Controlled Animation of Video Sprites

Arno Schödl   Irfan A. Essa

Georgia Institute of Technology
GVU Center / College of Computing
Atlanta, GA 30332-0280
http://www.cc.gatech.edu/cpl/projects/videotextures/

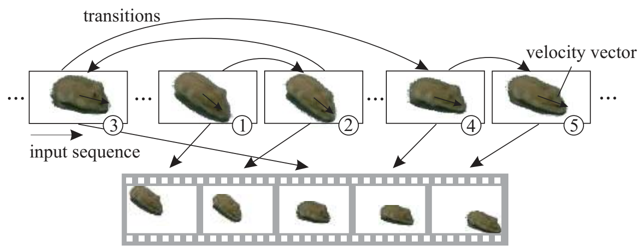

Figure 1: Creating an animation from video sprite samples.

Abstract

We introduce a new optimization algorithm for video sprites to animate realistic-looking characters. Video sprites are animations created by rearranging recorded video frames of a moving object. Our new technique to find good frame arrangements is based on repeated partial replacements of the sequence. It allows the user to specify animations using a flexible cost function. We also show a fast technique to compute video sprite transitions and a simple algorithm to correct for perspective effects of the input footage. We use our techniques to create character animations of animals, which are difficult both to train in the real world and to animate as 3D models.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.4.9 [Image Processing and Computer Vision]: Applications—;

Keywords: video sprites, video textures, character animation, optimization.

1 Introduction

Research has shown that captured real images and video can be used for modeling and rendering, which are traditionally realms of entirely synthetic 3D models. Captured video has also been used for animation, most markedly for animations that are very difficult to do with traditional tools, such as facial motion [Bregler et al. 1997]. This paper focuses on the image-based generation of character animations using video sprites. Video sprites are created by extracting a subject from a video sequence as a sequences of sprites, and rearranging this sequence into a new animation.

In this paper, we focus on the animation of animals. The two conventional options are either filming a real animal, which requires intensive training of the animal and is not always possible, or creating and animating a 3D model, which is costly because natural looking results are hard to achieve. Video sprites are a compelling alternative. Recording the animal is cheap, natural-looking motion comes for free because original animal footage is used, and the animation is synthesized by the computer, with little human assistance.

This paper focusses on new algorithms for generating better video sprite animations. The only video sprite animation shown in [Schödl et al. 2000], a fish, is constricted in two ways. First, perspective distortions are not corrected at all. This approach works fine for the fish floating in water, but more interesting subjects usually walk on land and perspective distortion is a significant problem. In this paper, we present a simple algorithm that effectively compensates for changes in perspective, so that recordings of the character at one location can be reused at any other location. Second, and more importantly, we introduce a new flexible technique for sprite control. The control techniques in the literature [Schödl and Essa 2001] only allow either interactive control or specification of a simple motion path. In computer animation using modeled characters, it is common to exert more flexible control by defining a cost function [Witkin and Kass 1988]. The animation system optimizes the parameters of the animation to minimize the cost function, and thereby generates the desired animation. In this paper we extend this approach to video sprites, presenting an algorithm to optimize the animation with respect to any user-defined cost function.

1.1 Related Work

Character animation has a long tradition in computer graphics. Most realistic-looking animations have been generated using 3D models. The oldest character animation technique for models is keyframing [Lasseter 1987; Dana 1992]. 3D models are posed manually by the artist at some keyframes, and their motion is smoothly interpolated in between the keyframes using splines. The technique has the advantage of being relatively intuitive to use, because the artist directly controls the output.

Instead of defining keyframes, motions can also be defined using cost functions. The internal motion parameters are then optimized automatically to minimize this function [Witkin and Kass 1988; Cohen 1992; Ngo and Marks 1993]. In this paper, we are using such cost functions to control the animation of video sprites. Cost functions can be combined with physical simulations to ensure that all generated motion parameters are physically plausible [Hodgins et al. 1995; Laszlo et al. 1996; Hodgins 1998].

The third option for controlling characters is to use data captured from the real world. A motion capture system produces a low-dimensional representation of a character's pose over time by tracking markers, which drives a 3D model. Several researchers have proposed algorithms to transform motion capture data in order to extend its range of applicability [Rose et al. 1998; Popovic and Witkin 1999; Gleicher 1998]. Others have proposed markerless motion capture through analysis of video to drive animations of 3D models, both for faces [Essa et al. 1996; Terzopoulos and Waters 1993; Williams 1990] and for whole body models [Bregler and Malik 1998].

A disadvantage of the described techniques is the need for accurate 3D models of the characters to animate. Image-based rendering avoids such models by using images captured from real scenes to create new renderings. The early work mainly deals with static scenes, which are rendered from different positions and viewing directions. For a purely rotating camera, this amounts to creating panoramic views by image stitching [Chen 1995; Szeliski and Shum 1997]. If some limited camera motion is desired, one can either directly create a representation of the plenoptic function [McMillan and Bishop 1995; Gortler et al. 1996; Levoy and Hanrahan 1996] or use approximate knowledge about 3D geometry [Seitz and Dyer 1996; Debevec et al. 1996].

To overcome the limitation of still scenes, the computer vision community has developed some scene motion models. Most of them work best with unstructured motion of fluids like water or smoke. They can be roughly classified as either parametric techniques [Szummer and Picard 1996; Soatto et al. 2001; Fitzgibbon 2001] or non-parametric techniques [Wei and Levoy 2000; Bar-Joseph 1999]. The non-parametric techniques were developed on the basis of prior work on image texture synthesis [Wei and Levoy 2000; Bonet 1997; Heeger and Bergen 1995].

