To Appear in the ACM Transactions on Graphics

Image-Based Spatio-Temporal Modeling and View Interpolation of Dynamic Events

Sundar Vedula[†], Simon Baker, and Takeo Kanade

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present an approach for modeling and rendering a dynamic, real-world event from an arbitrary viewpoint, and at any time, using images captured from multiple video cameras. The event is modeled as a non-rigidly varying dynamic scene, captured by many images from different viewpoints, at discrete times. First, the spatio-temporal geometric properties (shape and instantaneous motion) are computed. The view synthesis problem is then solved using a reverse mapping algorithm, ray-casting across space and time, to compute a novel image from any viewpoint in the 4D space of position and time. Results are shown on real-world events captured in the CMU 3D Room, by creating synthetic renderings of the event from novel, arbitrary positions in space and time. Multiple such re-created renderings can be put together to create re-timed fly-by movies of the event, with the resulting visual experience richer than that of a regular video clip, or switching between images from multiple cameras.

Keywords: Image-based modeling and rendering, dynamic scenes, spatio-temporal view interpolation, non-rigid motion, voxel models, space carving, scene flow.

† Currently with Media Processing Division, Sony Electronics, San Jose, California.

1 Introduction

The world around us consists of a large number of complex events occurring at any time. For example, a basketball game can be considered as an event that occurs within some area over a certain period. Similarly, a musical concert, a visit to a museum, or even a walk or daily chore are all examples of the millions of events that take place everyday. Clearly, most events around us are dynamic, meaning that things move in interesting ways, whether they are people, cars, animals, or other objects or natural phenomena that involve movement.

We are interested in creating digital models of these dynamic events. A useful capability would be if we could capture and digitize a dynamic event in its entirety. That way, we could re-render it from any viewpoint, similar to how it is possible to view synthetic graphics models from an arbitrary position. In addition, we would also be able to play back any temporal segment of this re-rendered view, even at a speed different from that of the original.

Most recent image-based modeling approaches have focused on modeling static scenes, and rendering them from different spatial viewpoints [6, 10, 15, 20, 21, 23]. There has been very little work on re-rendering a dynamic event across time, and the approaches developed have always assumed a restricted motion model. Either the scene consists of rigidly moving objects [18, 22] or point features moving along straight lines with constant velocity [33].

We propose an image-based approach to modeling dynamic events, by capturing images of the event from many different viewpoints simultaneously. By creating a true dynamic model of the event, we can then re-create its appearance from any arbitrary position in space, at an arbitrary time during the occurrence of the event (irrespective of when and where the images are sampled from). Figure 1 presents an illustrative example of this task which we call *Spatio-Temporal View Interpolation.* The figure contains four images captured by two cameras at two different time instants. The images on the left are captured by camera C_1, those on the right by camera C_2. The bottom two images are captured at the first time instant and the top two at the second. Spatio-temporal view interpolation consists of combining these four views into a novel image of the event at an arbitrary viewpoint and time. Although we have described spatio-temporal modeling in terms of two images taken at two time instants, our algorithm applies to an arbitrary number of images taken from an arbitrary collection of cameras spread over an extended period of time.

To generate novel views of a dynamic event we need to know how the pixels in the input

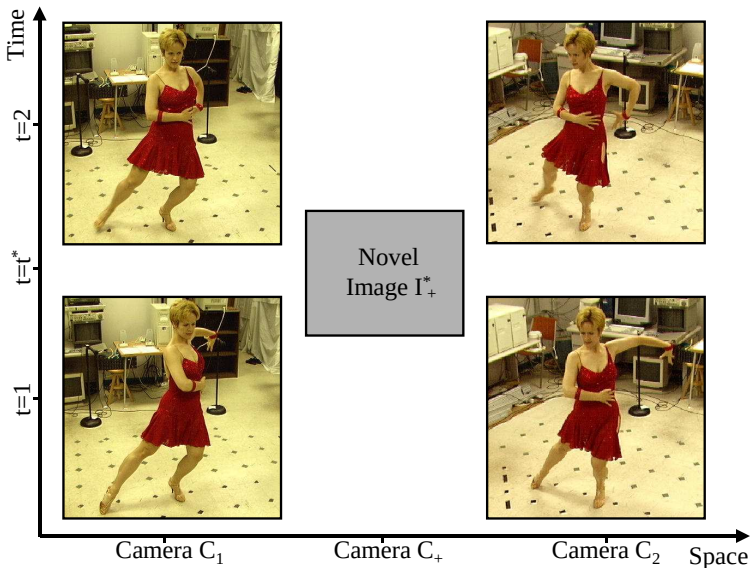


Figure 1: Spatio-temporal modeling and view interpolation consists of taking a collection of images of an event captured with multiple cameras at different times and re-rendering the event at an arbitrary viewpoint and time. In this illustrative figure, the two images on the left are captured with the same camera at two different times, and the two images on the right with a different camera at the same time instants. The novel image and time are shown as halfway between the cameras and time instants, but are arbitrary.

images are geometrically related to each other. In the various approaches to view interpolation of static scenes across space there are two common ways in which this geometric information is provided. First, there are algorithms that use *implicit* geometric information in the form of *point correspondences* [6, 23]. Second, there are approaches that use *explicit* 3D models [8, 20, 21].

Although either choice is possible, we decided to base our spatio-temporal view interpolation algorithm on explicit 3D models of the scene. The primary reason for this decision is that we would like our algorithms to be fully automatic. The correspondences that are used in image-based rendering algorithms are generally specified by hand, which becomes an enormous task given the number of frames in a dynamic sequence. Our algorithm is applicable to completely non-rigid events, uses no scene or object specific models, and requires no user input. Another reason is that the explicit representation is more compact when there are a large number of images.

1.1 Overview of the Approach

Our approach for spatio-temporal modeling is based on the explicit recovery of scene properties. For a dynamic event, we need to understand the time-varying geometry of the scene and the objects

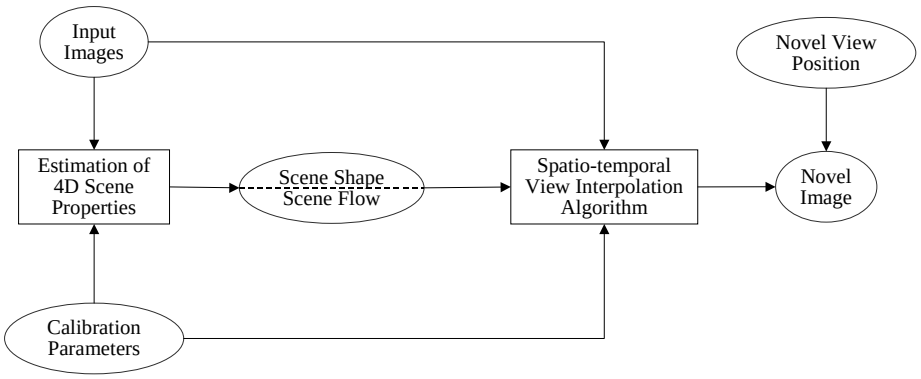


Figure 2: An overview of the various steps in spatio-temporal modeling and view interpolation.

that comprise the event. We model the scene as consisting of one or more opaque objects, each of which may be changing shape in a non-rigid way, which we refer to as the *motion* of the scene. We compute this motion by measuring *scene flow* (first proposed in [30]), which we define as a first order measure of the instantaneous non-rigid motion of all objects in the scene.

Figure 2 shows the steps in our approach to render the appearance of the scene at any time. First, the event is imaged by multiple time-synchronized video cameras, each of which gives us a sequence of images at known time instants. Also, the calibration parameters for each of these cameras (both extrinsic and intrinsic) are estimated using the algorithm of [25]. Using the input images and calibration parameters, we compute the scene shape and scene flow, to get a complete 4D model of the time-varying geometry and instantaneous motion of the scene. This is done at all time instants at which images are captured. The 4D model is different from just a sequence of shapes, since the availability of shape and instantaneous motion at the sampled time instants allows us to compute geometric information as a continuous function of time.

Then, the 4D scene properties, along with the input images and calibration parameters, become inputs to the spatio-temporal view interpolation algorithm. For any requested position and time of the novel view, the algorithm first estimates the interpolated scene shape at the desired time by flowing the computed shapes at the neighboring times using the scene flow. The points on this shape that correspond to each pixel in the novel image are determined, and then the corresponding points on the computed models at the sampled time instants are found. The known geometry of the scene at those times, with the camera calibration parameters is then used to project these corresponding points into the input images. The input images are sampled at the appropriate locations

and the estimates combined to generate the novel image at the intermediate space and time, one pixel at a time. This spatio-temporal view interpolation algorithm was first described in [29].

To obtain high quality results, there are a number of technical issues that have to be dealt with. First, we require that the 3D scene flow and the 3D voxel models it relates be consistent so that when the models are flowed no holes (or other artifacts) are generated. Second, a surface must be fit to the flowed voxel models to avoid artifacts introduced by the cubic voxels.

2 Spatio-Temporal Scene Modeling

The first step towards modeling dynamic events is to understand the geometric properties of a dynamic scene. The scene can consist of many objects, each of which can be fixed or moving. A fundamental property of the scene is the shape of all of the objects in it. In addition, since the scene is dynamic, the motion of each object is typically some combination of rigid motion (such as rotation and translation), and non-rigid motion (bending, warping). Recall that we use *scene flow* as a measure of the instantaneous motion (rigid and non-rigid combined) of all parts of the scene. So while shape is a static (or zeroth order) description of the geometry of the scene, scene flow is a measure of the first order of motion (or velocity) of the scene. Since our approach to spatio-temporal modeling is based on explicit recovery of scene properties, we first formally define scene flow and then look into how it can be computed from multiple views of the scene.

2.1 Inputs

We first describe the inputs to the algorithm. (For simplicity, we discuss details of the algorithm for the case of two neighboring time instants; the same approach generalizes to other time instants as well). We assume that the scene is imaged by N fully calibrated cameras with synchronized shutters. The inputs are the set of images I_i^t captured by cameras C_i with projection matrices P_i, where $i = 1, \ldots, N$ and $t = 1, \ldots, T$. See Figure 3 for an example set of input images for $T = 2$.

