Knowing When to Put Your Foot Down

Leslie Ikemoto[*]
University of California, Berkeley

Okan Arikan[†]
University of Texas, Austin

David Forsyth[‡]
University of California, Berkeley
University of Illinois, Urbana-Champaign

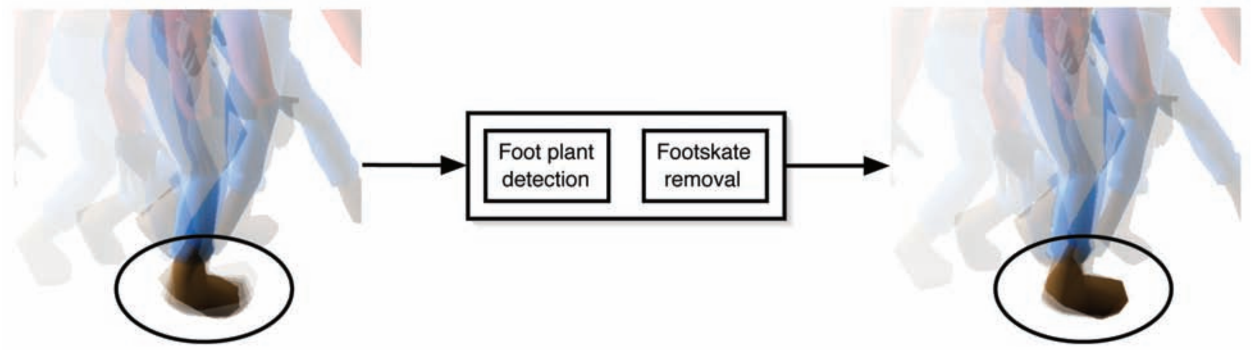


Figure 1: Motion editing can produce significant footskate (Section 1). On the **left** is an edited motion capture sequence. We superimpose partially translucent renderings of frames spaced evenly in time. As a result, a slowly moving part of the body – like the skating foot plant in this image – shows up as a dark region with blurry outlines. We introduce a robust oracle for detecting foot plants. When coupled with an off-the-shelf footskate remover, our system behaves like a black box (**center**) that cleans up motion at interactive rates (**right**). The foot is now planted firmly, as one can see from the sharp outline around the toe. Notice that the mild blur at the heel on the right results from the way the heel and then the toes are planted.

Abstract

Footskate, where a character's foot slides on the ground when it should be planted firmly, is a common artifact resulting from almost any attempt to modify motion capture data. We describe an online method for fixing footskate that requires no manual clean-up. An important part of fixing footskate is determining when the feet should be planted. We introduce an oracle that can automatically detect when foot plants should occur. Our method is more accurate than baseline methods that check the height or speed of the feet. These baseline methods perform especially poorly on noisy or imperfect data, requiring manual fixing. Once trained, our oracle is robust and can be used without manual clean-up, making it suitable for large databases of motion. After the foot plants are detected, we use an off-the-shelf inverse kinematics based method to maintain ground contact during each foot plant. Our foot plant detection mechanism coupled with an IK based fixer can be treated as a black box that produces natural-looking motion of the feet, making it suitable for interactive systems. We demonstrate several applications which would produce unrealistic motion without our method.

Keywords: Footskate, foot plant detection, motion synthesis

[*]e-mail: lesliei@cs.berkeley.edu
[†]e-mail: okan@cs.utexas.edu
[‡]e-mail: daf@cs.uiuc.edu

1 Introduction

Many interactive applications require realistic, high-quality character animation. This demand for realistic motion will almost certainly increase. However, motion usually competes for memory and other resources. Thus, heightened realism in motion cannot simply come from more data. Motion will probably need to be generated from a small dataset and modified to enrich variability. Motion blending ([Kovar and Gleicher 2004], [Rose et al. 1998]) and motion warping ([Witkin and Popović 1995]) are commonly used motion editing techniques. Another commonly used method rearranges motion frames to produce novel motion sequences. Motion graphs ([Kovar et al. 2002a], [Lee et al. 2002], [Arikan and Forsyth 2002]) encode which rearrangements are legal.

Algorithms that modify motion are undoubtedly useful, but they sometimes introduce undesirable artifacts that reduce believability. One such artifact is known as *footskate*, in which a character's foot slides on the ground after the character plants it instead of remaining firmly in place. Maintaining ground contact is an important aspect of realistic motion, so footskate is particularly objectionable.

