoid the latencies. The manner in which consecutive kernel threads (i.e., threads in a warp) access global memory can greatly impact throughput. Specifically, by properly aligning thread indices with segments in global memory, multiple memory reads and writes can be grouped into a single transaction (Fig. 3.12). Because global memory is not used extensively in this work, the details of the access patterns will not be discussed here, but can be found in the *CUDA Best-Practices Guide* (NVIDIA Corporation, 2010a). An in depth analysis of memory access patterns can also be found in (Ha *et al.*, 2010). Finally, note that global memory persists across kernel launches, meaning data will have the lifetime of the application, unless overwritten or freed.

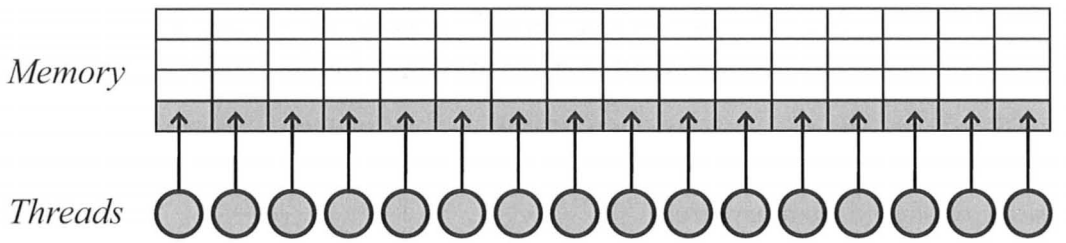

Figure 3.12: An effective global memory access pattern
Accesses are coalesced and aligned with memory segments

Page Locked Memory A small amount of *page-locked* or *pinned* memory can be allocated in the host memory space to provide higher-bandwidth memory transfers. Additionally, a block of page-locked host memory can be mapped into the device's global memory space to facilitate memory transfers that are concurrent with kernel execution. This increases bandwidth further and eliminates the need to explicitly allocate and copy data in device memory space, but requires manual synchronization to avoid potential data hazards.

Shared Memory

Shared memory is a unique and powerful resource in CUDA programming that can be thought of as a user-managed cache. Because it is located on-chip, it has very low latency (approximately 100 times lower than global memory). Shared memory is declared at the start of a kernel by specifying the amount required for a block (at runtime or compile-time). This is an important parameter, as there is a finite amount of shared memory available in each SM that must be shared amongst all active blocks.

Therefore, excessive shared memory usage can limit the number of active blocks on an SM and potentially degrade performance. The coalescing requirements of global memory do not apply to shared memory; however, if multiple threads of the same half-warp access the same 32-bit shared memory bank simultaneously, a *bank conflict* will occur and accesses will be serialized. In other words, shared memory utilization is optimal when 16 consecutive threads access different memory banks (Fig. 3.13).

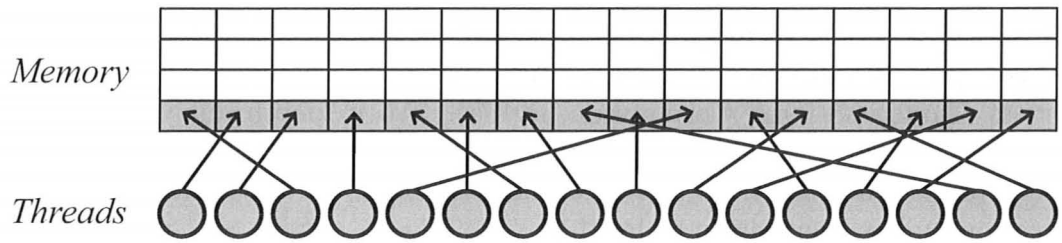

Figure 3.13: An effective shared memory access pattern
Each thread accesses a unique bank

Accessible to all threads within a block, but never the host, shared memory does not persist across kernel calls and has the lifetime of its thread block. The two main functions of shared memory are:

- to provide an intermediate storage area to avoid unnecessary global memory usage; and,
- to facilitate low-latency inter-thread data sharing.

Consider an application that involves the convolution of a 3×3 filter with image data, such as an edge detector. Assume that the task has been partitioned such that each thread applies the filter to a single pixel in the image. This would require each thread to load nine pixels from global memory to fully cover the image (ignoring issues of edge padding). Additionally, the data loaded by each thread is not mutually exclusive; in fact, each pixel would be accessed nine times. This scenario is an ideal candidate for shared memory. Consider the adoption of the following strategy for each thread block:

1. Each thread loads a single pixel from global memory into shared memory
2. Synchronization ensures all threads in the block have completed their loads
3. Each thread applies the filter to a single pixel in shared memory, reading the eight other required pixels from shared memory as well
4. Each thread writes a single result pixel back to global memory

Using this approach, each pixel is read from global memory just once, and the total number of global memory reads is reduced by a factor of nine, while accomplishing the

same operation. This illustrates the utility of shared memory, both as an intermediate storage area and an inter-thread communication channel.

There is an important trade-off that must be considered when partitioning (or *tiling*) a task: making the tiles too big can consume too much shared memory and reduce the number of active blocks on each SM, but making tiles too small generates a great deal of overhead and fails to fully exploit the GPU resources. These considerations are discussed further in Section 3.4.4.

Registers

Registers are a fast, low-latency (typically zero clock cycles), on-chip resource but, unlike shared memory, are only visible to their resident threads and have a lifetime limited to that of the thread. Most variables declared within a kernel are stored in registers, making them a common intermediate storage area. Similar to shared memory, registers are a limited commodity on an SM and are shared between all active blocks. Consequently, they can also affect the number of blocks simultaneously active on an SM.

Local Memory

Local memory behaves similarly to registers, with the same lifetime and visibility, but resides in an area of global memory and is therefore subject to extremely high latencies. Local memory can generally be considered an *overflow* memory space that is only utilized if there are insufficient registers available for all active warps. The exception is an array declaration inside a kernel, which automatically places data in local memory instead of registers.

Constant Memory

Constant Memory also resides in global memory but is cached on-chip to reduce latency when multiple threads in a block access the same location (Fig. 3.14). Constant memory has the lifetime of the application and is ideal for storing constants, such as filter parameters, or values that would normally be passed to a kernel as a function parameter. Normally, constant memory is written to once by the host during application initialization and accessed by kernels throughout execution; however, because it is such a small space, it may be necessary for the host to periodically overwrite data in constant memory for kernels to fully utilize its low-latency cache.

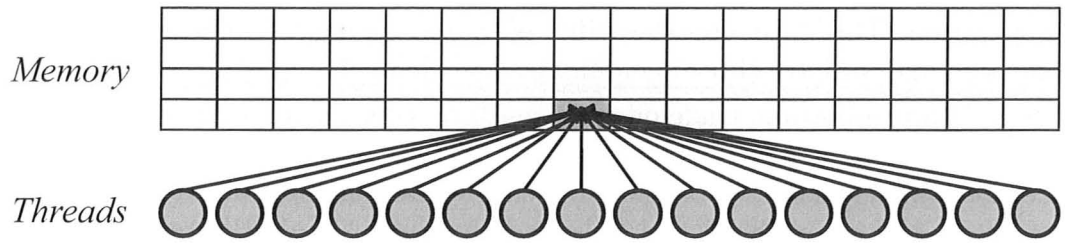

Figure 3.14: An effective constant memory access pattern
All threads access the same value simultaneously

Texture Memory

Texture Memory is another cached, read-only space that resides in global memory; however, unlike constant memory, it can span all of global memory (with some limitations) and has a less-restrictive access pattern. Texture memory requires an area of device memory (linear region or region allocated as a CUDA array) to be bound to a *texture reference* and can be accessed from kernels through *texture fetches*. Texture fetches are optimized for 2D spatial locality within the memory space (Fig. 3.15), providing a powerful alternative to global memory access patterns in some applications. Additionally, texture fetches use dedicated hardware for addressing calculations, removing the burden from kernels. Textures also provide a variety of addressing, filtering and clamping options that can be useful in specific applications (usually involving image processing). Finally, texture memory provides the easiest way to access data within graphics resources, such as Direct3D surfaces when using CUDA's Direct3D interoperability (Section 3.5.5).

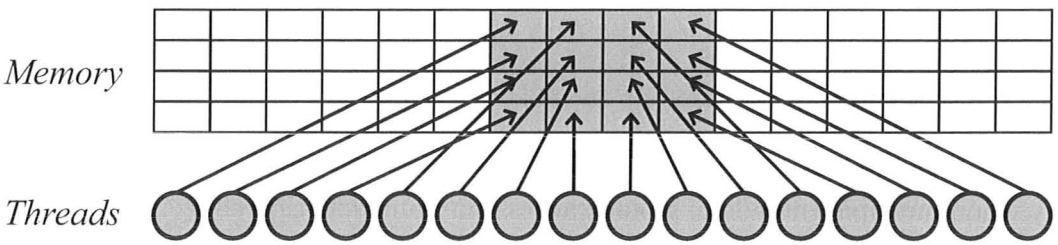

Figure 3.15: An effective texture memory access pattern
Threads access data with spatial locality

3.4.4 Optimization Concerns

It can be quite straightforward to directly port CPU code to the GPU; however, achieving a tangible speedup requires the programmer to adhere to a variety of often-conflicting optimization strategies and iteratively tune their application to the optimal configuration. Some prominent issues that should be considered are discussed below.

Multiprocessor Occupancy

The ratio of the number of active warps on an SM to the SM's total warp capacity is termed *multiprocessor occupancy*, and should be kept above 50 percent to ensure efficient warp-level scheduling. Recalling that each SM on a GT200 GPU can support a total of 32 warps, this translates to a recommended 16 active warps. Occupancy is affected by three factors:

- number of registers used by each thread;
- amount of shared memory used by each block; and,
- number of threads per block.

Register and shared memory usage affect multiprocessor occupancy in similar ways. Recall from Section 3.3 that each SM has just 16 Kbyte of shared memory and a 16 Kbyte register file, and these resources are shared by all active warps on an SM. Therefore, if each of the desired 16 warps consume more than one sixteenth of an available resource, the multiprocessor occupancy will be lowered to compensate.

For example, consider a kernel that uses a block size of 128 threads and requires each thread to utilize 32 registers. Dividing the 16 Kbyte register file (registers per SM) by 32 (register per thread) yields 512 threads per SM, which is equivalent to 16 warps and achieves the desired 50 percent multiprocessor occupancy. If, however, each thread required 33 registers, the same calculation would yield approximately 496 threads per SM. This number is then reduced to the lowest multiple of the block size, which is 384 (128×3) in this case. The addition of a single register in the kernel code therefore reduced the number of active warps on each SM from 16 to 12, lowering the multiprocessor occupancy from 50 percent to 38 percent. This would likely cause performance degradation in the application, as there may not be enough warps available for swapping to satisfactorily hide access latencies.

A similar argument can be made for shared memory, which is allocated at the block level instead of the thread level. Once again considering a block size of 128, assume a kernel requires each block to utilize 4 Kbyte of shared memory. With a total of 16 Kbytes available on each SM, this means there is enough available for four blocks to be active at once. Four blocks is equivalent to 512 threads or 16 warps, and adheres to the 50 percent occupancy guideline. If, however, a single additional byte

of shared memory was required by each block, there would only be enough resources available for three at once and the occupancy would again decrease to 38 percent.

The occupancy guideline can also be used gauge how many threads an application would require to fully utilize the GPU hardware. If each SM is to maintain 512 active threads (16 warps) and there are 30 SMs on a GT200 GPU, this implies that a grid should contain a minimum of 15,360 threads to maximally utilize all available resources. This gives a sense of the scale of the massively parallel computation afforded by the GPU.

The preceding discussion has focused on warp-level scheduling, but not block-level scheduling, which must also be considered. Ideally, an SM should have at least two thread blocks active for efficient scheduling, allowing a new block to be swapped in if another is waiting on a barrier synchronization or other high-latency operation. Consider once again a block size of 128 with 32 registers per thread. As discussed above, this would lead to occupancy of 50 percent, with four active thread blocks. If, however, a block size of 512 was used instead, the same occupancy would be achieved with only a single active block on each SM. While this would not impact fine-grained warp-level scheduling, coarse-grained block-level scheduling would be affected.

The common theme of these examples is that the manner in which a task is partitioned can significantly affect performance. If blocks are too large, there may not be enough to keep all available SMs busy and they may consume too many registers or too much shared memory; however, if they are too small, blocks will be unnecessarily serialized and there is less opportunity for inter-thread cooperation. This is one of the many trade-offs that must be considered when writing effective code for the GPU. NVIDIA provides an *occupancy calculator* as part of the CUDA toolkit to aide developers in assessing occupancy, but, in all but the most straightforward applications, iterative experimentation is the only way to identify ideal operating parameters.

There are additional considerations that can influence how a task is partitioned as well. As mentioned above, block sizes should always a multiple of 32 as this ensures even partitioning into warps and also makes global and shared memory access patterns easier to adhere too. It is also recommended that the block size be at least 192 to fully hide register latencies.

Memory Bandwidth

Maximizing memory bandwidth is critical in the data-centric applications that are most often implemented on the GPU. As a general guideline, access to global memory from within a kernel should be avoided whenever possible, through creative usage of texture, constant, and shared memory spaces. Shared memory is perhaps most valuable in this endeavour, as it can reduce the total number of accesses required

through data sharing and also provides a low-latency intermediate data storage space. Although they will not be reiterated here, adherence to the memory access patterns discussed in Section 3.4.3 is paramount in accomplishing this and can affect bandwidth by an order of magnitude.

Bandwidth between DRAM and the device is up to 159 Gbyte/s on GT200 GPUs (with significant access latencies), but the bandwidth of transfers between the host and device is limited by the PCI-E 2.0 bus at 8 Gbyte/s. Therefore, it is critical that transfers be minimized and packaged into a single batch transfer whenever possible. Additional bandwidth can also be realized using page-locked, mapped memory. In many situations, it can actually be more efficient to recalculate a piece of data and avoid a memory transfer altogether. It should be noted that memory-intensive applications can still perform very well on the GPU, as long as there is sufficient computation within a kernel to hide the latencies.

Instruction Throughput and Precision

The GT200 architecture does support double-precision floating point calculations; however, their performance is a fraction of single-precision performance and they should be avoided unless absolutely required by an application. Additionally, CUDA comes with a math library that contains a variety of intrinsic single- and double-precision functions (e.g., sin, exp, log2, log10), that offer higher throughput while sacrificing some precision. Finally, operations such as integer division and modulo have a very high latency on the GPU and should also be avoided if possible.

Warp divergence, discussed in Section 3.3, also affects instruction throughput as it serializes the execution of branches in code when threads in a warp do not agree on a common path. In many cases a task can be partitioned in a different way that avoids warp divergence with minimal effort. Barrier synchronizations should also be used sparingly as they introduce an additional constraint to an SM's scheduler that can cause warps to idle while waiting for others to finish executing. The ratio of arithmetic instructions to memory instructions in a kernel is known as *arithmetic intensity* and should be kept as high as possible to maximize throughput. In some cases, when arithmetic intensity is low, manually prefetching data in a kernel can hide the memory access latencies.

Other considerations

It should be clear that there are many constraints and trade-offs to consider when optimizing a CUDA application. This can be seen explicitly when attempting to identify the ideal *thread granularity* for a task; in other words, when deciding whether it is beneficial to put more work into each thread and use less threads. For example, it may be tempting to place a loop inside a kernel and have each thread service multiple

pieces of data, but this introduces warp divergence and lowers the number of blocks in a grid. If each iteration of the loop is arithmetically intense and there are still enough blocks to occupy all SMs, this strategy could realize a significant performance gain; however, other scenarios may see degradation in performance. It is very difficult to assess which optimization goals should take precedence without experimentation.

The most important optimization goal when porting an application to the GPU is the hardest to quantify: the task must be parallelized as much as possible. Increasing bandwidth or arithmetic intensity will prove futile if the task cannot be framed in a highly-parallel manner and mapped to the GPU appropriately. This is an area that requires both logic and creativity from the programmer, and should take precedence over all else.

3.5 Microsoft Direct3D

Direct3D, an API available as part of Microsoft's DirectX SDK, facilitates optimized 3D graphics rendering on the GPU and is used primarily in video game development. There have been several iterations of Direct3D since its inception in 1995, leading to Direct3D 11, which was recently released with Windows 7. This work uses Direct3D 9.0c, which was selected for its interoperability with CUDA, ease of use, and support for a variety of Microsoft operating systems. The remainder of this section focuses exclusively on Direct3D 9.

Direct3D provides an interface to the GPU driver for rendering realistic 3D scenes using the GPU's graphics pipeline (3.1), by exposing vertex buffers, surfaces, lights, textures, viewports, meshes, and numerous other resources and effects. A very small subset of Direct3D's capabilities are exploited in this work to efficiently render 3D models with realistic lighting, shading, and skinning. For a detailed look at programming with Direct3D 9, the reader is directed to the programming guide (Microsoft Corporation, 2003) or one of the many textbooks (Sherrod, 2006; Luna, 2003) or online tutorials on the topic. Here, the small subset of its features and terminology that are critical to understanding the proceeding chapter are introduced.

3.5.1 Resources and Rendering

Rendering an object with Direct3D involves transforming object geometry data into pixel data using the GPU's graphics pipeline (Fig. 3.1). The object's geometry is represented as a list of 3D coordinates (with respect to a local origin) called *vertices* that are stored in a *vertex buffer* in device memory. Vertex data is processed by vertex shaders to apply lighting and shading effects, then assembled into primitives (triangles, lines, or points) according to a list of indices in an index buffer that define the object's geometry. The processed vertices then enter a geometry pipeline

(Section 3.5.2) and are *rasterized*. In other words, a series of transforms are applied to manipulate the object's position and orientation in virtual space then create a 2D projection of the object according to the location and projective properties of a virtual camera. The resulting image is processed by pixel shaders (or fragment shaders) to apply colours, textures and other effects.

The result is an array of pixel data in device memory in the form of a Direct3D *surface*, which can be swapped into the video card's *back buffer* for display on a screen. Surfaces are a useful Direct3D resource that can be customized with a desired size (resolution), pixel format (32-bit ARGB in this work), and memory space (host or device). They also have a number of useful features, such as the ability to be locked by the host to access and modify individual pixels and efficient resizing and stretching options. Textures have similar features to surfaces, but can be applied to geometry during fragment shading and cannot reside in host memory space. Rendering an image to a texture then applying it to a small *quad* (square made of two triangle primitives), while using a large surface as a render target, is an efficient way of obtaining a modular 2D projection of an object on a large surface.

3.5.2 The Geometry Pipeline

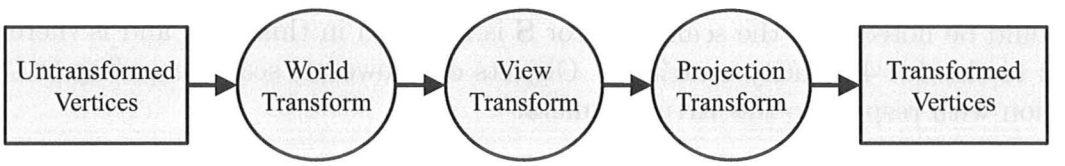

Figure 3.16: Geometry pipeline

The geometry pipeline (Fig. 3.16) effectively determines how a set of vertices that define an object's geometry are oriented and projected onto a 2D surface. In other words, it provides a mapping between a model's local 3D coordinate system to the surface's 2D coordinate system. This is accomplished through the application of three transforms: the *world transform*, the *view transform*, and the *projection transform*. Each transform can be expressed as one or more 4 $\times$4 matrices and applied to a vertex (expressed in homogeneous coordinates) by multiplying that vertex with each of the matrices.

In general, transforms can be decomposed into translation (t_x, t_y, t_z), rotation (r_x, r_y, r_z), and scaling (s_x, s_y, s_z) factors. Five geometric transformation matrices are used to express these factors: a translation matrix $\mathbf{T}$, a scaling matrix $\mathbf{S}$, and x-,

y-, and z-rotation matrices, $\mathbf{R}_x$, $\mathbf{R}_y$, and $\mathbf{R}_z$, respectively:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(r_x) & sin(r_x) & 0 \\ 0 & -sin(r_x) & cos(r_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y = \begin{bmatrix} cos(r_y) & 0 & -sin(r_y) & 0 \\ 0 & 1 & 0 & 0 \\ sin(r_y) & 0 & cos(r_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_z = \begin{bmatrix} cos(r_z) & sin(r_z) & 0 & 0 \\ -sin(r_z) & cos(r_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

World Transform Untransformed vertices are described with respect to the origin of an object's local coordinate system. The world transform moves vertices from model coordinates (or model space) to world coordinates (world space), effectively placing and orienting the object at a specific point within the virtual world. A vertex in model coordinates $\mathbf{p}_m = (x_m, y_m, z_m)$ can be transformed into world coordinates $\mathbf{p}_w = (x_w, y_w, z_w)$ according to

$$\mathbf{p}_w = \mathbf{p}_m \cdot \mathbf{R}_x \cdot \mathbf{R}_y \cdot \mathbf{R}_z \cdot \mathbf{S} \cdot \mathbf{T} \tag{3.2}$$

It should be noted that the scaling factor $\mathbf{S}$ is not used in this work and is therefore equal to the 4×4 identity matrix $\mathbf{I}^4$. Objects do, however, scale according to their location with respect to the virtual camera.

View Transform The view transform orients the world with respect to a virtual camera, effectively setting a point-of-view. In other words, it transforms vertices from world coordinates to camera coordinates. A vertex in world coordinates $\mathbf{p}_w = (x_w, y_w, z_w)$ can be transformed into camera coordinates $\mathbf{p}_c = (x_c, y_c, z_c)$ according to

$$\mathbf{p}_c = \mathbf{p}_w \cdot \mathbf{T} \cdot \mathbf{R}_x \cdot \mathbf{R}_y \cdot \mathbf{R}_z \tag{3.3}$$

The location of the virtual camera does not change in the proposed framework.

Projection Transform The projection transform scales the world according to distance from the camera to create the illusion of depth. In other words, it transforms vertices from camera coordinates to projection coordinates (according to the specified clipping planes, field of view, and aspect ratio) to simulate the lens of a real camera. The projection transform involves a scaling and perspective transform, meaning it

cannot be expressed using geometric transformation matrices. Since the projective transform remains fixed in this work, it is not critical to understanding the operation of the proposed framework and will not be discussed in detail.

Once in projection space, the visible transformed vertices are converted to screen (or surface) coordinates through rasterization. The overall result mimics the effect of a real camera capturing images.

3.5.3 3D Models

Direct3D models are stored as *X files*, a common format that most 3D modeling packages, such as Autodesk Maya (Autodesk Maya Press, 2009) or Blender (Roosendaal and Selleri, 2009), are able to produce. An X file is imported into Direct3D as a hierarchy of *bones* (or *frames*), arranged as a kinematic chain that stems from a single root frame. Collectively, the frames define the articulation of an object. Each bone has a corresponding geometric transformation matrix called a *frame matrix* that describes the bone's pose (in bone space) relative to its parent bone, in the same way the world transform describes an object's pose relative to the origin in world space (Section 3.5.2). In a sense, each frame can be considered a completely separate object, capable of rotation, translation and scaling; however, when dealing with traditional "skeleton-like" models, movement of child frames is constrained by joints, so their transformation matrix would be used exclusively to describe rotation caused by joint flexion (j_x^k, j_y^k, j_z^k), where k is the index of a joint. As a result, when the root frame is positioned using the world transform, all other frames in the hierarchy will be moved as well. Additionally, a flexion near the top of the hierarchy will propagate down and position all dependent bones and joints accordingly.

The other major component of an X file is a *mesh* that defines the geometry of the model. A mesh is a collection of vertex and index data that Direct3D can import into vertex and index buffers, respectively, to render the model. Each vertex in a mesh is associated with one or more frames that determine the vertex's location in model space. In other words, changes to a bone's frame matrix will affect all vertices associated with that bone. The mesh can therefore be considered "skin" surrounding the bones. When vertices in a mesh are influenced by more than one bone (e.g., skin around a joint), a technique called *skinning* must be applied to determine their final location in model space. Skinning can be computationally demanding, but is required for realistic articulated models. In this work, the GPU's vertex shaders are exploited using HLSL to perform skinning.

An imported model is rendered to the screen as described in Section 3.5.1, with a few additional steps:

1. The frame hierarchy is traversed (starting at the root) and each bone's frame matrix is updated according to its parent bone's frame matrix and the flexion

(j_x^k, j_y^k, j_z^k) of its corresponding joint
2. The frame hierarchy is traversed again and each frame matrix is applied to any associated vertices on the mesh
3. Skinning is performed on any vertices that are influenced by multiple bones

3.5.4 Lights, Materials, and Shading

Direct3D provides a variety of lighting and material options to produce realistic shading and shadow effects on rendered objects. One or more light sources with a configurable colour and type of light (*diffuse*, *ambient* and/or *spectral*) can be added to a scene as ambient, directional, point, or spot lights. In the case of point and spot lights, the light source has a fixed position in world space, whereas ambient and directional lights do not emanate from any specific location. Additionally, directional and spot lights are configured to point in a specific direction. The result realistically imitates light in the real world.

Every vertex has an associated *material* (optionally included in the X file of a model) that defines how light will affect it. In other words, a material determines what colour and type of light a vertex will reflect. For example, a material that reflects a great deal of spectral light would appear shiny while a material that only reflects ambient and diffuse light would appear dull. Additionally, a white surface would reflect all colours of light, whereas a purely red surface would appear black under a green or blue light, and a black surface reflects no light.

The effects of lights and surfaces are computed during vertex shading stage of the graphics pipeline. Adding additional lights to a scene increases rendering time; however, they are critical when an application requires a rendered model to accurately reflect the appearance of an object in the real world.

3.5.5 Direct3D/CUDA Interoperability

A number of Direct3D resources, such as vertex buffers, index buffers, textures and surfaces, can be accessed in CUDA's address space for reading and writing data within kernels. This is done by registering a resource with CUDA when it is first initialized in the device's memory. Once registered, a resource can efficiently be mapped and unmapped to CUDA's memory space at any point, as long the host does not attempt to access the resource while it is mapped. In this work, surfaces are mapped to CUDA's address space, where they are treated as CUDA arrays, bound to texture references, and accessed via texture fetches (Section 3.4.3).

3.6 GPU Computing in 3D Model-Based Tracking

As a prelude to the next chapter, which describes in detail how the GPU is exploited for model-based visual tracking, this section highlights the key features of CUDA and Direct3D that play a significant role in this work.

Rendering a Tracking Target Model The crux of 3D model-based tracking techniques is a realistic representation of the tracking target, which is precisely what the GPU's graphics pipeline is designed to provide. By configuring Direct3D's virtual camera and lights to accurately simulate the tracking environment and camera lens, rasterized projections of the model will closely reflect real images of the tracking target captured by the camera. In the terminology of the particle filter (Section 2.5), this means the particle images will be close approximations of the current frame and a comparison of extracted features will produce meaningful weights. Furthermore, because the performance of Direct3D has been driven by the video gaming market, which often demands frame rates of 60 fps or higher, model rendering will be extremely efficient - critical for real-time tracking.

Interfacing with a Tracking Target's State Vector An advantage of model-based tracking approaches is that the state vector being estimated (position, rotation, and joint flexions) represents the same parameters that control the 3D model of the tracking target. In other words, no mapping is needed between tracking target space and virtual world space. Direct3D's geometry pipeline makes this extremely intuitive, as the state vector used to configure particles can interface directly with a model's world transform matrices and frame matrices.

Accessing Particle Image Pixel Data The particle images produced by Direct3D would be useless if their pixel data could not be accessed efficiently by CUDA for parallel processing. Because Direct3D surfaces are interoperable with CUDA, they can be mapped into CUDA's memory space for feature extraction and weight calculation. The critical advantage is that the particle images are rendered directly to the GPU, meaning there is virtually no overhead from transferring data from the host to device, which is the primary limiting factor in most CUDA applications. Mapping the particle images to CUDA involves determining the address of the particle image data and simply passing that address to a CUDA kernel.

