VISUAL TRACKING OF HIGHLY ARTICULATED OBJECTS USING MASSIVELY PARALLEL PROCESSORS

BY

DENNIS JINN LIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor Thomas S. Huang, Chair
Professor Narendra Ahuja
Professor David A. Forsyth
Professor Wen-Mei W. Hwu

Abstract

Hand gesture recognition has the potential of simplifying human computer interactions. However, the human hand is a highly articulated object, capable of taking on many different appearances. In this work, we consider an analysis by synthesis approach to this difficult tracking problem. We attempt to overcome the vast amount of computation required by implementing the algorithm on commodity graphics processing units (GPUs). We also collect a lengthy sequence of hand motions from five cameras in order to train and test our algorithm. We show that to achieve good tracking performance, it is important to understand the way that the hand moves. It is of secondary importance to have a good estimate of the hand shape and to be able to process the frames as quickly as possible. Under heavily controlled circumstances, we are able to achieve full tracking accuracy.

Table of Contents

Chapter 1

Introduction

Humans gesture naturally as they interact with each other, and the ability to analyze these motions should help us understand each other and work with machines. Gesture-based interfaces may be useful in situations where carrying an input device can be dangerous or simply cumbersome. A solid hand tracking system can also serve as a first step towards automatic sign language recognition.

However, tracking the human hand poses many unique challenges. Although the overall hand region is easily identifiable via skin-color segmentation, it lacks internal feature points. As a result, locally based techniques like optical flow usually fail. Also, although the hand is constrained by its joints, it can take on a wide variety of shapes. The articulated nature of the motion, along with the relatively deep kinematic chain, makes it very difficult to characterize all possible hand images. Finally, the curse of dimensionality comes into play. The human hand has over 20 internal degrees of freedom, making brute-force techniques impossible.

1.1 Hand Anatomy

The human hand is composed of 27 bones, as shown in Figure 1.1. Eight of those are the carpal bones in the wrist, and they are relatively immobile. The "palm" region is composed of the metacarpal bones. Each finger is composed of three phalanges: proximal, intermediate, and distal. The joint between the metacarpal and the proximal phalange is the metacarpophalangeal (MCP) joint. The joint between the proximal and intermediate phalanges is the proximal interphalangeal (PIP) joint. Finally, the joint between the intermediate and distal phalange is the distal interphalangeal (DIP) joint. The thumb is slightly different in that it only has two phalanges, proximal and distal. The joint between the metacarpal and the proximal phalange is still the MCP, but the joint between the two phalanges is simply

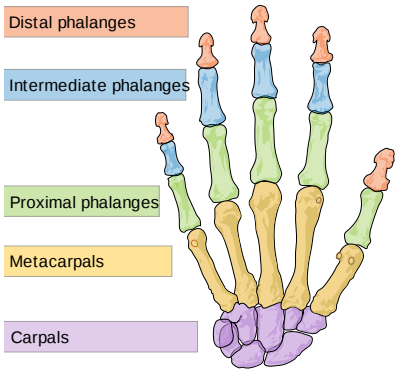


Figure 1.1: Human hand bones

referred to as the interphalangeal (IP) joint. It also differs from the fingers in that it has significant articulation at the wrist or carpometacarpal (CMC) joint.

Riordan [1] gives a view of the hand from a surgical perspective, briefly summarized here. The hand is powered both by muscles within the hand and by tendons connecting to the forearm. Abduction and adduction (the motion of spreading the fingers apart and pushing them back together) is primarily accomplished by muscles at the base of the fingers. The exception is the pinkie, which is abducted by a muscle connected to the ulna.

The flexion of the fingers (the act of closing the hand into a fist) is primarily performed by tendons that run along the palm into the forearm. There are also some muscles at the MCP joint to help us flex. Extension (the opposite of flexion) is much more complicated. The extensor tendons start with muscles in the forearm and run along the back of the hand. However, each tendon splits into three on the dorsal side of the proximal phalanx. One section attaches to the dorsal side of the middle phalanx. The other two wrap around the palmar side of the PIP joint, join again over the dorsal side of the middle phalanx, and then attach to the distal phalanx.

1.1.1 Modeling Tendon Actions

One of the earliest works on finger articulation was performed by Landsmeer [2] in 1961. In this paper, he proposed three models for tendon movement. Landsmeer's Model I deals with a tendon moving over a pulley. In this case the shortening is equal to $r\varphi$, where r is the radius of the pulley and φ is the amount that the joint is rotated. Landsmeer's Model II deals with a tendon running through a loop. In this case the shortening is $2r \sin \frac{1}{2}\varphi$. Landsmeer's Model III corresponds to a tendon running through a tendon-sheath. In this case, there is no simple formula which describes the tendon extrusion.

Taking a different approach, An et al. [3] dissected ten hands to measure tendon loca-

tions and their force and moment potential on different joints. Perhaps this information would be the most useful in determining an energy function for the fingers. However, these measurements are taken for a hand in a resting position. It is unclear how the force and moment potentials would be affected by changing joint-angles.

Chao et al. [4] had performed an analysis three years earlier for several pinching actions. In order to complete the analysis, they had to make quite a few simplifying assumptions. Brook [5] conducted a detailed analysis of the index finger that made fewer assumptions. Both systems yield large systems of equations that are indeterminate. This is because there are many more tendons than degree(s) of freedom (DoF) in the joints. According to [5], every finger in the hand is controlled by no less than six muscles, nine maximum in the fifth, and seven in the index finger. This means that modeling the tendons actually increases the dimensionality of the problem.

Furthermore, the results of [4] show that during a pinch action, a single tendon can exert three times the force of the pinch. This number increases to five in the grasping function. Under these circumstances, the inertia associated with the fingertips would seem to be negligible. As a result, constant velocity or constant acceleration motion models will have dubious value. The best we can hope for is that there is smooth behavior in the muscles that control the tendons.

1.1.2 Hand Dimensions

By dissecting six cadavers (all over the age of 65), Buchholz et al. [6] determined that the length and position of the kinematic segments can be modeled as a function of hand width and length.

His data, shown in Table 1.1, gives the distance between joint centers for each of the digits. Digit I is the thumb, where segment 1 is the carpal segment, 2 is the metacarpal, 3 is the proximal phalangeal, and 4 is the distal phalangeal. The remaining digits are the fingers, starting from the index finger and going to the little finger. In these cases, segment 1 is the carpometacarpal, 2 is the proximal, 3 is the middle phalangeal, and 4 is the distal phalangeal segment. In all cases, the numbers are intended to be multiplied by the overall hand length.

In addition, Buchholz measured the location of the first joint relative to the wrist. The results are shown in Table 1.2. The first joint is the CMC for the thumb and the MCP for the fingers. The x ratios should be multiplied by the hand length and the z ratios should be multiplied by the hand width.

Table 1.1: Relative ratio of phalanx lengths.

Segment	I	II	III	IV	V
1	0.118	0.463	0.446	0.421	0.414
2	0.251	0.245	0.266	0.244	0.204
3	0.196	0.143	0.170	0.165	0.117
4	0.158	0.097	0.108	0.107	0.093

Table 1.2: Relative position of CMC and MCP joints. Note that the z value of finger III is defined to be 0.

Dimension	I	II	III	IV	V
x	0.073	0.447	0.446	0.409	0.368
z	-0.196	-0.251	0.000	0.206	0.402

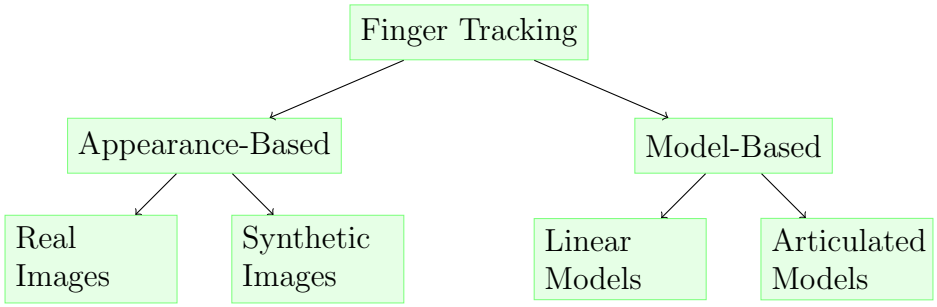


Figure 1.2: Techniques for finger tracking.

1.2 Related Works

A 2007 survey of full DoF hand pose estimation techniques may be found in [7]. This summary covers a subset of the systems described therein and introduces a few which have appeared after the survey. We divide the these finger trackers into a rough taxonomy shown in Figure 1.2.

1.2.1 Appearance-Based Techniques

The general general idea of appearance-based techniques is to build up a collection of images of the hand in different configurations. Then, recognizing the positions of the fingers becomes a database search problem. Using different metrics, the camera image is compared against the training images, and the pose information is read off the label associated with the best match.

Real Hand Images

Capturing the hand in many poses and from many directions is difficult logistically, and labeling the images is a labor-intensive process. As a result, there are very few corpora of hand training images. One exception is the system described in [8]. The authors of this work took video of 24 hand shapes drawn from American Sign Language (ASL). The subject held the signs while rotating his wrist. This provided 14 views spanning approximately 80 degrees along one rotation axis. The system then used a Fourier-based feature and a tree-based classifier to recognize the shapes. The system was able to track a 148 frame sequence containing the shapes and could correctly identify the word signed. It also tracked a longer 333 frame sequence and was able to find reasonable approximations of shapes that were not in the database.

Another instance of using actual hand images for training is [9], although this work only had a database of eight signs and five views. This system used local orientation histograms as features. It treated the recognition task as a database retrieval, using locality sensitive hashing (LSH) as an acceleration technique. It achieved 70% rank-1 retrieval accuracy even when presented with query images that were of a different subject taken with a different model of camera.

Both of these systems have fewer than a thousand examples in their training databases, severely restricting both the number of signs and global rotations at which they could be recognized. The authors of [8] stated that they collected a 7,000 image database signing 24 of the letters of ASL. Also, there is some work toward building an extensive ASL database [10]. However, neither of these data sets appear to have been used as training for tracking or recognition systems.

Synthesized Hand Images

Faced with the logistical challenge of collecting live hand images, researchers resort to generating them using computer graphics. For example, [11] built a regression model over a database of 8,000 hand images. More recent systems, however, treat the problem as a database lookup problem, searching for the nearest match in the training set and then reading the shape parameters associated with that image. The system in [12] matched a database of 107,328 images (4,128 views of 26 different hand poses) using chamfer distance of edges. In order to reduce the expensive chamfer distance computation, it used a Lipschitz embedding as an initial approximation. The system winnows the database using the approximation and performs the full distance computation only on the top 1,000 candidates. Still, the system requires 15 seconds per image on a 1.2 GHz Athlon system. A more recent version of that

system [13] uses a slightly smaller database consisting of 80,640 images (4032 views of 20 hand poses). The database was rendered in "under 24 hours" using the commercial software Poser. This system used a boosting algorithm to choose the exemplars for the Lipschitz embedding and a distance-based hashing algorithm to enable faster search of the database. With these enhancements, it could perform a query in 0.14 s using a 2.0 GHz Athlon.

Another work which used pregenerated templates is [14]. This system used both color and edge likelihoods. It uses a half-chamfer distance to match the edges, but it also takes into account the direction of the edge in the measurement. This system uses an integral image to accelerate the color likelihood computation. To facilitate search and tracking, this system uses a tree-based filter and a motion model trained with articulations captured from a data glove. It could track global motion (six DoF) using 16,055 templates at a rate of two seconds per frame on a 1 GHz Pentium IV. For another sequence, it could track six DoF global (with a restricted range) and two local DoF using 35,000 templates at a rate of three seconds per frame using a 2.4 GHz Pentium IV.

Our system generates synthetic images "on-the-fly" as opposed to precomputing them. On a practical front, this means that our generated images are going to be simpler. However, it is not clear that synthetic hands with higher apparent quality will yield better tracker accuracy. Also, the speed advantage which precomputation grants is being reduced by trends in computer architecture. While memory size continues to increase, memory bandwidth to the chip is not scaling. As a result, modern processors face a "memory wall," whereby the arithmetic units are consuming images from the database faster than the memory system can supply them. Our system, by producing the images inside the processing unit, avoids this bottleneck.

On the theoretical side, generating images on the fly gives us increased flexibility. We can generate more intermediate poses and views than the largest feasible database. We can also adapt more easily to users with different finger lengths. On the other hand, our system does not have access to the information on the global structure of hand shapes contained in the search trees and hashes. It is possible to envision a hybrid system where precomputed information is used to quickly narrow down candidates for initialization while fine-grained tracking is performed with dynamically generated images.

1.2.2 Model-Based Techniques

The model-based approach is an alternative to the appearance-based approach. These techniques try to understand and encode the underlying structure of the hand. The approaches in this section are more varied. Some try to perform analysis on portions of the hand image

to infer the finger location. Many propose hypothetical hand configurations and compare them with the camera image. Theoretically, because model-based systems are not restricted to a set of training images, they should be more flexible and user-independent. However, because they do not offload work to a precomputing state, they can be significantly more computationally expensive.

Linear Models

Linear models are the simplest ones possible, and there is some early work in applying them to finger tracking. For example, in their work describing the application particle filtering to computer vision, [15] includes an example of tracking a hand over a cluttered background. For this system, the palm remains parallel to the image plane and is allowed to translate and rotate. The fingers and thumb adduct and abduct independently, but there is no flexion or extension. The system worked in a 12 dimensional space. Two of the dimensions deal with in-plane translation. The remaining ten were trained using principal component analysis (PCA) and handle the hand shape and in-plane rotations. The system used edges as its feature and has a motion model derived from a training sequence. The system could track the hand for 500 frames in a cluttered scene. The restriction on the palm essentially reduces the problem from 3D to 2D. However, the particle filtering technique proves to be particularly robust, and we adopt it for our system (Section 2.5).

The work of [16] also uses PCA to model the hand shapes. However, their system works in 3D and factors in texture instead of simply looking for contours. The advantage of using PCA based models is that it becomes very easy to compute the shape given the model parameters. However, because PCA is a linear model, it is an awkward technique for representing the inherently nonlinear nature of articulated motion. In general, more recent hand tracking systems have adopted other approaches.

1.2.3 Skeletal Models

Given the structure of the hand, it is natural to actually carry a full kinematic model into the tracking system. One of the oldest systems covered in this survey is the DigitEyes system [17, 18]. It uses two cameras and can track a full 27 DoF. This system models the hand with a full kinematic model similar to the one found in Section 2.1.1 of this work. Their work, like ours, relies on the occluding contour of the hand. Instead of a general rendering, however, they use a wireframe-like model of "link" and "tip" components. The link models represent the line at the center of a phalanx (which is modeled as a cylinder). To measure the model against the image, the system takes 1D profiles at regularly spaced intervals perpendicular

to the link. The edges of the finger are found by looking at the derivatives along the profile. If only one silhouette edge can be determined, the center can be estimated with knowledge of the width of the finger. The "tip" model is a hemisphere model for the fingertip which is estimated in a similar fashion. Using the local trackers and sampling reduces the computation cost. The DigitEyes tracker uses a single hypothesis and the Levenburg-Marquardt (LM) nonlinear least-squares solver. They used the system to track a 200 frame sequence with no occlusion [17]. The system can run at 6.6 Hz with 27 DoF, on a 68040 processor. To handle occlusion, they developed a sophisticated technique of deciding which phalanx is in front of the other. This system was too slow to run in real time, but could track two fingers (9 DoF) where one occluded the other for 80 frames. Our work takes a similar approach, building a kinematic model of the hand and comparing it to images from multiple cameras. However, increased computation power allows us to fully render the hand and compare the entire camera image rather than carefully selected slices. We also have the power to use multiple hypotheses in our tracker to improve robustness.

Whereas [18] worked solely in 2D, [19] works entirely in 3D. Using four cameras, the system constructs a voxel-based model scene. Then, they fit a 31 DoF hand model to the 3D information. The tracker is a single-hypothesis system which uses "virtual torque" to align the model with the observed information. The system can run at two frames per second on a dual 1 GHz Pentium III. The paper does not mention tracking a sequence. Instead, it gives two examples of tracking results. Our system essentially uses the same information, occluding contours from multiple cameras. However, we avoid projecting to 3D, which should enable us to reduce memory usage while achieving the same accuracy.

The system described in [20] contains one of the most sophisticated tracking algorithms described in this review. It uses a graphical model with nonparametric belief propagation as a tracking system. The model is especially interesting in that it does not use the "joint-angle" representation adopted by most of the other systems. Instead, it opts for a highly redundant representation, specifying the location and position of each phalanx using six values. Energy functions in the graphical model then ensure that the parts actually connect properly to form a hand. The system uses chamfer distances on the edges and color likelihood over the silhouette as features. The system was tested on both global motion and on slight flexing of the fingers. For all the test sequences, the palm was mostly facing the camera. The process requires approximately four minutes per frame on a Pentium IV system. Two hundred particles were used in the nonparametric messages. Our system uses a significantly less sophisticated propagation model. This, however, makes updates much faster. Combined with multiple cameras, we can achieve processing that is much closer to real-time speeds. Our use of the full chamfer distance on silhouettes usually alleviates the need for repulsion

forces to keep the phalanxes from intersecting. By using a simpler parametrization of the hand, we reduce the number of dimensions and avoid kinematic singularities.

In contrast, the work of [21] uses a much simpler, single hypothesis of the hand pose. However, it uses one of the most sophisticated on-line models of the visual appearance of the hand. This model consists of 1000 facets, 22 articulation (hand pose) parameters, and 51 morphological (hand shape) parameters. It can handle lighting and texture, but not shadows. It tracks by using a quasi-Newton descent, taking special care of boundary conditions to avoid discontinuities in the objective function. It could track the same sequences as those used by [14], achieving good results with full 22 local DoF. Note that [14] only could handle two local DoF. This suggests an advantage in using model-based systems in tracking complex hand motion. No information was given in [21] on the execution time, but it is presumably much slower than our system. However, it may be advantageous to incorporate aspects of this more sophisticated model into our system. Perhaps this will permit our system to function well even with a single view of the hand.

Chapter 2

Hand Tracker

We use an analysis-synthesis approach for our hand tracker. The block diagram of Figure 2.1 highlights the major components. For each frame, we generate a number of candidate hand poses. Using that information and our 3D model of the hand, we render the candidate images. Then, we compare that image to the camera image using the error metric. Next, we feed the new error information into the solver, which proposes new candidate hand poses in an attempt to minimize the disparity. The cycle then repeats until the tracker's hand pose converges on the true pose. This system is relatively simple and elegant. It trivially adapts to multiple cameras; we simply render additional images for each camera viewpoint and hand pose. Since we are rendering from scratch, we can dynamically adapt to the user's hand shape. We also do not need to restrict the range of motion in order to build a database. Conversely, it is also easy to enforce static and dynamic constraints on the candidate hand poses. The main drawback of this approach is the amount of computation involved. We partially overcome this by using high throughput massively parallel processors such as NVIDIA GPUs.

The original version of the hand tracker was written in a combination of MATLAB and C++, with the components communicating with each other using TCP/IP. Since then, the tracker has evolved to use Python and C++, with the `boost::python` [22] library supplying the glue layer. Python, along with the NumPy [23] library for matrix processing and matplotlib [24] library for plotting, provides most of the functionality of MATLAB. It retains the ease of rapid prototyping development while providing a nicer interface for binding to the C++ routines. This allowed us to eliminate the TCP/IP layer and improve performance.

