# REAL-TIME GEOMETRIC MOTION BLUR

# FOR A DEFORMING POLYGONAL MESH

A Thesis

by

NATHANIEL JONES

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2004

Major Subject: Computer Science

# REAL-TIME GEOMETRIC MOTION BLUR

# FOR A DEFORMING POLYGONAL MESH

A Thesis

by

NATHANIEL JONES

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

| | |
|---|---|
| John Keyser<br>(Chair of Committee) | Donald H. House<br>(Member) |
| Jianer Chen<br>(Member) | Valerie Taylor<br>(Head of Department) |

May 2004

Major Subject: Computer Science

# ABSTRACT

Real-Time Geometric Motion Blur for a Deforming Polygonal Mesh. (May 2004)

Nathaniel Jones, B.S., Texas A&M University

Chair of Advisory Committee: Dr. John Keyser

Motion blur is one important method for increasing the visual quality of real-time applications. This is increasingly true in the area of interactive applications, where designers often seek to add graphical flair or realism to their programs. These applications often have animated characters with a polygonal mesh wrapped around an animated skeleton; and as the skeleton moves the mesh deforms with it. This thesis presents a method for adding a geometric motion blur to a deforming polygonal mesh. The scheme presented tracks an object's motion silhouette, and uses this to create a polygonal mesh. When this mesh is added to the scene, it gives the appearance of a motion blur on a single object or particular character. The method is generic enough to work on nearly any type of moving polygonal model. Examples are given that show how the method could be expanded and how changes could be made to improve its performance.

# DEDICATION

to my family, friends, and colleagues

# TABLE OF CONTENTS

Page

# LIST OF TABLES

Page

# LIST OF FIGURES

Page

# 1 INTRODUCTION

This thesis describes a method for generating a motion blur for an animating and deforming polygonal mesh.

## 1.1 Motivation

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

One is to add realism to a scene. The open camera shutter samples the world over a span of time. This creates a motion blur in any moving object in a film or traditional photography; it is inherent to the filming process and the blur feels natural to most viewers. This motion blur is often missing in real-time applications due to the difficulty or speed loss when creating it. While motion blur is rarely noticed by viewers, its lack is quite noticeable when one compares two computer generated films, one with motion blur and one without; the film with motion blur will appear to be much more realistic and appealing to the viewer. If we can add motion blur to a real-time application, then the viewer will be drawn into the scene more so than if the blur was lacking. Therefore, motion blur is often something designers want in their real-time applications.

---

This thesis follows the style and format of *Proceedings of SIGGRAPH 2002*.

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 something in the scene. By controlling the length of time the blur remains visible as well as display properties of the motion blur itself, a designer can get varying effects that may not be attainable using other methods, some of which are described in section 2.

A third use is to show an object in motion using just a still image. When viewing a still image from a frame set with no motion blur it is often quite difficult, if not impossible, to determine what motion is happening in a scene. Because of this, motion blur can be useful in still media such as comic books and digital imagery, or to just show what is happening within a scene using a single image.

There are many approaches to generating or showing motion blur; but most are insufficient to allow for all of these needs. They either produce results that are visually unacceptable, are not fast enough for real time, or do not deal with highly deforming meshes as can often be found in today's interactive applications. These other methods are discussed in section 2.

### 1.2 General Concept

The motion blur method presented in this thesis satisfies the needs of these uses while providing visually appealing results fast enough for use in real-time applications. Using a method similar to that of swept volume construction, the method first determines the set of edges that lie along the dividing point between the front and back of a mesh based upon its motion; this set of edges is known as the *silhouette of motion*. By connecting a

set of silhouettes captured at regular time intervals with increasingly transparent polygons, the method creates a visually appealing polygonal motion blur. This generated set of polygons is referred to as the *blur shell*.

### 1.3 Key Features

There are a number of features of the presented method that distinguish it from some of the other motion blur techniques in use today.

The method discussed here produces a motion blur for meshes that are animating and deforming. A number of methods that are targeted towards interactive applications make an assumption that the object's polygonal mesh is static. At one time this was a valid assumption, but in today's applications character meshes are often attached to a skeleton and as the skeleton animates, the mesh deforms to match the skeleton's motions.

The method gives designers quite a bit of control over how the blur looks and behaves, as well as the performance cost of a particular implementation. A designer could turn on or off the blur on selective portions of the mesh's surface or vary other attributes associated with the blur, such as its length. In addition, since the method outputs polygons, any polygonal operations can be done on the blur to add additional effects and control, such using a pixel shader to augment the appearance of the blur's polygons. It is also quite possible to use the presented technique to show motion in a single frame capture, which can help audiences figure out what is happening within a scene.

Even when the method is implemented only in software, there is little loss in performance and this loss can be largely alleviated by using today's programmable graphics hardware. Since the method manipulates and creates polygons, which are entities for which graphics hardware is adept at using, the method is quite hardware friendly.

Due to the generic nature of the method, it should be relatively simple to integrate it into most real-time graphical applications that have actors or objects made from deforming polygonal models.

### 1.4 Layout

The remainder of this paper is laid out as follows.

Section 2 will describe the previous work in the topic of motion blur creation, and why we would need an additional technique to the ones described. Section 3 will describe the method's algorithm in detail, as well as describing potential improvements to an implementation. In section 4, experimental performance results are examined by using an implementation of the method, and the restrictions put on the method by computer resources. Section 5 describes the potential future of this method, and where additional work should be focused upon. The paper is concluded in section 6.

# 2 PREVIOUS WORK

Since motion blur is such an important topic within graphics, there has been a good deal of research into the topic; some of this previous work is discussed here.

## 2.1 Offline Motion Blur Techniques

Motion blur has been an important aspect of computer generated imagery for many years. Some of the earliest and most fundamental work includes that of Korein and Badler, Potmesil and Chakravarty, and Dippe and Wold [Korein and Badler 1983; Potmesil and Chakravarty 1983; Dippe and Wold 1985]. The best looking and most widely used motion blur technique is temporal anti-aliasing [Korein and Badler 1983; Dachille and Kaufman 2000]. It is also the technique for adding motion blur to the computer graphics found in films. By this method, motion blur is created by super-sampling the scene in the time domain. What this means is that a particular pixel is sampled at various points in time around the current frame time; and the samples are then filtered to create the final color for a pixel. This essentially averages a pixel's color over a span of time. While this works well for offline rendering or ray-tracing, where time can be extensively controlled, it is far too slow for real-time applications.

The closest method for doing temporal anti-aliasing with current graphics hardware is to use the accumulation buffer to store past frames. The buffer and the current rendered frame are filtered together to produce an image that will approximate temporal anti-aliasing. One issue with this is that if an object is moving quickly, the accumulation

buffer technique will likely produce image ghosting as previous frames might have the object in a very different location than its position in the current frame.

A number of techniques have been proposed for creating motion blur in a 2D image space by applying post production techniques or by tracking objects in image space. These are usually too slow for real-time applications and do not use current hardware to achieve their results [Max and Lerner 1985; Browstow and Essa 2001]. As discussed in section 3.3.1, an attempt was made to use the method similar to the one presented in this paper to produce a motion blur in image space. This produced unacceptable results for a number of reasons discussed in section 3.3.1.

## 2.2 Real-Time Motion Blur Techniques

A common, though waning in popularity, method for creating motion blur is object cloning. This technique copies the last *n* frames of an object's motion into the scene, with each copy being slightly more opaque than the previous, until the current object is added to the scene at full opacity. There are a number of problems with this. One 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; and all but one of the objects must be rendered transparently. A second is that when the object picks up speed, the distance between the different copies will increase. As seen in Figure 1 there is no coherence between the copies and the visual result is unacceptable. This method is only acceptable for low polygon objects moving a slow speed. Unfortunately, objects that are moving slow are in little need of a motion blur.

