Real-Time Geometric Motion Blur for a Deforming Polygonal Mesh

Nathan Jones
Formerly: Texas A&M University
Currently: The Software Group
nathan.jones@tylertechnologies.com

John Keyser
Department of Computer Science
Texas A&M University
keyser@cs.tamu.edu

ABSTRACT

Motion blur is an important method for increasing the visual quality of real-time applications. This is especially true in the area of interactive applications, where designers often seek to add graphical flair to their programs. In many cases, these applications use animated characters, where a polygonal mesh is wrapped around an animated skeleton. As the skeleton moves, the mesh deforms. In this paper, we present a method for adding a geometric motion blur to a deforming polygonal mesh. The scheme we present keeps track of a character's motion silhouette, and uses this to create a polygonal mesh. When this mesh is inserted into the scene, it gives the appearance of an artistic motion blur for an object or particular character. This method is generic enough to work on nearly any type of moving polygonal model, and also approximates swept volumes.

Keywords: Motion Blur, Swept Volume, Silhouette of Motion

ACM Categories: I.3.6 Methodology and Techniques—Graphics data structures and data types, I.3.5 Computational Geometry and Object Modeling, I.3.7 Three-Dimensional Graphics and Realism—Animation

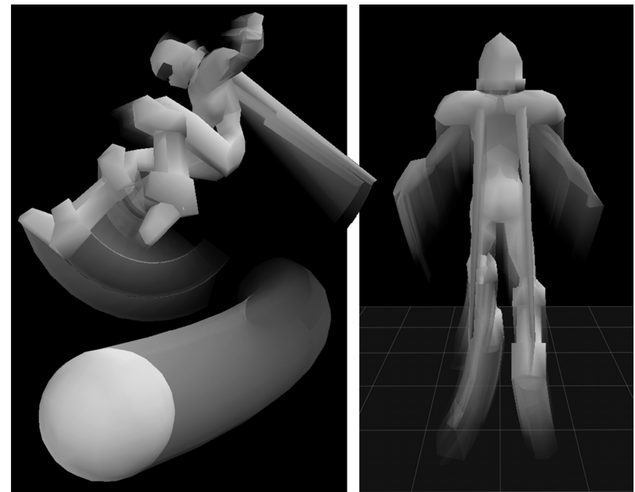
1 INTRODUCTION

Motion blur is important to real-time applications, such as games or training simulators. It is just one of many methods that can be used to portray an object in motion, and to draw the viewer into the scene. There are a number of uses for motion blur.

One is to add realism to a scene. In films, the shutter creates a motion blur inherent to the filming process, and this blur feels natural to most viewers. The blur is often missing in real-time applications due to the difficulty or speed loss when creating it. Its lack is quite noticeable if one sees two computer generated films, one with motion blur and one without. This is a "true" motion blur – attempting to generate an image covering a period of time – and should be distinguished from more artistic effects. Regardless of whether the motion blur is "true" or "artistic," if one can add motion blur to a real-time application, then the viewer will be drawn into the scene more so than if the blur is lacking.

A second use for motion blur is to create a desired special effect. It can be used to show a streaking object or just to draw attention to an object in the scene. By controlling the length of time the blur remains visible, a designer can get varying effects. Such effects (e.g. speed lines) are particularly useful in games.

A third use is to show motion in a still image. When viewing a still image with no motion blur on a figure, it is often difficult to determine what motion is happening in the scene. The motion blur for such effects must be shown to indicate direction (i.e. uniform sampling across the time domain, as in a simulation



of a true camera, would not be appropriate). Such motion blur is useful in still media such as comic books.

While there are many approaches to showing motion blur, most are insufficient to allow for all of these needs, produce results that are visually unacceptable, or do not deal with highly deforming meshes as can often be found in today's interactive games. These other methods will be discussed in section 2.

We present a method for generating visually appealing artistic motion blur effects. Using a method similar to that of swept volume construction, we determine the set of edges that lie along the dividing point between the front and back of a mesh based on its motion, or the *silhouette of motion*. By connecting a set of silhouettes captured at regular time intervals with polygonal quadrilaterals, and decreasing the opacity of the polygons the farther back in time they were created, we are able to create a visually appealing motion blur. We refer to this generated set of polygons as the *blur shell*, as it is an extra set of polygons that are attached to the original model.