2.2 Scene Shape

At each time instant, the shape of the scene is estimated as a 3D voxel model. Volumetric shape estimation has been widely studied and there are a variety of algorithms. One example of a vol-

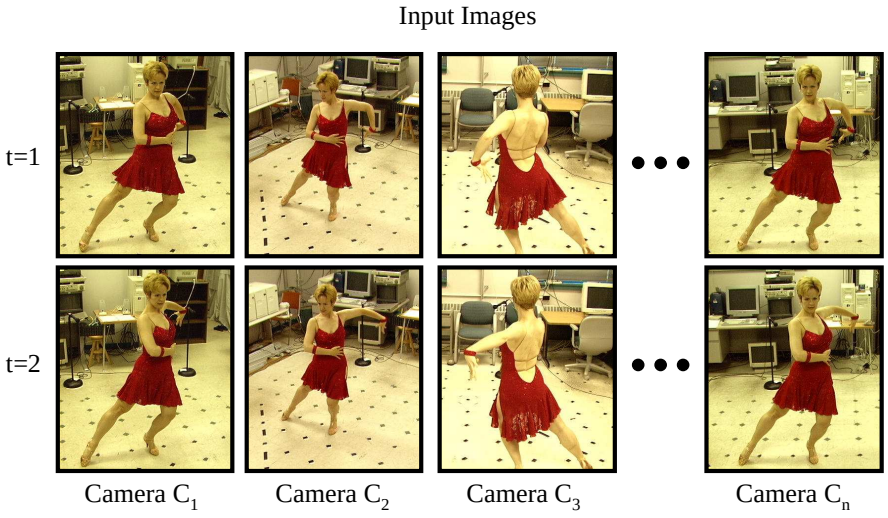


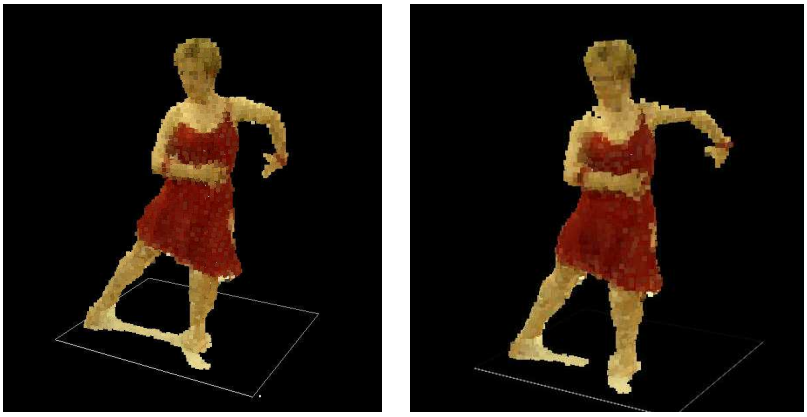
Figure 3: The input to our spatio-temporal view interpolation algorithm is a set of calibrated images at two or more consecutive time instants. From these images, 3D voxel models are computed at each time instant using the voxel coloring algorithm [24]. After we have computed the 3D voxel models, we then compute the dense non-rigid 3D motion or "scene flow" between these models using our scene flow algorithm [30].

umetric approach is shape from silhouette, first proposed in [3]. It reconstructs the visual hull which is a superset of the true shape, but with a large enough number of views, often produces reasonable results. Another example is [7] which uses a plane sweep algorithm, counting the number of image features back-projected to a voxel to determine occupancy. Similarly, Space Carving [24] volumetrically reconstructs the scene by keeping voxels that project to consistent colors in the various images. In [20] a multi-baseline stereo algorithm with volumetric merging is used to compute 3D shape. In [34], shape is recovered using a stereo algorithm iteratively in volumetric space. In addition, there is an entire body of work in two-view stereo algorithms, which may be run separately for pairwise sets of images and then merged in 3D space to compute shape.

For the purposes of this work, any volumetric multi-view shape reconstruction algorithm may be used. We chose the *voxel coloring* algorithm [24], because it is very easy to implement and works remarkably well. Using this algorithm, we compute a 3D voxel model of (the surface of) the scene from the input images individually at each time instant:

$$S^t = \{X_i^t \mid i = 1, \ldots, V^t\} \tag{1}$$

for $t = 1, \ldots, N$ and where $X_i^t = (x_i^t, y_i^t, z_i^t)^{\mathrm{T}}$ is one of the V^t surface voxels at time t. We compute the set of voxels S^t at each time instant t independently. Figure 4 illustrates the two



(a) t=1 (b) t=2

Figure 4: An example of a computed shape using the *voxel coloring* algorithm on the inputs in Figure 3. The voxel models are displayed with the voxels colored with the average color of the pixels that the voxel projects to in the inputs, taking into account only cameras that are visible to any particular voxel.

example voxel models for the input images in Figure 3.

2.3 Scene Flow: The Instantaneous Non-Rigid Motion of a Scene

Since the surfaces of all of the objects in the scene (or simply, the surface of the scene) are what are actually observed in the cameras, we define the scene flow for all points on the scene surface S^t. In particular, we define the scene flow as the instantaneous motion of the surface voxels:

$$F^t = \left\{ \frac{d\mathbf{X}_i^t}{dt} \mid i = 1, \ldots, V^t \right\}. \tag{2}$$

The scene flow is a representation of the scene motion, and is a dense three-dimensional vector field defined for every point on the surface in the scene. Knowledge of the scene flow has numerous potential applications ranging from motion analysis, to motion capture for character animation.

2.3.1 Relationship between Scene Flow and Optical Flow

The scene flow of a voxel describes how it moves across time. If the 3D voxel $X_i^t = (x_i^t, y_i^t, z_i^t)^{\mathrm{T}}$ at time t moves to:

$$X_i^t + F_i^t \;=\; (x_i^t + f_i^t, y_i^t + g_i^t, z_i^t + h_i^t)^{\mathrm{T}} \tag{3}$$

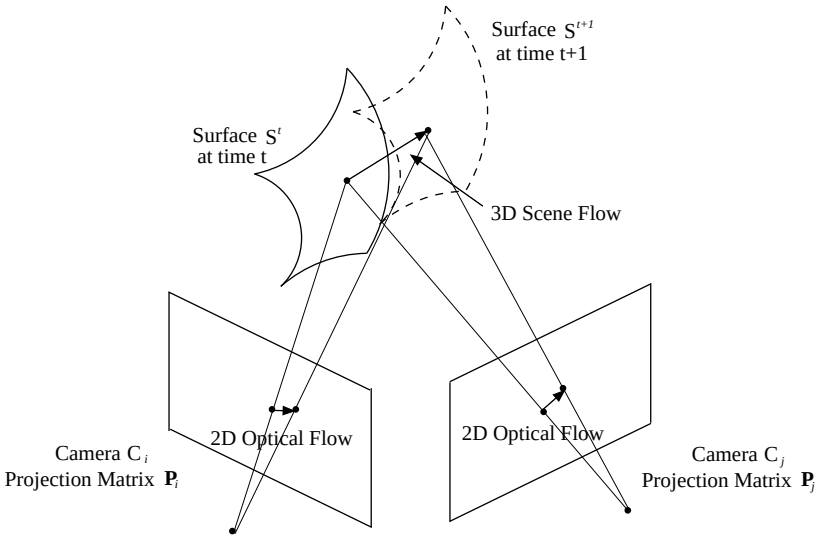


Figure 5: The scene flow is the instantaneous 3D motion of every point on every surface in the scene, illustrated here for a single point. Any optical flow is the 2D projection of the scene flow into an image.

at time $t+1$ its scene flow at time t is $\frac{\mathrm{d}X_i^t}{\mathrm{d}t} = F_i^t = (f_i^t, g_i^t, h_i^t)^{\mathrm{T}}$. We compute the scene flow F_i^t for every voxel in the model S^t at each time instant t using the scene flow algorithm described below.

The scene flow is the 3D motion of points in the world. If $\mathbf{x} = \mathbf{x}(t)$ is the 3D path of a point in the scene, its instantaneous scene flow is $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}$. Suppose that $\mathbf{u}_i(t)$ is the 2D path of the image of the point $\mathbf{x}(t)$ in camera C_i. The optical flow is the 2D motion $\frac{\mathrm{d}\mathbf{u}_i}{\mathrm{d}t}$ in the image plane of camera C_i.

The relationship between the scene flow and an optical flow is illustrated in Figure 5. Optical flow is the 2D projection of the scene flow into an image. The 2D path $\mathbf{u}_i(t)$ depends on the 3D point $\mathbf{x}(t)$. Hence, $\mathbf{u}_i(t) = \mathbf{u}_i(\mathbf{x}(t))$, where the relationship between $\mathbf{u}_i$ and $\mathbf{x}$ is:

$$u_i = \frac{[\mathbf{P}_i]_1 (x, y, z, 1)^{\mathrm{T}}}{[\mathbf{P}_i]_3 (x, y, z, 1)^{\mathrm{T}}} \tag{4}$$

$$v_i = \frac{[\mathbf{P}_i]_2 (x, y, z, 1)^{\mathrm{T}}}{[\mathbf{P}_i]_3 (x, y, z, 1)^{\mathrm{T}}} \tag{5}$$

where $[\mathbf{P}_i]_j$ is the j^{th} row of $\mathbf{P}_i$. Differentiating this expression gives:

$$\frac{\mathrm{d}\mathbf{u}_i}{\mathrm{d}t} = \frac{\partial \mathbf{u}_i}{\partial \mathbf{x}} \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}. \tag{6}$$

At fixed time t the differential relationship between $\mathbf{x}$ and $\mathbf{u}_i$ is represented by the 2×3 Jacobian matrix $\frac{\partial \mathbf{u}_i}{\partial \mathbf{x}}$. The three columns of the Jacobian matrix store the differential change in projected

image coordinates per unit change in x, y, and z. A closed-form expression for $\frac{\partial \mathbf{u}_i}{\partial \mathbf{x}}$ as a function of $\mathbf{x}$ can be derived by differentiating Equations (4) and (5) symbolically.