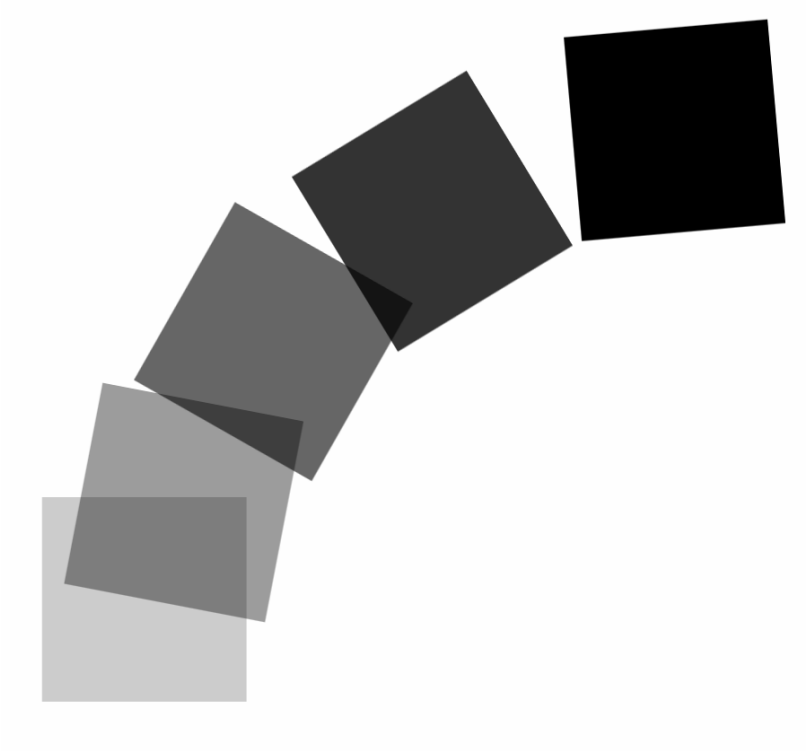

Figure 1. Object cloning of the past four frames of an animating square

A method proposed by Wloka and Zeleznik [Wloka and Zeleznik 1996] simulates a sweep style motion blur by physically disconnecting the front portion of an object from the back portion, and connecting the two halves with polygons that connect the two sections; this process is diagramed in Figure 2. As can be seen in the figure, to simulate the object's motion the connecting polygons are segmented, with all segments lying along the motion path between the current frame, where the front portion is placed, and the previous frame, where the back portion is positioned. A downside to this approach is that it does not consider an object using a deforming mesh; 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 meshes wrapped around an animated skeleton.

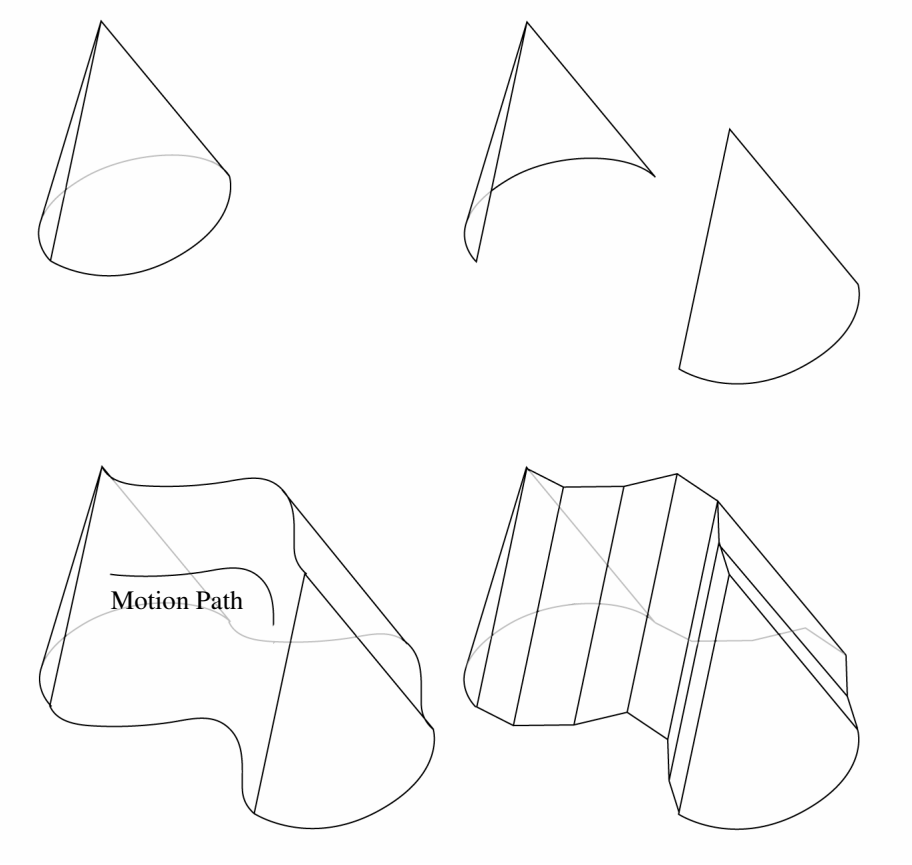


Figure 2. A graphical summary of the method presented by Wloka and Zeleznik [Wloka and Zeleznik 2001].

A geometric motion-based method along similar lines has also been proposed by NVidia [NVidia 2001]. Using a single previous frame, their method generates a motion vector, subdivides the model into halves, and creates an alpha-faded stretch between the two halves as seen in Figure 3. This is easily and quickly done using real-time hardware; however, the method is not very extensible, cannot produce a curved blur, and the blur result is of relatively poor quality. It is quite possible that NVidia has solved these issues using their newest hardware and programmable buffers, and has yet to share their findings with the community.

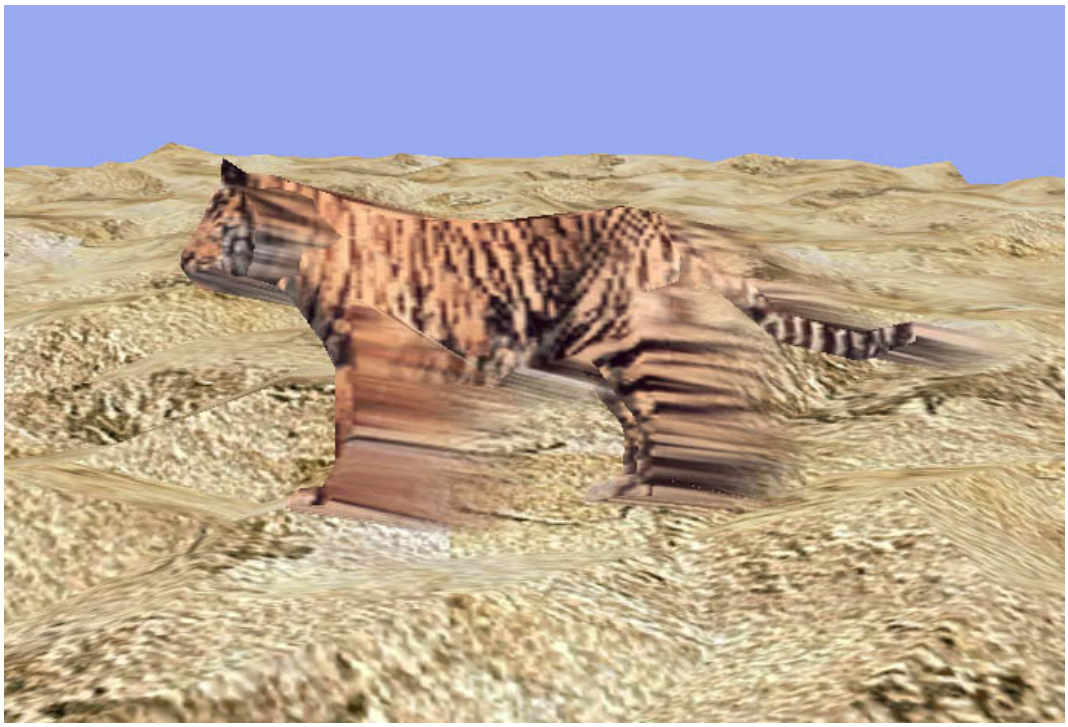

Figure 3. An image of NVidia's motion blur (found on their website) [NVidia 2001].