Recently, sample-based techniques that play back pieces of the original video have been introduced to create animations. They are better suited for structured motions than other techniques, because they preserve most of the image structure. Bregler et al.[1997] create lip motion for a new audio track from a training video of the subject speaking by replaying short subsequences of the training video fitting best to the sequence of phonemes. Cosatto shows a similar system for more general facial motion [Cosatto and Graf 1998]. More recently, video textures [Schödl et al. 2000] have been introduced as a more general approach suitable for many types of motions. Video sprites [Schödl and Essa 2001] are an extension to video textures aimed at animating moving objects. Recently, the video texture idea has inspired approaches that rearrange motion capture data using video texture-like algorithms to generate new animations [Lee et al. 2002; Arikan and Forsyth 2002; Kovar et al. 2002].

1.2 Overview

The paper mainly focusses on a new sprite optimization technique, but also describes the rest of the video sprite pipeline that is necessary to turn video sprites into a practical animation tool. We focus on animating animals because they are naturally quite difficult to direct and therefore our techniques for generating realistic animations give the most compelling results.

The animation pipeline starts by capturing videos of the desired characters, for this paper a hamster and a fly, in front of a green screen. Using background subtraction and chroma keying techniques, we extract the animals from the background (Section 2). This results in sequences of sprites, little bitmaps with opacity information, that are the basic elements of our animation. After correcting perspective distortion of the sprites, especially apparent size changes due to motion in the depth direction, we find sprite pairs that are suitable to serve as transition points from one location in the video to another (Section 3). Then we define a cost function that describes the sprite animation that we want to generate (Section 5) and find a frame sequence that minimizes this cost function (Section 4). Finally, the video sprites are composited onto the desired background. Section 6 shows some results for both the hamster and the fly input sequences. The whole pipeline is shown in Figure 2.

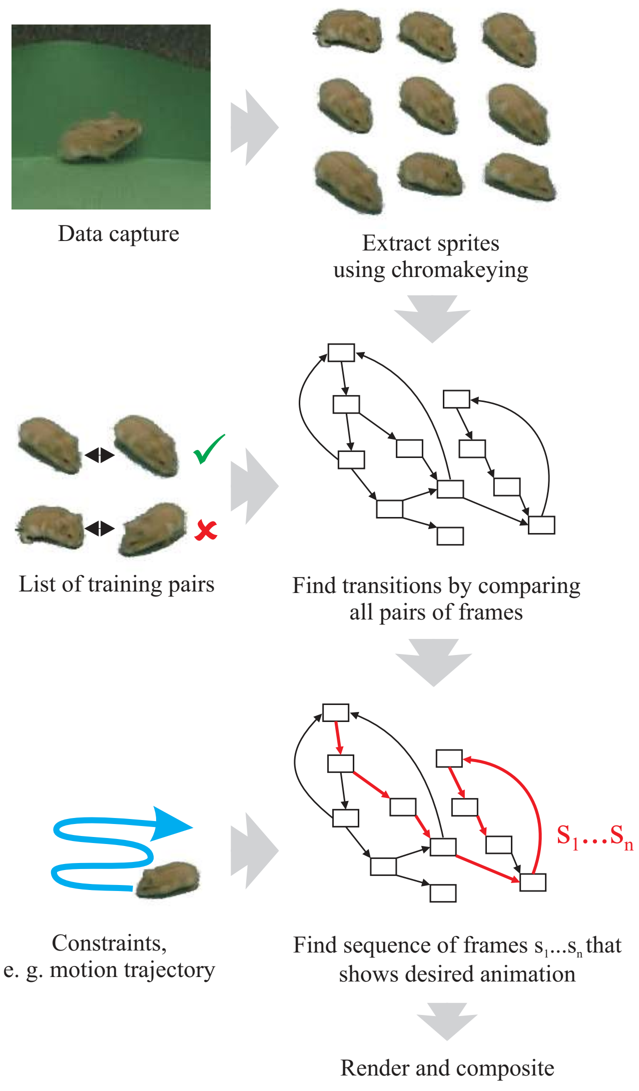
Data capture
Extract sprites using chromakeying
List of training pairs
Find transitions by comparing all pairs of frames
Constraints, e. g. motion trajectory
Find sequence of frames $s_1 \ldots s_n$ that shows desired animation
$S_1 \ldots S_n$
Render and composite

Figure 2: Overview of our processing pipeline.

2 Sprite extraction

To generate video sprites, the subjects are first recorded with one or multiple cameras, while they are roaming suitable enclosures lined with green paper for chroma keying. Figure 3 shows the hamster and fly enclosures. We then use chroma keying to separate the subjects from the background. During video capture, the subject often either touches the walls of its enclosure or casts shadows onto them. All footage where the subject is not entirely surrounded by the ground plane is automatically discarded to avoid visual artifacts. Since both subjects, like most animals, are laterally symmetrical, the amount of available training data is doubled by horizontally mirroring the footage. During the recording, the subjects are lit diffusely from above. Thus, using mirrored footage and mixing footage from multiple cameras does not introduce any shading artifacts.

Figure 3: Examples of raw input video frames.

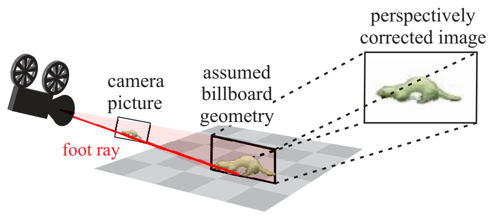

Figure 4: Correction for size varying with depth and perspective distortion.