2.3.2 Computing the Scene Flow from Multiple Optical Flows

Equation (6) expresses the fact that any optical flow $\frac{\mathrm{d}\mathbf{u}_i}{\mathrm{d}t}$ is the projection of the scene flow $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}$. Assuming that the optical flow has been computed, Equation (6) provides two linear constraints on the three unknowns in the scene flow. If we have two or more cameras viewing a particular point in the scene, Equation (6) can therefore easily be solved to recover the scene flow.

A natural question is whether the scene flow can be computed from a single optical flow. In [30] we showed that it is possible to do this if the rate of change of the depth of the scene is also known. The resulting algorithm, however, performs relatively poorly and multiple cameras are required to estimate the rate of change of the depth anyway. Hence we avoid using this algorithm.

Instead we use multiple optical flows from multiple cameras to estimate the scene flow. Each instance of Equation (6) provides two linear constraints on the scene flow. Let $\mathrm{Vis}(\mathbf{x})$ be the set of cameras in which $\mathbf{x}$ is visible and n be the number of cameras in this set. If we have $n > 2$, we can solve for $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}$ by setting up the system of equations $\mathbf{B}\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{U}$, where:

$$\mathbf{B} = \begin{bmatrix} \frac{\partial u_1}{\partial x} & \frac{\partial u_1}{\partial y} & \frac{\partial u_1}{\partial z} \\ \frac{\partial v_1}{\partial x} & \frac{\partial v_1}{\partial y} & \frac{\partial v_1}{\partial z} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \frac{\partial u_n}{\partial x} & \frac{\partial u_n}{\partial y} & \frac{\partial u_n}{\partial z} \\ \frac{\partial v_n}{\partial x} & \frac{\partial v_n}{\partial y} & \frac{\partial v_n}{\partial z} \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} \frac{\partial u_1}{\partial t} \\ \frac{\partial v_1}{\partial t} \\ \cdot \\ \cdot \\ \frac{\partial u_n}{\partial t} \\ \frac{\partial v_n}{\partial t} \end{bmatrix}. \tag{7}$$

This gives us $2n$ equations in three unknowns, and so for $n \geq 2$ we have an over-constrained system. This system of equations is degenerate if and only if the point $\mathbf{x}$ and the n camera centers are collinear. A singular value decomposition of $\mathbf{B}$ gives the solution that minimizes the sum of least squares of the error obtained by re-projecting the scene flow onto each of the optical flows.

Note that it is critical that only the cameras in the set $\mathrm{Vis}(\mathbf{x})$ are used. In our experiments we compute a voxel model of the scene using the *voxel coloring* algorithm [24]. This algorithm keeps track of the visibility of the voxels in the various cameras. We use this estimate of the visibility.

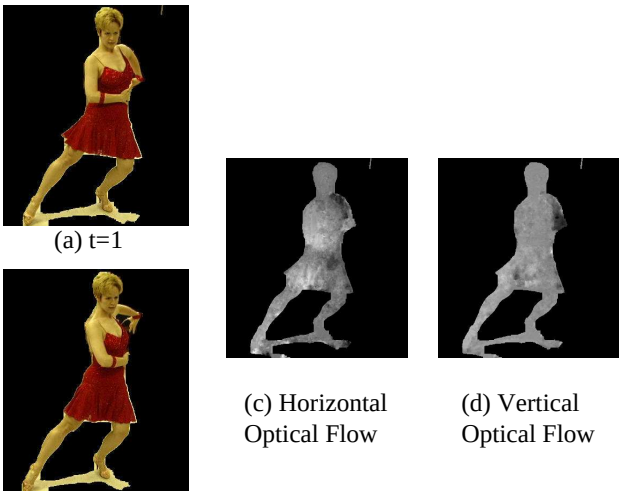


Figure 6: Two input images (shown after background subtraction) and the horizontal and vertical 2D optical flow fields computed using a hierarchical version of the Lucas-Kanade algorithm [17]. Darker pixels indicate larger flow to the right and bottom in the horizontal and vertical flow fields respectively.

Not using optical flow vectors from the regions close to the occlusion boundaries is also a good idea because these flow vectors tend to be relatively erroneous.

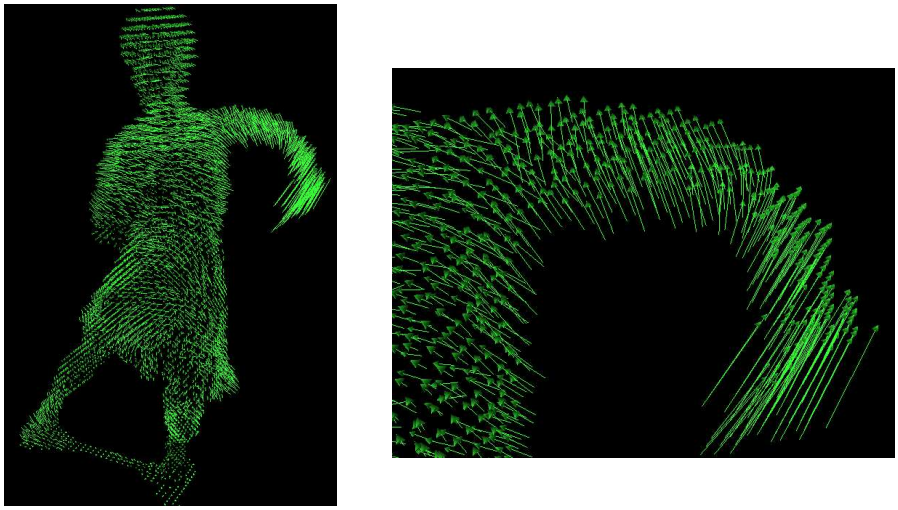
2.3.3 Experimental Results

The first step in the scene flow algorithm consists of computing the optical flow between the two input images for each camera. We use a hierarchical version of the Lucas-Kanade algorithm [17], although any other optical flow algorithm could be used [2]. Figure 6 shows one pair of horizontal and vertical optical flow fields computed for one pair of images in Figure 3. Darker pixels indicate a greater motion to the right and bottom, while lighter pixels indicate motion to the left and upwards.

Figure 7 shows two snapshots of the final computed scene flow. Scene flow is a dense flow field, so a motion vector is computed for every voxel in the scene. The close-up snapshot on the right shows the motion of the voxels as the dancer raises and stretches out her arm. Figure 8 shows the same flow vectors overlaid on the voxel model. Notice that the motion is highly non-rigid.

2.3.4 Computational Cost and Memory Requirements

The computational cost of the scene flow algorithm is linear in the number of cameras and in the number of surface voxels for which flow needs to be computed. In our example, we compute the



(a) (b)

Figure 7: The final scene flow computed from the inputs in Figure 3: (a) the complete model and (b) a close up of the dancer's arm. Notice that the overall motion of the dancer is highly non-rigid.

flow for approximately 6000 surface voxels, from 17 cameras. On an R10000 SGI O2, this takes about 10 seconds. Once the model and the input optical flows are loaded, the memory required is very small since the scene flow is computed one voxel at a time. The main bottleneck is the computation of the optical flow. The hierarchical Lucas-Kanade algorithm that we used takes about 1.5 minutes to compute the optical flow between a single pair of frames on an SGI. On today's machines (for example, a 2.8GHz Pentium IV), however, this time should be down to less than 10 seconds. Faster optical flow algorithms could also be used instead [2].

3 Spatio-Temporal View Interpolation

So far, we have discussed how to create a dynamic model of the time-varying motion of the scene. The geometry at any time instant, together with the scene flow constitutes a complete time-varying model of the dynamic event. With this, we can trace the motion of any point in the scene, or even a pixel in any image, as a continuous function of time. This representation is far richer than simply modeling the event as a sequence of 3D shapes, as in [20]. There, although the shape is available at discrete time instants, there are no temporal correspondences between shapes at neighboring times, and therefore one cannot estimate shape at any time but at the original time instants.

We now have all the intermediates necessary to address the spatio-temporal view interpolation

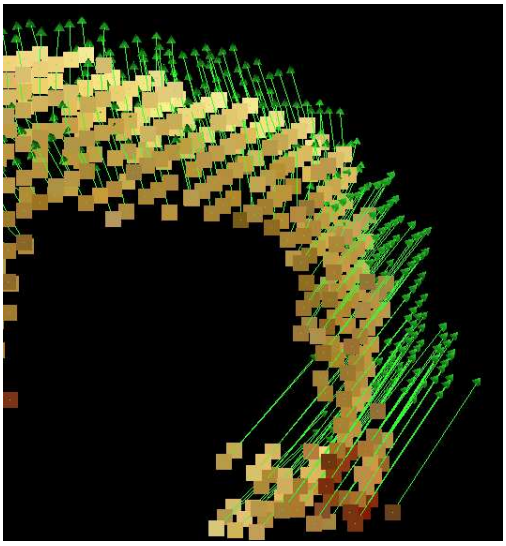


Figure 8: The scene fbw shown together with the voxel model; i.e. a combination of Figures 4 and 7(b).

problem. Recall that we seek to re-create the appearance of the dynamic event from an arbitrary camera position at an arbitrary time. We choose to generate this appearance pixel by pixel, by finding corresponding pixels in other input images and then blending them suitably. Recall the simple case shown in Figure 1, where we have images from two cameras at two different time instants. The subscript represents spatial position, and the superscript represents time. Let a novel image in this space be I^*_+. Now for any pixel (x, y) in this image p^*_+, we have

$$I^*_+(x, y) = f[I^1_1(x^1_1, y^1_1), I^1_2(x^1_2, y^1_2), I^2_1(x^2_1, y^2_1), I^2_2(x^2_2, y^2_2)] \tag{8}$$

where the (x^i_j, y^i_j) are the pixels in each of the four input images that view the same point in the scene as (x, y), and f is some function that weights each of these contributions in a suitable manner. Finding these corresponding pixels (x^i_j, y^i_j) across the different images is the most difficult part. There is no simple closed-form representation for determining the corresponding points, since the correspondence relationship depends on the shape of the scene, the non-rigid motion of the scene, and the visibility relationship that cameras have with different parts of the scene. The function f depends on the position of the novel camera with respect to each of the input cameras (where position refers to the general position in both the spatial and temporal domains).

