Video Textures

Arno Schödl[1,2] *Richard Szeliski*[2] *David H. Salesin*[2,3] *Irfan Essa*[1]

[1]Georgia Institute of Technology [2]Microsoft Research [3]University of Washington

Abstract

This paper introduces a new type of medium, called a *video texture*, which has qualities somewhere between those of a photograph and a video. A video texture provides a continuous infinitely varying stream of images. While the individual frames of a video texture may be repeated from time to time, the video sequence as a whole is never repeated exactly. Video textures can be used in place of digital photos to infuse a static image with dynamic qualities and explicit action. We present techniques for analyzing a video clip to extract its structure, and for synthesizing a new, similar looking video of arbitrary length. We combine video textures with view morphing techniques to obtain *3D video textures*. We also introduce *video-based animation*, in which the synthesis of video textures can be guided by a user through high-level interactive controls. Applications of video textures and their extensions include the display of dynamic scenes on web pages, the creation of dynamic backdrops for special effects and games, and the interactive control of video-based animation.

CR Categories and Subject Descriptors:
H.5.1 [Information Interfaces]: Multimedia Information Systems—video
I.3.3 [Computer Graphics]: Picture/Image Generation—display algorithms
I.4.9 [Image Processing and Computer Vision]: Applications

Keywords: Animation, image-based rendering, morphing, multimedia, natural phenomena, texture synthesis, video-based rendering, video-based animation, video sprites, view morphing.

1 Introduction

A picture is worth a thousand words. And yet, there are many phenomena, both natural and man-made, that are not adequately captured by a single static photo. A waterfall, a flickering flame, a flag flapping in the breeze—each of these phenomena has an inherently dynamic quality that a single image simply cannot portray.

The obvious alternative to static photography is video. But video has its own drawbacks. If we want to store video on a computer or some other storage device, we are forced to use a video clip of finite duration. Hence, the video has a beginning, a middle, and an end. The video becomes a very specific embodiment of a very specific period of time. Although it captures the time-varying behavior of the phenomenon at hand, the video lacks the "timeless" quality of the photograph.

In this work, we propose a new type of medium, which is in many ways intermediate between a photograph and a video. This new medium, which we call a *video texture*, provides a continuous, infinitely varying stream of video images. (We use the term "video texture" because of the strong analogy to image textures, which usually repeat visual patterns in similar, quasi-periodic ways.) The video texture is synthesized from a finite set of images by randomly rearranging (and possibly blending) original frames from a source video.

Video textures occupy an interesting niche between the static and the dynamic realm. Whenever a photo is displayed on a computer screen, a video texture might be used instead to infuse the image with dynamic qualities. For example, a web page advertising a scenic destination could use a video texture of a beach with palm trees blowing in the wind rather than a static photograph. Or an actor could provide a dynamic "head shot" with continuous movement on his home page. Video textures could also find application as dynamic backdrops or foreground elements for scenes composited from live and synthetic elements, for example, in computer games.

The basic concept of a video texture can be extended in several different ways to further increase its applicability. For backward compatibility with existing video players and web browsers, finite duration *video loops* can be created to play continuously without any visible discontinuities. The original video can be split into independently moving *regions*, and each region can be analyzed and rendered independently. We can also use computer vision techniques to separate objects from the background and represent them as *video sprites*, which can be rendered at arbitrary image locations. Multiple video sprites or video texture regions can be combined into a complex scene.

Video textures can also be combined with stereo matching and view morphing techniques to produce *three-dimensional video textures* that can be rendered from continually varying viewpoints. Most interesting, perhaps, is the ability to put video textures under interactive control—to drive them at a high level in real time. For instance, by interactively specifying a preferred segment within a source video, a jogger can be made to speed up and slow down according to the position of an interactive slider. Alternatively, an existing video clip can be shortened or lengthened by removing or adding video texture in the middle. We call these forms of high-level control *video-based animation*.

Creating video textures and applying them in all of these ways requires solving a number of problems. The first difficulty is in locating potential transition points in the video sequences, i.e., places where the video can be looped back on itself in a minimally obtrusive way. A second challenge is in finding a sequence of transitions that respects the global structure of the video. Even though a given transition may, itself, have minimal artifacts, it could lead to a portion of the video from which there is no graceful exit, and therefore be a poor transition to take. A third challenge is in smoothing visual discontinuities at the transitions—we solve this problem using morphing techniques. A fourth problem is in automatically factoring video frames into different regions that can be analyzed and synthesized independently. Furthermore, the various extensions described above involve new, additional challenges: creating good, fixed-length cycles; separating video texture elements from their backgrounds so that they can be used as video sprites; applying view morphing to

SIGGRAPH 2000, New Orleans, LA USA

video imagery; and generalizing the transition metrics to incorporate real-time user input.

In some cases, our solutions to these problems are (in retrospect) quite simple. We thus feel that the primary contribution of this paper may lie not so much in the technical solutions to each of these individual problems *per se*, but rather in the overall paradigm of reusing video frames to create video textures, video sprites, and video-based animation.

1.1 Related work

Increasingly, computer graphics is turning toward image-based modeling and rendering techniques [7, 13], where images captured from a scene or object are used as an integral part of the rendering process, sometime obviating the need for geometry altogether [5]. As Debevec points out [13], this trend parallels the one that occurred in music synthesis a decade ago, when sample-based synthesis replaced more algorithmic approaches like frequency modulation. To date, image-based rendering techniques have mostly been applied to still scenes such as architecture [8, 18], although they have also been used to cache and accelerate the renderings produced by conventional graphics hardware [27, 32].

Our work generalizes image-based rendering to the temporal domain. It can thus be thought of as a kind of "video-based rendering." A similar idea has been used in video games, in which hand-generated video loops have been created to simulate natural phenomena like fire or water. However, there has been little previous work on automatically generating motion by reusing captured video. Probably the work most closely related to our own is "Video Rewrite" [3], in which video sequences of a person's mouth are extracted from a training sequence of the person speaking and then reordered in order to match the phoneme sequence of a new audio track. Related 3D view interpolation techniques have also been applied to multiple video streams in the *Virtualized Reality* [16] and *Immersive Video* [19] projects. Pollard *et al.* [23] introduced the term "video sprite" for applying such techniques to an alpha-matted region of the video rather than to the whole image. Finkelstein *et al.* [10] also used alpha-matted video elements in their earlier multiresolution video work, which they called "video clip-art."