2.1 Perspective correction

The recorded subject is likely to be seen from different perspectives; most importantly the sprite size will change when the subject moves away from or toward the camera. Since both the hamster and the fly move on a flat ground plane, we compensate for that effect by reprojecting the camera image onto a hypothesized simple world geometry. The recording cameras are calibrated internally and externally in relation to the ground plane on which the subject moves [Bradksi and Pisarevsky 2000; Zhang 1998]. We assume that the subject is a plane, which is standing vertically on the ground plane, facing the camera like a billboard. We determine the subject's location in 3D by intersecting a ray from the camera through the lowest sprite point with the ground plane (Figure 4). Projecting the camera image onto this hypothesized geometry corrects for most perspective effects, and this corrected image is used for the subsequent steps of the algorithm. In the final rendering step, we perform perspective projection of the billboard image back into the virtual camera.

3 Defining transition costs

The main principle of generating video sprite animations is to find pairs of frames in the video sequence that are sufficiently similar so that they can be used for transitions from one part of the sequence to another without noticeable jerks. After correcting all sprite images for perspective correction, we compute pair-wise sprite image differences to find suitable transitions.

To decide similarity between frames, we compute six difference features for every sprite pair. It is hard to design a function that expresses human perception of similarity for a particular group of images. So we learn this function from examples, following [Schödl and Essa 2001]. We classify about 1000 pairs of sprite images manually as acceptable or not acceptable for a transition, including roughly the same number of examples from each of the two categories. Then we train a linear classifier which operates on the difference features.

To speed up computation, two different classifiers are actually used. The first linear classifier operates on a group of four features. These are precomputed for all sprites and can be compared efficiently: sprite velocity, average sprite color, sprite area, and sprite eccentricity. Only to sprite pairs that pass this first test is an additional linear classifier applied. This classifier uses two more difference features, the pixel-by-pixel differences in alpha value and the pixel color of the perspective-corrected sprite images. Both require access to the actual image data for every comparison and are therefore relatively slow to evaluate. This two-stage approach typically reduces computation time by about two orders of magnitude.

$$\begin{tikzcd} \arrow[r] & \boxed{i} \arrow[r] \arrow[d, leftrightarrow, "D_{i(j-1)}"'] \arrow[dr, dashed, "\text{transition } i\to j"] & \boxed{i+1} \arrow[r] \arrow[d, leftrightarrow, "D_{(i+1)j}"] & {} \\ \arrow[r] & \boxed{j-1} \arrow[r] & \boxed{j} \arrow[r] & {} \end{tikzcd}$$

Figure 5: To make a transition from frame i to frame j, frame i should be similar to frame $j-1$ and frame j similar to frame $i+1$.

All sprite pairs that either classifier rejected are no longer considered for transitions. If the pair of samples i and j is kept, we use the value of the second linear classifying function, operating on all six features, as a measure for visual difference D_{ij}. We use the visual difference to define the cost of a transition. For a transition from frame i to frame j, frame i's successor must be similar to frame j and frame j's predecessor must be similar to frame i. Thus, the cost $C_{i\to j}$ of a transition $i \to j$ is (Figure 5)

$$C_{i\to j} = D_{(i+1)j} + D_{i(j-1)}. \tag{1}$$

4 Controlling Video Sprites

After computing the available transitions between frames, the linear sequence structure turns into a directed graph. The nodes of the graph represent frames, and the directed edges represent transitions from one frame to another. A frame sequence $S = \{s_1, \ldots, s_n\}$ is represented by a path in this graph. The transitions or edges on this path have to be visually smooth, which we express by a smoothness cost function C_s:

$$C_s(S) = \sum_{i=1}^{n-1} C_{s_i \to s_{i+1}} \tag{2}$$

We also want the sequence of frames or nodes of the graph to show the desired animation, for example a walk from left to right. We can express this by specifying an additional control cost function C_c. Designing a control cost function is a trade-off between more precise sprite control and smoother transitions. Good control cost functions only constrain the aspects of motion that are critical for telling the story. In Section 5 we give some examples of useful control cost functions.

The total sequence cost C is the sum of the two components:

$$C(S) = C_s(S) + C_c(S). \tag{3}$$

To create a controlled sprite animation, we want to find the sequence S of frames for which the total cost C is minimal:

$$S = \arg\min_S C(S). \tag{4}$$

4.1 Limitations of current video sprite optimization techniques

Schödl and Essa [2001] have suggested two video sprite optimization techniques that work well for some classes of motions, but have serious limitations for more general animations. The first algorithm is a simple iterative learning technique known as *Q-learning* [Kaelbling et al. 1996], which is only practical for interactive control. For scripted animations, they suggest *beam search*, which works by maintaining a set $\mathcal{S}$ of partial transition sequences. In every step of the algorithm, a new set of sequences $\mathcal{S}'$ is generated, which contains for each sequence $S = \{s_1 ... s_n\} \in \mathcal{S}$ all possible extensions of that sequence by another frame s_{n+1}. Then the cost of all extended sequences in $\mathcal{S}'$ is computed, and the N best sequences in $\mathcal{S}'$ are kept and further extended in the next iteration. Over all iterations, the algorithm keeps track of the best sequence encountered so far.

This algorithm works well for cost functions that have a tight coupling between transition choices and cost, i.e., where the choice for the current transition has no grave unforeseeable consequences in the future. But for many other cost functions, beam search fails. In particular, among sequences of a length k, the function C must prefer those sequences that are good starting points, which, when further extended, lead to a good minimum of C. The number of subsequences that beam search concurrently evaluates only provides a very limited look-ahead, because the number of possible sequence continuations quickly multiplies with every additional transition.

Given this limited look-ahead, evaluating the potential of an extended sequence based on a limited starting subsequence is difficult for many interesting cost functions. For example, if the goal is to be at a certain point at a certain time, it is unknown a-priori what options the sprite has for getting to the goal and how long it will take. Other important examples are cost functions that link multiple video sprites, driving sprites to avoid collisions with each other or to walk in a desired formation. In these cases, the frame sequences of the sprites involved should be optimized jointly. Here the cost function C operates on two or more sequences, and instead of extending a single sequence by a single transition, two or more sequences must be extended by any combination of two or more transitions. This exponentially increases the number of transition variants and further reduces the average look-ahead.

