Interactive Real-Time Motion Blur

Matthias M. Wloka and Robert C. Zeleznik*
Computer Science Department
Brown University
Providence, RI 02912

Motion blurring fast-moving objects is highly desirable for virtual environments and 3D user interfaces. However, all currently known algorithms to generate motion blur are too slow for inclusion in interactive 3D applications.

We introduce a new motion blur algorithm that works in 3D on a per-object basis. The algorithm operates in real-time even for complex objects consisting of several thousand polygons. While it only approximates true motion blur, the generated results are smooth and visually consistent. We achieve this performance break-through by taking advantage of hardware-assisted rendering of semi-transparent polygons, a feature commonly available in today's workstations.

Keywords: motion blur, real-time, approximation, transparency, interaction, virtual reality

1 INTRODUCTION

Creators of virtual reality applications trade computation complexity against computation speed, trying to make the virtual environment look and act as real as possible while maintaining the crucial high frame rates. Currently, virtual reality applications do not integrate motion blur, since its computation cost is prohibitively high. We introduce a real-time motion blur approximation algorithm suitable for virtual reality applications.

1.1 Definition of Motion Blur

Insert Photograph 1

Figure 1: Motion blur is common in photography: fast moving objects are rendered as a semi-transparent blur, as illustrated in this photograph of a speeding car.

The term *motion blur* describes an effect caused by finite camera shutter speeds. If an object moves relatively quickly compared to the shutter speed, so that its image on the film moves while the shutter is open, then the object appears as a semi-transparent blur in the final image (see Figure 1). Decreasing the speed of the object or increasing the camera's shutter speed reduces the blurring, possibly to the point of creating a crisp, seemingly static image.

*This work was supported in part by an IBM fellowship for the first author, by the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, by ONR Contract N00014-91-J-4052, ARPA Order 8225, and also by IBM, NCR, Sun Microsystems, Hewlett Packard, Digital Equipment Corporation, and NASA.

Motion blur is common in photography and motion pictures. For example, still photographers deliberately use motion blur as a technique to express dynamic motion [1]. Motion picture cameras, due to their relatively slow shutter speeds, always record fast-moving objects as a blur on the individual frames of film that make up a movie.

1.2 Benefits of Motion Blur

Movies and interactive computer graphics, including virtual reality, rely on the viewer's ability to perceive smooth, continuous motion when presented with a rapid succession of individual frames. (A frame rate of 15 frames per second usually suffices.) While viewing such a presentation, human beings perceive motion blur as natural. In contrast, viewers perceive the apparent motion of crisp, unblurred objects as jerky and stilted — an effect observable in early stop-motion special effects [2]. This strobing effect is more pronounced the lower the frame rate.

Unfortunately, correctly simulating motion blur in 3D computer graphics is hard [3]. However, viewers find even crude approximations of motion blur convincing, an acceptance exploited by creators of animated cartoons and computer generated special effects [4].

Besides looking more natural, motion blur also establishes visual coherence between temporally disparate renderings of the same object. Until recently, 2D and 3D user interfaces disregarded such visual coherency, while commonly employing fast-moving ("popping" or "iconifying") objects. The resulting interface is initially confusing, because without coherency clues the user cannot easily synthesize what happens.

Work in 3D user interfaces [5] argues to animate the user interface, thereby shifting the user's cognitive load to the human perceptual system. Animation is adequate for relatively slow-moving objects. More recent work in 2D [6] demonstrates the usefulness of adding motion blur to the animated user interface.

Another potential benefit of motion blur is in the interaction with moving objects in virtual environments. The faster an object moves, the harder it is to select via the common point and click metaphors. This difficulty originates in the user's inability to intercept the object, in turn caused by an inability to predict future object positions and not being able to maneuver the pointing device accurately and quickly enough towards the predicted position.

Blurring fast-moving objects simplifies the interception task considerably. Because blurring minimizes the strobing effect and adds trails to the object, the user is better able to predict its motion path and velocity changes. Therefore, the user can assess future positions of the object more accurately. Also, a blurred object occupies a greater screen area than an unblurred object, thereby increasing the user's chances of pointing at the object.

Thus, virtual reality applications benefit greatly from the incorporation of motion blur: the 3D dynamic virtual environments look more natural to the participants, the inherently 3D user interfaces are rendered less confusing, and interacting with fast moving objects becomes easier. However, previous motion blur algorithms are too slow to maintain the required minimum frame rate of 15 frames per second.

1.3 Overview

Insert Photograph 2

Figure 2: This model of an airplane consists of 2020 triangles. We use it to test our algorithm.