Video textures can also be thought of as a temporal extension of 2D image texture synthesis. The multiscale-sampling techniques used for texture synthesis [6, 9, 12] have in fact been directly extended by Bar-Joseph into the space-time domain [1]. Bar-Joseph's work focuses on texture-type motions with limited semantic content such as fire or water close-ups, which are well modeled by the hierarchy of filter responses used for texture synthesis. Our approach can deal with these kinds of phenomena, but also deals with much more structured motions such as repetitive human actions. In this context, some of our analysis resembles earlier work in finding cycles in repetitive motions and "rectifying" these cycles into truly periodic sequences [20, 22, 25]. Our work also has some similarity to the "motion without movement" technique [11], in which patterns in an image appear to move continuously without changing their positions.

1.2 System overview

Given a small amount of "training video" (our input video clip), how do we generate an infinite amount of similar looking video? The general approach we follow in this paper is to find places in the original video where a transition can be made to some other place in the video clip without introducing noticeable discontinuities.

Our system is thus organized into three major components (Figure 1).

The first component of the system *analyzes* the input video to find the good transition points, and stores these in a small data table that becomes part of the video texture representation. This analysis component may also optionally trim away parts of the input video that are not needed, or segment the original video into independently moving pieces, in order to more easily analyze (and find the repetition in) these individual regions.

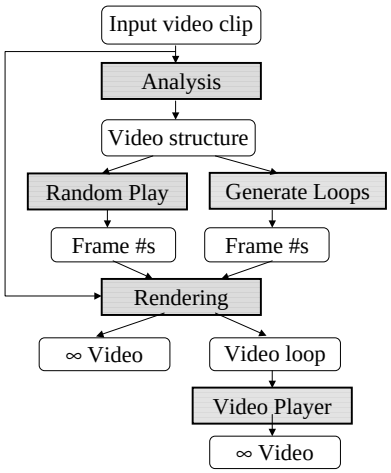


Figure 1 *System overview diagram.* An input video clip is fed into the *Analysis* component, which finds good transition points where the video can be looped back on itself. These transitions (the *Video structure*) are fed to one of two *Synthesis* components: either *Random Play*, which sequences the transitions stochastically; or *Generate Loops*, which finds a set of transitions that together create a single overall video loop of a given length. The *Rendering* component takes the generated sequence of frames, together with the original video clip, and produces either an infinite video texture sequence, or a video loop that can be played indefinitely by a standard *Video Player* in "loop" mode.

The second component of our system *synthesizes* new video from the analyzed video clip, by deciding in what order to play (or shuffle) the original video frames (or pieces thereof). We have developed two different approaches to perform this sequencing. The first approach is *random play*, which uses a Monte-Carlo (stochastic) technique to decide which frame should be played after a given frame, using the table of frame-to-frame similarities computed by the analysis algorithm. The second approach selects a small number of transitions to take in such a way that the video is guaranteed to loop after a specified number of frames. The resulting *video loop* can then be played by a conventional video player in "loop" mode.

Once the set of frames to be played has been selected, the *rendering* component puts together the frames (or frame pieces) in a way that is visually pleasing. This process may be as simple as just displaying or outputting the original video frames, or it may involve cross-fading or morphing across transitions and/or blending together independently moving regions.

The remainder of this paper describes, in more detail, the representation used to capture the structure of video textures (Section 2), our process for extracting this representation from source video (Section 3), and for synthesizing the video texture (Section 4). The rendering algorithms used to composite video sprites together and to smooth over visual discontinuities are described next (Section 5). The discussion of our basic results (Section 6) is followed by a description of some further extensions (Section 7). These include the extraction and rendering of video sprites, changing viewpoints using image-based rendering techniques, and the creation of video-based animation. The video clips associated with these results can be viewed on the CD-ROM, DVD, and Video Conference Proceedings. We conclude with a discussion of the potential of video textures and some ideas for future research.

2 Representation

Our video textures are essentially Markov processes, with each state corresponding to a single video frame, and the probabilities corresponding to the likelihood of transitions from one frame to another.

In practice, we have found two alternate (and equivalent) representations to be useful for storing these video textures. One is as a matrix of probabilities (Figure 3), in which each element P_{ij} of the matrix describes the probability of transitioning from frame i to frame j. The other is as a set of explicit *links* (Figure 6) from one frame i to another j, along with an associated probability. The first representation is advantageous when the matrix is dense, as the indices do not need to be stored explicitly. However, in most cases the set of allowable transitions is relatively sparse, and so the second representation is preferred.

In many cases, better results can be achieved by splitting the original video into regions and computing a video texture for each region separately. As discussed in more detail in Section 7.3, we sometimes also segment the video into different video sprite elements, each with its own affiliated alpha channel and compute a video texture for each sprite separately. In this case, additional information is stored along with the links to describe how the relative position of the sprite is changed as the link is crossed.

3 Analysis: Extracting the video texture

The first step in creating a video texture from an input video sequence is to compute some measure of similarity between all pairs of frames in the input sequence. In our current implementation, we use L_2 distance, since it is simple and works well in practice.

Before computing these distances, we often equalize the brightness in the image sequence (based on some background portions that do not change) in order to remove visual discontinuities that would otherwise appear when jumping between different parts of the input video. If the camera also has a small amount of jitter (e.g., from being handheld or shot in high wind conditions), we run video stabilization software over the sequence.

Once the frame-to-frame distances have been computed, we store them in the matrix

$$D_{ij} = \|\mathcal{I}_i - \mathcal{I}_j\|_2, \tag{1}$$

which denotes the L_2 distance between each pair of images $\mathcal{I}_i$ and $\mathcal{I}_j$. During the new video synthesis, the basic idea will be to create transitions from frame i to frame j anytime the successor of i is similar to j—that is, whenever $D_{i+1,j}$ is small.

A simple way to do this is to map these distances to probabilities through an exponential function,

$$P_{ij} \propto \exp(-D_{i+1,j}/\sigma). \tag{2}$$

All the probabilities for a given row of P are normalized so that $\sum_j P_{ij} = 1$. At run time, the next frame to display after frame i is selected according to the distribution of P_{ij}. The σ parameter controls the mapping between L_2 distance and relative probability of taking a given transition. Smaller values of σ emphasize just the very best transitions, while larger values of σ allow for greater variety at the cost of poorer transitions. We typically (but not always) set σ to a small multiple of the average (non-zero) D_{ij} values, so that the likelihood of making a transition at a given frame is fairly low.