Extracting Features As discussed in the previous chapter, the type of feature extraction used in this work is silhouette and edge detection. Both these tasks are ideal candidates for implementation on CUDA, as they exhibit a great deal of data locality (ideal for shared memory utilization), do not require significant conditional

logic (avoiding warp divergence), and consist primarily of floating point calculations (high arithmetic intensity). Most importantly, this type of feature extraction is highly parallelizable and can be partitioned into modular, independent sub-tasks that can be solved using cooperative threads; optimal conditions for compatibility with CUDA's execution hierarchy. Many other feature extraction techniques exhibit similar characteristics and would benefit greatly from CUDA acceleration.

Computing Weights Computing the weight of a particle from its particle image involves the reduction of thousands of individual pixel weights to a single value. Although reduction operations are intrinsically difficult to parallelize, they can still be accelerated significantly by CUDA. It should also be noted that since weights are relative values, single-precision floating point accuracy can be utilized with negligible consequence.

Chapter 4

GPU-Accelerated Particle Filtering for 3D Model-Based Tracking

This chapter presents a novel framework that uses a GPU-accelerated particle filter for high-speed, rigid or articulated, 3D model-based object tracking in monocular video. The framework estimates the pose of a 3D object by framing the task as a Bayesian state estimation problem and applying the SIR particle filter algorithm from Chapter 2 to each frame of a video sequence. In other words, the tracking target's pose at time t is represented by a state vector $\mathbf{x}_t$, and an estimate of that state $\hat{\mathbf{x}}_t$ is produced using a set of particles $\{\mathbf{x}_t^i\}_{i=1}^N$ and their corresponding weights $\{w_t^i\}_{i=1}^N$. For each frame of video, the particle set is propagated according to a motion model, updated by a new observation (i.e., frame data), $\hat{\mathbf{x}}_t$ is estimated, and the particle set is resampled. This is shown graphically in Fig. 4.1 with the update stage expanded to include the details of how weights are calculated through particle simulation and evaluation on the GPU. Additionally, Fig. 4.2 shows the data flow for one iteration of the system and Fig. 4.3 shows how the image segmentation, feature extraction, and weight calculation stages are partitioned between kernels and how CUDA's various memory channels are used to exchange data between the stages. It is important to note that the illustrative division of system stages used in Fig. 4.1 does not correspond directly to the actual task partitioning in kernels, which is designed for optimal performance (primarily by avoiding global memory accesses).

The goal of the chapter is to describe the functional and implementation details of the framework; however, justification for the design decisions will not be discussed, as this material has already been covered in Chapters 2 and 3. Additionally, a discussion of the performance and accuracy of the framework is reserved for Chapter 5. The system is described in terms of modular components (or stages) with defined inputs and outputs that reside on either the GPU or CPU. Section 4.1 presents an overview of the framework and each subsequent section describes an individual component.

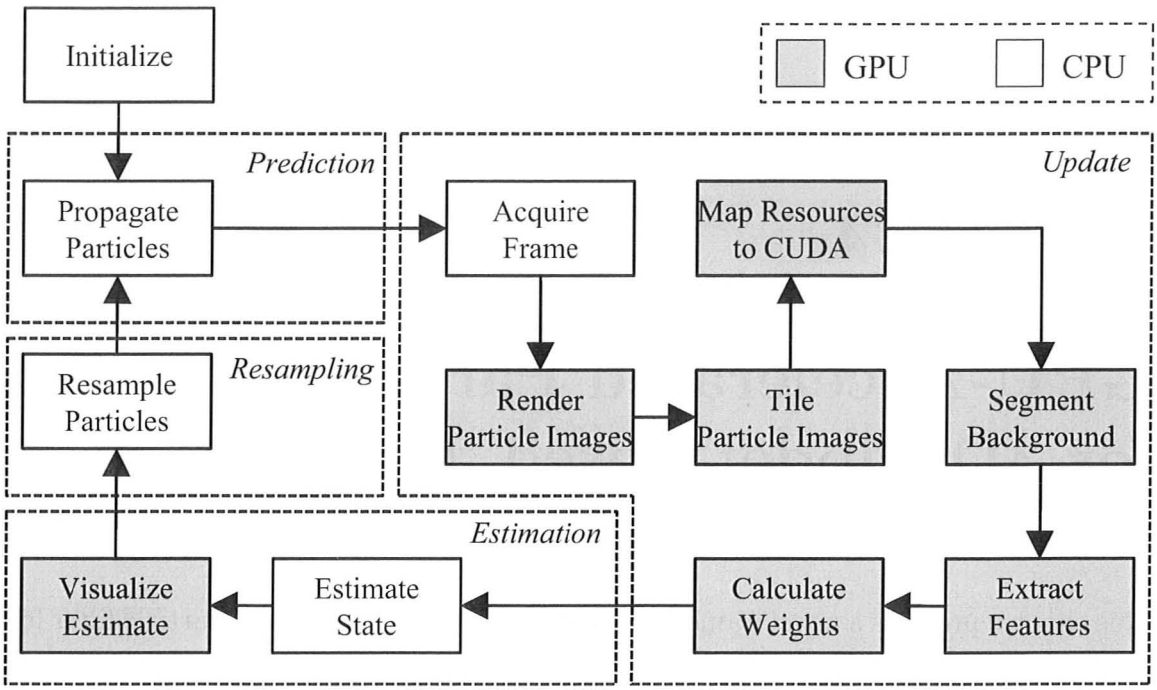

Figure 4.1: GPU-accelerated model-based tracking: system overview

4.1 System Overview

The system is initialized by importing a 3D model of the tracking target, generating an initial particle set, and learning the frame background. With the arrival of each new video frame at time t, the particle set from time $t-1$ is propagated according to the system dynamics. The model is configured to reflect the state defined by each particle and a 2D projection of each configuration is rendered to a texture (particle image). The particle images are tiled on a single surface referred to as the *particle grid*, which is mapped to GPU texture memory along with a surface containing the current frame from the camera. A kernel is launched to segment the tracking target from the background and perform feature extraction on the frame. A second kernel is launched to perform feature extraction on the particle grid, then obtain a quality-of-fit value between each pixel in the feature map of the frame and the corresponding pixels in the feature map of the particle grid. The quality-of-fit values associated with each particle are summed to yield the particles' weights. These weights are transferred back to the CPU where they are used to estimate and visualize the current state of the tracking target. Finally, the weights are used to resample the particle set for the next frame.

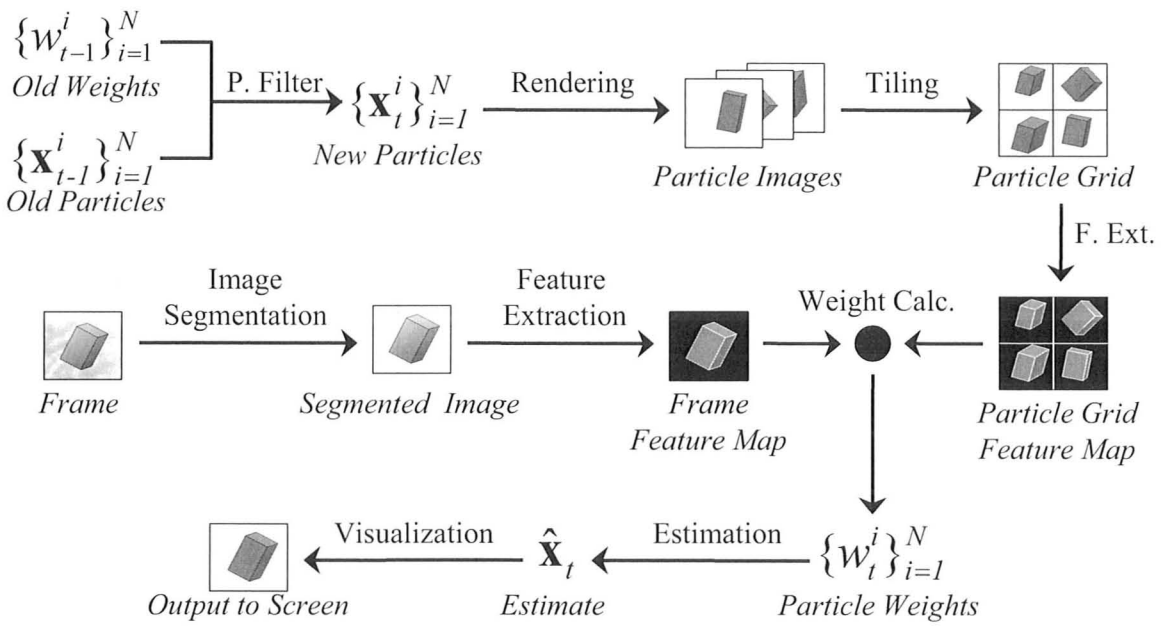

Figure 4.2: GPU-accelerated model-based tracking: data flow

4.2 Configuration and Initialization

The framework was designed to be highly customizable with a number of configurable options including:

- choice of tracking target;
- number of tracking target DOFs;
- number of particles N (must be a perfect square);
- particle image resolution (width must be a multiple of 32, height must maintain aspect ratio of video);
- system dynamics properties (i.e., choice of prior);
- feature extraction algorithm (e.g., Sobel, Canny, none);
- state estimation technique (e.g., average, weighted average);
- number of frames to buffer; and,
- amount of smoothing to apply to output.

The remainder of this section describes the processes involved in initializing the system according to these configuration options.

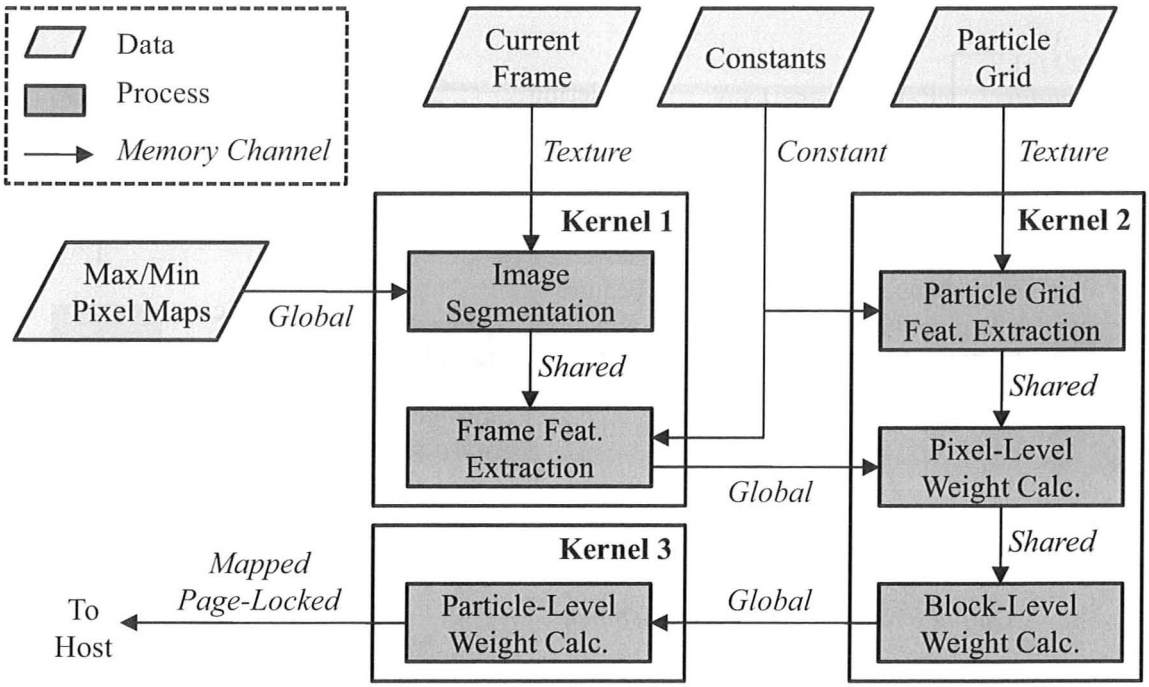

Figure 4.3: GPU-accelerated model-based tracking: kernel partitioning
Three kernels interact through a variety of CUDA memory spaces

4.2.1 Direct3D

Direct3D is configured to use a 32-bit ARGB pixel format and set to ignore the monitor's refresh rate (i.e., present frames as quickly as possible). Ambient and diffuse white light sources are placed in the scene, to approximate the light of an indoor environment, and generic view and projection matrices are created for use in all rendering operations. Vertex buffers, index buffers, surfaces, and textures are allocated in GPU memory, with their resolutions set to reflect the number of particles and particle image resolutions selected. An X file containing a 3D model of a rigid or articulated tracking target is imported, and vertex, index, and material buffers are filled with the model's data. It should be noted that render time increases with the number of vertices in a model. To prepare the model for rendering, its bone hierarchy is traversed and its frame matrices are configured to their default state. Finally, the generic HLSL vertex shader program used for skinning is loaded.

4.2.2 CUDA

CUDA is initialized on the same GPU as Direct3D and surfaces allocated for the current video frame and particle grid are registered for subsequent mapping to texture

memory. Additionally, the area in host memory used for particle weights is page-locked and mapped into device memory space. The edge detector filter parameters are transferred to constant memory, along with several values that can be precalculated to avoid unnecessary, high latency arithmetic (e.g., integer division) in kernels. Finally, the background of the tracking area is identified and transferred into global memory as described in Section 4.7.

4.2.3 Additional Details

An initial particle set is produced by seeding the *r250 uniform random number generator* (Kirkpatrick and Stoll, 1981) and scattering particles near the center of the frame. All particle weights are set to $1/N$. Additionally, to prepare the camera for high-speed streaming, an area in host memory is allocated for the camera's frame buffer (according to the configured size) and the buffer is filled.

4.3 Resampling and Particle Propagation

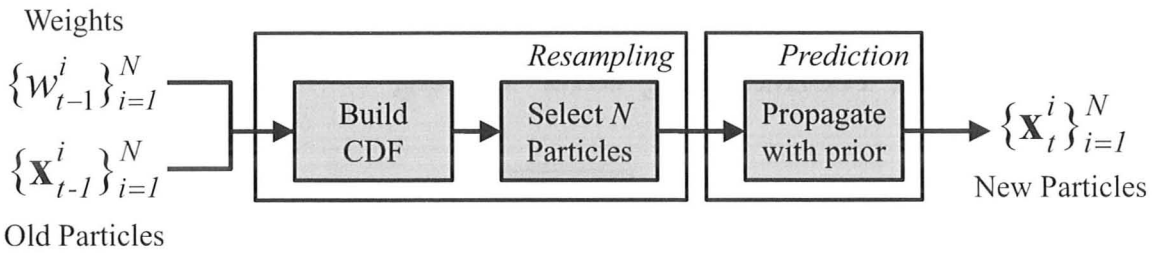

Figure 4.4: Particle resampling and propagation

The particle resampling stage probabilistically selects a new set of particles from the existing set based on a CDF constructed from the existing particles' weights. The particle propagation stage displaces the new particle set using a motion model (random walk, first-order, or second-order with configurable covariance) in an attempt to predict the pose of the tracking target in the next frame. The details of these stages were described in detail in Chapter 2, including the specific SIR particle filter algorithm used in the tracking framework. This information will not be reiterated here to conserve space, but is summarized in Fig. 4.4.

4.4 Frame Acquisition

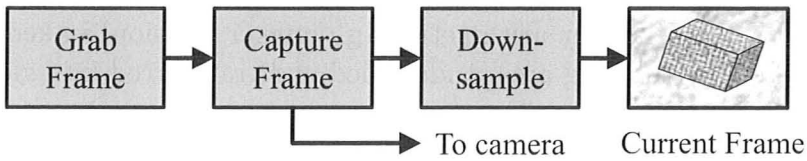

Figure 4.5: Frame acquisition

The frame acquisition stage produces a Direct3D surface (with the same resolution as a particle image) that contains current pixel data from the camera (Fig. 4.5). Each frame can be considered an observation (or measurement) in the terminology of the particle filter. The frame is pulled from a wrapping frame buffer (with a configurable size) in host memory and copied to a temporary, locked surface in GPU memory. Once the copy is complete, the surface is unlocked and an asynchronous instruction is given to the camera to acquire a new frame at the current location in the buffer. This allows the camera to capture frames concurrently with program execution, ensuring the system never has to idle while waiting for a new frame. To downsample the frame from the camera's resolution to the particle image resolution, the pixel data is copied from the temporary surface to a surface that has been registered with CUDA (with the resolution of a particle image) using an efficient Direct3D stretch operation.

4.5 Model Rendering and Tiling

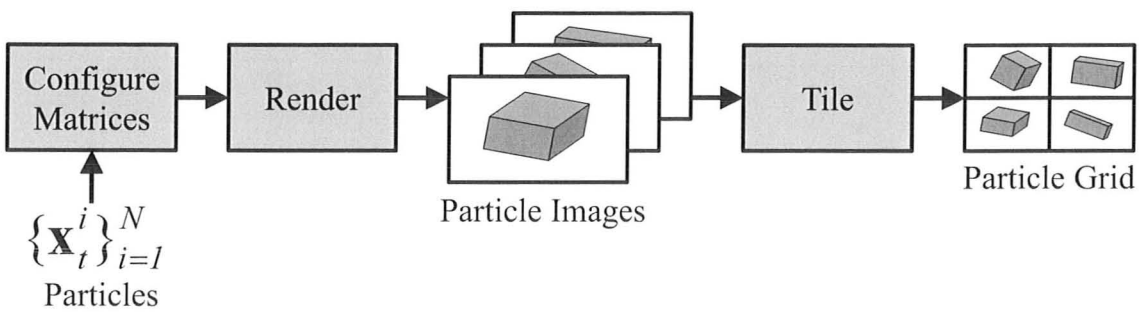

Figure 4.6: Model rendering and tiling

The goal of model rendering and tiling is to generate a particle grid containing N particle images that correspond to the states of particles $\{\mathbf{x}_t^i\}_{i=1}^N$ (Fig. 4.6). For each particle in the set, the model's world transform matrix is updated to reflect the global

state parameters of that particle. The model's bone hierarchy is then recursively traversed and its frame matrices are updated according to the local (joint) parameters of the particle. Once all matrices are updated, the bone hierarchy is recursively traversed a second time and the mesh corresponding to each bone is rendered to a texture with a white background, producing the particle image. The virtual lights, materials, and projective matrix parameters set during initialization ensure that particle images represent realistically shaded projections of the actual tracking target. Once all particle images are rendered, the textures are applied to quads that are tiled evenly across the particle grid surface, which was registered with CUDA during initialization. The result is a single, large surface that contains pixel data for N particle images and can be mapped into CUDA's texture memory as a single CUDA array.

4.6 Mapping Direct3D Resources to CUDA

The Direct3D surface containing the particle grid and current frame are registered with CUDA and can therefore be mapped to CUDA's texture memory and accessed by kernels using texture fetches. This is done by mapping each surface to a CUDA array and passing the address of that array to the kernels. The CUDA arrays are bound to textures when a kernel launches and unbound once the kernel has completed execution. The amount of time consumed by mapping and binding is small, but scales with the number of surfaces. This demonstrates the advantage of packaging particle images into a single particle grid, which requires just one mapping instead of hundreds.

4.7 Image Segmentation

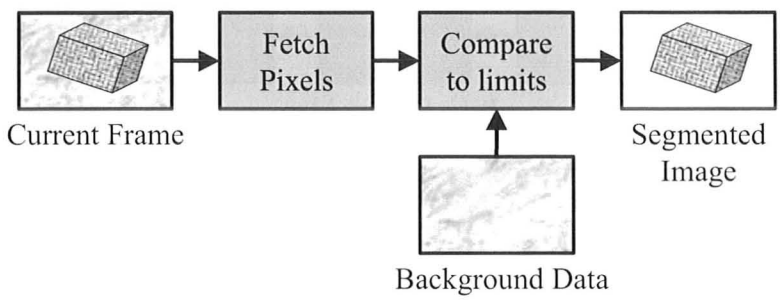

Figure 4.7: Image segmentation

A thresholding approach to background subtraction is applied to each frame in the video sequence to segment the tracking target from the background (Fig. 4.7). During

initialization, a 1 second training video sequence, which contains only the frame background, is captured and loaded into memory. Corresponding pixels in each frame are compared to identify the maximum and minimum value that each pixel location achieves. From this, maximum and minimum pixel value maps are generated and transferred into global memory. By observing pixel values over a period time, noise in the video sequence is accounted for.

During image segmentation, the CUDA array containing the current frame is bound to a texture reference and a kernel (*kernel 1* in Fig. 4.3) uses texture fetches to load pixels into registers as floating point data (red, green and blue pixel components are summed to produce a single value). Each pixel is compared to corresponding limits in the maximum and minimum pixel maps in global memory. If a pixel falls within a configurable threshold of the corresponding limits, it is assumed that the pixel represents image background and it is suppressed (set to white); otherwise, it is assumed the pixel is part of the tracking target and its value is preserved. Although this approach requires a static camera position be maintained, this is not a significant restriction in many applications.

The image segmentation operation is parallelized using a 256 (32×8) thread block that tiles the frame. The resulting floating point data represents the segmented image and is written to shared memory for subsequent feature extraction (by the same kernel). Warp divergence is a concern during image segmentation, specifically around the edges of the object where neighbouring threads will branch; however, because the frame data is very small with respect to the particle grid, there is minimal effect on overall processing time.

4.8 Feature Extraction

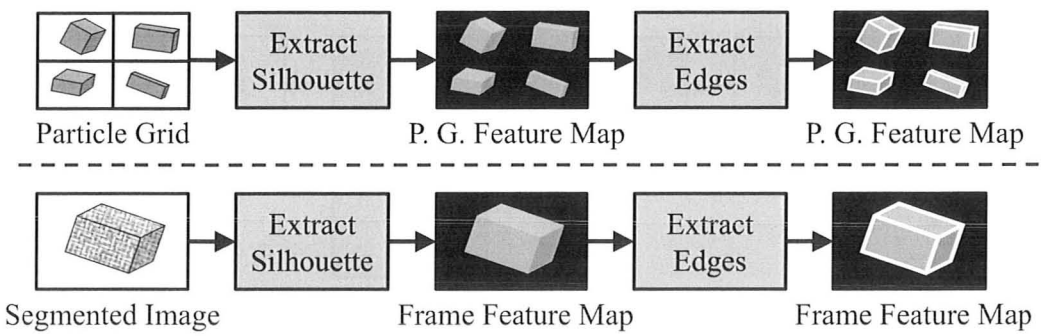

Figure 4.8: Feature extraction

The goal of feature extraction is to isolate comparable aspects of the current frame and particle images, while suppressing factors that will detract from the quality of the

comparison (Fig. 4.8). While there are many possible approaches to this, the framework was tested using silhouette and edge features, which highlight the shape and orientation of objects, while suppressing textures, colours, and some shading (contrasting shading can contribute to edge features). The framework can be configured to ignore feature extraction entirely; however, this is reasonable only when the model geometry, virtual light, material, and camera lens configurations accurately reflect the true nature of a tracking target and environment.

Feature extraction is performed separately on the segmented image and the particle grid by *kernel 1* and *kernel 2*, respectively (Fig. 4.2 and Fig. 4.3). *Kernel 1* executes first, operating on the segmented image floating point data in shared memory and stores the resulting feature map in global memory. *Kernel 2* operates on the particle grid, which resides in texture memory, and leaves the resulting feature map in shared memory for subsequent weight calculation. Because the feature extraction process is effectively identical for both kernels, the algorithm is described exclusively for the particle grid.

Silhouette detection is trivial but can still benefit from a parallel implementation; white pixels of the particle grid are considered part of the background and result in a black pixel on the feature map, whereas non-white pixels are considered part of the tracking target and are set to gray. To perform edge detection, the system can be configured to apply the Sobel or Canny edge detector algorithms. The trade-off is computational complexity versus quality of edge response. The Sobel detector involves two convolution operations, whereas the Canny detector involves three convolutions and additional steps for clamping, thresholding, and classification of edges. Because the Canny algorithm necessitates a great deal of branching (and consequently warp divergence) and includes an *arctan* operation to identify edge direction, it is not nearly as well-suited to a GPU implementation as the Sobel approach. Nonetheless, the quality of edge detection afforded by the Canny algorithm could be worth the additional complexity in certain tracking scenarios. In either case, pixels identified as edges are set to white on the feature map.

Feature detection is executed by a kernel with heavy reliance on shared memory and barrier synchronization, as shown in Fig. 4.9. Adjacent pixels are loaded into adjacent shared memory banks by adjacent threads using texture fetches. This adheres to the access patterns for both memory types. Each pixel is classified as a member of the image background or tracking target by setting a *silhouette flag* in a register. Convolution requires the edges of the image to be "padded" by extending the outermost pixels in each direction. To facilitate this, a barrier synchronization is issued and a subset of threads copy values around the perimeter of shared memory. After padding, another barrier synchronization is issued and the selected edge detector is executed. In the case of the Sobel detector, this means reading filter parameters from constant memory and using them to calculate the edge response of a pixel, accessing other

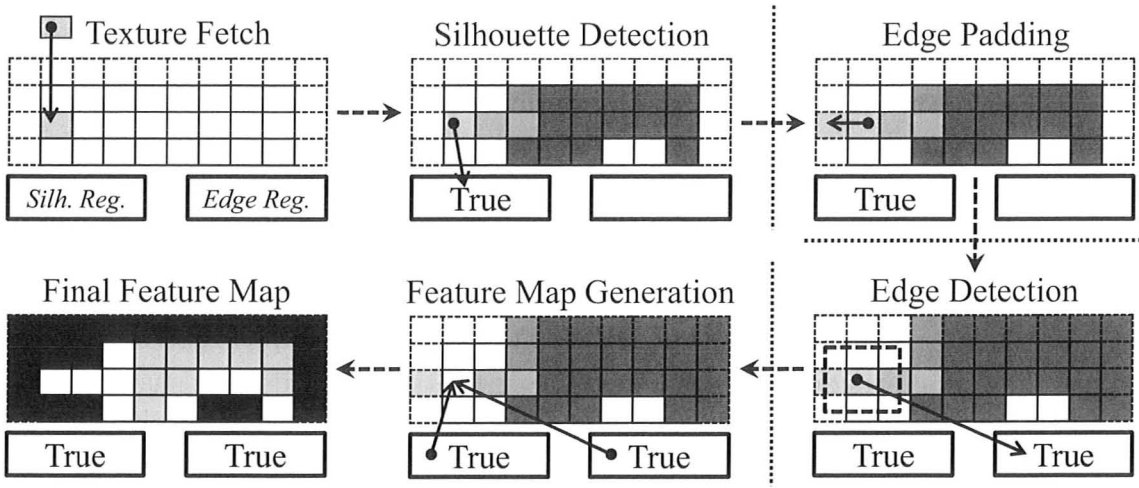

Figure 4.9: Shared memory and flag registers during feature extraction
Each grid represents pixel data in shared memory. Dashed lines indicated extended edges (top, left and right only). Barrier synchronizations are shown as dotted lines intersecting arrows.

pixels from adjacent shared memory banks as necessary, and setting an *edge flag* in a register if the result exceeds a configurable threshold. Another barrier synchronization is issued, ensuring that all threads in the block are finished with the image data in shared memory before it is overwritten by the feature map. Each shared memory bank is set to black (background) or gray (silhouette) according to the silhouette flag set earlier and valid edge pixels are overlaid in white according to the register flag, completing the feature map. The Canny detector follows a similar pattern, but with additional barrier synchronizations and shared memory writes required for each additional step involved in the algorithm.