1.1 Key Results

The primary distinguishing features of our method are that it:

- Produces a motion blur effect even for meshes that are animating and deforming,
- Handles rotational motion easily,
- Can be used to add an effect to interactive scenes, show motion in a still image, or enhance realism compared to scenes with no motion blur,
- Has low computational overhead,
- Easily integrates with other graphics applications,
- Should be able to take advantage of future hardware advances,
- Can approximate a swept volume computation.

2 PREVIOUS WORK

There has been a good deal of research into the topic of both artistic and "true" motion blur; some of this previous work will be discussed here.

Motion blur has been an important aspect of computer generated images for many years. Some of the earliest and most fundamental work includes that of Korein and Badler[8], Potmesil and Chakravarty[11], and Dippe and Wold[4]. Possibly the best looking and most widely used motion blur technique is temporal anti-aliasing[8][3]. This is done by super-sampling the scene in the time domain. A particular pixel is sampled at various points in time around the current frame; the samples are then filtered to create the final color for a pixel. While this works well for offline rendering or ray-tracing, it is far too slow for real-time applications, and current hardware does not support it well. Such methods try to compute a "true" motion blur, simulating the effect of a lengthier exposure time.

A number of techniques have been proposed for creating motion blur in 2D image space. Several of these are too slow for real-time applications and do not use current hardware to achieve results[9][2]. More recently, however, hardware support for some image-space motion blur effects, allowing for real-time simulated motion blur, has been added by most graphics hardware manufacturers, including NVidia[10]. 2D Image-space motion blurs allow one to account for both camera and object motion, whreas 3D motion blur only accounts for object motion (though camera motion can be inelegantly simulated by instead simulating the motion of objects relative to a fixed camera. We believe, however, that our 3D geometric motion blur allows several advantages over such 2D methods, particularly in terms of the artistic and special effects that can be created.

A common method for creating motion blur is to copy the last n frames of an object's motion into the scene, combining images via an accumulation buffer[6]. All images can have the same opacity, or each copy can be slightly less transparent than the previous, until the current object is added to the scene at full opacity (e.g. Shreiner et al. [12]). There are two problems with this. First is that the complete object must be rendered n+1 times, which can create a large increase in the number of polygons in the scene. The second is that as the object picks up speed, the distance between the n copies will increase and since there is no coherence between the copies, the visual result is unacceptable. Even without changes in motion, an image can easily appear as several discrete samples, rather than one smooth image.

A method proposed by Wloka and Zeleznik[14] simulates a sweep style motion blur by separating the front and back of an object, and connecting them with polygons that connect the two sections. The separating polygons are constructed along the motion path between frames of the object to simulate the object's motion. A downside to this approach is that it does not consider an object using a deforming mesh, where the sweep created by the mesh can intersect itself in many places, thereby not being as useful in today's programs that often deal with deformable mesh characters wrapped around an animated skeleton.

ATI has implemented an approach extending that of Wloka and Zeleznik[13]. Their approach involves distorting the geometry of an object within a vertex shader based on its motion. Opacity is adjusted to create more artistic effects, but this method is applicable to only simple geometry. In 2001, NVidia also proposed a "stretch" effect. Using one previous frame, they generate a motion vector, subdivide the model into halves, and create a (alpha-faded) stretch between the two halves. While this is easily done in real-time, the method is not very extensible and the quality of the blur result is relatively poor.

The method presented here uses a technique similar to that of swept volume computation. Assuming that the silhouette information is not discarded, but still connected in a manner we use to create the blur, our blur polygons give a reasonable approximation of the swept volume the object passed through to get to its current position and location. Swept volume computation is an ongoing field of study [1][7]. Though not discussed here, we believe our blur shell can be useful in other applications (including collision detection applications) where an approximation to a swept volume is desired.

3 METHOD

Here we will discuss the manner in which we generate and maintain the blur.