# 3 ALGORITHM

## 3.1 Previous Algorithm and Work

The method's first incarnations came about in a survey of various ways to use polygons to show the motion of a translating object. The focus of this early research was to determine which of a number of geometric motion deformations produced the most pleasing results, and which performed the best. These first set of methods used the set of edges between the front and back sides of motion to produce effects based on the objects motion. The four methods are discussed below.

As seen in Figure 4, the first of these methods was the most simple. The method simply left the back half of the object at the position it had in the last displayed frame, and moved the front half to the current position. The overall look was to stretch the object; as expected, the visual effect was quite poor.

The second method added alpha blended quadrilateral polygons to the model, with one edge of each polygon attached to the edges of the silhouette, and the opposite edge at the position the corresponding edge had in the last frame. Of the methods examined, this proved to give the best results, and it was quite fast to compute and to produce the effect for. It was slower than the first method, but it produced results that were far superior, as Figure 5 shows. This version of the motion blur is the basis for the remainder of this thesis.

Figure 4. First method from early research. This is a simple stretch across the object's motion silhouette.

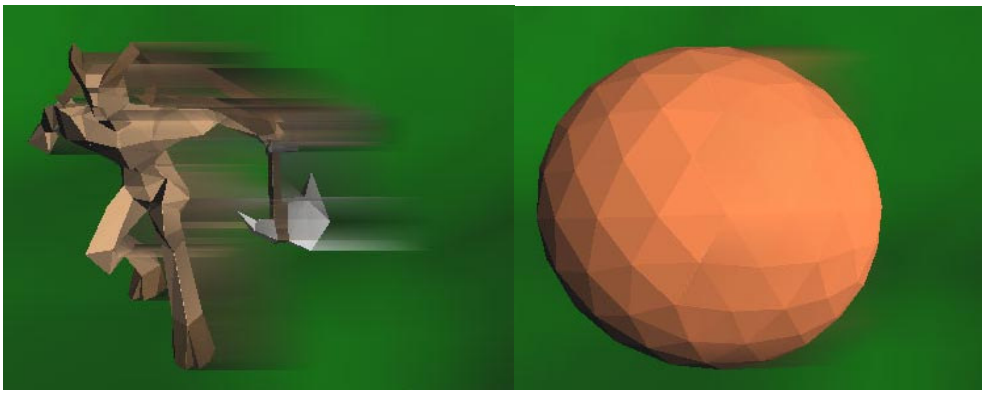

Figure 5. Second method from early research. This is the early version of the method used for this thesis, it attached alpha blended polygons to the motion silhouette.

The third method was quite similar to the first, but this time the back portion of the model was deformed into a parabolic arc, and as seen in Figure 6 this gave the back portion a more rounded look. The visual results were worse than the method it was based on; consequently it was not pursued further.

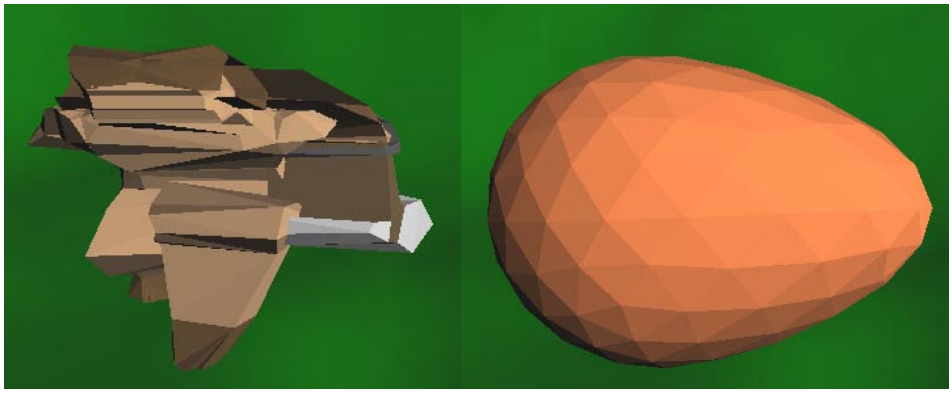

Figure 6. Third method from early research. Similar to the first, this created a stretch, but applied a parabolic deformation to the back half of the object.

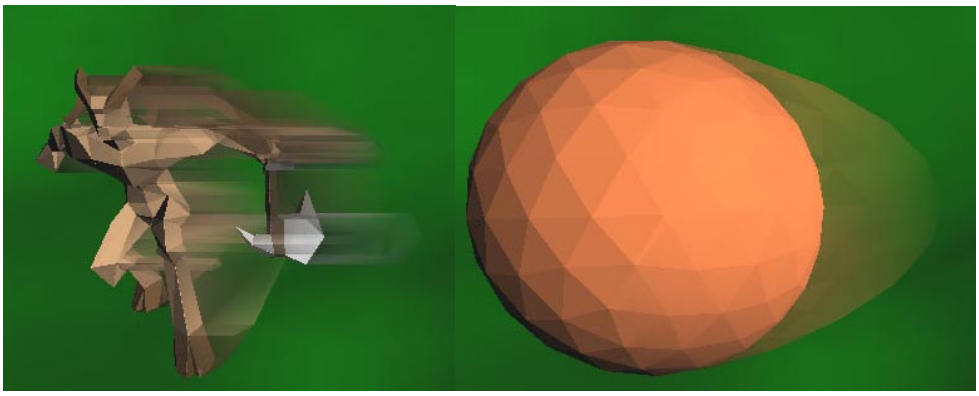

Figure 7. Fourth method from early research. Using a technique combination of the first and third methods, an extra set of polygons in a parabolic shape was added to the model.

The fourth method combined the second and third method into a motion blur. Instead of just adding the polygons to the silhouette edges as in the second method, it would cap off the blur with an alpha-faded and parabolic warped version of the third method as shown in Figure 7. Before starting this research, this was the method judged most likely to give pleasing results. Ultimately this produced an effect that was not as pleasing as the second method and was more computationally and graphically intensive than all the other methods.

None but the second method deserved extra attention, and it had many desirable visual and computational advantages over the other methods. Once this was determined, the second method was expanded to produce a more generic and useful algorithm.

## 3.2 Current Algorithm

The algorithm 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 this method uses to create the motion blur, the blur polygons would give a reasonable approximation of the swept volume the object passed through to get to its current position and location. Swept volume computation is a field of study that is still receiving attention by computer scientists [Abdel-Malek et al. 2003; Kim et al. 2003].

### *3.2.1 Implementation Specific Values*

In essence, the scheme described here 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. To do this, two pieces of information are needed:

- The length of time the blur should represent
- 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; the number of subdivisions directly controls the quality of the final blur. Depending on the effect desired, the time length can vary. To produce a longer

blur, a value of 0.2 seconds was found to give pleasing results, though if one is going for a more realistic blur the time length should be about the same as the time to render a few frames. The number of subdivisions directly controls how many samples of the silhouette of motion, discussed below, will be taken during the time span provided. The more subdivisions, the smoother and more rounded the blur will look, and the more computation will be needed to maintain it.