In this chapter, we describe each of the components in turn. We begin with the parameters used to model the hand and the distance measure that we use. We then discuss the implementation details of two "renderers." This component performs the image generation and the error computation steps in a single pass. The first version, based on OpenGL, is

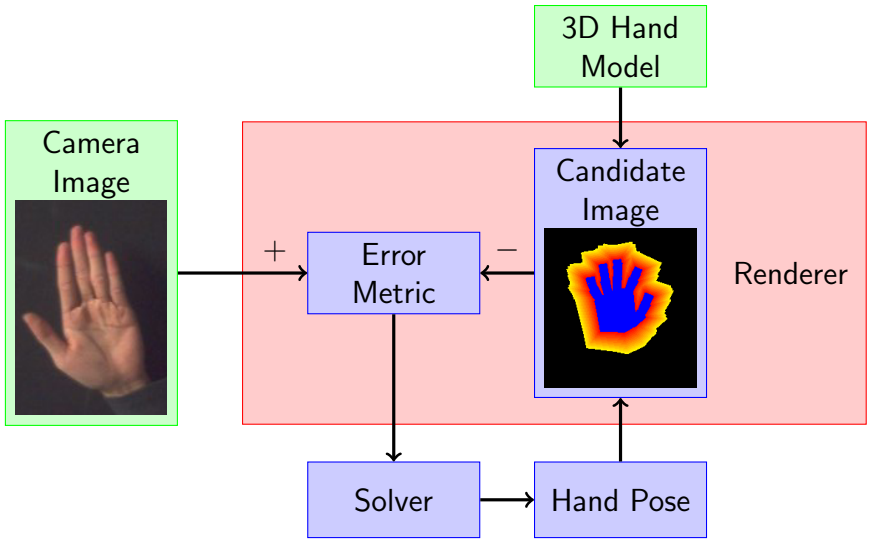


Figure 2.1: Block diagram of our hand tracker.

suitable for ATI and older NVIDIA GPU platforms. The second, based on a new software rendering engine, is designed to attain optimal performance with NVIDIA's CUDA platform. Finally, we detail our implementation for the solver routine.

2.1 Hand Model

To represent the hand for rendering, we need to know something about its parameters. We divide these into two parts. The first part is the "hand pose" model, which captures the dynamic properties of hand motion. These change from frame to frame. The remaining parameters, the "hand shape" model, are intrinsic values that remain constant throughout the sequence.

2.1.1 Hand Pose Model

As described in Section 1.1, the hand is a very complex anatomical structure. It is capable of many different types of movement, including subtle articulations of the metacarpals and slight twisting of the fingers. These are not very noticeable, however, and in the interest of keeping the model simple, we focus on the more major articulations. Even then, we take some liberties with the thumb. Also, we introduce a "palm fold" joint that has no real correspondence with physiology.

Our simplified hand model had 21 local DoF. For each finger, we use two degrees (flexion and adduction) at the MCP joint, and one degree (flexion) at the PIP and DIP joints. For

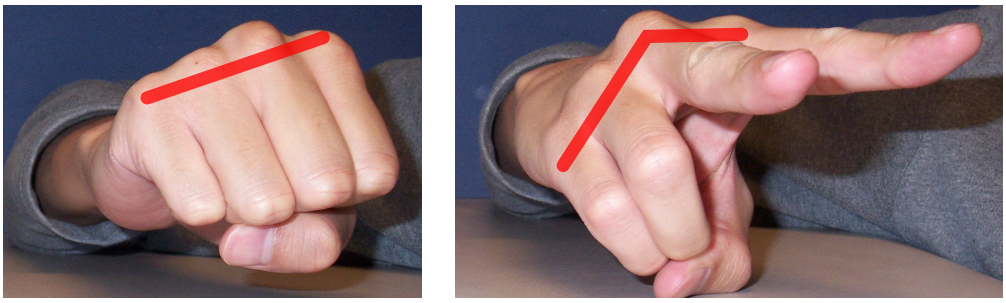


Figure 2.2: Illustration of hand folding. Note that for some gestures the knuckles no longer lie in a plane, and the axes for flexion are no longer aligned.

the sake of symmetry, we treat the thumb like a finger, pretending that its metacarpal bone is the the proximal phalange. That is, we "attach" the thumb to the hand at its CMC joint, giving it two degrees of freedom there. We treat the thumb's MCP joint as though it is the PIP joint, with only one degree of flexion. Finally, we treat the thumb's IP joint as though it was the DIP joint, with a degree of flexion. This model is arguably incorrect with the degrees of freedom in the wrong places. However, we have found it sufficient to describe the range of motion that we observe in the data set.

The remaining local DoF is designed to capture "palm folding." Most of the time, all the MCP joints of the fingers lie in the same plane. However, as illustrated in Figure 2.2, during certain motions the metacarpals shift, invalidating the approximation. As a kludge, we add an extra "joint," centered about the thumb base, which affects the bases of the middle, ring, and pinkie fingers.

Additionally, our pose model also contains 6 global DoF relating the position and orientation of the palm in space. We represent the orientation with Euler angles, choosing the ordering so that gimbal lock is unlikely. We represent the position as (x,y,z) coordinates. Note that these are the only part of the hand pose measured in length instead of as an angle.

2.1.2 Hand Shape Model

The hand shape model consists of three basic parts: phalanx lengths, keypoint locations, and joint axes. The phalanx lengths are fairly self-explanatory. There are three of these parameters for every finger, including the thumb. The keypoints represent the x and y location[1] of the bases of fingers. We also have two keypoints near the wrists. These points help define how we render the palm.

The joint axes are somewhat more complex. In theory, each axis can lie anywhere on the

[1]For the hand's local coordinate system, we use a convention where x is along the width, y is along the length, and z is along the depth.

unit sphere. However, since we model the finger as a segment in the "transformed" y direction, it does not make sense for the finger's joint axes to have a y component. We constrain the MCP flexion/extension axes to roughly lie in the plane of the palm, and we constrain the MCP abduction/adduction axes to be roughly perpendicular to the flexion/extension axes. The "hand fold" pseudo-joint, however, is allowed to be at any angle, and can be anywhere on the unit sphere.

2.2 Distance Function

The error metric is responsible for evaluating the closeness of the fit between the camera image and the rendered image. Since we intend to evaluate many candidates, it is important for this operation to be fast. However, it is equally important for the distance to be relatively smooth and free of local minima. This section will explore three possibilities and their implications.

The simplest possible error metric is the sum squared difference (SSD) error. Since we are dealing with binary silhouettes, the distance reduces to an XOR operation between the camera image and rendered image pixels. Essentially, it counts the number of pixels that are different between the two images. This can be computed very quickly and easily, and is the approach used by [25]. However, this metric is non-differentiable with respect to almost all parametrizations [26]. As a result, it is prone to local minima. If the camera image and the rendered images do not overlap at all, this distance gives no information on how to adjust the rendered image. Even if the two images are roughly aligned, this metric can send the wrong signal to the solver. An example of this is shown in Figure 2.3. Note that the SSD distance increases as the thumb abducts towards the correct pose. Also, the SSD is often minimized by reducing the number of pixels in the rendered image. As a result, experience has shown that this metric often causes the fingers to "shrivel" into the palm, making it a poor choice for effective tracking.

An improved error metric may be found by using the chamfer distance. The definition that that we use, based on the original proposal by [27], is as follows: Given two sets A and B, and an underlying distance metric d, the chamfer distance $d_{\text{cfr}}(A, B)$ is given by:

$$d_{\text{cfr}}(A, B) = w_1 d_{\text{half-cfr}}(A, B) + w_2 d_{\text{half-cfr}}(B, A) \tag{2.1}$$

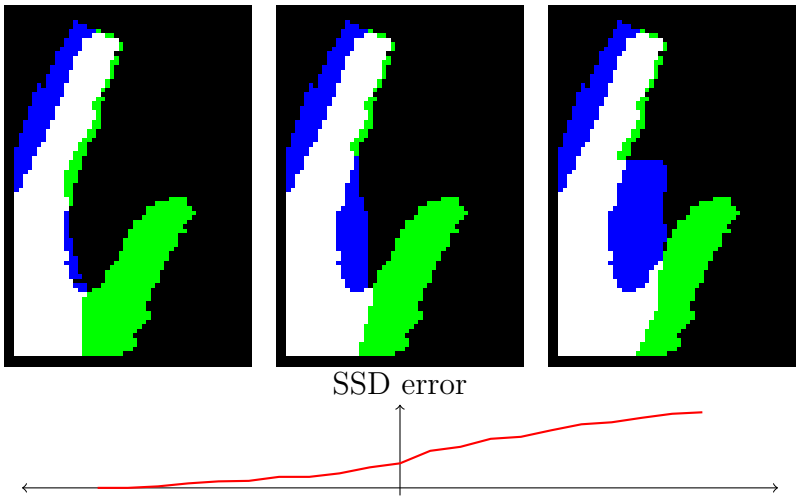


Figure 2.3: Illustration of SSD error. The green represents the camera image, and the blue represents possible rendered candidates. Note that the error increases as the thumb moves toward the correct location.

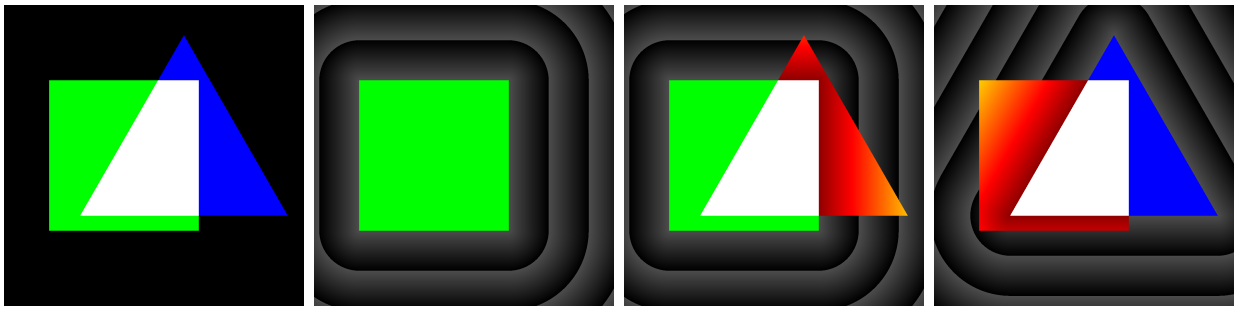
where w_1 and w_2 are arbitrary weights, and

$$d_{\text{half-cfr}}(A, B) = \sum_{a \in A} \min_{b \in B} d(a, b). \tag{2.2}$$

In short, the directed chamfer distance $d_{\text{half-cfr}}(A, B)$ is the average distance from a point in A to its closest point in B. Note that although it is commonly referred to as a "distance," $d_{\text{cfr}}(A, B)$ is not a metric.

One way to compute $d_{\text{half-cfr}}(A, B)$ begins with calculating a distance transform on B. As shown in Figure 2.4(b), the value of each pixel in the transformed image is the distance to the nearest element in B. (Pixels that are in B have a value of 0.) Then, calculating the $d_{\text{half-cfr}}(A, B)$ simply involves summing over all the values in the distance transform under A, as indicated by Figure 2.4(c). Similarly, as illustrated by Figure 2.4(d), $d_{\text{half-cfr}}(B, A)$ may be computed by a distance transform on A and summing over the pixels of B.

Computing only $d_{\text{half-cfr}}(camera, rendered)$ is a popular choice, since it requires only a single distance transform per camera image. By contrast, computing $d_{\text{half-cfr}}(rendered, camera)$ requires a distance transform for every candidate, which can be hundreds or thousands of times more numerous. Thus, this metric is used by [14] for edges. However, as shown in Figure 2.5, this metric suffers from the same issues as the SSD. The $d_{\text{half-cfr}}(camera, rendered)$ component of the full chamfer distance, illustrated in Figure 2.6, can help us avoid spurious local minima. In this case, as the thumb abducts toward the correct pose, the error imposed by the camera's view of the thumb decreases. Recognizing the importance of this term [12]



(a) (b) (c) (d)

Figure 2.4: The chamfer distance. (a) Two shapes A (blue) and B (green). (b) The distance transform of B. (c) The half-chamfer distance $d_{\text{half-cfr}}(A, B)$. (d) The half-chamfer distance $d_{\text{half-cfr}}(B, A)$.

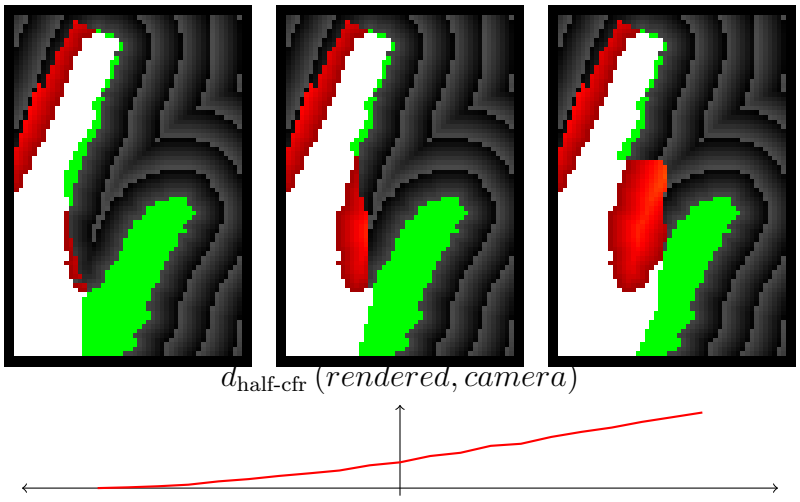


Figure 2.5: Illustration of the half-chamfer distance $d_{\text{half-cfr}}(rendered, camera)$. The green represents the camera image, and the blue represents possible rendered candidates. Note that this suffers the same issues as the SSD distance.

includes it in their tracker. However, because of resource constraints, they use a database-based embedding to approximate $d_{\text{half-cfr}}(camera, rendered)$. Using GPU acceleration, we will actually compute both terms directly.

2.3 OpenGL Renderer

As mentioned in the overview, we combine the candidate image rendering and distance metric computation in a single step. This task is accomplished by the "renderer" component. The OpenGL renderer is the older of the two renderers. It was written using general purpose GPU (GPGPU) style programming, and could work with cards that had limited programmability.

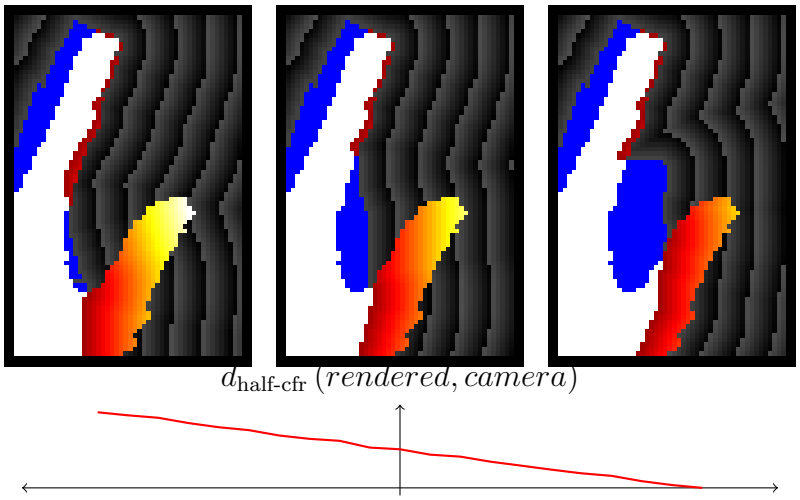


Figure 2.6: Illustration of the half-chamfer distance $d_{\text{half-cfr}}\,(camera, rendered)$. The green represents the camera image, and the blue represents possible rendered candidates. This error measure actually decreases as the thumb moves towards the correct location.

OpenGL, like DirectX, is a sophisticated application programming interface (API) designed for high performance rasterization. As such, there are a few minor perks to using OpenGL for rendering. First, OpenGL is designed for 3D. We simply need to specify points in the world coordinate, and the hardware will take care of the different viewpoints, even in the case of asymmetric view frustums. Similarly, the hardware rasterizer will find the "inside" of our polygons. This means that parts naturally get bigger as they get closer to the camera. Finally, the texturing hardware will automatically filter and interpolate the camera image. This allows us to zoom in and focus our attention to the hand region.

Before this renderer was written, most of the research was conducted using MATLAB. However, obtaining (and maintaining) an OpenGL context from within MATLAB seemed too impractical given MATLAB's limited integration with C. Thus, the initial version was a standalone program that spoke to a controlling MATLAB process using TCP/IP. However, the overhead of marshaling data across the TCP channel proved to be prohibitive. As a result, we replaced the MATLAB component with Python and NumPy. Python, being a lighter weight host with better C integration, could embed the renderer directly. This dramatically sped up performance and simplified development.

In order to obtain maximum performance, we applied several techniques. The first is to minimize the number of polygons drawn by simplifying the geometry. The second is to use the z-buffer to accelerate the computation of the distance transform. Finally, we use a combination of OpenGL Shading Language (GLSL) shaders and the mipmap generation hardware in order to compute the chamfer distance efficiently.

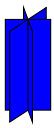


Figure 2.7: The polygon rendered to approximate a cylinder.

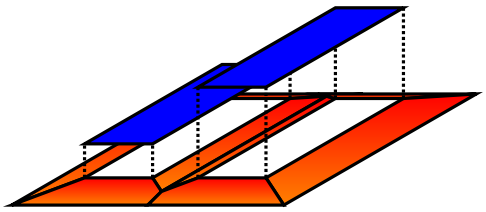


Figure 2.8: Approximating the chamfer distance by rendering extra polygons.

2.3.1 Hand Geometry Model

For both renderers, we approximate the finger phalanxes using cylinders. To simplify the geometry for OpenGL, we replace each phalanx with three overlapping 3D rectangles, as shown in Figure 2.7. When rendered without the lighting, the silhouette of one of these polygons approximates a cylinder. A similar method, with many more polygons, is applied to the palm, which is modeled as a single rigid body.

2.3.2 Approximating the Distance Transform

However, we can accelerate an approximate distance transform on the GPU using the technique proposed by [28].

Figure 2.8 illustrates the method. Every time we render a polygon (blue), we also render several extra "wings" (orange). These wings are forced back to the far edge of the framebuffer, so that they will always be occluded by palm polygons. We assign a texture coordinate of zero to the edge of the wing polygon that is adjacent to the corresponding palm polygon, and we assign some maximum value to the far edge. Then, the texture interpolation will color the each pixel with the distance to the associated palm polygon. In addition, the wing polygons are progressively sloped away from the camera. Therefore, when two wing polygons overlap, pixels corresponding to smaller distances (closer to a palm polygon) occludes pixels corresponding to larger distances. As a result, the final image consists of pixels which record the smallest distance value.

Table 2.1: Relative performance of CPU and GPU implementations. These timings are based on a Intel Core2 6600 with an NVIDIA GeForce 7600 GT. MESA is a CPU-based OpenGL implementation.

	Comparison/Sec		
Batch Size	GPU	MESA	Speedup
1	595.33	29.56	20x
4	1522.61	29.67	51x
16	2430.04	29.43	82x
64	2829.19	29.31	97x

2.3.3 Error Distance Computation

To avoid having to re-read the image from the framebuffer, we render the error image in a single pass. In order to do this, we apply a different GLSL fragment shader to the hand polygons and the wings. The shader for hand pixels produces a value of 0 if the corresponding pixel in the camera image is a 1 (indicating a match). Otherwise, that shader returns the value of the precomputed distance transform of the camera image. Similarly, the shader for the wing polygons produces a 0 if the corresponding pixel in the camera image is a 0. Otherwise, it returns an output proportional to its z value. Thus, after rendering, the framebuffer contains the "error image."

However, we do not actually need to error image; instead we merely need the sum of all its pixels. To avoid having to transfer all the pixels back to the central processing unit (CPU), we perform the summation on the GPU. To do this, we use the hardware mipmap-generation feature . We use the OpenGL call to build the mipmap pyramid. By reading the appropriate pixel near the top of the pyramid, we can retrieve the sum of a certain area of the original image.