Interactive applications generate more than ten frames each second, thus, each individual frame is on-screen for 0.1 seconds or less. In this time-span, a viewer is unable to distinguish correct from approximated motion blur [4]. We take advantage of this inability and approximate motion blur with a simple and fast algorithm that operates in object space. The algorithm assumes only the existence of hardware-assisted rendering of transparent polygons. The algorithm adds little computation overhead compared to polygon rendering

Insert Photograph 3

Figure 3: Our algorithm, while only approximating the correct solution, produces smooth, natural-looking motion blur in real-time, even for complex objects such as the plane shown here.

time and only slightly increases the number of rendered polygons, while producing smooth, natural-looking motion blur. It runs in real-time, even for moderately complex objects consisting of several thousand polygons, such as the airplane shown in Figure 2. Figure 3 presents an example of the generated motion blur for the same airplane.
The remainder of the paper is structured as follows. In Section 2, we discuss previous work in computer graphics on motion blur. We describe the basic algorithm for generating fast motion blur in Section 3. The various extensions of the algorithm, e.g., handling colored polygons, making the algorithm even faster, etc., are outlined in Section 4. In Section 5, we discuss the shortcomings and benefits of our algorithm; we also provide timing results for our implementation. We suggest directions of future work in Section 6.

2 PREVIOUS WORK

Insert Photograph 4

Figure 4: The only other real-time technique to blur objects consists of rerendering the object repeatedly. This technique does not banish strobing effects, as shown here. In addition, the technique is too slow for use in interactive applications when applied to moderately complex objects.

Much of the previous work on motion blur in computer graphics concentrates on how to generate motion blur for ray-tracing renderers. One of the earliest references [7] computes motion blur in a three step process. First, a ray-tracer determines visibility, as well as which pixels are moving points. Second, the algorithm convolves moving points with their image path functions to generate an

appropriate blur. Third and last, the static image is merged with the convolved image, properly taking depth relations into account.
Thereafter, research identifies motion blur as a temporal aliasing problem. Its solution for ray-tracing renderers is to supersample in the time domain with additional rays [3] [8] [9] [10].
All these solutions, since they depend on ray-tracing, are inherently too slow for interactive applications.
Other work also supersamples objects in time independent of ray-tracing. Rendering an object multiple times at different points in time approximates motion blur [11] [12]. However, if the number of temporal sampling points is finite and constant, the approximation is poor (see Figure 4). Furthermore, special purpose hardware [12] is necessary to merge these multiple images efficiently. Even then, the cost of rendering an object multiple times is prohibitive.
A simple example suffices to illustrate: if we render an object ten times per frame (which is not sufficient to produce smooth object blurs, even for moderately fast-moving objects) and assume a frame rate of 15 frames per second, then we must render the object 150 times per second. If the object consists of 2,000 polygons, like the plane in Figure 4, and is rendered in stereo, the renderer must process 600,000 arbitrary and possibly transparent polygons per second, a figure that exceeds the current performance capabilities of even high-end graphics workstations.
Haeberli and Akeley suggest increasing performance by applying repeated integration [13] to the rendering process [12]. In repeated integration, the renderer creates an initial frame by repeatedly rendering an object as described above. The resulting frame represents the object at time samples $t_1, \ldots, t_n$. Thereafter, the renderer does not discard the frame, but instead modifies it repeatedly. We first remove the initial object sample at time t_1 from the frame by rendering a frame of the object at time t_1 and subtracting it from the initial frame. The current frame now represents the time samples $t_2, \ldots, t_n$. Then we add the rendering of the object at time t_{n+1}. The resulting frame thus combines the samples $t_2, \ldots, t_{n+1}$.
While this method reduces the rendering overhead to rendering a motion-blurred object only twice for n samples, it is not appropriate for our purposes. We want to generate motion blur by supersampling the object's movement between subsequent frames. Repeated integration only incorporates information from the n previous frames and does not include information from the time period in-between subsequent frames.
The various approaches described in [11] [14] [15] [16] [17] all operate on the 2D projection of the motion onto the film plane. Therefore, these algorithms cannot easily be adapted to take advantage of the graphics accelerators common in today's workstations. The resulting performance penalty in rendering polygons therefore disqualifies these algorithms for use in virtual reality applications.
In other work, Reeves introduces a motion blur algorithm for his particle systems in [18]. Unfortunately, the algorithm only applies to particles — volumeless points that are the size of a pixel.
Finally, Chen has devised an algorithm to morph 2D images that correlate on a pixel basis [19] and has extended it to synthesize motion blur. While the morphing part of his algorithm almost operates in real-time, automatically correlating the source images does not. Thus, it is unsuitable for general, interactive, or dynamic scenes.
In summary, all known algorithms for computing motion blur are inadequate for use in virtual reality applications. They are either too slow — they generate too much computation overhead or cannot take advantage of today's hardware-assisted rendering capabilities — or do not apply to complex dynamic and interactive polygonalized scenes.