In essence, our scheme is designed to create a blur shell. This is a set of polygons that when added to the scene gives the impression that there is a motion blur associated with the object. Blur shell polygons are stored in a circular list attached to each polygon (this is described below). To do this, we need two pieces of information to be provided by the application:

1. The length of time the blur should represent
2. The number of subdivisions in the blur shell

The time span of the blur is the maximum amount of time any one part of the blur exists in the shell (i.e. the period of time the blur represents); and the subdivisions control the quality of the blur. The more subdivisions, the smoother and more rounded the blur will look, and the more computation will be needed to maintain it.

The silhouette of motion need not be updated on a continual basis. Instead, the silhouette is updated every (time_length / #_subdivisions) seconds, which may be more or less often than a frame update. The only portion of our method that must be performed on a per-frame basis is the building and rendering of the current silhouette of motion, discussed in section 3.3.

There are two main steps we must complete in order to use our method. These are:

- On each update, find the Silhouette of Motion of the mesh based on its motion.
- On each frame we want to display the blur, we must construct polygonal strips from the Silhouette of Motion.

3.1 Requirements

There are a few assumptions we make regarding the mesh to be blurred.

The mesh can be deformed in real time, as by a skeleton, but the world coordinates of each vertex must be known. In many cases, this adds little overhead, as many skeletal animated meshes know their vertex positions in relation to the skeletal root. If an object's mesh is not deforming, but the object is translating, then the positions of the vertices only need to be updated when the silhouette of motion is updated. In addition to world coordinate positions of the vertices, each vertex must also know its own normal in world space. We found good results were obtained in many cases when each vertex's normal was simply the average of the normals of adjacent polygons. Note, however, that the ideal normal value to use for finding a silhouette of motion could be (but usually is not) different from the normal to be used for rendering; this will be highlighted later. Unless stated otherwise, a vertex's world position and world normal will be referred to simply as its position and normal for the remainder of this paper.

The mesh data structure is augmented to include additional information needed to maintain the silhouette of motion and to create the blur shell. Associated with each polygon is:

- A circular list (represented in an array) with length equal to the number of subdivisions to be used in the blur. Each element of the list represents one blur update. Each list element contains:
  - A flag marking whether or not an edge of the polygon exists on the silhouette of motion
  - Two vertex positions and normals that represent an edge on the silhouette of motion, if there is one.
- Two indices describing the last two vertices on the polygon that were part of the silhouette of motion.
- The number of updates that have passed since the polygon had an edge on the silhouette.

In addition, a global index is used to keep the system synchronized between and during updates. We shall refer to this as the update index. This index corresponds to which element in the circular list of each polygon is the most recent.

3.2 Finding the Silhouette of Motion

We define the *silhouette of motion*, or SoM, as the set of edges on a polygonal mesh that represents the local silhouette as seen when looking at an object along its axis of motion. If the majority of a mesh does not move and only one portion of the mesh is in motion (as would be the case if a character moved just an arm) then only the SoM for the arm would change, not that of the rest of the mesh. A non-moving portion of an object does not have a SoM.

At each update we need to find the current SoM. Due to the difficulty in determining a full-mesh SoM, we use a per-polygon silhouette detection technique. Our method examines each polygon, and determines if one of its edges lies on the SoM, and then records the edge in the polygon's list element that corresponds to the current update index. The collection of all edges found in this manner will be an approximation to the actual silhouette.

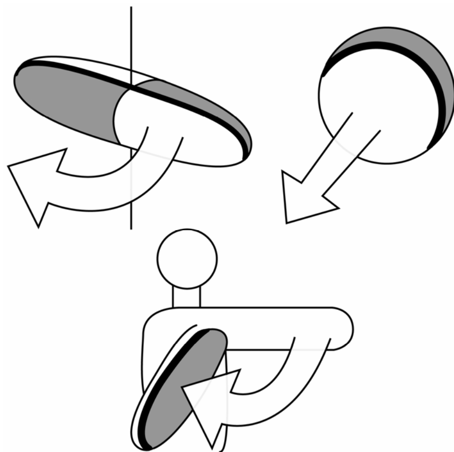


Figure 1: Various silhouettes of motion. The heavy line shows the silhouette; dark areas are the back side of the object