3.1 Preserving dynamics

Of course, video textures need to preserve more than just similarity across frames: the dynamics of motion need to be preserved as well. Consider, for example, a swinging pendulum (Figure 2). Each frame

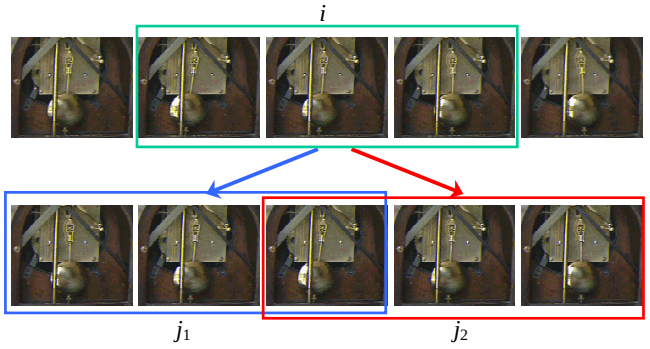


Figure 2 *Finding good transitions in a pendulum sequence.* Frame i in the top row matches both frames j_1 and j_2 of the bottom row very closely. However, of these two possibilities, only frame j_2 comes from a sequence with the correct dynamics. The two possibilities are disambiguated by considering the sequence of frames surrounding i, j_1, and j_2. Frames $i-1$, i, and $i+1$ match j_2-1, j_2, and j_2+1 but not j_1-1, j_1, and j_1+1.

of the left-to-right swing will have a corresponding frame in the right-to-left swing that looks very similar (indicated by the blue arrow in Figure 2). However, transitioning from frame i in the left-to-right swing to a frame that looks very similar to $i+1$ in the right-to-left swing will create an abrupt and unacceptable change in the pendulum's motion.

One possible way to overcome this problem might be to match velocities (e.g., using optical flow computed at each frame), in addition to matching the visual similarity between frames. However, flow computations can be quite brittle (they can be almost arbitrary in the absence of texture), so we have opted for the following simpler alternative.

We solve the problem of preserving dynamics by requiring that for a frame to be classified as similar to some other frame, not only the frames themselves, but also temporally adjacent frames within some weighted window must be similar to each other. In other words, we match subsequences instead of individual frames. Such a subsequence match can be achieved by filtering the difference matrix with a diagonal kernel with weights $[w_{-m}, \ldots, w_{m-1}]$,

$$D'_{ij} = \sum_{k=-m}^{m-1} w_k D_{i+k,j+k}. \tag{3}$$

In practice, we use $m = 1$ or 2, corresponding to a 2- or 4-tap filter, with binomial weights. (Making the filter even-length allows the decision to transition from some frame i to some other frame j to be determined as much by the similarity of i and $j-1$ as by the similarity of $i+1$ and j, removing any asymmetry in this decision.) After filtering and computing the probabilities from the filtered difference matrix, the undesired transitions no longer have high probability.

Figure 3 shows this behavior using two-dimensional images of the D_{ij} and P_{ij} tables for the pendulum sequence of Figure 2. (These images bear some resemblance to those found in earlier papers on the analysis of periodic motion [20, 22, 25].) Here, the new probabilities P'_{ij} are computed from the dynamics-preserving distances D'_{ij} in the same way as P_{ij} were computed from D_{ij} (in equation (2)). In the original unfiltered tables, the periodic nature of the pendulum is readily visible, as is the tendency to match both forward and backward swings. After filtering, only swings in the same direction are matched. (The bright knots are where the pendulum pauses at the ends of its swing, and hence has more self-similarity.) The accompanying video clips show how false jumps are eliminated.

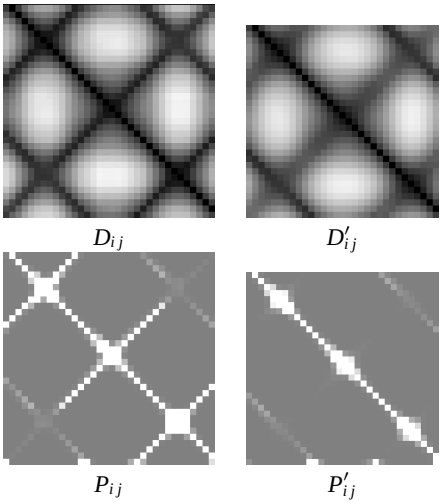


Figure 3 *Unfiltered and filtered distance matrix and transition probabilities for the clock pendulum sequence.* While the filtering has only a moderate effect on the distance table, the effect becomes pronounced when passed through the exponential function. The filtered images are slightly smaller because the filter kernel has to fit completely into the matrix and thus frames near the beginning and the end are thrown away.

Figure 4 *First and last frame of clock sequence with dead-end.* A hand moves into the field of view at the end. If only instantaneous transition costs are used, the video texture will get stuck in the last frame.

3.2 Avoiding dead ends and anticipating the future

The decision rule we have described so far looks only at the local cost of taking a given transition. It tries to match the appearance and dynamics in the two frames, but gives no consideration to whether the transition might, for example, lead to some portion of the video from which there is no graceful exit—a "dead end," in effect (Figure 4).

Much better results can be achieved by planning ahead—by trying to predict the anticipated (increased) "future cost" of choosing a given transition, given the future transitions that such a move might necessitate.

More precisely, let D''_{ij} be the *anticipated future cost* of a transition from frame $i-1$ to frame j, i.e., a cost that reflects the expected average cost of future transitions. We define D''_{ij} by summing over all future anticipated costs,

$$D''_{ij} \;=\; (D'_{ij})^p \;+\; \alpha \sum_k P''_{jk}\, D''_{jk}. \tag{4}$$

Here, p is a constant used to control the tradeoff between taking multiple good (low-cost) transitions versus a single, poorer one. (Higher p favors multiple good transitions; lower p favors a single poorer one.) The constant α is used to control the relative weight of future transitions in the metric. For convergence, we must choose $0 < \alpha < 1$ (in practice, we use $0.99 \le \alpha \le 0.999$). The probabilities P''_{jk} are defined as before, but using D'' instead of D',

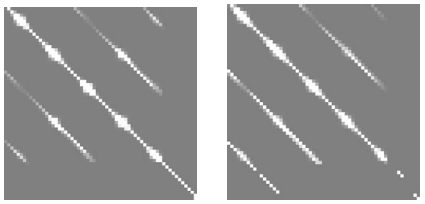


Figure 5 *Probability matrices for clock sequence with dead end.* The original probability matrix causes the player to run to the end and get stuck. The new matrix based on future costs causes the system to "jump out" early, before getting stuck in the dead end.

$$P''_{ij} \;\propto\; \exp\left(-D''_{i+1,j}/\sigma\right). \tag{5}$$

The pair of equations (4) and (5) can be solved using a simple iterative algorithm, i.e., by alternating the evaluation of (4) and (5). Unfortunately, this algorithm is slow to converge.