3 BASIC ALGORITHM

The basic algorithm for generating motion blur operates in real-time. The algorithm blurs individual objects, not whole images. We assume that the object is polygonal and that the polygons represent a closed volume. The simulated shutter opens at time t_0 and closes at time t_1. We assume knowledge about object and camera position and orientation during that time-interval. To make the algorithm perform in real-time, we rely on rendering hardware capable of properly resolving depth relationships between transparent polygons, i.e., modern z-buffer engines.
We base our algorithm on the following observations. The image of an object blurs when the object's image on the film plane moves during the time the (simulated) shutter is open, i.e., during the (simulated) exposure time. Generating motion blur on the film plane by blurring the object's projection cannot exploit modern rendering hardware that greatly accelerates transforming polygons from object space into image space.
Therefore, we solve the motion blur problem in object space. In object space, motion blur is caused by an object occupying an extended volume of space during (simulated) exposure time. We call this extended volume the *motion volume*. Since not all of the object occupies all of the motion volume during the (simulated) exposure time, the motion volume is of varying color density.
To represent this motion volume accurately, we have to sweep all of the object's polygons along their motion paths during exposure time. During the sweep, hidden surface calculations between the object's polygons themselves and with potential background objects must be made to obtain accurate color density values for the motion volume. These operations are expensive to perform and are not explicitly supported in current hardware.
We can approximate the motion volume by blending the object at times t_0 and t_1 with the surfaces obtained by sweeping out the visible edges of each of the polygons in the object. Each of these edges produces a semi-transparent surface representing its contribution to the color density of the motion volume. We can then let the hardware integrate these semi-transparent surfaces via alpha-blending during the z-buffering process. However, this algorithm is still too costly for two reasons. First, the algorithm generates too many sweep polygons, and second, current hardware requires semi-transparent polygons to be depth-sorted to guarantee proper alpha-blending [20]. Since the algorithm produces many overlapping polygons and sorting them is too expensive, the resulting image is likely to have alpha-blending errors.

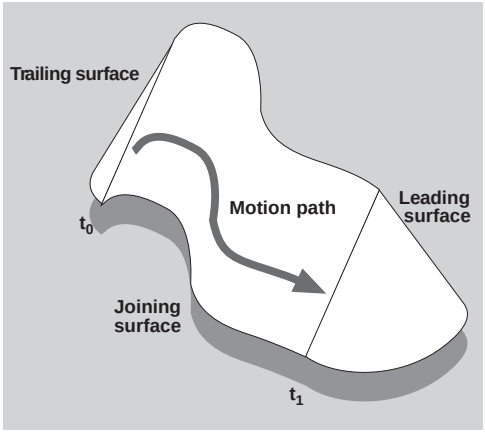

Figure 5: As a cone moves over time, it sweeps out a motion volume. The motion volume consists of a leading, trailing, and joining surface, shown here.

Instead, we choose to approximate the motion volume by sweeping only a selected set of edges, those representing the visible silhouette of the object as seen from the axis of motion. To distinguish this silhouette from the silhouette seen by the viewer, we call the motion vector based silhouette the *contour*. The contour is the set of edges joining a polygon that faces the direction of motion with a polygon that faces away from the direction of motion. Sweeping these contour edges provides a surface that closely approximates the geometry of the motion volume. If rendered semi-transparently, it also roughly approximates the color density of the motion volume. Modern z-buffer engines then resolve the hidden-surface relations for the semi-transparent motion volume polygons and potentially other objects.

Since the motion volume represents the volume the object occupies in the time interval $[t_0, t_1]$, the trailing surface of the motion volume is equivalent to the back of the object (with respect to the object's motion path) at the moment of departure, t_0. The leading surface of the motion volume is equivalent to the front of the object (again, with respect to the object's motion path) at the time of arrival, t_1. The joining surface (between the leading and trailing surfaces) of the motion volume represents the volume swept by the object's contour during the time interval (t_0, t_1). Figure 5 depicts the leading, trailing, and joining surface of an object's motion volume.

The basic algorithm constructs a motion volume in four steps, described in detail in the following four subsections. First, we compute the motion vector of the object. Second, on the basis of the motion vector we split the object into two parts: the front and the back. Third, we construct the leading, trailing, and joining surfaces of the motion volume from the front and back parts of the object. Fourth and last, we assemble the motion volume from the leading, trailing, and joining surfaces.

3.1 Computing the Motion Vector

The motion vector of an object describes the object's motion path over time. The motion path depends on the motion of the object as well as the camera motion. For now, we assume that the object's and the camera's motion are both linear translations. In section 4, we generalize the algorithm to better handle curved motion paths, as well as object and camera rotation.

Since our motion blur algorithm works on a per-object basis, we render the overall frame at a single instant, t_1. We therefore use the camera parameters as of time t_1. The motion vector of the object thus has to take into account the camera movement from time t_0 to time t_1.

Since we assume camera movement to be a linear translation, the velocity vector for the camera is $vel\_vec_{cam} = pos_{cam}(t_1) - pos_{cam}(t_0)$, where $pos_{cam}(t)$ describes the position of the camera at time t. Analogously, the velocity vector for the object is $vel\_vec_{obj} = pos_{obj}(t_1) - pos_{obj}(t_0)$. The motion vector of the object describes the total apparent motion of the object with respect to the instantaneous frame at time t_1; it thus is the difference of the two velocity vectors:

$$motion\_vec_{obj} = vel\_vec_{obj} - vel\_vec_{cam}$$

Finally, in order to apply the motion vector to the object polygons defined in object space, we transform the motion vector from world space into object space. Figure 6 shows an object and its motion vector.