2.3.4 Performance

Of course, the major advantage of pushing the work to the GPU is the speed. As Table 2.1 shows, even mediocre GPU hardware can significantly outperform a CPU-based OpenGL implementation. However, to achieve this speed, it is necessary to batch transactions. To do this, we allocate a large framebuffer, and then proceed to tile it with multiple hands. Having multiple hands in flight means that the GPU can parallelize the rendering better and save on per-transaction overhead. A side effect of this is that it is most efficient to be able to submit a batch of at least 64 renderings before receiving any results.

2.3.5 Drawbacks

Unfortunately, although we are nominally rendering a 3D object, our task does not quite match the traditional workload of a GPU. The most obvious mismatch is that most applications do not need to render thousands of frames per second. Even after batching the workload, we still wish to render many hundreds of frames per second. Clearing the framebuffer between each rendering pass is a simple but fairly expensive operation, as it rewrites every pixel. Also, modern GPU workloads rely on shipping large batches of geometry and expect to expend large amounts of processing on fragment shader work. This is true if we are dealing with relatively complex 3D objects that are rendered with sophisticated lighting equations. However, our hands have a relatively simple geometry and a trivial amount of work in the fragment shaders.

Another issue is that the OpenGL interface, which is designed to give implementations flexibility, is very opaque from a programmer's point of view. Thus, we can only speculate on some of the reasons for poor performance. On possibility is that the fingers are long and skinny, leading to large regions of abrupt changes in z value. I suspect that this means that we are defeating the early z-cull feature of the hardware. This feature is designed to avoid rendering fragments known to be behind something that has already been rendered. However, it is normally implemented by a hierarchy of z-buffers, which relies on the fact that the depth map is relatively smooth. Also, profiling seems to indicate a bottleneck in the raster operation, where finished pixels are written to the framebuffer. However, writing only one output channel instead of the four channels for red-green-blue-alpha (RGBA) data does not seem to increase throughput. This may be caused by issues in the layout of the framebuffer in the memory or by an issue in updating the z buffer.

Finally, there is an issue with the accuracy of the rendering. The polygons only approximate the idealized cylinders which should make up the fingers. Also, the wings do not exactly compute the distance transform. For simplicity, we use only four wing polygons per rectangle. This yields an exactly accurate result along the edges, but it underestimates the distance near the corners. Worse, the wings do not extend forever, meaning that the result can be highly inaccurate if there is a wild mismatch between the camera and rendered images. As a result of all of this, the resulting distance function is filled with artifacts and is not smooth with respect to the input parameters.

2.4 Software Renderer

In 2007, NVIDIA unveiled its CUDA programming language, which allows for a C-like environment for programming its GPUs. This change corresponded to a shift in GPU architecture. Whereas older-generation GPUs have fixed pipelines with limited programmability, the newer generation hardware was simply a giant unified bank of simple processors. These processing units are much more flexible than the simplified models of old. They can perform arbitrary writes to GPU memory. Furthermore, the CUDA environment allows programs to access the on-chip cache on the GPU itself, and we use this ability to remove the raster operation bottleneck we saw with the OpenGL renderer.

In developing the software renderer, we actually wrote three versions. The pure-python version was the initial prototype, and serves as the reference implementation for the unit tests. The C++ version came next, and it defines the data types necessary for transferring information to and from the rest of the system. Finally, we have the highly tuned CUDA version which actually does the heavy lifting. Having multiple implementations increases the development overhead. However, having a Python-C hybrid allows us to maintain unit tests to ensure correctness. We also can use Python's reference counting mechanism in order to maintain the lifetime of GPU memory resources.

2.4.1 Hand Geometry and Distance Transform

Since we were writing our own renderer from scratch, we were not constrained to the primitives and operations offered by OpenGL. Instead, we use a simple routine renderer designed to quickly render the relatively simple geometry of the hand. It is geared towards generating a silhouette and at the same time computing the distance transform. As a result, we build the hand image from a small set of geometric primitives, rather than completely from polygons. This simplifies the rendering pipeline, as there is no longer a need to perform polygonization. We also chose our primitives to simplify distance transform computation.

Like the OpenGL renderer of Section 2.3, the software rendering engine starts with a the absolute position and orientation of the palm and the set of joint-angles. It uses this information to generate a 3D "skeleton" of the hand. This produces a line segment for each phalanx, and two large polygons describing the palm. We then project this skeleton into each of the camera views.

Ideally, we would like to render each of the phalanxes as a cylinder with hemispherical end caps. The perspective projection of that 3D shape would have non-parallel edges and distorted caps. However, to simplify the mathematics, we use an orthographic approximation, which is a rectangle capped by a semicircle. That is, we represent the phalanx with a

"capsule" that consists of the set of all points which are less than a distance r from the 2D skeleton line segment. Note that we used a perspective projection when converting 3D skeleton segments into 2D segments. We only resort to the orthographic approximation afterward. Also, for simplicity, we currently keep the width of the phalanx a constant, whereas it should be modulated by the distance to the camera. In practice, however, these approximations work well when the hand is relatively far away from the camera.

In a similar fashion, we render the palm as two "rounded polygon" primitives. Each of these is the set of all points which are less than a distance r from the interior of the 2D projection of the polygon. This is approximately the 3D shape associated with all the points that are within r of the 3D polygon. Sectioning the palm into two pieces helps with the hand fold. One quadrilateral section extends from the two wrist keypoints to the MCPs of the pinkie and index finger. A second triangular section runs from the MCP of the index finger to the MCP of the thumb to the radial wrist keypoint. This fold does not quite line up with the "hand fold" joint, and as a result it is possible for the quadrilateral to become nonconvex under extreme circumstances. In practice, however, this has not been an issue.

To actually produce the rendering, we define a function $d_{c_i}(\mathbf{p})$ for each capsule c_i. This is defined to be the distance skeleton line segment s_i minus its radius r_i. Note that we now have an implicit representation of capsule i, with $d_{c_i}(\mathbf{p}) < 0$ for the interior of the capsule. We produce a similar function of the palm polygon $d_p(\mathbf{p})$. We render the pixel $R(\mathbf{p})$ by

$$R(\mathbf{p}) = \min\left(\min_i d_{c_i}(\mathbf{p}), d_p(\mathbf{p})\right). \tag{2.3}$$

This yields an implicit representation of the hand; $R(\mathbf{p}) < 0$ on the interior of the hand. Since we performed all the distance computations in 2D, $R(\mathbf{p})$ also happens to be the value of the distance transform for points outside the hand.

2.4.2 Performance

Unlike the OpenGL renderer, which only touches the regions on or near the hand, the software renderer performs the same computation for every pixel of the output. As a result, it performs significantly more computation. However, because the computation is much more regular and because we are able to take advantage of the on-chip cache, the software renderer significantly outperforms the OpenGL renderer.

We benchmarked this algorithm on two machines. One is a development desktop which has an Intel Core2 Quad with an NVIDIA GeForce 9800 GX2. The other is a node of a cluster, which has two dual-core AMD Opterons attached to an NVIDIA Tesla S1070 containing four GPUs. In both cases, we use exactly one CPU core and one GPU. Table 2.2

Table 2.2: Timing of various components. The values given are the times needed for one comparison.

System	Drawing Only (μs)	CPU Points (μs)	GPU Points (μs)	Full Tracker (μs)
Core2 Quad Q6600 + G92 (GeForce 9800 GX2)	49.4	58.6	50.7	52.7
Opteron 2216 + GT200 (Tesla S1070)	31.1	71.0	31.7	35.0

summarizes the results. The initial system only performed the drawing step on the GPU. The task of finding the vertexes for the capsules and the polygons remained on the CPU. The drawing part alone runs fast, achieving 20,000 comparisons per second on the development desktop and 32,000 comparisons per second on a cluster node. However, as the second column of the table shows, the point calculations slow things down, leading to a noticeable decrease in performance on the desktop system. However, the performance drops off dramatically on the cluster node, which features a slower CPU and a faster GPU. Moving the point calculation onto the GPU restores the lost performance. As the last column suggests, the rendering portion dominates the overall tracker time. Thus, while it is possible to move more of the tracker to the GPU, we will begin to suffer diminishing returns.

2.5 Solver

Introduced to the vision community by Isard et al. as the CONDENSATION [15] algorithm, particle filters have proved to be a successful method of tracking complex states. Unlike Kalman filters and its nonlinear extensions, which are based on unimodal Gaussians, the particle filter can represent an arbitrary probability distribution. As a result, it can maintain multiple hypotheses about the target state, which proves useful when there can be ambiguities in the observations. In this section, we give a brief review of the sequential importance sampling (SIS) filter, structured on the tutorial paper by Arulampalam et al. [29]. We follow with a description of the parametrization that we have chosen for our implementation.

Every tracking problem is characterized by two models. The first describes how the system state evolves over time:

$$\mathbf{x}_k = \mathbf{f}_k(\mathbf{x}_{k-1}, \mathbf{v}_{k-1}). \tag{2.4}$$

This function describes how the state sequence $\{\mathbf{x}_k\}$ evolves over time under the influence

of an independently identically distributed (i.i.d.) noise sequence $\{\mathbf{v}_k\}$. The second equation

$$\mathbf{z}_k = \mathbf{h}_k(\mathbf{x}_k, \mathbf{n}_k) \tag{2.5}$$

describes how $\{\mathbf{z}_k\}$, the sequence of our observations, relates to the underlying $\{\mathbf{x}_k\}$, possibly corrupted by a (different) i.i.d. noise $\{\mathbf{n}_k\}$. The tracking problem then reduces to determining $\{\mathbf{x}_k\}$ given $\{\mathbf{z}_k\}$. In particular, we usually want to find $\mathbf{x}_k$ given $\mathbf{z}_{1:k} = \{\mathbf{z}_i, i = 1, \ldots, k\}$, the set of all observations until time k.

For tracking, we commonly make the Markov assumption $p(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{z}_{1:k-1}) = p(\mathbf{x}_k|\mathbf{x}_{k-1})$. That is, once we "know" $\mathbf{x}_{k-1}$, we have extracted all available information from the previous observations $\mathbf{z}_{1:k-1}$. With this, the tracking problem breaks down into two steps. The first step, "propagates"the system state from the last time frame to the present.

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

The model of state evolution $p(\mathbf{x}_k|\mathbf{x}_{k-1})$ is determined by (2.4) and by the known statistics of $\mathbf{v}_{k-1}$. The second step, "updates" the current estimate with the newly arrived observation.

$$p(\mathbf{x}_k|\mathbf{z}_{1:k}) = \frac{p(\mathbf{x}_k|\mathbf{z}_k) p(\mathbf{x}_k|\mathbf{z}_{1:k-1})}{p(\mathbf{z}_k|\mathbf{z}_{1:k-1})}. \tag{2.7}$$

The denominator of the fraction $p(\mathbf{z}_k|\mathbf{z}_{1:k-1})$ is a normalizing constant which is very difficult to compute directly. Fortunately, finding its value is not necessary for our filter.

The SIS filter works by representing $p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})$ by a set of N_s particles $\{\mathbf{x}^i_{0:k}, w^i_k\}_{i=1}^{N_s}$. For every i, $\mathbf{x}^i_{0:k}$ corresponds to a possible evolution of the state from initial time to k. The weight w^i_k is proportional to the probability that the corresponding $\mathbf{x}^i_{0:k}$ represents the true trajectory. The weights are normalized so that $\sum_i w^i_k = 1$. As a result, we have the approximation

$$p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k}) \approx \sum_{i=1}^{n} w^i_k \delta(\mathbf{x}_{0:k} - \mathbf{x}^i_{0:k}). \tag{2.8}$$

With the mathematical derivation detailed in [29], it can be shown that

$$w^i_k \propto w^i_{k-1} \frac{p(\mathbf{z}_k|\mathbf{x}^i_k) p(\mathbf{x}^i_k|\mathbf{x}^i_{1:k-1})}{q(\mathbf{x}^i_k|\mathbf{x}^i_{0:k-1}, \mathbf{z}_{1:k})} \tag{2.9}$$

where $q(\mathbf{x}^i_k|\mathbf{x}^i_{0:k-1}, \mathbf{z}_{1:k})$ is an arbitrary distribution from which we draw $\mathbf{x}^i_k$. We choose $q(\mathbf{x}^i_k|\mathbf{x}^i_{0:k-1}, \mathbf{z}_{1:k}) = p(\mathbf{x}^i_k|\mathbf{x}^i_{1:k-1}) = p(\mathbf{x}^i_k|\mathbf{x}^i_{k-1})$, causing the denominator to cancel with the numerator.

Algorithm 1 A SIS particle filter without resampling.

Given $\{\mathbf{x}_{k-1}^i, w_{k-1}^i\}$ and $\mathbf{z}_k$
for $i = 1$ to N_s **do**
  draw $\mathbf{x}_k^i$ from $p(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i)$
  $w_k^i \leftarrow w_{k-1}^i p(\mathbf{z}_k|\mathbf{x}_k^i)$
end for
normalize w_k^i so that $\sum_i w_k^i = 1$

Algorithm 2 An efficient resampling algorithm.

Given $\{\mathbf{x}_k^i, w_k^i\}$
$c_0 \leftarrow 0$
for $i = 1$ to N_s **do**
  $c_i \leftarrow c_{i-1} + w_k^i$
end for{c_{N_s} should be 1 because w_k^i are normalized}
for $i = 1$ to N_s **do**
  draw r uniformly from $[0, 1)$
  use binary search to find s such that $c_{s-1} \leq r < c_s$
  $\hat{\mathbf{x}}_k^i \leftarrow \mathbf{x}_k^s$
  $\hat{w}_k^i \leftarrow 1/N_s$
end for

This gives us Algorithm 1. It will work initially, but we will eventually choose a $\mathbf{x}_k^i$ so that $p(\mathbf{z}_k|\mathbf{x}_k^i)$ is very low. After that point, we devote computational effort to updating that particular trajectory without providing any benefit to our estimate of $p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})$. Eventually, we find a poor choice of $\mathbf{x}_k^i$ for every i, and the tracker loses effectiveness.

To avoid this fate, we introduce resampling. This step replaces $\{\mathbf{x}_k^i, w_k^i\}$ with $\{\hat{\mathbf{x}}_k^i, \hat{w}_k^i\}$, concentrating the particles with larger weights. To do this, we draw samples from the discrete distribution of

$$p(\mathbf{x}_k|\mathbf{z}_{1:k}) \approx \sum_{i=1}^{n} w_k^i \delta(\mathbf{x}_k - \mathbf{x}_k^i). \tag{2.10}$$

That is, we choose $\hat{\mathbf{x}}_k^j \leftarrow \mathbf{x}_k^i$ with probability proportional to w_k^i. Since we performed a set of independent draws from (2.10), $\hat{w}_k^i = 1/N_s$ after the update. Algorithm 2 illustrates an efficient method of performing this update. Although it might be better to wait until certain conditions are met, we resample after every iteration. This makes our approach consistent with the sampling importance resampling (SIR) algorithm described in [29].

In our specific implementation of the particle filter, $\mathbf{x}$ is the 27-dimensional hand pose vector from Section 2.1.1. Our observations are $\mathbf{z}_k = \{\mathbf{c}_c\}_{c \in \mathcal{C}}$, the set of images from our

cameras. Then, we define

$$p(\mathbf{z}_k|\mathbf{x}_k^i) \propto e^{-\gamma\left[\sum_{c\in\mathcal{C}} d_{\text{cfr}}\left(r_c(\mathbf{x}_k^i),\mathbf{c}_c\right)\right]}. \tag{2.11}$$

In this equation $r_c(\mathbf{x}_k^i)$ is the rendering of $\mathbf{x}_k^i$ from the viewpoint of camera c and $d_{\text{cfr}}(\cdot,\cdot)$ is the chamfer distance. The parameter γ is a scaling constant, which we usually default to 0.5.

There is one other modification we make to the standard SIR algorithm. Instead of updating the tracker once per frame, we feed the same camera images to the tracker several times. The level of repeat is controlled by the `repeat` parameter. This allows the tracker "more time" to catch up to the motion in the camera. However, by feeding the same information multiple times, we are implying to the tracker an observation error that is lower than what the data suggests. This means that the tracker will underestimate the variance in the hand state. The mean, however, should still provide useful data.

Algorithm 3 summarizes our tracker. Note that the nested loops of lines 11–15 are highly parallelizable. In particular, all of the distance calculations of line 13 are independent of each other. Note that the outer loop divides the work over cameras. This should make it easy to process each camera image on a different GPU or on a different node of a cluster, making the system relatively scalable in terms of handling cameras. There is still communication necessary to distribute the $\mathbf{x}_k^i$ at the beginning of every iteration and to collect the $d_{c,i}$ at the end. The amount of data to be exchanged is relatively small, but there will be synchronization overhead.

Algorithm 3 The hand tracker.

Given initial state $\mathbf{x}_0$
for $i = 1$ to N_s **do**
  $\mathbf{x}^i \leftarrow \mathbf{x}_0$ {We resample after every step so we don't need to keep track of the weights.}
end for
while more frames **do**
  Acquire camera images $\{\mathbf{c}_c\}_{c \in \mathcal{C}}$
  for $r = 1$ to `repeat` **do**
    for $i = 1$ to N_s **do**
      draw $\mathbf{x}_k^i$ from $p(\mathbf{x}_k^i | \mathbf{x}_{k-1}^i)$
    end for
    for all $c \in \mathcal{C}$ **do**
      for $i = 1$ to N_s **do**
        $d_{c,i} \leftarrow d_{\text{cfr}}\left(r_c(\mathbf{x}^i), \mathbf{c}_c\right)$
      end for
    end for
    for $i = 1$ to N_s **do**
      $w_k^i \leftarrow e^{-\gamma\left[\sum_{c \in \mathcal{C}} d_{c,i}\right]}$
    end for
    normalize w_k^i so that $\sum_i w_k^i = 1$
    resample according to Algorithm 2
  end for
end while

Chapter 3

Data

3.1 Data Collection

In order to collect image data for training and testing, we use the camera rig pictured in Figure 3.1. Lighting was provided by two umbrella lights and overhead fluorescent lighting. We used five Point Grey Dragonfly cameras connected by FireWire 400 to a central data-collection computer. The cameras are synchronized by the FireWire bus and operate at 30 frames per second. They operate at a 640×480 resolution. The cameras are color, but that is implemented with a Bayer filter, which does not increase the number of independent samples generated by the camera. Still, the five cameras combined produce more data than the disk write throughput. However, our sequences are relatively short, and the capture machine has sufficient memory to buffer the frames. No frames were dropped during the capture. An example of the camera output is shown in Figure 3.2.

For our test and training cases, we use the three gestures from the childhood game rock-paper-scissors. Each of the training and test sequences consists of a single take showing 60 transitions from one state to another. The sequences were designed to ensure that each of the 6 possible transitions occur 10 times each. We annotate every five frames with a label

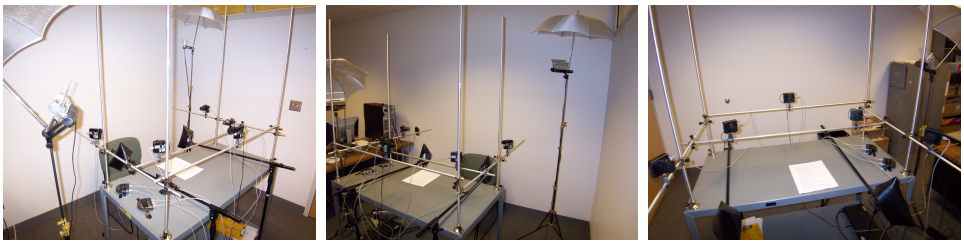
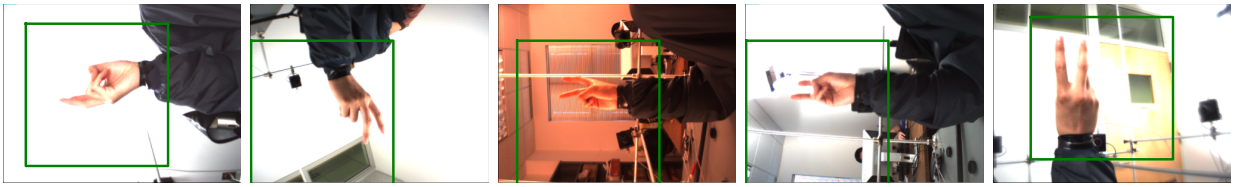


Figure 3.1: The camera rig used to capture our training and test sequences. (The cameras were in slightly different locations when the sequences were captured.)