Many authors have previously noted this problem and introduced successful techniques for fixing footskate (Section 2). However, none of these methods is completely automatic. To our knowledge, all require knowing *a priori* when feet should be planted. Typically, a simple test, such as checking the height and speed of the foot, is used to mark the frames. However, as far as we know, none of these

Copyright © 2006 by the Association for Computing Machinery, Inc.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481 or e-mail permissions@acm.org.
I3D 2006, Redwood City, California, 14–17 March 2006.
© 2006 ACM 1-59593-295-X/06/0003 $5.00

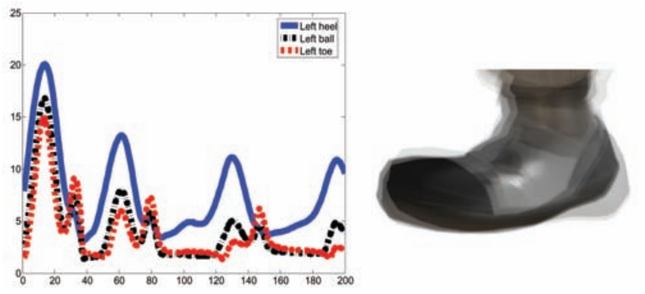


Figure 2: **Speed and height cannot reliably discriminate foot plants. Left:** A graph plotting the height of the joints in the left foot during 4 foot plants of walking motion in our original data set. There are actually double peaks when the foot is not planted, which may lead to mislabeling. **Right:** Despite what one might expect, foot plants are not perfectly stationary even in good motion capture data. For example, we show a left foot plant from our motion capture database. As in Figure 1, we overlay partially translucent renderings of frames spaced evenly in time. The foot is not stationary, so it appears with a blurry outline. If the foot was planted firmly, it would appear sharper (like the right-hand image in Figure 1).

simple tests works reliably (Figure 2). One reason they do not work all the time is that motion capture data is usually noisy. In addition, a character's skeleton is not an accurate representation of a human skeleton. Because these tests do not work reliably, all frames must be later checked by hand.

Clearly, checking frames by hand is not feasible for large datasets, which may contain many hundreds of thousands of frames. We present an automatic method for foot plant detection that is scalable and efficient (Section 3). By labeling a small set of frames, a user trains a classifier to detect when the foot should be planted. The classifier then automatically labels the remainder of the frames. Training time is short (our oracle required less than 3 minutes of labeled examples), and the classifier is accurate and efficient. We compare our results to results obtained using the simple tests commonly used to detect foot plants (Section 7). Once foot plants are detected, we can fix motion sequences containing footskate using an off-the-shelf real-time inverse kinematics solution. We use [Kovar et al. 2002b], but other techniques could be used instead.

We show that the combination of our foot plant detector and an off-the-shelf footskate remover can be treated as a black box that cleans up motion in real-time (Section 7). As we demonstrate, this black box can produce plausible results from motion modifications that change the original motion drastically. In interactive applications, one cannot check the frames produced by an algorithm before they are displayed. Our method is robust, making it suitable for interactive systems.

2 Previous Work

Synthesized motion must often meet constraints that define how it should look. The problem of fixing footskate can be viewed as the problem of computing positional and temporal constraints such that the feet stay planted when they should be.

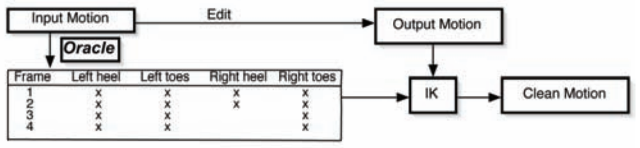
Much previous work on footskate clean-up successfully attacks the problem of finding and enforcing positional constraints. Positional constraints can come from the original data, as in ([Bodenheimer et al. 1997], [Shin et al. 2001]). Commercial motion processing packages often have the user set a parameter that specifies how well the end effectors should track the original data. Other papers describe techniques for computing positional constraints. [Kovar et al. 2002b] solves for the target position that results in minimum error over the window of frames in the foot plant.