3.2 Splitting the Object and Defining the Contour-Line

The motion vector defines the front and back parts of an object. We define all object polygons that face into the direction of motion as front polygons, and all object polygons that face away from the direction of motion as the back polygons. We therefore classify all object polygons as belonging either to the front or the back:

$$\begin{aligned} FrontPolygons &= \{p | PolygonNormal(p) \bullet motion\_vec_{obj} \geq 0.0\} \\ BackPolygons &= \{p | PolygonNormal(p) \bullet motion\_vec_{obj} < 0.0\} \end{aligned}$$

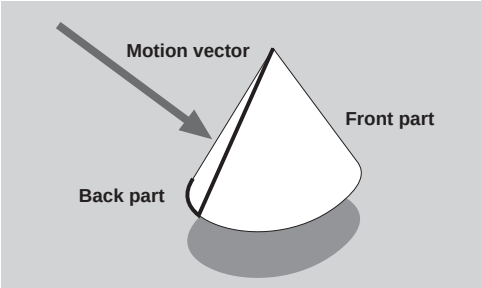

Figure 6: The motion vector of an object determines the contour-line of an object. The contour-line, shown in bold here, splits an object into a front and a back part.

An edge that joins a front with a back polygon is called a *contour edge*. The contour edges separate an object's front part from the object's back part.

$$\begin{aligned} EdgeSet_1 &= \{e| \quad RightSide(e) \in FrontPolygons \ \wedge \\ &\qquad LeftSide(e) \in BackPolygons\} \\ EdgeSet_2 &= \{e| \quad RightSide(e) \in BackPolygons \ \wedge \\ &\qquad LeftSide(e) \in FrontPolygons \ \} \\ ContourEdges &= EdgeSet_1 \cup EdgeSet_2 \end{aligned}$$

Only the visible contour edges should contribute to the shading of the motion volume surface. Therefore, we define the *contour-line* as the subset of contour edges that are visible. We approximate visibility by testing whether a contour edge belongs to a camera front-facing polygon.

$$\begin{aligned} ContourLine &= \{e \in ContourEdges| \quad RightSide(e) \text{ is camera front-facing} \vee \\ &\qquad LeftSide(e) \text{ is camera front-facing}\} \end{aligned}$$

Figure 6 shows an object and its motion vector. It also illustrates the resulting front and back parts of the object and the contour-line.

3.3 Constructing the Leading, Trailing, and Joining Surfaces

In this section, we construct the leading and trailing surfaces of the motion volume from the front and back parts of the object. Sweeping the contour-line produces the motion volume's joining surface.
As mentioned above, the front part of the object at time t_1 is equivalent to the leading surface of the motion volume. We therefore use the front of the object as the leading surface of the motion volume.

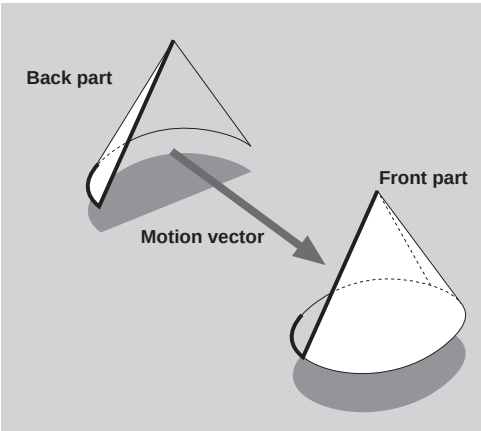

Figure 7: We approximate the motion blur of an object by placing its front part at the object's position as of the end of the exposure time, and translating the object's back to the object's position as of the beginning of the exposure time.

Since we later join the leading surface with the trailing surface of the motion volume via the joining surface, we have to ensure that the contour-edges used to create the leading and the trailing surfaces are coherent with one another. The coherency required is the ability to connect all leading contour edges with their respective trailing contour edges via a four-sided polygon.
We ensure this coherency by approximating the motion volume further. Instead of using the back part of the object as of time t_0, as described earlier, we use the back part of the object as of time t_1. We thus implicitly assume that the back part of the object does not

change in the time interval $[t_0, t_1]$. However, we can also guarantee coherency: since we create the leading and trailing surfaces from the front and back parts of the same object, their contour-lines are identical.
Therefore, the trailing surface of the motion volume is the back part of the object at time t_1, translated to the object's position as of time t_0, i.e., translated by $-motion\_vec_{obj}$. In Figure 7 we show the leading and trailing surfaces of the motion volume we thus created.

