Programmable Motion Effects

Johannes Schmid
Disney Research Zurich
ETH Zurich

Robert W. Sumner
Disney Research Zurich

Huw Bowles
Disney Research Zurich

Markus Gross
Disney Research Zurich
ETH Zurich

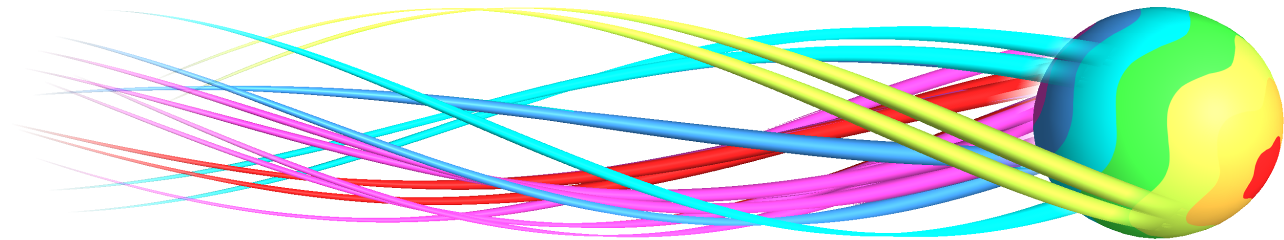

Figure 1: *The trails behind this spinning ball are generated by a programmable motion effect, a new rendering mechanism for depicting motion in computer-generated animation. Customized styles can be achieved in a flexible manner, similar to the way programmable surface shaders are used to generate customized surface appearances.*

Abstract

Although animation is one of the most compelling aspects of computer graphics, the possibilities for depicting the movement that make dynamic scenes so exciting remain limited for both still images and animations. In our work, we experiment with motion depiction as a first-class entity within the rendering process. We extend the concept of a surface shader, which is evaluated on an infinitesimal portion of an object's surface at one instant in time, to that of a programmable motion effect, which is evaluated with global knowledge about all portions of an object's surface that pass in front of a pixel during an arbitrary long sequence of time. With this added information, our programmable motion effects can decide to color pixels long after (or long before) an object has passed in front of them. In order to compute the input required by the motion effects, we propose a 4D data structure that aggregates an object's movement into a single geometric representation by sampling an object's position at different time instances and connecting corresponding edges in two adjacent samples with a bilinear patch. We present example motion effects for various styles of speed lines, multiple stroboscopic images, temporal offsetting, and photorealistic and stylized blurring on both simple and production examples.

1 Introduction

A huge variety of animation techniques, ranging from keyframing to dynamic simulation to motion capture, populates the animation toolbox and can be used to create dynamic and compelling worlds full of action and life. During the rendering process, production-quality renderers employ motion blur as a form of temporal antialiasing. While motion blur effectively removes aliasing artifacts such as strobing, it does so at the cost of image clarity. Fast-moving objects may be blurred beyond recognition in order to properly remove high-frequency signal content. Although correct from a signal-processing point of view, this blurring may conflict with the animator's concept of how motion should be treated based on his or her creative vision for the scene. Since the appreciation of motion is a perceptual issue, the animator may wish to stylize its depiction in order to stimulate the brain in a certain manner, in analogy to the way an impressionist painter may stylize a painting in order to elicit some aesthetic response. To this end, comic book artists, whose entire medium of expression is based on summarizing action in still drawings, have pioneered a variety of techniques for depicting motion. Similar methods are employed in 2D hand-drawn animation to emphasize, accent, and exaggerate the motion of fast-moving objects, including speed lines, multiple stroboscopic copies, streaking, stretching, and stylized blurring. Although these effects have played an important role in traditional illustration and animation for the past century, computer-graphics animation cannot accommodate them in a general and flexible manner because production renderers are hard-wired to deliver realistic motion blur. As such, stylized motion effects are relegated to one-off treatment and infrequently used.

In our research, we experiment with motion effects as first-class entities within the rendering process. Rather than attempting to reproduce any particular style, we aim at creating a general-purpose rendering mechanism that can accommodate a variety of visualization styles, analogous to the way surface shaders can implement different surface appearances. Many effects from traditional mediums are directly related to an object's movement through a region of space and take the form of transient, ephemeral visuals left behind. This observation motivates a simple yet powerful change in the rendering process. We extend the concept of a surface shader, which is evaluated on an infinitesimal portion of an object's surface at one instant in time, to that of a programmable motion effect, which is evaluated with global knowledge about all portions of an object's surface that pass in front of a pixel during an arbitrary long sequence of time. With this added information, our programmable motion effects can decide to color pixels long after (or long before) an object has passed in front of them, enabling speed lines, stroboscopic copies, streaking, and stylized blurring (Figure 1). By rendering different portions of an object at different times, our effects also encompass stretching and bending. Other effects that extend beyond the object's position in space, such as clouds of dust

ACM Reference Format
Schmid, J., Sumner, R., Bowles, H., Gross, M. 2010. Programmable Motion Effects.
ACM Trans. Graph. 29, 4, Article 57 (July 2010), 9 pages. DOI = 10.1145/1778765.1778794
http://doi.acm.org/10.1145/1778765.1778794.

Copyright Notice
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 0730-0301/2010/07-ART57 $10.00 DOI 10.1145/1778765.1778794
http://doi.acm.org/10.1145/1778765.1778794

or smoke, flashes of color that fill the frame, and textual annotations (e.g., "BANG" or "POW") are not addressed by our framework. Traditional motion blur is a special case within our system, implemented as a motion effect program that averages the relevant surface contributions during a specified shutter time. In general, however, our method dissolves the classic notion of a scene-wide shutter time and allows each motion effect program to independently specify its operating time range, so that a single rendered frame may compose information from different periods of time.

From a technical standpoint, the most challenging aspect of our system is efficiently computing global information about an object's movement. We make this computation in the context of a ray tracer, where a single ray cast is modified to find every part of an object that has moved past a given pixel throughout an arbitrary long time range. Since the motion of an object may be complex, an analytical solution to the problem is not feasible. Instead, we construct a new geometric object called a *time aggregate object* (TAO) that aggregates an object's movement into a single geometric representation. This representation is augmented with additional information that enables the reconstruction of a set of points representing the path along the surface of the object that is visible to the pixel as the object moves through the scene, along with the associated times. Using this global information, one can develop shading algorithms that utilize information about an object's movement through an arbitrarily large window of time.