The example above is simplistic in that it illustrates the case of creating a novel image using sampled images from just two cameras at two time instants. In reality, we need to combine samples

from many more cameras, to account for difficulties with calibration errors, and the fact that just two cameras almost never have completely overlapping areas. In fact, the more complicated the scene is, the more cameras it usually takes to ensure that every part of the scene is visible by at least two cameras (the minimum required to recover shape). The spatio-temporal view interpolation algorithm that we describe uses all images that contain pixels corresponding to a particular pixel in the novel image, and the weighting function also generalizes to an arbitrary number of images.

3.1 High-Level Overview of the Algorithm

Suppose we want to generate a novel image I_+^* from virtual camera C_+ at time t^*, where $t \leq t^* \leq t+1$. The first step is to "flow" the voxel models S^t and S^{t+1} using the scene flow to estimate an interpolated voxel model S^*. The second step consists of fitting a smooth surface to the flowed voxel model S^*. The third step consists of ray-casting across space and time. For each pixel (u, v) in I_+^* a ray is cast into the scene and intersected with the interpolated scene shape (the smooth surface). The scene flow is then followed forwards and backwards in time to the neighboring time instants. The corresponding points at those times are projected into the input images, the images sampled at the appropriate locations, and the results blended to give the novel image pixel $I_+^*(u, v)$. Our algorithm can therefore be summarized as:

1. Flow the voxel models to estimate S^*.
2. Fit a smooth surface to S^*.
3. Ray-cast across space and time.

We now describe these three steps in detail starting with Step 1. Since Step 3. is the most important step and can be explained more easily without the complications of surface fitting, we describe it next, before explaining how fitting a surface modifies the algorithm.

3.2 Flowing the Voxel Models

The scene shape is described by the voxels S^t at time t and the voxels S^{t+1} at time $t+1$. The motion of the scene is described by the scene flow F_i^t for each voxel X_i^t in S^t. We now describe how to interpolate the shapes S^t and S^{t+1} using the scene flow. By comparison, previous work on shape

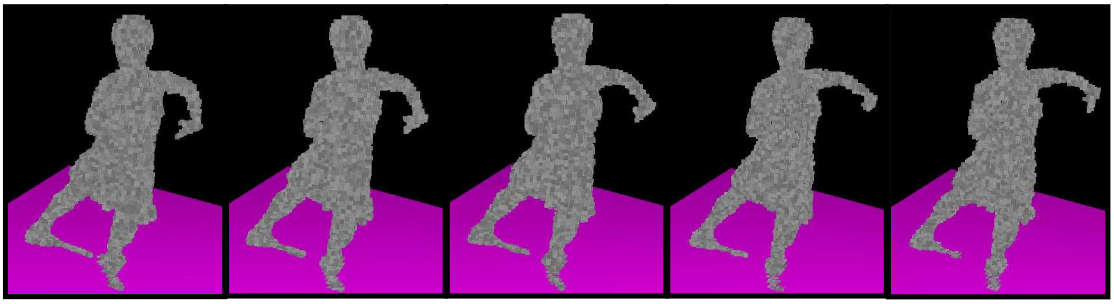


Figure 9: The scene shape can be interpolated between neighboring time instants by flowing the voxels at time t forwards with an appropriate multiple of the scene flow. Notice how the arm of the dancer flows smoothly upwards and outwards from $t = 1.00$ to $t = 2.00$.

interpolation is based solely on the shapes themselves rather than on a flow field connecting them or on interpolating between manually selected feature points [1, 4, 14, 27]. We assume that the voxels move at constant speed in straight lines and so flow the voxels with the appropriate multiple of the scene flow. (In making this constant linear motion assumption we are assuming that the motion is *temporally* "smooth" enough. This assumption does not impose *spatial* smoothness or rigidity on the motion.) If t^* is an intermediate time ($t \leq t^* \leq t+1$), we interpolate the shape of the scene at time t^* as:

$$S^* = \{X_i^t + (t^* - t) \times F_i^t \mid i = 1, \ldots, V^t\} \tag{9}$$

i.e. we flow the voxels forwards from time t. Figure 9 contains an illustration of voxels being flowed in this way. Equation (9) defines S^* in an asymmetric way; the voxel model at time $t+1$ is not even used. Symmetry and other properties of the scene flow are discussed in Section 4.

3.3 Ray-Casting Across Space and Time

Once we have interpolated the scene shape we can ray-cast across space and time to generate the novel image I_+^*. As illustrated in Figure 10, we shoot a ray out into the scene for each pixel (u, v) in I_+^* at time t^* using the known geometry of camera C_+. We find the intersection of this ray with the flowed voxel model. Suppose for now that the first voxel intersected is $X_i^{t^*} = X_i^t + (t^* - t) \times F_i^t$. (Note that we will describe a refinement of this step in Section 3.4.)

We need to find a color for the novel pixel $I_+^*(u, v)$. We cannot project the voxel $X_i^{t^*}$ directly into an image because there are no images at time t^*. We can find the corresponding voxels X_i^t at

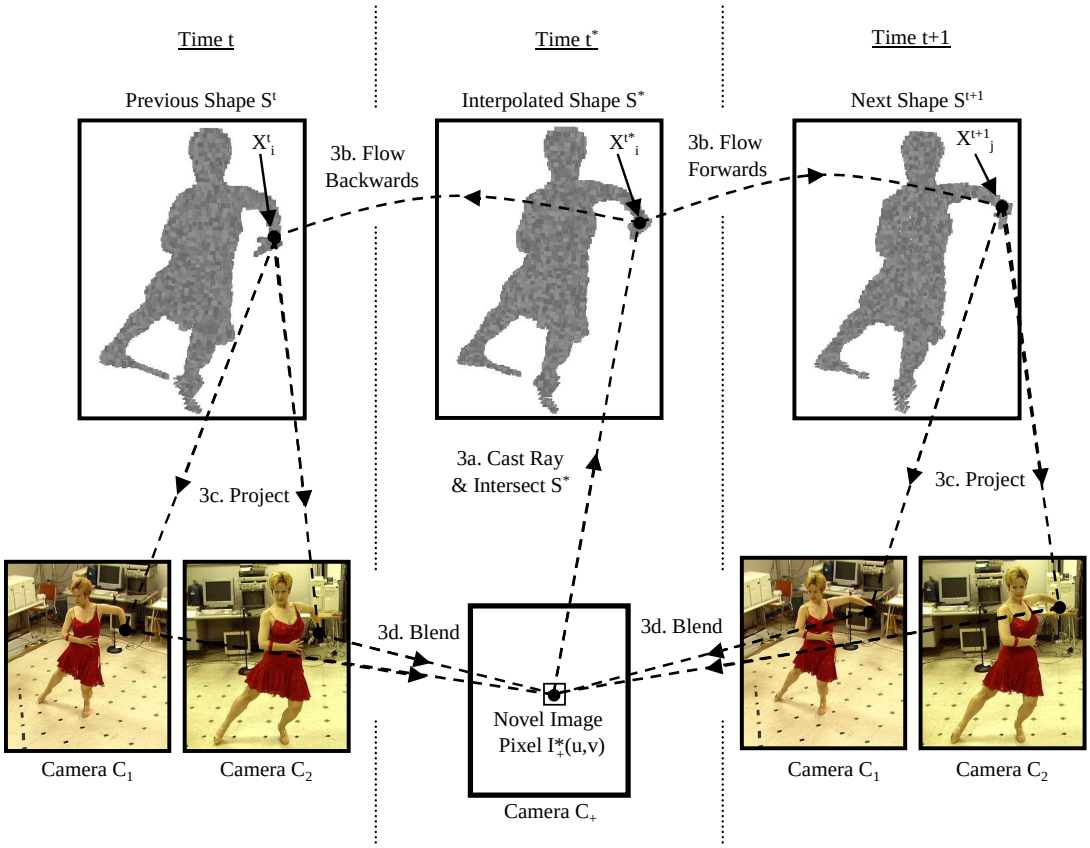


Figure 10: Ray-casting across space and time. 3a. A ray is shot out into the scene at time $t = t^*$ and intersected with the fbwed voxel model. (In Section 3.4 we generalize this to an intersection with a smooth surface fi t to the fbwed voxels.) 3b. The scene fbw is followed forwards and backwards in time to the neighboring time instants. 3c. The voxels at these time instants are projected into the images and the images sub-sampled at the appropriate locations. 3d. The resulting samples are blended to give $I^*_+(u, v)$.

time t and $X^{t+1}_j = X^t_i + F^t_i$ at time $t + 1$, however. We take these voxels and project them into the images at time t and $t + 1$ respectively (using the known geometry of the cameras C_i) to get multiple estimates of the color of $I^*_+(u, v)$. This projection must respect the visibility of the voxels X^t_i at time t and X^{t+1}_j at time $t + 1$ with respect to the cameras at the respective times.

Once the multiple estimates of $I^*_+(u, v)$ have been obtained, they are *blended*. Ideally we would like the weighting function in the blend to satisfy the property that if the novel camera C_+ is one of the input cameras C_i and the time is one of the time instants $t^* = t$, the algorithm should generate the input image I^t_i, *exactly.* We refer to this requirement as the *same-view-same-image* principle. (Theoretically, it is possible to do even better and generate a "super-resolution" [11] image.)

There are two components in the weighting function, space and time. The temporal aspect is

the simpler case. We just have to ensure that when $t^* = t$ the weight of the pixels at time t is 1 and the weight at time $t+1$ is 0. We weight the pixels at time t by $(t+1) - t^*$ and those at time $t+1$ so that the total weight is 1; i.e. we weight the later time $t^* - t$.