Before finding the SoM, we must first determine which side of motion each vertex is on. The facing of any vertex is determined by the sign of the dot product between the normal of the vertex and the vertex's direction vector. The direction points from the vertex's previous position to its current position. Note here that the goal is for this vertex normal to separate the "forward moving" from the "backward moving" edges. In some cases, it may be the case that a different normal from the one used for rendering would work better, but usually using the same one works fine. Generally, if the model is highly tessellated and normals vary smoothly, the normals work well. If a model contains replicated vertices with varying normals (i.e. creases), the creased edge itself can be treated as a polygon

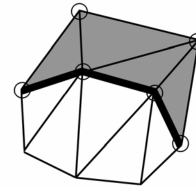


Figure 2: SoM found on a polygonal mesh. The heavy line shows the edges on the SoM; the dark area is the back facing portion of the mesh; circled vertices are back facing.

Next we find the set of edges that lie on the SoM. Assuming the mesh is triangular, each polygon with one front facing vertex (positive dot product) and two back facing vertices (negative dot product) records the two back facing vertices' position and normals in the list element corresponding to the update index. For non-triangular meshes, if there is at least one front-facing vertex, and a pair (or pairs) of adjacent back-facing vertices, then the back-facing edges are marked for the SoM. In addition, the flag is set in the element, stating that this polygon has vertices to contribute to the current blur silhouette. The vertices recorded are the two vertices of the polygon that make up an edge that lies on the SoM. The indices corresponding to the edge's end points are recorded in the polygon; these represent the most recent set of vertices in the mesh to lie on the silhouette. In addition, the number of updates since the last time an edge was on the silhouette is reset to zero.

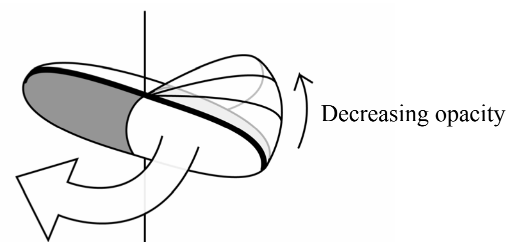


Figure 3: The results of successive SoM updates; each SoM is connected to the previous one by polygons. The small arrow depicts decreasing opacity.

All other polygons turn off the flag, increment the value containing the number of updates since the last time an edge of the polygon was found on the silhouette of motion by one, and record the positions and normals of the last blurred vertices in the list element corresponding to the update index. Unless these polygons have generated a blur within the last few updates, they will not contribute any vertices to this blur update.

Once this is done for all polygons, we have a set of polygon edges that define the silhouette of motion. We know that for convex objects, at any one blur update there are generally $O(\sqrt{n})$ edges on the silhouette; where n is the number of polygons within the mesh [5]. Thus, for any one update there will be relatively few edges to deal with, assuming convex objects. For objects composed of many convex parts (e.g. an articulated figure) or limited concavities, this number is still (relatively small).

We note that by exploiting frame-to-frame coherence, it may be possible to obtain the SoM more quickly. While we have

not explored this option, we believe this is unlikely to offer much improvement, as silhouettes can emerge on a moving object at locations far from the previous SoM. Also, there are numerous other algorithms for computing silhouettes that could be used for the SoM as well, but we believe this per-vertex computation test gives us highly efficient performance and offers the potential of being adapted to future-generation WGF-compliant graphics hardware. Our method examines only individual vertices and polygons; methods that make use of adjacency information across the mesh are not likely to be as easily adapted to hardware computation.

3.3 Rendering the Blur Shell

In order to display the blur, we rely on the use of polygon strips, specifically quadrilateral strips.

Just as each polygon generates part of the SoM independently, each polygon independently displays any blur associated with the edges that belong to it. To do so, a number of steps need to be taken.

Using the number stored with each polygon, we determine if the number of updates since the polygon last had an edge on the silhouette is within some constant defined by the designer. Any polygon that has not had an edge on the silhouette within this value contributes no polygons to the blur shell.