Our primary contribution is an approach to motion depiction for three-dimensional computer animation that fits naturally into current rendering paradigms and offers the same generality and flexibility as programmable surface shading. We also make the technical contribution of the TAO data structure and present several examples of programmable motion effects.

2 Background

James Cutting [2002] presents a detailed treatise about motion depiction in static images that examines parallels in art, science, and popular culture. He identifies five categories that encompass the vast majority of motion depiction techniques used to-date: photographic blur, speed lines, multiple stroboscopic images, shearing, and dynamic balance.

Of these categories, photographic blur has received by far the most attention within computer graphics, as an answer to temporal aliasing problems in computer animation. Since a rendered animation represents a sampled view of continuous movement, disturbing aliasing artifacts such as strobing can occur if the temporal signal is sampled naively [Potmesil and Chakravarty 1983]. Consequentially, temporal antialiasing is a core component of all production-quality renderers, and the research community has developed many sophisticated algorithms for this purpose. These methods can be seen as convolution with a low-pass filter to remove high-frequency details, yielding a blurry image. Or, in analogy to traditional photographic processes, they can be interpreted as opening a virtual shutter for a finite period of time during which fast-moving objects distribute their luminance over regions of the film, creating blurred motion.

Sung, Pearce, and Wang [2002] present a taxonomy of motion blur approaches, with associated references, and reformulated these published methods in a consistent mathematical framework. Two features of this framework help distinguish existing motion-blur algorithms from our presented work. First, existing methods can be interpreted as scene-wide integration over a fixed shutter time. In our work, we abandon the notion of a fixed shutter time and empower individual motion effect programs to determine the time range over which to operate. In this way, a single rendered image may combine information from a variety of different time ranges. This added flexibility incurs a more complicated compositing situation (Section 4.6) when multiple motion effects overlap in depth. Second, existing methods employ a spatio-temporal reconstruction filter responsible for averaging contributions from different times. In our method, we generalize this filtering concept to an arbitrary program that allows a variety of different looks to be expressed. The averaging operation used in traditional motion blur becomes a special case within this framework.

The value of stylized motion depiction is evident from its treatment in traditional artistic mediums, including the "Futurism" art movement of the 1900's [Hulten 1986] and techniques taught as tools-of-the-trade to comic book artists [McCloud 1993] and animators [Goldberg 2008; Whitaker and Halas 2002]. Perceptual studies even provide direct evidence that speed lines influence low-level motion processing in the human visual system [Burr and Ross 2002]. Due to the importance of stylized motion depiction, many researchers have explored ways to incorporate it into computer-generated imagery, animation, photographs, and video.

Masuch and colleagues [1999] describe the use of speed lines, stroboscopic copies, and arrows in 3D graphics as a post-processing operation, Lake and colleagues [2000] present a similar method for speed line generation, and Haller and colleagues [2004] present a system for generating these effects in computer games. These methods are specialized for the targeted effects and may not generalize easily to different visual styles. A framework for generating visual cues based on object motion is introduced by Nienhaus and Döllner [2005]. This method allows one to define rules for the depiction of certain events or sequences based on a scene graph representation of geometry and a behavior graph representation of animation. The authors do not address the issue of how to implement the depictions in a generalized fashion. Researchers also present algorithms to add stylized motion cues to 2D animation [Hsu and Lee 1994; Kawagishi et al. 2003] and video [Bennett and McMillan 2007; Collomosse et al. 2005], to filter motion in images [Liu et al. 2005] or 3D animation [Wang et al. 2006; Noble and Tang 2007; Chenney et al. 2002] in order to create a magnified or cartoony effect, to create stylized storyboards from video [Goldman et al. 2006], and to summarize the action in motion-capture data [Assa et al. 2005; Bouvier-Zappa et al. 2007] or sequences of photographs [Agarwala et al. 2004].

Taken all together, these methods cover the five categories of motion depiction techniques that Cutting [2002] proposes. However, existing algorithms target specific looks and must explicitly parameterize stylistic deviations (e.g., Haller and colleagues' system [2004] represents speed lines as connected linear segments with a parameter to control line thickness over time). The central thrust and distinguishing characteristic of our contribution is an open-ended system for authoring motion effects as part of the rendering process. We extend the concept of programmable surface shading [Hanrahan and Lawson 1990] to take the temporal domain into account for pixel coloring. While our motion effect programs define how this extra dimension should be treated for a certain effect, they can still call upon conventional surface shaders for the computation of surface luminance at a given instant in time. We show examples in four of the five categories proposed by Cutting (stylized blurring, speed lines, multiple stroboscopic images, and shearing). And, notably, within these categories, different styles can be achieved with the same flexibility afforded by programmable surface shaders.

3 Method Principles

Many stylized motion effects from traditional mediums summarize an object's movement over a continuous range of time with transient, ephemeral visuals that are left behind. Motivated by this observation, we propose an alternative rendering strategy that operates

on the scene configuration during an arbitrarily long time range T. In this section, we introduce the concept of motion effect programs, our *time aggregate object* data structure, and the renderer's compositing system. Section 4 discusses more specific implementation details.

3.1 Motion Effect Programs

In analogy to a state-of-the-art renderer that relies on surface shaders to determine the color contributions of visible objects to each pixel, we delegate the computation of a moving object's color contribution within the time range T to *motion effect programs*. A motion effect program needs to know which portions of all surfaces have been "seen" through a pixel during T. In general, this area is the intersection of the pyramid extending from the eye location through the pixel corners with the objects in the scene over time. Although Catmull [1978] presents an analytic solution to this pixel coverage problem for static scenes, extending it to the spatio-temporal domain is non-trivial. As such, we follow the approach of Korein and Badler [1983] and collapse the pyramid to a single line through the pixel center before computing analytic coverage. The surface area seen by the pixel for a particular object then becomes a line along the surface, which we call a *trace*. We address the spatial aliasing incurred by this simplification via supersampling in the spatial domain.

A motion effect program calculates a trace's contribution to the final pixel color. In doing so, it utilizes both positional information (the location of a trace on the object's surface) and temporal information (the time a given position was seen) associated with the trace. It can evaluate the object's surface shaders as needed and draw upon additional scene information, such as the object's mesh data (vertex positions, normals, texture coordinates, etc.), auxiliary textures, the camera view vector, and vertex velocity vectors.