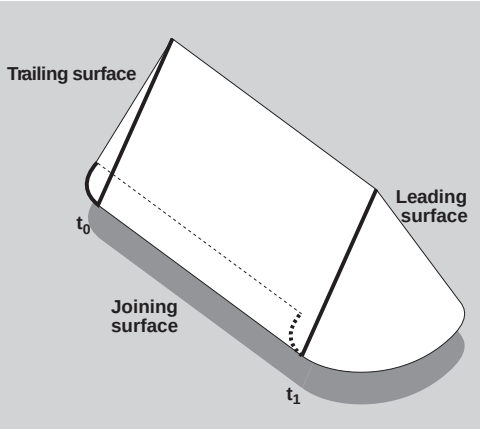

Figure 8: We construct the joining surface by extruding the contour-line along the motion vector, thus connecting the leading and trailing surfaces with a set of four-sided polygons.

Finally, we construct the joining surface of the motion volume. As already mentioned, the joining surface consists of a set of four-sided polygons. Each joining polygon connects a leading contour-line edge with its corresponding trailing contour-line edge. Figure 8 shows the resulting leading, trailing, and joining surfaces of the motion volume.

3.4 Assembling the Motion Volume

Insert Photograph 9 (There are no photographs 5 through 8.)

Figure 9: We simulate motion blur by assembling the leading, trailing, and joining polygons constructed previously. The transparency values of the polygons model the temporal filter used; here we are modeling a ramp filter.

The leading, trailing, and joining polygons constructed in Section 3.3 constitute the motion volume. To make the motion volume look like motion blur, we need to associate transparency values with the various polygons. The transparency values we assign reflect the temporal filter [9] selected to model the motion blur. A variety of choices of filters are available, for example, Gaussian, triangular, ramp, or box filters.
In all the examples shown in this paper, we use a ramp filter, because it gives visually pleasing results. Therefore, we always render the leading polygons as fully opaque. The trailing polygons' transparency varies with the length of the motion vector divided by the extent of the object measured along the axis of the motion vector. Therefore, we assign a transparency of

$$max(0.1, 1.0 - length(motion\_vec_{obj})/extent_{motion\_vec_{obj}}(obj))$$

to the trailing polygons. In particular, if the motion vector's length is zero, we render the trailing polygons as opaque.
The joining polygons vary in transparency linearly from fully opaque (where they join with the leading polygons) to whatever transparency value is assigned to the trailing polygons. Figure 9 shows an example of a motion volume constructed by the algorithm outlined in this section.

4 EXTENSIONS

In this section we extend the basic algorithm described in Section 3. In particular, we outline how to add lighting normals and color; how to better approximate non-linear motion; how to handle object rotation; and how to precompute contour edges to accelerate the algorithm further.

4.1 Lighting Normals and Color

Insert Photograph 10

Figure 10: We extend the algorithm to apply to colored objects that have associated lighting normals, such as this cube.

We extend the basic algorithm to also apply to colored objects with associated lighting normals, such as the cube shown in Figure 10. We assign to each vertex of the joining polygons an individual lighting normal and color. A joining polygon vertex derives its lighting normal and color from the corresponding vertex belonging to a leading or trailing polygon. The hardware then shades the joining polygons appropriately by linearly interpolating lighting normals and colors.
If a joining polygon connects a leading polygon with a camera back-facing trailing polygon, or similarly, connects a camera back-facing leading polygon with a trailing polygon, the resulting joining polygon is possibly camera front-facing and thus displayed. The joining polygon therefore exhibits lighting normal and color data derived from both the leading and trailing polygons. But an observer does not see both the leading and the trailing polygons (one of them is camera back-facing, and therefore culled), and perceives an incorrectly colored blur — the joining polygon displays wrong lighting normals and color values.
To make the joining polygons more consistent, we therefore use lighting normals and color only from visible leading and trailing polygons. Joining polygon vertices corresponding to vertices of camera back-facing leading or trailing polygons copy the lighting normal and color of the vertices of the visible trailing or leading polygon, respectively. The resulting joining polygon thus appropriately displays constant lighting normals and color. Figure 11 shows the resulting effect.

4.2 Piecewise Linear Motion Paths

In Section 3.1 we restricted the motion vector to a linear translation. In this subsection, we generalize the algorithm to handle piecewise linear motion. Using piecewise linear motion paths, it is possible to approximate general motion paths. Therefore, instead of approximating the object blur with a single segment motion volume, we split the joining polygons into multiple, piecewise linear segments.
To create an n segment motion blur from time t_0 to time t_n, we require knowledge of the camera and object positions as of times $t_0, t_1, \ldots, t_n$. The basic algorithm then changes in four places. First, we compute n linear motion vectors $motion\_vec_{obj}[i]$ for the time intervals $[t_{i-1}, t_i]$, instead of one single motion vector. Second, the front and back parts of the object, as well as the contour-line are computed with respect to the vector $\sum_{i=1}^{n} motion\_vec_{obj}[i]$. Third, we translate the trailing polygons by the sum of the inverse motion vectors, i.e., by $-\sum_{i=1}^{n} motion\_vec_{obj}[i]$. Fourth and finally, we construct a different set of joining polygons.