### *3.2.2 Steps to be Taken*

To produce the blur, two main steps must be taken. The first regularly updates the current silhouette of motion, which is discussed in more depth in 3.2.4. Instead of continually updating the silhouette, it is updated at certain time intervals; the time between these updates is

$$\frac{timeLength}{subdivisions}$$

. However, the best results are obtained when the update can be performed on every frame. Instead of updating every few frames, the system could also update multiple times between frames. This would give the method the ability to sample more accurately over a time frame, and allow this method to gain some of the benefits that exist when using temporal anti-aliasing. The implementation would be slowed according to how often this was updated. The second main step is to build and render the blur, which is the only portion of the method that must be performed on a per-frame basis; the method for doing this is discussed in section 3.2.5.

Using Figure 8 as a visual guide, a walk-through of the generic algorithm will be helpful in understanding the method. The example starts with a sphere in motion, as seen in part a. Next, in part b, the method divides the sphere into two portions. One portion is the area of the sphere that is pointing toward the direction of motion; the other area is the portion of the sphere pointing away from the motion. The dividing line between these two sections is called the silhouette of motion. As seen in part c, the sphere keeps moving, and at each update interval the method recalculates the silhouette of motion. After a number of updates, the method has a short history of the past silhouettes of motion. If we were to imagine connecting the silhouettes together, similar to what can be seen in part d, we can see that a curved tube is formed, and this tube could represent the volume swept out as the sphere moved. To actually create the motion blur, the method connects the silhouettes with a series of polygons. The older the silhouette that is attached to a polygon, the less opaque the polygon is. This can be seen in part e; and at this point the method has done everything it needs to do. By visually removing the silhouettes, the full blur effect can be seen, as shown in part f.

This was a simple example on a non-deforming sphere. The same technique holds if just a single portion of the object is moving while the rest of the object stays still. Only the moving portions are affected by this algorithm.

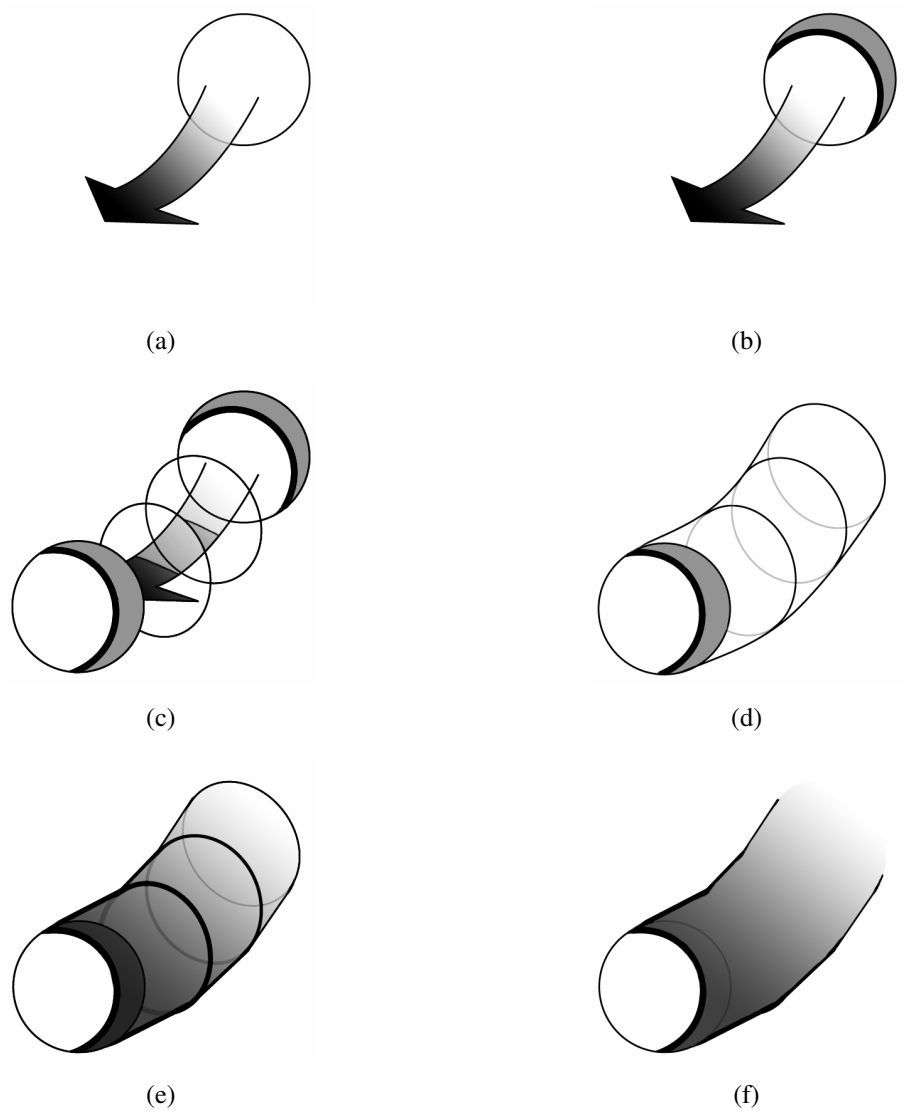


Figure 8. Graphical walk through of the method using a moving sphere. The heavy line on the sphere is the SoM, while the shaded portion is the half of the object that is facing away from the direction of motion.

### *3.2.3 Requirements*

There are a few assumptions that are made regarding the mesh to be blurred. In order for the method to function properly, these must be met by the implementation. These should be easy to obtain in most real-time applications.

Unlike some other methods, the mesh to be blurred can be deformed in real-time, as by a skeleton, but the world coordinates of each vertex must be known at the time the silhouette is updated. In many cases, this adds little overhead, as most skeletally animated meshes know their vertex positions in relation to the skeletal root. The world coordinates must still be known even if an object's mesh is not deforming, but the object itself is moving. In addition to the world coordinate positions of the vertices, the normal, in world space, of each vertex is also needed when updating; these normals are primarily used to detect the silhouette of motion. The best results were obtained when all vertices' normals were simply the average of the connected polygon's normals. This is especially important on low polygon objects, since a polygon with normals that all face the same direction cannot produce a blur. 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 must include some information needed to maintain the silhouette of motion and to create the blur shell. Associated with each polygon are a number of additional data structures.

The first 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

and contains a flag marking whether or not an edge of the polygon exists on the silhouette of motion, and two vertex positions and normals that represents an edge on the silhouette of motion, if one is found. This list is needed to track the polygon's edges that were on past silhouettes of motion.

Each polygon also needs two indices describing the last two vertices of the polygon that were part of the silhouette of motion. These will allow for the blur to be connected to the original mesh. Also, a value representing the number of updates that have passed since the polygon last had an edge on the silhouette is needed. This will be used to determine if a polygon should stop contributing to the blur.

In addition to the per-polygon values, a global index is used to keep the system synchronized between and during updates, referred to as the *update index*. This index corresponds to which element in a polygon's circular list is the most recent and the index is updated on every silhouette update.

### *3.2.4 Finding the Silhouette of Motion*

The *silhouette of motion*, or SoM, is a term describing 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 that one portion would change, not that of the rest of the mesh. A non-moving portion of an object does not have a SoM. Figure 9 shows a variety of objects and possible silhouettes of motion for the displayed motion.

At each update the current SoM must be found. Due to the difficulty in determining a full-mesh SoM, a per-polygon silhouette detection technique is used. The 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 a close approximation to the actual silhouette.