(0) (1) (2) (3) (4)

Figure 3.2: Raw camera output (after Bayer filtering) for frame 120 of the training set. The boxes indicate the crop location.

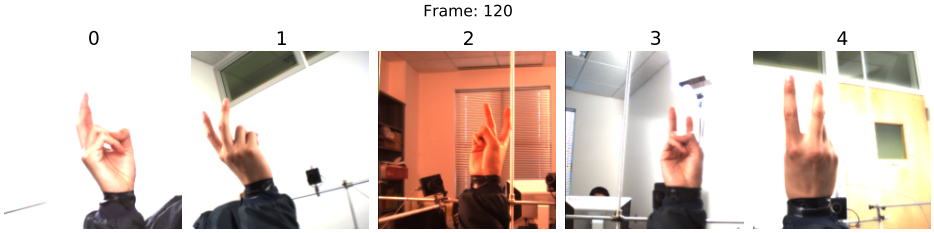


Figure 3.3: Sample frame of the training sequence after cropping and rotation.

of "rock," "paper," "scissors," or "transition." This allows us to compute the evaluation described in Section 4.1.

The test sequence was shot on the same day as the training sequence. The cameras were not moved between the shots. Care was taken to provide a clear color boundary at the wrist and to ensure that other skin-colored regions (such as the face) were not in the frame. The cameras were equipped with a manual zoom lens, and we adjusted the positions and zoom so that the hand mostly remained within the view throughout the sequence. We report frame numbers as they are recorded in the video. The actual training sequence starts at frame 120 and runs to frame 1800. Also, in true computer science fashion, the cameras are numbered 0–4 instead of 1–5.

We do some preprocessing on the frames before feeding them to the rest of the system. First, we rotate the image to be upright, and we crop to a 384×384 square. This is shown in Figure 3.3. We use these images as a basis for manual labeling. For the actual tracker, however, we need to perform color segmentation in order to obtain a silhouette. We do this using manually trained color histograms. The histograms are in the red-green-blue (RGB) color space, and we use 64 bins for each dimension. Because of the relatively simple background, the color segmentation is fairly straightforward. However, the lighting conditions for each camera are different, so we use separate histograms for each camera. The results of the color segmentation are shown in Figure 3.4.

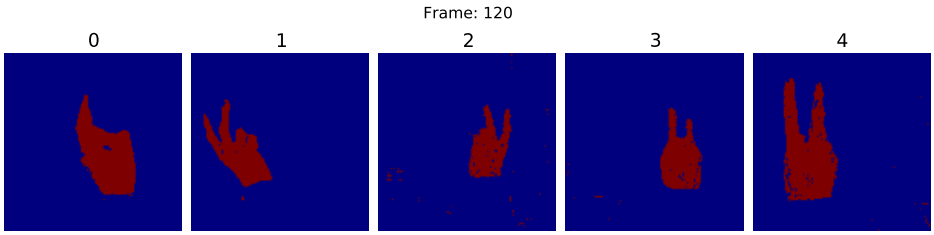


Figure 3.4: Sample frame of the training sequence after color segmentation.

3.2 Keypoints

In order to perform the different calibration steps in Section 3.3 and Section 3.4, we need to be able to identify keypoints of the hand. We use four points associated with each finger, corresponding to the MCP, PIP, DIP, and fingertip. (For the thumb, we use the CMC, MCP, PIP, and thumb tip.) Note that the MCP joints of the fingers are closer to the knuckle on the back of the hand than to the crease at the edge of the palm at the front of the hand. In addition, we have two keypoints near the wrist to indicate the base of the palm.

In order to streamline the labeling process, we developed a python-based labeling tool, depicted in Figure 3.5. This tool allows us to drag the keypoints so that they align with the image. Using matplotlib's plotting facility, the tool can zoom and pan the image to provide a more accurate view. It also allows us to set a confidence value for every point. We label points that are not visible in a particular view with a large circle, indicating greater uncertainty. The "Solve Points" uses a nonlinear solver to ensure that the points are consistent with the current camera model. The "Solve Pose" button shifts the points so that they are consistent with the current camera and shape model.

Using this tool, we label every 15 frames of the training set (113 frames in total). We use certain key frames (see Section 3.3) to calibrate the camera. For the remainder of the frames, we use the "Solve Points" button to ensure that our labeling is consistent in 3D.

3.3 Camera Calibration

We model the camera as a simple pinhole with the image centered around the principal point. We do not take into account radial distortion or even the fact that our cropped images are actually off-center relative to the lens axis.[1] These distortions are relatively minor, and our simplifications correspond to how the renderer actually generates the image. We also assume square pixels with no skew, and therefore the intrinsic parameters reduce to a single

[1]The OpenGL renderer can handle off-center projection matrices. The software renderer currently lacks that capacity, although it would be relatively easy to add.

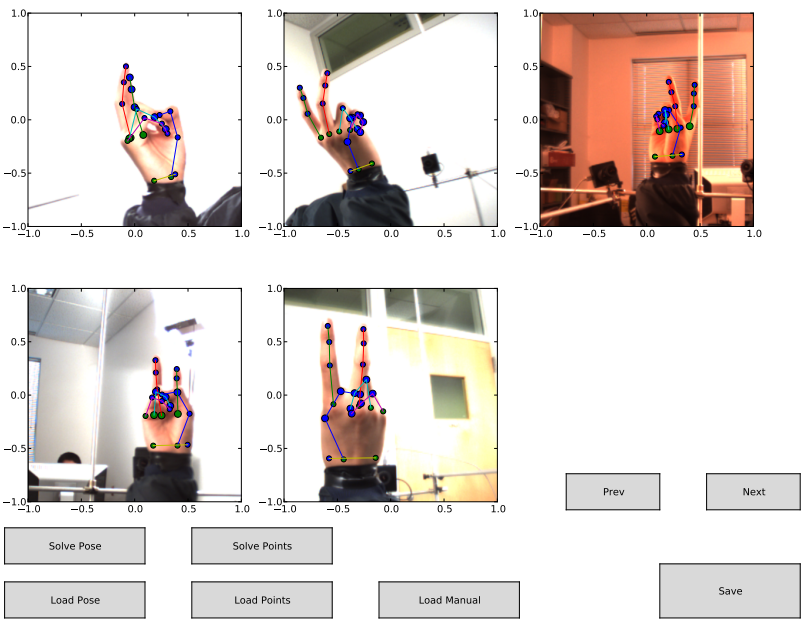


Figure 3.5: Screen shot of the keypoint labeling tool. Large circles indicate high uncertainty in the label while small circles indicate high confidence.

field-of-view value. However, that value is conflated with the position of the camera, so we can simply fix it to a constant (corresponding to about 28°). Therefore, our camera model has only the six extrinsic parameters (three dimensions of position and three dimensions of rotation).

There is no canonical center for the center (origin) of the world coordinate system, so we arbitrarily define that to be the location of camera 0. Without an absolute reference, the camera parameters can only be recovered up to a constant scale factor. This scale factor cancels out in the rendering, but to keep things consistent, we choose our scale so that the PIP of the index finger is 5 units long. That makes one unit in world coordinates approximately 1 cm long.

In order to solve for the camera parameters, we manually labeled keypoints from six frames (120, 285, 435, 480, 495, 525) of the video. Let $\mathbf{x}_{c,i}^{f}$ be the location of the i^{th} keypoint in camera view c of frame f. Let $w_{c,i}^{f}$ be the corresponding confidence-based weight. We weigh points that are visible 10 times more than points that are occluded. Then we want to solve for the camera parameters θ_c of camera c. In the process, we need to find the location of the keypoints in world coordinates. Let us denote the 3D location of keypoint i of frame

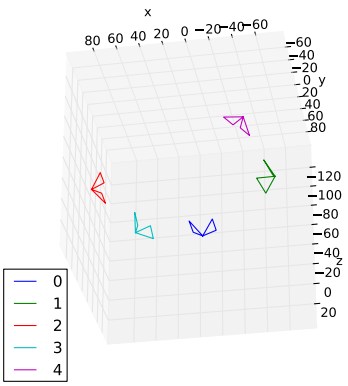


Figure 3.6: Solved camera parameters. Note that the virtual locations do not correspond with physical locations because the cameras had different focal lengths.

f as $\mathbf{w}_i^f$. Then, we formulate an unconstrained nonlinear least squares problem:

$$\min \sum_f \left(\left(\sum_c \sum_i w_{c,i}^f \| C(\mathbf{w}_i^f, \theta_c) - \mathbf{x}_{c,i}^f \|^2 \right)^2 + \left(\|\mathbf{w}_4^f - \mathbf{w}_5^f\|^2 - 25 \right) \right) \tag{3.1}$$

where $C(\mathbf{w}, \theta)$ projects the 3D point $\mathbf{w}$ using camera parameters θ. The first term matches the camera calibration the camera parameters match the labeled points. The second term helps maintain the scale by ensuring the distance between the base of the index finger and its DIP in 3D is five. We could use a scale factor for this regularization term. However, since it should be possible to drive the term to close to zero, we leave the scale at one.

After obtaining the initial solution, we use our labeling tool to inspect the results. After a few rounds of interactive tweaking, we were able to produce results which look visually consistent and yield a value of zero for (3.1). These results are illustrated in Figure 3.6.

3.4 Hand Shape Calibration

The fixed parameters of our hand model were described in Section 2.1.2. We denote these as h and presume that they are constant across the frames of a sequence and across the training and test set. In to estimate h, we use the 113 frames that we labeled (Section 3.2).

To solve for the hand parameters, we take a multi-step approach. First, we initialize an initial estimate of the hand parameters $\hat{h}$. For the phalanx lengths, we use the mean of the distances between the appropriate 3D points. For the other parameters, we use some manually initialized values. We then solve the pose $\mathbf{p}^f$ individually for each frame f, using

the unconstrained optimization problem

$$\min \left(\sum_c \sum_i w_{c,i}^f \| C(P_i(\mathbf{p}^f; \hat{h}), \theta_c) - \mathbf{x}_{c,i}^f \|^2 \right) + K^{\text{pose}}(\mathbf{p}^f). \tag{3.2}$$

The first term of the equation is similar to the first term in (3.1). As before, $\mathbf{x}_{c,i}^f$ is the i^{th} labeled keypoint of camera c of frame f, and $w_{c,i}^f$ is its corresponding weight. Also as before, $C(\cdot, \theta)$ projects a 3D point using θ as the camera parameters. Instead of solving for a point $\mathbf{w}$, however, we use the result of $P_i(\mathbf{p}; h)$. This function returns the world coordinate of the i^{th} keypoint given pose $\mathbf{p}$ and hand parameters h. The function K^{pose} is a soft constraint on the pose. For each joint, we define an upper (u_j) bound and lower (l_j) bound, as shown in Table 3.1. We selected these values manually, ensuring that no frame of the final solution exceeds the bound by more than 10. We also make sure that the same joints on different fingers have the same bound. Then, let p_j be the element of $\mathbf{p}$ corresponding to joint j, and

$$K^{\text{pose}}(\mathbf{p}) = \sum_j (0.01 k_j(p_j))^2 \tag{3.3}$$

where

$$k_j(p_j) = \begin{cases} p_j - l_j & p_j < l_j \\ u_j - p_j & p_j > u_j \\ 0 & \text{otherwise.} \end{cases} \tag{3.4}$$

Note that the formulation of K^{pose} ensures that the overall optimization remains a nonlinear least squares.

Finally, we solve for h, along with all the poses $\mathbf{p}^f$ simultaneously:

$$\min \sum_f \left(\left(\sum_c \sum_i w_{c,i}^f \| C(P_i(\mathbf{p}^f; \hat{h}), \theta_c) - \mathbf{x}_{c,i}^f \|^2 \right) + K^{\text{pose}}(\mathbf{p}^f) \right) + K^{\text{shape}}(h). \tag{3.5}$$

The soft constraint $K^{\text{shape}}(h)$ contains several terms. It removes the ambiguities introduced in the hand's local coordinate system. To do this, we ensure that the base of the thumb at the origin and that the segment between the two wrist points are parallel to the local x axis. We also ensure that the two rotation axes at the base of the thumb are perpendicular to each other. In addition, we make sure that the MCP flexion, PIP, and DIP axes of the fingers (not the thumb) lie in the positive-x region of the xz plane. Similarly, we force the axis for MCP adduction to lie in the positive-x region of the xz plane.

Table 3.1: Soft constraints used for solving the hand pose. These are the values used in (3.4).

j	Upper bound (u_j)	Lower bound (l_j)
0	70	0
1	85	0
2	115	-20
3	60	0
4	130	-40
5	140	-20
6	105	-45
7	60	-40
8	130	-40
9	140	-20
10	105	-45
11	60	-40
12	130	-40
13	140	-20
14	105	-45
15	60	-40
16	130	-40
17	140	-20
18	105	-45
19	60	-40
20	20	-45
21	180	-180
22	180	-180
23	180	-180
24	20	-10
25	5	-20
26	-40	-80

3.5 Ground Truth Pose Data

As described in Section 3.2, we manually labeled every 15 frames of the training data. When we solved for the hand shape in Section 3.4, we also obtained pose information for those frames. However, in order to train a motion model (Section 4.3.3) or a classifier for evaluation (Section 4.1), we need labels for every frame of the sequence. To generate these labels, we employ our general purpose particle filter. However, since the intent is not to evaluate tracker performance, we make multiple passes through our sequence and we add additional terms to enforce regularity.

The underlying basis for solving for the pose of a particular frame is our particle filter with hard constraints, described in Section 4.3.2. (Although we do impose the constraints during the solver pass, the final results are usually far away from the bounds. Thus, the constraints probably did not inhibit the final solution.) However, we operate this particle filter more like an annealing solver which minimizes an objective function. To that end, we model the likelihood as

$$L \propto \exp -0.1 \left[\sum_c d_{\text{cfr}} \left(r_c(\mathbf{p}^f), \mathbf{c}_c^f \right) + \|\mathbf{p}^f - \mathbf{p}^{f-1}\|^2 + \|\mathbf{p}^f - \mathbf{p}^{f+1}\|^2 + \sum_c \sum_i \|C(P_i(\mathbf{p}^f; \hat{h}), \theta_c) - \mathbf{x}_{c,i}^f\|^2 \right] \quad (3.6)$$

where $\mathbf{p}^f$, the pose of frame f, is what we are solving for. In this formula, the first term represents the match between rendering of $\mathbf{p}^f$ and the camera images. The next two terms ensure that $\mathbf{p}^f$ remains close to the pose of the previous ($\mathbf{p}^{f-1}$) and next ($\mathbf{p}^{f+1}$) frames. The appropriate term is omitted if f is at the beginning or end of the sequence. The final term compares the keypoints associated with $\mathbf{p}^f$ to the manual labels $\mathbf{x}_{c,i}^f$. This term is omitted if there are no manual labels for that frame.

To actually perform the optimization, we treat each frame individually. For frame f, we initialize the tracker with the previous solution of frames $f-2$ through $f+2$. (We initialize with only the available frames near the boundaries.) We then run the tracker for 20 iterations and look at the weighted mean of the result. We only update the result for the frame if the new result has a better score than the old. We then make multiple passes through the frames forwards and backwards until the results roughly converge. Sample results from this process are shown in Figure 3.7 and Figure 3.8. Note that frames 255 and 270 have manual point labels while the intermediate frames rely on the solver results.

Figure 3.7: Sample sequence of ground truth tracking results (part 1).

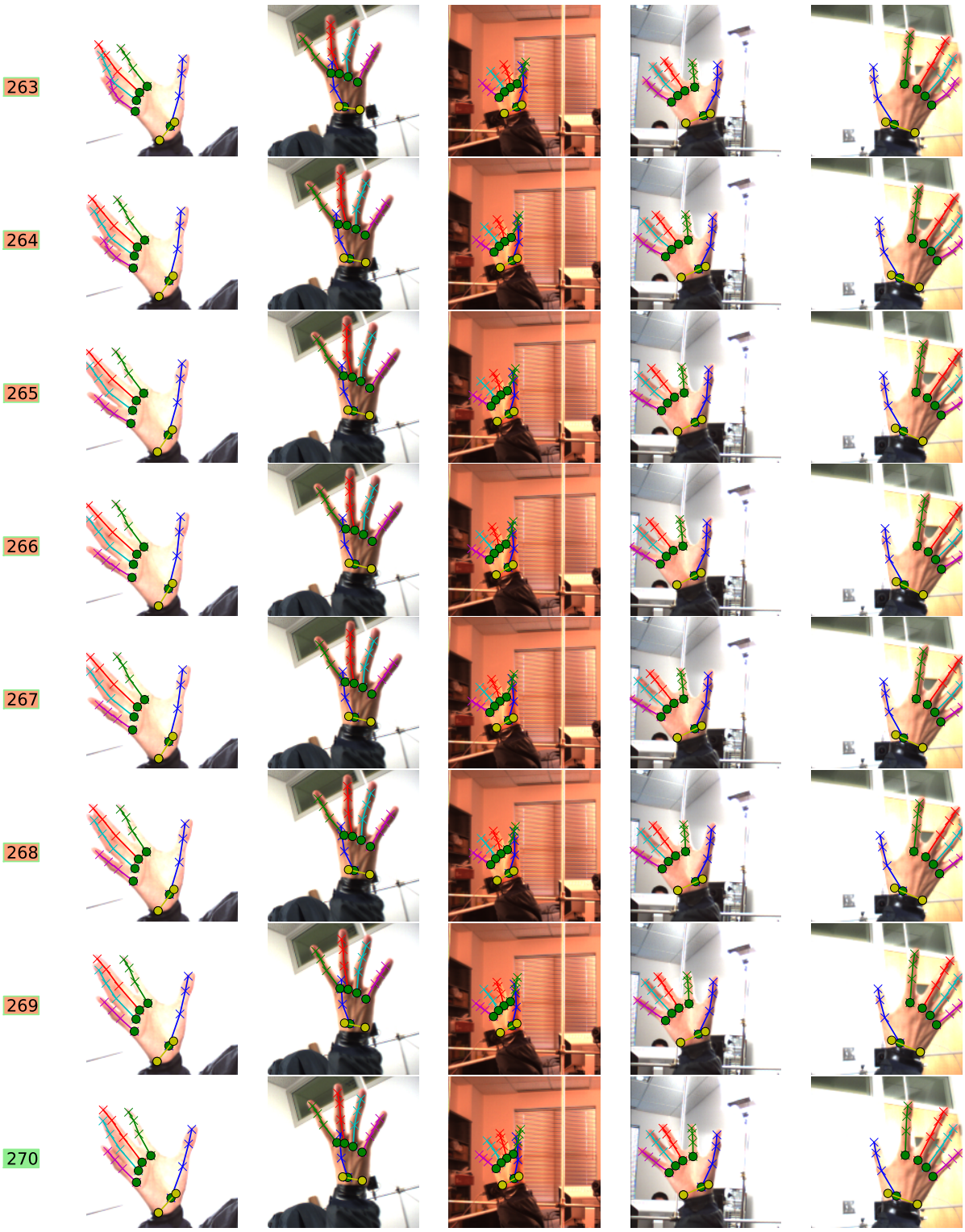


Figure 3.8: Sample sequence of ground truth tracking results (part 2).