If the polygon recently had an edge on the SoM, then it continues to contribute to the blur shell. The polygon will add pairs of vertices to a quad strip in order to build up the blur. At most, there will be a number of vertex pairs equal to the number of subdivisions plus one, (the extra pair connects the blur to the base mesh). To create the effect of a blur, the polygon strip needs to fade out. To do this, the vertices that make up the polygon strips will have decreasing alpha values, relying on the graphics hardware to blend the blur shell's polygons.

To create the quad strip, we first add the mesh vertices that belonged to the edge the polygon last had on the SoM. Then we cycle through each of the polygon's list elements, starting with the one corresponding to the update index. The two vertices stored in each list element, with their normals, are added to the quad strip. The normals are used to help retain the shading the mesh had at the point the vertices were stored. If we go through a number of list elements equal to the number of subdivisions, we stop. We also stop the strip if too many elements in a row were flagged as not having vertices on the SoM. Each time a pair of vertices is added, the alpha value for the remaining vertices will be reduced by (initial_alpha / subdivisions). This allows the strip to start at some initial alpha value, and if the strip is displayed with the complete number of subdivisions, the last set of vertices added will be completely transparent. It is important to note that the blur shell itself may have color influenced by the original mesh, but does not have to (e.g. if one is adding artistic effects or speed lines).

Once all polygons have rendered their polygonal strips, the total set of polygons rendered will create a blur shell that is connected to the original deformed mesh.

At most, each polygon will add subdivisions*2 triangles to the scene. So, if we look at the size of the silhouette, which is theoretically bounded by $O(\sqrt{n})$, then we will be adding approximately $O(\text{subdivisions}*\sqrt{n})$ quadrilaterals to the scene to create our motion blur. The number of additional polygons rendered for the blur is thus less than that needed by some of the other motion blur techniques. In practice there are additional polygons rendered due to the dying off of some polygons that have not been on the silhouette in several updates, and their associated strips; but this number is limited and does not affect performance much. Also, as stated before, the amount of non-convexity can affect the $O(\sqrt{n})$ bound.

The method as described is not geared toward texture-mapped objects, but rather is more appropriate for artistic sweeps (such as speed lines), visual effects, or non-textured objects; there are many applications and games where this is sufficient. However, there are a number of ways that textured objects could be incorporated. One such method would be to use the UV coordinates of the model's vertices in the blur, thereby having a textured blur. The downside to this is that the motion blur will be textured its entire length by a single line of texels, those between the two vertices on the SoM. This is the approach used in other space-based approaches, but it is less than ideal. Another possibility would be to have the blur polygons use a different texture than the model. With this method, the UV coordinates of the blur shell's vertices can vary (possibly using a higher mipmap level), and a full texture could be used and seen on the blur. Yet another option would be to use a pixel shader to add texture, not necessarily a texture map, to the blur, thereby reducing its flat appearance.

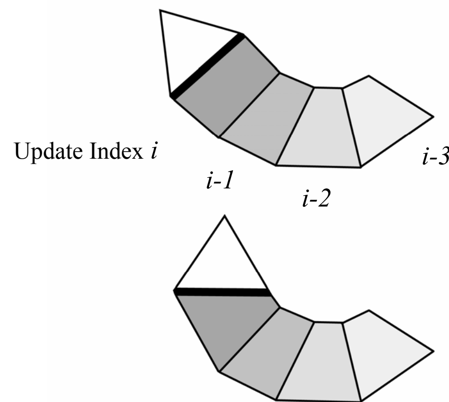


Figure 4: (Top) A strip generated by a polygon when rendering the blur. The heavy line shows a SoM edge, the gray values represent decreasing alpha values for the polygons. (Bottom) Same polygon, showing the strip created in a subsequent frame but before the next SoM update.

3.4 Analysis and Limitations

While this basic method gives good results, there are, of course, limitations in the accuracy of the blur. Most of these do not impact the visual look of the blur much, or are easily corrected.