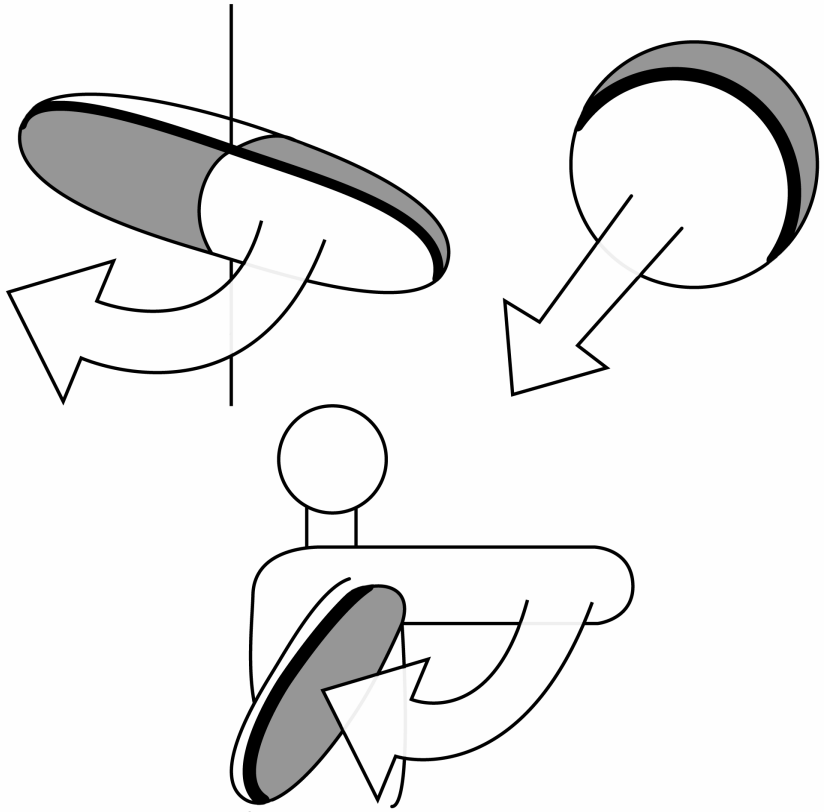

Figure 9. Various silhouettes of motion. The heavy line is the SoM, gray areas are on the back side of motion, and the arrow depicts movement.

Before finding the SoM, the method first determines which side of motion each vertex is on. The facing of any vertex is determined by the sign of the dot product between the world normal of the vertex and the vertex's direction vector. The direction vector points from the vertex's previous position to its current position.

Next the method finds the set of edges that lie on the SoM. 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. An example of this on a set of polygons is shown in Figure 10. A note here is that the opposite method for choosing the silhouette is also valid; a SoM edge could be the edge between two front facing vertices, if the third vertex is back facing. As long as a consistent method is used, either is appropriate. In addition to recording these two vertices, the flag is set in the element, stating that this polygon has an edge to contribute to the silhouette associated with the update index. Also, 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 and will be used to connect the polygon to the blur it will create. In addition, the number of updates since the last time an edge was on the silhouette is reset to zero.

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. These two recorded vertices will be used to seamlessly fill in holes that crop up when displaying the blur; see section 3.2.5. Unless

these polygons have generated a blur within the last few updates, they will not contribute any vertices to this blur update.

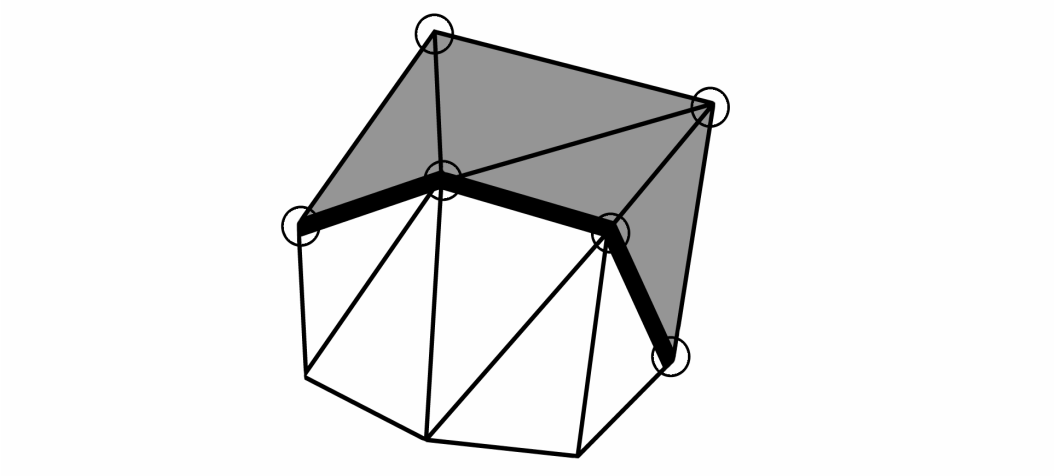

Figure 10. The SoM as found on a polygonal mesh. The heavy line are the edges on the SoM, circled vertices are those with normals facing away from the direction of motion, and the shaded polygons are on the back side of motion.

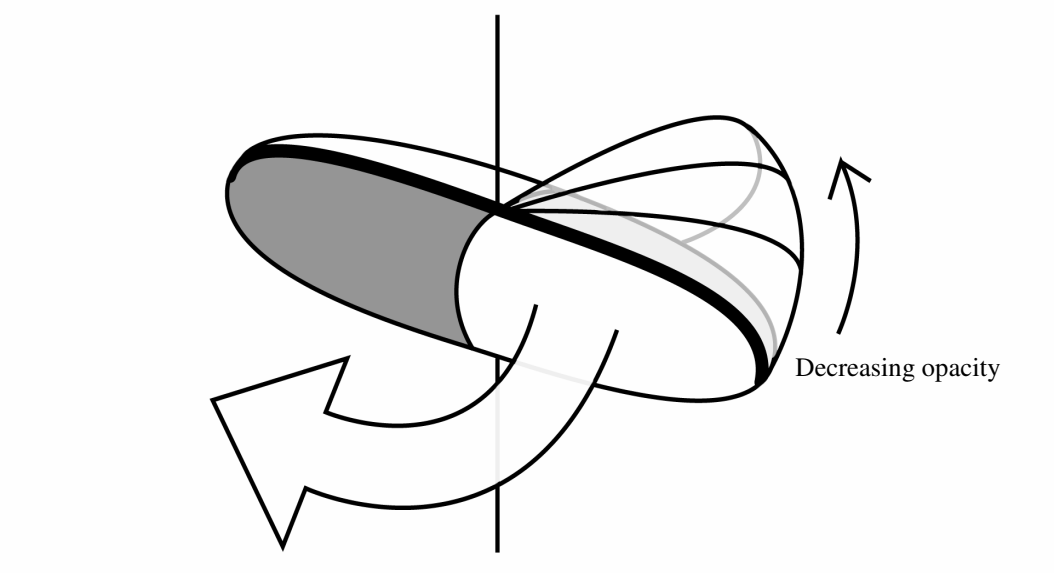


Figure 11. Results of successive SoM updates. The sequence of SoM updates are connected to produce a motion blur.

Once this has been done for all polygons, the method will have a set of polygon edges that defines the silhouette of motion. By collecting these silhouettes over the course of a number of updates, the method will be able to connect them in a manner similar to Figure 11, thus creating the motion blur. We can know that 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 [Glisse 2003]. Thus, for any one update there will be relatively few edges to deal with.

A note here is that by exploiting frame-to-frame coherence, it may be possible to obtain the SoM more quickly. While this option has not been fully explored, it is unlikely to offer much improvement, as silhouettes can emerge on a moving object at

locations far from the previous SoM. Due to the way most objects move, there is a good deal of coherence between the SoM of adjacent frames, but the effort to exploit this would likely add so much overhead as to make the effort futile.

### *3.2.5 Rendering the Blur Shell*

In order to display the blur, the method relies on the use of polygon strips, specifically quadrilateral strips. Each polygon independently creates a polygon strip to display any blur associated with the edges that belong to it. To do so, a number of steps need to be taken.