The spatial component is slightly more complex because there may be an arbitrary number of cameras. The major requirement to satisfy the principle, however, is that when $C_+ = C_i$ the weight of the other cameras is zero. One way this can be achieved is as follows. Let $\theta_i(u,v)$ be the angle between the rays from C_+ and C_i to the flowed voxel $X_i^{t^*}$ at time t^*. The weight of pixel (u,v) for camera C_i is then:

$$\frac{1/(1-\cos\theta_i(u,v))}{\sum_{j=1}^{\mathrm{Vis}(t,u,v)} 1/(1-\cos\theta_j(u,v))} \tag{10}$$

where $\mathrm{Vis}(t,u,v)$ is the set of cameras for which the voxel X_i^t is visible at time t. This function, a variation of *view-dependent texture mapping* [8], ensures that the weight of the other cameras tends to zero as C_+ approaches one of the input cameras. It is also normalized correctly so that the total weight of all of the visible cameras is 1.0. An equivalent definition is used for the weights at time $t+1$. More sophisticated weighting functions [5] could be used that, for example, take into account the voxel-carving consistency measure, how frontal the surface is, or estimate the parameters of complex BRDF functions [26]. The investigation of such approaches is left as future work.

In summary, ray-casting across space and time consists of the following four steps:

3a. Intersect the (u,v) ray with S^{t^*} to get voxel $X_i^{t^*}$.

3b. Follow the flows to voxels X_i^t and X_j^{t+1}.

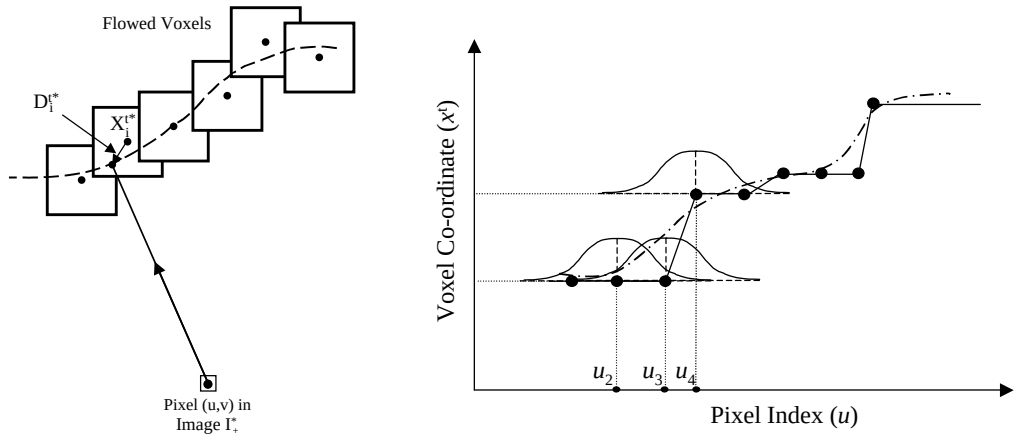
3c. Project X_i^t & X_j^{t+1} into the images at times t & $t+1$.

3d. Blend the estimates as a weighted average.

For simplicity, the description of Steps 3a. and 3b. above is in terms of voxels. We now describe the details of these steps when we fit a smooth surface through these voxels.

3.4 Ray-Casting to a Smooth Surface

The ray-casting algorithm described above casts rays from the novel image onto the model at the novel time t^*, finds the corresponding voxels at time t and time $t+1$, and then projects those



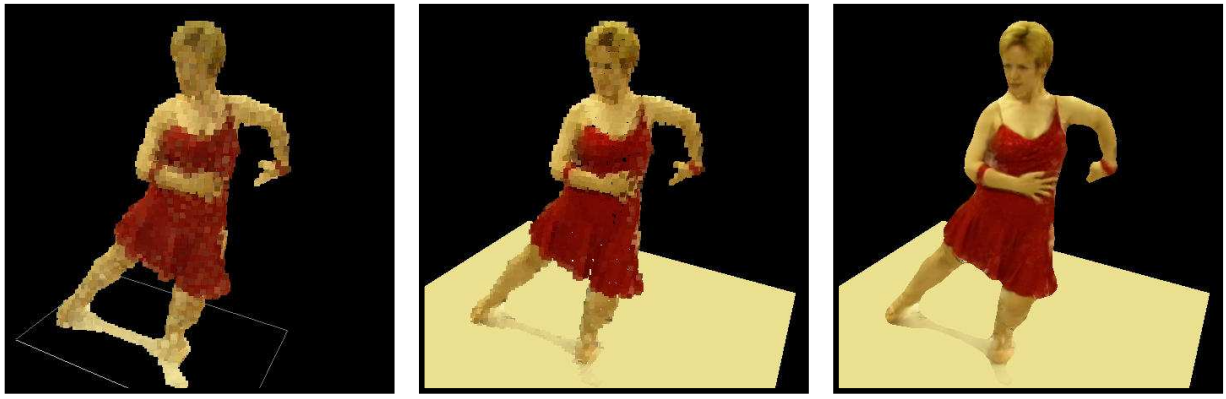
(a) (b)

Figure 11: (a) Ray-casting to a smooth surface. We intersect each cast ray with a smooth surface interpolated through the voxel centers. The perturbation to the point of intersection D_i^{t*} can then be transferred to the previous and subsequent time instants. (b) The voxel coordinate changes in an abrupt manner for each pixel in the novel image. Convolution with a simple Gaussian kernel centered on each pixel changes its corresponding 3D coordinate to approximate a smoothly fi t surface.

points into the images to find a color. However, the reality is that voxels are just point samples of an underlying smooth surface. If we just use voxel centers, we are bound to see cubic voxel artifacts in the final image, at least unless the voxels are extremely small.

The situation is illustrated in Figure 11(a). When a ray is cast from the pixel in the novel image, it intersects one of the voxels. The algorithm, as described above, simply takes this point of intersection to the be center of the voxel X_i^{t*}. If, instead, we fit a smooth surface to the voxel centers and intersect the cast ray with that surface, we get a slightly perturbed point $X_i^{t*} + D_i^{t*}$. Assuming that the scene flow is constant within each voxel, the corresponding point at time t is $X_i^t + D_i^{t*}$. Similarly, the corresponding point at $t+1$ is $X_j^{t+1} + D_i^{t*} = X_i^t + F_i^t + D_i^{t*}$. If we simply use the centers of the voxels as the intersection points rather than the modified points, a collection of rays shot from neighboring pixels will all end up projecting to the same points in the images, resulting in obvious box-like artifacts. See Figure 12(b) for an illustration of such artifacts.

Fitting a surface through an arbitrary set of voxel centers in 3D is a well studied problem [12, 16]. Most algorithms only operate on regular grids, however. Fitting a 3D surface through the voxel centers of an irregular grid is a much harder problem. However, the main requirement of the fit surface in our case is just that it prevents the discrete jump while moving from one voxel



(a) Colored Voxel Model (b) Ray-Casting With Cubic Voxels (c) Ray-Casting With Surface Fit

Figure 12: The importance of fitting a smooth surface. (a) The voxel model rendered as a collection of voxels, where the color of each voxel is the average of the pixels that it projects to. (b) The result of ray-casting without surface fitting, showing that the voxel model is a coarse approximation. (c) The result of intersecting the cast ray with a surface fit through the voxel centers results in a far better rendering.

to a neighbor. What is important is that the interpolation between the coordinates of the voxels be smooth. We propose the following simple algorithm to approximate the surface fit.

For each pixel u_i in the novel image, the 3D coordinates of the corresponding voxel at time t, $X^t = (x^t, y^t, z^t)$ are stored to give a 2D array of (x, y, z) values. Figure 11(b) shows the typical variation of the x component of X^t with the image coordinate u_i. Because of the discrete nature of the voxels, this function changes abruptly at the voxel centers, whereas, we really want it to vary smoothly, say like the dotted line. We apply a Gaussian smoothing operator centered at each pixel (shown for u_2, u_3, and u_4) to the function $x^t(u)$ to get a new value of $\overline{x}^t$ (and similarly for $y^t(u)$ and $z^t(u)$). The smoothed 3D vector function $\overline{X}^t = (\overline{x}^t, \overline{y}^t, \overline{z}^t)$ is then used in place of $X^t = (x^t, y^t, z^t)$; i.e. the perturbation $D_i^{t*} = \overline{X}^t - X^t$.

Figure 12 illustrates the importance of surface fitting. Figure 12(a) shows the voxel model rendered as a collection of voxels. The voxels are colored with the average of the colors of the pixels that they project to. Figure 12(b) shows the result of ray-casting, but just using the voxels. Figure 12(c) shows the result after intersecting the cast ray with the smooth surface. As can be seen, without the surface fitting step the rendered images contain substantial voxel artifacts.

4 Ideal Properties of the Scene Flow

In Section 3.2 we described how to flow the voxel model forward to estimate the interpolated voxel model S^*. In particular, Equation (9) defines S^* in an asymmetric way; the voxel model S^{t+1} at time $t+1$ is not even used. A related question is whether the interpolated shape is continuous as $t^* \to t+1$? Ideally we want this property to hold, but how do we enforce it?

One suggestion might be that the scene flow should map *one-to-one* from S^t to S^{t+1}. Then, the interpolated scene shape will definitely be continuous. The problem with this requirement, however, is that it implies that the voxel models must contain the same number of voxels at times t and $t+1$. It is therefore too restrictive to be useful. For example, it outlaws motions that cause the shape to expand or contract. The properties that we really need are:

Inclusion: Every voxel at time t should flow to a voxel at time $t+1$: i.e. $\forall t, i\ X_i^t + F_i^t \in S^{t+1}$.

Onto: Every voxel at time $t+1$ should have a voxel at time t that flows to it: $\forall t, i,\ \exists j$ s.t. $X_j^t + F_j^t = X_i^{t+1}$.

These properties imply that the voxel model at time t flowed forward to time $t+1$ is *exactly* the voxel model at $t+1$:

$$\{X_i^t + F_i^t \mid i = 1, \ldots, V^t\} \ = \ S^{t+1}. \tag{11}$$

This means that the scene shape will be continuous at $t+1$ as we flow the voxel model forwards.