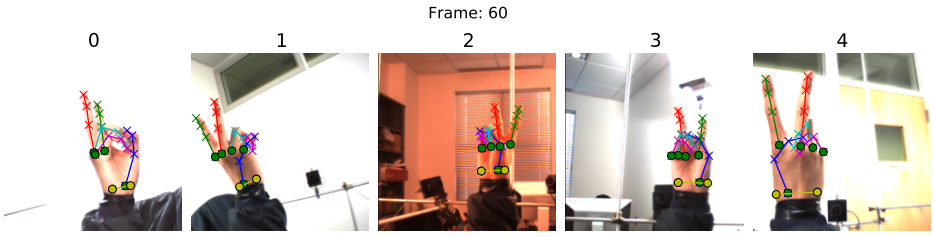


Figure 3.9: First frame of the test sequence after cropping and rotation. This shows the manually initialized pose for tracker initialization.

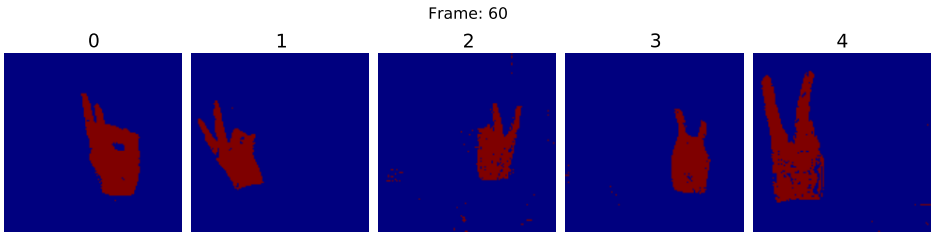


Figure 3.10: First frame of the test sequence after color segmentation.

3.6 Test Data

The test sequence was recorded on the same day as the training sequence under similar conditions. This starts at frame 60 of the video and runs to frame 1800. Figure 3.9 shows the first frame of the test sequence. Although there appeared to be slight lighting variations (possibly due to the differences in motion), we reuse the histograms that we used for color segmentation of the training sequence on the test sequence. The resulting segmentation, illustrated in Figure 3.10, while not perfect, is sufficient for our needs. This sequence is qualitatively very similar to the training sequence, although the hand is rotated slightly differently, and it swings in a somewhat different direction.

We do not annotate the test sequence to the same degree as the training sequence. We label the keypoints for only the first frame, illustrated in Figure 3.9. This labeling is consistent with the camera and hand shape of the training sequence, and is used to solve for the initial pose. We also annotate "rock," "paper," "scissors," or "transition" every five frames. This allows us to perform the evaluation described in Section 4.1.

Chapter 4

Classifier Experiments

In this chapter we describe the experimental results of running our tracker on the data. First, we discuss our evaluation metric, which is based on the output of a multiclass classifier. We then consider the results of running various forms of the tracker on the full data set. Finally, we consider various corruptions of the original data, including reduced framerates, fewer cameras, and distorted hand shapes.

4.1 Evaluation Metric

Determining an appropriate evaluation metric was a minor challenge. We experimented with using Mahalanobis distance, distance based on the covariance of the tracker particles, and raw chamfer distance. The main issue is that the there is a high degree of ambiguity in the images. As a result, it is difficult to judge when the error of a particular tracker output is "too large." It is also difficult to accurately compare trackers using different subsets of the cameras or with different numbers of particles in its particle filter. In the end, we settled on a classification-based score. For this, train a classifier to distinguish among "rock," "paper," and "scissors" using the labeled training set. We then evaluate that classifier using the joint-angle data from the tracker output. The score is the percent accuracy of these classification results compared against the hand labeled data. This metric is relatively forgiving, as the forced-choice classifier will always return one of the three possible responses. A non-functioning tracker will yield a score of 33%. However, it is highly improbable for a tracker to obtain a high score without maintaining some degree of accuracy.

We used support vector machines (SVMs) as our classifiers, as implemented by LIBSVM [30]. We performed no normalization of the pose data at all before feeding it into the classifier, although we excluded poses that were labeled "transition." We chose radial basis functions (RBFs) as the kernel function for our SVM. This nonlinear kernel has the form

of $K(\mathbf{x}, \mathbf{y}) = e^{-\gamma\|\mathbf{x}-\mathbf{y}\|^2}$ and we set $\gamma = 0.0001$ for our experiments. We set the penalty parameter C of the SVM to 10, but that should be of minor importance since the final classifier can perfectly separate the training data.

The SVM classifier is normally a binary classifier. However, there are several ways of adapting it to handle multiple classes. The first is the "one-versus-one" method, which is the default for LIBSVM. Under this scheme, we train three classifiers, one for "rock-versus-scissors," one for "rock-versus-paper," and one for "paper-versus-scissors." At evaluation, we run the test data through all three classifiers. Each classifier casts a vote for a particular pose. For example, if the "rock-versus-scissors" classifier returns scissors, the "rock-versus-paper" returns rock, and the "paper-versus-scissors" return scissors, the overall result is scissors. In the case of inconsistent results, the classifier returns the first choice, which is "rock" in our case.

An alternative method of adapting the SVM to handle multiple classes is to use the "one-versus-all" technique. Under this method, we also train three classifiers. However, these classifiers are "rock-versus-not-rock," "paper-versus-not-paper," and "scissors-versus-not-scissors." At evaluation time, we run the test data through all three classifiers. In theory, exactly one of the classifiers will return positive, indicating the final result. In the case of ambiguity, we use the raw SVM score as a confidence measure, choosing the class corresponding to the highest score.

We evaluated both of these alternatives, along with a version which uses only the local components of the pose data. (It ignores the three global translations and the three global rotation parameters of the palm.) These versions yield similar results, with the "one-versus-one" version using the full pose information performing slightly better than the others. This is the version that we will use for the rest of this work.

Looking at the training data, one can see that the classification task is very simple. Figure 4.1 shows the data projected into 2D using the ISOMAP [31]. ISOMAP is a nonlinear dimension reduction technique which tries to preserve the geodesic distance between points. The large markers indicate the ground truth labeling. Note that all the points associated with one pose are close to each other and distant from the other poses. While the distances are distorted by the projection from the original high-dimensional space to 2D, the figure suggests that the poses are easily separable. With our choice of a RBF kernel with a very low value of γ, the SVM essentially becomes a nearest neighbor classifier. This is indicated by the small dots, which illustrate the SVM classifier results. These show that the decision boundary is roughly halfway between the clusters. As a result, we expect that almost all classification schemes will yield similar results.

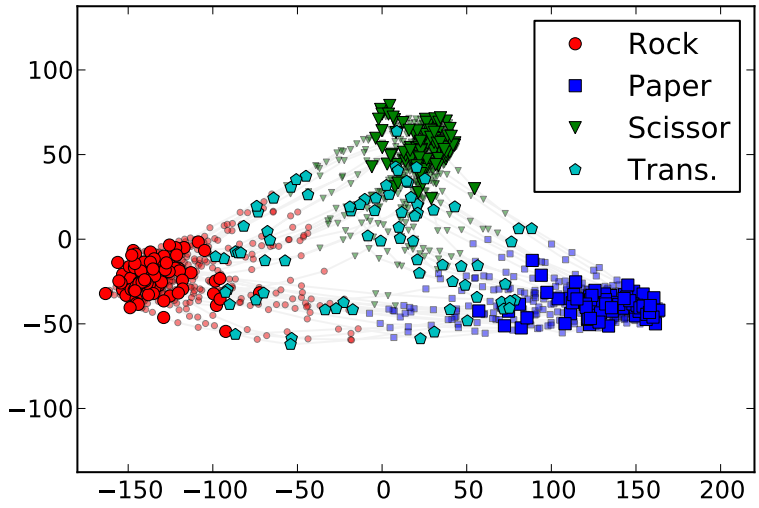


Figure 4.1: Overview of SVM classification. This figure shows the training data projected into two dimensions using ISOMAP. The large markers indicate the manual labels used to train the SVM. The smaller markers represent the remainder of the data, and they are drawn according to the result of the SVM.

4.2 Visualizing Results

In the following discussions, we will be comparing tracker performance under varying conditions. Since a particle filter is an inherently stochastic process, we need to conduct multiple trials in order to obtain an accurate view of the performance. Unless otherwise specified, we conduct 31 runs of each setting. To visualize the results, we use box plots [32]. Figure 4.2 gives an example. This plot shows several key statistics in a clear fashion. The red line through the middle indicates the median of the data set. The top and bottom of the box represent upper and lower quartiles, respectively. The plus symbols indicate outliers, which are defined as points which lie more than 1.5 times the interquartile distance (IQR) away from the median. The IQR is the difference between the upper quartile and the lower quartile. On a normal distribution, the IQR represents about 5 standard deviations, and covers the middle 98% of the data. The "whiskers" extend to cover the range of the data values, excluding the outliers. These plots make it easy to see the median value and provide a good visual indication of the variation of the results. When comparing across three variables, it becomes too cumbersome to show box plots; in these cases, we provide tables of the median and mean values.

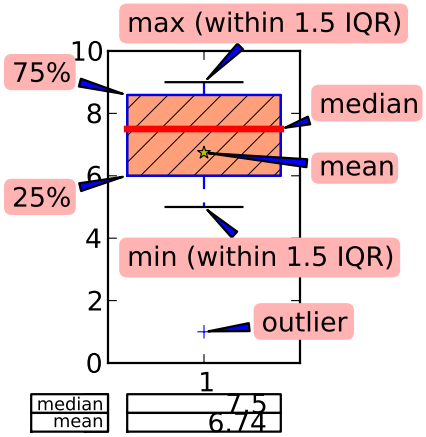


Figure 4.2: A sample box plot.

4.3 System Models

When we described Algorithm 3, we left out the exact mechanism we used for drawing new candidate poses in line 9. In this section, we describe several variations for drawing from $p(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i)$. For these tests, we use all five of our available cameras and completely accurate hand models. We also process every frame, at times computing at significantly slower than real-time speeds. The purpose of this section is to explore the consequences of the different static and dynamic hand models, as well as to tune parameters for the trackers.

4.3.1 Simple Model

The simplest pose model is to use

$$p(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i) = \mathcal{N}(\mathbf{x}_{k-1}^i, \Sigma) \tag{4.1}$$

where $\mathcal{N}(\mathbf{x}_{k-1}^i, \Sigma)$ is a multivariate normal distribution with mean $\mathbf{x}_{k-1}^i$ and a constant covariance Σ. This corresponds to a "constant position model," where the system equation of (2.4) is simply

$$\mathbf{f}_k(\mathbf{x}_{k-1}, \mathbf{v}_{k-1}) = \mathbf{x}_{k-1} + \mathbf{v}_{k-1} \tag{4.2}$$

and the "noise" term $\mathbf{v}_{k-1}$ is distributed according to the zero-mean $\mathcal{N}(0, \Sigma)$. This is also how we implement line 9 of Algorithm 3—we simply draw from $\mathcal{N}(0, \Sigma)$ and add it to $\mathbf{x}_{k-1}^i$.

As for the parameter Σ, we use

$$\Sigma = \sigma\Sigma^{\text{global}} \tag{4.3}$$

where Σ^{global} is a diagonal covariance matrix that is approximately the variance exhibited by

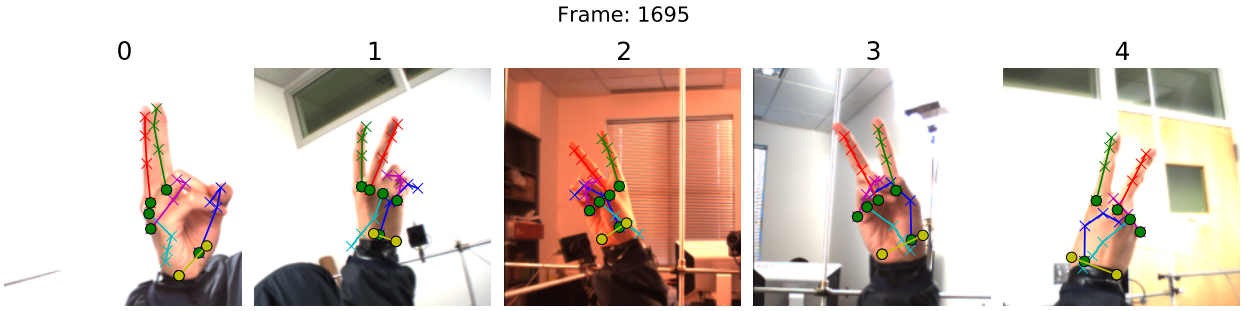


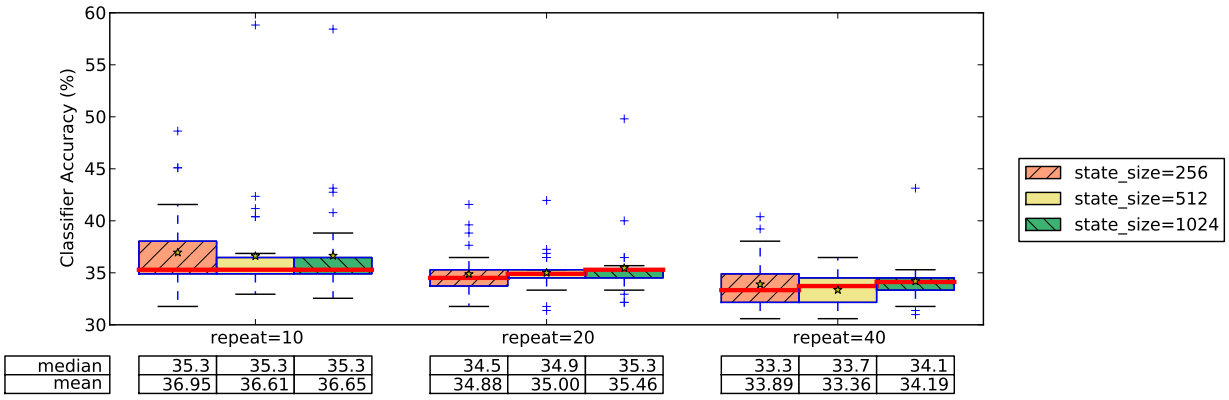
Figure 4.3: Sample frame from a tracking result with the simple model with with `repeat` = 20 and $\sigma = 0.1$. Note that although the solver gets the shape generally correct, the ring and pinkie fingers are in completely unnatural positions.

the joint-angles in our training set. The scale factor σ is a tunable parameter which defaults to 0.1.

Even this crude model is capable of roughly keeping track of the palm. However, it has a number of drawbacks. One is that it can wrap around; since we measure joint-angles in degrees, a value of 720 is the same as a value of 0. This has a tendency to produce results with very large values, and may pose problems when computing the weighted mean. More significantly, however, this model often proposes completely unrealistic hand poses, as illustrated in Figure 4.3. Note that the ring finger is bent downward to an unattainable degree. Also note that the distal phalanx of the pinkie is folded backwards. Finally, because the abduction DoF of the MCP joints are unconstrained, the palm is out of position relative to the fingers.

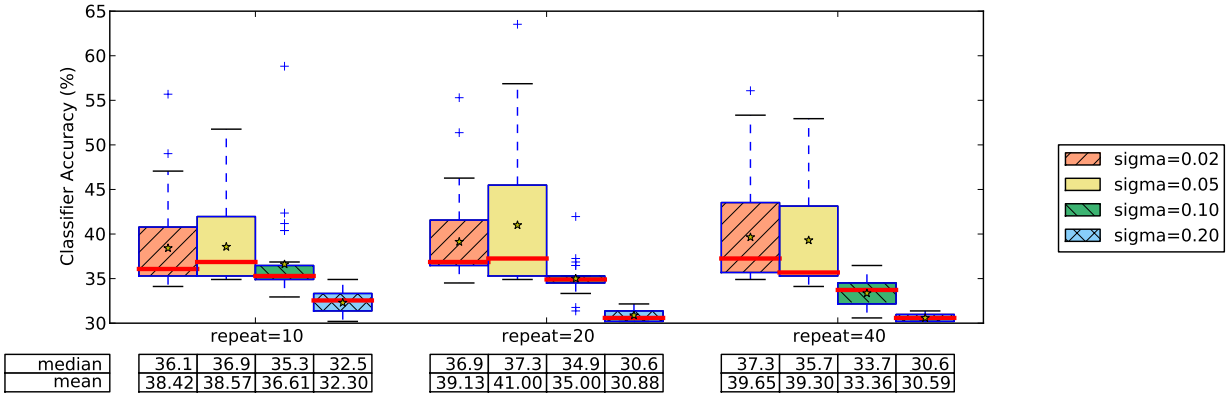
Quantitative results are illustrated in Figure 4.4. In general, this tracker performs very poorly, occasionally yielding results that are merely chance. Examining the tracks more closely, we see that many of the mistakes were associated with the classifier returning 0 (rock). This is the classifier output in the case of inconsistent results from the one-versus-one classifiers. However, changing to a one-versus-all classifier or renumbering the pose names does not affect the overall outcome. It may be that "rock" is simply the closest labeled pose in the data set to the strange poses that this tracker outputs.

Fixing the state size and examining the relationship between `repeat` and σ shows that this tracker is relatively fragile. As Figure 4.5 shows, at low values of `repeat`, a high value of σ is needed for the particles to drift fast enough to cover the change from one frame to the next. At high values of `repeat`, however, a high σ allows the tracker to confuse itself by drifting too much. As a result, it is difficult to tune this tracker to achieve good performance, and the exact parameters necessary will likely vary from dataset to dataset.



median	35.3	35.3	35.3	34.5	34.9	35.3	33.3	33.7	34.1
mean	36.95	36.61	36.65	34.88	35.00	35.46	33.89	33.36	34.19

Figure 4.4: Results of tracking using the simple tracker. In these trials, $\sigma = 0.1$.



median	36.1	36.9	35.3	32.5	36.9	37.3	34.9	30.6	37.3	35.7	33.7	30.6
mean	38.42	38.57	36.61	32.30	39.13	41.00	35.00	30.88	39.65	39.30	33.36	30.59

Figure 4.5: Results of tracking using the simple tracker. In these trials, `state_size` $= 512$.

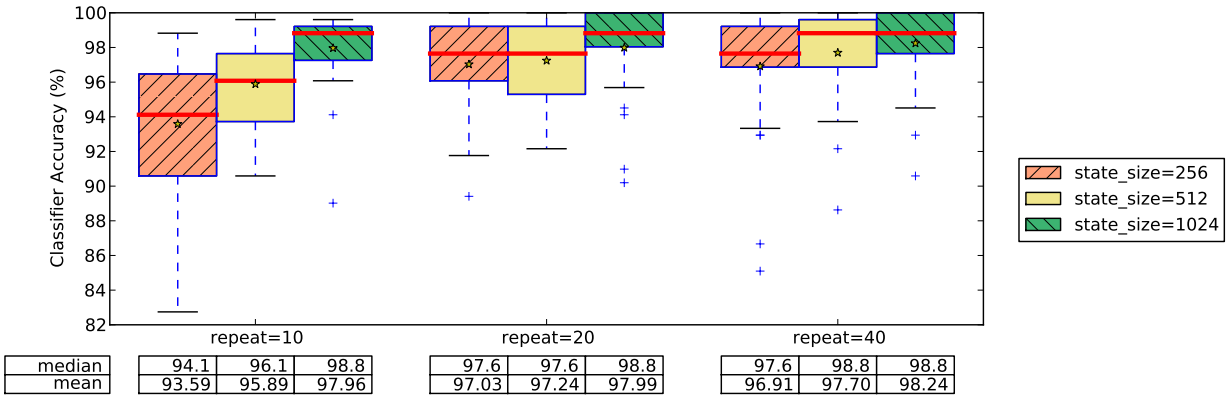
4.3.2 Constrained Pose

Since the unconstrained model can generate completely unrealistic poses, we add some hard constraints to the joints. The system equation now becomes

$$\mathbf{f}_k(\mathbf{x}_{k-1}, \mathbf{v}_{k-1}) = K(\mathbf{x}_{k-1} + \mathbf{v}_{k-1}) \tag{4.4}$$

where K is a function which maps out-of-bounds joint angles to angles which are in bounds. We apply the constraints on a per-joint basis.

$$K(\mathbf{x}) = \begin{bmatrix} K_0(x_0) \\ K_1(x_1) \\ \vdots \\ K_{26}(x_{26}) \end{bmatrix} \tag{4.5}$$



median	94.1	96.1	98.8	97.6	97.6	98.8	97.6	98.8	98.8
mean	93.59	95.89	97.96	97.03	97.24	97.99	96.91	97.70	98.24

Figure 4.6: Results of tracking using the constrained tracker. In these trials, $\sigma = 0.1$.

where x_j is the j^{th} component of $\mathbf{x}$.