4.2 Optimization by repeated subsequence replacement

The main problem of beam search is its sequential mode of operation. The algorithm effectively commits to a sequence of transitions in order, and there is no possibility to go back and make corrections. As an alternative, this paper presents a hill-climbing optimization that instead optimizes the sequence as a whole, and determines the transitions in no particular order. It starts with an initial random sequence, and repeatedly tries to change the sequence. It only keeps those changes that lower the cost. It would be nice to use global optimization techniques instead of greedy hill-climbing, but unfortunately the computational demand of these techniques that work by keeping track of multiple sequence candidates or allowing temporary cost increases to avoid local minima is excessive for the problem at hand.

A key component of the algorithm is its method to make changes to an existing transition sequence. The most obvious way to make a change to a sequence is to replace one frame of the sequence with another one. Unfortunately, this is rarely possible. Given the sequence $s_1, \ldots, s_n$, replacing s_k with some new frame t results in the sequence $s_1, \ldots, s_{k-1}, t, s_{k+1}, \ldots, s_n$, which requires two new transitions $s_{k-1} \to t$ and $t \to s_{k+1}$. It is highly unlikely that both transitions are available and have reasonably low cost. Instead of replacing a single frame in the sequence, it is better to replace a whole subsequence. Given two locations in the sequence k and l, we replace the subsequence $s_k, \ldots, s_l$ with a sequence that connects s_k and s_l in some new way.

In order to find a good subsequence to connect s_k and s_l, we

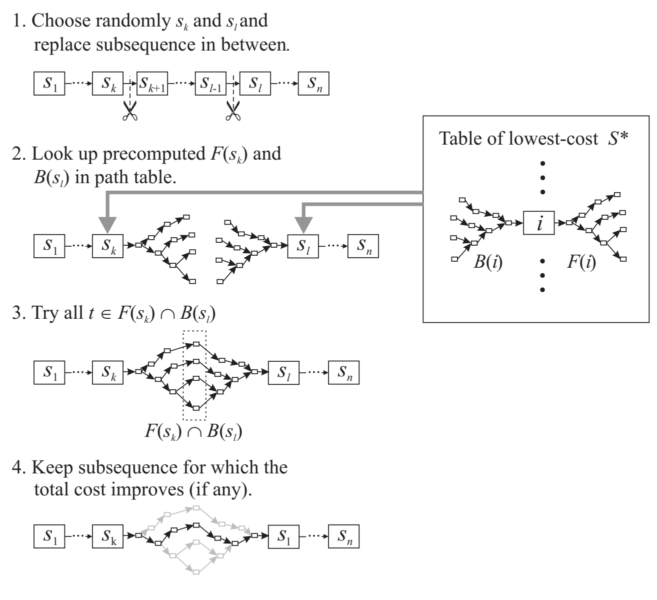
1. Choose randomly s_k and s_l and replace subsequence in between.
s_1
s_k
s_{k+1}
s_{l-1}
s_l
s_n
Table of lowest-cost S^*
2. Look up precomputed $F(s_k)$ and $B(s_l)$ in path table.
i
$B(i)$
$F(i)$
3. Try all $t \in F(s_k) \cap B(s_l)$
$F(s_k) \cap B(s_l)$
4. Keep subsequence for which the total cost improves (if any).

Figure 6: A single optimization step, replacing a middle subsequence.

heuristically pick a cost function $\hat{C}$ that is context-free, so it can be formulated as a sum of per-transition costs and approximates the total sequence cost function C. In practice, we simply use the smoothness component of C:

$$\hat{C} = C_s. \tag{5}$$

Since $\hat{C}$ can be formulated as a sum of per-transition costs, for any pair of frames i and j, Dijkstra's algorithm can precompute the lowest-cost transition sequence between frames i and j with respect to $\hat{C}$, $S^*(i \to j)$. Unfortunately, $S^*(s_k \to s_l)$ is not a good subsequence to connect s_k and s_l, because for any chosen cost function $\hat{C}$, any subsequence of a path optimal with respect to $\hat{C}$ will itself be optimal. After enough replacements, the whole sequence will be optimal with respect to $\hat{C}$, but not necessarily with respect to C and no further replacement would introduce any more changes. Instead, we pick a state t and replace $s_k, \ldots, s_l$ with $S^*(s_k \to t) S^*(t \to s_l)$ (of course, t is inserted in the middle only once), which allows the generation of a large variety of sequences.

Which frames t are suitable to generate subsequences? If $\hat{C}$ approximates C well, to minimize C it is reasonable to consider candidates t that lead to small values of $\hat{C}$. Before starting the optimization, we use Dijkstra's algorithm to precompute (for every input frame i) the set of N frames t for which the sequences $S^*(i \to t)$ have the lowest costs with respect to $\hat{C}$. Those t form the set $F(i)$ (for *f*orward transitions). Similarly, for every frame i, the N ts with the lowest-cost paths $S^*(t \to i)$ form the set $B(i)$ (for *b*ackward transitions) (Figure 6). Typically, each set forms a tree that has long branches of many continuous frames with few transitions. After choosing a subsequence $s_k, \ldots, s_l$ to replace, the optimization tries all t in the intersection of the respective forward and backward sets $F(s_k) \cap B(s_l)$ until it finds a t that lowers the total sequence cost C. Since the subsequence $s_k, \ldots, s_l$ that is replaced usually has few transitions (with little accumulated cost), s_k is likely to lie in $B(s_l)$ and s_l is likely to lie in $F(s_k)$, and thus $F(s_k) \cap B(s_l)$ is not empty. If the intersection is empty or there is no subsequence that improves the cost, the algorithm tries a different pair (k, l).