An efficient CUDA implementation requires tasks be partitioned in a way that maximizes utilization of GPU resources. Here, a 32×8 thread block is used, regardless of particle image resolution or number of particles; the number of blocks in the grid, however, varies with these parameters. To ensure blocks evenly tile the particle grid, the width of particle images must be a multiple of 32 and the height must preserve the aspect ratio. To minimize the number of texture fetches, it is desirable for each block to load as much data into shared memory as possible, without excessively impacting multiprocessor occupancy. Using the occupancy calculator, it was determined that each thread can safely load three 32-bit pixels into shared memory while maintaining an SM occupancy of 50 percent (Fig. 4.10). This pixel access pattern full exploits shared memory, while adhering to all memory access patterns, maintaining occupancy of 50 percent, and minimizing register usage.

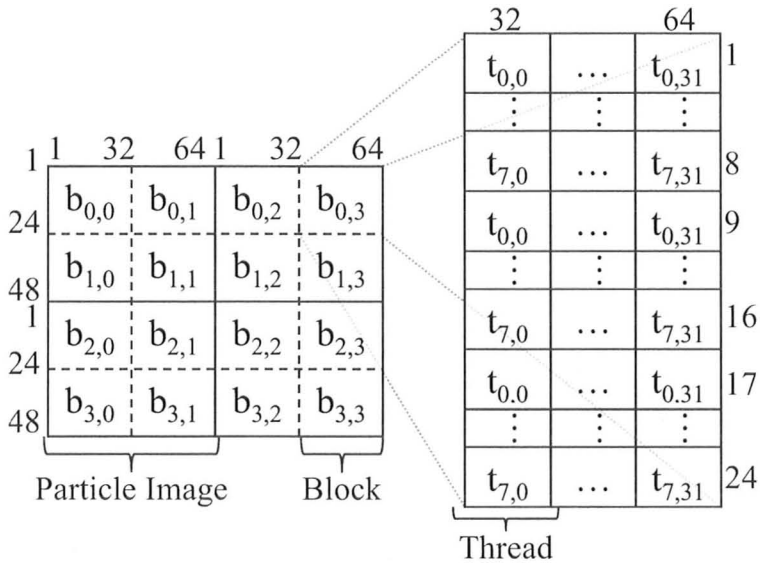

Figure 4.10: Particle grid GPU mapping
Partitioning of a 2×2 particle grid with particle image resolution of 64×48, block size of 32×8, each thread fetching 3 pixels along the y-direction. Pixel indices shown outside boxes, block IDs and thread IDs shown inside the left and right boxes, respectively.

4.9 Particle Weight Computation

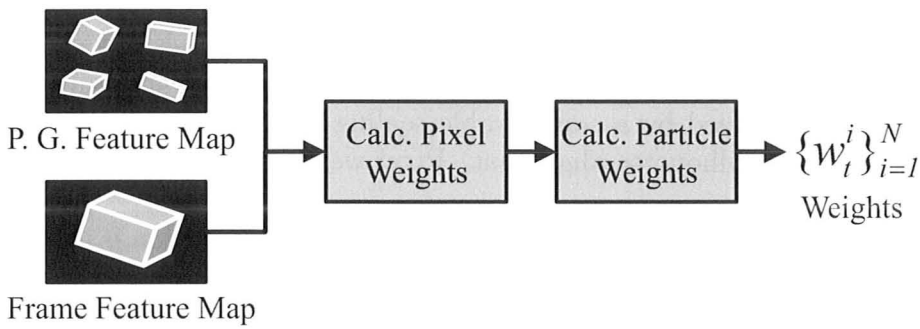

Figure 4.11: Particle weight computation

The goal of weight calculation is to compare the feature map of the current frame, which is in global memory, to the feature map of each particle image in the particle grid, which is partitioned across shared memory (Fig. 4.11). The result is a weight for each particle with pixel-level accuracy. A divide-and-conquer approach is used to

first generate a weight for each pixel in the particle grid, then sum pixel weights to generate a weight for each thread block, and finally sum block weights of all blocks that tile a given particle image to generate a weight for that particle (Fig. 4.12).

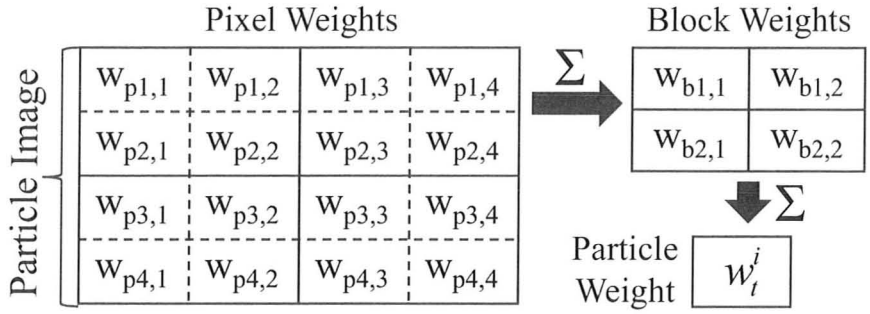

Figure 4.12: Weight calculation
Pixel weights are summed to produce block weights, which are summed to produce a particle weight.

Since shared memory does not persist across kernel launches, and it would require substantial global memory accesses to launch a new kernel, weight calculation is done by the same kernel that performs feature extraction on the particle grid (*kernel 2* in Fig. 4.3). Each thread loads three registers with the frame pixels corresponding to the previously fetched particle grid pixels and pixel weights w_{pixel} are calculated according to

$$w_{pixel} = \epsilon_{max} - |p_{frame} - p_{particle}| \tag{4.1}$$

where p_{frame} is the floating point value of a frame pixel, $p_{particle}$ is a particle pixel, and ϵ_{max} is the maximum possible error (i.e., residual) the pixels can achieve, which varies according to the feature extraction configuration. Additionally, weights of edge pixels are multiplied by a configurable scaling factor, giving higher priority to edge alignment than silhouette alignment. Pixel weights are written back to shared memory.

The most computationally complex aspect of weight calculation is the summation of pixel weights. This step is performed by a parallel reduction algorithm similar to (Harris, 2007). To begin, each row of a thread block reduces its corresponding shared memory row to a single value. This done in five iterations, the first three of which are partially shown in Fig. 4.13. In the first iteration, 16 adjacent threads each sum two adjacent shared memory cells and store the results in shared memory, overwriting the first operand. The next iteration uses eight threads to sum the results of the previous iteration, and so on. This algorithm is applied to three shared memory rows by each row of the thread block, yielding a single column of 24 weights for the block, without any bank conflicts. Finally, a single thread serially adds the values in this column to

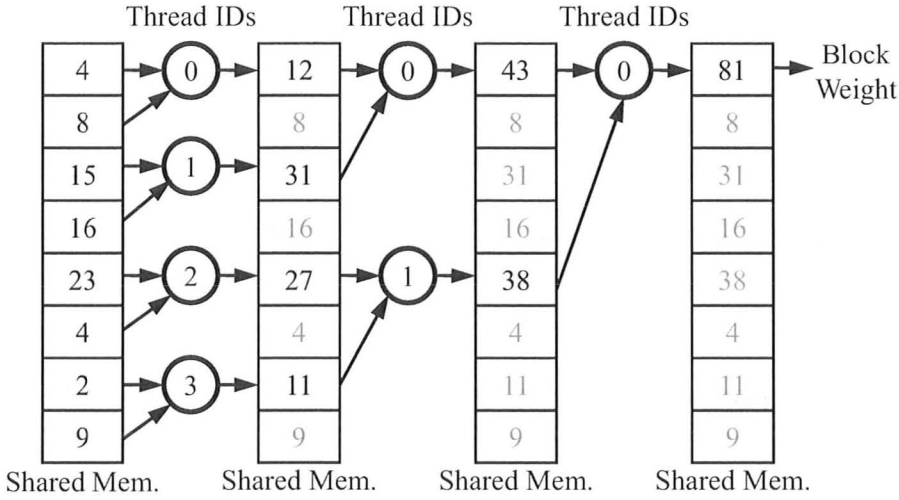

Figure 4.13: Parallel reduction algorithm to sum pixel weights.

produce a single weight for the entire block and stores it in global memory.

The weights of all blocks that tile each particle image are then summed by a single thread block in *kernel 3*. Launching a new kernel ensures that all previous threads in the grid have completed execution, acting as a global synchronization barrier. The resulting particle weights $\{w_t^i\}_{i=1}^N$ are written to page-locked host memory for a high-bandwidth transfer across the PCI-E bus to the CPU.

The primary drawback to GPU computing is the latency associated with data transfers between the GPU and CPU. The described approach circumvents excessive data transfers by performing model rendering, projection, feature extraction, and particle image evaluation entirely on the GPU, requiring only one floating point value per particle be transferred to the host.

4.10 State Estimation and Visualization

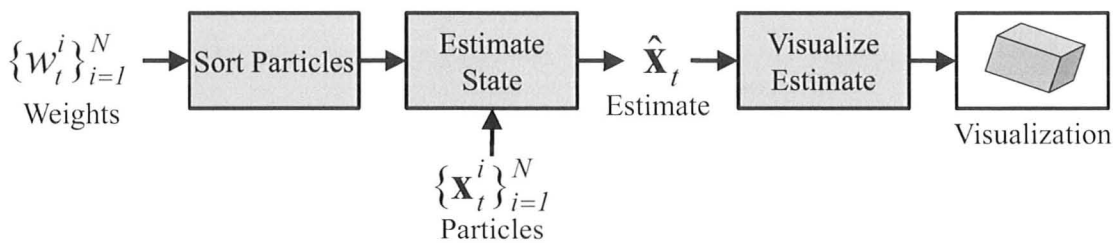

Figure 4.14: State estimation and visualization

The system estimates the state of a tracking target $\hat{\mathbf{x}}_t$ on the CPU using the weights obtained from the GPU (Fig. 4.14). This can be done by selecting the single highest weighted particle (2.8), or by computing the average (2.9) or weighted average (2.10) of the M highest weighted particles. The quick-sort algorithm is used to sort the particles by their weights and an estimate is calculated based on the states of applicable particles in the set $\{\mathbf{x}_t^i\}_{i=1}^N$. This estimate is added to a history of previous estimates, which is used when calculating velocity and acceleration values for the motion model used in particle propagation.

The estimate is also used to render a visualization of the object pose to the screen, both as a wire frame overlay on the video sequence and as a fully shaded projection. To compensate for the jitter effect of the particle filter, the framework can be configured to average the current estimate with one or more previous estimates before displaying the model. This can make the system seem less responsive at lower frame rates, but has no actual effect on tracking, as the smoothed estimates are not stored. When tracking is successful, the end result is a 3D model of the tracking target that follows the true tracking target as it moves through the video sequence, with a one-to-one mapping in every degree of freedom.

Chapter 5

Results and Analysis

The GPU-accelerated model-based tracking framework described in Chapter 4 was tested with the goal of determining how many particles are required to accurately and robustly track an object moving in a video sequence and to determine the frame rate at which the required particles can be simulated and evaluated. This chapter presents the methodology, results, and analysis of a variety of experiments conducted on the system, using various settings and tracking targets.

5.1 Methodology

The framework was tested using five unique 600-frame sequences, each of which shows one of two tracking targets moving in real or synthetic video with 6, 8, or 10 DOFs. This section will describe the nature of the experiments conducted, the metrics used to assess the results, and the hardware and software with which the tests were run.

5.1.1 Tracking Targets

To demonstrate the generality of the framework, two unique tracking targets (Fig. 5.1) and applications are considered:

- a rigid "wand" with 6 DOFs for AR applications; and,
- an articulated hand with a flexible wrist and index finger (6 to 10 DOFs) for HCI applications.

The tracking wand was designed to have a unique projection when viewed from any angle, meaning there is no ambiguity when observing its edge and silhouette features. The 3D model of the wand was created with the open-source 3D creation suite Blender (Roosendaal and Selleri, 2009) and is comprised of 36 vertices (68

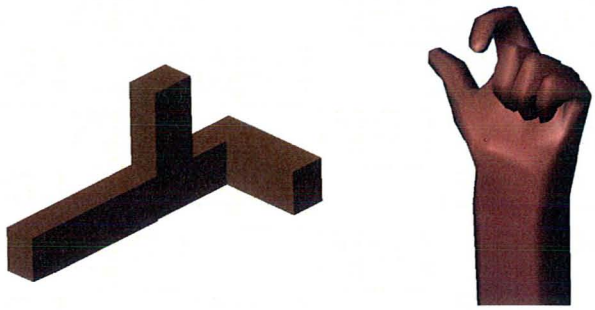

Figure 5.1: 3D tracking target models
Rigid wand and articulated hand

edges, 34 faces). The wand is used as a tracking target in real and synthetic video experiments to demonstrate the utility of the system when tracking an ideal target object. The wand could also be useful in AR applications, which often require virtual objects to be "attached" to real objects in video with great accuracy.

An articulated 3D hand model with six global DOFs, two DOFs in the wrist joint, one DOF in the index finger MP joint, and one DOF for index finger PIP and DIP joints was also created with Blender (Fig. 5.2) and is comprised of 355 vertices (742 edges, 389 faces) attached to a skeleton of nine bones and seven joints. It is possible to "lock" one or more joints to reduce the dimensionality of the estimation problem, depending on the requirements of the tracking application. In its default position, the hand has a fully extended finger and thumb, with all other fingers flexed against the palm. Hand tracking has many applications, specifically in the domain of HCI, where natural hand motion is used as an alternative to the mouse and keyboard.

5.1.2 Experimental Parameters and Metrics

In each experiment, the system was assessed in terms of *tracking quality* (i.e., accuracy and robustness) and *performance* (i.e., frame rate) as the number of particles varies between 36 and 3,600 and the particle image resolution varies between 32×24 and 160×120; however, the overall resolution of the particle grid is limited by the GPU, constraining the particle count at the highest resolution to approximately 2,300. Additionally, in each experiment, the *optimal operating parameters* are identified, qualitatively defined as the system settings wherein an increase in either particle image resolution or particle count would yield negligible gains in tracking quality and needlessly lower the frame rate. Tracking quality and performance results are examined in detail for the optimal system settings of each experiment. In the presentation of all results, this thesis adopts the convention of labeling axes on only the bottom- or left-most graph(s) of each figure to avoid redundancy. Finally, when observing

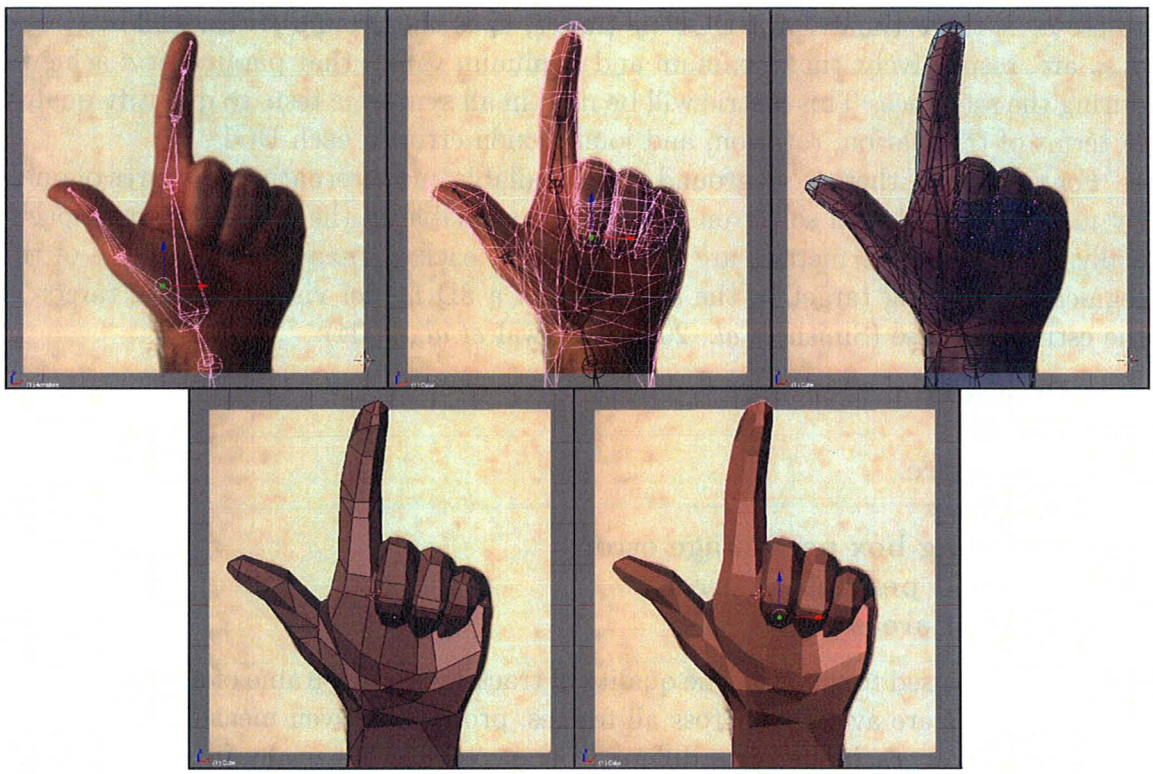

Figure 5.2: Hand model creation with Blender

quality results, it is important to remember that tracking with a particle filter is an intrinsically random operation, meaning anomalies can occur during testing; no tests were repeated in response to such anomalies to preserve the integrity of data.

The system was evaluated using both Sobel and Canny edge detectors for feature extraction; however, the Canny detector demonstrated negligible gains in tracking quality and impeded performance, therefore, results are shown exclusively for experiments that used silhouette and Sobel detection. The difference in performance and quality of the two detectors is discussed in Appendix B. Additionally, Appendix C discusses the impact that the system dynamics model has on tracking quality.

To quantify tracking quality in synthetic tests, an estimate $\hat{\mathbf{x}}_t$ is directly compared to a well-defined ground truth $\mathbf{x}_t$, providing a measure of tracking error. Specifically, the average error of a state vector parameter d during a sequence is calculated as the mean absolute error (MAE) between the ground truth and estimate of that parameter:

$$Error = \frac{1}{600} \sum_{t=1}^{600} \frac{|\hat{d}_t - d_t|}{d_{max} - d_{min}} \tag{5.1}$$

where $\hat{d}_t$ is the estimate of a DOF at time t, d_t is the ground truth, and d_{max} and d_{min} are, respectively, the maximum and minimum values that parameter d achieves during the sequence. This metric will be used in all synthetic tests to quantify quality in terms of translation, rotation, and joint flexion error in each DOF.

For real video, there is no ground truth available and alternative measures of quality must be used based solely on information available in the test sequence. Specifically, seven separate metrics are employed that each compare the silhouette of the segmented tracking target to the silhouette of a 3D model visualizing the target in the estimated pose (Smith *et al.*, 2005; Agarwal *et al.*, 2004):

- **Precision**
- **Recall**
- **F-measure**
- **Accuracy**
- **Bounding box percentage error**
- **Centroid percentage error**
- **Ratio of areas**

Each metric is used to quantify the quality of tracking in each frame of a test sequence, and the results are averaged across all frames, producing seven measures of tracking quality for a test, each offering a different perspective on the evaluation. These seven metrics are used in all experiments involving real video.

The first four metrics frame tracking as a classification task, where each pixel can be classified as either part of the tracking target or part of the background (within a bounding box containing both objects). Each classification can be characterized as one of the following:

- **True positive (TP)**: Pixel correctly classified as part of the tracking target.
- **True negative (TN)**: Pixel correctly classified as part of the background.
- **False positive (FP)**: Pixel incorrectly classified as part of the tracking target.
- **False negative (FN)**: Pixel incorrectly classified as part of the background.

The metrics are formulated as follows (van Rijsbergen, 1979):

$$Precision = \frac{1}{600}\sum_{t=1}^{600}\frac{TP_t}{TP_t + FP_t} \tag{5.2}$$

$$Recall = \frac{1}{600}\sum_{t=1}^{600}\frac{TP_t}{TP_t + FN_t} \tag{5.3}$$

$$FMesaure = \frac{1}{600}\sum_{t=1}^{600}\frac{2TP_t}{2TP_t + FN_t + FP_t} \tag{5.4}$$

$$Accuracy = \frac{1}{600}\sum_{t=1}^{600}\frac{TP_t + TN_t}{TP_t + TN_t + FP_t + FN_t} \tag{5.5}$$

where TP_t is the number of pixels classified as true positives at time t, and similar notation is used for the other three classes. In other words, precision is the ratio of pixels correctly classified as part of the tracking target to the total number of pixels classified as part of the tracking target. Recall is the ratio of pixels correctly classified as part of the tracking target to the total number of pixels in the actual tracking target. F-measure is a single metric that combines precision and recall (their harmonic mean), and accuracy is an overall ratio of number of pixels correctly classified to the number of pixels in the bounding box containing both silhouettes.

The average bounding box percentage error for a sequence (across all frames t) is expressed in terms of the geometric distance between the coordinates of the centre of the bounding box (b_t^x, b_t^y) containing the tracking target and the coordinates of the centre of the bounding box $(\hat{b}_t^x, \hat{b}_t^y)$ containing the visualization of the estimate:

$$BoundingBoxError = \frac{1}{600}\sum_{t=1}^{600}\frac{\sqrt{(\hat{b}_t^x - b_t^x)^2 + (\hat{b}_t^y - b_t^y)^2}}{\sqrt{(b_{x,max} - b_{x,min})^2 + (b_{y,max} - b_{y,min})^2}} \tag{5.6}$$

where $b_{x,min}$ and $b_{x,max}$ are, respectively, the minimum and maximum values the center of the bounding box containing the tracking target achieves in the x-direction during the sequence, and similar notation is used for the y-direction. This formulation expresses the percentage error for each frame as the ratio of the linear distance (in pixels) between the two bounding box centres to the total distance traveled by the centre of the tracking target's bounding box. Using the same notation, the average centroid percentage error for a sequence is defined as:

$$CentroidError = \frac{1}{600}\sum_{t=1}^{600}\frac{\sqrt{(\hat{c}_t^x - c_t^x)^2 + (\hat{c}_t^y - c_t^y)^2}}{\sqrt{(c_{x,max} - c_{x,min})^2 + (c_{y,max} - c_{y,min})^2}} \tag{5.7}$$

The ratio of areas metric provides a measure of quality based on silhouette overlap:

$$RatioOfAreas = \frac{1}{600}\sum_{t=1}^{600}\frac{|T_t \cap E_t|}{|T_t \cup E_t|} \tag{5.8}$$

where T_t is the set of pixels in the tracking target at time t, E_t is the set of pixels in the visualization of the estimate at time t, and $|\cdot|$ denotes the cardinality (i.e., number of pixels) of the set. This notation provides an alternative (equivalent) formulation

for precision and recall as well:

$$Precision = \frac{1}{600}\sum_{t=1}^{600}\frac{|T_t \cap E_t|}{|E_t|} \tag{5.9}$$

$$Recall = \frac{1}{600}\sum_{t=1}^{600}\frac{|T_t \cap E_t|}{|T_t|} \tag{5.10}$$

Performance results quantify the speed at which the system operates and are expressed in frames per second. Additionally, performance is compared to a similar CPU-based implementation of the system in terms of overall speedup and speedup of particle evaluation, which is the target GPU-acceleration. All performance results were obtained using CUDA's GPU timers, which are accurate within 0.5 μs, but benchmark the overall system time, and can therefore be susceptible to variations in performance caused by background applications running on the system.

5.1.3 Test Bench

Tests were conducted on an Intel Core2 Duo E8400 3.0-GHz PC with 4 Gbyte of 1,333 MHz DDR3 RAM, running Windows 7, CUDA 3.1, and DirectX 9.0c. An NVIDIA GeForce GTX295 graphics card (hardware specification shown in Table 5.1) was used for GPU-acceleration; however, only one of the two available GPUs was utilized due to restrictions within Direct3D. The 600-frame (approximately 5 second) video sequences were captured by an Allied Vision Technology GC660C 119-fps 659×493 GigE CCD camera (Allied Vision Technologies, 2010) at a resolution of 320×240.

Table 5.1: NVIDA GTX295 Graphics Card Hardware Specification

SMs	60 (2×30)
SPs	480 (2×240)
Shader Clock	1,242 MHz
Core Clock	576 MHz
Memory Clock	999 MHz
Memory Size	1,792 MB GDDR3 (2×896 MB)
Memory Bandwidth	223.8 GB/s
Power Consumption	289 W
Driver Version	197.45

5.2 Tracking Quality Results

This section describes the tracking quality results for five separate experiments using metrics (5.1) to (5.8). Experimental results using real video are presented first, beginning with 6-DOF rigid wand tracking in Section 5.2.1, followed by 6-DOF rigid hand tracking in Section 5.2.2 and 10-DOF articulated hand tracking in Section 5.2.3. Synthetic video results follow for 6-DOF rigid wand tracking in Section 5.2.4 and 8-DOF articulated hand tracking in Section 5.2.5.

5.2.1 Rigid Wand Tracking - Real Video

Rigid wand tracking (6-DOF) is framed as an AR application, wherein the tracked pose parameters of the wand are applied to a virtual Omaha teapot, which is overlaid on the scene background. The user is intended to experience a one-to-one mapping between movement of the wand and the teapot.

In the sequence, the wand is shown moving quickly through all DOFs with several instances of simultaneous translation and rotation, including a 360 degree rotation along the x-axis. The wand was tracked using a second-order motion model on each DOF and a weighted average of the 25 highest-weighted particles was used to compute the estimate. A wand tracking demonstration is shown in Fig. 5.3, a sample of the particle grid is shown in Fig. 5.4, and tracking quality results are shown in Fig. 5.5.

Fig. 5.3 shows a column of six unique (non-consecutive) video frames, each containing the tracking target in a different pose. For each frame, five perspectives on the functionality of the tracking system are presented in a row. In the left-most column, the unprocessed input video frame is shown. The second column shows the results of image segmentation and feature extraction; the background is shown in black, the silhouette is gray, and edges are overlaid in white. The third column presents an *error map*, which offers three comparisons between an image of the the silhouette of the segmented tracking target and an image of the silhouette of the 3D model of the tracking target configured in the estimated pose; the residual between the two images is shown in black, the bounding box of the the tracking target is shown in blue and the bounding box of the estimate is shown in green, the centroid of the tracking target is shown as a cluster of blue pixels and the centroid of the estimate is shown as a cluster of red pixels. The fourth column shows the input video frame with the estimate rendered as a wire frame overlay. Finally, the fifth column shows a fully shaded 3D model rendered in the estimated pose, replacing the tracking target in the scene. The object in this screen could be a 3D model of the tracking target, or an alternative object relevant to the application, such as the teapot used in Fig. 5.3.

