Machine Learning for Video-Based Rendering

Arno Schödl
arno@schoedl.org

Irfan Essa
irfan@cc.gatech.edu

Georgia Institute of Technology
GVU Center / College of Computing
Atlanta, GA 30332-0280, USA.

Abstract

We present techniques for rendering and animation of realistic scenes by analyzing and training on short video sequences. This work extends the new paradigm for computer animation, *video textures*, which uses recorded video to generate novel animations by replaying the video samples in a new order. Here we concentrate on *video sprites*, which are a special type of video texture. In video sprites, instead of storing whole images, the object of interest is separated from the background and the video samples are stored as a sequence of alpha-matted sprites with associated velocity information. They can be rendered anywhere on the screen to create a novel animation of the object. We present methods to create such animations by finding a sequence of sprite samples that is both visually smooth and follows a desired path. To estimate visual smoothness, we train a linear classifier to estimate visual similarity between video samples. If the motion path is known in advance, we use beam search to find a good sample sequence. We can specify the motion interactively by precomputing the sequence cost function using Q-learning.

1 Introduction

Computer animation of realistic characters requires an explicitly defined model with control parameters. The animator defines keyframes for these parameters, which are interpolated to generate the animation. Both the model generation and the motion parameter adjustment are often manual, costly tasks.

Recently, researchers in computer graphics and computer vision have proposed efficient methods to generate novel views by analyzing captured images. These techniques, called *image-based rendering*, require minimal user interaction and allow photorealistic synthesis of still scenes[3].

In [7] we introduced a new paradigm for image synthesis, which we call *video textures*. In that paper, we extended the paradigm of image-based rendering into *video-based rendering*, generating novel animations from video. A video texture

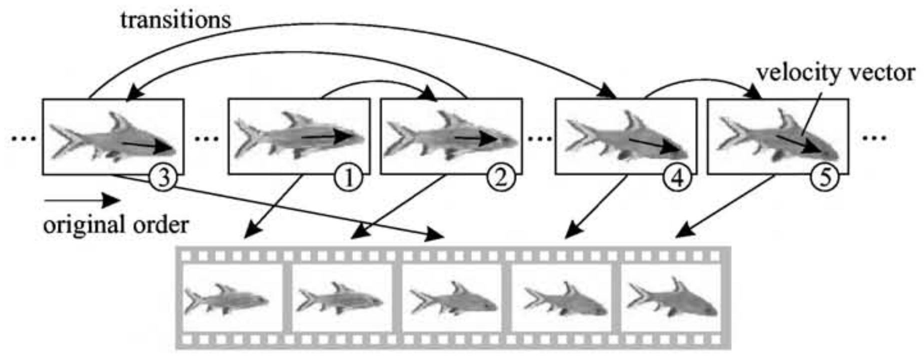

Figure 1: An animation is created from reordered video sprite samples. *Transitions* between samples that are played out of the original order must be visually smooth.

turns a finite duration video into a continuous infinitely varying stream of images. We treat the video sequence as a collection of image samples, from which we automatically select suitable sequences to form the new animation.

Instead of using the image as a whole, we can also record an object against a blue-screen and separate it from the background using background-subtraction. We store the created opacity image (alpha channel) and the motion of the object for every sample. We can then render the object at arbitrary image locations to generate animations, as shown in Figure 1. We call this special type of video texture a *video sprite*.

A complete description of the video textures paradigm and techniques to generate video textures is presented in [7]. In this paper, we address the controlled animation of video sprites. To generate video textures or video sprites, we have to optimize the sequence of samples so that the resulting animation looks continuous and smooth, even if the samples are not played in their original order. This optimization requires a visual similarity metric between sprite images, which has to be as close as possible to the human perception of similarity. The simple L_2 image distance used in [7] gives poor results for our example video sprite, a fish swimming in a tank. In Section 2 we describe how to improve the similarity metric by training a classifier on manually labeled data [1].