- In some cases, the silhouette is not continuous along the mesh (e.g. when a polygon is perpendicular to the direction of motion). These are rare in our testing, and do not significantly reduce the visual quality of the motion blur.
- Holes can occur in the blur shell when a polygon loses a silhouette edge for a few updates, and then regains it. This is very easily fixed: we add some hysteresis so that polygons may continue to contribute a (faded) shell over a few frames before being removed. That is, we continue to extend the blur shell even though no new updates are occurring. Coupled with the next item, the look is smooth.
- When a polygon has not contributed an edge to the SoM in a number of updates, we stop rendering the polygon's strip. When the strip stops being rendered, there could be a noticeable popping from one frame to the next. To avoid this visual artifact, we fade the entire polygonal strip in proportion to the number of updates since the polygon last had an edge on the SoM, making it disappear smoothly.

- For highly concave objects, it is possible (in particularly bad, though rarely seen configurations) that the blur shell attached to the concave section will "pierce" the back of the object, when this concave portion would never be seen (and thus not need to be blurred). The same is true if the model contains polygons enclosed in the interior. Both of these can be fixed by selective polygon blurring, tagging certain polygons so that they will not create blur shells. In key applications, it is likely that a selective choice of polygons to possibly have blurs would be desirable (and this would require only one bit per polygon), to give greater artistic control over the look of the motion blur to the designer.
- Currently, backface culling is turned off when rendering the blur polygons. This guarantees that all blur polygons will be visible. It does not produce the best possible visuals, but it does produce them quickly. A solution would be to have the blur polygons always face away from the edge that created it. Unfortunately this causes areas of the object to appear unblurred from certain directions. Another option would be to render just the blur, with occlusion culling turned on, to another buffer, and then to combine that blur buffer with the normal rendered image to integrate the blur effect. This effect is now possible with graphics hardware, and easily incorporated into multipass rendering, though we have not implemented it. Yet another option would be to render the blur polygons front to back, to occlude blur polygons that are behind others. There are a number of problems with doing this, with the primary issue being performance. The method can generate many blur polygons, and sorting a large number of these each frame is entirely too slow for real-time applications.

4 IMPLEMENTATION AND RESULTS

We have performed experiments using a Windows® PC with an AMD® Althon® XP 2200+ 1.8 GHz processor with 1 GB of main memory, and a GeForce 4 4400 graphics card. We first show results for a deforming mesh attached to an animated character skeleton; then we compare results for various resolutions of spheres, in order to reduce the number of outside speed impediments inherent in our animation system.

Note that the performance of our method depends on several factors that are difficult to examine exhaustively. These include the complexity of the moving objects, the performance of the graphics hardware, implementation optimizations, whether the application is CPU-bound or rendering-bound, etc. Our current implementation has not been optimized for hardware, e.g. we do not use vertex buffers. However, an understanding of the complexity of our method can still be obtained from examining some example cases.

We implemented the method described into an animation application co-developed by one of the authors. The application allows a user to create a skeletal-bound deforming polygonal mesh. The examples below all run at reasonable speeds (>60 fps), and we present relative performance loss (in this system) due to the inclusion of motion blur, as straight fps offers little insight. The example mesh we will use in the first portions of these results is a 1075 polygon mesh attached to a skeleton with 20 articulated joints. For the blur, we chose for the blur to be 0.2 seconds in length and to have 10 subdivisions, a fairly high quality setting.

To measure performance loss by our method we use a character animating through a walk cycle without our method in use. We then activate only the SoM detection code, without displaying the motion blur. Finally, we run our code with the full motion blur effect. For each, we give the percentage of the frames per second lost when compared with the results when not using our motion blur method.

Model Polys	SoM Edges	Blur Shell Polys	SoM Detection Performance Loss	SoM and Rendered Blur
1075	212	3566	0.37%	1.03%

Table 1: Performance results on an animated character. The SoM edges and Blur Shell polygons columns are approximate due to rapid changes in the SoM while the character animates.

As shown by this example, the largest loss in performance is from the rendering of the blur shell. While this seems to hold true for low polygon models, we will show that as the polygonal complexity of a mesh increases, the SoM detection portion of our method comprises a majority of our performance loss.

Now we will discuss results when examining a set of non-deforming spheres. Each sphere is of varying polygonal complexity, so we are able to see the differences in performance over a range of meshes. Due to the nature of our motion blur method, the lack of a deforming mesh does not change the validity of our results. For consistency, we use the same values for the blur as we did for the character shown above.