Insert Photograph 11

Figure 11: The cube shown earlier is motion blurred in this picture.

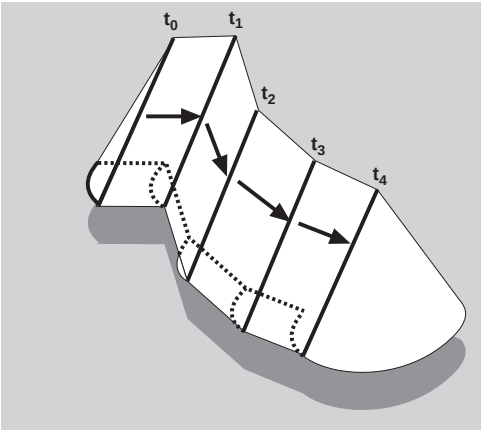

Figure 12: To approximate more complex motion, we split the joining surface of the motion volume into multiple segments. The resulting motion blur, shown here, is piecewise linear.

We connect each leading contour-line edge with the corresponding trailing contour-line edge with n four-sided polygons. Each of the n polygons represents the translated trailing contour-line edge swept along one of the piecewise linear motion vectors. In particular, the ith such polygon connects the translated trailing contour-line edge translated again by $\sum_{j=1}^{i-1} motion\_vec_{obj}[j]$ to the trailing contour-line edge translated by $\sum_{j=1}^{i} motion\_vec_{obj}[j]$.
Analogous to the basic algorithm, we assign transparency values to vertices of the joining polygons. The hardware then linearly interpolates the vertex values for the transparency of the joining polygons.
A vertex's transparency value depends on its distance from the leading contour-line edge. The vertices belonging to the trailing contour-line edge translated to $\sum_{j=1}^{i} motion\_vec_{obj}[j]$ (see above) have a transparency of

$$max(0.1, 1.0 - (\sum_{j=1}^{n-i} length(motion\_vec_{obj}[n-j+1]))/extent_{motion\_vec_{obj}}(obj))$$

Figure 12 illustrates the construction of a multi-segment motion blur.
Similarly, we assign lighting normals and color to the individual vertices of the joining polygons. For each joining polygon vertex, we linearly interpolate its lighting normal and color between the lighting normal and color of the corresponding vertices on the leading and trailing contour-line edges. The interpolation depends on the vertex's distance (in the same sense as for the vertex's transparency value) from the leading and trailing contour-line edges. The hardware then linearly interpolates the polygon lighting normal and color for each joining polygon.[1]

[1] Actually, we only interpolate vertex color. To save computation time, we do not interpolate vertex normals and instead simply reuse the lighting normals of the leading contour-line edges.

While piecewise linear multi-segments can adequately approximate high curvature motion paths, their performance deteriorates when using a large number of segments. In the worst case the number of polygons for each segment approaches the number of polygons of the object. However, in the average case the number of polygons for each segment is only a small fraction of the total number of object polygons. In either case, the number of polygons is fewer than the number of polygons generated by the only other technique used for interaction, i.e., rerendering the object multiple times with varying transparency. In addition, our algorithm requires vastly fewer segments to produce a similarly smooth motion blur, as compared to the motion blur generated by rerendering the object multiple times.

4.3 Object Rotation

Here we develop the basic algorithm further to handle object rotation. The method we describe here applies to moving objects that rotate slowly; our algorithm seems ill-suited for moving objects that rotate rapidly.

We expand our motion blur algorithm to take into account object orientation. Instead of merely translating the trailing polygons to the object's position at the time the simulated shutter opens, we also rotate them so they match the object's orientation at that time.

We apply the same technique to multi-segment blurs. For an n-segment blur, we generate properly positioned contour-lines for $n + 1$ points in time. Again, instead of just translating the contour-lines, we also orient them to match the object's orientation at each of those points in time. The joining polygons then connect these contour-lines as before.

However, if an object rotates quickly compared to the object's translational speed, our algorithm becomes ineffective. Depending on the direction of rotation and the number of segments used, the generated joining polygons possibly intersect with one another and no longer describe the motion volume surface accurately. In other cases, the joining polygons may degenerate into self-intersecting polygons.

While our algorithm does not handle fast rotating objects in general, there is a solution for the special case when an object is rotating, but otherwise moving slowly. Instead of using the motion vector to define the contour edges, we use the axis of rotation. Similarly, we make the transparency proportional to the rotational speed.

4.4 Precomputation of Contour Edges

The final extension suggested here further accelerates the computation of motion blur for individual objects. We choose a set of prespecified motion vectors for a given object that determine a set of contour edges and associated front and back parts of the object. We precompute and store these contour edges and associated front and back parts. When computing the motion blur of the object, we match the actual motion vector to the nearest prespecified motion vector. We use the associated precomputed contour edges and front and back parts of the object instead of computing and using the actual data.