So far, we have only discussed how to change a middle piece of the sequence. The beginning and end can be modified even more easily. After picking a frame l, the algorithm simply replaces the beginning of the sequence $s_1, \ldots, s_l$ by $S^*(t \to s_l)$, trying all $t \in$

$B(s_l)$. Similarly, to replace the end $s_k, \ldots, s_n$ by $S^*(s_k \rightarrow t)$, it tries all $t \in F(s_k)$.

To perform the optimization, the algorithm starts with a sequence that is either randomly initialized or for example computed by beam search. It then repeatedly picks random pairs k and l ($k \leq l$) and tries different replacements, as described above. This process repeats until it cannot find any more replacements that improve the total cost C.

5 Control cost functions

In this section we describe useful control cost components to achieve specific effects of motion control. Since video sprite animations are generated by playing back from a set of recorded samples with a limited number of transitions, there is only a discrete number of smooth sprite sequences available. When applying an animation constraint, the animator has to strike a balance between achieving the desired motion and the animation smoothness. This makes it more critical than for 3D model animations to keep the control cost function as lenient as possible.

The control cost is calculated with respect to the whole animation sequence which can involve multiple sprites. For each time step i, the current state of a sprite is a triplet $(p, v, f)_i$, where p is the location of the sprite, v is the sprite's velocity and f the input frame where the current sprite image is copied from. A particular control cost c at time step i, c_i, depends on the sprite states of the one or more sprites in the animation:

$$c_i = \mathrm{f}((p_1, v_1, f_1)_i, (p_2, v_2, f_2)_i, \ldots) \tag{6}$$

The total control cost C_c is the sum of costs over all constraints and time steps. In the following paragraphs, we describe the types of control functions used to generate results for some common motion constraints.

5.1 Location constraint

Here is a very simple cost function to constrain a sprite to a particular location at time step i:

$$c_i(p) = \gamma(p - p_{\text{target}})^2 \tag{7}$$

p_{target} is the target location of the sprite and γ is a coefficient controlling the strength of the contraint. By calculating p_{target} based on some other sprite's location and velocity, this constraint can also be used for sprite formations.

5.2 Path constraint

This constraint makes the sprite move down a path that has been defined using line segments. It must be implemented somewhat differently than the location constraint. The latter would either, if it is weak, not assure that the sprite takes a straight path, or if it is strong, favor fast sprite movement to the target over smooth movements. The constraint must penalize any deviation from the defined path and favor the right movement direction.

We use the same cost function as Schödl and Essa [2001], constraining only the motion path, but not the velocity magnitude or the motion timing. The path is composed of line segments and the algorithm keeps track of the line segment that the sprite is currently expected to follow. For each frame, one function component penalizes the Euclidian distance between the current sprite position and the current line segment. The second one causes the sprite to maintain the right direction toward the end of the path. This second component depends on the angle between the current sprite velocity and the line segment:

$$c_i(p, v) = \delta|\angle v - \angle v_{\text{target}}| + \phi((p - p_{\text{target}}) \cdot \perp v_{\text{target}})^2, \tag{8}$$

where v_{target} and p_{target} are the *unit* direction vector and the endpoint, respectively, of the current line segment. $\angle$ is an operator to compute the angle of a vector, and δ and ϕ are again strength coefficients. In addition, we use a special constraint for the last frame of the sequence that penalizes any portion of the path that the sprite has not reached at all. It measures the distance missing on the current line segment and the lengths of all path segments that have not yet been reached. This penalty drives the sequence to get increasingly longer until the sprite actually moves along the whole path.

5.3 Anti-collision constraint

This pairwise constraint prevents two sprites from colliding with each other. We use a simple conical error surface to avoid collisions:

$$c_i(p_1, p_2) = \mu \max(d_{\text{min}} - \|p_1 - p_2\|_2, 0). \tag{9}$$

d_{min} is the desired minimum distance between the two sprites. To avoid numerical instabilities, we do not use errors that rise faster, like hyperbolics. A large strength coefficient μ still gives the desired effect. This constraint can also be used to control the elbowroom needed for flocking. Narrow functions let characters bump into each other, wide ones leave plenty of personal space.

5.4 Frame constraint

Sometimes it is useful to force a sprite to be copied from a certain group of frames G, which for example show the sprite in a particular pose. This can be achieved by simply adding a fixed cost λ per frame if the condition is not met:

$$c_i(f) = \begin{cases} 0 & \text{if } f \in G, \\ \lambda & \text{otherwise} \end{cases} \tag{10}$$

6 Results

After optimization, the system plays back the generated sprite sequence and projects the sprite images into a virtual camera, using simple depth-sorting to handle overlapping sprites. The animation is then composited onto a natural background. To get a clean composite, the system uses the alpha channel and contact shadows extracted using chroma keying.

6.1 The hamster

The hamster data set was generated from 30 minutes of hamster footage, yielding 15,000 usable sprite frames, or 30,000 after mirroring. The first example animation (Figure 7a) uses a path constraint that makes the hamster run around in a circle. The video on the website shows various stages of the optimization, in increments of 30 seconds of processing time.

The second example (Figure 7b) shows the hamster loitering at the screen center, but he manages to move out of the way in time before a large pile of conference proceedings crashes down. The optimization by subsequence replacement can handle constraints that require a larger look-ahead than beam search would be able to handle. To start, we use a weak location constraint to keep the hamster near the center of the screen. At 4 seconds later – the time of impact – we use a stronger hamster location constraint that favors the far right corner, without specifying the specific path he should take. The optimization correctly anticipates the change in cost and moves the hamster out of harm's way.