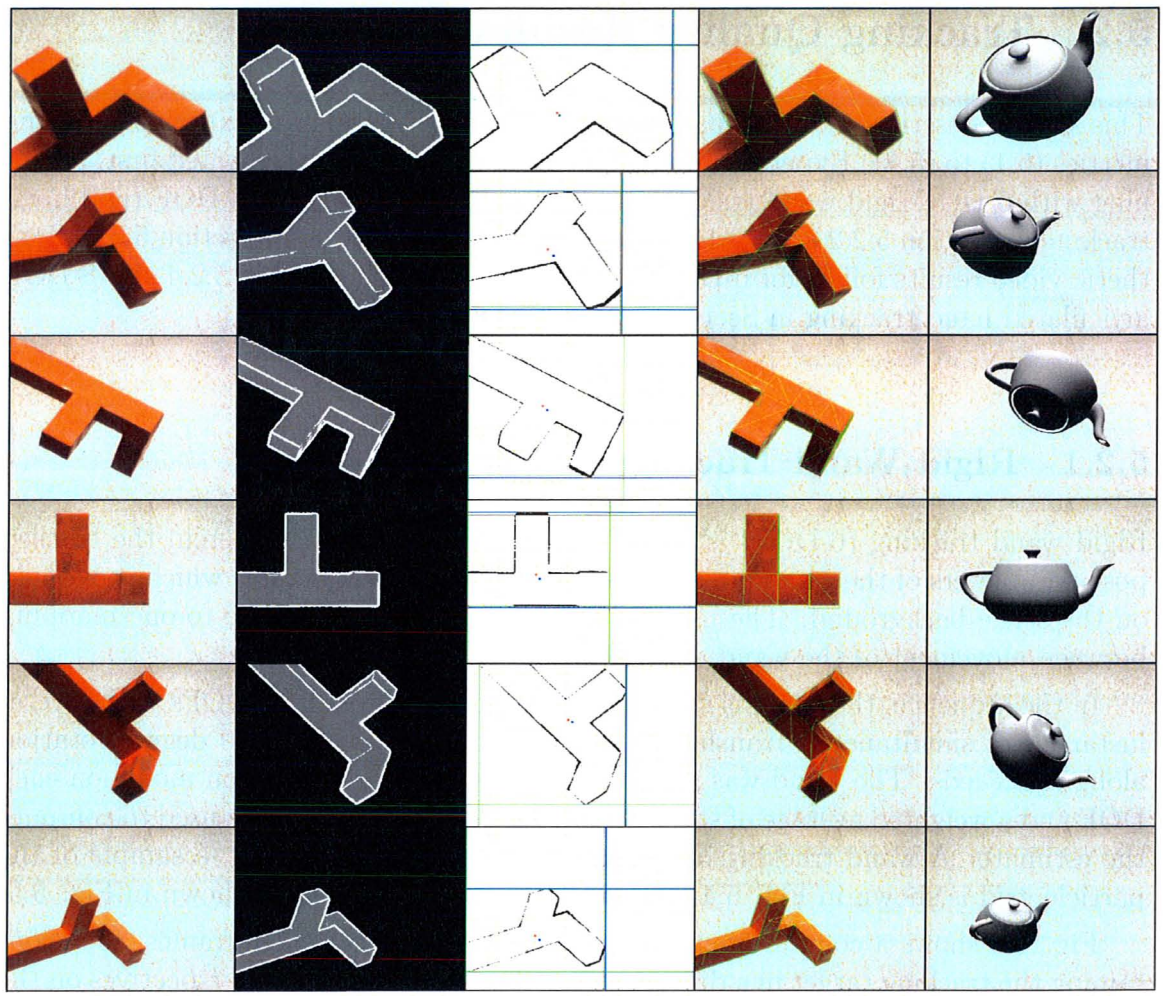

Figure 5.3: 6-DOF rigid wand tracking demonstration
From left to right: input video, feature map, error map (centroids, bounding boxes, and residual), wire frame overlay, estimated teapot pose

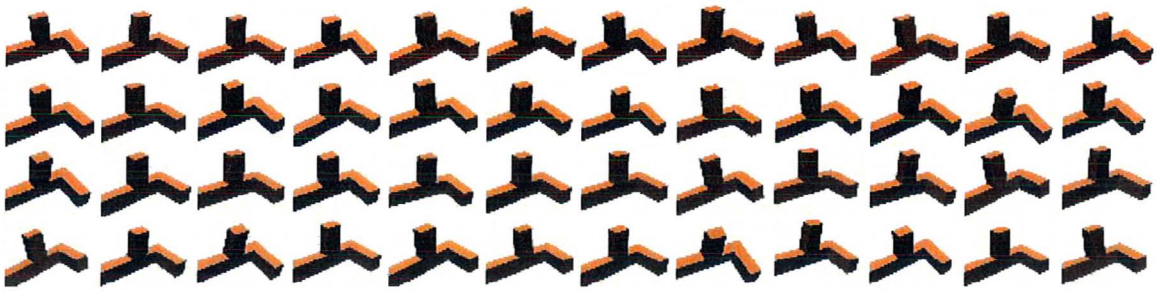

Figure 5.4: 4×12 sample of the wand particle grid

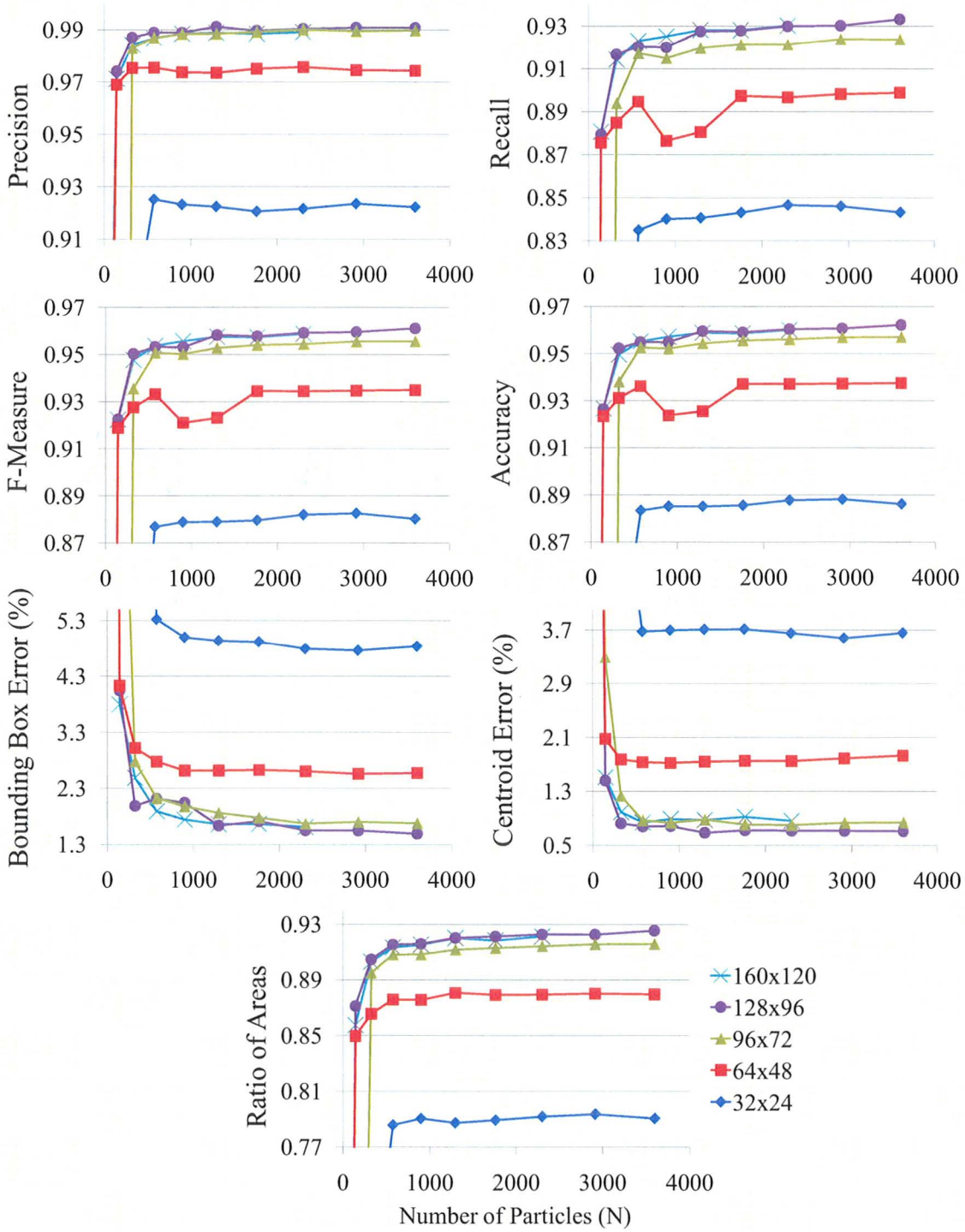

Figure 5.5: 6-DOF rigid wand tracking quality results

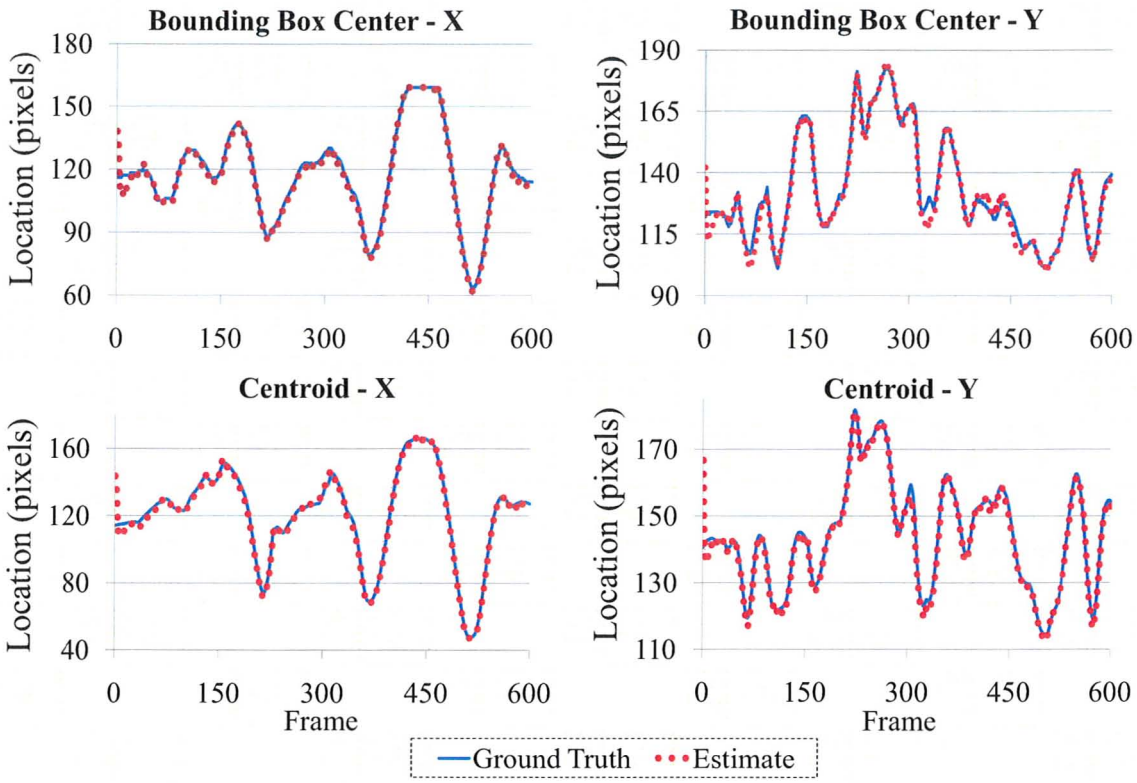

Figure 5.6: 6-DOF rigid wand tracking at optimal settings
576 particles, 96×72 particle image resolution

The sample of the particle grid shown in Fig. 5.4 represents a small subset of the overall grid, which contains between 36 and 3,600 particle images depending on the system configuration. Each particle image shows the 3D model of the tracking target in a slightly different pose, according to the state parameters produced by the particle filter. Note that figures similar to Fig. 5.3 and Fig. 5.4 are provided for all subsequent real-video experiments, but are not described in detail to avoid redundancy.

Fig. 5.5 demonstrates that tracking quality increases with both particle image resolution and particle count; however, for a resolution of 96×72 and a particle count of approximately 600, increasing either parameter provides diminishing returns. These settings are therefore considered optimal for this experiment, providing an average precision, F-measure, and accuracy greater than 95 percent, recall and ratio of areas greater than 90 percent, bounding box error less than 2 percent, and centroid error less than 1 percent. A qualitative assessment agrees with the quantitative findings. A user experiences the sensation of holding the virtual teapot and tracking integrity is maintained even when the wand is moved sporadically.

Fig. 5.6 shows the location of the tracking target's centroid and bounding box

during the 600-frame sequence with an overlay of the estimated locations when the object is tracked using the optimal settings described above. With the exception of a few frames at the start of the sequence (before an initial "lock" is obtained), minimal divergence between the ground truth and estimate can be seen. The most notable error occurs around frame 320, where the object quickly changes trajectory along the y-direction. Here, the particle filter's prediction stage can be seen propagating the particle set (and estimate) based on previous velocity and acceleration, then quickly recovering with the arrival of few new observations (i.e., frames).

5.2.2 Rigid Hand Tracking - Real Video

Rigid hand tracking (6-DOF) is framed as an HCI application, wherein the user has control over a virtual hand with an extended index finger and thumb. In the sequence, a hand is shown moving quickly through all DOFs with a focus on translation. The hand was tracked using a second-order motion model on each DOF and a weighted average of the 36 highest-weighted particles was used to compute the estimate. A demonstration of rigid hand tracking is shown in Fig. 5.7, a sample of the particle grid is shown in Fig. 5.8, and tracking quality results are shown in Fig. 5.9.

Results are similar to wand tracking, demonstrating an increase in tracking quality with both particle image resolution and particle count. In this case, all metrics saturate once approximately 300 particles are used with a particle image resolution of 96×72 or higher. These settings are therefore considered optimal for the sequence; the hand is tracked with an average precision, F-measure, and accuracy greater than 94 percent, recall and ratio of areas greater than 90 percent, bounding box error less than 4 percent, and centroid error less than 2 percent.

Though similar to wand tracking, results indicate that rigid hand tracking performs with slightly lower quality. This can be attributed to two factors. First, unlike the wand, the projection of a hand does not offer a unique silhouette at any angle, meaning feature maps for various orientations can appear identical, except for edge features. Second, a realistic, anatomically correct 3D model of the human hand is much more challenging to create than the simple geometry of the tracking wand; as a result, even ideal tracking conditions will not generate ideal results according to the metrics, since a "perfect fit" is impossible.

Bounding box and centroid tracking results (for the optimal configuration) shown in Fig. 5.10 are similar to the wand results, but with a slight divergence (approximately 15 pixels) between the ground truth and estimate around frame 160. This can be attributed to a change in the tracking target's thumb flexion, which the rigid hand model is not designed to handle. However, when movement is limited to the 6 DOFs being tracked, bounding box and centroid movement are accurately estimated.

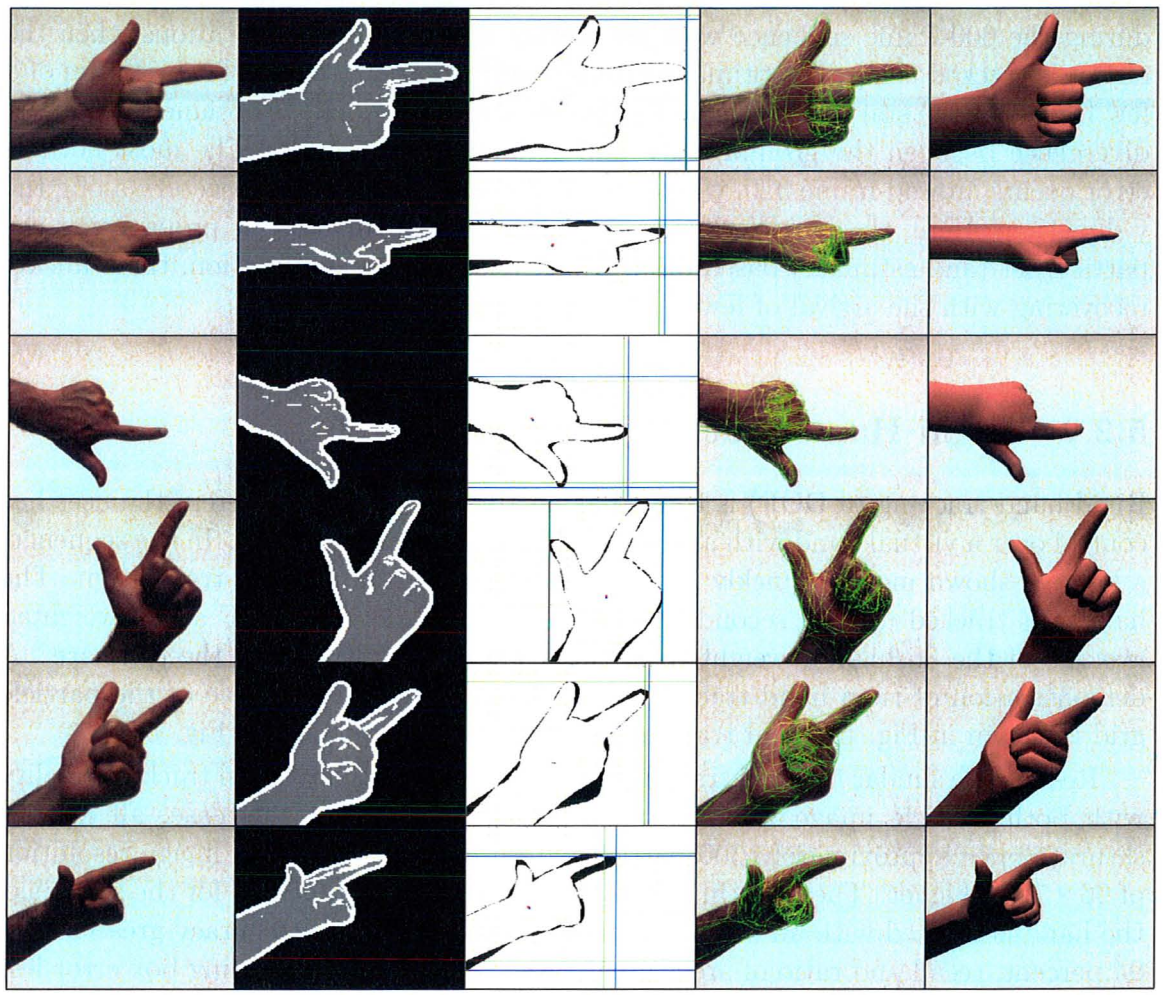

Figure 5.7: 6-DOF rigid hand tracking demonstration
From left to right: input video, feature map, error map (centroids, bounding boxes, and residual), wire frame overlay, estimated virtual hand pose

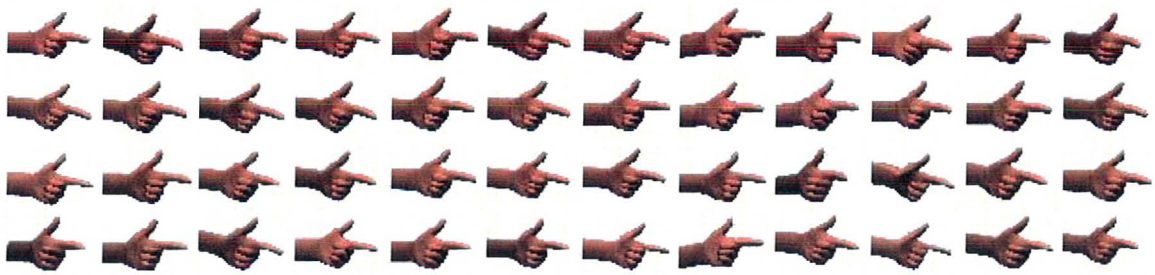

Figure 5.8: 4×12 sample of the rigid hand particle grid

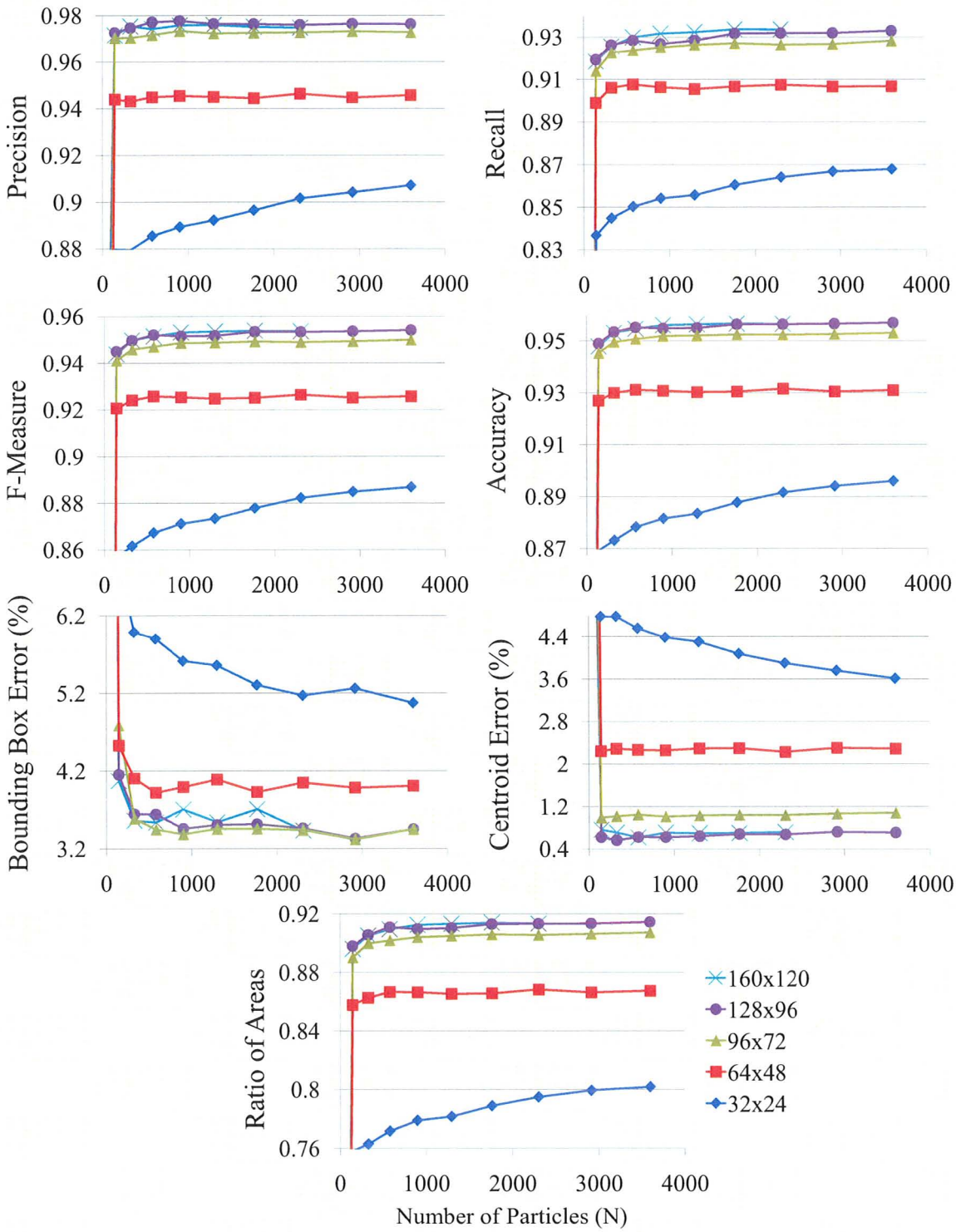

Figure 5.9: 6-DOF rigid hand tracking quality results

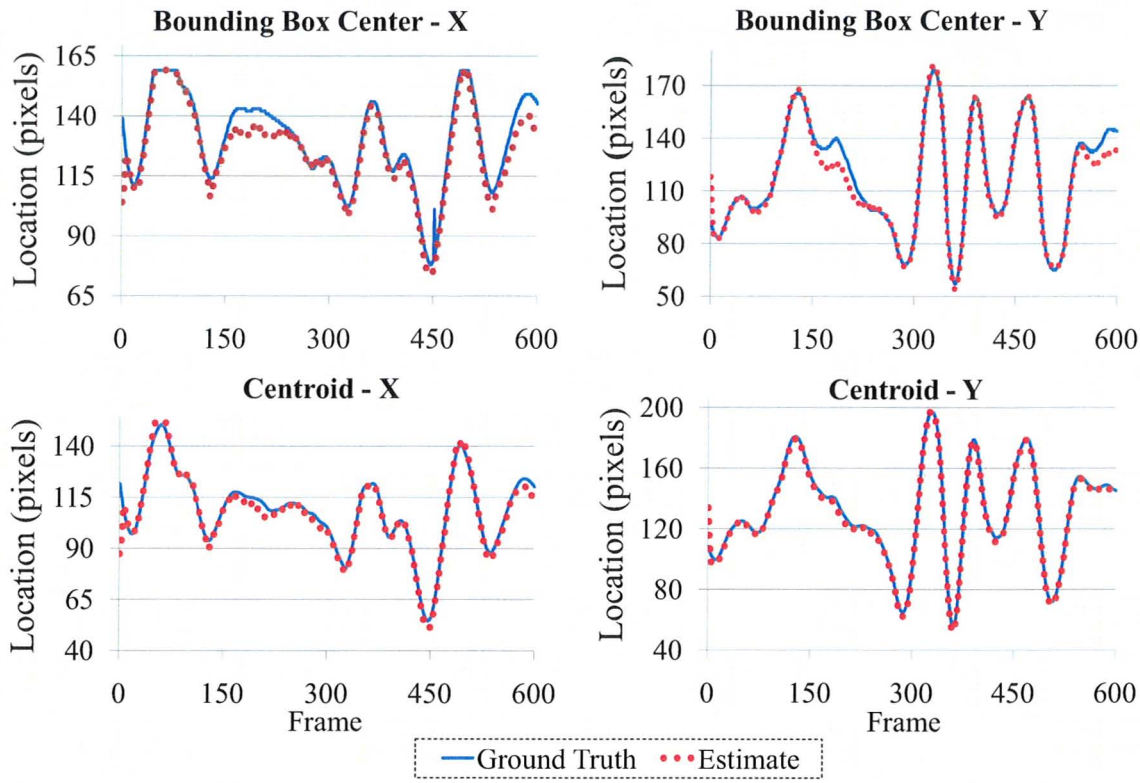

Figure 5.10: 6-DOF rigid hand tracking at optimal settings
324 particles, 96 × 72 particle image resolution

5.2.3 Articulated Hand Tracking - Real Video

Articulated bare hand tracking (10-DOF) was tested with six global DOFs, two DOFs in the wrist, and two DOFs in the index finger joints. The thumb is locked in a fully extended position and all other fingers are flexed against the palm (Fig. 5.1). This arrangement could be used to provide natural HCI in certain applications, such as a virtual point-and-click interface, and is demonstrated here by giving a user control over a virtual hand.

In the sequence, a hand is shown moving quickly through a variety of motions, including four wrist flexions, three full finger flexions, and a variety of translations and rotations, with several instances of both at once. The hand does not leave the frame and does not rotate in a manner that completely occludes the finger. The hand was tracked using a second-order motion model for the global parameters, a first-order motion model for the local parameters, and a weighted average of the 25 highest-weighted particles for the state estimate. A demonstration of 10-DOF hand tracking is shown in Fig. 5.11, a sample of the particle grid is shown in Fig. 5.12,

and tracking quality results are shown in Fig. 5.13.

Quantitative results are similar to the first two experiments, with all metrics scaling well with increasing particle count and resolution; however, when dealing with a higher-dimensional problem, more particles are required before tracking becomes robust and accurate. Specifically, a particle image resolution of 128×96 and a particle count of approximately 1,300 are needed to yield an average precision, F-measure, and accuracy of 95 percent, recall and ratio of areas greater than 90 percent, bounding box error less than 4 percent, and centroid error less than 1 percent. These settings are considered optimal for the 10-DOF tracking experiment.

Qualitatively, 10-DOF hand tracking performs well in the estimation of translation parameters, but rotation tracking is less accurate due to the ambiguity of projections. Although finger joint estimation is robust, it is not as responsive, occasionally demonstrating a small lag since the joints are at the end of the kinematic chain. These problems could potentially be remedied with a hierarchical particle filter or an alternative feature extractor that is more responsive to the slight change of the projected pixels caused by finger flexion. Nonetheless, the majority of hand poses are estimated accurately, giving a user the sensation of being in direct control of the virtual hand.

Fig. 5.14 shows the centroid and bounding box of the articulated hand being accurately tracked at the optimal settings. The most significant deviations occur in the x-direction of the bounding box around frames 280 and 350. These anomalies can be attributed to slight errors in finger joint estimation, as the size of the bounding box is a function of finger tip location. However, beyond these bounding box offsets, the optimal settings demonstrate effective tracking according to both metrics.

This task is inherently more challenging than the first two experiments for several reasons. Projections of a hand are not necessarily unique, leaving room for ambiguity; for example, if the edge of the screen intersects the hand at the wrist, it is not possible to know if a rotation is due to global rotation (i.e., moving the arm) or wrist flexion. Additionally, the hand can be rotated such that the index finger is completely occluded, meaning observations do not contribute to the estimation of its joint parameters. Furthermore, the pixel-by-pixel weight calculation algorithm provides optimal results when small movements produce significant changes in projected pixels, which is not always the case in hand tracking; for example, flexion in the PIP or DIP of the index finger displaces few pixels compared to flexion in the wrist. Finally, because the joints in the hand are arranged as a kinematic chain, an estimation error at the root of the chain can easily propagate and perturb other parameter estimates. It should also be noted that the complexities of a human hand are significantly more difficult to model than the simple wand described above, meaning even an ideal track will not lead to ideal results according to the metrics used.