The constraints themselves are listed in Table 4.1. These values are exactly 20 degrees more lax than those found in Table 3.1. Since we ensured that the values in Table 3.1 were within 10 degrees of the final solution in Section 4.3.2, the constraints in Table 4.1 are at least 10 degrees more lax than any pose found in our manually labeled frames.

$$K_i(x_i) = \begin{cases} l_i & x_i \leq l_i \\ x_i & l_i < x_i \leq h_i \\ h_i & h_i < x_i \end{cases}. \tag{4.6}$$

We apply the constraints in Table 4.1 on a per-particle basis. As a result it is unlikely for the weighted mean of the particles to actually achieve any of the constraints.

Even without considering inter-joint dependencies, these simple constraints are sufficient to significantly improve tracker performance, as shown in Figure 4.6. The classification accuracy usually surpasses 95%, and some of the trials yield a perfect score. This figure shows the general trend that increasing `repeat` is more productive than increasing `state_size`. However, beyond a certain point, increasing either will not improve the median score, and no setting can consistently provide a perfect score on every trial.

4.3.3 Motion Model

A more sophisticated model involves taking into account the actual motions exhibited by the training data.

$$\mathbf{f}_k(\mathbf{x}_{k-1}, \mathbf{v}_{k-1}) = K(\mathbf{x}_{k-1} + m(\mathbf{x}_{k-1}, \mathbf{v}_{k-1})) \tag{4.7}$$

Table 4.1: Hard constraints for tracking.

j	Upper bound (u_j)	Lower bound (l_j)
0	70	0
1	85	0
2	115	-20
3	60	0
4	130	-40
5	140	-20
6	105	-45
7	60	-40
8	130	-40
9	140	-20
10	105	-45
11	60	-40
12	130	-40
13	140	-20
14	105	-45
15	60	-40
16	130	-40
17	140	-20
18	105	-45
19	60	-40
20	20	-45
21	180	-180
22	180	-180
23	180	-180
24	20	-10
25	5	-20
26	-40	-80

where $m(\mathbf{x}, \mathbf{v})$ is a motion model which takes into account the current position when suggesting an update direction. We propose a simple motion model based on our training data. Note that we continue to apply the per-joint constraints as in Section 4.3.2.

To create our model, we partition the training data into k regions using k-means clustering. This gives cluster centers $\{\mathbf{c}_i\}_{i=1}^{k}$. First, we partition the training data by the clustering.

$$f \in F_i \text{ iff } \arg\min_j \|\mathbf{p}^f - \mathbf{c}_j\| = i. \tag{4.8}$$

Then, we train (full) covariance matrices based on the clustering.

$$\Sigma_i = \operatorname{cov}\left\{(\mathbf{p}^{f+1} - \mathbf{p}^f)\right\}_{f \in F_i}. \tag{4.9}$$

Note that these covariances are based on the differences between adjacent frames, not on the poses themselves. This means that, even with $k = 1$, we are still using our training data for motion information.

During tracking, we reverse the process used to generate the model.

$$m(\mathbf{x}, \mathbf{v}) = \sqrt{\Sigma_{(\arg\min_j \|\mathbf{x} - \mathbf{c}_j)}}\mathbf{v} \tag{4.10}$$

where $\sqrt{\cdot}$ is a matrix square root (we use the Cholesky decomposition) and $\mathbf{v}$ is distributed according to $\mathcal{N}(0, 1)$. In words, we compute the distance from the current candidate particle to each of the cluster centers. Then, we look up the corresponding Σ_i and alter $\mathbf{v}$ so that it is distributed by $\mathcal{N}(0, \Sigma_i)$. In summary, we essentially have

$$\mathbf{f}_k(\mathbf{x}_{k-1}, \mathbf{v}_{k-1}) = K(\mathbf{x}_{k-1} + m(\mathbf{x}_{k-1}, \mathbf{v}_{k-1})) \tag{4.11}$$

where $\mathbf{v}_{k-1}$ is distributed by $\Sigma_{(\arg\min_j \|\mathbf{x} - \mathbf{c}_j)}$.

As shown in Table 4.2, this is a very powerful model, achieving better accuracy than the constraints alone while expending an order of magnitude fewer particles. In particular, with sufficiently many repeats, it is possible to reliably achieve a classification accuracy of 100%. Note that as k increases above three, the performance decreases. This is probably due to the fact that we only have 1680 training data points. Dividing that set into five or ten may not leave enough points to accurately estimate a 27-dimensional covariance matrix.

The table also shows that the motion-model tracker performs well even with a very small number of particles. For instance, running with `state_size` = 128 and `repeat` = 1 at 30 frames per second and five cameras requires 19,200 comparisons per second. This is well within the capabilities of a single NVIDIA GT200 GPU, and yields a respectable 93.9%

Table 4.2: Results of tracking using a motion-model tracker. The values represent the median/mean accuracy of 31 trials with $\sigma = 1.0$.

k	`state_size`	`repeat` = 1	`repeat` = 2	`repeat` = 5	`repeat` = 10
1	128	92.5/ 91.90	98.8/ 97.27	100.0/ 98.89	100.0/ 99.03
1	256	94.9/ 94.45	99.2/ 98.89	100.0/ 99.44	100.0/ 99.33
1	512	95.7/ 95.52	99.6/ 99.24	100.0/ 99.51	100.0/ 99.71
1	1024	96.5/ 96.56	99.6/ 99.52	100.0/ 99.84	100.0/100.00
3	128	93.3/ 92.89	98.8/ 97.71	100.0/ 99.37	100.0/ 99.17
3	256	94.5/ 94.54	99.2/ 98.46	100.0/ 99.11	100.0/ 99.47
3	512	96.1/ 96.07	99.6/ 99.66	100.0/ 99.89	100.0/ 99.91
3	1024	96.5/ 96.66	100.0/ 99.62	100.0/ 99.85	100.0/ 99.99
5	128	88.2/ 88.41	97.3/ 96.37	99.6/ 98.94	100.0/ 99.27
5	256	91.0/ 90.75	98.0/ 98.00	100.0/ 99.77	100.0/ 99.68
5	512	92.5/ 92.61	98.8/ 98.61	100.0/ 99.90	100.0/ 99.97
5	1024	94.1/ 93.74	98.8/ 98.92	100.0/ 99.97	100.0/ 99.85
10	128	84.3/ 84.30	95.7/ 95.23	99.6/ 98.48	100.0/ 99.80
10	256	88.6/ 88.72	97.3/ 96.86	100.0/ 99.39	100.0/ 99.86
10	512	90.2/ 89.61	98.0/ 97.91	100.0/ 99.92	100.0/ 99.66
10	1024	92.5/ 92.25	98.8/ 98.62	100.0/ 99.95	100.0/ 99.99

median accuracy when $k = 3$. Using ten times more processing power (`state_size` = 256 and `repeat` = 5), it is possible to almost assure 100% tracking accuracy. By comparison, the constrained tracker with (`state_size` = 512 and `repeat` = 20) consumes eight times the number of particles and yields only 97.6% accuracy.

4.4 Reduced Framerate

There is a certain amount of overhead in reading the frames from the camera. There might be limited bandwidth on the inputs of the host machine, especially when there are a large number of cameras. There is also some amount of processing overhead and CPU–GPU communication needed for every frame. Also, reducing the framerate allows us to increase the exposure time of the camera. This may improve the image quality at the expense of possible motion blur.

We simulate a reduced camera framerate by processing only one in every five frames. At this speed, there can be very large displacement between subsequent frames, and the problem looks more like a detection problem as the knowledge of the pose at the previous frame becomes less informative. In an attempt to compensate for the increase in pose differences

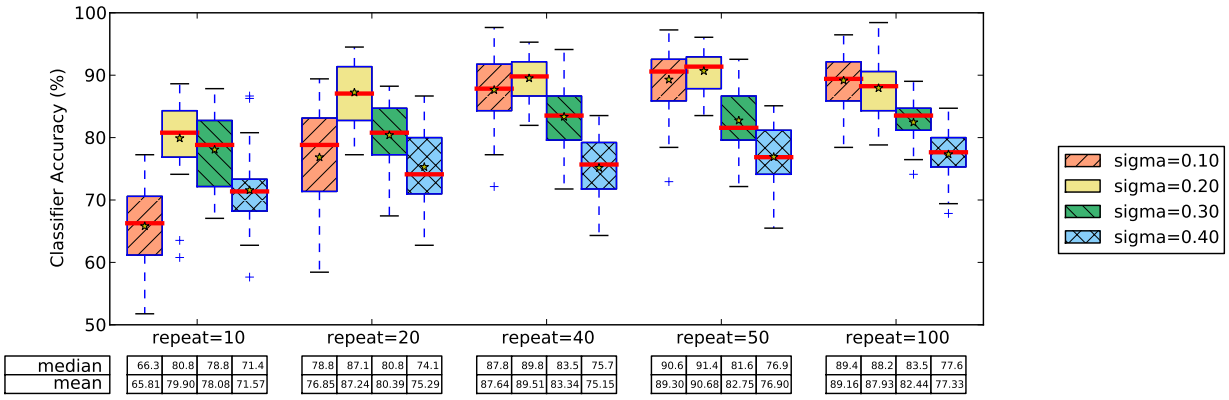


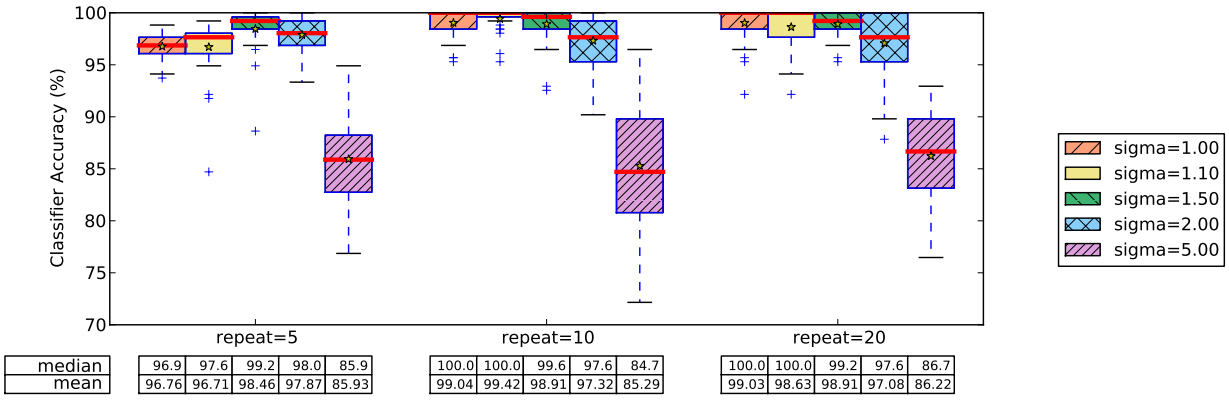
Figure 4.7: Results of reduced framerate tracking using the constrained tracker. In these trials, we process only every 5 frames, and `state_size` = 512.

between frames, we experiment with different values of σ. The results are summarized in Figure 4.7.

Comparing Figure 4.7 with Figure 4.6 shows a significant drop in performance at the reduced framerate. For example, processing one in every five frames with `repeat` = 50 consumes as many comparisons as processing every frame with `repeat` = 10. However, the former is 5% less accurate than the latter. Furthermore, the highest score achieved at the reduce framerate, with `repeat` = 100, is 92.7%. This is between the performance of `repeat` = 5 and `repeat` = 10 at the full framerate. This implies that running at the full framerate can reduce the number of comparisons needed by a factor of two to four. Finally, note that Figure 4.7 demonstrates the optimal σ varies with `repeat`. As with the simple model, this suggests that we are beyond the boundary of reliable tracking.

With a motion model, however, the problem becomes much more tractable. Figure 4.8 shows the results of tracking using a motion model with $k = 3$. Note that, while the σ variable serves the same purpose as in the constrained tracker, the version used for the motion-model tracker uses a different scale. For the motion-model tracker, $\sigma = 1.0$ is the default σ, and that value represents the standard deviation of the training data without any additional scaling. In theory, since we are reducing the number of frames by five, we should use $\sigma = 5.0$ to compensate. However, that yields very poor performance. This is probably due to the fact that the actual motion of the video consists of short bursts of activity. As a result, the maximum change in pose over five frames is not five times the change found over one frame.

Compared to Table 4.2, the results for the reduced framerate tracking are similar. At the reduced framerate, `repeat` = 5 is equivalent to `repeat` = 1 at the full framerate. Under these circumstances, it is possible to tweak σ so that the reduced framerate tracker outperforms



	repeat=5					repeat=10					repeat=20				
median	96.9	97.6	99.2	98.0	85.9	100.0	100.0	99.6	97.6	84.7	100.0	100.0	99.2	97.6	86.7
mean	96.76	96.71	98.46	97.87	85.93	99.04	99.42	98.91	97.32	85.29	99.03	98.63	98.91	97.08	86.22

Figure 4.8: Results of reduced framerate tracking using the motion-model tracker ($k = 3$). In these trials, we process only every 5 frames, and `state_size` $= 512$.

the regular tracker. Similarly, reduced framerate `repeat` $= 10$ outperforms regular framerate `repeat` $= 2$. In these cases, it seems that the pattern of higher `repeat` trumping `state_size` holds. However, the reduced framerate tracker never quite reaches the accuracy level of the full framerate tracker. (The median of reduced framerate trials reaches 100%, but the mean never breaks 99.9%.)

4.5 Camera Subsets

To ensure that we had enough information to perform tracking, we used five cameras spaced around the hand. However, this is a large number of cameras to set up and maintain, making the system relatively impractical for real-world deployment. Also, each camera comes with additional computation burden. In this section, we examine how many and which cameras we need to obtain good performance. We conduct trials with every possible subset of one, two, and four cameras. The results are summarized in Table 4.3.

The second column for each tracker deserves some explanation. In order to convert the chamfer distance into a probability distribution, we use the formula

$$p(\mathbf{p}) \propto e^{-\gamma\left[\sum_{c\in C} d_{\mathrm{cfr}}(r_c(\mathbf{p}),\mathbf{c}_c)\right]}. \tag{4.12}$$

That is, we sum the chamfer distance from each camera and scale that sum by γ before taking the exponent. The default γ is 0.5, which was a value tuned by experimentation. If we reduce the number of cameras, however, we should increase γ in order to keep the values

Table 4.3: Results of tracking using a subset of the cameras. The values represent the median/mean accuracy of 31 trials with `state_size` = 512.

camera set ($\mathcal{C}$)	constrained (`repeat` = 20, $\sigma = 0.1$)		motion-model ($k = 3$, `repeat` = 2, $\sigma = 1.0$)	
	$\gamma = 0.5$	$\gamma = \frac{2.5}{\lvert\mathcal{C}\rvert}$	$\gamma = 0.5$	$\gamma = \frac{2.5}{\lvert\mathcal{C}\rvert}$
{0}	42.7/ 43.02	41.6/ 42.15	83.1/ 82.67	73.3/ 74.83
{1}	60.0/ 60.90	44.3/ 45.49	91.4/ 91.45	83.9/ 83.49
{2}	65.1/ 62.83	46.7/ 46.88	82.0/ 82.44	74.9/ 74.09
{3}	52.5/ 53.00	50.2/ 49.55	83.9/ 81.97	71.8/ 70.37
{4}	54.1/ 53.75	51.0/ 51.87	96.9/ 94.40	87.1/ 85.01
{0,1}	71.4/ 70.49	65.1/ 65.36	96.9/ 95.75	96.5/ 95.37
{0,2}	67.8/ 68.30	63.9/ 65.53	98.0/ 97.56	95.7/ 94.24
{0,3}	64.7/ 65.43	63.9/ 64.53	98.4/ 96.82	95.7/ 94.24
{0,4}	77.3/ 76.53	69.0/ 69.59	99.2/ 98.63	99.6/ 98.46
{1,2}	80.0/ 78.20	72.9/ 72.81	90.6/ 89.85	91.8/ 90.28
{1,3}	82.7/ 82.90	80.0/ 80.24	98.0/ 96.89	97.6/ 96.14
{1,4}	66.7/ 67.92	63.1/ 62.61	97.6/ 97.18	98.0/ 97.41
{2,3}	75.3/ 76.14	78.8/ 79.42	97.3/ 96.31	97.6/ 96.08
{2,4}	66.3/ 67.46	66.7/ 67.89	99.6/ 98.48	98.8/ 96.74
{3,4}	63.9/ 61.39	62.0/ 62.29	88.6/ 87.91	87.5/ 87.27
{0,1,2}	75.7/ 75.27	74.1/ 74.14	98.0/ 97.82	98.8/ 98.12
{0,1,3}	92.5/ 92.07	93.3/ 92.09	99.6/ 98.86	99.6/ 98.58
{0,1,4}	86.3/ 85.52	82.7/ 82.86	98.8/ 98.75	98.8/ 98.86
{0,2,3}	77.6/ 78.10	76.9/ 78.10	99.6/ 98.79	99.6/ 98.49
{0,2,4}	92.9/ 92.95	95.7/ 93.98	99.6/ 99.17	99.6/ 99.54
{0,3,4}	83.9/ 82.34	77.3/ 76.58	100.0/ 99.44	99.6/ 98.60
{1,2,3}	88.6/ 87.45	88.6/ 88.84	98.8/ 98.10	99.2/ 98.39
{1,2,4}	72.2/ 72.38	71.8/ 71.80	98.4/ 97.82	98.0/ 97.46
{1,3,4}	79.2/ 78.06	76.1/ 75.71	99.2/ 98.49	98.8/ 97.08
{2,3,4}	79.2/ 75.86	80.0/ 76.91	99.2/ 97.84	99.2/ 97.51
{0,1,2,3}	92.9/ 92.46	92.2/ 91.59	100.0/ 99.62	99.6/ 99.43
{0,1,2,4}	96.9/ 95.93	97.6/ 96.38	99.2/ 99.06	99.2/ 98.92
{0,1,3,4}	96.1/ 95.53	94.9/ 95.02	99.6/ 99.30	99.6/ 99.65
{0,2,3,4}	94.5/ 94.70	95.7/ 93.56	100.0/ 99.63	100.0/ 99.54
{1,2,3,4}	85.9/ 85.50	81.6/ 82.47	99.2/ 98.71	99.2/ 98.43
{0,1,2,3,4}	97.6/ 97.24	97.6/ 97.24	99.6/ 99.66	99.6/ 99.66

in the same range. Thus, we use a γ of

$$\gamma = \frac{2.5}{|\mathcal{C}|} \tag{4.13}$$

where $|\mathcal{C}|$ is the number of cameras in the subset.