Once positional constraints are determined, they can be enforced using a variant of inverse kinematics. [Kovar et al. 2002b] adjusts the joint angles in the legs, the root position, and the lengths of bones to satisfy foot plant constraints at each frame. [Lee and Shin 1999] use analytical IK solutions to reduce the number of parameters in their numerical IK computation. Applying IK separately at every frame ([Rose et al. 1998], [Kovar et al. 2002b]) may produce visual discontinuities in the motion. [Kovar et al. 2002b] proposes blending the IK adjustments into surrounding frames. Other techniques optimize for the smoothest motion that meets all of the constraints ([Lee and Shin 1999], [Gleicher 1997], [Gleicher 1998]).

Fewer techniques have addressed finding the proper temporal constraints for fixing footskate. A commonly used technique is to check the height of the foot, where it is assumed a foot plant occurs when the foot is close to the ground. However, this technique is easily fooled. For example, the test will probably return false results if the character skids to a stop. Another often used method is to check the speed of the foot. This technique is also unreliable. When motion capture data is taken, markers are placed on top of the feet, not on the bottom. Therefore, the markers usually have some speed even during foot plants. Furthermore, marker data is noisy, and skeletal fitting introduces further error. Therefore, joint speed is not a reliable indicator.

[Liu and Popović 2002] detect frames in which the feet are stationary. [Bindiganavale and Badler 1998] detect zero-crossings in acceleration space of end effectors. Like the height and speed tests discussed previously, both methods work well for non-noisy data. However, motion capture data is usually noisy, so the results from both of these tests can be unreliable on motion capture data.

Our method uses a classifier to detect when foot plants should occur. Hence, this paper continues a recent thread of using classifiers for motion synthesis problems. ([Ren et al. 2005], [Arikan et al. 2005], [Ikemoto and Forsyth 2004]) use classifiers to determine whether synthesized motion looks natural or not.



Frame	Left heel	Left toes	Right heel	Right toes
1	x	x	x	x
2	x	x	x	x
3	x	x		x
4	x	x		x

Figure 3: **Overview.** At run-time, an application edits an input motion in the motion database (e.g., using motion blending or warping). After editing, the output motion may contain footskate. To produce clean motion, an inverse kinematics solver needs to know in which frames the feet should be planted. Using such a labeling, the IK solver can modify the output motion to produce clean motion. This paper describes a method for obtaining such a labeling automatically. During pre-processing, an oracle labels each frame as to whether the left heel, left toes, right heel, or right toes are planted. This labeling can be stored efficiently (requiring only 4 bits per frame). At run-time, we simply look up the labels.

3 Overview

Given a frame of motion, we would like an oracle to detect automatically if the feet are planted. Our system detects two types of foot plants – heel plants and toe plants. (However, it is easy to have the system detect other types of foot plants as well.)

We can formulate this as a classification problem. Given a feature vector describing the motion of the feet over a short period of time about a frame (Figure 4), we seek a labeling of the frame.

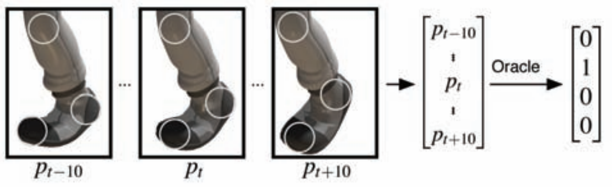

p_{t-10}
p_t
p_{t+10}
Oracle
$\begin{bmatrix} p_{t-10} \\ \vdots \\ p_t \\ \vdots \\ p_{t+10} \end{bmatrix}$
$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$

Figure 4: **The oracle.** In order to classify a frame, we compute the positions of the knees, ankles, and toes in a 21-frame window centered about the frame (**left**). (In the figure, p_t is the knee, ankle, and toe positions at time equals t.) We then stack these positions $(p_{t-10},\dots,p_t,\dots p_{t+10})$ into a feature vector (**center**). The oracle accepts these feature vectors and outputs a 4-bit labeling corresponding to which parts of which feet are supposed to be planted in that frame (**right**). (For clarity, we show only the left foot in this figure, but positions from both feet are used in the feature vector.)

We describe the feature set we use in Section 4. To build the oracle, we train a k nearest-neighbors classifier interactively in a semi-supervised process. The training procedure is described in Section 5, and the classifier is described in Section 6.