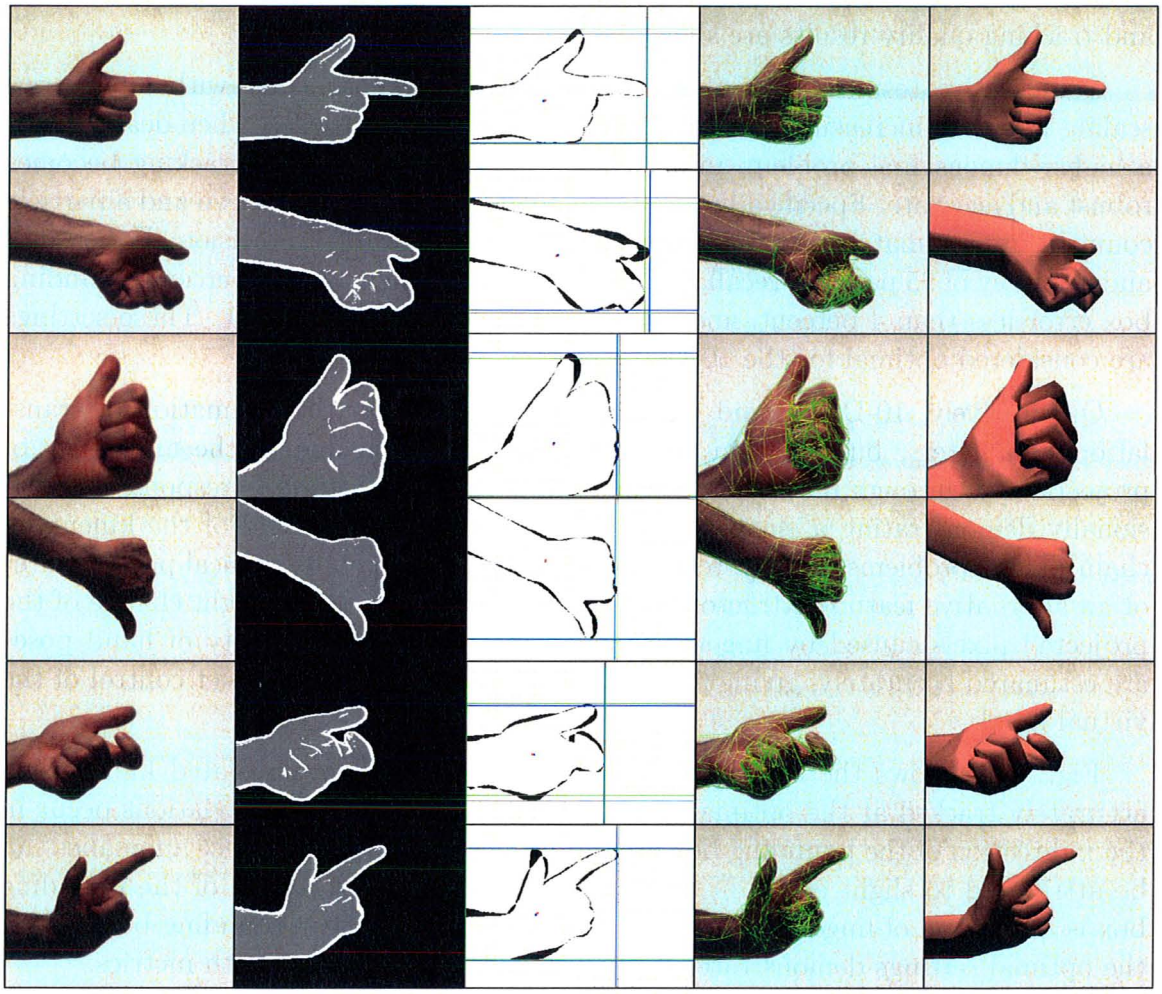

Figure 5.11: 10-DOF articulated hand tracking demonstration
From left to right: input video, feature map, error map (centroids, bounding boxes, and residual), wire frame overlay, estimated virtual hand pose

Figure 5.12: 4×12 sample of the articulated hand particle grid

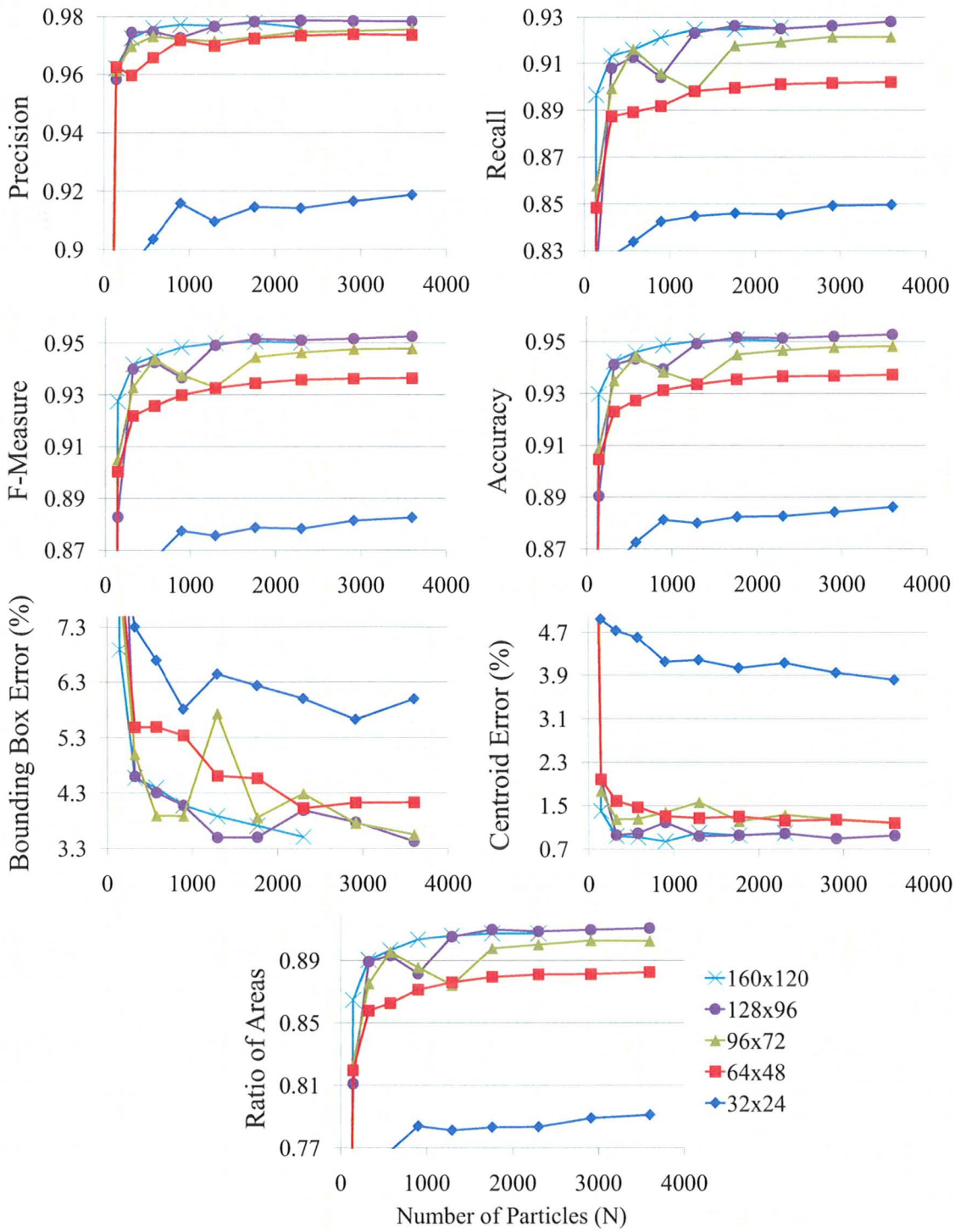

Figure 5.13: 10-DOF articulated hand tracking quality results

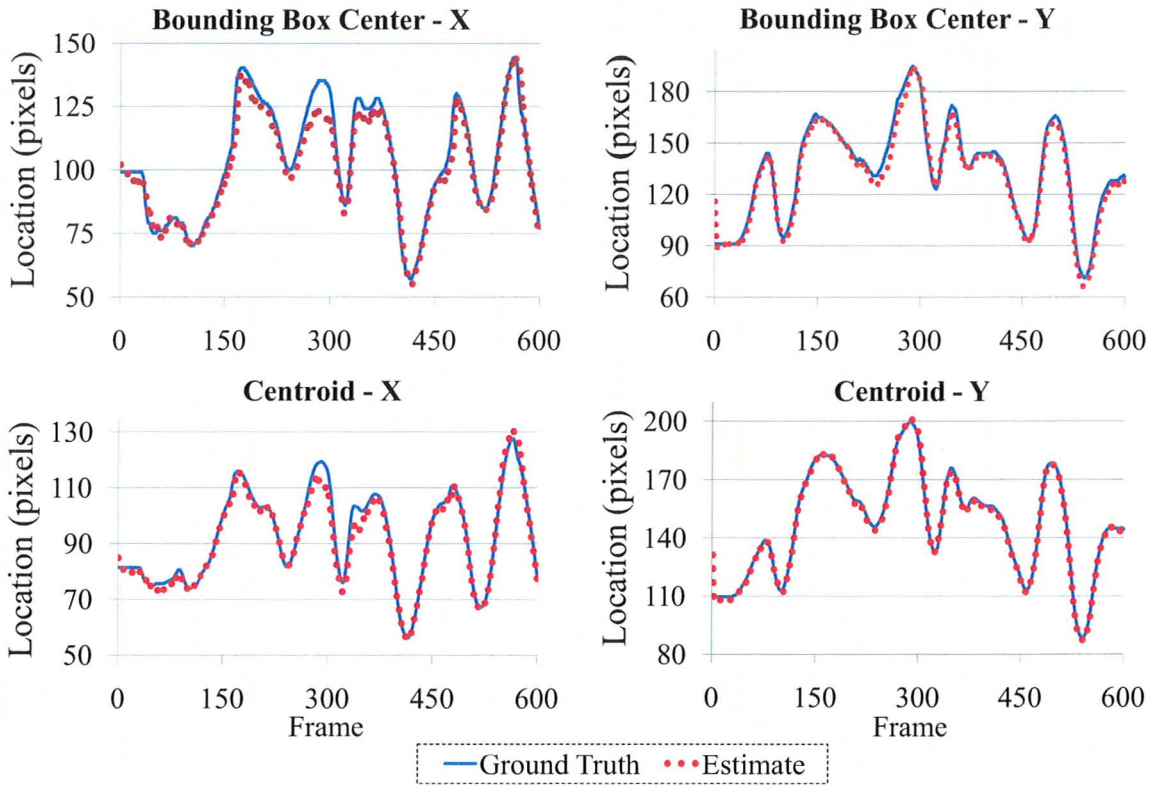

Figure 5.14: 10-DOF articulated hand tracking at optimal settings
1,296 particles, 128×96 particle image resolution

5.2.4 Rigid Wand Tracking - Synthetic Video

A synthetic rigid wand experiment was conducted to reinforce the results of Section 5.2.1 by quantifying tracking quality against a well-defined ground truth. In the first half of the 600-frame sequence, the 3D wand model is shown moving very quickly through each of its six DOFs individually. During the second half of the sequence, the wand is in constant motion in all DOFs simultaneously, with several instances of sharp velocity changes. Additionally, to make tracking more challenging, a small amount of random noise is introduced to each parameter. The wand was tracked using a second-order motion model on each DOF and a weighted average of the 36 highest-weighted particles used to compute the estimate. Though not theoretically necessary, silhouette and Sobel edge detection was used to make results comparable to real-video experiments. Fig. 5.15 show the quality of tracking in each DOF according to metric (5.1) and a brief tracking demonstration is shown in Fig. 5.16.

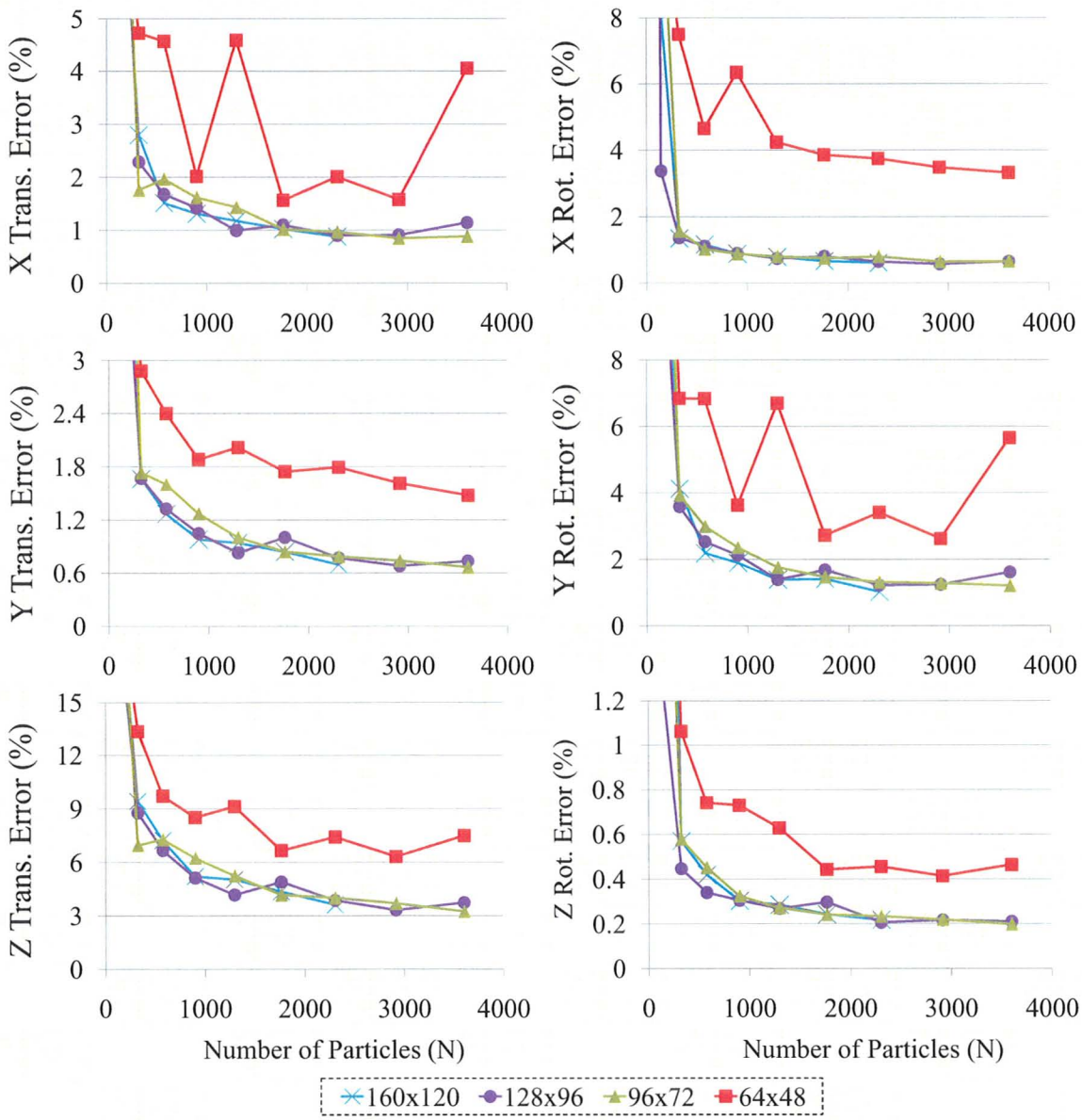

Figure 5.15: 6-DOF synthetic wand tracking quality results

Synthetic results agree with real video results, showing an increase in quality as particle count and particle image resolution increase. Percentage error in each degree of freedom converges at or below 1.5 percent, with the exception of translation in the z-direction, which converges to approximately 5 percent. This can be attributed to the fact that the z-direction is representative to the object's distance from the camera (i.e., depth) and is estimated solely based on the relative size of the object

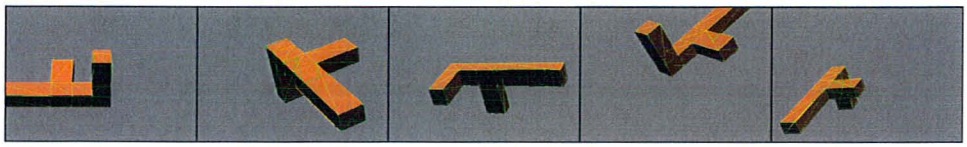

Figure 5.16: 6-DOF synthetic wand tracking demonstration
Estimate shown as wire frame overlay

in the frame. The depth of an object becomes increasingly difficult to estimate as it moves away from the camera and fewer pixels appear in its projection. Obtaining accurate depth information using a single camera is a widely studied task in computer vision, and an average error of 5 percent is acceptable in many applications.

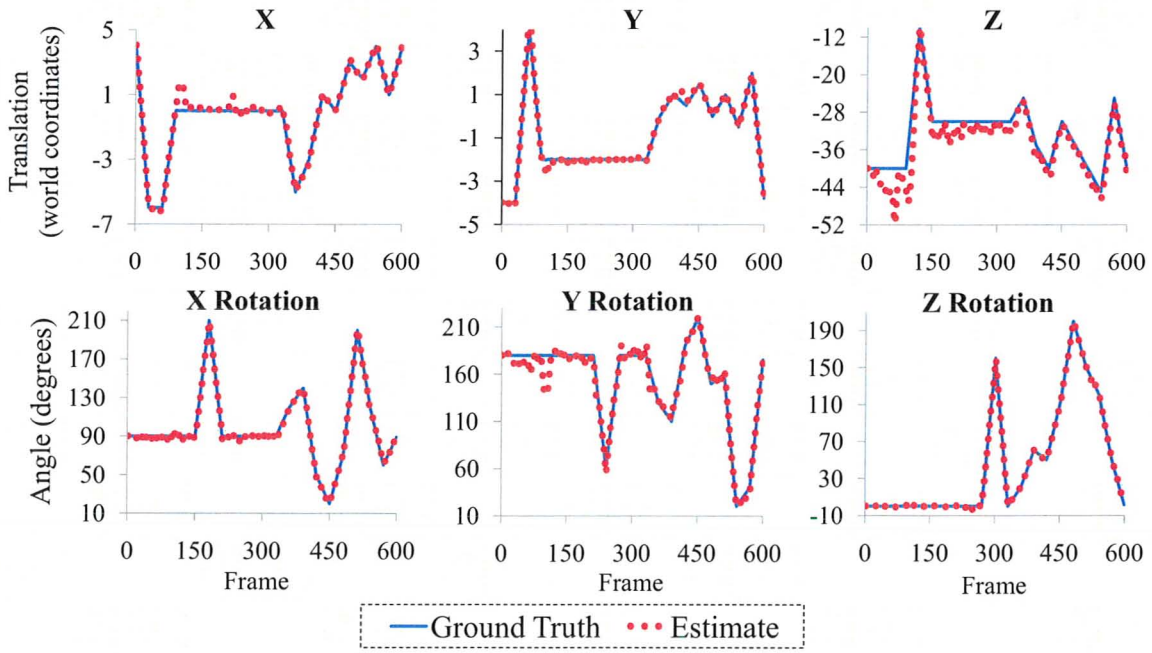

Figure 5.17: 6-DOF synthetic wand tracking at optimal setting
900 Particles, 96×72 resolution

Synthetic results demonstrate approximately 1,000 particles and a resolution of at least 96×72 are required before quality begins to saturate, indicating a slightly higher optimal configuration than the 600 particles identified in the real-video wand tracking experiment. At these settings, the wand is tracked with an average error of approximately 1.5 percent for x- and y- translation, 6 percent for z-translation, 1 percent for x-rotation, 2.5 percent for y-rotation, and 0.5 percent for z-rotation. Note that these results, based on a well-defined ground truth, are comparable to those

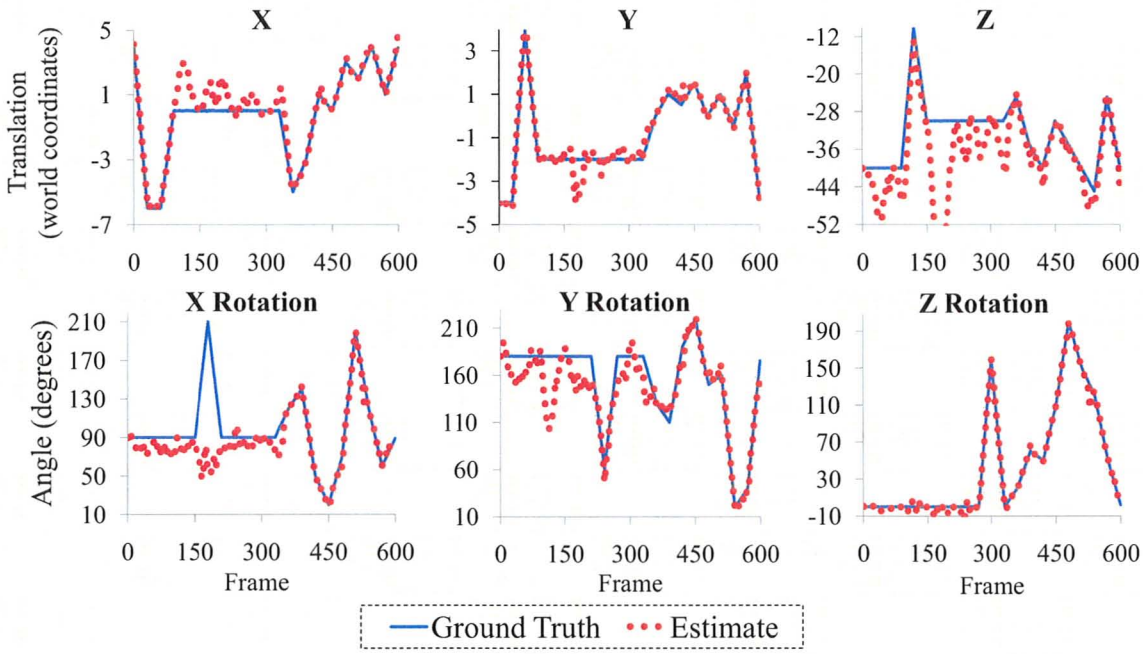

Figure 5.18: 6-DOF synthetic wand tracking (324 Particles, 64 × 48 resolution)

presented in Section 5.2.1, which are based on approximate metrics.

Fig. 5.17 shows the ground truth and estimate for each DOF being tracked at the optimal configuration. Note that rotation parameters are expressed in degrees and translation parameters are expressed in Direct3D's native world coordinate system. Tracking proves to be highly accurate, with the exception of the z-translation parameter, for reasons described above. The minor anomaly that occurs in the x- and y-translation parameters around frame 100 is also of interest; as both parameters come to an abrupt stop from a sustained period of high velocity, the system dynamics model predicts continued motion and "overshoots" the estimate, but quickly recovers within a few frames. For the purposes of comparison, Fig. 5.18 and Fig. 5.19 show the same sequence being tracked using very low (324 particles, 64 × 48 particle image resolution) and very high (2,304 particles, 160 × 120 particle image resolution) settings, respectively. Note the variations in quality, particularly in the z-translation parameter.

5.2.5 Articulated Hand Tracking - Synthetic Video

A synthetic video sequence showing the 3D hand model moving with eight DOFs (six global and two index finger joints) was used to evaluate tracking quality with reference to a known ground truth. The sequence is challenging, with nearly constant motion

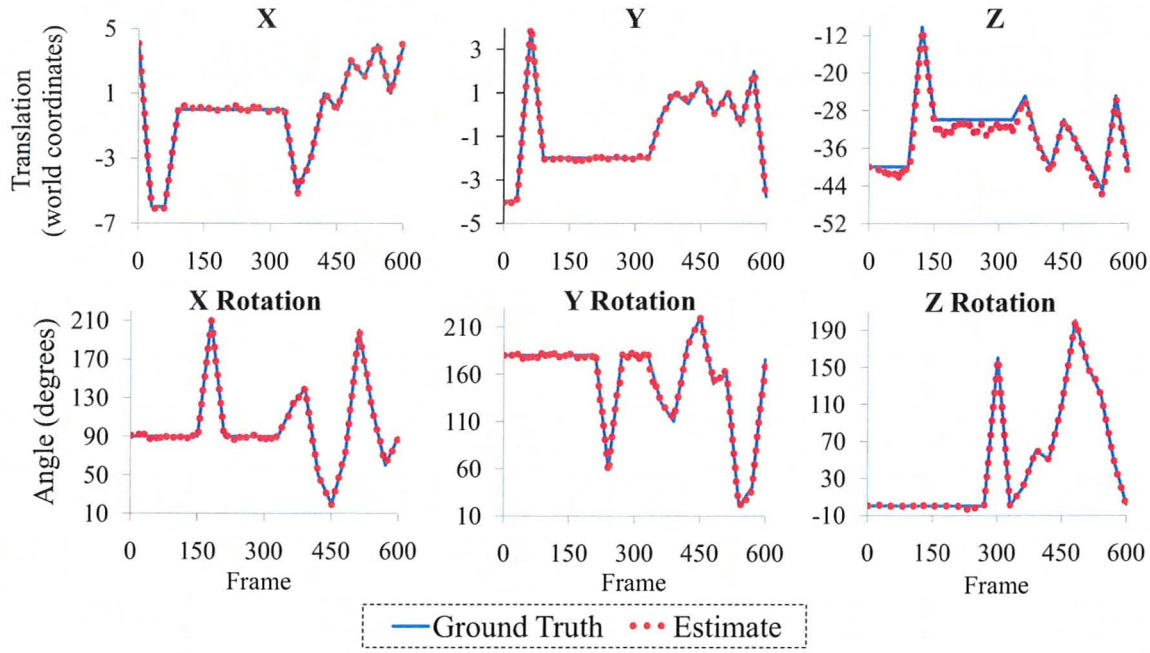

Figure 5.19: 6-DOF synthetic wand tracking (2,304 Particles, 160 × 120 resolution)

in multiple DOFs simultaneously, slight random noise added to each DOF, sharp changes in velocity, and some partial occlusion of the index finger. Although synthetic tracking benefits from the 3D model perfectly matching the shape, color, and shading of the tracking target, Sobel edge detection was still used and all other articulated tracking challenges discussed in Section 5.2.3 remain. The hand was tracked using a second-order motion model for the global parameters, a first-order motion model for the local parameters, and a weighted average of the 36 highest-weighted particles for the state estimate. Fig. 5.20 presents a demonstration of synthetic hand tracking and the MAE for translation, rotation, and joint flexion is shown in Fig. 5.21.