Adjusting γ in this way does not appear to improve performance. In general, the trackers are relatively robust to changes in γ. At very high values of γ, however, minor variations in the chamfer distance lead to very large shifts in probability, and the tracker loses its ability to maintain multiple hypotheses.

4.6 Suboptimal Hand Shape

Throughout the previous discussion, we have been using the highly tuned hand shape that we calibrated in Section 3.4. However, in practice, it would be impractical to calibrate the hand shape to this degree. Thus, in this section, we examine the effect of miscalibrations on tracker performance.

It might be easier to estimate overall hand length than to compute the lengths of individual phalanxes. Thus, we consider adjusting the proportion of the phalanx lengths to match those given by Buchholz et al. [6]. (See Table 1.1 in Section 1.1.) Next, we investigate the effects of misestimating the hand size by making the hand 10% smaller and larger. We also consider the effects of misestimating the joint angles by forcing all of the joint axes to be perfectly horizontal. Finally, we consider a hand model that lacks the palm-folding joint.

Figure 4.9 shows the effect of the suboptimal shapes under different camera subsets. These cameras represent the best median accuracy of Table 4.3. We also repeat the performance result of the calibrated hand model as a comparison.

Note that increasing the hand size is particularly disastrous for recognition. By contrast, using a model that is smaller than the real hand does not affect performance as much. The next most significant drop in performance comes from using simplified joint axes. This causes around 7% drop in classifier accuracy, although it seems to increase performance in the single camera case. In general, changing the relative proportion of the phalanx lengths does not appear to affect tracker performance, particularly because all of our gestures have the fingers either fully extended or fully flexed. Although the "hand fold" joint was necessary to achieve a good fit with the manually labeled point data in Section 3.4, it does not seem to actually be necessary for tracking. In fact, eliminating the hand fold dimension seems to actually improve performance, probably because of the reduced search space.

The picture is somewhat different if we use the motion model. Figure 4.10 shows the

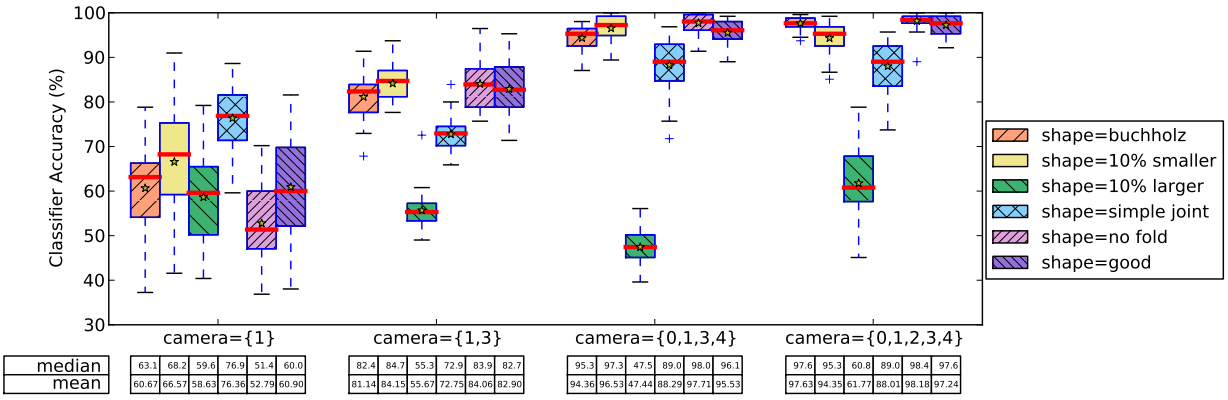


Figure 4.9: Results of tracking using the constrained tracker with different shape models. In these trials, `state_size` = 512, `repeat` = 20, and $\sigma = 0.1$.

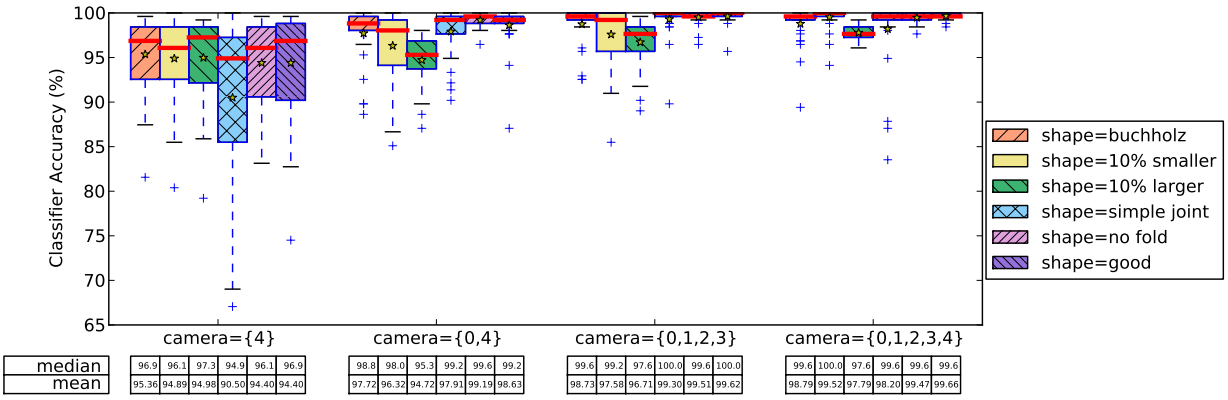
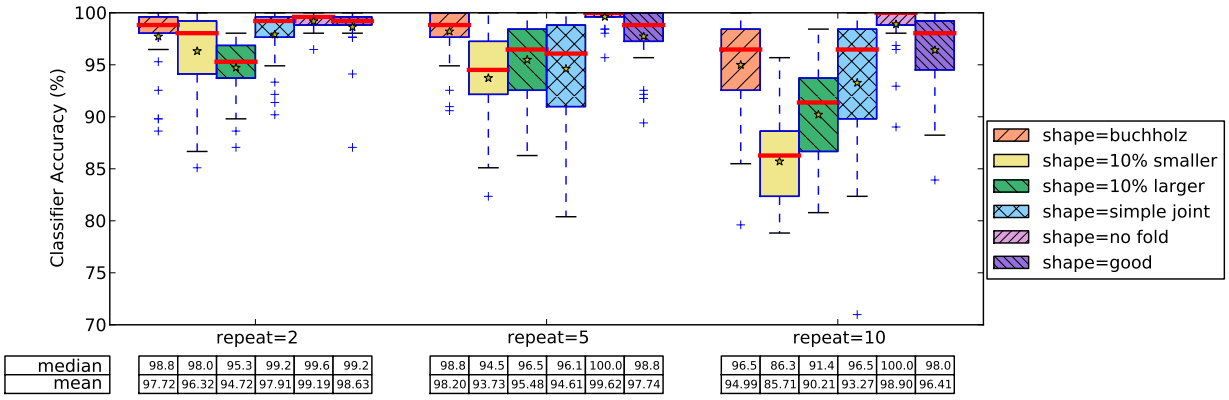


Figure 4.10: Results of tracking using the motion-model tracker ($k = 3$) with different shape models. In these trials, `state_size` = 512, `repeat` = 2, and $\sigma = 1.0$.

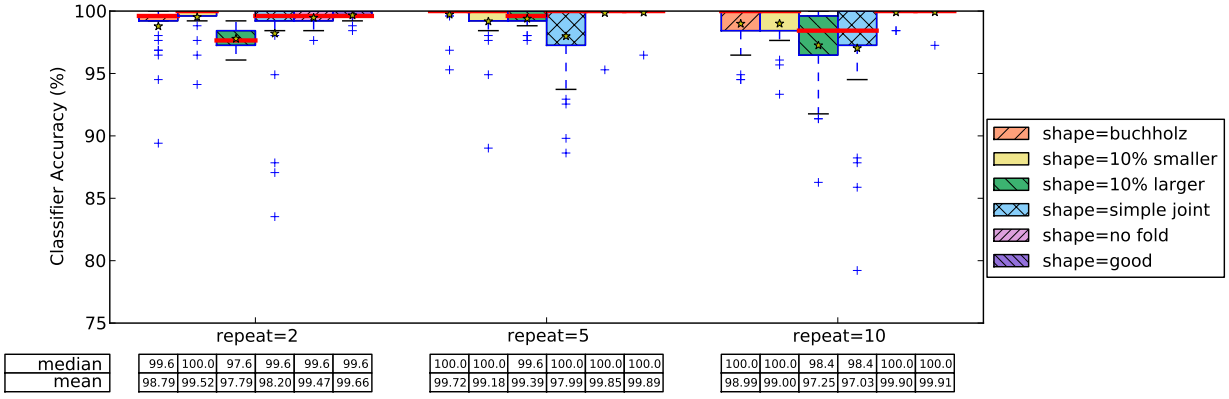
results. Increasing the hand size no longer differs as much as decreasing the hand size. Also, removing the fold joint seems to have a more deleterious effect. Changing the proportions of the phalanxes seems to have more of an impact, especially with a large number of cameras.

In both cases, the effect of suboptimal shapes increases as the number of cameras decreases. This is especially apparent in the two-camera case with the motion-model tracker. Fortunately, as shown in Figure 4.11, increasing the `repeat` can bring the performance back up to levels similar to adding cameras. However, as shown in Figure 4.12, even with all five cameras, incorrect hand shapes can still degrade performance. In particular, note that instead of saturating to 100% accuracy as `repeat` increases, the performance degrades beyond a certain point.



median	98.8	98.0	95.3	99.2	99.6	99.2	98.8	94.5	96.5	96.1	100.0	98.8	96.5	86.3	91.4	96.5	100.0	98.0
mean	97.72	96.32	94.72	97.91	99.19	98.63	98.20	93.73	95.48	94.61	99.62	97.74	94.99	85.71	90.21	93.27	98.90	96.41

Figure 4.11: Results of tracking using the motion-model tracker ($k = 3$) with different shape models. In these trials, $\mathcal{C} = \{0, 4\}$, `state_size` $= 512$, and $\sigma = 1.0$.



median	99.6	100.0	97.6	99.6	99.6	99.6	100.0	100.0	99.6	100.0	100.0	100.0	100.0	100.0	98.4	98.4	100.0	100.0
mean	98.79	99.52	97.79	98.20	99.47	99.66	99.72	99.18	99.39	97.99	99.85	99.89	98.99	99.00	97.25	97.03	99.90	99.91

Figure 4.12: Results of tracking using the motion-model tracker ($k = 3$) with different shape models. In these trials, $\mathcal{C} = \{0, 1, 2, 3, 4\}$, `state_size` $= 512$, and $\sigma = 1.0$.

4.7 Conclusion

As we demonstrated in Section 4.3.3, when we have optimal conditions (restricted set of motions, all five cameras, and a highly tuned hand shape) we can achieve almost perfect tracking using 192,000 comparisons per second. This is approximately the peak throughput of six NVIDIA GT200 chips. Without constraints on the motions, we will need four to eight times more computation capacity to achieve similar (but inferior) results. We can reduce the number of cameras to four without compromising quality. However, further reductions increase errors. Finally, almost every deviation from the optimal hand shape introduces more errors. The exception is in the hand fold joint, which does not seem to actually benefit tracking. In all cases, we want to make sure that we can run the camera at the highest framerate possible. This means that a perfect tracker will require an unreasonable level of calibration. It will also consume more computation power than is commonly available on a

2009-level workstation.

In the end, however, the system parameters depend highly on the application. If only 90% accuracy is required, it may be feasible to field a two-camera system running with a motion model with `state_size` = 512 and `repeat` = 2. As shown in Figure 4.10, this system remains within tolerances even when given somewhat inaccurate hand shape information. It also only consumes 61,440 comparisons per second, which is within the peak performance of two GT200 cards. Given that dual-card setups are available for gaming machines, this is within reach of a commercially available, if somewhat exotic, desktop system. Also, Moore's law marches on, and new processors will become available with even more capacity. Given the scalable nature of our tracking algorithm, it should be possible to achieve real-time tracking of highly complex articulated objects in the near future.

Chapter 5

Tracking Experiments

While the experiments in Chapter 4 highlight how various parameters affect performance, they do not explain how well the tracker works in general. To highlight this aspect, we recorded two additional test sequences:

- The "easy" sequence (Figure 5.1) is a relatively simple sequence with the hand rotating in place while fingers are extended and flexed. The fingers remain extended throughout a large portion of the sequence. This sequence contains 1680 frames (numbered 120–1799 inclusive).

- The "pen" sequence (Figure 5.2) is a more challenging sequence of the hand twirling a pen. The pen causes partial occlusion of the hand, and it can force joints beyond their normal bounds. Also, because the hand is similar to the background, there are often segmentation errors. This sequence contains 1680 frames (numbered 120–1799 inclusive).

For each sequence, we solve for the camera calibration, but we keep the hand shape parameters that we solved in Section 3.4. The new sequences are of the same hand, and although these sequences were acquired over six months later, we presume that the hand shape has not changed significantly. We begin tracking with manual initialization at frame 120, and run the tracker until frame 1799.

5.1 Evaluation Metric

In order to understand how well the tracker performs, we need an evaluation metric. This should be a technique-independent method, and this precludes the use of the chamfer distance used by the tracker (Section 2.2). We consider three potential candidates:

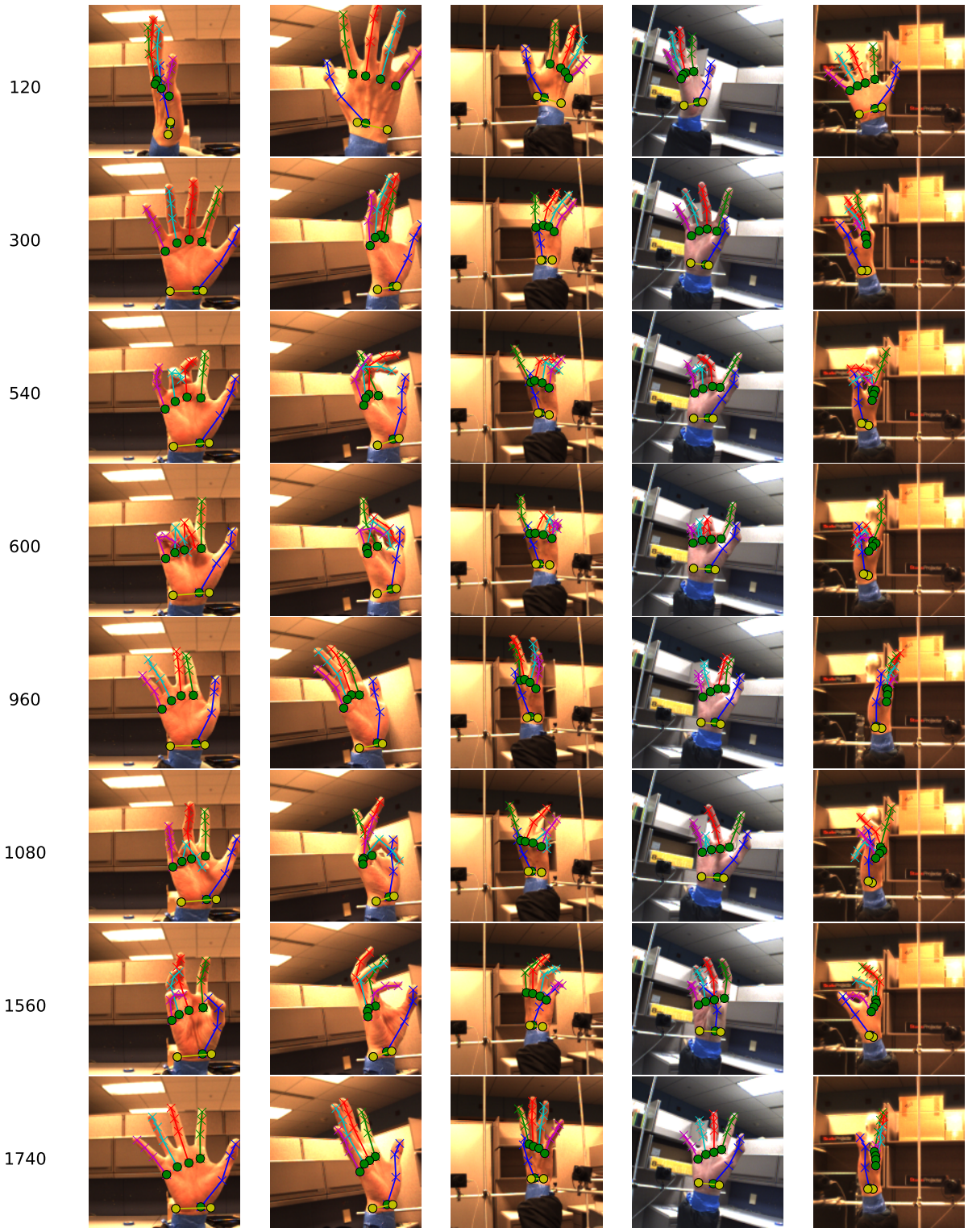


Figure 5.1: Sample frames from the "easy" sequence, along with ground truth labels.

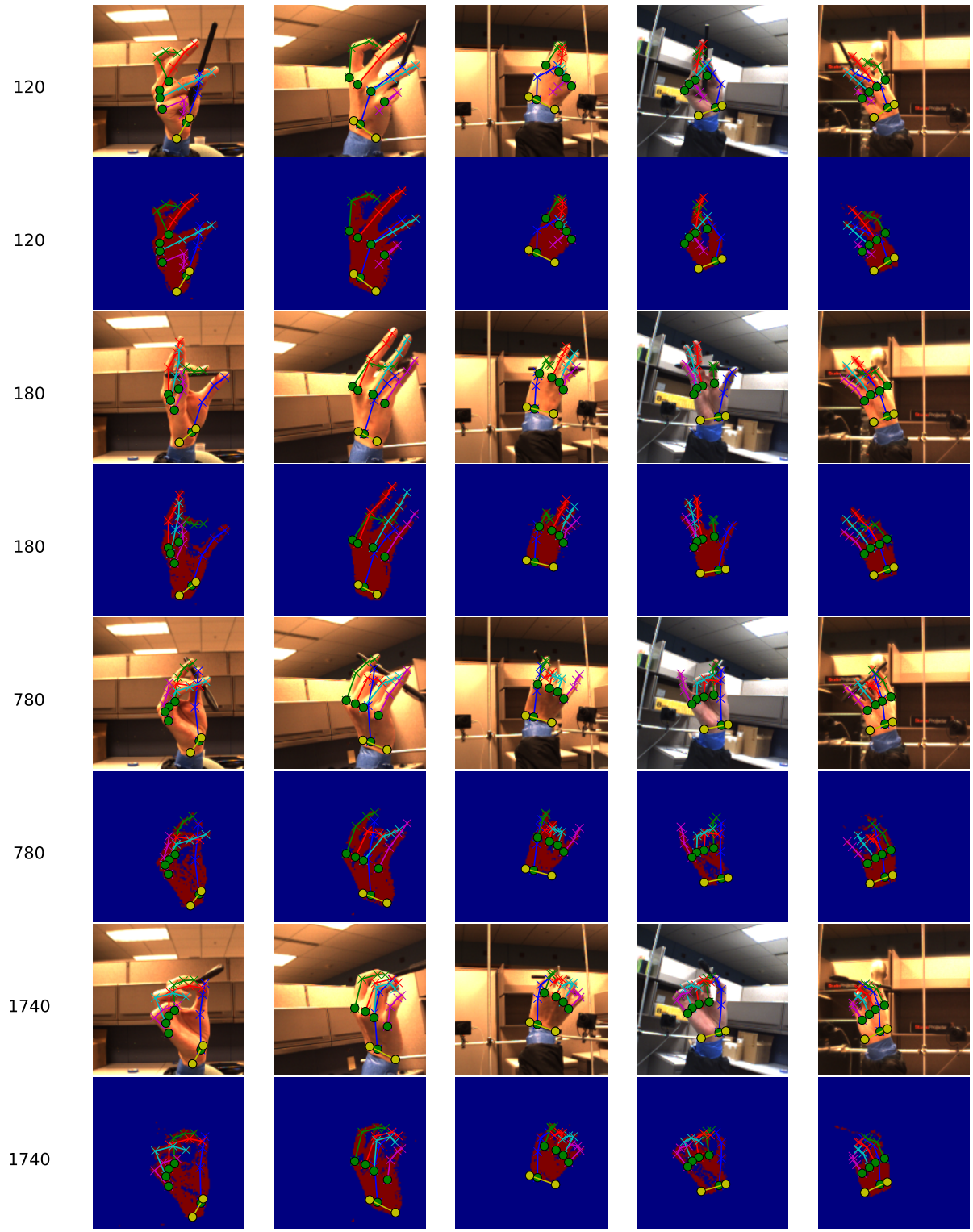


Figure 5.2: Sample frames from the "pen" sequence, along with ground truth labels. Note that because of occlusion and lighting, some of the hand segmentation is wrong.