3.2 Time Aggregate Objects

Computing a trace is a four-dimensional problem in space and time, where intersecting the 4D representation of a moving object with the plane defined by the view ray yields the exact trace. Unfortunately, since the influence of the underlying animation mechanics on an object's geometry can be arbitrarily complex, a closed-form analytic solution is infeasible. Monte Carlo sampling [Cook et al. 1984] could be considered as an alternative, since it is used by prominent production renderers [Sung et al. 2002] to produce high-quality motion blur. However, it is also not effective in the present scenario since the time period associated with a trace may be very short in comparison to the time range over which the motion effect is active. For example, a ball may shoot past a pixel in a fraction of a second but leave a trailing effect that persists for several seconds. A huge number of samples distributed in time would be required in order to effectively sample the short moment during which the ball passes.

Consequentially, we propose a new geometric data structure that allows our system to reconstruct a linear approximation of the full trace from a single ray cast. Our data structure is inspired by the 4D polyhedra used in Grant's temporal anti-aliasing method [1985] and aggregates an object's geometry sampled at a set of times t_i (Figure 2) into a single geometric primitive. In addition, corresponding edges of adjacent samples are connected by a bilinear patch, which is the surface ruled by the edge as its vertices are interpolated linearly between t_i and t_{i+1}. We call the union of the sampled object geometry and swept edges a *time aggregate object* (TAO).

The intersection of a view ray with a bilinear patch of the TAO represents a time and location where the ray, and thus also a trace, has crossed an edge of the mesh. By computing all such intersections (not just the closest one) and connecting the associated edge crossings with line segments, we obtain a linear approximation of the trace. Intersections with the sampled geometry represent additional time and space coordinates, which improve the accuracy of the trace approximation. The accuracy of the approximated traces thus depends on the number of TAO intersections, which scales with both the geometric complexity of the object's geometry and the number of samples used for aggregation. In practice, we can use finely sampled TAOs without a prohibitive computation time, since the complexity of ray-intersection tests scales sub-linearly with the number of primitives when appropriate spatial acceleration structures are used [Fóris et al. 1996].

3.3 Compositing

A motion effect program acts on a trace as a whole, which may span a range of depths and times. If objects and contributions from their associated programmable motion effects are well separated in depth, our renderer can composite each effect's total contribution to the image independently according to depth ordering. The compositing algorithm used can be defined by the effect itself, drawing upon standard techniques described by Porter and Duff [1984]. Compositing becomes more complex when multiple traces overlap, since there may be no unequivocal ordering in depth or time. Additionally, different motion effect programs may operate over different, but overlapping, time domains since a single scene-wide shutter time is not enforced.

We resolve this ambiguity by introducing additional structure to the way in which motion effects operate. All traces are resampled at a fixed scene-wide resolution. Each motion effect program processes its traces' samples individually, and outputs a color and coverage value if that sample should contribute luminance to the rendered pixel. Our compositing system processes the output samples in front-to-back order, accumulating pixel color according to the coverage values until 100% coverage is reached.

4 Implementation

We implement our method as a plug-in to Autodesk Maya 2010 [Aut 2010]. Our system is divided into two parts: a module to create and encapsulate TAOs from animated 3D objects, and a new rendering engine, which generates images with motion effects. The rendering engine computes a color for each pixel independently by computing all intersections of the pixel's view ray with the TAOs, and connecting them to form a set of traces. Then it calls the motion effect programs for each object, which compute that object's and effect's contribution to the pixel color using the object's traces. Finally, a compositing step computes the resulting pixel color.

4.1 TAO Creation

We implement the TAO concept as a custom data structure within Maya. We assume that animated objects are represented by triangle meshes with static topology. Our system builds a TAO by sampling an object's per-vertex time dependent data (e.g., vertex positions, normals, and texture coordinates) at a set of times t_i, aggregating those samples into a single geometric primitive, and connecting adjacent edges with bilinear patches (Figure 2). For a mesh with N_E edges and N_F triangles, N_t object samples yield a TAO with $N_t \cdot N_F$ triangles and $(N_t - 1) \cdot N_E$ bilinear patches.

The density and placement of the object sample times t_i determine how well the motion of an object is approximated by our TAO data structure. Since the approximation is linear by nature, it perfectly captures linear motion. For rotation and non-rigid deformation, however, proper sampling of the motion is necessary

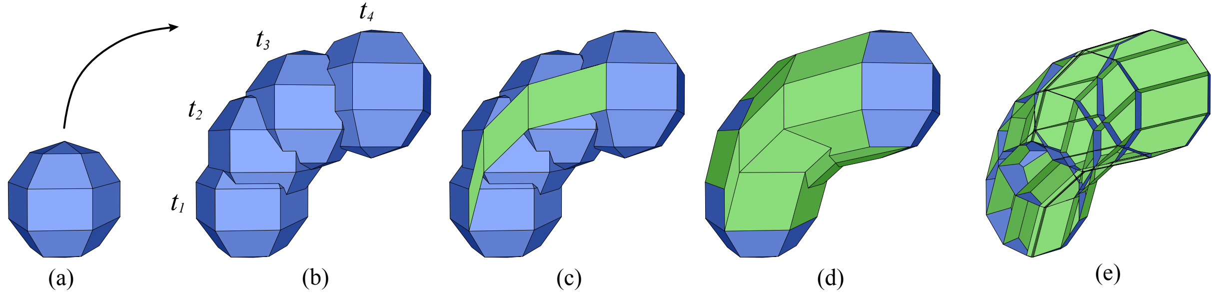

Figure 2: *The time aggregate object (TAO) data structure encodes the motion of an object (a) using copies of the object sampled at different times $t_1 \ldots t_4$ (b) and bilinear patches that connect corresponding edges in adjacent samples edges. These patches are shown in (c) for one edge and in (d) for the whole mesh. This data structure is not just the convex hull of the moving object, but has a complex inner structure, as seen in the cutaway image (e).*

Figure 3: *Undersampled (left) versus properly sampled (right) motion. The trace sampling rate is 10 times larger in the right image.*

(Figure 3). Our systems supports both a uniform sampling and an adaptive sampling strategy that starts with a uniform temporal sampling and repeatedly inserts or deletes sample positions based on the maximum non-linearity α_i between the vertex positions of adjacent samples:

$$\alpha_i = \max_j angle(\mathbf{v}_j(t_{i-1}), \mathbf{v}_j(t_i), \mathbf{v}_j(t_{i+1})), \tag{1}$$

where $angle(A, B, C)$ is the angle between the line segments AB and BC and $\mathbf{v}_j(t_i)$ is the position of vertex j in object sample i. First, the adaptive sampling strategy iteratively removes sample times whenever α_i is smaller than a given coarsening threshold. In a second step, it insert two new samples at $\frac{t_i - t_{i-1}}{2}$ and $\frac{t_{i+1} - t_i}{2}$ if α_i exceeds a refinement threshold. In order to keep the maximum number of object samples under control, the refinement criterion is applied iteratively on the sample with the largest α_i until the maximum number of samples is reached. In practice, we assume a certain amount of coherence in the motion of nearby vertices, and our system approximates the optimal sample placement by considering only a subset of the mesh vertices in Equation 1.