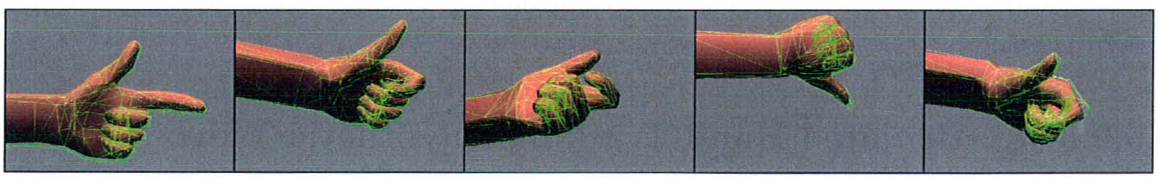

Figure 5.20: 8-DOF synthetic articulated hand tracking demonstration
Estimate shown as wire frame overlay

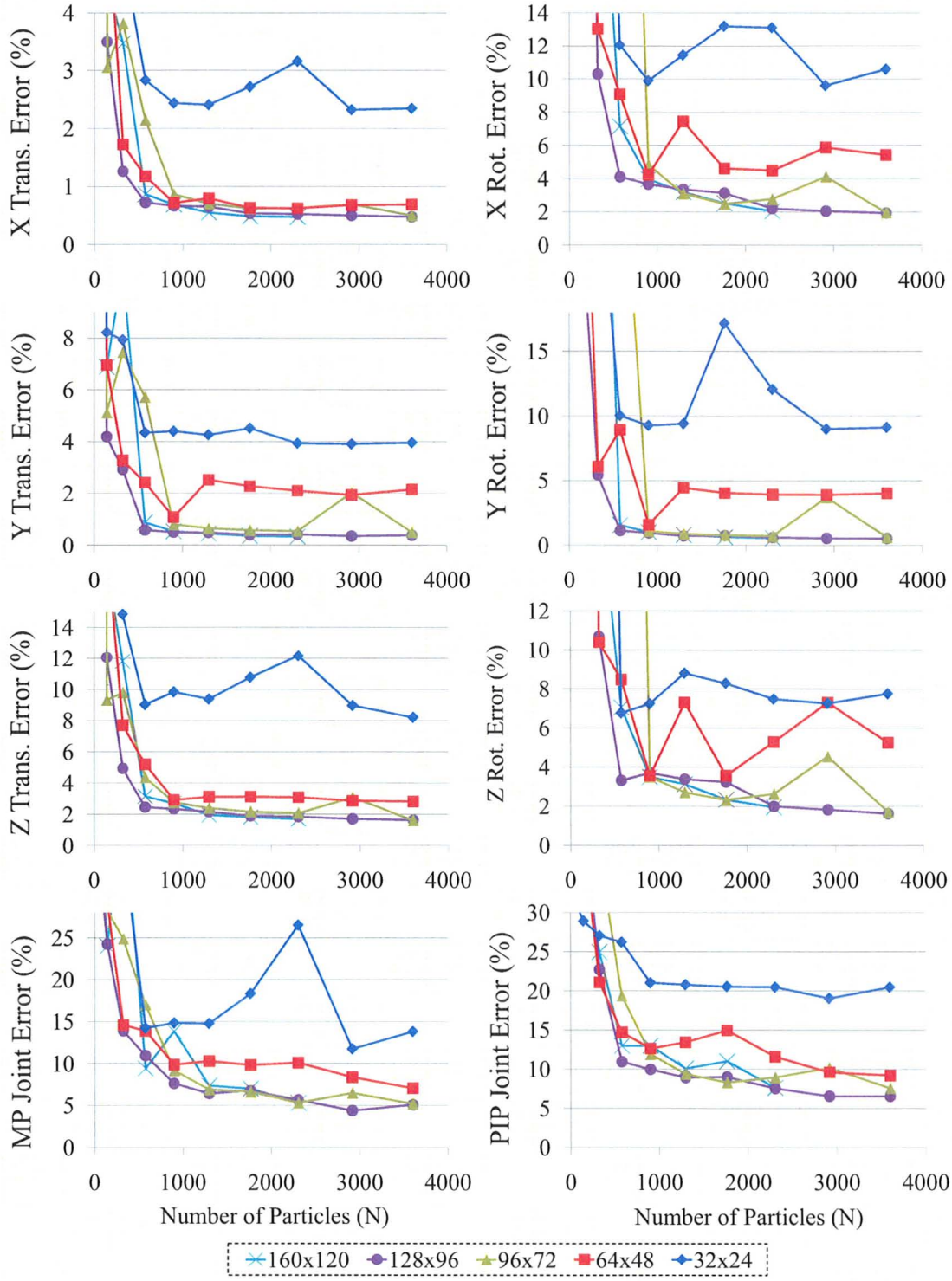

Figure 5.21: 8-DOF synthetic articulated hand tracking quality results

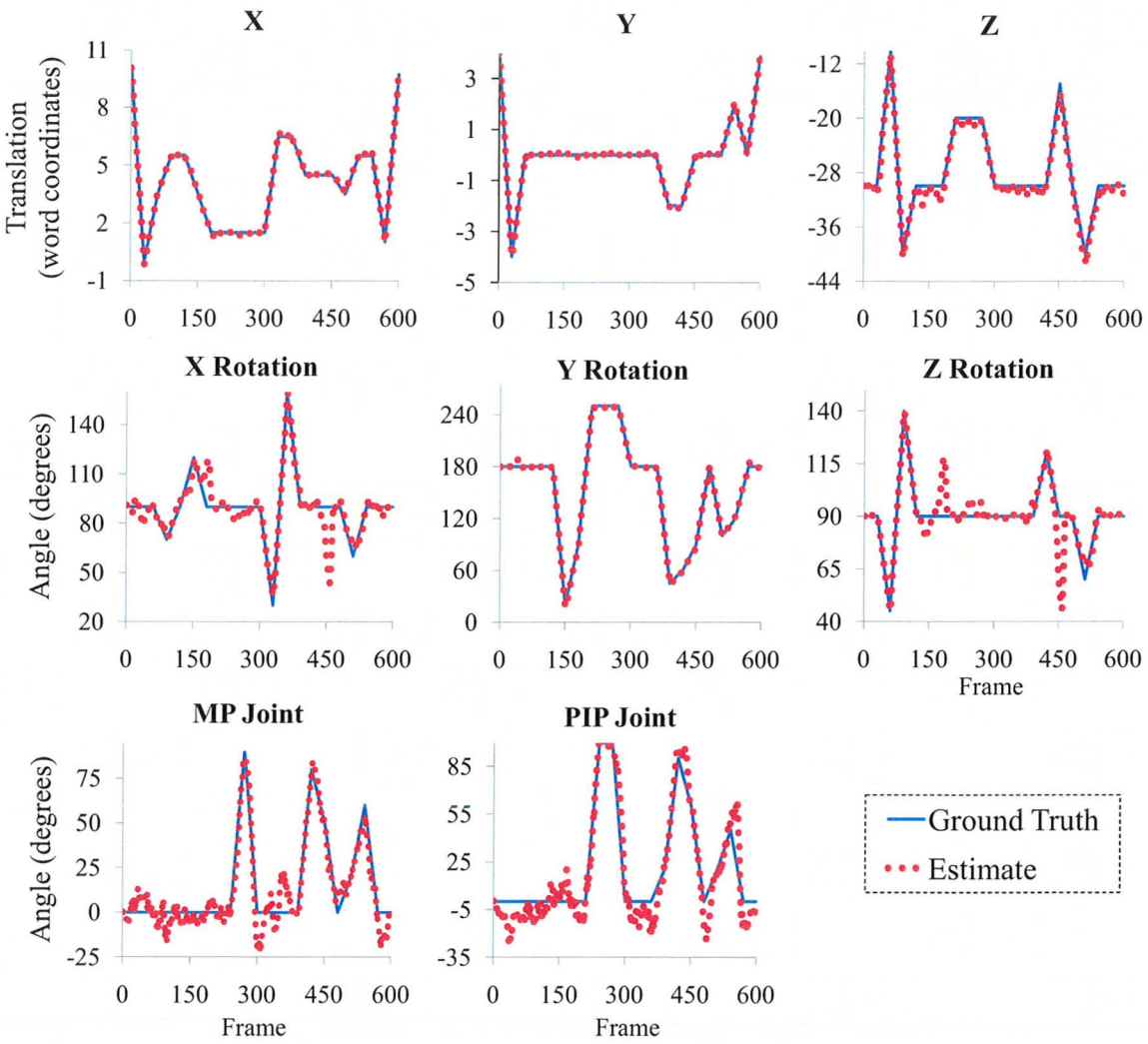

Figure 5.22: 8-DOF synthetic articulated hand tracking at optimal settings
1,296 Particles, 128×96 resolution

Synthetic tracking quality results agree with the real-video results, indicating optimal tracking with a particle image resolution of 128×96 and particle count of approximately 1,200. At these settings, the hand is tracked with average translation error of 1.1 percent, rotation error of 2.5 percent (approximately 3 degrees), and joint error of 7.7 percent (approximately 7 degrees). Fig. 5.22 shows the ground truth and estimated value of each DOF at the optimal tracking configuration.

Higher errors in joint tracking are expected because large changes in finger joint angles displace few pixels in the projected image relative to changes in global rotation or translation. Fig. 5.22 demonstrates that the PIP joint was tracked least-accurately,

as would be expected as it is at the end of the model's kinematic chain. This can be seen explicitly around frame 350, where an overestimation of the MP joint angle propagates down the kinematic chain and causes an underestimation of the PIP joint angle to compensate. Around frame 450, tracking of the x-rotation and z-rotation parameters momentarily deviates from the ground truth (due to simultaneous changes of velocity in seven DOFs); however, the robustness of the particle filter can be seen as it recovers within 20 to 30 frames.

5.3 Performance Results

This section discusses the performance of the 3D model-based tracking framework in terms of its frame rate and speedup when compared to an equivalent CPU-based implementation (using the same test bench). It should be noted that the CPU implementation still uses the GPU to render and tile the particle images, as this is not a reasonable task for the CPU to execute. All performance results describe tracking with live video, which takes slightly longer than prerecorded sequences that do not require frame acquisition. Although the difference in performance is minor, results are presented for both wand and hand tracking experiments (rendering time is longer for the 3D hand model as it has more polygons).

5.3.1 Wand Tracking

The frame rates at which the wand can be tracked for a variety of particle counts and particle image resolutions are shown in Fig. 5.23 for both the GPU- and CPU-based implementations. The GPU-accelerated performance exceeds CPU performance in all tests, demonstrating a gradual decline in performance with increasing particle count, whereas the CPU implementation's performance quickly drops to under 10 fps for more than a few hundred particles. This is especially true for particle image resolutions of 96×72 or higher, which are of particular interest according to the quality results presented above.

Fig. 5.24 shows the speedup of particle evaluation (i.e., the target of GPU-acceleration in this framework), as well as overall speedup of the system over the CPU-based implementation. Particle evaluation speedup varies between 2.5 times and 25 times depending on particle count and particle image resolution, with speedups of at least 20 times for configurations of interest. Overall speedup results range between 1.2 times and 15 times, with speedups of 5 to 10 times for configurations of interest.

Table 5.2 and Table 5.3 contain a detailed breakdown of the time taken by each step of the tracking system on both the CPU and GPU for the optimal system configurations identified in real (Section 5.2.1) and synthetic (Section 5.2.4) wand tracking experiments, respectively. Note that the *render particle images* stage is executed on

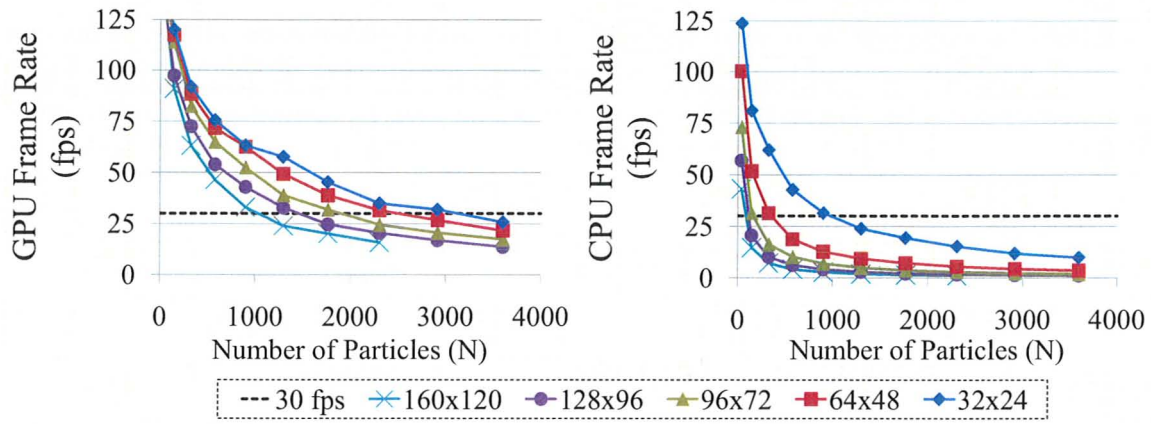

Figure 5.23: Wand tracking performance results

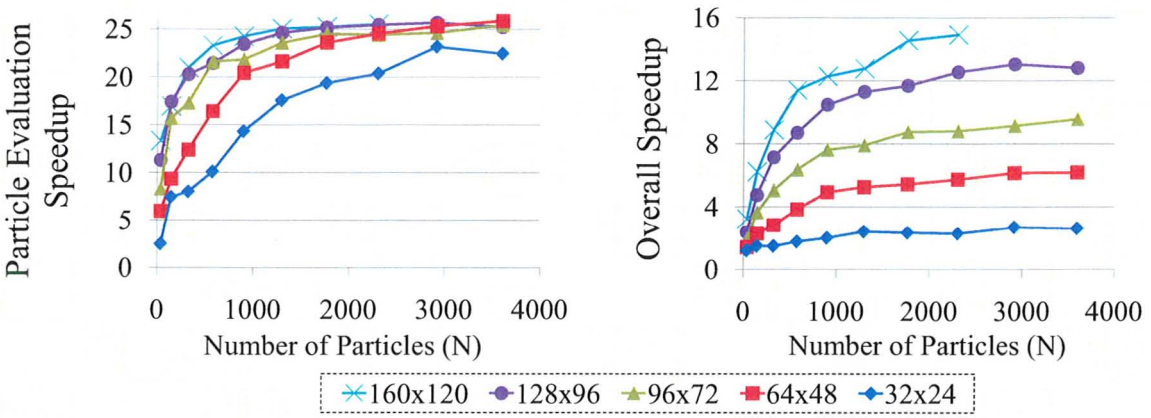

Figure 5.24: Wand tracking speedup results

the GPU in both implementations, and the *acquire frame, particle filter* and *other* stages are executed on the CPU in both implementations. According to both tables, GPU-accelerated performance significantly exceeds CPU-only performance, speeding up particle evaluation by a factor of approximately 21 times. Using 30 fps as a benchmark for real-time performance, both real and synthetic wand tracking tasks prove to be possible in real time using GPU acceleration, while neither are realistic on the CPU implementation.

The results in these tables can also be used to demonstrate *Amdahl's law* (Amdahl, 1967), which provides a relationship between the speedup of a parallelizable portion of a program and the overall application speedup:

$$TotalSpeedup = \frac{1}{(1-p) + p/s} \tag{5.11}$$

Table 5.2: Wand tracking performance analysis (576 particles, 96 × 72)

Step	GPU (ms)	CPU (ms)	Speedup
Render Particle Images	2.49	2.38	0.96x
Acquire Frame	4.92	4.61	0.94x
Map Resources	1.87	2.55	1.36x
Feat. Det. & Weight Calc.	**3.99**	**86.24**	**21.59x**
Particle Filter	0.59	0.58	0.99x
Other	1.53	1.74	1.14x
Total	15.40 (64.9 fps)	98.12 (10.2 fps)	6.37x

Table 5.3: Wand tracking performance analysis (900 particles, 96 × 72)

Step	GPU (ms)	CPU (ms)	Speedup
Render Particle Images	3.42	3.56	1.04x
Acquire Frame	4.68	4.71	1.01x
Map Resources	2.64	2.47	0.94x
Feat. Det. & Weight Calc.	**6.06**	**132.17**	**21.82x**
Particle Filter	0.84	0.85	1.01x
Other	1.44	1.80	1.25x
Total	19.08 (52.4 fps)	145.56 (6.9 fps)	7.63x

where p is the percentage of an algorithm's execution time that can be parallelized and s is the factor by which this portion can be sped up. For example, in Table 5.2, the feature detection and weight calculation stage is the target of parallelization. This step takes 86.24 ms of the 98.12 ms total taken for one iteration of the tracking algorithm, meaning $p = 86.24/98.12 = 0.88$. Setting $s = 21.59$ (the factor by which feature extraction and weight calculation can be sped up through parallelization), Amdahl's law yields a theoretical overall speedup of 6.18 times, which is similar to the 6.37 times observed. A similar relationship can be observed in all other results.

Amdahl's law explains the appearance of the overall speedup graph in Fig. 5.24, where increasing particle image resolutions yields significantly greater speedups at high particle counts, despite the particle evaluation stage being sped up by factor of approximately 25 at any resolution. Higher particle image resolutions increase the amount of time taken by feature extraction and weight calculation, meaning this step accounts for a larger percentage of the overall algorithm. This increases p in Amdahl's law, consequently increasing the total speedup.

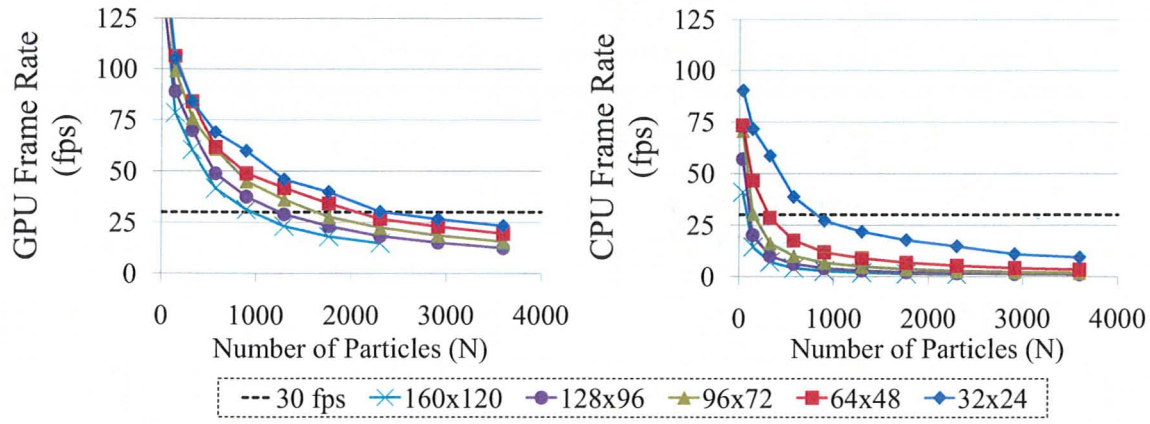

Figure 5.25: Hand tracking performance results

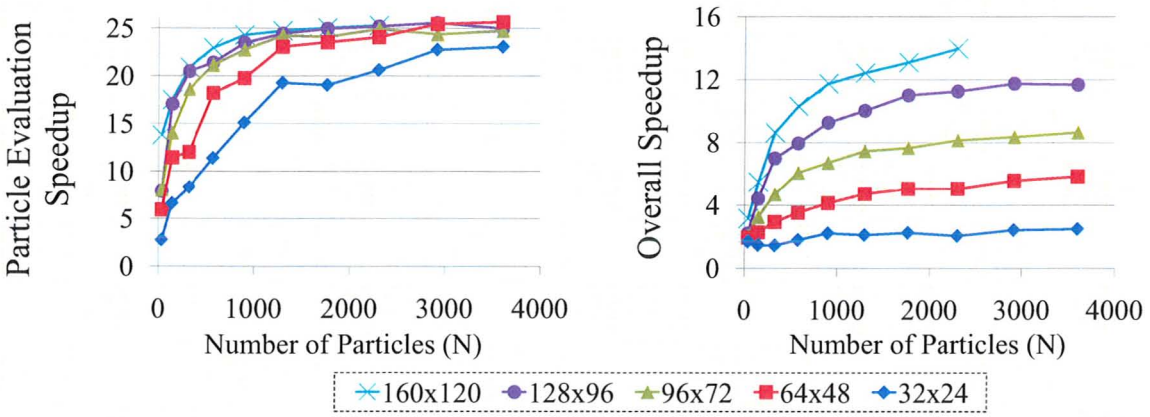

Figure 5.26: Hand tracking speedup results

5.3.2 Hand Tracking

Hand tracking performance results are essentially identical to wand tracking; however, since they demonstrate slightly slower frame rates as the higher polygon count of the hand model increases rendering time, they are included for completeness. Fig. 5.25 shows GPU and CPU frame rate results, Fig. 5.26 shows speedup results, and Tables 5.4 and 5.5 present detailed performance information for the optimal system settings identified for rigid (Section 5.2.2) and articulated (Section 5.2.3) hand tracking experiments, respectively. The analysis of this data is effectively identical to that presented for wand tracking and will not be reiterated here.

Table 5.4: Hand tracking performance analysis (324 particles, 96 × 72)

Step	GPU (ms)	CPU (ms)	Speedup
Render Particle Images	3.53	3.00	0.85x
Acquire Frame	2.52	2.72	1.08
Map Resources	1.89	2.00	1.06x
Feat. Det. & Weight Calc.	**2.80**	**51.96**	**18.58x**
Particle Filter	0.71	0.55	0.78x
Other	1.72	1.74	1.01x
Total	13.16 (76.0 fps)	61.98 (16.1 fps)	4.71x

Table 5.5: Hand tracking performance analysis (1,296 particles, 128 × 96)

Step	GPU (ms)	CPU (ms)	Speedup
Render Particle Images	10.31	9.89	0.96x
Acquire Frame	5.50	5.16	0.94x
Map Resources	2.04	2.35	1.15x
Feat. Det. & Weight Calc.	**13.36**	**339.93**	**25.45x**
Particle Filter	1.93	1.89	0.98x
Other	1.49	1.80	1.21x
Total	34.62 (28.9 fps)	361.03 (2.8 fps)	10.43x

5.4 Summary

The performance and quality results of the three real-video experiments are integrated and presented in Fig. 5.27, directly showing the trade-off between frame rate and quality. Here, quality is measured as an average of F-measure (5.4), accuracy (5.5), bounding box error (5.6), centroid error (5.7), and ratio of areas (5.8), where bounding box and centroid error have been reformulated to be compatible with the other metrics (i.e., expressed on a scale where zero is low quality and one is high quality). In nearly all cases, a particle image resolution of 128 × 96 provides the optimal quality at a given frame rate, followed by a resolution of 160 × 120 and 96 × 72. These plots also demonstrate the computational demands of tracking in more than six DOFs, as the quality of the two rigid tracking experiments saturates around 70 fps, whereas saturation occurs at 30 fps for the articulated hand tracking experiment.

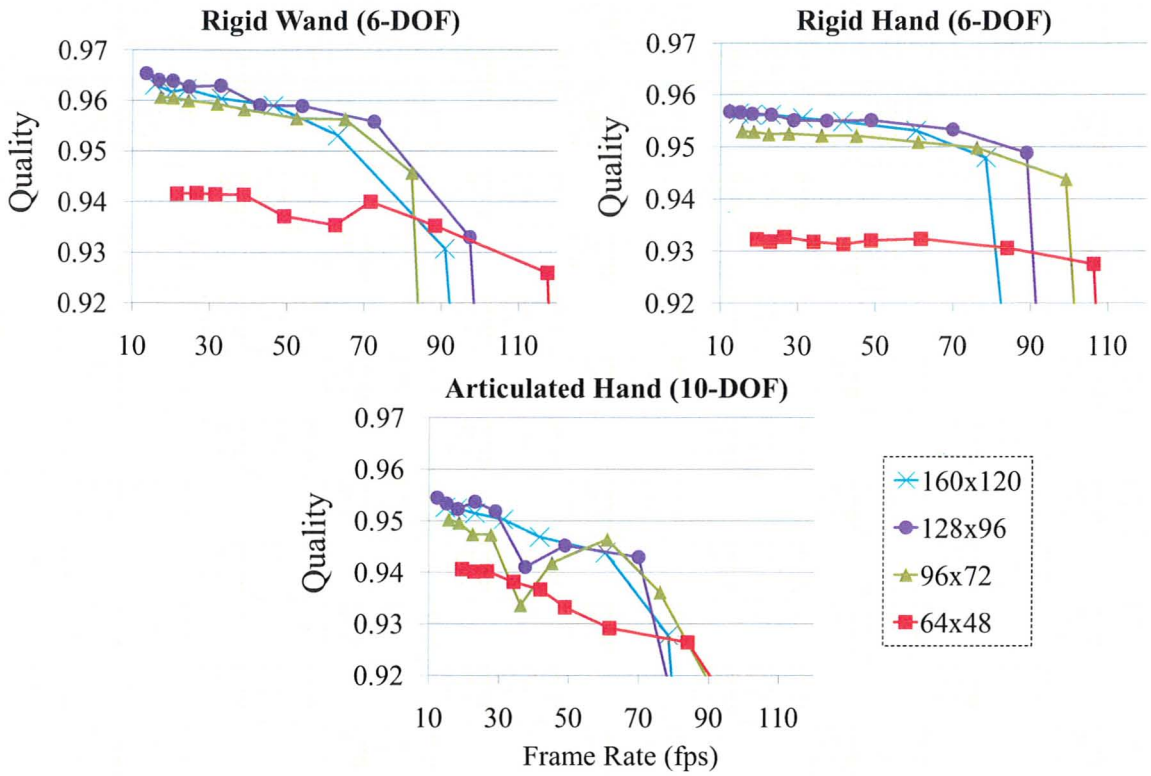

Figure 5.27: Summary of real-video experiments

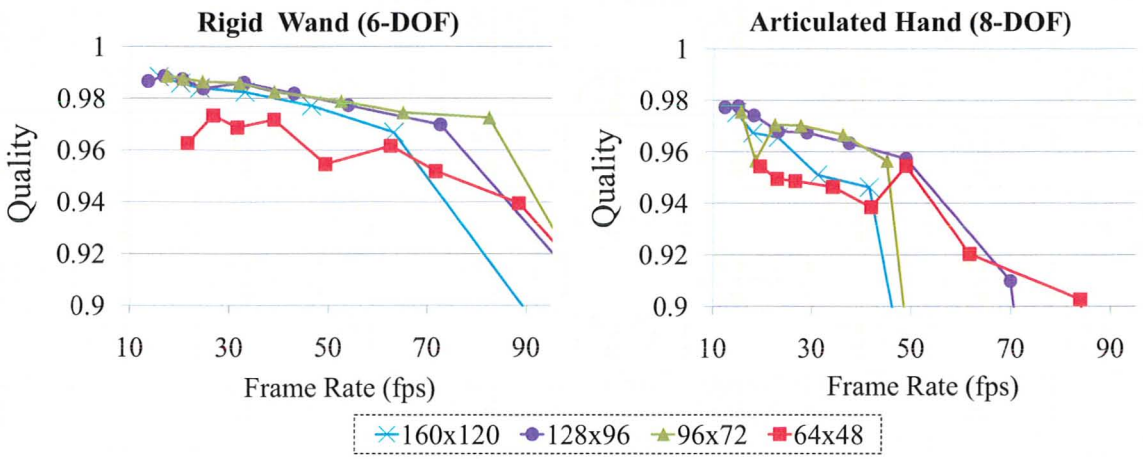

Figure 5.28: Summary of synthetic video experiments

An integrated summary of tracking quality versus frame rate for the two synthetic experiments is provided in Fig. 5.28. Here, quality is defined as the average of

the MAE values measured for each DOF. Results are similar to the real tracking experiments, showing quality saturating around 70 fps for rigid object tracking and approximately 30 fps for articulated object tracking; however, unlike the real-video experiments, the synthetic tests often favour a particle image resolution of 96×72 as the setting for optimal quality at a given frame rate.

Limitations on the performance of the GPU implementation are primarily consequences of warp divergence necessitated by aspects of the edge detection algorithm, such as the padding of memory structures prior to filter convolution. Additionally, summing all pixel weights to produce a single particle weight is inherently difficult to parallelize. Nonetheless, the GPU is exploited in an effective manner: using the settings of Table 5.5, the GPU generates, processes and reduces approximately 1.3 Gbyte of pixel data to 146 Kbyte of particle weight data each second. The primary drawback of most GPU applications are the excessive memory transfer latencies; however, these are largely avoided in this framework as the GPU is both the source and consumer of image data, requiring only particle weights be transferred to the CPU.