Using the value stored with each polygon, the method determines 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 time frame contributes no polygons to the blur shell. In addition, if the polygon does not currently have an edge on the SoM, the entire polygon strip will be alpha faded accordingly. As the number of updates since the polygon had an edge on the SoM increases, the alpha value of the entire strip decreases, until it is at full transparency at the time constant specified. This keeps the blur strips from popping out of the scene, by giving just enough transition to smooth the visual impact of the sudden strip disappearance. After this time, the polygon must again have an edge on the SoM in order to display a blur.

If a 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 blur effect, the polygon strip needs to fade out towards the end. To do this the vertices that make up the polygon strips decrease in alpha value, relying on the graphics hardware to blend the blur shell's polygons.

To create the quad strip, the method first adds the mesh vertices that belonged to the edge the polygon last had on the SoM. Then the method cycles through each of the polygon's list elements, starting with the one corresponding to the current update index. The two vertices stored in each list element, with their normals, are added to the quad strip. There, the normals are used to help retain the shading the mesh had at the point the vertices were stored, though if the scene has moving light sources the blur's shading can potentially change. If a number of vertex pairs equal to the number of subdivisions are added, the method stops adding vertices to the strip. 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

$$\frac{initial\_\alpha}{subdivisions}$$

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.

The polygon strip created by each polygon will look similar to that found in Figure 12. Since the silhouette update does not necessarily happen on every frame, the polygon

strip is connected to the original polygon; this allows the polygon to move without having to update the silhouette all the time. If the blur is displayed again before another SoM update, and the polygon has moved since the blur was last rendered, the polygon and strip would look similar to the bottom image of Figure 12, with only the mesh polygon having moved.

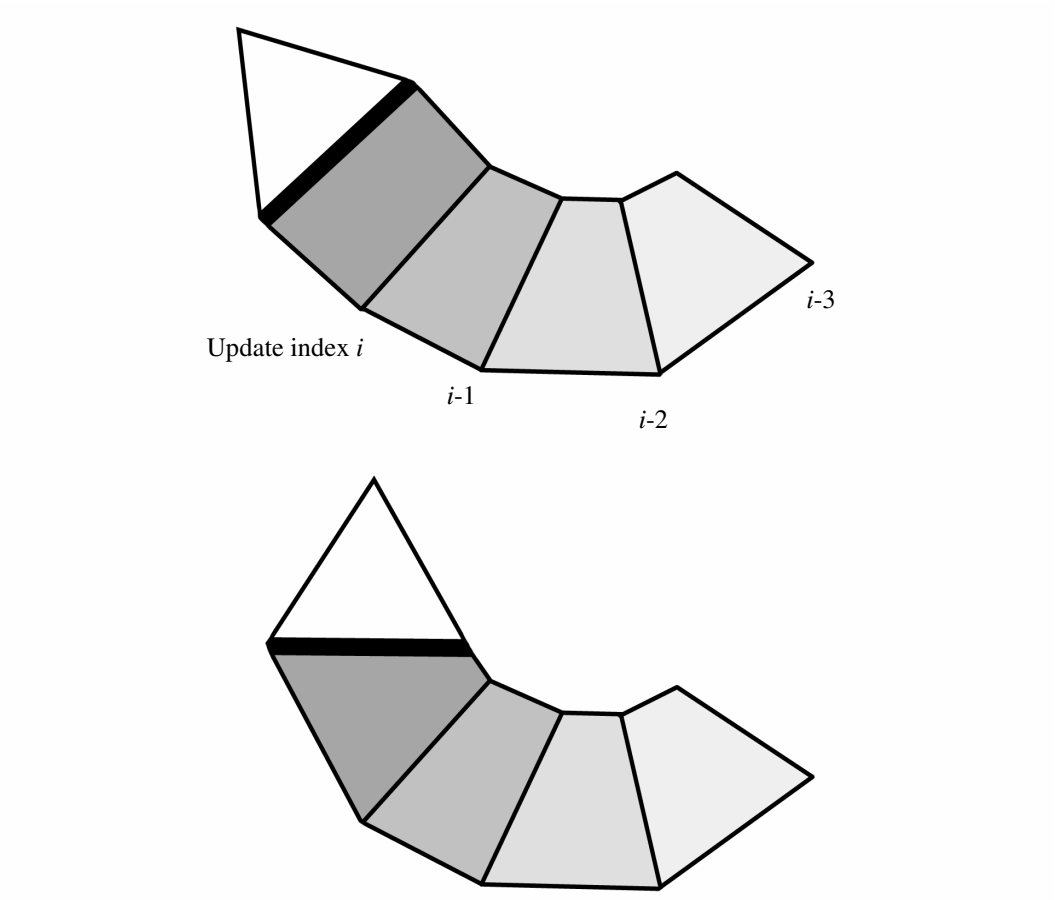


Figure 12. A quadrilateral polygon strip generated by a polygon. Over the course of a few SoM updates each polygon records the positions of an edge's vertices. As these vertices are rendered, their alpha value is controlled to give the strip a faded look.

Once all polygons have generated their polygonal strips, the total set of polygons rendered will create a blur shell that is connected to the original deformed mesh. As stated earlier, the blur shell is what gives the appearance of a motion blur.

At most, each polygon will add a number of triangles equal to twice the number of subdivisions 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(sqrt(n)*subdivisions) quadrilaterals to the scene to create the motion blur. Even so, the number of additional polygons rendered for the blur is likely less than that needed by some of the other motion blur techniques, such as the object cloning method. In practice there are additional polygons rendered due to polygon edges that have not been on the silhouette in several updates, and are currently fading out their associated strips. The speed and amount of model deformation are the prime factors that determine how many of these dieing polygon strips are displayed. The more animated the model is, the more likely it is to have a large number of polygon strips dieing at any one time.

### 3.3 Analysis and Limitations

While the method described gives very good results, there are still several issues regarding the accuracy and visual quality of the blur. Most of these issues do not impact the visual look of the blur much, or can be corrected with ad hoc methods.

There exist cases where the silhouette is not continuous along the mesh. These are rare when using models with continuous meshes, and do not significantly reduce the visual quality of the motion blur. However, if the model is not continuous, there can

appear to be gaps in the blur. This is due to the silhouette of motion crossing this gap in the model, and consequently the silhouette edges are not connected. In any case, there is little that can be done to remedy this; and the effort would be wasted as the issue does not hamper the visual quality much.

Several steps have been taken to reduce the appearance of holes within the blur. Holes are caused when there are gaps within the blur caused by a polygon losing a silhouette edge for only a few updates, and then regaining a silhouette edge, thereby continuing to display a blur. Holes are one of the most visible errors a viewer will see when shown the method in use; but the method patches these in such a way as to be invisible to the user. The primary method for patching the holes is for a polygon to record the position and normal of the last pair of vertices on the SoM, even if the polygon no longer has an edge on the SoM. The method uses this when building the polygonal strip to fill in any hole that is found.

After a period of updates where a polygon has not contributed an edge to the SoM, the polygon's strip will stop being rendered. When the strip stops being rendered, there is a noticeable popping from one frame to the next, as in one frame there was a strip of polygons, and in the next the strip is missing. Most viewers do not see what is happening, but they see the popping artifact and it is unpleasant. To avoid this visual artifact, the method fades the entire polygonal strip in proportion to the number of updates since the polygon last had an edge on the SoM. By the time the polygonal strip should be removed, it will have faded over the course of a few frames to an alpha value of zero.

For highly concave objects, it is possible to have the blur shell attached to the concave section “pierce” the back of the object. The same is true if the model contains polygons enclosed in the interior. While these detract from the appearance of the method, as there are blur polygons that appear to be protruding out of an inappropriate location of the mesh, both of these issues can be partially remedied by selective polygon blurring. With this approach, certain polygons are tagged so they will not create blur shells and bypass the SoM detection. This is also the appropriate addition to allow for interactively turning on and off portions of the model to blur.