We use our foot plant oracle to label the frames in the motion database during pre-processing. At run-time, an application may edit the motions in the database, potentially introducing footskate. To clean up an edited motion, we look up the foot plant labels on the original frame. The edited motion and the labels are then given to the inverse kinematics solver, which outputs clean motion. Figure 3 illustrates our system.

Each frame requires only a 4-bit label (left heel, left toe, right heel, and right toe), so the labels can be stored efficiently. In addition, the oracle is fast and accurate, so pre-processing the database can be done rapidly and reliably, or the oracle can be used on-the-fly.

4 Features

The goal of the oracle is to decide whether a frame of motion contains a foot plant or not. The oracle makes its decision based on the skeletal configuration at the frame and a description of the dynamics. To encode the dynamics, we use the skeletal configurations in a window of nearby frames. (We use a 21 frame window.) The oracle is a function F such that:

$$F(p_{t-10},\dots,p_t,\dots,p_{t+10}) = \begin{bmatrix} L_{left\_heel} \\ L_{left\_toe} \\ L_{right\_heel} \\ L_{right\_toe} \end{bmatrix}$$

where p_t is the position of the joints in the skeleton at time t. L_a is 1 if the oracle thinks that part of the foot is planted, and 0 otherwise. Figure 4 contains a schematic of the oracle.

In practice, we do not need to put all of the joints into the feature vector. The configuration of the upper body and upper parts of the legs are largely irrelevant. We use the positions of the knees, ankles, and toes, since these joints seem most useful for discriminating foot plants. The position and orientation of the figure on the plane is also irrelevant, so we transform the window of frames so that the central frame is at the origin with a zero orientation about the y-axis.

5 Training

We approximate the oracle F with a k nearest neighbors classifier (where k is 10). Other classifiers (such as SVM) would probably also work well.

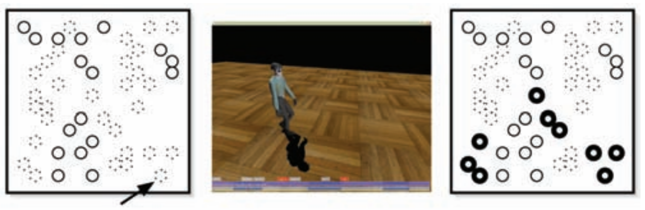


Figure 5: **Oracle training.** The oracle is approximated by a k nearest neighbors classifier which uses a small example pool. Imagine that the feature space is 2-dimensional. If we plot the feature vector for every frame in the motion database, it would look similar to the figure on the **left**. The solid circles denote frames which have already been hand-labeled, and form our example pool. Dotted circles denote the other still unlabeled frames. The system picks a frame which is maximally different from the already labeled frames (i.e., it picks the dotted circle which is most distant from the solid circles). The oracle labels the 200-frame sequence containing this frame and displays its results to the user (**center**). The user checks whether this labeling is correct. If it is not, the user corrects the labels, and the oracle puts the 200 labeled frames into the example pool (**right**). If it is correct and the user believes the oracle can correctly discriminate foot plants, the training can stop.

The classifier is trained from a set of hand-labeled examples. The system displays a motion, and the user annotates each frame with a combination of the labels `left heel plant, left toe plant, right heel plant,` and `right toe plant`.

The classifier is trained interactively (as in [Arikan et al. 2003]). The system chooses a 200-frame motion sequence at random, which the user annotates. The system then trains the classifier on the annotated examples, and picks another sequence. The classifier labels this sequence and presents its results to the user. The user can check whether this labeling is correct. Errors typically show up as incorrectly localized foot plant boundaries. If the labeling is wrong, the user can correct the labels, and have the system retrain the classifier. Using on-line learning exposes the state of the classifier. The user can stop training whenever she or he thinks the classifier can discriminate correctly. The training procedure is illustrated in in Figure 5.

We use importance sampling to select examples for classification. Once the oracle has seen an example frame annotated by the user, frames that are similar to that example are likely to be labeled correctly. However, dissimilar frames may be classified incorrectly. Therefore, the system chooses examples which are maximally different from those previously chosen and user annotated.

The importance sampling function assigns high importance to frames that are far away from the example frames in the feature space. Nearby frames get low importance. The motion database is first split into non-overlapping 200-frame sequences. The importance function assigns a value to each sequence based on the distance to the previously annotated examples. Let $d_{min}(q)$ be the minimum Euclidean distance between the feature vector of a query frame q to the example frames in the oracle's training set. The importance sampling chooses the 200-frame sequence which contains the frame with the highest d_{min}.