4.2 TAO Intersection

Our renderer generates a view ray for each pixel according to the camera transform. Each ray is intersected with the primitives of the TAO (mesh faces and bilinear patches) to compute all intersections along the ray, not just the one which is closest to the camera. Normals, texture coordinates, and other surface properties are interpolated linearly over the primitives to the intersection point and stored.

For the intersection with bilinear patches, we use the algorithm described by Ramsey and colleagues [2004]. If an edge connects $\mathbf{v}_j$ and $\mathbf{v}_k$, then for each $0 < i < N_i - 1$ a bilinear patch is defined as

$$\begin{aligned} \mathbf{p}_{ijk}(r,s) = (1-r)(1-s)\cdot\mathbf{v}_j(t_i) + (1-r)\cdot s\cdot\mathbf{v}_k(t_i) \\ + r\cdot(1-s)\cdot\mathbf{v}_j(t_{i+1}) + r\cdot s\cdot\mathbf{v}_k(t_{i+1}). \end{aligned} \tag{2}$$

Solving for a ray intersection yields a set of patch parameters (r, s) that corresponds to the intersection point. With parameter r, we can compute the time at which the ray has crossed the edge:

$$t = t_i + r\cdot(t_{i+1} - t_i). \tag{3}$$

The parameter s represents the position along the edge at which the crossing has happened. It can be used to interpolate surface properties stored at the vertices to the intersection point, and to compute the position of the intersection point in a reference configuration of the object. Mesh face primitives in the TAO are intersected according to standard intersection algorithms. A parametric representation of the intersection points with respect to the primitives is necessary for interpolating the surface properties.

To accelerate the intersection computation, we partition the image plane into tiles and create a list of all primitives for which the bounding box intersects a given tile. Each view ray only needs to be intersected with the primitive list of the tile containing the ray.

4.3 Trace Generation

The set of all intersection points of a ray with the TAO can be used to reconstruct the locus of points traced by the ray on the moving object. Consider the interpretation of the ray-TAO intersection points on the input mesh. As an individual triangle passes fully by the ray, the ray crosses the edges of the triangle an even number of times, assuming that the ray was not intersecting at the beginning or at the end of the observed time. Each edge crossing corresponds to an intersection of the ray with a bilinear patch of the TAO, and vice versa. Since our system is restricted to a piecewise linear approximation of motion, we assume that the trace forms a linear segment in between two edge crossings. The task of constructing the trace is thus equivalent to connecting the intersection points in the proper order.

To facilitate this discussion, we refer to the volume covered by a triangle swept between two object samples within the TAO as a Time Volume Element (TVE). This volume is delimited by two corresponding triangles that belong to adjacent object samples and the three bilinear patches formed by the triangle's swept edges (Figure 4). Since there is an unambiguous notion of inside and outside, a ray originating outside of a TVE will always intersect the TVE an even number of times, unless it exactly hits the TVE's boundary. A pair of consecutive entry and exit points corresponds to a segment of the trace on the triangle. Therefore, by sorting all ray-TVE intersection points according to their distance from the viewer and pairing them sequentially, we can construct all trace segments that

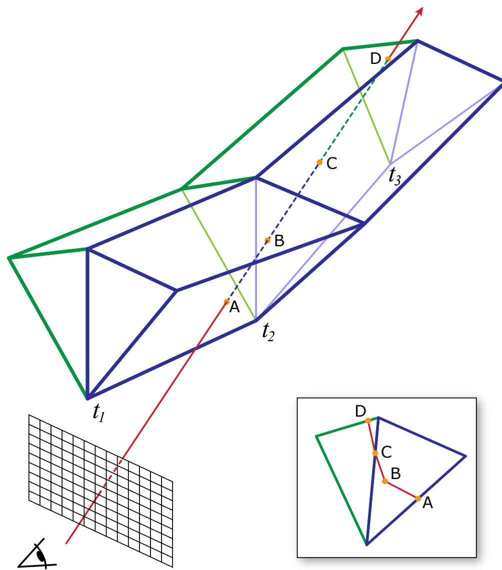

Figure 4: *This figure shows two triangles at 3 consecutive object samples, t_1, t_2 and t_3, which results in four Time Volume Elements (TVEs). A ray intersects the TVEs 4 times. At intersection A, the ray enters the blue triangle though an edge. Point B is an intersection with a sampled mesh triangle, indicating that the ray moves from one time sample to the next. In C, the ray leaves the blue and enters the green triangle, which it finally exits at D. The reconstructed trace visualized on the two triangles is shown in the inset figure.*

cross the corresponding triangle within the time range spanned by that individual TVE.

Next, we consider trace connectivity. TVEs that share a common TAO primitive (triangle or bilinear patch) are adjacent in space. If the shared primitive is a triangle, the TVEs were formed by the same triangle of the input mesh in adjacent object samples. If it is a bilinear patch, the triangles that formed the TVEs are adjacent to one another on the input mesh, separated by the edge that formed the patch. This combined adjacency information determines the connectivity of trace segments associated with neighboring TVEs. Our system processes TVEs in turn to reconstruct the individual trace segments, and then uses this connectivity information to connect them together into different connected components.

4.4 Trace Resampling

At this point, we have the traces that the current pixel's view ray leaves on moving objects in the form of connected sequences of intersections with the TAO. Each intersection point corresponds to an animation time and a position on an object, and, in conjunction with the trace segments that connect to it, we can determine and interpolate surface properties such as attached shaders, normals, and texture coordinates.

In order to facilitate compositing, our system imposes a consistent temporal sampling on all traces by dividing them into individual trace fragments of fixed spacing in time. This resampling should be dense enough so that depth conflicts among different traces can be resolved adequately. The visibility of individual trace fragments during each sampling interval is also determined at this stage. Our system does not delete occluded or back facing trace fragments, however. Instead, it is left up to the motion effect program to decide whether obstructed segments should contribute to pixel coverage.