- The root mean squared (RMS) error in joint space. Under this system, we describe ground truth of the hand with the 27-dimensional vector as given in Section 2.1.1. Since this is the native output of the tracker, it is trivial to compute the RMS error. There may be a minor issue because the global position portion of the pose is measured in distance units where every other dimension is measured in angle.

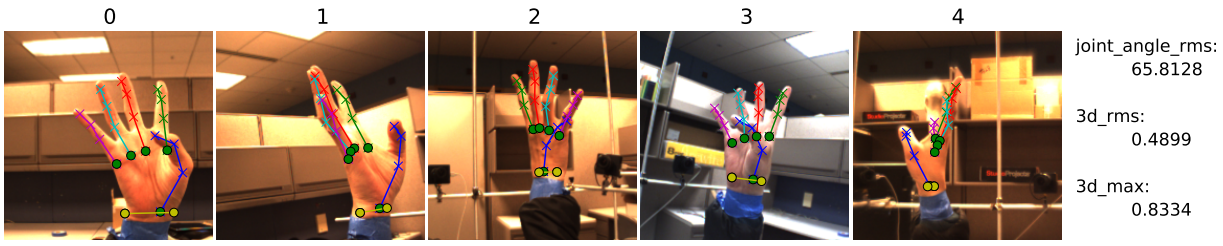
- The RMS distance of the keypoints. We label the positions of certain keypoints (described in Section 3.2) in 3D as ground truth. (Note that this labeling does not necessarily respect the hand shape.) We then compute the position of the key points based on the pose output of the tracker. Then, we compute RMS of the distances between the corresponding points. This should be roughly equivalent to computing the mean or some method of averaging of the distances.

- The maximum distance of the keypoints. This measure is similar to the previous one, except that we take the maximum of the distances instead of the mean. This emphasizes the worst mismatch, and is reminiscent of the Hausdorff distance.

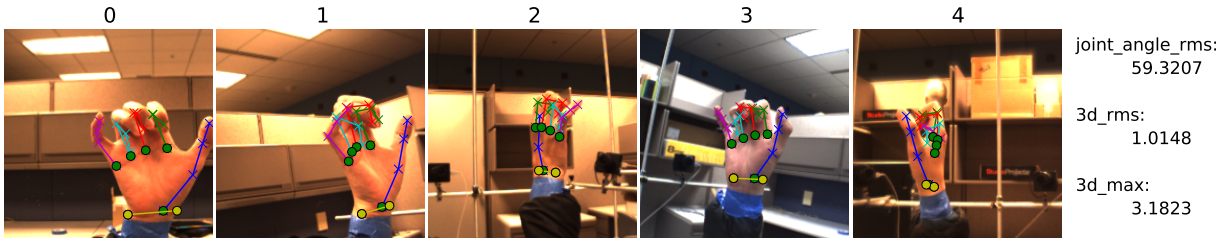
Figure 5.3 shows the result of choosing the RMS error in joint-angle space. Note that the state depicted in Figure 5.3b has a lower error than that in Figure 5.3a even though there is a significant mismatch of the ring and index fingers in the second example. One issue with RMS error in joint-angle space is that it weights all of the joints equally, even though those at the base of the kinematic chain are going to have a larger effect on the overall appearance. There is also a possibility of ambiguities in the representation, which results in two solutions that look very similar varying significantly in joint-angle space. Overall, although a very low joint-angle error implies a good match with the ground truth, medium and sometimes even high joint-angle errors do not signal a significant mismatch.

Unlike the joint-angle based metric, the two 3D-point based metrics generally correlate well. The results in this chapter would probably be similar regardless of which one was used. Figure 5.4 illustrates the rather subtle difference between the two measures. Again, Figure 5.4a has a higher RMS error than Figure 5.4b even though the latter has a more significant ring finger error. It appears that taking the 2-norm can average out errors while taking the maximum emphasizes the grossest mismatch. As a result, we adopt the maximum distance of 3D keypoints as our per-frame error metric.

Once we have settled on a metric for a single frame, we need some way of collating the data across an entire sequence. One possibility is to continue the use of the maximum operator and report the highest error across the frames. The problem with this technique is that it emphasizes the importance of the most difficult frame at the expense of reporting on the other frames in the sequence. Computing the mean or some other average across the



(a)



(b)

Figure 5.3: Comparison of joint-angle based error metric against 3D-point based error metrics. (a) An example with high joint-angle error. (b) An example with a low joint-angle error.

frames is another possibility. However, the exact meaning of such a measure is difficult to understand. Since the overall goal of the tracker is to service a higher layer, we settled on reporting the percentage of frames where the tracker error was below a certain threshold. A threshold value of 2.5 was chosen based on inspection. It allows the tracker to report 100% success on some runs of the "easy" sequence.

5.2 Results

To obtain the results, we manually label the ground truth of every 60 frames. We then conduct 31 runs of tracking at various settings. As Figure 5.5 shows, the tracker is easily able to handle the "easy" sequence. Even at lower values of `repeat`, a proper tuning of σ grants a fairly reliable result. However, even at high values of `repeat`, it is not possible to always obtain good tracking, although the number of outliers appears to drop. These results serve as a reasonable baseline of what a finger tracking system should achieve.

Unfortunately, the results for the "pen" sequence are not as rosy. Regardless of the settings, the tracker performs with only middling accuracy. In some frames, the pen breaks the fingers in two, causing tracking degradation. For example, one of the least reliably tracked frames was frame 180. As shown in Figure 5.6, the fingers are broken into two pieces in the first and last camera view. It appears that a motion model or some way of explicitly modeling the pen will be necessary to overcome these challenges.

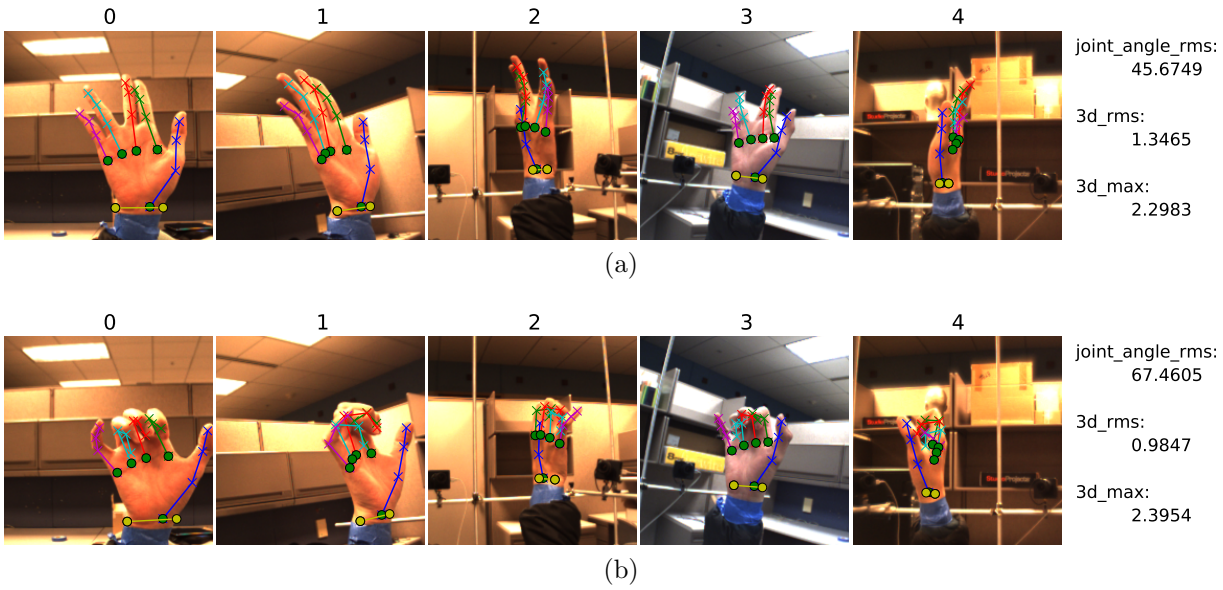


Figure 5.4: Comparison of joint-angle based error metric against 3D-point based error metrics. (a) An example with high RMS error. (b) An example with a low RMS error.

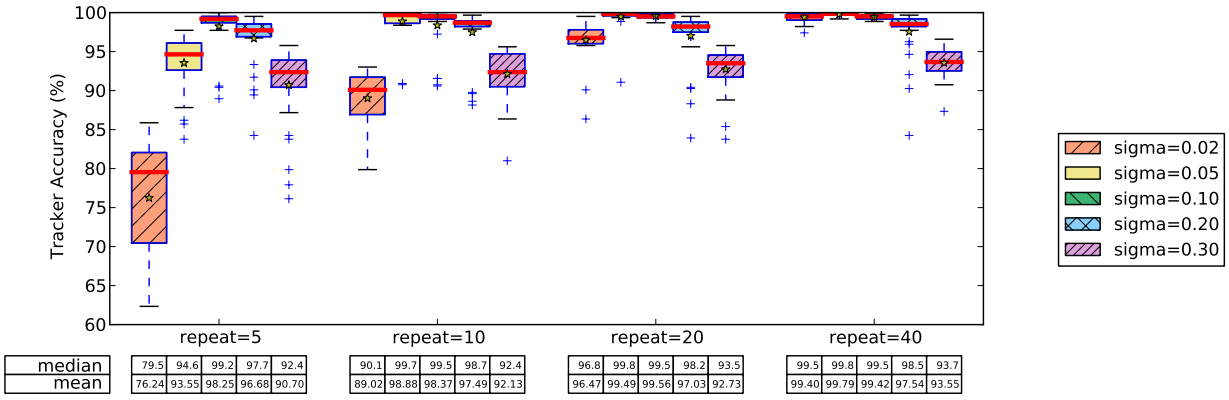


Figure 5.5: Results of tracking the "easy" sequence. In these trials, $\mathcal{C} = \{0, 1, 2, 3, 4\}$, and `state_size` $= 512$.

5.3 Conclusion

In this chapter, we demonstrate that the system is capable of performing general purpose tracking under some circumstances. It still requires more computation power than is generally available for a real-time application. Also, the results are highly sensitive to the silhouette information. Instances where the fingers are flexed yield very little information in the silhouette, leading to poor tracking. Occlusions also present great difficulties for the tracker. Based on the conclusions from Chapter 4, we would need an enhanced motion model to overcome these challenges. In summary, this work represents a significant step in under-

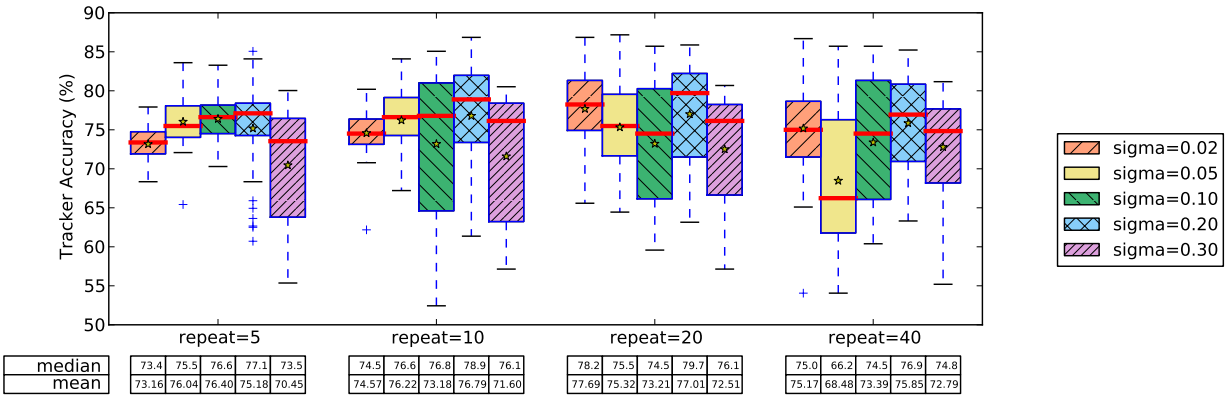


Figure 5.6: Results of tracking the "pen" sequence. In these trials, $\mathcal{C} = \{0, 1, 2, 3, 4\}$, and `state_size` $= 512$.

standing how to track the human hand, but it is one of many needed for a fully automatic and robust system.

References

[1] D. C. Riordan, MD, "A walk through the anatomy of the hand and forearm," *Journal of Hand Therapy*, vol. 8, no. 2, pp. 68–78, Apr.–Jun. 1995.

[2] J. M. F. Landsmeer, "Studies in the anatomy of articulation," *Acta Morphologica Neerlando-Scandinavica*, vol. 3, pp. 287–303, 1961.

[3] K. N. An, E. Y. Chao, W. P. Cooney, III, and R. L. Linscheid, "Normative model of human hand for biomechanical analysis," *Journal of Biomechanics*, vol. 12, no. 10, pp. 775–788, 1979.

[4] E. Y. Chao, J. D. Opgrande, and F. E. Axmear, "Three-dimensional force analysis of finger joints in selected isometric hand functions," *Journal of Biomechanics*, vol. 9, no. 6, pp. 387–396, 1976.

[5] N. Brook, J. Mizrahi, M. Shoham, and J. Dayan, "A biomechanical model of index finger dynamics," *Medical Engineering and Physics*, vol. 17, no. 1, pp. 54–63, Jan. 1995.

[6] B. Buchholz, T. J. Armstrong, and S. A. Goldstein, "Anthropometric data for describing the kinematics of the human hand," *Ergonomics*, vol. 35, no. 3, pp. 261–273, Mar. 1992.

[7] A. Erol, G. Bebis, M. Nicolescu, R. D. Boyle, and X. Twombly, "Vision-based hand pose estimation: A review," *Computer Vision and Image Understanding*, vol. 108, no. 1-2, pp. 52–73, 2007.

[8] C. Tomasi, S. Petrov, and A. Sastry, "3d tracking = classification + interpolation," in *Ninth IEEE International Conference on Computer Vision*, Nice, France, Oct. 13–16, 2003, pp. 1441–1448.

[9] H. Zhou, D. Lin, and T. Huang, "Static hand gesture recognition based on local orientation histogram feature distribution model," in *IEEE Conference on Computer Vision and Pattern Recognition Workshop*, 2004, p. 161.

[10] P. Dreuw, C. Neidle, V. Athitsos, S. Sclaroff, and H. Ney, "Benchmark databases for video-based automatic sign language recognition," in *Proceedings of the Sixth International Conference on Language Resources and Evaluation*, Marrakech, Morroco, May 2008. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2008/summaries/287.html

[11] R. Rosales, V. Athitsos, L. Sigal, and S. Sclaroff, "3d hand pose reconstruction using specialized mappings," in *Eighth IEEE International Conference on Computer Vision*, 2001, pp. 378–385.

[12] V. Athitsos and S. Sclaroff, "Estimating 3d hand pose from a cluttered image," in *IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, June 2003, pp. 432–439.

[13] M. Potamias and V. Athitsos, "Nearest neighbor search methods for handshape recognition," in *Proceedings of the 1st International Conference on PErvasive Technologies Related to Assistive Environments*, 2008, pp. 1–8.

[14] B. Stenger, A. Thayananthan, P. Torr, and R. Cipolla, "Model-based hand tracking using a hierarchical Bayesian filter," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2006, pp. 1372–1384.

[15] M. Isard and A. Blake, "Condensation—Conditional density propagation for visual tracking," *International Journal of Computer Vision*, vol. 29, no. 1, pp. 5–28, 1998.

[16] T. Heap and D. Hogg, "Towards 3d hand tracking using a deformable model," in *Second IEEE International Conference on Automatic Face and Gesture Recognition*, Killington, VT, Oct. 14–16, 1996, pp. 140–145.

[17] J. M. Rehg and T. Kanade, "Visual tracking of high DOF articulated structures: An application to human hand tracking," in *Third European Conference on Computer Vision*, vol. 801/1994, 1994, pp. 35–46.

[18] J. M. Rehg, "Visual analysis of high DOF articulated objects with application to hand tracking," Ph.D. dissertation, Carnegie Mellon University, Pittsburg, PA, 1995.

[19] E. Ueda, Y. Matsumoto, M. Imai, and T. Ogasawara, "Hand pose estimation using multi-viewpoint silhouette images," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 4, 2001, pp. 1989–1996. [Online]. Available: http://robotics.aist-nara.ac.jp/~etsuko-u/iros2001.pdf

[20] E. Sudderth, M. Mandel, W. Freeman, and A. Willsky, "Visual hand tracking using nonparametric belief propagation," in *IEEE Conference on Computer Vision and Pattern Recognition Workshop*, Jun. 2004, p. 189.

[21] M. de La Gorce, N. Paragios, and D. J. Fleet, "Model-based hand tracking with texture, shading and self-occlusions," in *IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 23–28, 2008, pp. 1–8.

[22] D. Abrahams and R. W. Grosse-Kunstleve, "Building hybrid systems with Boost.Python," *C/C++ Users Journal*, vol. 21, no. 7, pp. 29–36, July 2003. [Online]. Available: http://www.osti.gov/energycitations/product.biblio.jsp?osti_id=815409

[23] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, May/Jun. 2007.

[24] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, May/Jun. 2007.

[25] W.-Y. Chang, C.-S. Chen, and Y.-D. Jian, "Visual tracking in high-dimensional state space by appearance-guided particle filtering," *IEEE Transactions on Image Processing*, vol. 17, no. 7, pp. 1154–1167, 2008.

[26] M. B. Wakin, D. L. Donoho, H. Choi, and R. G. Baraniuk, "The multiscale structure of non-differentiable image manifolds," in *Proceedings of SPIE Wavelets XI*, M. Papadakis, A. F. Laine, and M. A. Unser, Eds., vol. 5914, 2005, p. 59141B.

[27] H. G. Barrow, J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf, "Parametric correspondence and chamfer matching: Two new techniques for image matching," in *Proc. 5th Int. Joint Conf. Artificial Intelligence*, 1977, pp. 659–663.

[28] K. E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver, "Fast computation of generalized voronoi diagrams using graphics hardware," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, 1999, pp. 277–286.

[29] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for on-line non-linear/non-Gaussian Bayesian tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, Feb. 2002.

[30] C.-C. Chang and C.-J. Lin. (2009, Nov.). LIBSVM: A library for support vector machines. [Online]. Available: http://www.csie.ntu.edu.tw/~cjlin/libsvm

[31] M. Balasubramanian, E. L. Schwartz, J. B. Tenenbaum, V. de Silva, and J. C. Langford, "The isomap algorithm and topological stability," *Science*, vol. 295, no. 5552, p. 7a, 2002.

[32] J. W. Tukey, *Exploratory Data Analysis*. Reading, MA: Addison-Wesley, 1977.