A faster variant on equation (4) can be derived by making the following observation. As $\sigma \to 0$, the P''_{jk} in equation (5) will tend to 1 for the best transition, and 0 otherwise. We can therefore replace equation (4) with

$$D''_{ij} \;=\; (D'_{ij})^p \;+\; \alpha \min_k D''_{jk}. \tag{6}$$

This equation is known in the reinforcement learning community as Q-learning [15]. It corresponds to finding the best possible continuation (path) through a graph with associated costs on edges, and has been proven to always converge.

We can further increase the computational efficiency of the algorithm by being selective about which rows in D''_{ij} are updated at each step. Heuristically speaking, the lowest cost path often involves a transition from a frame near the end of the sequence, and the cost of this transition has to be propagated forward. We initialize with $D''_{ij} = (D'_{ij})^p$ and define

$$m_j \;=\; \min_k D''_{jk}. \tag{7}$$

Iterating from the last row to the first, we alternately compute

$$D''_{ij} \;=\; (D'_{ij})^p \;+\; \alpha\, m_j \tag{8}$$

and update the corresponding m_j entries using equation (7). We repeat these sweeps from back to front until the matrix entries stabilize.

Figure 5 shows the probability tables before and after applying the future cost computation. The original probability matrix causes the player to run to the end and get stuck. The new matrix based on future costs causes the system to "jump out" early, before getting stuck in the dead end.

3.3 Pruning the transitions

While the above techniques can be used to produce perfectly good video textures, it is often desirable to prune the set of allowable transitions, both to save on storage space, and to improve the quality of the resulting video (suppressing non-optimal transitions).

We have examined two pruning paradigms:

1. Select only local maxima in the transition matrix for a given source and/or destination frame.

2. Set all probabilities below some threshold to zero.

The first strategy finds just the "sweet spots" in the matrix of possible transitions between frames, since often a whole neighborhood of frames has good and very similar transitions to some other neighborhood of frames, and only the best such transition needs to be kept. This can be combined with the second strategy, which is applied after the first. Both strategies are generally applied after the future cost computation has been done.

In the case of video loops, which are described in Section 4.1, we use a slightly different pruning strategy. For video loops, we would like to find sequences of frames that can be played continuously with low *average cost*, defined as the sum of all the transition costs D'_{ij}, divided by the total length of the sequence. It is straightforward to show that the average cost of a sequence of transitions is just the weighted average of the average costs of the transitions. Thus, for video loops, after pruning all transitions that are not local minima in the distance matrix, we compute the average cost for each transition, and keep only the best few (typically around 20).

4 Synthesis: Sequencing the video texture

Once the analysis stage has identified good transitions for the video texture, we need to decide in what order to play the video frames. For this *synthesis stage*, we have developed two different algorithms: *random play* and *video loops*.

Random play is very simple to describe. The video texture is begun at any point before the last non-zero-probability transition. After displaying frame i, the next frame j is selected according to P_{ij}. Note that usually, $P_{i,i+1}$ is the largest probability, since $D'_{ii} = 0$ (however, this is not necessarily true when using the anticipated future cost D''_{ij}, which is how the system avoids dead ends). This simple Monte-Carlo approach creates video textures that never repeat exactly and is useful in situations in which the video texture can be created on the fly from the source material.

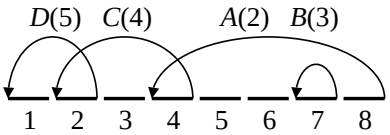
When a conventional digital video player is used to show video textures, it is necessary to create video loops that do in fact repeat with a fixed period. In this case the video texture can be played in standard "loop mode" by such a player. Generating such loops with the highest possible quality is actually a rather difficult problem, to which we devote the rest of this section.

4.1 Video loops

Consider a loop with a single transition $i \rightarrow j$, from *source frame* i to *destination frame* j, which we call a *primitive loop*. In order for the single transition to create a (non-trivial) cycle we must have $i \geq j$. Thus, the *range* of this loop is $[j, i]$. The *cost* of this loop is the filtered distance between the two frames D'_{ij}.

One or more primitive loops can be combined to create additional cyclic sequences, called *compound loops*. To add one (primitive or compound) loop to another, their ranges must overlap. Otherwise, there is no way to play the first loop after the second has played. The resulting compound loop has a range that is the union of the ranges of the two original loops, and a length and cost that is the sum of the original lengths and costs. Compound loops may contain several repeated instances of the same primitive loop, and can thus be represented by a multiset, where the ordering of the loops is not important.

Forward transitions $i \rightarrow j$, where $i + 1 < j$, can be added into a cycle as well. Although we have an algorithm that efficiently checks whether a multiset of forward and backward jumps is playable, the dynamic programming algorithm described below, which finds the lowest cost compound loop of a given length, does not work with forward jumps, and we currently have no suitable extension. Our algorithm for creating compound loops of minimal average cost therefore considers only *backward transitions* (transitions $i \rightarrow j$ with $i \geq j$).



length	$A(2)$	$B(3)$	$C(4)$	$D(5)$
1		$B(3)$		
2		$B^2(6)$		$D(5)$
3		$B^3(9)$	$C(4)$	
4		$B^4(12)$		$D^2(10)$
5	$A(2)$	$B^5(15)$	$CD(9)$	$CD(9)$
6	$AB(5)$	$AB(5)$	$C^2(8)$	$D^3(15)$
		⋮		

Figure 6 *Dynamic programming table for finding optimal loops.* Each entry lists the best compound loop of a given length that includes the primitive loop listed at the top of the column. Total costs are shown in parentheses.

In the remainder of this section we present the two algorithms we need to generate *optimal loops*—that is, video loops with minimal cost for a given sequence length. The first algorithm selects a set of transitions that will be used to construct the video loop. The second algorithm orders these transitions in a legal fashion—that is, in an order that can be played without any additional transitions.

4.2 Selecting the set of transitions

The most straightforward way to find the best compound loop of a given length L is to enumerate all multisets of transitions of total length L, to select the legal ones (the compound loops whose ranges form a continuous set), and to keep the best one. Unfortunately, this process is exponential in the number of transitions considered.