Chapter 6

Conclusion

This thesis presented a novel framework for high-speed 3D model-based visual tracking using a GPU-accelerated particle filter. Specifically, markerless 6+ DOF tracking in monocular video was framed as a Bayesian state estimation problem, facilitating the application of Bayesian filters, such as the particle filter. The particle filter's computationally demanding weight update stage, which consists of 3D model simulation and evaluation in this context, was efficiently mapped to a GPU's massively parallel SMs, where it is executed by thousands of concurrent threads.

The GPU implementation efficiently uses Direct3D and NVIDIA CUDA to exploit a variety of GPU features, such as thread cooperation through shared memory and barrier synchronization, CUDA/Direct3D interoperability, and cached memory accesses using constant and texture memory. Furthermore, the tracking algorithm is effectively partitioned to maximize GPU occupancy while also ensuring there is enough arithmetic intensity within each thread. The performance of most GPU-based applications is limited by excessive memory transfer and access latencies. These are largely avoided in this work, as both particle simulation and evaluation execute on the GPU, making it the source and consumer of image data. For each 45 Mbyte particle grid that is generated and processed by the GPU with each iteration of the tracking algorithm, only 5 Kbyte is transferred to the CPU through high-bandwidth, page-locked memory.

Five separate tracking experiments were conducted, demonstrating tracking with 6, 8, and 10 DOFs in real or synthetic video, using either a fabricated rigid wand or an articulated hand as a tracking target. Experimental results indicate tracking quality up to 96 percent in real video tests and up to 98 percent in synthetic tests. Performance results demonstrate accurate and robust tracking with speeds up to 60 fps for rigid object tracking tasks and 29 fps for articulated tracking. Both quality and performance are dependent on the number of particles being simulated and evaluated, as well as the resolution of the particle images. Rigid object tracking demands

anywhere between 300 and 1,000 particles and a particle image resolution of 96×72 depending on the tracking task, whereas articulated hand tracking demonstrated a need for a minimum of 1,000 particles and a particle image resolution of 128×96.

The GPU-accelerated framework outperformed a similar CPU-based implementation in all tests. Particle evaluation (the target of GPU-acceleration) executed between 2.5 and 25 times faster than the CPU, resulting in overall algorithm speedups between 1.2 times and 16 times, depending on particle count and particle image resolution. Using 30 fps as an approximate benchmark for real time performance, the GPU-accelerated system successfully tracked objects in real time for all five experiments, whereas the CPU implementation operated between 3 and 16 fps, unable to achieve real-time performance. These results indicate that modern consumer-level GPUs are a suitable platform for 30+ fps 3D-model-based visual tracking using a particle filter.

The framework was presented using the SIR particle filter for state estimation; however, tracking quality could presumably be improved by introducing one of the many proposed variations of the particle filter, such as a hierarchical particle filter, auxiliary particle filter, or appearance-augmented filter. Nonetheless, the goal of this work was not to modify the traditional particle filter, but rather to demonstrate that this approach, which is often regarded as too computationally intensive for real-time model-based tracking, is a viable option with GPU acceleration. Furthermore, the modular natural of the framework makes it trivial to replace the SIR filter with a more appropriate simulation-based estimation technique for a particular application.

This adaptability extends to the choice of an alternative feature extractor. While silhouette and Sobel edge detection were used as examples here, as long as the selection is parallelizable and does not demand excessive conditional logic that would lead to warp divergence in CUDA kernels, any application-specific technique could be utilized instead.

Using a rigid wand and bare hand tracking targets, AR and HCI using a were presented as potential applications; however, the framework could be customized to support a variety of tracking tasks. Because the choice of tracking target and number of DOFs is variable, applications such as visual servoing, performance-driven animation, and medical imaging could also realize significant speedup through GPU-acceleration of 3D model-based tracking approaches. With the recent release of NVIDIA's Fermi architecture, designed primarily with the GPU-computing community in mind, the utility of GPU-acceleration in high-performance computing tasks, such as model-based tracking and countless other applications, will certainly continue to grow.

Appendix A

SIS Particle Filter Derivation

This appendix presents a derivation of the recursive weight-update equation

$$w_t^i \propto w_{t-1}^i \frac{p(\mathbf{z}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{z}_t)} \tag{A.1}$$

that is fundamental to the operation of the SIS particle filter.

To begin, recall the goal of the SIS particle filter is to recursively approximate the posterior PDF $p(\mathbf{x}_t|\mathbf{z}_{1:t})$ of a dynamic state space model using a series of point masses (particles) $\{\mathbf{x}_t^i\}_{i=1}^N$, drawn from an importance (or proposal) PDF $q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{z}_t)$, and their corresponding normalized weights $\{w_t^i\}_{i=1}^N$ according to

$$p(\mathbf{x}_t|\mathbf{z}_{1:t}) \approx \sum_{i=1}^{N} w_t^i \delta(\mathbf{x}_t - \mathbf{x}_t^i), \tag{A.2}$$

where w_t^i is defined by (A.1). To derive this recursive relationship, first consider the sequential (i.e., non-recursive) approximation:

$$p(\mathbf{x}_{0:t}|\mathbf{z}_{1:t}) \approx \sum_{i=1}^{N} w_t^i \delta(\mathbf{x}_{0:t} - \mathbf{x}_{0:t}^i), \tag{A.3}$$

which considers the entire path of $\mathbf{x}_t$ since $t = 0$. The normalized weight of a particle w_t^i can then be defined as

$$w_t^i \propto \frac{p(\mathbf{x}_{0:t}^i|\mathbf{z}_{1:t})}{q(\mathbf{x}_{0:t}^i|\mathbf{z}_{1:t})} \tag{A.4}$$

where $p(\cdot)$ is a posterior PDF that can be evaluated up to proportionality but not sampled from, and $q(\cdot)$ is an importance PDF that samples can be easily drawn from.

To draw a sample from $q(\mathbf{x}_{0:t}^i|\mathbf{z}_{1:t})$ in its general form would require N samples of

$x_{0:t}$, meaning the computational complexity of the task would increase with time. To circumvent this, $q(\cdot)$ is chosen specifically to be of the form

$$q(\mathbf{x}_{0:t}|\mathbf{z}_{1:t}) = q(\mathbf{x}_t|\mathbf{x}_{0:t-1}, \mathbf{z}_{1:t})q(\mathbf{x}_{0:t-1}|\mathbf{z}_{1:t-1}). \tag{A.5}$$

This formulation allows N particles to be drawn according to $\mathbf{x}_t^i \sim q(\mathbf{x}_t|\mathbf{x}_{0:t-1}, \mathbf{z}_{1:t}), i = 1, ..., N$ and used to augment existing samples $\{\mathbf{x}_{0:t-1}^i\}_i^N$ for each new observation $\mathbf{z}_t$. In other words, as long as $q(\cdot)$ can be factored as shown in (A.5), the computational complexity of sampling particles does not increase over time; instead, new observations are used to recursively update previous samples.

The final stage therefore involves the derivation of a recursive weight-update relationship. By expressing $p(\mathbf{x}_{0:t}|\mathbf{z}_{1:t})$ as

$$p(\mathbf{x}_{0:t}|\mathbf{z}_{1:t}) = \frac{p(\mathbf{z}_t|\mathbf{x}_{0:t}, \mathbf{z}_{1:t-1})p(\mathbf{x}_t|\mathbf{x}_{0:t-1}, \mathbf{z}_{1:t-1})p(\mathbf{x}_{0:t-1}|\mathbf{z}_{1:t-1})}{p(\mathbf{z}_t|\mathbf{z}_{1:t-1})}, \tag{A.6}$$

and substituting (A.5) and (A.6) into (A.1), w_t^i can be expressed as:

$$w_t^i = \frac{p(\mathbf{z}_t|\mathbf{x}_{0:t}^i, \mathbf{z}_{1:t-1})p(\mathbf{x}_t^i|\mathbf{x}_{0:t-1}^i, \mathbf{z}_{1:t-1})}{p(\mathbf{z}_t|\mathbf{z}_{1:t-1})q(\mathbf{x}_t^i|\mathbf{x}_{0:t-1}^i, \mathbf{z}_{1:t})} \times \underbrace{\frac{p(\mathbf{x}_{0:t-1}^i|\mathbf{z}_{1:t-1})}{q(\mathbf{x}_{0:t-1}^i|\mathbf{z}_{1:t-1})}}_{w_{t-1}^i}. \tag{A.7}$$

Rearranging and simplifying (A.7) yields

$$w_t^i = w_{t-1}^i \frac{p(\mathbf{z}_t|\mathbf{x}_{0:t}^i, \mathbf{z}_{1:t-1})p(\mathbf{x}_t^i|\mathbf{x}_{0:t-1}^i, \mathbf{z}_{1:t-1})}{p(\mathbf{z}_t|\mathbf{z}_{1:t-1})q(\mathbf{x}_t^i|\mathbf{x}_{0:t-1}^i, \mathbf{z}_{1:t})} \tag{A.8}$$

which gives a recursive relationship between a particle's weight at time t and time $t-1$. Because weights are normalized, the factor $p(\mathbf{z}_t|\mathbf{z}_{1:t-1})$ can be considered an irrelevant constant, and (A.8) simplifies to

$$w_t^i \propto w_{t-1}^i \frac{p(\mathbf{z}_t|\mathbf{x}_{0:t}^i, \mathbf{z}_{1:t-1})p(\mathbf{x}_t^i|\mathbf{x}_{0:t-1}^i, \mathbf{z}_{1:t-1})}{q(\mathbf{x}_t^i|\mathbf{x}_{0:t-1}^i, \mathbf{z}_{1:t})} \tag{A.9}$$

Finally, by noting that only the filtered estimate of the posterior $p(\mathbf{x}_t|\mathbf{z}_{1:t})$, is required at each iteration, the particle path history $\mathbf{x}_{0:t-1}^i$ and observation history $\mathbf{z}_{1:t-1}$ can be thrown away, further simplifying (A.9) to

$$w_t^i \propto w_{t-1}^i \frac{p(\mathbf{z}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{z}_t)}, \tag{A.10}$$

which was to be shown.

Appendix B

Additional Results: Comparison of Feature Detectors

The results presented in Chapter 5 focused exclusively on tracking experiments that used silhouette and Sobel edge detection for feature extraction. This appendix contains additional accuracy and performance results for the 6-DOF rigid hand tracking experiment from Section 5.2.2, using two alternative feature extraction configurations:

- Canny edge detection and silhouette detection
- Feature extraction disabled (i.e., direct pixel-to-pixel comparison between particle images and segmented frame)

The methodology, metrics, test bench, system configuration, and video sequence remain the same as Section 5.2.2, but tests were conducted only at optimal resolutions (96×72 and 128×96), with each graph showing a comparison between Sobel, Canny, and no feature extraction. A demonstration of the three possible feature extraction techniques is shown in Fig. B.1. Performance results are presented in Fig. B.2, followed by speedup results in Fig. B.3, and a breakdown of performance details at optimal system settings (1296 particles, 128×96 particle image resolution) is shown in Table B.1, Table B.2, and Table B.3, in the same manner as Section 5.3.2. Quality results are presented in Fig. B.4 for each of the seven metrics described in Section 5.1.2. Finally, a summary of performance and quality results is shown in Fig. B.5, similar to those presented in Section 5.4. A brief discussion of these supplementary results follows.

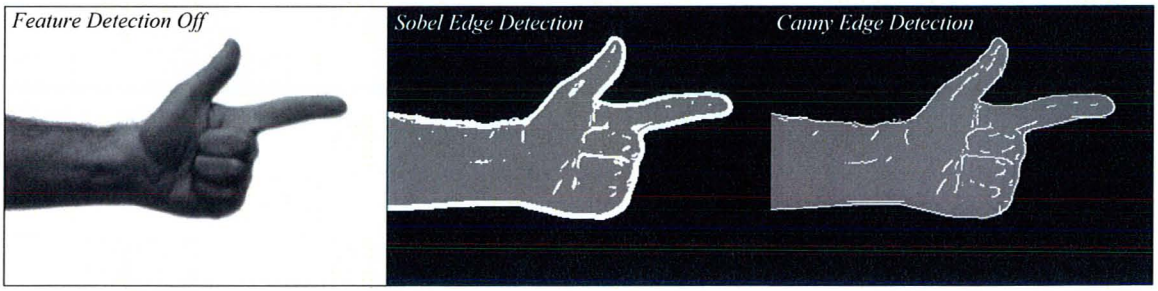

Figure B.1: Feature extraction comparison: demonstration

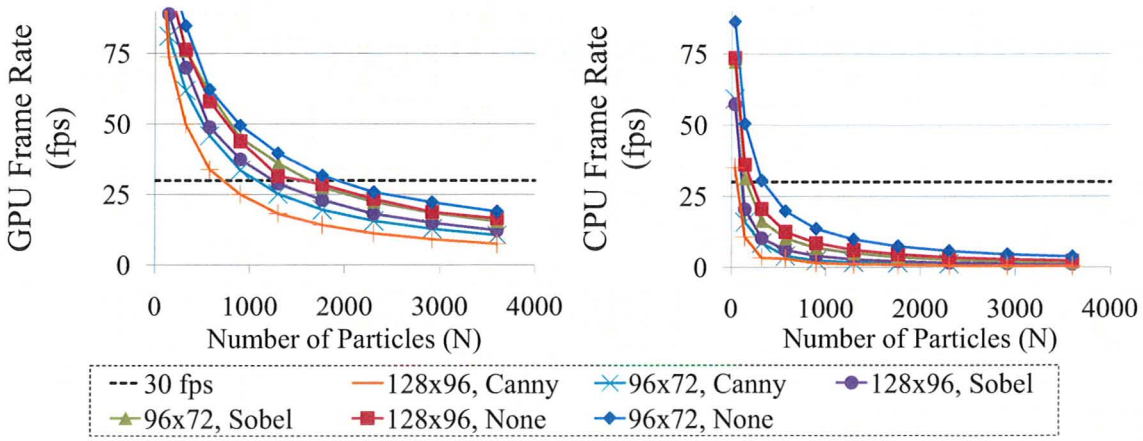

Figure B.2: Feature extraction comparison: performance results

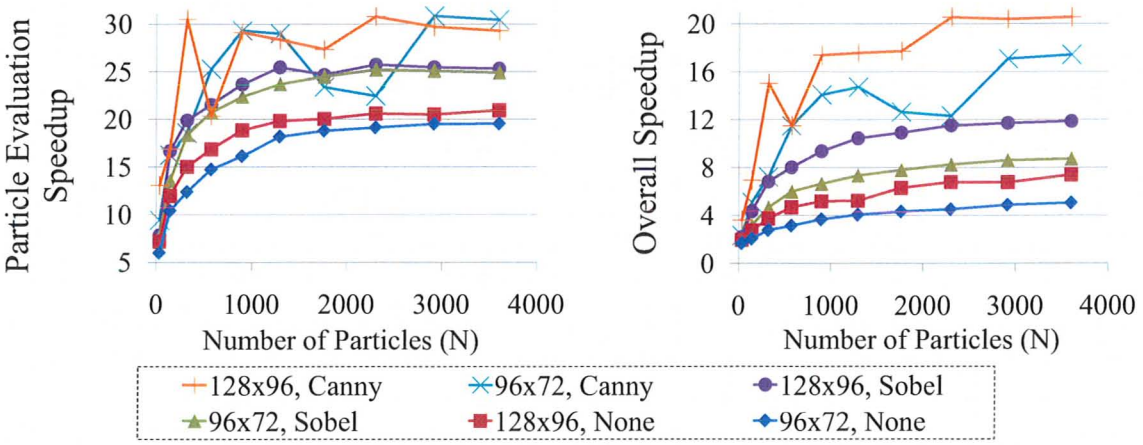

Figure B.3: Feature extraction comparison: speedup results

The functional differences between the three feature extraction techniques can be seen in Fig. B.1. When disabled, a direct pixel-to-pixel comparison is performed between the segmented frame and the particle images. Sobel and Canny edge detection both work in conjunction with silhouette detection to produce a hybrid edge-silhouette map; however, the Sobel detector is more of a brute force approach, resulting in wider, rougher edges.

The finer edges come at the cost of additional computational complexity, as shown in Fig. B.2. The tracking frame rate on the GPU increases gradually with increasing particle count for all feature extractors and resolutions; however, Canny edge detection operates approximately 10 fps slower than Sobel detection. This can be attributed to the additional stages involved in Canny edge detection, described in Section 2.5.2. Additionally, disabling feature extraction entirely yields an increase of 2 to 5 fps over Sobel edge detection. Canny detection performs extremely poorly on the CPU, dropping below 30 fps for just 100 particles. The CPU performs slightly better when feature extraction is disabled entirely, but still drops below 30 fps at a particle count of 350, even at the lower resolution of 96×72.

Speedup results, shown in Fig. B.3, demonstrate that the gains of the GPU implementation are least significant when feature extraction is disabled (5 to 20 times particle evaluation speedup, and 2 to 8 times overall speedup), while speedups are most significant for the more computationally demanding Canny edge detector (9 to 30 times particle evaluation speedup, and 2 to 20 times overall speedup), with the exception of a few anomalies during CPU testing. The performance breakdowns shown in Table B.1, Table B.2, and Table B.3 reinforce this analysis, showing how speedups achieved through a GPU implementation scale according to the computational complexity of an algorithm.

In terms of quality (Fig. B.4), it is interesting to note that Sobel edge detection actually outperformed Canny detection in all metrics except for precision. This can be attributed to the difference in geometry between the hand tracking target and the representative 3D model. Since the edges do not align perfectly, even if tracking is optimal, the wider edges of Sobel detection prove to be valuable in this particular tracking task; however, this is not necessarily the case in all scenarios. For all resolutions and particle counts, quality was lowest when feature extraction was disabled entirely. Fig. B.5 summarizes the performance and quality results, demonstrating Sobel edge detection at a resolution of 128×96 provides the highest quality at any frame rate, justifying its use for the majority of testing in this thesis.

Table B.1: Hand tracking performance analysis (no feature extraction)
1,296 particles, 128×96

Step	GPU (ms)	CPU (ms)	Speedup
Render Particle Images	13.56	9.83	0.72x
Acquire Frame	5.26	5.34	1.01x
Map Resources	2.06	2.13	1.04x
Feat. Det. & Weight Calc.	**7.30**	**144.57**	**19.81x**
Particle Filter	1.98	1.86	0.94x
Other	1.56	1.70	1.07x
Total	31.73 (31.5 fps)	165.42 (6.0 fps)	5.21x

Table B.2: Hand tracking performance analysis (Sobel edge detection)
1,296 particles, 128×96

Step	GPU (ms)	CPU (ms)	Speedup
Render Particle Images	10.31	9.89	0.96x
Acquire Frame	5.50	5.16	0.94x
Map Resources	2.04	2.35	1.15x
Feat. Det. & Weight Calc.	**13.36**	**339.93**	**25.45x**
Particle Filter	1.93	1.89	0.98x
Other	1.49	1.80	1.21x
Total	34.62 (28.9 fps)	361.03 (2.8 fps)	10.43x

Table B.3: Hand tracking performance analysis (Canny edge detection)
1,296 particles, 128×96

Step	GPU (ms)	CPU (ms)	Speedup
Render Particle Images	10.66	10.02	0.94x
Acquire Frame	5.58	2.65	0.49x
Map Resources	2.08	3.88	1.87x
Feat. Det. & Weight Calc.	**33.31**	**944.08**	**28.34x**
Particle Filter	1.91	1.91	1.00x
Other	1.60	1.95	1.22x
Total	54.94 (18.2 fps)	964.49 (1.0 fps)	17.56x

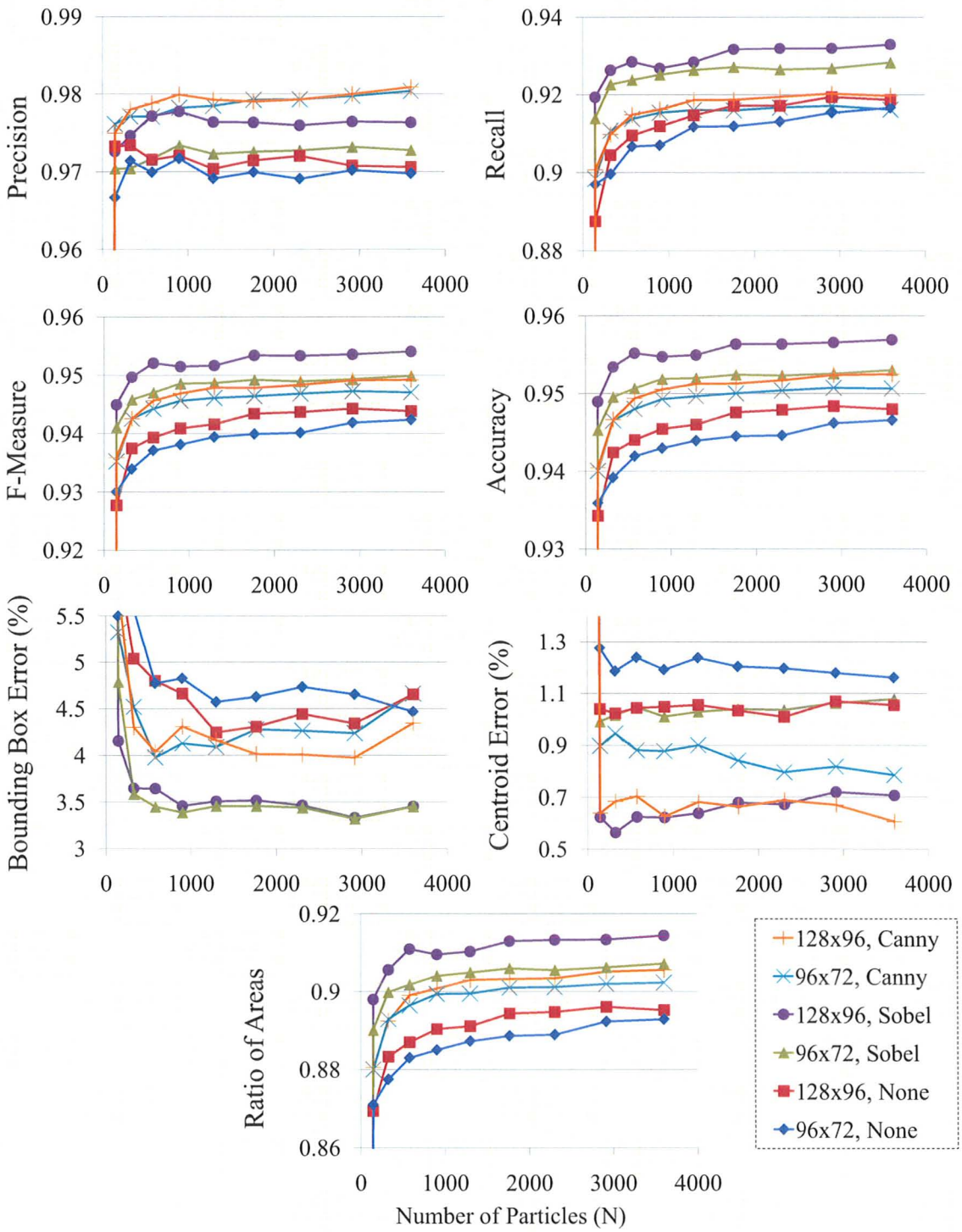

Figure B.4: Feature extraction comparison: quality results

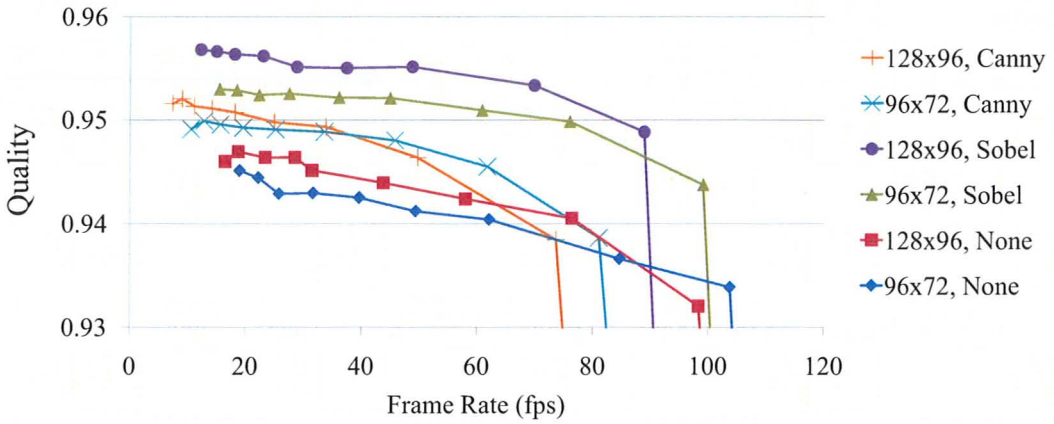

Figure B.5: Feature extraction comparison: summary

Appendix C

Additional Results: Comparison of Motion Models

All experiments in Chapter 5 used a second-order (acceleration) system dynamics model for prediction of global state vector parameters (i.e., translation and rotation). This appendix compares the effect of utilizing a first-order (velocity) or random walk motion model instead (Section 2.5.1), in the context of the wand tracking experiment described in Section 5.2.1. The methodology, metrics, test bench, system configuration, and video sequence remain the same as Section 5.2.1, but tests were conducted only at optimal resolutions (96×72 and 128×96), with each graph showing a comparison between the three motion models. Quality results are presented in Fig. C.1 for each of the seven metrics described in Section 5.1.2 and a summary of quality results is shown in Fig. C.2, using the definition of quality from Section 5.4. Finally, because the wide range of results makes it difficult to distinguish between velocity and acceleration model results, Fig. C.3 examines four particular particle counts of interest in greater detail. A brief discussion of these supplementary results follows.

All metrics shown in Fig. C.1 clearly indicate significantly improved performance using a velocity or acceleration model over a random walk model. Specifically, results demonstrate a 20 to 35 percent increase for all classification-based metrics, approximately 30 percent increase in the ratio of areas metric, a 10 to 20 percent decrease in bounding box error, and a 2 to 10 percent decrease in centroid error. The difference in quality between the velocity and acceleration metrics, however, is effectively indiscernible. The amalgamated quality results in Fig. C.2 confirm these findings, with an approximate 15 to 20 percent increase in overall quality from the random walk system dynamics model to the higher-order models.