We have found our sampling scheme effective. In particular, our example pool should contain examples from different types of motion (e.g., running, dancing, skipping). The dynamics of different types of motion are fairly distinct from each other. Therefore, we expect our sampling scheme to pick frames from each of the different types of motions in our database, and this occurs in practice. However, there are many other good schemes for selecting classification examples.

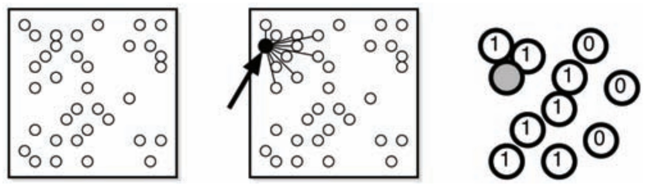


Figure 6: **Classifying a frame.** During training, we populate a *k* nearest neighbors classifier with examples of frames containing and not containing foot plants (**left**). Each example is labeled with whether it contains a left heel plant, left toe plant, right heel plant, and/or right toe plant. To classify a query frame, we locate its *k* nearest neighbors (**center**). Determining whether the query contains a left heel plant entails looking at the left heel plant labels on its neighbors (**right**), and averaging.

We double the size of the example pool by mirroring the example sequences and their labels, such that left foot plants become right foot plants and vice versa. The training set can include motions from multiple actors, provided that all the motions were fit onto the same skeleton.

Unfortunately, errors in training data are unavoidable. It is very difficult for a user to precisely localize the frame in which a foot plant begins and the frame in which a foot plant ends. This means that nearly identical frames occurring at the boundaries of foot plants are liable to have contradictory labels. We will address this problem later when we conduct queries (Section 6).

6 Classification

Once the oracle is trained, labeling a motion dataset is an automatic process. To get a labeling for a frame *q*, we examine the labels on each of the *k* closest samples. Let us first determine whether there is a left heel plant. Each of the *k* samples carries a binary label of 1 if it contains a left heel plant or a 0 if it does not. We add these labels together and divide by *k* to get the left heel plant label for *q*. This gives us a real-valued label between 0 (if the classifier is certain *q* does not contain a left heel plant) and 1 (if the classifier is certain *q* does). We do the same for the left toe, right heel, and right toe. See Figure 6 for a schematic diagram.

We would now like to obtain a binary labeling for *q*. Unfortunately, we cannot simply threshold the real-valued label. As discussed in Section 5, some of the training data of frames towards the beginnings and ends of foot plants will probably be contradictory. Therefore, the real-valued query labels tend to fluctuate rapidly towards the beginnings and ends of foot plants. The fluctuations may occur around a threshold value, so we cannot threshold the labels.

A key observation is that if a frame contains a foot plant, its neighboring frames are likely to contain foot plants as well. Therefore, if the threshold value in a frame is high indicating a foot plant, we can lower the threshold we use in the next frame.

We use two thresholds, t_{high} (0.6) and t_{low} (0.4). If a query value is greater than t_{high}, it is automatically labeled as a foot plant. If it is below t_{low}, it is labeled as not a foot plant. If a query value is between the two thresholds and the previous frame was classified as a foot plant, this frame is also classified as a foot plant. This technique successfully fixes the fluctuations at the boundaries of foot plants. It is commonly known as hysteresis ([Canny 1986]).

Note that our method is robust to noise in the motion data. As long as the user classifies the example set consistently, the classifier will label frames in the database consistently. Therefore, our method is sensitive to user-noise, but not to data noise.

7 Results

Our oracle was trained on 9,410 user-labeled example frames (under 3 minutes of motion data sampled at 60 Hz), so training our classifier took little time. We double the number of examples by mirroring the example sequences and annotations (Section 5).

The oracle is efficient. It takes about 22.5 ms on a 3.2 GHz Pentium 4 to classify a single frame (i.e., compute the feature vector, locate the nearest neighbors, and determine the frame's labels). While the nearest neighbor search dominates the running time, it is unlikely that the user will annotate many example frames. Therefore, we used a brute-force *k*-nearest neighbors classifier, where the query frame is compared against every other frame. However, the time required to find the nearest neighbors will increase linearly with the number of examples in the training set. If a large example pool is available, an approximate nearest-neighbors algorithm such as locality sensitive hashing [Indyk and Motwani 1998] could speed up the search.