Instead, we use a dynamic programming algorithm. Our algorithm constructs a table of L rows, where L is the maximum loop length being considered, and N columns, where N is the number of transitions, or primitive loops, being considered (see Figure 6). The algorithm builds a list of the best compound loop of a given length that contains at least one instance of the primitive loop listed at the top of the column. Each cell in the table lists the transitions in the compound loop and the compound loop's total cost.

The algorithm works by walking through the table, updating cells one row at a time. For each cell, it examines all compound loops of shorter length in that same column, and tries to combine them with compound loops from columns whose primitive loops have ranges that overlap that of the column being considered. (This assures that the created compound loops are actually playable, since the ranges of the constituent compound loops must overlap.) For example, the entry in row 5 column C is obtained by combining the entry in row 3 column C with the entry in row 2 column D, which is possible since primitive loops C and D have ranges that overlap and have lengths that sum to 5. The combination with the lowest total cost becomes the new entry.

For each of the LN cells examined, the algorithm must combine at most $L - 1$ compound loops from its column with at most $N - 1$ entries from the other columns. The total computational complexity of the algorithm is therefore $O(L^2N^2)$, with a space complexity of

$O(LN)$. Note that the full descriptions of the compound loops need not be stored during the computation phase: only backpointers to the originating cells (constituent compound loops) are needed.

4.3 Scheduling the primitive loops

After finding the list of primitive loops in the lowest cost compound loop, the transitions have to be scheduled in some order so that they form a valid compound loop. This is done as follows (we use the scheduling of $\{ABCD\}$ in this example):

1. Schedule the transition that starts at the very end of the sequence as the very first transition to be taken. This would be A in our example.
2. The removal of this transition $i \rightarrow j$ may break the remaining primitive loops into one or more sets of continuous ranges. In our example, the removal of A breaks the remaining loops into two continuous-range sets $\{C, D\}$ and $\{B\}$. Frame j is always contained in the first such set and we schedule next any transition from this set whose source frame occurs after j. In our example, C is the only transition that meets these criteria.
3. Repeat the previous step, removing transitions $i \rightarrow j$ until there are no more primitive loops left in the first range. In our example, D would be removed next by this repeated step.
4. Schedule any primitive loop in each of the following disjoint ranges, using the algorithm beginning at step 2. In our example, B is the only primitive loop left.
5. Continue with step 2 until all primitive loops are removed.

In our example, the loops are scheduled in the order A, C, D, B.

The computational complexity of this algorithm is quadratic in the number of transitions in the compound loop. The scheduling algorithm can either be run in a deterministic fashion (e.g., taking the first legal transition encountered), or in a stochastic fashion (randomly selecting from the legally available transitions). The latter variant, which utilizes transitions with precisely the same frequency as in the compound loop, is an alternative to the Monte-Carlo sequencing algorithm presented earlier.

5 Rendering

Although we favor transitions that introduce only small discontinuities in the motion, there are cases where no unnoticeable transitions are available in the sequence. This section describes techniques for disguising discontinuities in the video texture, and for blending independently analyzed regions together.

Instead of simply jumping from one frame to another when a transition is made, the images of the sequence before and after the transition can be blended together with standard *cross-fading*: frames from the sequence near the source of the transition are linearly faded out as the frames from the sequence near the destination are faded in. The fade is positioned so that it is halfway complete where the transition was scheduled.

Although cross-fading of the transitions avoids abrupt image changes, it temporarily blurs the image if there is a misalignment between the frames. The transition from sharp to blurry and back again is sometimes noticeable. In some situations, this problem can be addressed by taking very frequent transitions so that several frames are always being cross-faded together, maintaining a more or less constant level of blur.

Our implementation of the cross-fading algorithm supports multi-way cross-fades, i.e., more than two subsequences can be blended together at a time. The algorithm computes a weighted average of all frames participating in a multi-way fade,

$$B(x,y) = \sum_i \alpha_i \mathcal{I}_i(x,y), \tag{9}$$

where the blending weights α_i are derived from the shifted weighting kernels associated with each participating frame, normalized such that $\sum_i \alpha_i = 1$.

Another approach to reducing the blurriness of the transitions is to *morph* the two sequences together, so that common features in the two sets of frames are aligned. The method we use is based on the *de-ghosting* algorithm described by Shum and Szeliski [29] and is also related to automatic morphing techniques [2].

To perform the de-ghosting, we first compute the optical flow between each frame $\mathcal{I}_i$ participating in the multi-way morph and a reference frame $\mathcal{I}_R$ (the reference frame is the one that would have been displayed in the absence of morphing or cross-fading). For every pixel in $\mathcal{I}_R$, we find a *consensus* position for that pixel by taking a weighted average of its corresponding positions in all of the frames $\mathcal{I}_i$ (including $\mathcal{I}_R$). Finally, we use a regular inverse warping algorithm to resample the images such that all pixels end up at their consensus positions. We then blend these images together.

When the video texture consists of several independently analyzed regions, the rendering algorithm blends these regions together smoothly. We use the *feathering* approach commonly used for image mosaics [31], where the contribution of a given region (our analysis regions are typically overlapping) tapers gradually towards its edge.

6 Basic results

The accompanying video clips (available on the CD-ROM, DVD, and Video Conference Proceedings) demonstrate several different video textures produced by the methods described so far. Here, we summarize these basic results; in the next section, we develop some extensions to the basic algorithms and show some additional results.

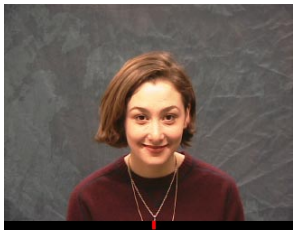
Candle flame. A 33-second video of a candle flame was turned into four different video textures: one random play texture; and three different video loops, each containing three different primitive loops. One of the video loops repeats every 426 frames. The other two repeat every 241 frames; these each use the same set of three primitive loops, but are scheduled in a different order. In the figure at right, the position of the frame currently being displayed in the original video clip is denoted by the red bar. The red curves show the possible transitions from one frame in the original video clip to another, used by the random play texture.

Clock. This example shows the necessity for both the preservation of dynamics and the future cost computation. The input video sequence shows a clock with a swinging pendulum. Without considering dynamics, a forward-swinging pendulum is likely to match equally well with a backward-swinging frame, causing unnatural jumps in the motion. Adding in the temporal filtering solves this problem. At the end of the input video, a hand moves into the frame. Without the future cost computation, the video texture will reach a dead end, from which no transition to the earlier video will work without a visual jump. The future cost computation solves this problem by increasing the probability of a transition before the hand comes into frame.