As we can see, and as stated above, the SoM detection is where we lose most of our performance when using meshes with a large number of polygons. The blur shell rendering overhead is largely negligible for the high resolution spheres.

Model Polys	SoM Edges	Blur Shell Polys	SoM Detection Performance Loss	SoM and Rendered Blur Loss
528	30	479	0.002%	.15%
1224	47	615	0.55%	.60%
2484	66	828	0.77%	.79%
5670	218	2920	0.89%	.91%
9800	336	4362	0.94%	.95%

Table 2: Performance results on non-deforming spheres. The SoM edges and Blur Shell polygons columns are approximate due to rapid changes in the SoM while the sphere animates.

Even though we show results when using spheres, we feel that the animated character is more representative of the types of deforming meshes found in actual applications. Even so, we only see approximately a 1% performance loss when using our method. In section 5 we briefly discuss modifications that can be made to an implementation to improve performance. Due to this, we feel that our method is suited for real-time graphics.

It is important to discuss the source of the performance losses we do see. Step 1 of our method, finding the SoM, is a CPU bound operation; while step 2, rendering the blur, is mostly a set of graphics hardware operations. For modern graphics hardware, the rendering of a few thousand extra polygons is often a near negligible performance hit. This is reflected in our results: when rendering the high polygon spheres we see little change in performance. Since SoM detection is a CPU bound operation, we would expect that for the larger spheres, the performance loss for SoM detection would grow. We see this in our results, as we lose most of our performance in step 1. Thus, as the model complexity increases, we expect step 1 will take the majority of our method's computation time. In the future, however, we believe that the

entire computation could potentially be shifted to the graphics hardware, eliminating what CPU-bound operations there are currently. We do not feel our method is likely to have noticeable rendering cost unless the application is already rendering-bound, and the blur is applied to most objects.

5 Conclusion and Future Work

We have proposed a method for producing a motion blur in real-time for a deforming polygonal mesh. The method creates a polygonal mesh through the finding and extrusion of the silhouette of motion.

We would like to point out that our method improves on the effects of the current (*geometric* blur based) "best-practice" methods discussed earlier. In particular, we can handle deforming meshes and arbitrary rotation, which those methods do not. Our blur shell does not have to fade – it can be of uniform opacity with replicated models and thus give results the same as these earlier methods. While those other methods can be implemented on current hardware, we believe ours will be just as easily implemented on next-generation hardware.

There are a number of potential future improvements to our motion blur technique. One key such improvement would be to use today's graphics hardware to better manage the blur shell and SoM updates. An example would be to use programmable graphics hardware to detect SoM edges (a rather straightforward computation). This would allow for much faster SoM detection and rendering, and would require less overhead when communicating with the graphics pipeline. Current graphics hardware does not support our entire approach – in particular, the generation of vertices that lie on the blur shell is not supported (unless an array of such vertices were allocated in advance, which would waste vertices, and negate our advantage of having a relatively small blur shell). However, we believe that future hardware advances, particularly the coming WGF standard, will be likely to support this method, allowing all calculations to occur on the graphics card.

An extension to our method would be to restrict the blur to only affect certain portions of the mesh. This would allow for selective blurring, in addition to fixing the enclosed polygon issue discussed in section 3.4.

In addition, an object's mesh could be managed in such a manner as to be of more benefit to our SoM detection algorithm. The algorithm works on any generic polygonal mesh, but it should be possible to tailor a mesh to better suit our needs. One approach would be to pre-compute associations between edges and adjacent vertices; allowing for a faster SoM calculation.

We believe our scheme provides a visually appealing motion blur for an arbitrarily deforming mesh that can be used to enhance realism, add special effects desired by a designer, or to show motion in a still image. This is done in such a way as to allow for performance and quality trade offs that are quite needed in today's real-time graphics applications.

5.1 Acknowledgement

This work was supported in part by NSF grant CCR-0220047.