A minor issue occurs when the motion blur created is short, and the portion of the object that is creating the blur is thick. What can happen is that if the silhouette of motion runs down the middle of the area viewed, and the blur polygons do not extend far enough beyond the SoM to be seen past the edge of the object, there can appear to be no blur. This is usually not an issue because the artifact is usually only apparent on slow moving portions of the mesh; and slow moving portions usually do not need a motion blur.

When drawing the blur polygons, the method turns off z-buffer checking to keep transparent polygons from occluding other blur polygons that might need to be rendered. This is a well known issue when using transparent polygons in an application that uses the z-buffer. A solution that is usually used to solve this issue is to draw the polygons from front to back. This requires that all polygons be sorted based on their distance from the camera. This can be a very expensive operation due to the large number of polygons

that can be generated. This drastic loss in performance is unacceptable in a real-time application, so this solution is not used in the method.

The largest issue with the proposed method is that it produces a motion blur in world space, not image space. What this means, is that the motion blur is not dependent on the camera's movements. Real motion blur would arise if an object was non-moving and the viewpoint moved such that the object moved in image space. The method described in this paper does not produce this effect; it only produces a blur when the object is in motion. A good example of the difference between what this method does and what is real motion blur, is to imagine that there is an object and a camera looking at it. If the object is spun, but not the camera, a real motion blur would show the blur when viewed from the camera; the method will also do this, since the object is in motion. However if the camera is moved around the object, while the object stays still, a real motion blur would again be seen, but the method described here will produce no such blur. Under most circumstances, the camera is not moving fast enough compared to the speed of the animation that this becomes noticeable. However, if the camera is moving quickly, the fact that the blur is created in world space is noticeable if one knows what to look for.

A potential solution to this would be to track the object's silhouette of motion in camera space instead of using world space. This would allow the object to be blurred when the camera moves; but the ability to show motion blur in a paused scene from any angle would be lost.

Given this however, since the blur is done in world space, the method provides the feature that if the scene's animation is paused, it is still possible to see the motion blur,

as it is just another part of the scene. While the scene is paused, the camera can be moved around, and at nearly any angle and position it is possible to see the motion of the object due to the polygonal motion blur. This effect is not possible using most other motion blur techniques.

At one point in development, an image space motion blur using the method defined here was attempted. While the effort produced unfavorable results, it proved to be a valuable experience.

### *3.3.1 Image Space Motion Blurring*

Throughout the development of this thesis, a number of experiments were made. Some produced successful results, others did not. Those that worked have been integrated into the method.

The largest of the experiments that produced unacceptable visual results was an attempt to use a method similar to the SoM detection given above to produce a motion blur in image, or screen, space. Instead of using a vertex's direction vector to determine the SoM, the direction from the vertex to the camera was used. This way, the silhouette found would be the one seen by the camera. This portion of the algorithm worked, and the new silhouette was found without issue. Once the silhouette was found, the method created the blur on the silhouette edges in 2D image space. This worked fine as long as the camera did not move. However, once the camera started moving around the object, the combination of object deformation, animation, and the changing view would cause

the silhouette to move across the object extremely quickly and unpredictably. This caused severe graphical artifacts that were not reasonably fixed.

Since the polygons were created in 2D image space, after the object had been rendered, there were occlusion issues where the blur generated on edges behind another portion of the model would appear to be in front of the object. In addition to occlusion issues, there were blur shading issues since before the blur was being rendered in world space with world space lighting, now the blur polygons were being rendered in 2D image space, with no world lighting. These, and other, issues can be seen in Figure 13. Due to these problems, little attempt was made to fix the occlusion and shading issue; and it was determined that the method of silhouette detection and blur rendering described in this thesis was inappropriate for producing an image space motion blur.

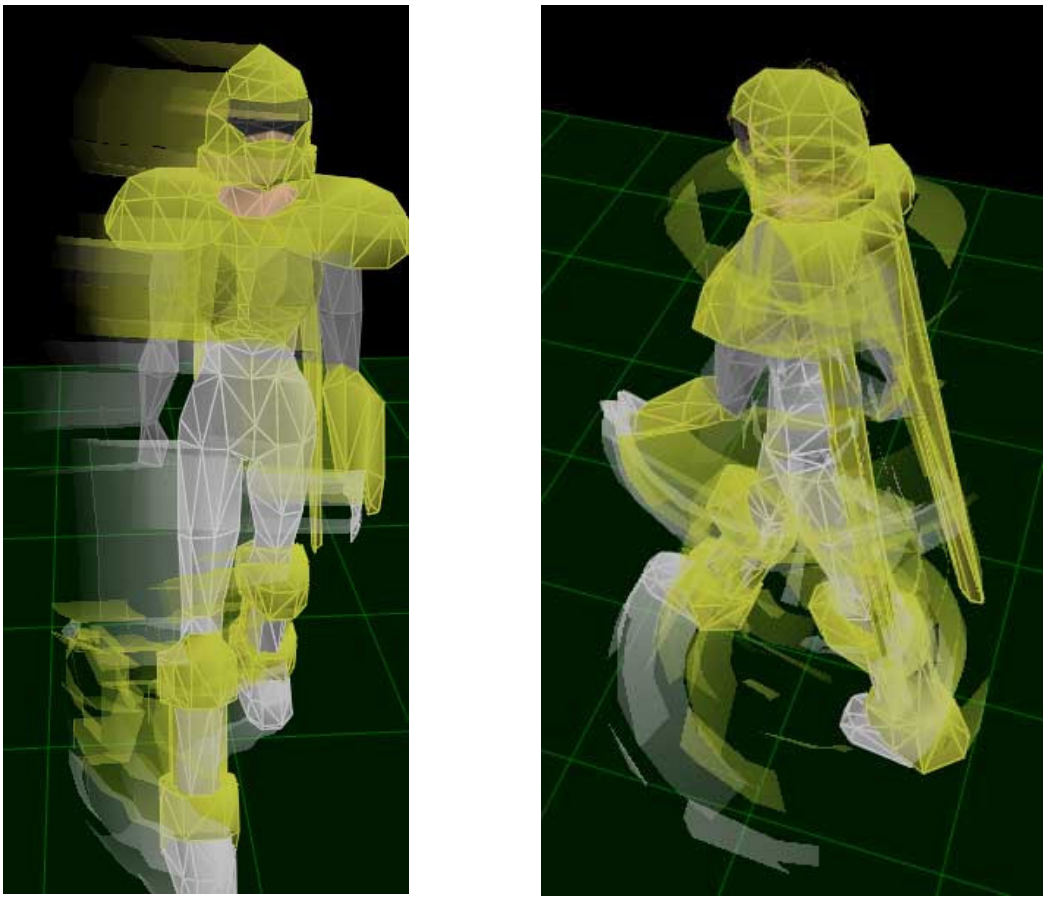

Figure 13. A pair of images from an image space motion blur attempt. Due to a number of issues, this extension to the method was not pursued past a certain point.

# 4 RESULTS

## 4.1 Experimental Results

Experiments have been performed 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. The first results shown are from a deforming polygonal mesh attached to an animated character skeleton; then the results will be compared to the results for various resolutions of spheres, in order to reduce the number of outside speed impediments inherent in the animation system used to house the implementation.

It is important to note that the performance of this method will depend on a number of factors that are difficult to examine exhaustively. These include the complexity of the moving objects, how often the silhouette of motion changes, the performance of the graphics hardware, implementation optimizations, whether the application is CPU-bound or rendering-bound, etc. The current implementation has not been optimized to use hardware, e.g. it does not use display lists or vertex arrays for storing and displaying vertices; it was designed to test the validity of the generic method, not to see how fast this particular implementation could run. However, an understanding of the complexity of the method can still be obtained from examining a number of example cases.