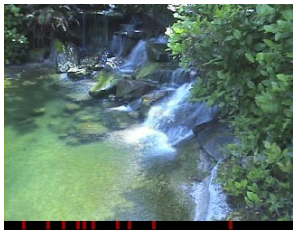
Flag. A 38-second video of a flying flag was cyclified using the lowest average cost loop contained in the video. Video textures were created using no fading, cross-fading, and morphing. Cross-fading improves the quality of the transition, at the cost of a small amount of blurring. Morphing works even better at removing the jump without introducing blur, even though the alignment is one stripe off the geometrically correct alignment. The wrong alignment that causes a fold to magically disappear during transition is almost invisible to the unsuspecting observer.

Campfire. A 10-second video of a campfire was cyclified using a single transition. The transition is hardly visible without crossfading, but crossfading over four frames hides it entirely. Although the configuration of the flames never replicates even approximately, the transition is well hidden by the high temporal flicker.

Portrait. A 25-second video of a woman posing for a portrait was turned into a random-play video texture with 20 transitions. Although the frames across the transitions are already quite similar, the morphing performs a subtle alignment of details, such as the eye positions, which hides the transitions almost entirely. Such video textures could be useful in replacing the static portraits that often appear on web pages.



Waterfall. This example of a waterfall works less well. The original 5 minute video sequence never repeats itself, and yet, unlike the campfire, there is a great deal of continuity between the frames, making it difficult to find any unnoticeable transitions. Our best result was obtained by selecting a 6-second source clip, and using cross-fading with frequent transitions so that averaging is always performed across multiple subsequences at once. Although the resulting video texture is blurrier than the original video clip, the resulting imagery is still fairly convincing as a waterfall.



Blowing grass. Here is another example that does not work well as a video texture. Like the waterfall sequence, the original 43-second video of blowing grass never repeats itself. Unlike the waterfall sequence, blurring several frames together does not produce acceptable results. Our automatic morphing also fails to find accurate correspondences in the video frames. The best we could do was to cross-fade the transitions (using a 4-second clip as the source), which creates occasional (and objectionable) blurring as the video texture is played.

7 Extensions

In this section, we present several extensions to the basic idea of video textures: *sound synthesis*, in which an audio track is re-rendered along with the video texture; *three-dimensional video textures*, in which view interpolation techniques are applied to simulate 3D motion; *motion factorization*, in which the video frames are factored into separate parts that are analyzed and synthesized independently; and *video-based animation*, in which video texture is modified under interactive control.

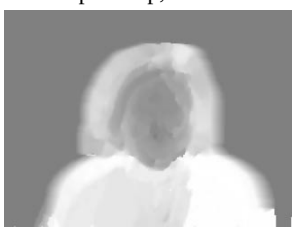
7.1 Sound synthesis

Adding sound to video textures is relatively straightforward. We simply take the sound samples associated with each frame and play them back with the video frames selected to be rendered. To mask any popping effects, we use the same multi-way cross-fading algorithm described in Section 5. The resulting sound tracks, at least in the videos for which we have tried this (*Waterfall* and *Bonfire*), sound very natural.

7.2 Three-dimensional video textures

Video textures can be combined with traditional image-based rendering algorithms such as view interpolation [5, 18, 24] to obtain *three-dimensional video textures*. These are similar to the 3D video-based characters demonstrated in several video-based view interpolation systems [16, 19, 23], except that they are based on synthetic video textures instead of captured video clips.

3D Portrait. We created a three-dimensional video texture from three videos of a smiling woman, taken simultaneously from three different viewing angles about 20 degrees apart. We used the center camera to extract and synthesize the video texture, and the first still from each camera to estimate a 3D depth map, shown here. (As an alternative, we could have used some other 3D image-based modeling technique [21].) We then masked out the background using background subtraction (a clear shot of the background was taken before filming began). To generate each new frame in the 3D video animation, we mapped a portion of the video texture onto the 3D surface, rendered it from a novel viewpoint, and then combined it with the flat image of the background warped to the correct location, using the algorithm described in [26].



7.3 Motion factorization

For certain kinds of more complex scenes, we can divide the original video into independently moving parts, and analyze each one separately. This kind of *motion factorization* decreases the number of frame samples necessary to synthesize an interesting video texture. Interdependencies between different parts of the synthesized frames could later be added with supplemental constraints.

The simplest form of motion factorization is to divide the frame into independent regions of motion, either manually or automatically.

Swings. In this example, the video of two children on swings is manually divided into two halves: one for each swing. These parts are analyzed and synthesized independently, then recombined into the final video texture. The overall video texture is significantly superior to the best video texture that could be generated using the entire video frame.

Balloons. For this example, we developed an automatic segmentation algorithm that separates the original video stream into regions that move independently. We first compute the variance of each pixel across time, threshold this image to obtain connected regions of motion, and use connected component labeling followed by a morphological dilation to obtain the five region labels (shown as color regions in this still). The independent regions are then analyzed and synthesized separately, and then recombined using feathering.

Motion factorization can be further extended to extract independent *video sprites* from a video sequence. For instance, we can use background subtraction or blue-screen matting [30] to identify connected portions of the video image that change with time and to extract these portions as foreground elements with alpha. To create a video sprite, we first factor out the positional information by placing the element's centroid at the origin in each frame. We call this *registering* the element. We also store the *velocity* of the sprite at each frame, defined as the difference in the unregistered elements' centroid positions. In the analysis phase the distance between frames is computed as a linear combination of the registered elements' colors, alphas, and moving directions and speeds. The synthesis phase is performed by utilizing the optimal transitions computed by the analysis and adding back in the stored velocities across the transitions.

Fish. We used background subtraction to create a video sprite of a fish, starting from 5 minutes of video of a fish in a tank. Unfortunately, fish are uncooperative test subjects who frequently visit the walls of the fish tank, where they are hard to extract from the scene because of reflections in the glass. We therefore used as source material only those pieces of video where the fish is swimming freely. (This requires generalizing the future cost computation to handle the possibility of multiple dead ends, but is otherwise straightforward.)

Using this technique, the fish swims freely in two-dimensional space. Ideally, we would like to constrain its motion—for example, to the boundaries of a fish tank. The next section describes approaches to this problem.