There are two disadvantages associated with this modification. First, since a precomputed motion vector replaces the actual motion vector, the contour edges for the direction of the generated motion blur are inaccurate. We can minimize this inaccuracy by storing a larger number of motion vectors and associated contour edges and front and back parts. These increased storage requirements, however, are the second disadvantage.

On the other hand, this precomputation reduces the most expensive part of the basic algorithm to a table lookup operation. Therefore, the motion blur computations are negligible. Furthermore, the number of different contour edge sets is finite. In particular for simple objects, it is therefore feasible to compute and store all possible front and back parts and associated contour edges. The resulting storage overhead is small, and the generated motion blur is equivalent to the dynamically computed solution.

This technique is most useful for objects with a preferred direction of motion. For example, an airplane usually moves fast only when flying forward, so that motion blur is unlikely to be generated for any direction other than roughly straight forward. Prespecified motion vectors thus cluster around this preferred direction. The airplane therefore requires only a small number of motion vectors, resulting in a small storage overhead. For best results, we recommend to distribute the prespecified motion vectors stochastically to reduce the perceived error.

5 DISCUSSION

We discuss the disadvantages and advantages of our algorithm and its extensions. Where appropriate, we compare it with the algorithm that renders an object multiple times with varying transparency; that algorithm is the only other algorithm that runs close to real-time (see Section 2). Table 13 lists timing results in our implementation for rendering an object without motion blur ("no motion blur"), with motion blur generated by our described basic algorithm with color interpolation ("single segment"), with motion blur generated by the described multi-segment algorithm with three segments ($n = 3$) and color interpolation, and with motion blur generated by rerendering the object ten times with varying transparency ("render 10 times"),

5.1 Disadvantages

Our algorithm has three salient disadvantages. First, it only roughly approximates real motion blur. In particular, shading information for the joining polygons derives from the contour edges, as opposed to all visible edges. Therefore, shading of the joining polygons depends on the direction of motion, which possibly leads to inconsistent blurs when rapidly changing direction.

[2]The display refresh rate limits frame rate to 68.11 Hz.

Motion Blur Algorithm	Cube (12 triangles) max. frame rate	$\frac{algorithm}{no\ blur}$	Airplane (2020 triangles) max. frame rate	$\frac{algorithm}{no\ blur}$
No Motion Blur	68.11 fps[2]	1.0	23.77 fps	1.0
Single Segment	68.11 fps	1.0	17.74 fps	0.74
Multi-Segment	68.11 fps	1.0	12.02 fps	0.51
Render 10 times	23.55 fps	0.35	2.17 fps	0.09

Figure 13: We compare execution times of the described algorithm to other techniques. The timing results were generated on an HP 735 with a TVRX T4 graphics board.

Second, our algorithm adapts poorly to object rotations if these rotations are fast compared to the overall object movement (see also Section 4). Similarly, we do not address object scaling or warping (independently moving individual vertices of an object).
In comparison, rerendering the object multiple times with varying transparency does not share this disadvantage. It deals equally well with straight-line and curved motion, as well as object rotation, scaling, or warping. However, it is only practical for simple or slow moving objects. When an object moves quickly and therefore, when motion blur is needed most, the generated temporal aliasing becomes unacceptably crude, or conversely, the solution becomes non-real time.
Finally, the third flaw is hardware related. Current z-buffers require transparent polygons to be rendered in order [20], yet our algorithm does not sort the generated transparent polygons. However, possible resulting alpha-blending errors are not noticeable, because our algorithm generates few overlapping, transparent polygons.

5.2 Advantages

The many benefits of our algorithm outweigh the above disadvantages. Most notably, it is the only motion blur algorithm that runs in real-time for fast-moving and complex objects, making it fit for interactive applications, such as virtual reality. To achieve this performance, it relies on standard z-buffering hardware that is capable of rendering transparent polygons.
The algorithm induces low computation overhead, even when compared to rendering an object without motion blur. The run-time of the algorithm itself is small compared to polygon rendering time (especially if we precompute contour edges and front and back parts, as described in Section 4). Polygon rendering time increases as the algorithm generates additional polygons to simulate motion blur. However, if the number of multi-segments is kept small (less than 5), the algorithm usually produces less additional polygons than the number of polygons of the original object. The number of additional polygons generated is orders of magnitude less than for the algorithm that renders the object multiple times.
Since the described algorithm operates on an per-object basis, as opposed to on an per-image-basis, performance increases further. Instead of uniformly applying motion blur techniques to all visible objects, static or dynamic, the application programmer can single out the fast moving objects in a scene. Since in a typical scene only few objects move rapidly, applying motion blur techniques to only these few objects is comparatively inexpensive.
When comparing our algorithm to rerendering an object multiple times with varying transparency, other advantages become apparent. First, our algorithm is object speed independent. In contrast, multiple object rerendering is object speed dependent: the faster an object moves, the more samples are required to maintain the same motion blur quality, i.e., the same (lack of) strobing.
Second, our algorithm samples the position and orientation of the object only $n+1$ times for an n-segment motion blur. The multiple rerendering algorithm typically requires many more object samples to achieve a similar quality motion blur. If sampling an object is expensive, for example, if a dynamic, physically-based simulation governs the object's behavior, our algorithm is favorable.
Third and finally, our algorithm produces a smooth blur. Since real motion blur is smooth as well, the viewer immediately accepts our solution. Furthermore, the smoothness effectively reduces strobing effects. In contrast, the multiple rerendering algorithm generates a jagged blur, unless it renders the object once for every pixel moved, which in turn is prohibitively inefficient.