4.1 Duplicate Voxels

Is it possible to enforce these two conditions without the scene flow being one-to-one? It may seem impossible because the second condition seems to imply that the number of voxels cannot get larger as t increases. It is possible to satisfy both properties, however, if we introduce what we call *duplicate voxels*. Duplicate voxels are additional voxels at time t which flow to different points at $t+1$; i.e. we allow two voxels X_i^t and X_j^t ($i \neq j$) where $(x_i^t, y_i^t, z_i^t) = (x_j^t, y_j^t, z_j^t)$ but yet $F_i^t \neq F_j^t$. A voxel model is then still just a set of voxels and but can satisfy the two desirable properties above. There may just be a number of duplicate voxels with different scene flows. Note, that the main purpose of duplicates voxels is essentially to eliminate holes. Other techniques for removing holes include "splatting" [35] and sparse surface fitting [9].

Duplicate voxels also make the formulation more symmetric. If the two properties *inclusion* and *onto* hold, the flow can be inverted in the following way. For each voxel at the second time instant there are a number of voxels at the first time instant that flow to it. For each such voxel we can add a duplicate voxel at the second time instant with the inverse of the flow. Since there is always at least one such voxel (onto) and every voxel flows to some voxel at the second time (inclusion), when the flow is inverted in this way the two properties also hold for the inverse flow.

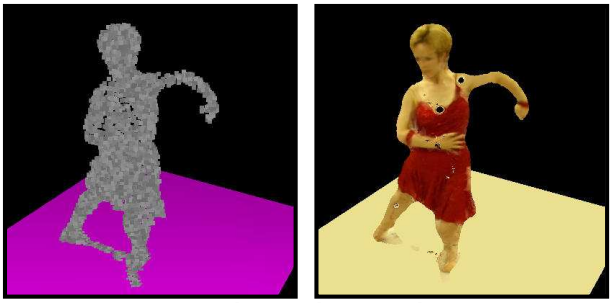
So, given forwards scene flow where *inclusion* and *onto* hold, we can invert it using duplicate voxels to get a backwards scene flow for which the properties hold also. Moreover, the result of flowing the voxel model forwards from time t to t^* with the forwards flow field is the same as flowing the voxel model at time $t+1$ backwards with the inverse flow. We can then formulate shape interpolation symmetrically as flowing either forwards and backwards. Whichever way the flow is performed, the result will be the same.

The scene flow algorithm described in Section 2 does not guarantee either of the two desirable properties. Therefore, we take the scene flow computed there and modify it as little as possible to ensure that the two properties hold. First, for each voxel X_i^t we find the closest voxel in S^{t+1} to $X_i^t + F_i^t$ and change the flow F_i^t so that X_i^t flows there. Second, we take each voxel X_i^{t+1} at time $t+1$ that does not have a voxel flowing to it and add a duplicate voxel at time t that flows to it by averaging the flows in neighboring voxels at $t+1$.

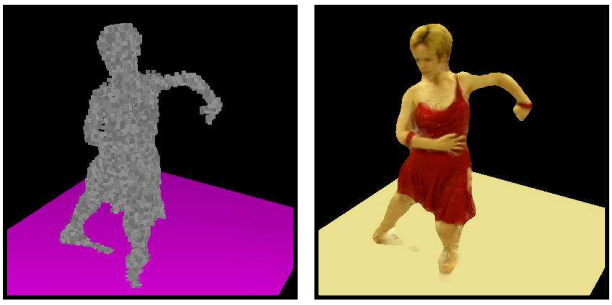
4.2 Results With and Without Duplicate Voxels

The importance of the duplicate voxels is illustrated in Figure 13. This figure contains two rendered views at an intermediate time, one with duplicate voxels and one without. Without the duplicate voxels the model at the first time instant does not flow *onto* the model at the second time. When the shape is flowed forwards holes appear in the voxel model (left) and in the rendered view (right). With the duplicate voxels the voxel model at the first time does flow *onto* the model at the second time and the artifacts disappear.

The need for duplicate voxels is illustrated in the movie "duplicate_voxels.mpg" (available from http://www.ri.cmu.edu/projects/project_464.html.) This movie consists of a sequence of frames generated using our algorithm to interpolate across time only. (Results interpolating across space are included later.) The movie contains a side-by-side comparison with and without duplicate



(a) Without Duplicate Voxels



(b) With Duplicate Voxels

Figure 13: A rendered view at an intermediate time, with and without duplicate voxels. Without the duplicate voxels, the model at the first time does not flow *onto* the model at the second time. Holes appear where the missing voxels should be. The artifacts disappear when the duplicate voxels are added.

voxels. Without the duplicate voxels (right) the movie is jerky because the interpolated shape is discontinuous. With the duplicate voxels (left) the movie is very smooth. The best way to observe this effect is to play the movie several times. The first time concentrate on the left hand side with the duplicate voxels. The second time concentrate on the right hand side. Finally, play the movie one last time and study both sides at the same time.

5 Optimization Using Graphics Hardware

Spatio-temporal view interpolation involves two fairly computationally expensive operations. It is possible to optimize these using standard graphics hardware as we now discuss.

5.1 Intersection of Ray with Voxel Model

Steps 3a. and 3b. of our algorithm involve casting a ray from pixel (u, v) in the novel image, finding the voxel $X_i^{t^*}$ that this ray intersects, and then finding the corresponding voxels X_i^t and X_i^{t+1} at

the neighboring time instants. Finding the first point of intersection of a ray with a voxel model is potentially an expensive step, since the naive algorithm involves an exhaustive search over all voxels. In addition, extra book-keeping is necessary to determine the corresponding voxels at times t and $t+1$ for each flowed voxel $X_i^{t^*}$.

We implement this step as follows. Each voxel in the model S^t is given a unique ID, which is encoded as a unique (r, g, b) triplet. This voxel model is then flowed as discussed in Section 3.2 to give the voxel model S^* at time t^*. This voxel model S^* (see Equation (9)) is then rendered as a collection of little cubes, one for each voxel, colored with that voxel's unique ID. In particular, the voxel model S^* is rendered from the viewpoint of the novel camera using standard OpenGL. Lighting is turned off (to retain the base color of the cubes), and z-buffering turned on, to ensure that only the closest voxel along the ray corresponding to any pixel is visible. Immediately after the rendering, the color buffers are read and saved. Indexing the rendered image at the pixel (u, v) gives the ID (the (r, g, b) value) of the corresponding voxel at time t (and hence at time $t+1$.)

This method of using color to encode a unique ID for each geometric entity is similar to the *item buffer* [32] which is used for visibility computation in ray tracing.

5.2 Determining Visibility of the Cameras

Step 3c. of our algorithm involves projecting X_i^t and X_j^{t+1} into the input images. But how do we compute whether the X_i^t was actually visible in camera C_k?

Again, we use a z-buffer approach similar to the previous case, except this time, we don't need to encode any sort of information in the color buffers (that is, there are no voxel IDs to resolve). The occlusion test for Camera C_k runs as follows. Let $\mathbf{R}_k$ and $\mathbf{t}_k$ be the rotation matrix and translation vector for camera C_k relative to the world coordinate system. Then, X_i^t is first transformed to camera coordinates:

$$\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{bmatrix} \mathbf{R}_k & \mathbf{t}_k \\ \mathbf{0} & 1 \end{bmatrix} \mathbf{X}_i^t. \tag{12}$$

The image coordinates of the projection $\mathbf{u} = (u, v)$ are obtained by multiplying by the 3×4 camera matrix P_k

$$\mathbf{u} = P_k \mathbf{x}. \tag{13}$$

The voxel model is then rendered, with the camera transformation matrix set to be exactly that corresponding to the calibration parameters of camera C_k. After the rendering, the hardware z-buffer is read. This z-buffer now gives the depth to the nearest point on the shape for any particular pixel in the rendered image, and therefore any pixel in the real image as well, since the viewpoints are identical for both. In reality, the value of the hardware z-buffer is between zero and one, since the true depth is transformed by the perspective projection that is defined by the near and far clipping planes of the viewing frustum. However, since these near and far clipping planes are user-specified, the transformation is easily invertible and the true depth-map can be recovered from the value of the z-buffer at any pixel.

Let (u, v) be the image coordinates in image I_k, as computed from Equation (13). The value of the z-buffer at that pixel, $z_k(u, v)$ is compared against the value of x_3 (which is the distance to the voxel $\mathbf{X}(i, j)$ from the camera). If $x_3 = z_k(u, v) + h$, where h is half the length of the side of the voxel (i.e. the distance from the center to the front of the voxel), then the voxel $\mathbf{X}(i, j)$ is visible in the camera. Instead if $x_3 > z_k(u, v) + h$ the voxel $\mathbf{X}(i, j)$ is occluded by another part of the scene.

6 Experimental Results

We applied our spatio-temporal view interpolation algorithm to two highly non-rigid dynamic events: a "Paso Doble Dance Sequence" and a "Person Lifting Dumbbells." These events were both imaged in the CMU 3D Room [13].

6.1 Sequence 1: Paso Doble Dance Sequence

The first event is a "Paso Doble Dance Sequence", the example sequence used in the paper so far. See Figure 3 for four example input images at two time instants. In the sequence, the dancer turns as she un-crosses her legs and raises her left arm.

The input to the algorithm consists of 15 frames from each of 17 cameras, in total 255 images. Hand-marking point correspondences in 255 images is clearly impossible. Hence it is important that our algorithm is fully automatic. The input frames for each of the cameras are captured 1/10 of a second apart, so the entire sequence is 1.5 seconds long. The 17 cameras are distributed all around the dancer, with 12 of the cameras on the sides, and five cameras overhead.