4.5 Motion Effect Programming

A motion effect program operates on the resampled representation of a trace. It is called once per object and processes all trace fragments associated with that object. The motion effect program has two options when processing an individual trace fragment. It can simply discard the fragment, in which case no contribution to the final pixel color will be made. Or, it can assign a color, depth, and pixel coverage value and output the fragment for compositing. The coverage value determines the amount of influence a fragment has on the final pixel color in the compositing step. When making this decision, the motion effect program can query an object's surface shader, evaluate auxiliary textures (e.g., noise textures, painted texture maps, etc.), or use interpolated object information.

The effect program can also subdivide the trace even further if the effect requires a denser sampling of the object's surface. For example, further subdivision could be the needed if the effect must integrate a surface texture of high frequency. Decreasing the trace resampling distance (Section 4.4) has a similar effect, although it affects all motion effects in the scene.

4.6 Compositing

By emitting trace fragments, each motion effect program specifies its desired contribution to the pixel color. The compositing engine combines all of the emitted fragments to determine the final pixel color by processing fragments in depth order (front to back) using a clamped additive model based on the coverage values. The coverage values of the fragments, which range between 0 and 1, are accumulated until a full coverage of 1 is reached or until all fragments have been processed. If the coverage value exceeds 1, the last processed fragment's coverage is adjusted so that the limit is reached exactly. The final pixel color is computed by summing the processed fragment colors weighted by the corresponding coverage values. The accumulated coverage value is used as the alpha value for the final pixel color.

5 Results

In this section, we show a number of results obtained with our programmable motion effect renderer and describe how the effect programs were set up to achieve these results.

5.1 Motion Effects

To better explain the mechanism of programmable motion effects, we supply pseudo-code algorithms for five basic effects used in our examples. These algorithms show the framework of each effect using a common notation. A trace segment *ts* consists of two end points, denoted *ts.left* and *ts.right*. Each of the end points carries information about the animation time at which this surface location has been intersected by the view ray and linearly interpolated surface properties. They may be interpolated further with the INTERPOLATE function, which takes a trace segment and a time at which it should be interpolated as parameters. By *ts.center*, we designate the interpolation of the end points' properties to the center of the segment. The SHADE function evaluates the object's surface shader with the given surface parameters and returns its color. The animation time of the current frame being rendered is denoted as "current time." The output of a motion effect program is a number of fragments, each with a color and coverage value, that represent the effect's contribution to the pixel color. Passing a fragment to the compositing engine is designated by the keyword **emit**.

Instant Render The most basic program renders an object at a single instant in time. The program loops through all trace seg-

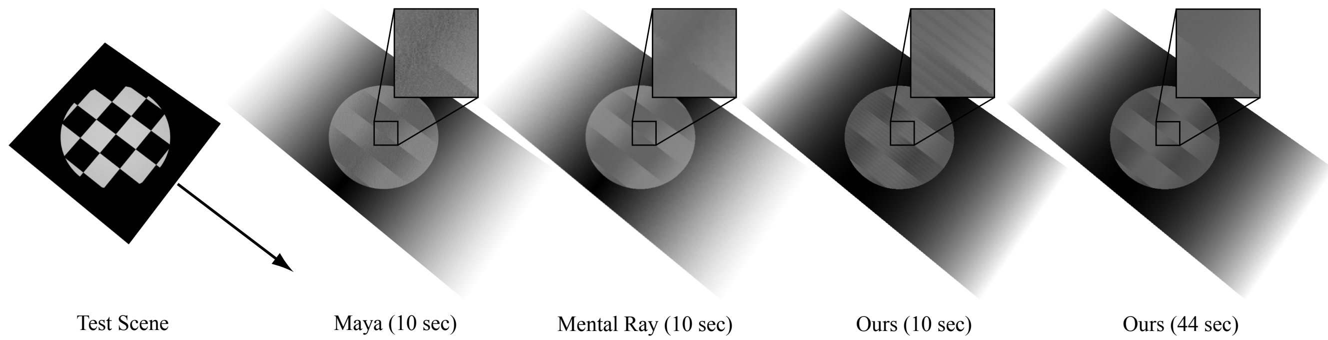

Figure 5: *A comparison of our motion blur implementation with production renderers. The test scene, rendered without blur, consists of a translating checkerboard square with a stationary spotlight. The three middle images were created with equal render time. The rightmost image shows a high-quality render obtained with our system with a denser trace sampling.*

ments and checks whether the desired time lies within a segment. If so, it interpolates the surface parameters to that point, evaluates the surface shader accordingly, and emits just one fragment to the compositing stage with a coverage value of 1. Any surfaces behind that fragment will be hidden, as is expected from an opaque surface.

Algorithm 1 INSTANT RENDER(*time*)

```
for all trace segments ts do
  if ts.left.time <= time < ts.right.time then
    eval_at := INTERPOLATE(ts, time)
    color := SHADE(eval_at)
    coverage := 1
    emit fragment(color, coverage)
```

Weighted Motion Blur Photorealistic motion blur integrates the surface luminance within a given shutter time. We extend this process with an arbitrary weighting function $w(t)$ that can be used both for realistic and stylized blurring. For photorealistic blur, the weighting function is the temporal sampling reconstruction filter function. A flat curve corresponds to the commonly used box filter. In general, the weighting function need not be normalized, which means that luminance energy is not necessarily preserved. This flexibility increases the possibilities for artistic stylization. For example, a flat curve with a spike at $t = 0$ results in motion blur that has a clearly defined image of the object at the current frame time.

Algorithm 2 MOTION BLUR($w(t)$)

```
for all trace segments ts do
  if ts time interval overlaps with {t | w(t) > 0} then
    clip ts against {t | w(t) > 0}
    color := SHADE(ts.center)
    δt := ts.right.time - ts.left.time
    coverage := δt · w(ts.center.time)
    emit fragment(color, coverage)
```

The effect program in Algorithm 2 describes weighted motion blur in its most basic form. A more advanced implementation could take more than one surface shading sample within a segment and use a higher order method for integrating the luminance. With the given effect program, however, the fidelity of the image can be improved by increasing the global trace sampling rate (Section 4.4). A comparison of motion blur from our implementation with that from production renderers is shown in Figure 5.