6 CONCLUSIONS AND FUTURE WORK

Blurring fast-moving objects is highly desirable in interactive applications in general, and virtual reality in particular. However, the execution times of previous motion blur algorithms restrict their applications to the 2D domain. Our algorithm is fast enough to make motion blur feasible for complex objects in interactive, 3D environments.
The motion blur algorithm described here achieves its speed by approximating real motion blur. We opt for a result that looks like motion blur and is fast, not accurate and slow. Our reasoning is that in an interactive application the generated image of a motion blurred object is on-screen for 0.1 seconds or less. In this time span, a viewer does not notice whether the displayed blur is correct, as long as the image roughly approximates motion blur — a phenomenon of which cartoon animators have long taken advantage.
The algorithm and its extensions produces motion blur in real-time for fast moving objects. However, improvements are certainly possible. Throughout our description of the algorithm, we have pointed out where we make approximations to the real solution and why. For future work, these decisions should be reexamined and possibly revised, while maintaining the real-time performance.

Other research directions remain unexplored. Our solution for fast rotating objects needs improvement (see Section 4). Similarly, we do not address object warping (see Section 5). Finally, better transparency functions are possible, for example, simulating other temporal filters (see Section 3.4).

Acknowledgements

We would like to thank Dan Robbins for drawing the illustrations, John Hughes for criticizing early drafts of this paper, and our sponsors listed on the first page.

References

[1] Michael Langford. *The Master Guide to Photography*. Knopf: Distributed by Random House, New York, 1982.

[2] RKO-Radio Pictures. "King Kong" Director: Ernest B. Schoedsack, Special Effects: Willes O'Brian, 1933.

[3] Robert L. Cook, Thomas Porter, and Loren Carpenter. "Distributed Ray Tracing" *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):137–145, July 1984.

[4] F. Thomas and O. Johnston. *Disney Animation: The Illusion of Life*. Abbeville Press, New York, NY, 1981.

[5] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. "Cone Trees: Animated 3D Visualizations of Hierarchical Information" In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, pages 189–194, 1991.

[6] Bay-Wei Chang and David Ungar. "Animation: From Cartoons To The User Interface" *UIST Proceedings*, pages 45–55, November 1993.

[7] M. Potmesil and I. Chakravarty. "Modelling Motion Blur In Computer-Generated Images" *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):389–399, July 1983.

[8] Mark E. Lee, Richard A. Redner, and Samuel P. Uselton. "Statistically Optimized Sampling For Distributed Ray Tracing" *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):61–67, July 1985.

[9] Mark A. Z. Dippe and Erling Henry Wold. "Antialiasing Through Stochastic Sampling" *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):69–78, July 1985.

[10] Robert L. Cook. "Stochastic Sampling In Computer Graphics" *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[11] J. Korein and N. Badler. "Temporal Anti-Aliasing In Computer Generated Animation" *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):377–388, July 1983.

[12] Paul Haeberli and Kurt Akeley. "The Accumulation Buffer: Hardware Support For High-Quality Rendering" *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):309–318, August 1990.

[13] P. S. Heckbert. "Filtering By Repeated Integration" *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):315–321, August 1986.

[14] Edwin Catmull. "An Analytic Visible Surface Algorithm For Independant Pixel Processing" *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):109–115, July 1984.

[15] C. W. Grant. "Integrated Analytic Spatial And Temporal Anti-Aliasing For Polyhedra In 4-Space" *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):79–84, July 1985.

[16] N. L. Max and D. M. Lerner. "A Two-And-A-Half-D Motion-Blur Algorithm" *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):85–93, July 1985.

[17] Nelson Max. "Polygon-Based Post-Process Motion Blur" *The Visual Computer*, 6(6):308–314, December 1990.

[18] W. T. Reeves. "Particle Systems – A Technique For Modeling A Class Of Fuzzy Objects" *ACM Transactions on Graphics*, 2:91–108, April 1983.

[19] Shenchang Eric Chen and Lance Williams. "View Interpolation For Image Synthesis" In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288, August 1993.

[20] Hewlett Packard. *Starbase Reference: HP 9000 Computers*, first edition, January 1991. HP Part No. 98592-90067.