In the next scene (Figure 7c) we add another hamster that is moving to the left side, crossing the first hamster's path. We use similar location constraints for both hamsters, but add an anti-collision constraint between the two. We optimize the sum of both hamsters' cost in turns of approximately 100 random changes, always changing only one path while keeping the other constant.

The optimization can also incorporate character actions. The next sequence (Figure 7d) shows the hamster entering the scene, and after following a curved path that we created using the path constraint, he stands up on two legs and looks at the audience. We manually classified all desirable input sprite frames where the hamster is standing up facing the viewer, and once the hamster reaches the bottom of the path, we add a frame constraint to the cost function for 2 seconds (Section 5.4). Unfortunately, the desired frames showing the hamster standing up are very rare among the recorded sprite frames. We therefore found it necessary to adapt the approximating cost function $\hat{C}$ used for computing replacement subsequences. We look for the condition when the cost function C includes the frame constraint for the majority of frames in the subsequence that has been selected for replacement. We then use a modified approximating cost function $\hat{C}$ that also includes the frame constraint to compute S^* (Section 4.2).

In the last hamster sequence (Figure 7e), we add a second hamster that is constrained to stay in a formation with the first using a location constraint. The target location changes for each frame depending on the first hamster's location and velocity. Additionally, the second hamster also shares the first one's frame constraint, which makes the second stand up at the same time as the first.

6.2 The fly

For the second subject, the fly, we use one hour of raw video footage, from which we extract, after automatically discarding extended periods with no motion, 28,000 usable sprite frames, doubled to 56,000 by mirroring. A special challenge was that despite our efforts to eliminate ranges of frames with no motion, most sprite images still showed a fly moving in place instead of crawling. To generate the animation, we need a more animated fly, which we create by modifying $\hat{C}$ to favor frames with large velocity.

The fly example is in fact created not by fixing the sprites' start locations as in the other examples but by fixing their end locations to match the grid, leaving the start positions unconstrained. We use animation length constraints for each fly that penalize animation sequences under 100 frames (3.3 seconds) and many pairwise anti-collision constraints to generate the animation. In the final animation, only two pairwise anti-collision constraints are violated, but these violations are hard to pick out in the busy scene. This animation took one day to compute, much longer than any of the other results.

7 Future work and conclusion

Animating video sprites is still a challenging problem. Certain footage like the fly data set only has a limited number of useful motions and only a few transitions connecting them. Thus not all motions can be animated. In addition to the hamster and the fly, we have also captured 2 hours each of a cat and a ferret, but both sequences lacked enough transitions to even make a straight left-right walk possible because the animals hardly moved, probably scared by the all-green environment.

A promising avenue of future research in video sprites is real-time data capture, sprite extraction and transition finding. Currently capturing more than a couple of hours of sprite video is impractical, and a lot of time is wasted on storage and retrieval of large amounts of video. Storing only usable sprite data would increase memory and storage efficiency. Data capture would take place in a more natural larger environment that the animal would roam at leisure.

Warping techniques could allow transitions between less similar sprites. A major challenge is that warping is slow and can never be applied to all frame pairs, so any fast rejection algorithm must predict well whether the warp is worth trying. In the work on computing transitions, we were often surprised how sprites with very similar features such as color and shape can still be very different in more subtle aspects such as pose or perspective. To simplify the problem, future research could be directed more specifically toward a certain class of video sprites, for example humans. However, expectations for human animations are much higher than for animals. For example, human facial and hand motions are often important for convincingly conveying a story.

When recording video sprites, we undistort for perspective effects mainly to obtain more sprite transitions by making sprite images more similar to each other. We assume that we can undo the distortion to the extent that a frame recorded from one viewing angle can be used for any other viewing angle. This is a reasonable assumption for long focal lengths, relatively small depth variations of the scene, and a virtual animation camera with a similar azimuth as the recording camera. If these assumptions do not hold, one option is to use a better 3D geometry approximation than the planar one used in this paper to better compensate for perspective. But accurate 3D geometry is difficult to obtain, especially if the subject is deforming non-rigidly while it is moving. Alternatively, we could add an additional constraint into the sequence optimization which finds sprite sequences that are not only smooth and show the desired action, but are also showing the object from a viewing angle similar to the one of the virtual camera. Using multiple cameras with different azimuths would increase the amount of available sprite data, seen from different viewing angles. This approach would be conceptually similar to unstructured lumigraphs [Buehler et al. 2001].

In conclusion, we have presented new techniques for character animation with video sprites. We introduced a fast rejection technique for computing video sprite transitions, presented a correction for perspective effects, and most importantly, showed a new powerful algorithm to generate animations, which are specified by cost functions and impossible to create using previous techniques.

References

ARIKAN, O., AND FORSYTH, D. A. 2002. Interactive motion generation from examples. In *Proceedings of SIGGRAPH 2002*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series.

BARJOSEPH, Z. 1999. *Statistical learning of multi-dimensional textures*. Master's thesis, The Hebrew University of Jerusalem.

BONET, J. S. D. 1997. Multiresolution sampling procedure for analysis and synthesis of texture images. *Proceedings of SIGGRAPH 97* (August), 361–368. ISBN 0-89791-896-7. Held in Los Angeles, California.

BRADKSI, G., AND PISAREVSKY, V. 2000. Intel's computer vision library: Applications in calibration, stereo, segmentation, tracking, gesture, face, and object recognition. In *In Proceedings of IEEE Computer Vision and Pattern Recognition Conference 2000*, vol. II, II:796–797. Demonstration Paper.

BREGLER, C., AND MALIK, J. 1998. Tracking people with twists and exponential maps. In *Proceedings of Computer Vision and Pattern Recognition 1998*, 8–15.