Figure 14 shows a collection of frames from a virtual slow-motion fly-through of this dance

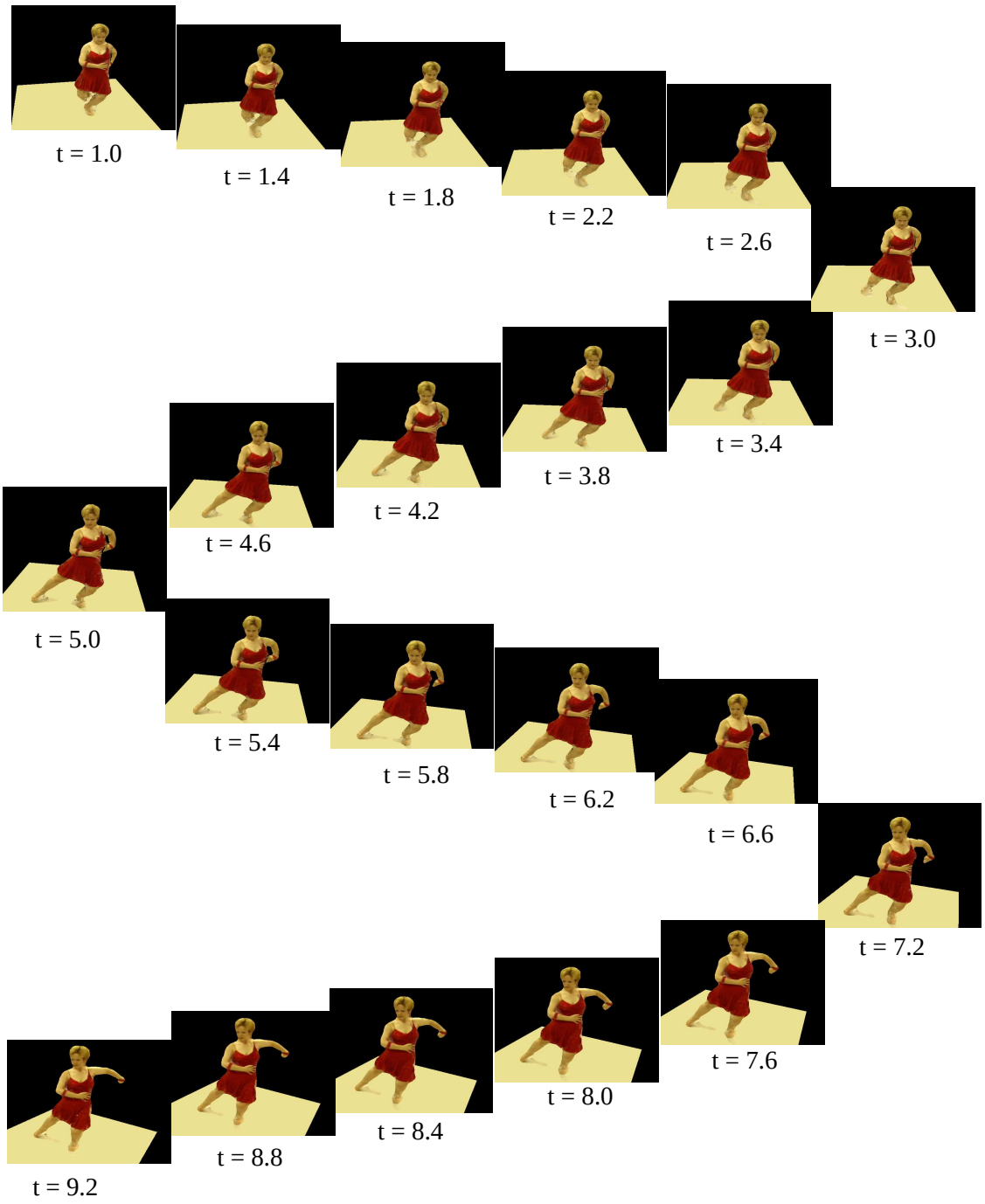


Figure 14: A collection of frames from the movie "dance_flyby.mpg" (available from our website at http://www.ri.cmu.edu/projects/project_464.html) created from the "Paso Doble Dance Sequence." Some of the inputs are included in Figure 3. The virtual camera moves along a path that first takes it towards the scene, then rotates it around the scene, and finally takes it away from the dancer. The new sequence is also re-timed to be ten times slower than the original camera speed. In the movie, we include a side by side comparison with the closest input image in terms of both space and time. This comparison makes the inputs appear like a collection of snap-shots compared to our spatio-temporally interpolated movie.

sequence. The path of the camera is initially towards the scene, then rotates around the dancer, and finally moves away. Watch the floor (which is fixed) to get a good idea of the camera motion. We create 9 synthetic views between each neighboring pair of input frames. The complete movie "dance_flyby.mpg," (also available from http://www.ri.cmu.edu/projects/project_464.html), was created by assembling the spatio-temporal interpolated images. Also shown is a comparison with what would have been obtained had we just switched between the closest input images, measured both in space and time. Note that the movie created in this manner looks like a collection of snap-shots, whereas the spatio-temporal fly-by is a much smoother and natural looking re-rendering of the event.

There are some visible artifacts in the fly-by movie, such as slight blurring, and occasional discontinuities. The blurring occurs because our shape estimation is imperfect. Therefore corresponding points from neighboring cameras are slightly misaligned. The discontinuities are because of imperfect scene flow; a few voxels flow to the wrong place at the next time instant.

6.2 Sequence 2: Person Lifting Dumbbells

The second event consists of a "Person Lifting Dumbbells". The person pushes up a pair of dumbbells from their shoulder to full arm extension, and then brings them down again. The input to the algorithm consists of 9 frames from each of 14 cameras for a total of 126 images. Again, hand-marking point correspondences in so many images would be impossible. Two images at consecutive time steps from each of two cameras are shown in Figure 15. This figure also contains the voxel model computed at the first time instant and the scene flow computed between these models.

From these inputs we used our spatio-temporal view interpolation algorithm to generate a re-timed slow-motion movie of this sequence. Again, we interpolated 9 frames between each pair in the input. To better illustrate the motion, we also left the novel viewpoint fixed in space. The novel viewpoint doesn't correspond to any of the camera views and could easily be changed. Figure 16 shows a number of sample frames from the re-timed movie "dumbbell_slowmo.mpg", also available from http://www.ri.cmu.edu/projects/project_464.html. Notice the complex non-rigid motion on the shirt as the person flexes his muscles.

Just like for the dance sequence, we also include a side-by-side comparison with the closest input image. In this movie, the viewpoint doesn't change and so the closest image always has the

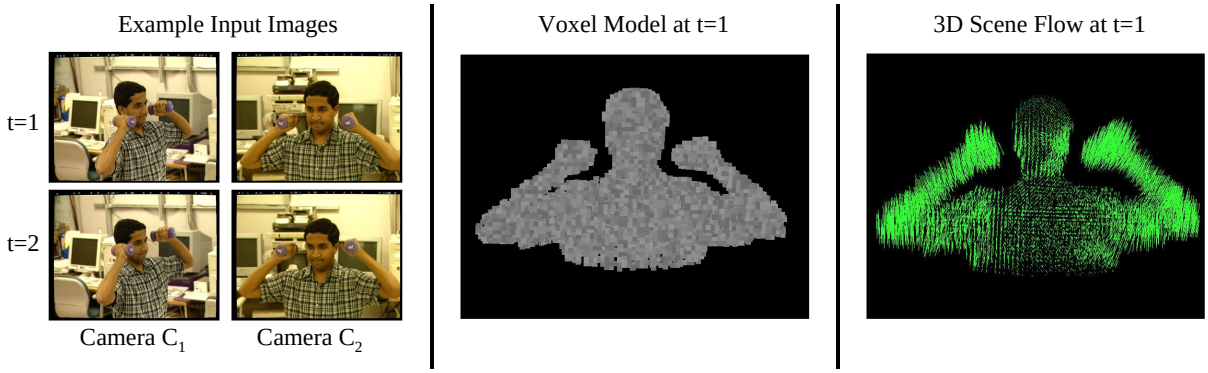


Figure 15: Some of the input images, an example voxel model, and one of the scene flows used for the "dumbbell" sequence. The complete sequence consists of 9 frames from each of 14 cameras for a total of 126 images. Hand-marking point correspondences in this many images would be very time consuming. From these images we compute 9 voxel models, one at each time instant, and 8 pairs of scene flows between them. One voxel model and one scene flow are shown. We then use all this information to generate a re-timed slow-motion movie of the event. See Figure 16 for snapshots of the movies clip created.

same pose. The closest image in time just steps through the original sequence from that camera.

6.3 Computational Cost and Memory Requirements

The computation time of spatio-temporal view interpolation is linear in the number of pixels in the output image, irrespective of the complexity of the model, as for most image-based rendering algorithms. It is also linear in the number of input images that are used to contribute to each pixel of the output. In our examples we compute a 640×480 novel image, using the six closest images (three closest cameras at each of two time instants). The algorithm takes about 5 seconds to run for each output frame on an SGI 02, using graphics hardware to compute the ray-voxel intersection and the visibility. The largest memory requirement is storing the input images. Compared to this, the overhead of storing the voxels and the scene flow is minimal. See [28] for more details.

7 Conclusions

In this paper we have formulated the problem of spatio-temporal modeling and view interpolation which involves creating a dynamic model and re-rendering a non-rigidly varying, real-world event using images captured from different positions in space and time.