7.4 Video-based animation

Instead of using visual smoothness as the only criterion for generating video, we can also add some user-controlled terms to the error function in order to influence the selection of frames.

The simplest form of such user control is to interactively select the set of frames S in the sequence that are used for synthesis. In this case, we perform the analysis phase as before, optionally pruning the list of transitions. In the synthesis stage, however, we recompute the probabilities of the transitions, using a modified form of equation (5), which takes into account the distance from the destination of the transition to the set of user-specified frames S:

$$P_{ij} \propto \exp\left(-(D''_{i+1,j}/\sigma + w\,\mathrm{distance}(j,S))\right) \qquad (10)$$

Here, w trades off the weight of the user-control against the smoothness of the transitions.

Runner. We took 3 minutes of video of a runner on a treadmill, starting at a slow jog and then gradually speeding up to a fast run. As the user moves a slider selecting a certain region of the video (the black region of the slider in the figure), the synthesis attempts to select frames that remain within that region, while at the same time using only fairly smooth transitions to jump forward or backward in time. The user can therefore control the speed of the runner by moving the slider back and forth, and the runner makes natural-looking transitions between the different gaits.

We expect it will be possible to extend this type of *parametric motion control* to other types of movements as well, thereby allowing much greater directorial control in the post-processing phase of video production.

Watering can. As another example, we took a 15-second clip of a watering can pouring water into a birdbath. The central portion of this video, which shows the water pouring as a continuous stream, makes a very good video texture. We can therefore shorten or extend the pouring sequence by using the same technique as we did for the runner, only advancing the slider automatically at a faster or slower speed. Thus, the same mechanism can be used to achieve a natural-looking *time compression* or *dilation* in a video sequence.

Mouse-controlled fish. Instead of directly specifying a preferred range of frames, we can select frames based on other criteria. For example, in order to interactively guide the path of the fish presented earlier with a mouse, we could give preference to frames in which the fish's video sprite has a certain desired velocity vector.

In particular, if x is the current position of the fish, y the desired position of the fish (say the mouse location), and v_i the velocity at frame i, then we can use the following distance function:

$$D'_{ij} = w_1 \left\|\mathcal{I}_i - \mathcal{I}_j\right\|_2 + w_2\, E(v_i, v_j) + w_3\, E(y - x,\, v_j) \qquad (11)$$

where w_1, w_2, w_3 are user-specified weights, $\left\|\mathcal{I}_i - \mathcal{I}_j\right\|_2$ is a modified image distance metric that takes into account the difference in the two image sprites' alpha channels, and $E(v, v')$ is a "velocity error function". In our current implementation, E is proportional to the angle between v and v'.

In the runner example (Equation 10), in order to achieve interactive performance we added the extra error term to D''_{ij} directly, instead of adding the term to D'_{ij} and re-running the precomputed future cost computation. It turns out that this technique does not work so well for directed movement: the system has trouble finding good sequences on the fly that will avoid later bad transitions. To do this right, a larger-scale anticipated future cost computation is required. We therefore compute the future cost D''_{ij} from D'_{ij} using the techniques described in Section 3.2. Unfortunately, we have to do this precomputation for all possible values of E. In practice, we perform the precomputation for a set of eight different directions and discretize the user input to one of these directions on the fly, choosing the precomputed probability table accordingly.

Fish tank. The final example we show is a complete fish tank, populated with artificial fish sprites. The tank includes two sets of bubbles, two independently swaying plants, and a small number of independently moving fish. The fish can also be scripted to follow a path (here, the SIGGRAPH "2000" logo), using the same techniques described for the mouse-controlled fish.

8 Discussion and future work

In his 1966 science-fiction short story, "Light of Other Days," Bob Shaw describes a material called *slow glass*, which traps photons coming into it, and emits them a year or two later [28]. Slow glass can be exposed in scenic locations (such as a woodland lake) and then placed on people's walls, where it gives a three-dimensional illusion of having a scenic view of your own.

Video textures (which were partially inspired by this story) share some of the characteristic of slow glass, but also differ in important ways. Like slow glass, they are an attempt to capture the inherent dynamic characteristics of a scene. (Using a video camera array to capture a time-varying light field [17] would be another approach, but capturing enough data to play back for a year would be prohibitively expensive.) Video textures attempt to capture the inherent *characteristics* of a dynamic scene or event, without necessarily capturing all of the stochastically-varying detail inherent in a particular segment of time.

Video textures also have the potential to give the artist creative control over the appearance of the dynamic events they are depicting. By capturing a wide variety of similar looking video that is periodic or quasi-periodic, the user can then select which portions to use, and blend smoothly between different parameter settings. The video texture analysis and synthesis software takes care of making these transitions smooth and creating segments of the desired duration.

How well do video textures work? For motions that are smooth and repetitive or quasi-repetitive, such as the kids on the swing, the candle flame, the swaying balloons, the runner, and the smiling woman, the illusion works quite well. For complex stochastic phenomena with little discernible structure, like the water pouring out of the can, it also works well. We run into trouble when the phenomena are complex but also highly structured, like the grass blowing in the wind (we have also thus far failed at creating a convincing video texture for waves on a beach). Other highly structured phenomena like full-body human motion will also likely fail, unless we start using some higher-level motion and structure analysis.

Our work suggests a number of important areas for future work:

Better distance metrics. To create video textures, we need a distance metric that reliably quantifies the perceived discontinuity of a frame transition. For most of our examples we used a simple L_2 distance between images. Finding better features and distance functions will be crucial for improving the quality of video textures and for increasing their applicability. We have some initial promising results applying a wavelet-based distance metric [14] to some of our sequences. We have also improved the metric for the fish example by modeling it as a linear combination of several features and learning the coefficients from hand-labeled training transitions.

Better blending. To suppress residual visual discontinuities, we are currently using blending and morphing. We would like to explore techniques that allow for blending and morphing separately in different frequency bands both in space and time, perhaps using multiresolution splining techniques [4].

Maintaining variety. A significant problem with generating long (infinite) sequences of video from the same set of frames is that, after a while, the algorithm will find some optimal paths and more or less play the same series of frames over and over again. This requires that in addition to σ, which controls randomness, we define a parameter that penalizes a lack of variety in the generated sequences. Such a parameter would enforce that most (if not all) of the frames of the given input sequence are sometimes played and probabilistically vary the generated order of frames.