Speed Lines Speed lines are produced by seed points on an object that leave streaks in the space through which they travel. They can be distributed automatically or placed manually. The program computes the shortest distance between trace segments and seed points. If the distance is smaller than a threshold, the seed point has passed under or close to the pixel, and the pixel should be shaded accordingly. A falloff function based on distance or time can be used to give the speed line a soft edge.

Algorithm 3 SPEED LINES(*seed vertices*, *width*, *length*)

```
for all trace segments ts do
  for all seed vertices v do
    if DISTANCE(ts, v) < width
    and current time − ts.center.time < length then
      color := SHADE(ts.center)
      compute coverage using a falloff function
      emit fragment(color, coverage)
```

Stroboscopic Images This effect places multiple instant renders of the object at previous locations. It imitates the appearance of a moving object photographed with stroboscopic light. The intensity of the stroboscopic images is attenuated over time to make them appear as if they are washing away. We keep the locations of the stroboscopic images fixed throughout the animation, but they could also be made to move along with the object. The falloff function can be composed of factors considering the time of the stroboscopic image, geometric properties of the mesh (e.g., the angle between the motion vector and the normal at a given point), or an auxiliary modulation texture to shape the appearance of the stroboscopic images.

Algorithm 4 STROBOSCOPIC IMAGES(*spacing*, *length*)

```
for all trace segments ts do
  if ts.left.time < current time − length then
    t1_mod := ts.left.time modulo spacing
    t2_mod := ts.right.time modulo spacing
    if t1_mod <= 0 < t2_mod then
      color := SHADE(ts.center)
      compute coverage using a falloff function
      emit fragment(color, coverage)
```

Algorithm 5 TIME SHIFT(*ts*, *shift magnitude*)

```
time shift := |ts.center.v_motion| · shift magnitude
shift time values in trace segment ts according to time shift
return ts
```

Time Shifting The time shifting program modulates the instantaneous time selected from the trace. We use it in conjunction with

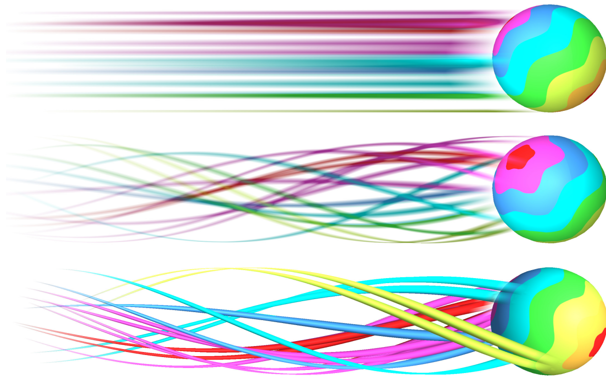

Figure 6: *Different speed-line styles.*

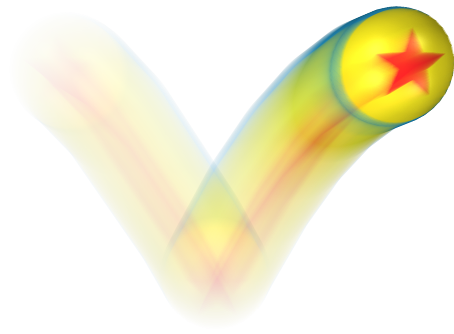

Figure 7: *A bouncing ball rendered with non-uniform blur.*

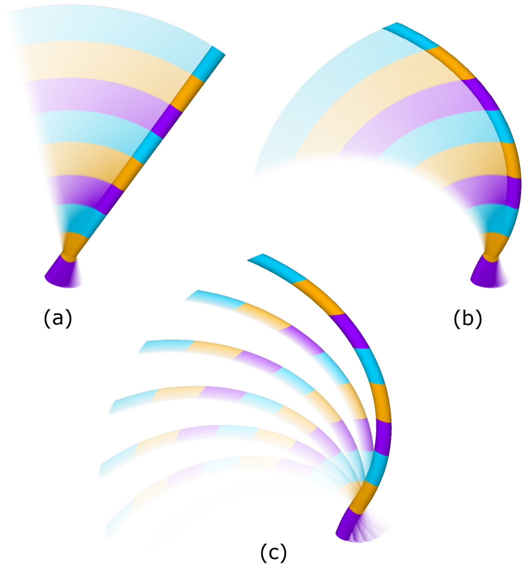

Figure 8: *A spinning rod showing (a) weighted motion blur, (b) weighted motion blur and time shift, and (c) stroboscopic images with time shift.*

the INSTANT RENDER so that each pixel in the image may represent a different moment in time. If fast-moving parts of an object are shifted back in time proportional to the magnitude of their motion, these parts appear to be lagging behind. They "catch up" with the rest of the object when it comes to a stop or changes direction. This effect is only visible if an object's motion is not uniform across the surface, as in the case of rotation. In this algorithm, v_{motion} designates the motion vector of the corresponding part of the surface.

5.2 Examples

Translating and Spinning Ball The results in Figure 6 demonstrate speed lines in different visual styles. The first two images use the SPEED LINES effect (Algorithm 3) with a falloff function that fades in linearly at both ends of the speed lines and modulates the width of the speed line over time. For the last image, the distance of the seed point to the current pixel is used to manipulate the normal passed to the surface shader, giving a tubular appearance.

Bouncing Ball The example in Figure 7 shows a bouncing toy ball, rendered with a modified version of the MOTION BLUR effect (Algorithm 2). The effect program computes a reference time for the input trace, and uses the difference between this reference time and the current time to determine the amount of blur, or the shutter opening time in conventional terms. As a result, the blur increases toward the end of the trail.

Spinning Rod This result shows a rod spinning about an axis near its lower end. Figure 8 (a) combines a MOTION BLUR effect (Algorithm 2) that uses a slightly ramped weighting function with an INSTANT RENDER effect (Algorithm 1) to render a crisp copy of the rod. Figure 8 (b) additionally uses the TIME SHIFT function (Algorithm 5) with a negative shift magnitude so that quickly moving surface parts lag behind. As shown in the accompanying video, these parts "catch up" when the rod stops moving or changes direction. In Figure 8 (c), MOTION BLUR is replaced with a STROBOSCOPIC IMAGES effect (Algorithm 4). The falloff function fades the stroboscopic images out as time passes and additionally uses the angle between the motion vector and the surface normal to make the rod fade toward its trailing edge.