We compare our detection algorithm to two baseline algorithms that check the speed and height of the feet. These algorithms are commonly used to detect foot plants. During a foot plant, the foot should be stationary and touching the ground. Thus, these baseline algorithms threshold the speed and height of the feet. We fit these thresholds by trial-and-error. The video that accompanies this paper contains several examples of our results. We compare our results visually with results from our baseline algorithms.

We hand labeled 2000 frames of data against which to test the oracles. Each frame carries 4 labels (left heel/toe plant and right heel/toe plant), so there are a total of 8000 labels in our test set.

The speed-based classifier determines 57.45% of them correctly (4596 out of 8000). The height-based classifier determines 57.00% of them correctly (4560 out of 8000). Our classifier determines 90.78% of them correctly (7262 out of 8000).

While it may seem that our classifier does not score perfectly, accuracy rates are not a perfect means for assessing error. Some types of errors are more problematic than others (e.g., labels can fluctuate during a foot plant, or a foot plant can be too short or too long).

Our classifier actually detects every foot plant correctly and makes no problematic errors. Accuracy is not perfect because the boundaries of the foot plants differ from the hand labeled examples. It is difficult for a user to precisely localize the boundaries of foot plants (cf. Section 5), so our hand labeled data may be mislabeled. Our test set contains about 50 foot plants. The labels on the first and last 1 to 5 frames of a foot plant appear to be ambiguous, so we expect 50-250 frames – or 200-1000 labels – to be difficult to classify. Therefore, the error our oracle makes is within expected bounds. In fact, we can get a better estimate of the error by seeing what the oracle thinks and checking if the user agrees. We checked the 8000 labels our oracle output by hand, and disagreed with only 6 them. This yields an alternative accuracy of 99.93%.

Varying the parameters to our oracle does not change the accuracy rates very much (Table 1).

8 Conclusions and Future Work

One of the limitations of our method is that the feature set we use is tied to a particular skeleton. When motion capture data is taken, markers are placed at different positions on the body, so skeletal joint positions usually vary from dataset to dataset. It would be interesting to change the feature set to one that was skeleton-independent, but we have not yet explored this point.

It is natural to try scaling the features. We have tried using our oracle on another dataset, applying a uniform scaling parameter to the feature vectors to account for the difference in the sizes of the legs. Our original data set contained motion of a graduate student.

Varying the number of nearest neighbors					
1	**5**	**15**	**20**	**30**	**40**
91.00%	91.10%	91.06%	90.90%	91.03%	91.10%

Varying the window size					
3	**11**	**31**	**41**	**61**	**121**
91.03%	90.90%	90.79%	91.05%	90.76%	90.51%

Changing the joints in the feature vector	
Adding the root's position	**Removing the knee positions**
90.70%	90.80%

Table 1: **Varying parameters has little effect on accuracy.** This table contains accuracy rates for our classifier on the same test set used in Section 7. We varied parameters independently. The first row of each table corresponds to the value of the parameter being changed, and the second row reports accuracy rates. These rates vary little between each other, and are very similar to the accuracy rate of our classifier (90.78%) using 10 nearest neighbors, a 21-frame window, and using the knee, ankle, and toe positions of each leg in the feature vector. Therefore, our method does not require careful parameter tuning.

We used our classifier – without additional training – to label the foot plants of a professional football player performing specialized football maneuvers. The two data sets were taken at different studios and fit onto different skeletons.

Our test set contained 595 hand-labeled frames. Out of the 2380 foot plant labels in the set, our classifier marked 2253 correctly, giving us 94.66% accuracy. However, even though our accuracy rate is higher than on our original dataset, our classifier actually performed worse. Our classifier made some problematic labeling errors towards the boundaries of foot plants – the labels sometimes fluctuated, and they sometimes began a few frames too early or extended a few frames too far. Errors like these are relatively unsurprising, since the relative scales of the joints between the two skeletons differ (e.g., the heel joint on the football dataset is significantly higher than on the graduate student dataset). It seems probable that non-uniform scaling would perform better.