References

[1] Abdel-Malek, K., Blackmore, D., and Joy, K. 2003. Swept Volumes: Foundations, Perspectives, and Applications. submitted to *International Journal of Shape Modeling*, http://www.engineering.uiowa.edu/~amalek/papers/swept-volume-review.pdf

[2] Brostow, G. and Essa, I. 2001. Image-Based Motion Blur for Stop Motion Animation. *Proceedings of Siggraph 2001*. 561-566.

[3] Dachille, F., and Kaufman, A. 2000. High-Degree Temporal Antialiasing. Proceedings of Computer Animation, 49-54.

[4] Dippe, M., and Wold, E. 1985. Antialiasing Through Stochastic Sampling. *Computer Graphics (SIGGRAPH '85 Proceedings)*, vol. 19, no. 3, 69-78.

[5] Glisse, M. 2003. *On the Theoretical Complexity of the Silhouette of a Polyhedron*. Thesis, INRIA Lorainne, Nancy.

[6] Haberli, P., Akeley, K., 1990. The Accumulation Buffer: Hardware Support for High Quality Rendering. *Proceedings of SIGGRAPH '90*. 309-318.

[7] Kim, Y., Varadhan, G., Lin, M., Manocha, D. 2003. Fast Swept Volume Approximation of Complex Polyhedral Models. *Proceedings of ACM Symposium on Solid Modeling and Applications*. 11-22.

[8] Korein, J, and Badler, N. 1983. Temporal Anti-Aliasing in Computer Generated Animation. *Computer Graphics (SIGGRAPH '83 Proceedings),* vol. 17, 377–388.

[9] Max, N. and Lerner, D. 1985. A Two-and-a-half-D Motion Blur Algorithm. *Computer Graphics (SIGGRAPH '85 Proceedings),* vol. 19, no. 3, 85-93.

[10] NVidia. 2004. Developer Relations Notes for NVidia SDK 8.0. http://developer.nvidia.com/object/sdk_home.html

[11] Potmesil, M., and Chakravarty, I. 1983. Modelling Motion Blur in Computer-Generated Images. *Computer Graphics (SIGGRAPH '83 Proceedings),* vol. 17, 389-399.

[12] Shreiner, D., Woo, M., Neider, J., Davis, T. 2004. *OpenGL Programming Guide*, *Fourth Edition,* Addison-Wesley.

[13] Tatarachuk, N., Brennan, C., Isidoro, J. 2003. Motion Blur Using Geometry and Shading Distortion. in *Shader X^2 – Shader Programming Tips and Tricks with DirectX 9.* Wordware Publishing.

[14] Wloka, M., and Zeleznik, R. 1996. Interactive Real-Time Motion Blur. *The Visual Computer*, vol. 12, no. 6, 283-295.

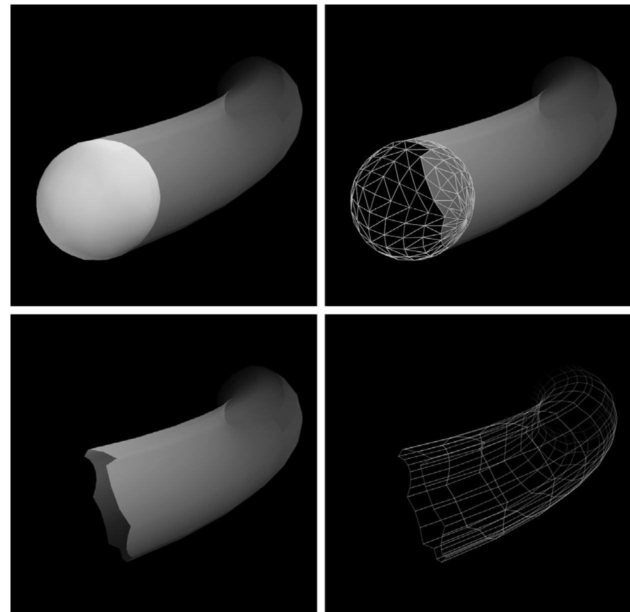


Image 1: (Upper left) Sphere with motion blur. (Top right) A wireframe view of the sphere and its blur. It is easy to see the SoM as it is the set of edges that connects to the blur polygons. (Lower left) The isolated blur shell. (Lower right) The blur shell viewed in wireframe.