Toy UFO In Figure 9, we have attached speed lines to a UFO model to accentuate and increase the sensation of its speed. To keep the effect convincing when the camera is moving with the UFO, the length and opacity of the speedlines are animated over time. An animated noise texture is sampled using the texture coordinates from the seed point of each speed line, which gives the attenuation value for that speed line.

Pinocchio's Peckish Pest The last set of examples show our results on a production-quality scene. Figure 10 uses a combination of speed lines and weighted motion blur. The motion blur's weighting curve has a spike around the current time, so that Pinocchio is shown clearly in every frame, even when he is moving quickly. In Figure 11 a comparison between conventional motion blur (also rendered by our system) and our weighted motion blur is made.

Statistics about all examples are shown in Table 1. The values are taken from a representative frame for each animation, where the full TAO is within the visible screen area. Render time depends greatly on the motion and the amount of screen space covered by the objects. The numbers shown resulted from rendering an image with 1280x720 pixels on an 8-core 2.8 GHz machine.

6 Conclusion

In this paper, we have presented a novel approach to depict motion in computer-generated animation. Our method fits naturally into current rendering paradigms and offers the same generality and flexibility as programmable surface shading. Our results demonstrate that it is a powerful platform for experimenting with different depiction styles.

Limitations of our work motivate a number of rich future research possibilities. In our current implementation, the screen-space re-

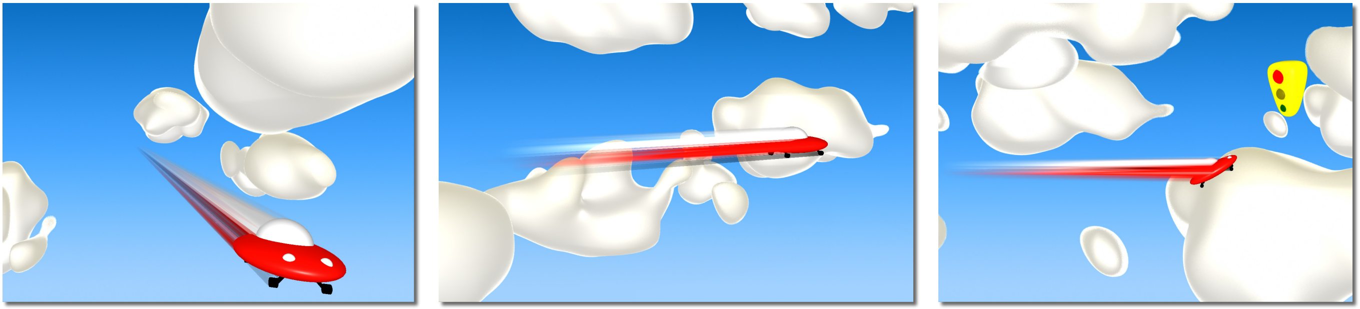

Figure 9: *Speed lines applied to a flying UFO animation.*

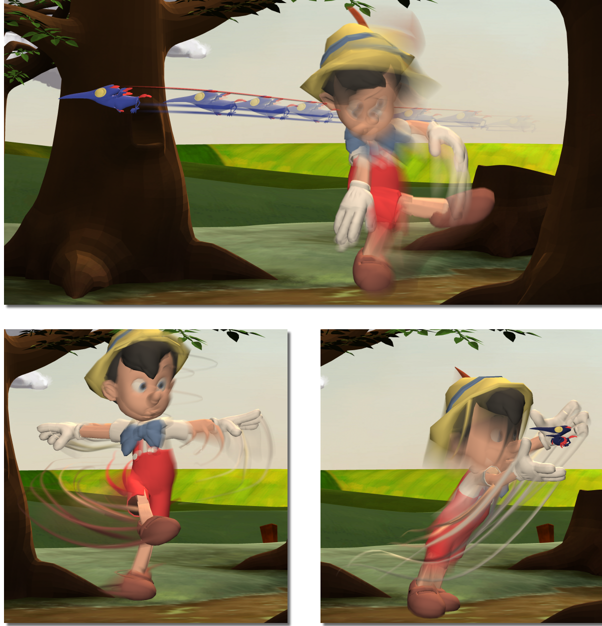

Figure 10: *Results from applying speed line and motion blur effects (bottom), and the stroboscopic image effect (top) to an animation of Pinocchio and a woodpecker.*

Figure 11: *A comparison of conventional motion blur (left) with our weighted motion blur including speed lines (right).*

gion of the motion effect is limited to the convex hull of the object's motion, since the motion effect programs operate only on pixels that intersect the TAO. One future research avenue would be to consider installing the motion depiction framework further upstream in the rendering process as a geometry shader, allowing new geometry to be created and expanding the range of possible depiction styles.

We experiment with the core idea of programmable motion effects and provide a proof-of-concept that they can be used to express a variety of motion depiction styles. However, our renderer im-

Example	# Tri	TAO	Trace	Mem	Time
Spinning Ball	1200	160	0.1	12	2.6
Rod	80	100	0.1	0.5	0.25
UFO (shot 2)	3400	20	0.5	4.5	2
Pinocchio (shot 2)	37k	60	0.005	130	4.3
Pinocchio (shot 3)	122k	60	0.005	300	65

Table 1: *Example statistics for a representative frame. # Tris: number of source mesh triangles; TAO: number of object samples (Section 4.1); Trace: trace sampling distance (Section 4.4); Mem: memory required to store the TAO in MiB; Time: render time in minutes.*

plements only the most basic functionality and does not consider global illumination, reflection, refraction, caustics, participating media, or other effects that are standard in production renderers. One immediate avenue of future work is applying the core principle of recursive ray tracing to our framework by casting secondary rays for shadowing, reflection, and refraction. Naturally, computation time may become an issue if many secondary rays are used.

The performance of our system is reasonable for the examples shown, with most frames requiring only a few minutes to render. The slowest aspect of the system lies in the way it interfaces with Maya. Creating the TAO structure requires sampling the animation system at tens or hundreds of sample positions to cache time varying mesh data. Additionally, many of our motion effect programs evaluate an object's surface shader or other auxiliary textures. If the parameters of these shaders and textures are themselves animated, then each and every evaluation must call back to the animation software in order to work in the proper temporal context. Due to Maya's system design and assumptions about the distinct separation of animation and rendering, such out-of-context shader network evaluations are prohibitively expensive. This observation speaks to a future animation and rendering design that does not draw a hard line between the two, but rather couples both as tightly as possible.