Our method has 2 potential sources of latency: classification and footskate removal. The classifier requires the configuration of the legs 10 frames beyond the current frame ($\frac{1}{6}$ of a second at 60 frames per second), and the footskate removal technique we use ([Kovar et al. 2002b]) requires 30 frames of look-ahead to locate a target position for the foot. We fix the first problem by doing classification as a preprocessing step. The oracle can label every frame in the database during pre-processing; at run-time, we simply retrieve the labels for the current frame. Since each frame only requires a 4-bit label, the storage cost is low. The second source of latency — footskate removal — is more problematic, and is the second limitation of our approach. However, we believe that with appropriate regression techniques we can locate a suitable target position for the foot without requiring a long lookahead. We plan to explore this point further.

We have described a fast and reliable method for detecting foot plants. Our technique requires only a small labeled training set and does not need careful parameter tuning. When combined with an inverse kinematics solver, it can robustly clean-up footskate artifacts in edited motion, making it suitable for interactive applications. The accompanying video demonstrates our technique.

Acknowledgments

We would like to thank the Berkeley graphics group for their helpful comments and suggestions. This work was generously supported by ONR N00014-01-1-0890 and by a donation of motion capture data from Sony Computer Entertainment America.

References

ARIKAN, O., AND FORSYTH, D. 2002. Interactive motion generation from examples. In *Proceedings of ACM SIGGRAPH 02, 2002.*

ARIKAN, O., FORSYTH, D., AND O'BRIEN, J. 2003. Motion synthesis from annotations. *SIGGRAPH*.

ARIKAN, O., FORSYTH, D. A., AND O'BRIEN, J. F. 2005. Pushing people around. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM Press, New York, NY, USA, 59–66.

BINDIGANAVALE, R., AND BADLER, N. I. 1998. Motion abstraction and mapping with spatial constraints. *Lecture Notes in Computer Science 1537*.

BODENHEIMER, B., ROSE, C., ROSENTHAL, S., AND PELLA, J. 1997. The process of motion capture: Dealing with the data. In *Computer Animation and Simulation '97. Proceedings of the Eurographics Workshop*.

CANNY, J. 1986. A computational approach to edge detection. *IEEE T. Pattern Analysis and Machine Intelligence 8*, 6, 679–698.

GLEICHER, M. 1997. Motion editing with spacetime constraints. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*.

GLEICHER, M. 1998. Retargetting motion to new characters. In *Proceedings of SIGGRAPH 1998*, 33–42.

IKEMOTO, L., AND FORSYTH, D. A. 2004. Enriching a motion collection by transplanting limbs. In *SCA '04: Proceedings of the 2004 Symposium on Computer animation*, ACM Press, New York, NY, USA, 99–108.

INDYK, P., AND MOTWANI, R. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of 30th STOC*, 604–613.

KOVAR, L., AND GLEICHER, M. 2004. Automated extraction and parameterization of motions in large data sets. *ACM Trans. Graph. 23*, 3, 559–568.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 473–482.

KOVAR, L., SCHREINER, J., AND GLEICHER, M. 2002. Footskate cleanup for motion capture editing. In *Proceedings of the 2002 ACM Symposium on Computer Animation (SCA)*.

LEE, J., AND SHIN, S. Y. 1999. A hierarchical approach to interactive motion editing for human-like figures. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 39–48.

LEE, J., CHAI, J., REITSMA, P., HODGINS, J., AND POLLARD, N. 2002. Interactive control of avatars animated with human motion data. In *Proceedings of SIGGRAPH 2002*, 491–500.

LIU, C. K., AND POPOVIĆ, Z. 2002. Synthesis of complex dynamic character motion from simple animations. *ACM Transactions on Graphics 21*, 3 (July), 408–416.

REN, L., PATRICK, A., EFROS, A. A., HODGINS, J. K., AND REHG, J. M. 2005. A data-driven approach to quantifying natural human motion. *ACM Trans. Graph. 24*, 3, 1090–1097.

ROSE, C., COHEN, M. F., AND BODENHEIMER, B. 1998. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Applications 18*, 5, 32–41.

SHIN, H. J., LEE, J., SHIN, S. Y., AND GLEICHER, M. 2001. Computer puppetry: An importance-based approach. *ACM Trans. Graph. 20*, 2, 67–94.

WITKIN, A., AND POPOVIĆ, Z. 1995. Motion warping. In *Proceedings of SIGGRAPH 1995*, 105–108.