Video sprites usually require some form of motion control. We present two techniques to control the sprite motion while preserving the visual smoothness of the sequence. In Section 3 we compute a good sequence of samples for a motion path scripted in advance. Since the number of possible sequences is too large to explore exhaustively, we use beam search to make the optimization manageable.

For applications like computer games, we would like to control the motion of the sprite interactively. We achieve this goal using a technique similar to Q-learning, as described in Section 4.

1.1 Previous work

Before the advent of 3D graphics, the idea of creating animations by sequencing 2D sprites showing different poses and actions was widely used in computer games. Almost all characters in fighting and jump-and-run games are animated in this fashion. Game artists had to generate all these animations manually.

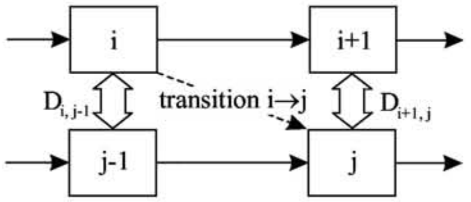

Figure 2: Relationship between image similarities and transitions.

There is very little earlier work in research on automatically sequencing 2D views for animation. Video Rewrite [2] is the work most closely related to video textures. It creates 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. To our knowledge, nobody has automatically generated an object animation from video thus far.

Of course, we are not the first applying learning techniques to animation. The NeuroAnimator [4], for example, uses a neural network to simulate a physics-based model. Neural networks have also been used to improve visual similarity classification [6].

2 Training the similarity metric

Video textures reorder the original video samples into a new sequence. If the sequence of samples is not the original order, we have to insure that *transitions* between samples that are out of order are visually smooth. More precisely, in a transition from sample i to j, we substitute the successor of sample i by sample j and the predecessor of sample j by sample i. So sample i should be similar to sample $j-1$ and sample $i+1$ should be similar to sample j (Figure 2).

The distance function D_{ij} between two samples i and j should be small if we can substitute one image for the other without a noticeable discontinuity or "jump". The simple L_2 image distance used in [7] gives poor results for the fish sprite, because it fails to capture important information like the orientation of the fish. Instead of trying to code this information into our system, we train a linear classifier from manually labeled training data. The classifier is based on six features extracted from a sprite image pair:

- difference in velocity magnitude,
- difference in velocity direction, measured in angle,
- sum of color L_2 differences, weighted by the minimum of the two pixel alpha values,
- sum of absolute differences in the alpha channel,
- difference in average color,
- difference in blob area, computed as the sum of all alpha values.

The manual labels for a sprite pair are binary: visually acceptable or unacceptable. To create the labels, we guess a rough estimator and then manually correct the classification of this estimator. Since it is more important to avoid visual glitches than to exploit every possible transition, we penalize false-positives 10 times higher than false-negatives in our training.

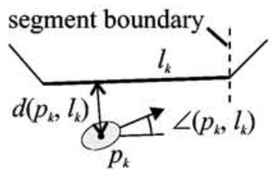

Figure 3: The components of the path cost function.

All sprite pairs that the classifier rejected are no longer considered for transitions. If the pair of samples i and j is kept, we use the value of the linear classifying function as a measure for visual difference D_{ij}. The pairs i, j with $i = j$ are treated just as any other pair, but of course they have minimal visual difference. The cost for a transition T_{ij} from sample i to sample j is then $T_{ij} = \frac{1}{2} D_{i,j-1} + \frac{1}{2} D_{i+1,j}$.

3 Motion path scripting

A common approach in animation is to specify all constraints before rendering the animation [8]. In this section we describe how to generate a good sequence of sprites from a specified motion path, given as a series of line segments. We specify a cost function for a given path, and starting at the beginning of the first segment, we explore the tree of possible transitions and find the path of least cost.

3.1 Sequence cost function

The total cost function is a sum of per-frame costs. For every new sequence frame, in addition to the transition cost, as discussed in the previous section, we penalize any deviation from the defined path and movement direction. We only constrain the motion path, not the velocity magnitude or the motion timing because the fewer constraints we impose, the better the chance of finding a smooth sequence using the limited number of available video samples.