BREGLER, C., COVELL, M., AND SLANEY, M. 1997. Video rewrite: Driving visual speech with audio. *Proceedings of SIGGRAPH 97* (August), 353–360. ISBN 0-89791-896-7. Held in Los Angeles, California.

BUEHLER, C., BOSSE, M., MCMILLAN, L., GORTLER, S. J., AND COHEN, M. F. 2001. Unstructured lumigraph rendering. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, 425–432. ISBN 1-58113-292-1.

CHEN, S. E. 1995. Quicktime VR - an image-based approach to virtual environment navigation. *Proceedings of SIGGRAPH 95* (August), 29–38. ISBN 0-201-84776-0. Held in Los Angeles, California.

COHEN, M. F. 1992. Interactive spacetime control for animation. *Computer Graphics (Proceedings of SIGGRAPH 92) 26*, 2 (July), 293–302. ISBN 0-201-51585-7. Held in Chicago, Illinois.

COSATTO, E., AND GRAF, H. P. 1998. Sample-based synthesis of photo-realistic talking heads. *Computer Animation '98* (June). Held in Philadelphia, Pennsylvania, USA.

DANA, P. 1992. Issues and techniques for keyframing transformations. In *Graphics Gems III*. Academic Press, Boston, 121–123. ISBN 0-12-409673-5.

DEBEVEC, P. E., TAYLOR, C. J., AND MALIK, J. 1996. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Proceedings of SIGGRAPH 96* (August), 11–20. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

ESSA, I., BASU, S., DARRELL, T., AND PENTLAND, A. 1996. Modeling, tracking and interactive animation of faces and heads using input from video. In *Proceedings of Computer Animation Conference 1996*, IEEE Computer Society Press, 68–79.

FITZGIBBON, A. W. 2001. Stochastic rigidity: Image registration for nowhere-static scenes. In *Proceeding of IEEE International Conference on Computer Vision 2001*, I: 662–669.

GLEICHER, M. 1998. Retargeting motion to new characters. *Proceedings of SIGGRAPH 98* (July), 33–42. ISBN 0-89791-999-8. Held in Orlando, Florida.

GORTLER, S. J., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. F. 1996. The lumigraph. *Proceedings of SIGGRAPH 96* (August), 43–54. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

HEEGER, D. J., AND BERGEN, J. R. 1995. Pyramid-based texture analysis/synthesis. *Proceedings of SIGGRAPH 95* (August), 229–238. ISBN 0-201-84776-0. Held in Los Angeles, California.

HODGINS, J. K., WOOTEN, W. L., BROGAN, D. C., AND O'BRIEN, J. F. 1995. Animating human athletics. *Proceedings of SIGGRAPH 95* (August), 71–78. ISBN 0-201-84776-0. Held in Los Angeles, California.

HODGINS, J. K. 1998. Animating human motion. *Scientific American 278*, 3 (Mar.), 64–69 (Intl. ed. 46–51).

KAELBLING, L. P., LITTMAN, M. L., AND MOORE, A. P. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research 4*, 237–285.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. In *Proceedings of SIGGRAPH 2002*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series.

LASSETER, J. 1987. Principles of traditional animation applied to 3d computer animation. *Computer Graphics (Proceedings of SIGGRAPH 87) 21*, 4 (July), 35–44. Held in Anaheim, California.

LASZLO, J. F., VAN DE PANNE, M., AND FIUME, E. 1996. Limit cycle control and its application to the animation of balancing and walking. *Proceedings of SIGGRAPH 96* (August), 155–162. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

LEE, J., CHAI, J., REITSMA, P. S. A., HODGINS, J., AND POLLARD, N. 2002. Interactive control of avatars animated with human motion data. In *Proceedings of SIGGRAPH 2002*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series.

LEVOY, M., AND HANRAHAN, P. 1996. Light field rendering. *Proceedings of SIGGRAPH 96* (August), 31–42. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

MCMILLAN, L., AND BISHOP, G. 1995. Plenoptic modeling: An image-based rendering system. *Proceedings of SIGGRAPH 95* (August), 39–46. ISBN 0-201-84776-0. Held in Los Angeles, California.

NGO, J. T., AND MARKS, J. 1993. Spacetime constraints revisited. *Proceedings of SIGGRAPH 93* (August), 343–350. ISBN 0-201-58889-7. Held in Anaheim, California.

POPOVIC, Z., AND WITKIN, A. 1999. Physically based motion transformation. *Proceedings of SIGGRAPH 99* (August), 11–20. ISBN 0-20148-560-5. Held in Los Angeles, California.

ROSE, C., COHEN, M. F., AND BODENHEIMER, B. 1998. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics & Applications 18*, 5 (September - October), 32–40. ISSN 0272-1716.

SCHÖDL, A., AND ESSA, I. A. 2001. Machine learning for video-based rendering. In *Advances in Neural Information Processing Systems*, MIT Press, USA, T. K. Leen, T. G. Dietterich, and V. Tresp, Eds., vol. 13, 1002–1008.

SCHÖDL, A., SZELISKI, R., SALESIN, D. H., AND ESSA, I. A. 2000. Video textures. *Proceedings of SIGGRAPH 2000* (July), 489–498. ISBN 1-58113-208-5.

SEITZ, S. M., AND DYER, C. R. 1996. View morphing: Synthesizing 3d metamorphoses using image transforms. *Proceedings of SIGGRAPH 96* (August), 21–30. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

SOATTO, S., DORETTO, G., AND WU, Y. N. 2001. Dynamic textures. In *Proceeding of IEEE International Conference on Computer Vision*, II: 439–446.