In doing so, we have introduced the concept of scene flow to characterize non-rigid scene motion and developed a theory of how it relates to scene geometry and optical flow. We have

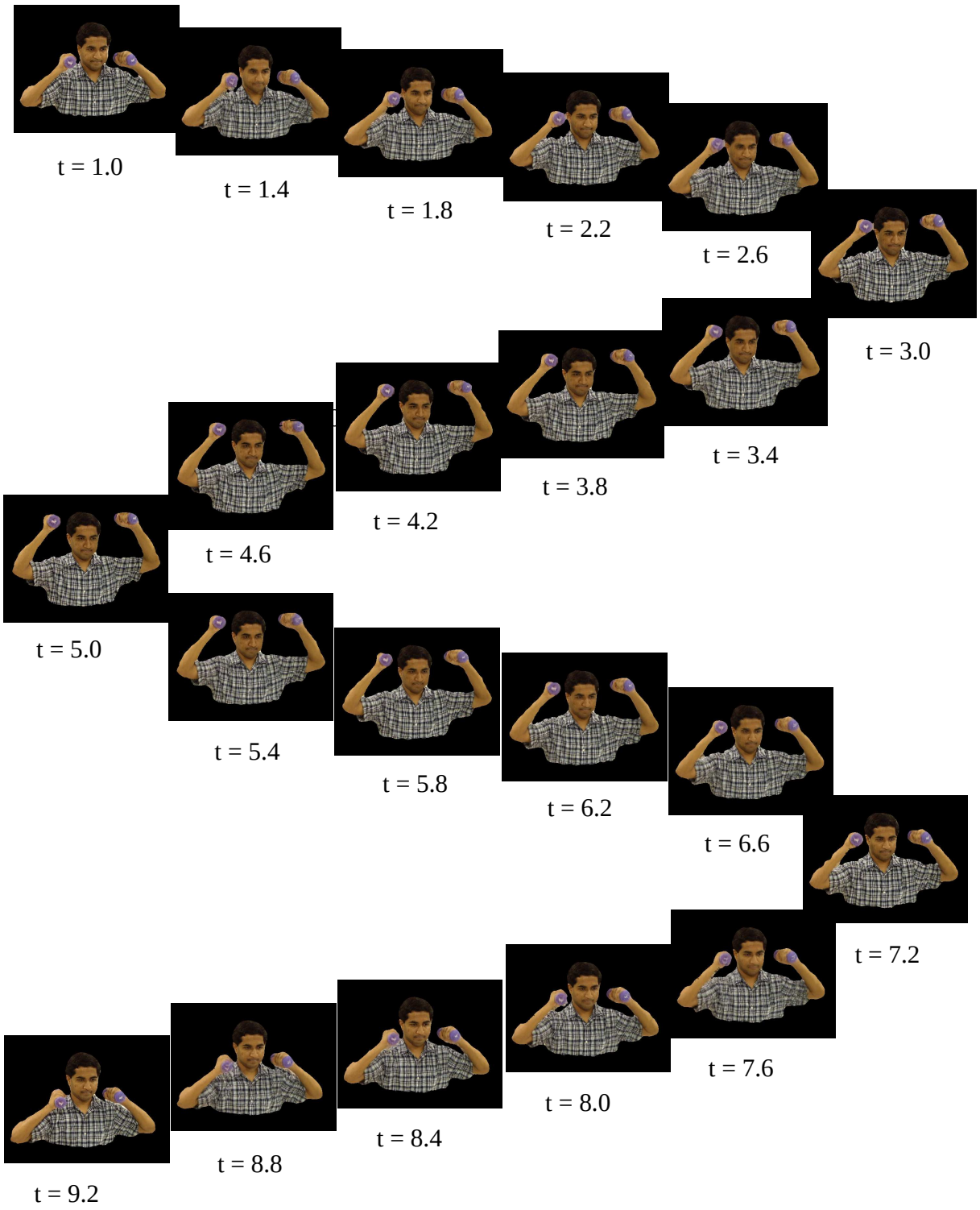


Figure 16: A collection of frames from the re-timed slow motion movie "dumbbell_slowmo.mpg" of the "dumbbell" sequence. See Figure 15 for an illustration of the inputs. Notice the complex non-rigid motion of the shirt and the articulated motion of the arms. The movie "dumbbell_slowmo.mpg" is also available from our website at URL http://www.ri.cmu.edu/projects/project_464.html.

also proposed an algorithm to compute scene flow, giving us a continuous 4D model of the scene (geometry and instantaneous motion) without using any domain-specific knowledge.

Our spatio-temporal view interpolation algorithm provides a way of generating novel views of the event. Using the geometry and motion model computed in the spatio-temporal modeling phase, the sampled images are combined to synthesize the appearance of time-varying real world events from novel viewpoints at previously unsampled time instants. We have shown results of the algorithm by generating movie clips that show visual fly-bys of the event taking place. They are also re-timed, thus giving smooth slow motions replays with a dynamic viewpoint. This continuous control over both the position and the timing of any one shot has many potential applications.

8 Future Work

This paper has been a first attempt at an approach for spatio-temporal modeling and view interpolation, and lends itself to a number of possible extensions and new research directions. One immediate possibility is the investigation of other approaches based on extensions of Lumigraphs [10, 5] or image-based visual hulls [19], to solve the same problem.

Our algorithm to compute scene flow is also by no means the only possible one. People are still proposing new optical flow algorithms after over twenty years of work. A natural question is whether it is better to first compute optical flow and then scene flow, or instead just compute the scene flow directly from the image measurements, as in [31], or perhaps using a 3D version of Lucas-Kanade [17]. Empirically, we found our optical flow-based algorithm [30] to work better than the direct scene-flow algorithm. This, however, does not mean that there are not better direct algorithms. Note, however, than our rendering algorithm is somewhat robust to erroneous scene flow in constant intensity regions because a slightly wrong motion would still end up sampling inside the constant intensity region.

It was assumed that no higher level information about scene geometry was known. If for example, we know that the scene only involves humans - articulated models of shape and/or knowledge of human behavioral patterns can constrain the search space significantly, while also providing implicit smoothness priors. On the other hand, the domain knowledge needs to be good - errors due to calibration and real-world noise can result in the system not converging to any solution, while attempting to fit the data to a model that is too stiff.

An interesting application for spatio-temporal view interpolation is the modeling of fast moving scenes, where the normal temporal sampling rate is inadequate. This will give us a unique 3D re-timing capability, so that fly-through reconstructions can be performed in slow motion, without the familiar jerkiness associated with the limited number of frames in regular slow-motion replays. In fact, with the continuous control of timing and spatial position of the novel viewpoint, special effects such as motion blur, time-lapsed fades, and other effects so far found only in video games (that use purely synthetic data) can now be extended to use visual models of the real world.

Acknowledgments

Elements of this work appeared previously in [29] and [30]. We would like to thank the reviewers of both of those papers, as well as the ACM TOG reviewers, for their comments and suggestions.

References

[1] M. Alexa, D. Cohen-Or, and D. Levin. As-rigid-as-possible shape interpolation. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 2000.

[2] J.L. Barron, D.J. Fleet, and S.S. Beauchemin. Performance of optical flow techniques. *International Journal of Computer Vision*, 12(1):43–77, 1994.

[3] B.G. Baumgart. *Geometric Modeling for Computer Vision*. PhD thesis, Stanford University, Palo Alto, 1974.

[4] D. Breen and R. Whitaker. A level-set approach for the metamorphosis of solid models. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):173–192, 2001.

[5] C. Buehler, M. Bosse, L. McMillan, S. Gortler, and M. Cohen. Unstructured lumigraph rendering. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 2001.

[6] S.E. Chen and L. Williams. View interpolation for image synthesis. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 1993.

[7] R. Collins. A space sweep approach to true multi-image matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 358–363, June 1996.

[8] P.E. Debevec, C.J. Taylor, and J. Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 1996.

[9] H.Q. Dinh, G. Turk, and G. Slabaugh. Reconstructing surfaces by volumetric regularization using radial basis functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(10):1358–1371, 2002.

[10] S.J. Gortler, R. Grzeszczuk, R. Szeliski, and M.F. Cohen. The Lumigraph. In *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH)*, 1996.

[11] T.S. Huang and R. Tsai. Multi-frame image restoration and registration. *Advances in Computer Vision and Image Processing*, 1:317–339, 1984.

[12] A. Kadosh, D. Cohen-Or, and R. Yagel. Tricubic interpolation of discrete surfaces for binary volumes. *IEEE Transactions on Visualization and Computer Graphics*, 2000.

[13] T. Kanade, H. Saito, and S. Vedula. The 3D room: Digitizing time-varying 3D events by synchronized multiple video streams. Technical Report CMU-RI-TR-98-34, Robotics Institute, Carnegie Mellon University, 1998.

[14] A. Lerios, C.D. Garfinkle, and M. Levoy. Feature-based volume metamorphosis. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 1995.

[15] M. Levoy and M. Hanrahan. Light field rendering. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 1996.

[16] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3D surface reconstruction algorithm. *Computer Graphics*, 21(4):163–169, 1992.

[17] B.D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *International Joint Conference on Artificial Intelligence*, 1981.

[18] R. Manning and C. Dyer. Interpolating view and scene motion by dynamic view morphing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1999.

[19] W. Matusika, C. Buehler, R. Raskar, L. McMillan, and S. Gortler. Image-based visual hulls. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 2000.

[20] P.J Narayanan, P.W. Rander, and T. Kanade. Constructing virtual worlds using dense stereo. In *Proceedings of the Sixth IEEE International Conference on Computer Vision*, 1998.

[21] Y. Sato, M. Wheeler, and K. Ikeuchi. Object shape and reflectance modeling from observation. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 1997.

[22] E. Schechtman, Y. Capsi, and M. Irani. Increasing space-time resolution in video. In *Proceedings of the Seventh European Conference on Computer Vision*, 2002.

[23] S.M. Seitz and C.R. Dyer. View morphing. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 1996.

[24] S.M. Seitz and C.R. Dyer. Photorealistic scene reconstruction by voxel coloring. *International Journal of Computer Vision*, 35(2):151–173, 1999.

[25] R .Tsai. An efficient and accurate camera claibration technique for 3d machine vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1986.

[26] G. Turk and M. Levoy. Zippered polygon meshes from range images. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 1994.

[27] G. Turk and J.F. O'Brien. Shape transformation using variational implicit functions. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 1999.

[28] S. Vedula. *Image Based Spatio-Temporal Modeling and View Interpolation of Dynamic Events*. PhD thesis, Robotics Institute, Carnegie Mellon University, 2001.

[29] S. Vedula, S. Baker, and T. Kanade. Spatio-temporal view interpolation. In *Proceedings of the 13th ACM Eurographics Workshop on Rendering*, June 2002.

[30] S. Vedula, S. Baker, P. Rander, R. Collins, and T. Kanade. Three-dimensional scene flow. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 1999.

[31] S. Vedula, S. Baker, S. Seitz, and T. Kanade. Shape and motion carving in 6D. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, 2000.

[32] H. Weghorst, G. Hooper, and D.P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, 1984.

[33] Y. Wexler and A. Shashua. On the synthesis of dynamic scenes from reference views. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2000.

[34] C. Zitnick and T. Kanade. A volumetric iterative approach to stereo matching and occlusion detection. Technical Report CMU-RI-TR-98-30, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1998.

[35] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Computer Graphics Annual Conference Series (SIGGRAPH)*, 2001.