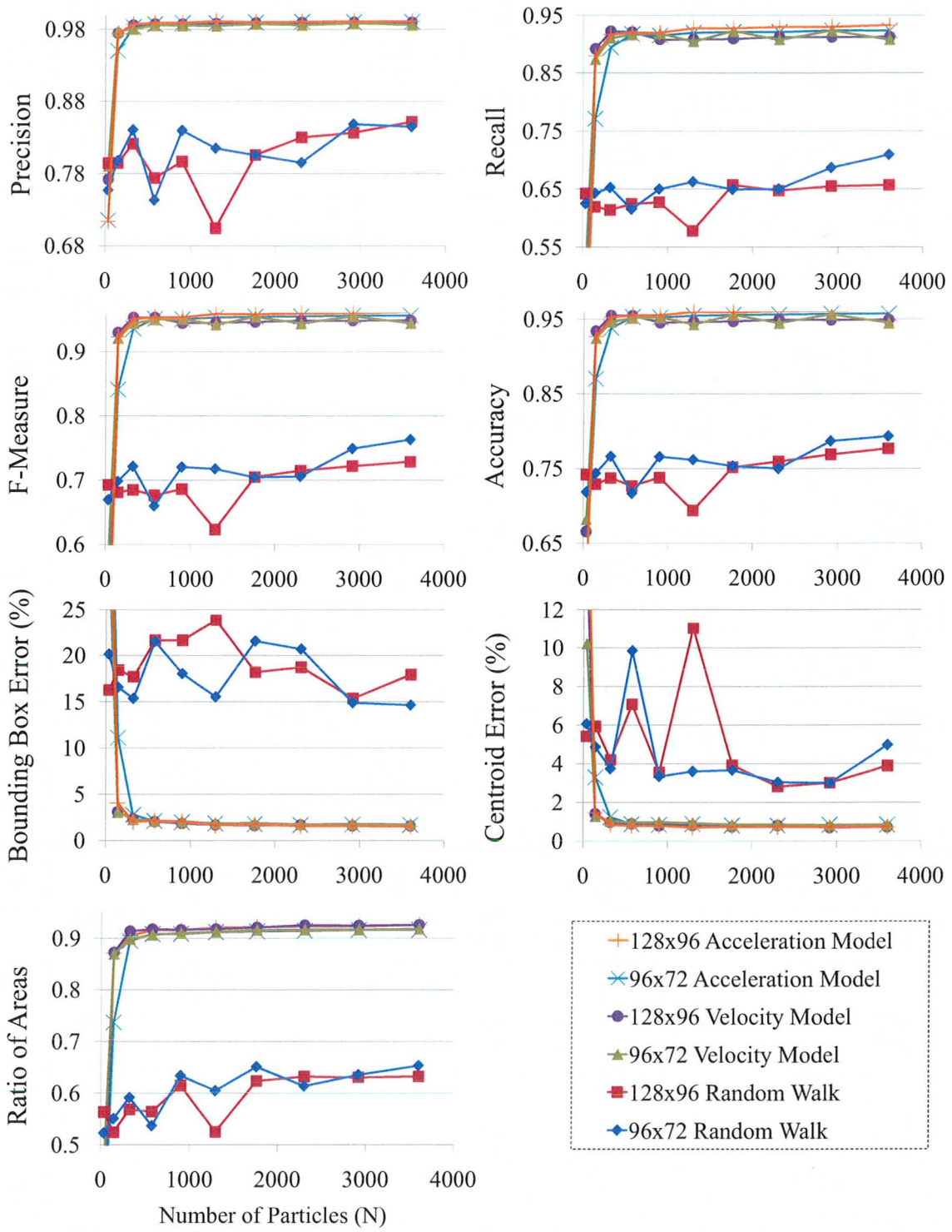

Figure C.1: Motion model comparison: quality results

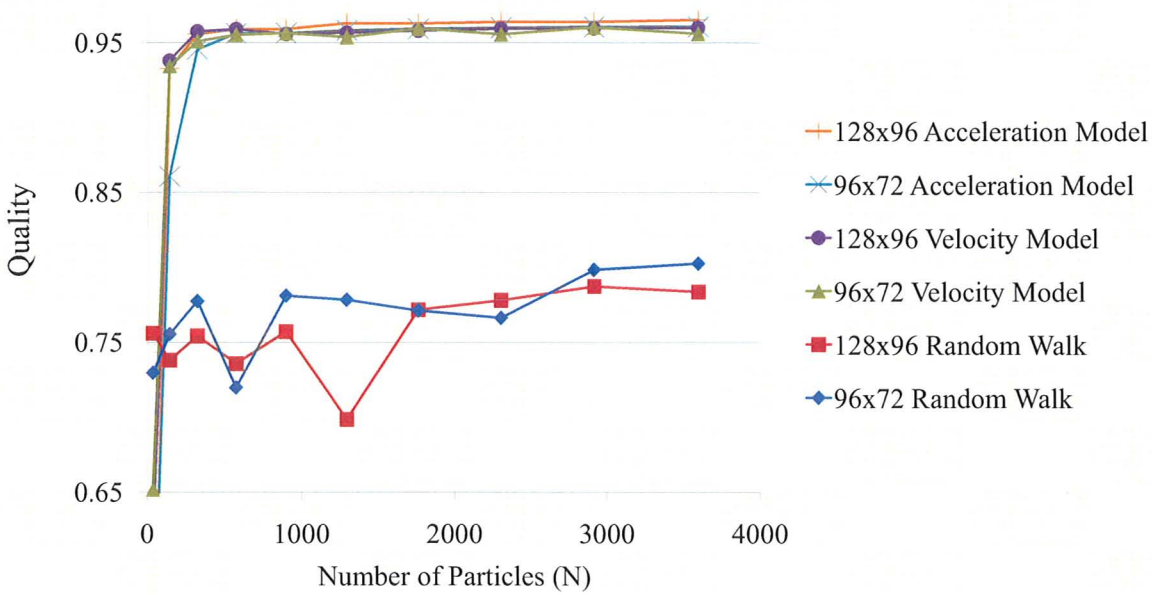

Figure C.2: Motion model comparison: summary

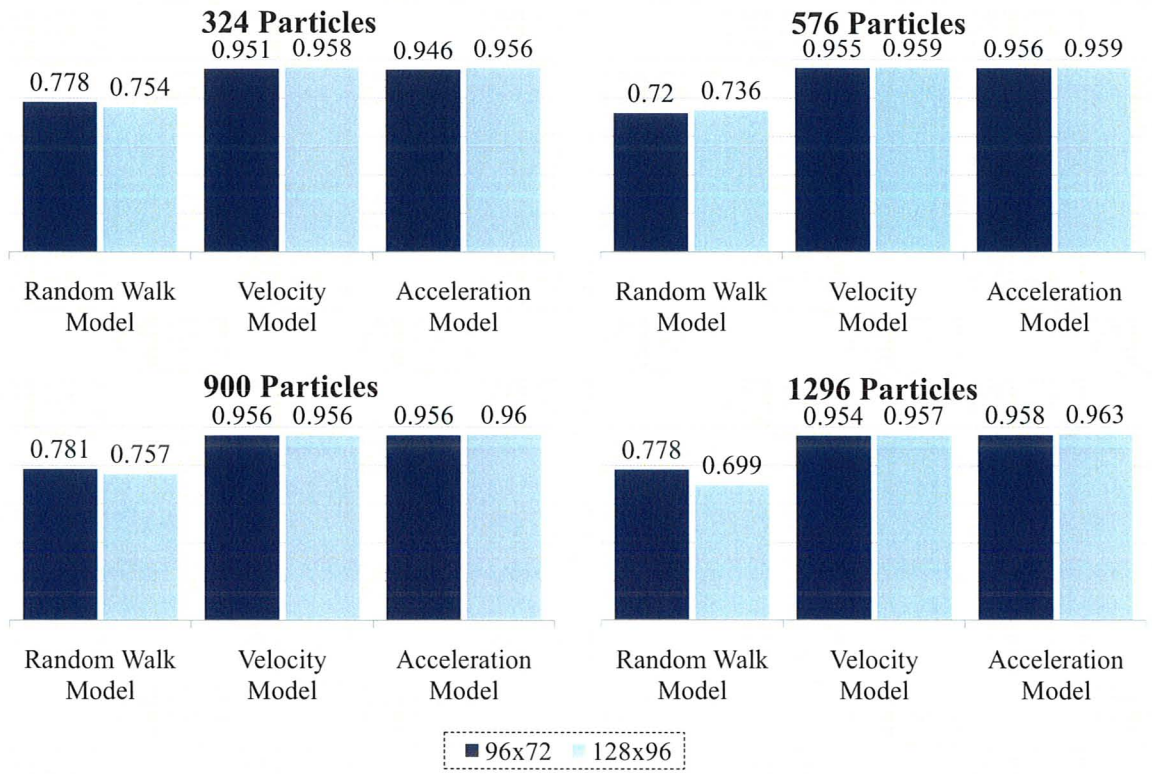

Figure C.3: Motion model comparison: summary details

Fig. C.3 gives a more detailed look at four particle counts in the range of optimal system settings identified in Chapter 5. Here, the minute increase in quality afforded by the use of an acceleration motion model over a velocity model can be seen (in the three highest particle counts). While this increase is effectively negligible, because the choice of motion model does not have an appreciable effect on the frame rate of tracking, the second-order acceleration system dynamics model is considered the optimal choice for this particular tracking task.

Bibliography

Agarwal, S., Awan, A., and Roth, D. (2004). Learning to detect objects in images via a sparse, part-based representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **26**(11), 1475–1490.

Albrecht, I., Haber, J., and Seidel, H. P. (2003). Construction and animation of anatomically based human hand models. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 98–109, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Allied Vision Technologies (2010). Gc660c [online: http://www.alliedvisiontec.com/us/products/cameras/gigabit-ethernet/prosilica-gc-series/gc660.html].

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485.

Arulampalam, M. S., Maskell, S., Gordon, N., and Clapp, T. (2002). A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, **50**(2), 174–188.

Autodesk Maya Press (2009). *Learning Autodesk Maya 2010*. Autodesk, Inc., San Rafael, CA.

Ben, M. and Jiantong, W. (2010). An adaptive particle filter for mems based sins nonlinear initial alignment. In *Information and Automation (ICIA), 2010 IEEE International Conference on*, pages 1504–1509.

Bhusnurmath, A. and Taylor, C. (2008). Graph cuts via ℓ_1 norm minimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **30**(10), 1866–1871.

Brandao, B., Wainer, J., and Goldenstein, S. (2006). Subspace hierarchical particle filter. In *Computer Graphics and Image Processing, 2006. SIBGRAPI '06. 19th Brazilian Symposium on*, pages 194–204.

Bray, M., Koller-Meier, E., and Van Gool, L. (2007). Smart particle filtering for high-dimensional tracking. *Computer Vision and Image Understanding*, **106**(1), 116–129.

Breitenstein, M., Kuettel, D., Weise, T., Van Gool, L., and Pfister, H. (2008). Real-time face pose estimation from single range images. In *CVPR'08: IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8.

Bretzner, L., Laptev, I., and Lindeberg, T. (2002). Hand gesture recognition using multi-scale colour features, hierarchical models and particle filtering. In *FGR '02: Proceedings of the Fifth IEEE International Conference on Automatic Face and Gesture Recognition*, page 423, Washington, DC, USA. IEEE Computer Society.

Brown, J. A. and Capson, D. W. (2010b). Gpu-accelerated 3-d model-based tracking. In *Journal of Physics: Conference Series. Proceedings of the High Performance Computing Symposium (HPCS2010), June 6-9, 2010*, Toronto, Ontario, Canada.

Brown, J. A. and Capson, D. W. (submitted for publication, 2010a). A framework for 3d model-based visual tracking using a gpu-accelerated particle filter. *IEEE Transactions in Visualization and Computer Graphics*.

Cabido, R., Concha, D., Pantrigo, J. J., and Montemayor, A. S. (2009). High speed articulated object tracking using gpus: A particle filter approach. In *ISPAN '09: Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 757–762, Washington, DC, USA. IEEE Computer Society.

Candy, J. V. (2009). *Bayesian Signal Processing: Classical, Modern and Particle Filtering Methods*. Wiley-Interscience, New York, NY, USA.

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **8**(6), 679–698.

Chang, W.-Y., Chen, C.-S., and Jian, Y.-D. (2008). Visual tracking in high-dimensional state space by appearance-guided particle filtering. *IEEE Transactions on Image Processing*, **17**(7), 1154–1167.

Chen, Z. (2003). Bayesian filtering: From kalman filters to particle filters, and beyond. Technical report, McMaster University.

Colic, A., Kalva, H., and Furht, B. (2010). Exploring nvidia-cuda for video coding. In *MMSys '10: Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 13–22, New York, NY. ACM.

Cope, B., Cheung, P. Y., Luk, W., and Howes, L. (2010). Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Transactions on Computers*, **59**, 433–448.

Coutinho, B. R., Teodoro, G. L. M., Oliveira, R. S., Neto, D. O. G., and Ferreira, R. A. C. (2009). Profiling general purpose gpu applications. *Computer Architecture and High Performance Computing, Symposium on*, pages 11–18.

Cui, J. and Sun, Z. (2004). Visual hand motion capture for guiding a dexterous hand. In *Automatic Face and Gesture Recognition, 2004. Proceedings. Sixth IEEE International Conference on*, pages 729–734.

Datla, S. and Gidijala, N. S. (2009). Parallelizing motion jpeg 2000 with cuda. In *2009 Second International Conference on Computer and Electrical Engineering*, volume 1, pages 630–634.

de La Gorce, M., Paragios, N., and Fleet, D. (2008). Model-based hand tracking with texture, shading and self-occlusions. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8.

Delamarre, Q. and Faugeras, O. (1998). Finding pose of hand in video images: A stereo-based approach. In *In IEEE Proc. of the third International Conference on Automatic Face and Gesture Recognition*, pages 585–590. IEEE Computer Society.

Dong, Y. and DeSouza, G. (2009). A new hierarchical particle filtering for markerless human motion capture. In *Computational Intelligence for Visual Intelligence, 2009. CIVI '09. IEEE Workshop on*, pages 14–21.

Doucet, A. (1998). On sequential monte carlo sampling methods for bayesian filtering. Technical report, University of Cambridge, Department of Engineering, Signal Processing Group, England.

Doucet, A., Godsill, S., and Andrieu, C. (2000). On sequential monte carlo sampling methods for bayesian filtering. *Statistics and Computing*, **10**, 197–208.

Doucet, A., De Freitas, N., and Gordon, N., editors (2001). *Sequential Monte Carlo methods in practice*. Springer.

Erol, A., Bebis, G., Nicolescu, M., Boyle, R. D., and Twombly, X. (2005). A review on vision-based full dof hand motion estimation. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops*, page 75, Washington, DC, USA. IEEE Computer Society.

Ferreira, J., Lobo, J., and Dias, J. (2010). Bayesian real-time perception algorithms on gpu. *Journal of Real-Time Image Processing*, pages 1–16.

Fox, D. (2003). Adapting the sample size in particle filters through kld-sampling. *International Journal of Robotics Research*, **22**, 2003.

Fung, J. and Mann, S. (2008). Using graphics devices in reverse: Gpu-based image processing and computer vision. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 9–12.

Fung, J., Tang, F., and Mann, S. (2002). Mediated reality using computer graphics hardware for computer vision. In *ISWC '02: Proceedings of the 6th IEEE International Symposium on Wearable Computers*, pages 83–90.

Gaikwad, A. and Toke, I. M. (2010). Parallel iterative linear solvers on gpu: A financial engineering case. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 607–614.

Gonzalez, R. C. and Woods, R. E. (1992). *Digital Image Processing.* Addison-Wesley Longman Publishing, Boston, MA.

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

Gumpp, T., Azad, P., Welke, K., Oztop, E., Dillmann, R., and Cheng, G. (2006). Unconstrained real-time markerless hand tracking for humanoid interaction. In *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*, pages 88–93.

Ha, P. H., Tsigas, P., and Anshus, O. (2010). The synchronization power of coalesced memory accesses. *IEEE Transactions on Parallel and Distributed Systems*, **21**(7), 939–953.

Halfhill, T. R. (2008). Parallel processing with cuda. *Microprocessor Report.*

Harris, M. (2007). Optimizing parallel reduction in cuda. Technical report, NVIDIA Corporation.

Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A. (2003). Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101.

Hartley, T. D., Catalyurek, U., Ruiz, A., Igual, F., Mayo, R., and Ujaldon, M. (2008). Biomedical image analysis on a cooperative cluster of gpus and multicores. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 15–25.

Heap, T. and Hogg, D. (1996). Towards 3d hand tracking using a deformable model. In *In Face and Gesture Recognition*, pages 140–145.

Ho, Y. and Lee, R. (1964). A bayesian approach to problems in stochastic estimation and control. *IEEE Transactions on Automatic Control*, **9**(4), 333–339.

Hopf, M. and Ertl, T. (1999). Hardware based wavelet transformations. In *Vision, Modeling, and Visualization '99*, pages 317–328.

Huang, Y. and Llach, J. (2008). Tracking the small object through clutter with adaptive particle filter. In *Audio, Language and Image Processing, 2008. ICALIP 2008. International Conference on*, pages 357–362.

Hwu, W. M., Rodrigues, C., Ryoo, S., and Stratton, J. (2009). Compute unified device architecture application suitability. *Computing in Science and Engg.*, **11**(3), 16–26.

Isard, M. and Blake, A. (1998). CONDENSATION - conditional density propagation for visual tracking. *Inter. Journal of Comp. Vis.*, **29**, 5–28.

Junxia, G., Xiaoqing, D., Shengjin, W., and Youshou, W. (2008). Adaptive particle filter with body part segmentation for full body tracking. In *Automatic Face Gesture Recognition, 2008. FG '08. 8th IEEE International Conference on*, pages 1–6.

Kanazawa, K., Koller, D., and Russell, S. (1995). Stochastic simulation algorithms for dynamic probabilistic networks. In *Proceedings of the Eleventh Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 346–35, San Francisco, CA. Morgan Kaufmann.

Kanter, D. (2008). NVIDIA's GT200: Inside a parallel processor [online: http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242].

Kato, M., Chen, Y. W., and Xu, G. (2006). Articulated hand tracking by pca-ica approach. In *FGR '06: Proceedings of the 7th International Conference on Automatic Face and Gesture Recognition*, pages 329–334, Washington, DC, USA. IEEE Computer Society.

Kinsner, M., Capson, D., and Spence, A. (2008). Scale-space ridge detection with GPU acceleration. In *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*, pages 1527–1530.

Kirk, D. B. and mei W. Hwu, W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers, Burlington, MA.

Kirkpatrick, S. and Stoll, E. P. (1981). A very fast shift-register sequence random number generator. *Journal of Computational Physics*, **40**(2), 517–526.

Kitagawa, G. (1996). Monte carlo filter and smoother for non-gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics*, **5**(1), 1–25.

Kuchnio, P. and Capson, D. W. (2009). A parallel mapping of optical flow to compute unified device architecture for motion-based image segmentation. In *ICIP '09: 2009 IEEE International Conference on Image Processing*, pages 2325–2328.

Larsen, E. S. and McAllister, D. (2001). Fast matrix multiplies using graphics hardware. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 55–61.

Lee, S., Kim, G. J., and Choi, S. (2009). Real-time tracking of visually attended objects in virtual environments and its application to lod. *IEEE Transactions on Visualization and Computer Graphics*, **15**(1), 6–19.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010). Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, **38**(3), 451–460.

Lehment, N. H., Arsic, D., Kaiser, M., and Rigoll, G. (2010). Automated pose estimation in 3d point clouds applying annealing particle filters and inverse kinematics on a gpu. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pages 87–92.

Lengyel, J., Reichert, M., Donald, B. R., and Greenberg, D. P. (1990). Real-time robot motion planning using rasterizing computer graphics hardware. In *In Proc. SIGGRAPH*, pages 327–335.

Lenz, C., Panin, G., and Knoll, A. (2008). A gpu-accelerated particle filter with pixel-level likelihood. In *VMV'08: International Workshop on Vision, Modeling and Visualization*.

Lepetit, V. and Fua, P. (2005). Monocular model-based 3d tracking of rigid objects: A survey. In *Foundations and Trends in Computer Graphics and Vision*, pages 1–89.

Liu, J. S. and Chen, R. (1998). Sequential monte carlo methods for dynamic systems. *Journal of the American Statistical Association*, **93**, 1032–1044.

Liu, K.-Y., Tang, L., Li, S.-Q., Wang, L., and Liu, W. (2009). Parallel particle filter algorithm in face tracking. In *ICME'09: Proceedings of the 2009 IEEE international conference on Multimedia and Expo*, pages 1813–1816.

Lozano, O. and Otsuka, K. (2008). Simultaneous and fast 3d tracking of multiple faces in video by gpu-based stream processing. In *ICASSP '08: IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 713–716.

Luna, F. (2003). *Introduction to 3D Game Programming with DirectX 9.0*. Wordware Publishing, Plano, TX.

Maggio, E., Smerladi, F., and Cavallaro, A. (2007). Adaptive multifeature tracking in a particle filtering framework. *IEEE Transactions on Circuits and Systems for Video Technology*, **17**(10), 1348–1359.

Medeiros, H., Gao, X., Kleihorst, R., Park, J., and Kak, A. (2008). A parallel implementation of the color-based particle filter for object tracking. In *ACM SenSys Workshop on Applications, Systems, and Algorithms for Image Sensing*.

Microsoft Corporation (2003). *Microsoft DirectX 9.0c SDK Documentation*.

Mussi, L., Cagnoni, S., and Daolio, F. (2009). Gpu-based road sign detection using particle swarm optimization. In *ISDA*, pages 152–157.

Musso, C., Oudjane, N., and Legland, F. (2001). Improving regularized particle filters. In A. Doucet, N. de Freitas, and N. Gordon, editors, *Sequential Monte Carlo Methods in Practice. New York*, number 12, pages 247–271. Statistics for Engineering and Information Science.

NVIDIA Corporation (2009a). *NVIDIA CUDA Architecture*. Santa Clara, CA.

NVIDIA Corporation (2009b). NVIDIAs Next Generation CUDA Compute Architecture: Fermi. Technical report, NVIDIA Corporation.

NVIDIA Corporation (2010a). *CUDA Best Practices Guide*. Santa Clara, CA, 3.1 edition.

NVIDIA Corporation (2010b). *CUDA Programming Guide*. Santa Clara, CA, 3.1 edition.

Odobez, J.-M., Gatica-Perez, D., and Ba, S. O. (2006). Embedding motion in model-based stochastic tracking. *IEEE Transactions on Image Processing*, **15**(11), 3515–3531.

Owens, John, D., Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, Aaron, E., Purcell, and Timothy, J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, **26**(1), 80–113.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). Gpu computing. *Proceedings of the IEEE*, **96**(5), 879–899.

Papoulis, A. (1984). *Probability, Random Variables, and Stochastic Processes*. McGraw Hill.

Park, S.-H., Kim, Y.-J., Lee, H.-C., and Lim, M.-T. (2008). Improved adaptive particle filter using adjusted variance and gradient data. In *Multisensor Fusion and Integration for Intelligent Systems, 2008. MFI 2008. IEEE International Conference on*.

Pauwels, K. and Van Hulle, M. (2008). Realtime phase-based optical flow on the gpu. In *CVGPU'08: Comput. Vis. Pattern Recognit. Workshop*, pages 1–8.

Pezzementi, Z., Voros, S., and Hager, G. D. (2009). Articulated object tracking by rendering consistent appearance parts. In *ICRA'09: Proceedings of the 2009 IEEE international conference on Robotics and Automation*, pages 1225–1232, Piscataway, NJ, USA. IEEE Press.

Pitt, M. K. and Shephard, N. (1999). Filtering via simulation: Auxiliary particle filters. *Journal of the American Statistical Association*, **94**(446), 590–599.

Pock, T., Unger, M., Cremers, D., and Bischof, H. (2008). Fast and exact solution of total variation models on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8.

Poli, G., Saito, J. H., Mari, Jo, a. F., and Zorzan, M. R. (2008). Processing neocognitron of face recognition on high performance environment based on gpu with cuda architecture. In *SBAC-PAD '08: Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, pages 81–88, Washington, DC, USA. IEEE Computer Society.

Qu, W. and Schonfeld, D. (2007). Real-time decentralized articulated motion analysis and object tracking from videos. *IEEE Transactions on Image Processing*, **16**(8), 2129–2138.

Ripley, B. D. (1987). *Stochastic simulation.* John Wiley & Sons, Inc., New York, NY, USA.

Ristic, B., Arulampalam, S., and Gordon, N. (2004). *Beyond the Kalman Filter: Particle Filters for Tracking Applications.* Artech House.

Roosendaal, T. and Selleri, S. (2009). *The Official Blender 2.3 Guide.* Springer-Verlag/Wien, Austria.

Rui, Y. and Chen, Y. (2001). Better proposal distributions: Object tracking using unscented particle filter. In *CVPR01 II*, pages 786–793.

Rumpf, M. and Strzodka, R. (2001). Level set segmentation in graphics hardware. In *Proceedings. 2001 International Conference on Image Processing*, pages 1103–1106.

Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, Toronto, Canada.

Sankaranarayanan, A. C., Srivastava, A., and Chellappa, R. (2008). Algorithmic and architectural optimizations for computationally efficient particle filtering. *IEEE Transactions on Image Processing*, **17**(5), 737–748.

Schoenemann, T. and Cremers, D. (2010). A combinatorial solution for model-based image segmentation and real-time tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **32**(7), 1153–1164.

Shapiro, L. G., Stockman, G. C., Shapiro, L. G., and Stockman, G. (2001). *Computer Vision.* Prentice Hall.

Sherrod, A. (2006). *Ultimate Game Programming With DirectX.* Charles River Media, Boston, MA.

Smith, K., Gatica Perez, D., Odobez, J.-M., and Ba, S. (2005). Evaluating multi-object tracking. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops*, page 36.

Stenger, B., Mendonca, P., and Cipolla, R. (2001). Model-based 3d tracking of an articulated hand. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 2, pages II–310 – II–315 vol.2.

Stenger, B., Thayananthan, A., Torr, P. H., and Cipolla, R. (2006). Model-based hand tracking using a hierarchical bayesian filter. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **28**(9), 1372–1384.

Sudderth, E. B., M, M. I., Freeman, W. T., and Willsky, A. S. (2004). Visual hand tracking using nonparametric belief propagation. In *Propagation, IEEE Workshop on Generative Model Based Vision*, page 189.

Sun, C., Agrawal, D., and El Abbadi, A. (2003). Hardware acceleration for spatial selections and joins. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 455–466.

Sundaresan, A. and Chellappa, R. (2009). Multicamera tracking of articulated human motion using shape and motion cues. *IEEE Transactions on Image Processing*, **18**(9), 2114–2126.

Tan, G., Guo, Z., Chen, M., and Meng, D. (2009). Single-particle 3d reconstruction from cryo-electron microscopy images on gpu. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 380–389.

Tsuchiyama, R., Nakamura, T., Iizuka, T., Asahara, A., and Miki, S. (2010). *The OpenCL Programming Book*. Fixstars Corporation, Tokyo, Japan.

Ueda, E., Matsumoto, Y., Imai, M., and Ogasawara, T. (2003). A hand-pose estimation for vision-based human interfaces. *IEEE Transactions on Industrial Electronics*, **50**(4), 676–684.

van der Merwe, R., de Freitas, N., Doucet, A., and Wan, E. (2001). The unscented particle filter. In *Advances in Neural Information Processing Systems 13*.

van Rijsbergen, C. J. (1979). *Information Retrieval*. Butterworths, London.

Wang, H., Suter, D., Schindler, K., and Shen, C. (2007). Adaptive object tracking based on an effective appearance filter. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **29**(9), 1661–1667.

Wang, T., Wang, J., Li, C., Wang, H., and Liu, J. (2009). Target tracking based on adaptive particle filter. In *Industrial and Information Systems, 2009. IIS '09. International Conference on*, pages 297–300.

Weng, C., Tseng, C., Ho, M., and Huang, C. (2008). A vision-based hand motion parameter capturing for hci. In *Audio, Language and Image Processing, 2008. ICALIP 2008. International Conference on*, pages 1219–1224.

Wu, Y. and Huang, T. (2001). Hand modeling, analysis and recognition. *Signal Processing Magazine, IEEE*, **18**(3), 51–60.

Wu, Y., Lin, J. Y., and Huang, T. S. (2001). Capturing natural hand articulation. In *In ICCV*, volume 2, pages 426–432.

Xu, T., Wu, H., Zhang, T., Kühnlenz, K., and Buss, M. (2009). Environment adapted active multi-focal vision system for object detection. In *ICRA'09: Proceedings of the 2009 IEEE international conference on Robotics and Automation*, pages 1102–1107.

Yang, C., Wu, Q., Chen, J., and Ge, Z. (2009). Gpu acceleration of high-speed collision molecular dynamics simulation. In *2009 Ninth IEEE International Conference on Computer and Information Technology*, pages 254–259.

Yang, Z., Zhu, Y., and Pu, Y. (2008). Parallel image processing based on cuda. In *CSSE '08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, pages 198–201, Washington, DC, USA. IEEE Computer Society.

Yu, Q. and Medioni, G. (2008). A gpu-based implementation of motion detection from a moving platform. In *CVGPU'08: IEEE Computer Vision and Pattern Recognition Workshop*, pages 1–6.

Zhou, S. K., Chellappa, R., and Moghaddam, B. (2004). Visual tracking and recognition using appearance-adaptive models in particle filters. *IEEE Transactions on Image Processing*, **13**(11), 1491–1506.