SZELISKI, R., AND SHUM, H.-Y. 1997. Creating full view panoramic mosaics and environment maps. *Proceedings of SIGGRAPH 97* (August), 251–258. ISBN 0-89791-896-7. Held in Los Angeles, California.

SZUMMER, M., AND PICARD, R. W. 1996. Temporal texture modeling. In *Proceeding of IEEE International Conference on Image Processing*, vol. 3, 823–826.

TERZOPOULOS, D., AND WATERS, K. 1993. Analysis and synthesis of facial image sequences using physical and anatomical models. *IEEE Transactions on Pattern Analysis and Machine Intelligence 15*, 6, 56–579.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. *Proceedings of SIGGRAPH 2000* (July), 479–488. ISBN 1-58113-208-5.

WILLIAMS, L. 1990. Performance-driven facial animation. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4 (August), 235–242. ISBN 0-201-50933-4. Held in Dallas, Texas.

WITKIN, A., AND KASS, M. 1988. Spacetime constraints. *Computer Graphics (Proceedings of SIGGRAPH 88) 22*, 4 (August), 159–168. Held in Atlanta, Georgia.

ZHANG, Z. 1998. A flexibe new technique for camera calibration. Tech. Rep. 98-71, Microsoft Research. www.research.microsoft.com/~zhang/Calib/.

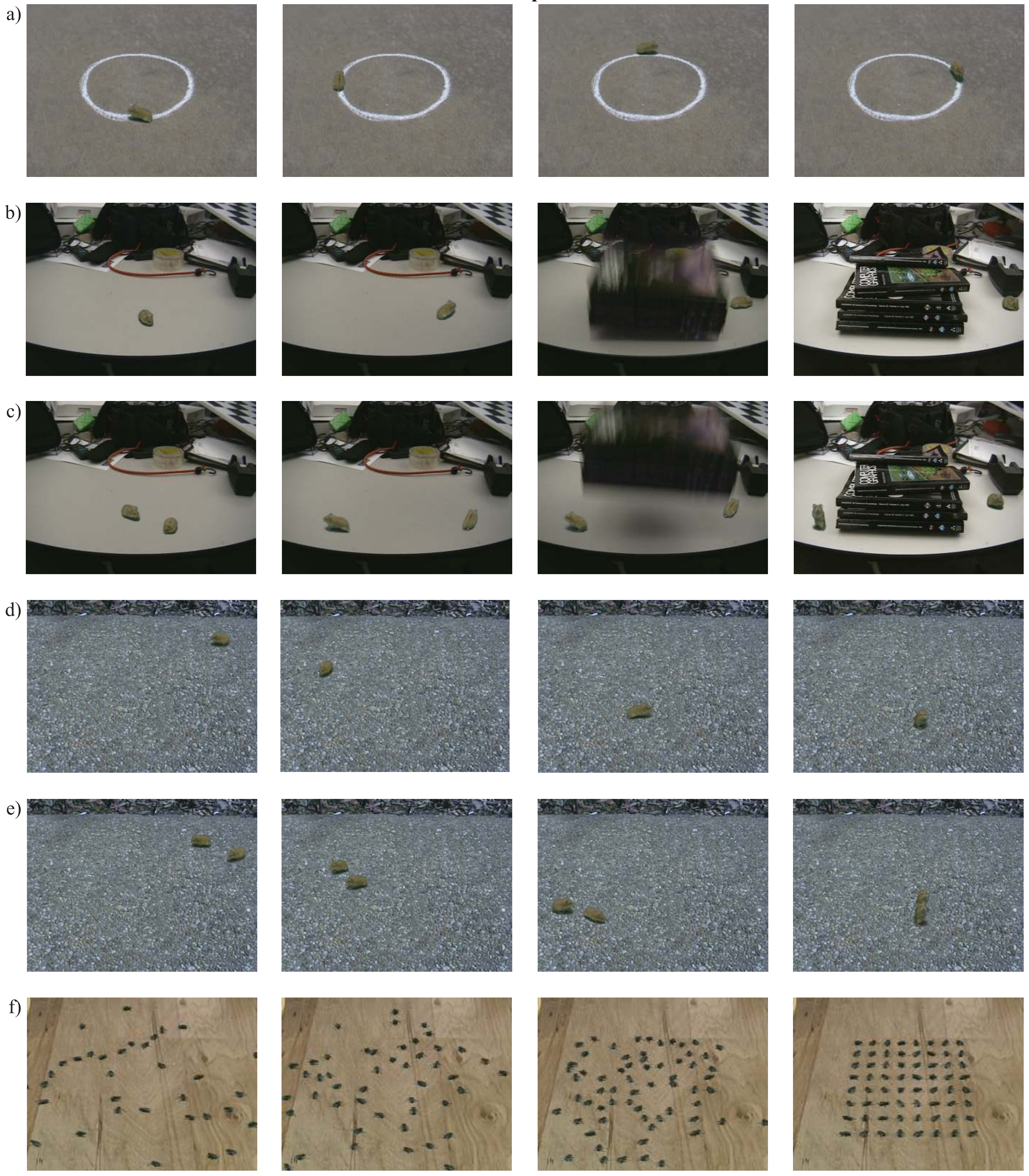

Figure 7: The generated hamster and fly animations. Sequence a) shows the hamster walking in a circle. The video illustrates the optimization process by showing various intermediate steps. Sequence b) demonstrates the ability of the iterated subsequence replacement algorithm to anticipate changes in the error function, in this case guiding the hamster out of harm's way in time. Sequence c) is similar to b), but involves two hamsters that are constrained not to collide. Sequences c) and d) include a sprite action at the end; the hamster stands up, looking at the audience. The second hamster follows the first and imitates its actions. Sequence e) shows 64 flies arranging on a grid. It demonstrates collision avoidance on a large scale.