Our current TAO data structure requires the mesh connectivity to be constant throughout the animation. This limitation prohibits the use of our system in scenarios where the connectivity changes, such as in the presence of level of detail or certain physical simulation techniques. A related issue comes with the requirement for the motion to be sampled consistently within one TAO, even if some parts of the input object move with a different complexity than other parts. Sampling each primitive's motion at its optimal rate would improve performance and flexibility.

The implementation presented in this paper uses a ray tracing approach for rendering, but we believe that the general concept can be adapted to other rendering paradigms. It would be interesting to investigate how programmable motion effects can be implemented on GPUs and in the Reyes architecture [Cook et al. 1987]. An al-

ternative approach would be to replace the TAO data structure with an image sequence that stores additional per pixel data such as face correspondence and surface parameters. Motion effect programs could combine a number of these images to create one frame. This approach can be seen as an extension of deferred shading for motion depiction.

Acknowledgements

We would like to thank Martin Senn, Dawn Rivera-Ernester, Andy Hendrickson, and the WDAS summer interns for their help with the Pinocchio animation. We are also grateful to David Spuhler and Matthias Bühlmann for their work on the project, and to Mark Pauly, Robert Cook, Wojciech Jarosz, Jovan Popović, Frédo Durand, and William Freeman for discussions about the project. Figures 3, 7, 10, 11 © Disney Enterprises.

References

AGARWALA, A., DONTCHEVA, M., AGRAWALA, M., DRUCKER, S., COLBURN, A., CURLESS, B., SALESIN, D., AND COHEN, M. 2004. Interactive digital photomontage. *ACM Trans. Graph. 23*, 3.

ASSA, J., CASPI, Y., AND COHEN-OR, D. 2005. Action synopsis: pose selection and illustration. *ACM Trans. Graph. 24*, 3.

AUTODESK. 2010. *Autodesk Maya 2010*. http://www.autodesk.com/maya.

BENNETT, E. P., AND MCMILLAN, L. 2007. Computational time-lapse video. *ACM Trans. Graph. 26*, 3.

BOUVIER-ZAPPA, S., OSTROMOUKHOV, V., AND POULIN, P. 2007. Motion cues for illustration of skeletal motion capture data. In *Non-Photorealistic Animation and Rendering 2007*.

BURR, D. C., AND ROSS, J. 2002. Direct evidence that "speedlines" influence motion mechanisms. *Journal of Neuroscience 22*, 19.

CATMULL, E. 1978. A hidden-surface algorithm with anti-aliasing. *SIGGRAPH Comput. Graph. 12*, 3.

CHENNEY, S., PINGEL, M., IVERSON, R., AND SZYMANSKI, M. 2002. Simulating cartoon style animation. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*.

COLLOMOSSE, J. P., ROWNTREE, D., AND HALL, P. M. 2005. Rendering cartoon-style motion cues in post-production video. *Graph. Models 67*, 6.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21*, 4.

CUTTING, J. E. 2002. Representing motion in a static image: constraints and parallels in art, science, and popular culture. *Perception 31*, 10.

FÓRIS, T., MÁRTON, G., AND SZIRMAY-KALOS, L. 1996. Ray shooting in logarithmic time. In *WSCG'96 - 4-th International Conference in Central Europe on Computer Graphics and Visualization'96*.

GOLDBERG, E. 2008. *Character Animation Crash Course!* Silman-James Press.

GOLDMAN, D. B., CURLESS, B., SALESIN, D., AND SEITZ, S. M. 2006. Schematic storyboarding for video visualization and editing. *ACM Trans. Graph. 25*, 3.

GRANT, C. W. 1985. Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. *SIGGRAPH Comput. Graph. 19*, 3.

HALLER, M., HANL, C., AND DIEPHUIS, J. 2004. Non-photorealistic rendering techniques for motion in computer games. *Comput. Entertain. 2*, 4.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. *SIGGRAPH Comput. Graph. 24*, 4.

HSU, S. C., AND LEE, I. H. H. 1994. Drawing and animation using skeletal strokes. *Proceedings of SIGGRAPH 94*.

HULTEN, P., Ed. 1986. *Futurism & Futurisms*. Abbeville Press.

KAWAGISHI, Y., HATSUYAMA, K., AND KONDO, K. 2003. Cartoon blur: Non-photorealistic motion blur. *Computer Graphics International Conference*.

KOREIN, J., AND BADLER, N. 1983. Temporal anti-aliasing in computer generated animation. *SIGGRAPH Comput. Graph. 17*, 3.

LAKE, A., MARSHALL, C., HARRIS, M., AND BLACKSTEIN, M. 2000. Stylized rendering techniques for scalable real-time 3d animation. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*.

LIU, C., TORRALBA, A., FREEMAN, W. T., DURAND, F., AND ADELSON, E. H. 2005. Motion magnification. *ACM Trans. Graph. 24*, 3.

MASUCH, M., SCHLECHTWEG, S., AND SCHULZ, R. 1999. Speedlines: depicting motion in motionless pictures. In *ACM SIGGRAPH 99 Conference abstracts and applications*.

MCCLOUD, S. 1993. *Understanding Comics*. Kitchen Sink Press.

NIENHAUS, M., AND DÖLLNER, J. 2005. Depicting dynamics using principles of visual art and narrations. *IEEE Computer Graphics and Applications 25*, 3.

NOBLE, P., AND TANG, W. 2007. Automatic expressive deformations for implying and stylizing motion. *The Visual Computer 23*, 7.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. *SIGGRAPH Comput. Graph. 18*, 3.

POTMESIL, M., AND CHAKRAVARTY, I. 1983. Modeling motion blur in computer-generated images. *SIGGRAPH Comput. Graph. 17*, 3.

RAMSEY, S. D., POTTER, K., AND HANSEN, C. 2004. Ray bilinear patch intersections. *Journal of Graphics, GPU, and Game Tools 9*, 3.

SUNG, K., PEARCE, A., AND WANG, C. 2002. Spatial-temporal antialiasing. *IEEE Transactions on Visualization and Computer Graphics 8*, 2.

WANG, J., DRUCKER, S. M., AGRAWALA, M., AND COHEN, M. F. 2006. The cartoon animation filter. *ACM Trans. Graph. 25*, 3.

WHITAKER, H., AND HALAS, J. 2002. *Timing for Animation*. Focal Press.