Better tools for creative control. Another important area of future research will be the addition of more creative control over video textures. An alternative to interactively controlling the parameters in a video animation would be to specify control points or keyframes as in conventional keyframe animation. For this, we need to develop optimization techniques that generate smoothly playing video textures that obey user-supplied constraints. Better video animation control would enable us to generate complex scenes such as crowds; the animation controller could also be enhanced to include behavioral aspects such as flocking.

While many areas remain to be explored, we believe that video textures provide an interesting new medium with many potential applications, ranging from simple video portraits to realistic video synthesis. Video textures are just one example of the more general class of techniques we call video-based rendering. By re-using real-world video footage (in a manner analogous to image-based rendering), we can achieve a degree of photorealism and naturalness hard to match with traditional computer graphics techniques. We hope that this work will spur further research in this field, and that video textures, along with video-based rendering in general, will ultimately become an essential part of the repertoire of computer graphics techniques.

References

[1] Z. Bar-Joseph. Statistical learning of multi-dimensional textures. Master's thesis, The Hebrew University of Jerusalem, June 1999.

[2] D. Beymer. Feature correspondence by interleaving shape and texture computations. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'96)*, pages 921–928, San Francisco, California, June 1996.

[3] C. Bregler, M. Covell, and M. Slaney. Video rewrite: Driving visual speech with audio. *Computer Graphics (SIGGRAPH'97)*, pages 353–360, August 1997.

[4] P. J. Burt and E. H. Adelson. A multiresolution spline with applications to image mosaics. *ACM Transactions on Graphics*, 2(4):217–236, October 1983.

[5] S. E. Chen. QuickTime VR – an image-based approach to virtual environment navigation. *Computer Graphics (SIGGRAPH'95)*, pages 29–38, August 1995.

[6] J. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. *Computer Graphics (SIGGRAPH'97)*, pages 361–368, August 1997.

[7] P. Debevec *et al.*, editors. *Image-Based Modeling, Rendering, and Lighting*, SIGGRAPH'99 Course 39, August 1999.

[8] P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Computer Graphics (SIGGRAPH'96)*, pages 11–20, August 1996.

[9] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *Seventh International Conference on Computer Vision (ICCV'99)*, pages 1033–1038, Kerkyra, Greece, September 1999.

[10] A. Finkelstein, C. E. Jacobs, and D. H. Salesin. Multiresolution video. *Proceedings of SIGGRAPH 96*, pages 281–290, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[11] W. T. Freeman, E. H. Adelson, and D. J. Heeger. Motion without movement. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):27–30, July 1991.

[12] D. J. Heeger and J. R. Bergen. Pyramid-based texture analysis/synthesis. *Proceedings of SIGGRAPH 95*, pages 229–238, August 1995.

[13] *Workshop on Image-Based Modeling and Rendering*, Stanford University, March 1998. http://graphics.stanford.edu/workshops/ibr98/.

[14] C. E. Jacobs, A. Finkelstein, and D. H. Salesin. Fast multiresolution image querying. *Proceedings of SIGGRAPH 95*, pages 277–286, August 1995.

[15] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4, 1996.

[16] T. Kanade, P. W. Rander, and P. J. Narayanan. Virtualized reality: constructing virtual worlds from real scenes. *IEEE MultiMedia Magazine*, 1(1):34–47, Jan-March 1997.

[17] M. Levoy and P. Hanrahan. Light field rendering. In *Computer Graphics Proceedings, Annual Conference Series*, pages 31–42, Proc. SIGGRAPH'96 (New Orleans), August 1996. ACM SIGGRAPH.

[18] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. *Computer Graphics (SIGGRAPH'95)*, pages 39–46, August 1995.

[19] S. Moezzi *et al.* Reality modeling and visualization from multiple video sequences. *IEEE Computer Graphics and Applications*, 16(6):58–63, November 1996.

[20] S. A. Niyogi and E. H. Adelson. Analyzing and recognizing walking figures in xyt. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'94)*, pages 469–474, Seattle, Washington, June 1994.

[21] F. Pighin, J. Hecker, D. Lischinski, D. H. Salesin, and R. Szeliski. Synthesizing realistic facial expressions from photographs. In *Computer Graphics (SIGGRAPH'98) Proceedings*, pages 75–84, Orlando, July 1998. ACM SIGGRAPH.

[22] R. Polana and R. C. Nelson. Detection and recognition of periodic, nonrigid motion. *International Journal of Computer Vision*, 23(3):261–282, 1997.

[23] S. Pollard *et al.* View synthesis by trinocular edge matching and transfer. In *British Machine Vision Conference (BMVC98)*, Southampton, England, September 1998.

[24] S. M. Seitz and C. M. Dyer. View morphing. In *Computer Graphics Proceedings, Annual Conference Series*, pages 21–30, Proc. SIGGRAPH'96 (New Orleans), August 1996. ACM SIGGRAPH.

[25] S. M. Seitz and C. R. Dyer. View invariant analysis of cyclic motion. *International Journal of Computer Vision*, 25(3):231–251, December 1997.

[26] J. Shade, S. Gortler, L.-W. He, and R. Szeliski. Layered depth images. In *Computer Graphics (SIGGRAPH'98) Proceedings*, pages 231–242, Orlando, July 1998. ACM SIGGRAPH.

[27] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical images caching for accelerated walkthroughs of complex environments. In *Computer Graphics (SIGGRAPH'96) Proceedings*, pages 75–82, Proc. SIGGRAPH'96 (New Orleans), August 1996. ACM SIGGRAPH.

[28] B. Shaw. Light of other days. In *Other Days, Other Eyes*. Ace Books, New York, 1972. (also published in *Analog* and various sci-fi anthologies).

[29] H.-Y. Shum and R. Szeliski. Construction of panoramic mosaics with global and local alignment. *International Journal of Computer Vision*, 36(2):101–130, February 2000.

[30] A. R. Smith and J. F. Blinn. Blue screen matting. In *Computer Graphics Proceedings, Annual Conference Series*, pages 259–268, Proc. SIGGRAPH'96 (New Orleans), August 1996. ACM SIGGRAPH.

[31] R. Szeliski and H.-Y. Shum. Creating full view panoramic image mosaics and texture-mapped models. In *Computer Graphics (SIGGRAPH'97) Proceedings*, pages 251–258, Los Angeles, August 1997. ACM SIGGRAPH.

[32] J. Torborg and J. T. Kajiya. Talisman: Commodity realtime 3D graphics for the PC. In *Computer Graphics Proceedings, Annual Conference Series*, pages 353–363, Proc. SIGGRAPH'96 (New Orleans), August 1996. ACM SIGGRAPH.