The path is composed of line segments and we keep track of the line segment that the sprite is currently expected to follow. We compute the error function only with respect to this line segment. As soon as the orthogonal projection of the sprite position onto the segment passes the end of the current segment, we switch to the next segment. This avoids the ambiguity of which line segment to follow when paths are self-intersecting.

We define an animation sequence $(i_1, p_1, l_1), (i_2, p_2, l_2)...(i_N, p_N, l_N)$ where i_k, $1 \leq k \leq N$, is the sample shown in frame k, p_k is the position at which it is shown, and l_k is the line segment that it has to follow. Let $d(p_k, l_k)$ be the distance from point p_k to line l_k, $v(i_k)$ the estimated velocity of the sprite at sample i_k, and $\angle(v(i_k), l_k)$ is the angle between the velocity vector and the line segment. The cost function C for the frame k from this sequence is then

$$C(k) \;=\; T_{i_{k-1}, i_k} \;+\; w_1 \, |\angle(v(i_k), l_k)| + \; w_2 \; d(p_k, l_k)^2, \tag{1}$$

where w_1 and w_2 are user-defined weights that trade off visual smoothness against the motion constraints.

3.2 Sequence tree search

We seed our search with all possible starting samples and set the sprite position to the starting position of the first line segment. For every sequence, we store the total cost up to the current end of the path, the current position of the sprite, the current sample and the current line segment.

Since from any given video sample there can be many possible transitions and it is impossible to explore the whole tree, we employ beam search to prune the set of sequences after advancing the tree depth by one transition. At every depth we keep the 50000 sequences with least accumulated cost. When the sprite reaches the end of the last segment, the sequence with lowest total cost is chosen. Section 5 describes the running time of the algorithm.

4 Interactive motion control

For interactive applications like computer games, video sprites allow us to generate high-quality graphics without the computational burden of high-end modeling and rendering. In this section we show how to control video sprite motion interactively without time-consuming optimization over a planned path.

The following observation allows us to compute the path tree in a much more efficient manner: If w_2 in equation (1) is set to zero, the sprite does not adhere to a certain path but still moves in the desired general direction. If we assume the line segment is infinitely long, or in other words is indicating only a general motion direction l, equation (1) is independent of the position p_k of the sprite and only depends on the sample that is currently shown. We now have to find the lowest cost path through this set of states, a problem which is solved using Q-learning [5]: The cost F_{ij} for a path starting at sample i transitioning to sample j is

$$F_{ij} = T_{ij} \ + w_1 \left|\angle(v(j), l)\right| \ + \ \alpha \ \min_k F_{jk}. \tag{2}$$

In other words, the least possible cost, starting from sample i and going to sample j, is the cost of the transition from i to j plus the least possible cost of all paths starting from j. Since this recursion is infinite, we have to introduce a decay term $0 \leq \alpha \leq 1$ to assure convergence. To solve equation (2), we initialize with $F_{ij} = T_{ij}$ for all i and j and then iterate over the equation until convergence.

4.1 Interactive switching between cost functions

We described above how to compute a good path for a given motion direction l. To interactively control the sprite, we precompute F_{ij} for multiple motion directions, for example for the eight compass directions. The user can then interactively specify the motion direction by choosing one of the precomputed cost functions.

Unfortunately, the cost function is precomputed to be optimal only for a certain motion direction, and does not take into account any switching between cost functions, which can cause discontinuous motion when the user changes direction. Note that switching to a motion path without any motion constraint (equation (2) with $w_1 = 0$) will never cause any additional discontinuities, because the smoothness constraint is the only one left. Thus, we solve our problem by precomputing a cost function that does not constrain the motion for a couple of transitions, and then starts to constrain the motion with the new motion direction. The response delay allows us to gracefully adjust to the new cost function. For every precomputed