The method has been implemented into an animation application co-developed with Paul Edmondson, a previous student. The application allows the user to define a set of animations for a skeleton, and allows for a polygonal mesh to be bound to this skeleton. As the skeleton animates, the mesh is deformed to match the motions of the underlying

skeleton in a manner similar to that found in today's real-time graphical applications. Unfortunately the application is not designed with speed and efficiency in mind; that was not one of the design goals for the application. Its inefficiencies can skew the timing results, even though the system animates at reasonable speeds, above 60 frames per second for a character with 20 points of articulation and a mesh with over 1000 polygons. Due to these inefficiencies, performance shall not be discussed in actual frames per second for the complete system, mesh deformation and blur; but rather the performance lost while using the method described in this paper against the implementation's performance when not using the method.

The example mesh used in the first portions of these results is a 1075 polygon mesh attached to a skeleton with 20 articulated joints. The blur is 0.2 seconds in length and it has 10 subdivisions, a fairly high quality setting causing the SoM to update every $1/50^{th}$ of a second, or nearly once every frame.

To measure performance loss by this method, a character animating through a set of animations, without the method turned on, is used. Then, only the SoM detection code, without displaying the motion blur, is activated. Finally, the system is run with the full motion blur effect enabled. For each, the percentage of the frames per second lost when compared with the results when not using the motion blur method is given. These results are summarized in Table 1. The values given for polygon counts in the tables are averaged values, as the number of polygons displayed and the number that contain an edge on the SoM change radically throughout an animation.

Table 1.
Performance Results on an Animated Character

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

As shown by this example, the largest loss in performance is from the rendering of the blur shell. As stated earlier, this is mostly caused by the dieing strips created on a highly deforming object. It will be shown that as the polygonal complexity of a mesh increases, the SoM detection portion of the method will comprise a majority of the performance loss.

Now results from a set of non-deforming, but animating, spheres are examined. The spheres are of increasing polygonal complexity, so that it is possible to see the differences in performance over a range of meshes. Due to the nature of the motion blur method, the lack of a deforming mesh does not change the validity of the results. For consistency, the values for the character's blur will be the same values used for the spheres.

Table 2.
Performance Results on Non-Deforming Spheres

| Model Polygons | SoM Edges | Blur Shell Polygons | SoM Detection Performance Loss | SoM and Rendered Blur Performance Loss |
|---|---|---|---|---|
| 1224 | 47 | 615 | 0.55% | 0.60% |
| 2484 | 66 | 828 | 0.77% | 0.79% |
| 5670 | 128 | 2920 | 0.89% | 0.91% |
| 9800 | 336 | 4362 | 0.94% | 0.95% |

As can be seen in Table 2, and as stated above, the SoM detection is where most of the performance is lost when using meshes with a large number of polygons. The blur shell rendering overhead is largely negligible for the high resolution spheres.

Even though results are shown when using spheres, the animated character is much more representative of the types of deforming meshes found in actual applications. Even so, there is only approximately a 1% performance loss when using the method described. Section 5 discusses modifications that can be made to an implementation to improve performance. Due to the small loss in performance when using this method, it is well suited for real-time graphics.

## 4.2 Resource Usage

When discussing a new algorithm or method and its impact on performance, it is important to discuss what computer resources are used and required by said algorithm; and to analyze ways to reduce the requirements of these resources.

As such, it is important to examine the sources of the performance loss seen in the results in section 4.1. For a graphical application, there are two primary resources that are needed and consumed: the main CPU and the graphics hardware of the machine.

In the implementation used to gather the results in section 4.1, finding the SoM was strictly a CPU bound operation. All calculations needed are performed by the CPU, and all data is managed by the main memory of the computer. By looking at the results for the high polygon objects, we can see that the method spends most of its time calculating the silhouette of motion. Due to this, the bottleneck of the method is in the SoM calculation, and since it is currently managed by the CPU, the SoM detection is a CPU bound operation. The overall method performance could be best improved by examining ways to reduce the CPU cost to detect the SoM.

It should also be noted that the animation program used in these results is also CPU bound; the mesh is deformed in software. By using the method, the number of polygons in the object can be increased by 50% to 400%. If the application was rendering-bound, and hardware bound, then this increase in the number of polygons would have a larger impact on performance. However, since the performance loss due to the blur polygons being rendered is so low, we can see that the animation system is CPU bound.

In order to free up the CPU, it should be possible to offload most of this processing onto the graphics hardware; though currently the storage of the SoM tracking data structures would need to be done on the non-graphics hardware side. This will likely change in the near future as hardware buffers become more generic. Until the time when all of these calculations and data storage can be done on the graphics hardware, there would need to be communications between the GPU and the CPU; and as the industry has found out, this communication can be time intensive. So while this is a viable direction of research, it will be a while before an optimal solution arises.

In the second portion of the algorithm, the method renders the blur using the set of edges found on the SoM. Currently the application running on the CPU generates the geometry, and the graphics hardware renders it to the display buffer. For modern graphics hardware, rendering a few thousand extra polygons is often a near negligible performance hit. However, all the blur polygons are transparent; causing many pixels to be rendered many times, so high fill rate graphics hardware is needed to keep up performance. Luckily, today's hardware is specifically designed to have a high fill rate. While the display itself is fairly cheap for the hardware, generating the geometry in the first place is not. Since the polygon strips are created by code running on the CPU, there is a resource bottleneck here.

There are a number of ways that could be used to improve the geometry generation time. One way would be to use vertex arrays to manage the blur polygons. By using a data structure that is tightly linked to the graphics hardware, it should be possible to reduce the calls to vertex creation functions by setting values within a vertex array.

Another option would be to try and move the entire set of operations onto the graphics hardware, and to generate the polygons using some sort of shader.

Using the implementation results above as indicators we can see that as the model complexity increases, the SoM will take the majority of the method's computation time.

### 4.3 Performance and Quality Tradeoffs

When looking at the algorithm, one can see that there are many choices for a developer to make when using this method. Such choices are how many subdivisions the blur should have, how long the blur should last, etc. While a lot of these choices are purely stylistic in nature, there is a practical side to making them. By manipulating the variables associated with the method, it is possible to trade run time performance with image quality.

Aside from moving the implementation to be based more in hardware, the most important decision a developer using this method can make is how many subdivisions the motion blur should have. The higher this number, the smoother the blur; but the more computation time needed. Since this value determines how often the SoM is updated, and as stated above, the SoM detection is the most expensive set of operations done, it can be the most powerful customization tool for this method. If the update speed is too high, then the method will spend more time than necessary to determine the SoM. If the update speed is too low, the blur will appear blocky and unattractive. The best visuals are obtained by having the SoM updated at least on every frame; but in some applications this may require too much processing. Due to this, it is in the application

designer's best interest to carefully determine the number of subdivisions that is most appropriate to their application. This said, it is likely possible to modify the method to allow for on-the-fly modification of the subdivisions or SoM update interval.

### 4.4 Visual Results

Figure 14 shows the method in use on a character walking in place. On the left is a full body image showing the motion blur across the entire character. Notice that even in this still image, it is possible to tell how the character is moving. The top right image is a close up of the blur associated with the legs. Here, the blur is rendered in wire frame so that the individual polygons of the blur strips can be seen. The bottom right image is of the same frame of animation, but this time the model is shown in wireframe, making it easier to see how the blur connects to the model.

In Figure 15 a series of views from the same frame of animation is shown. Notice how the blur easily shows the curved motion in the arms and feet. Also, when compared to Figure 14 it is possible to tell that in this image, the character is moving faster. The left image is the standard view of the method in use. In the middle the blur is rendered in wireframe so that the blur polygons can be seen. The right is again the same frame, but the model is in wireframe so that the blur is easily distinguishable from the model. In this image it is easy to see how the blur polygons attach to the thighs of the character.

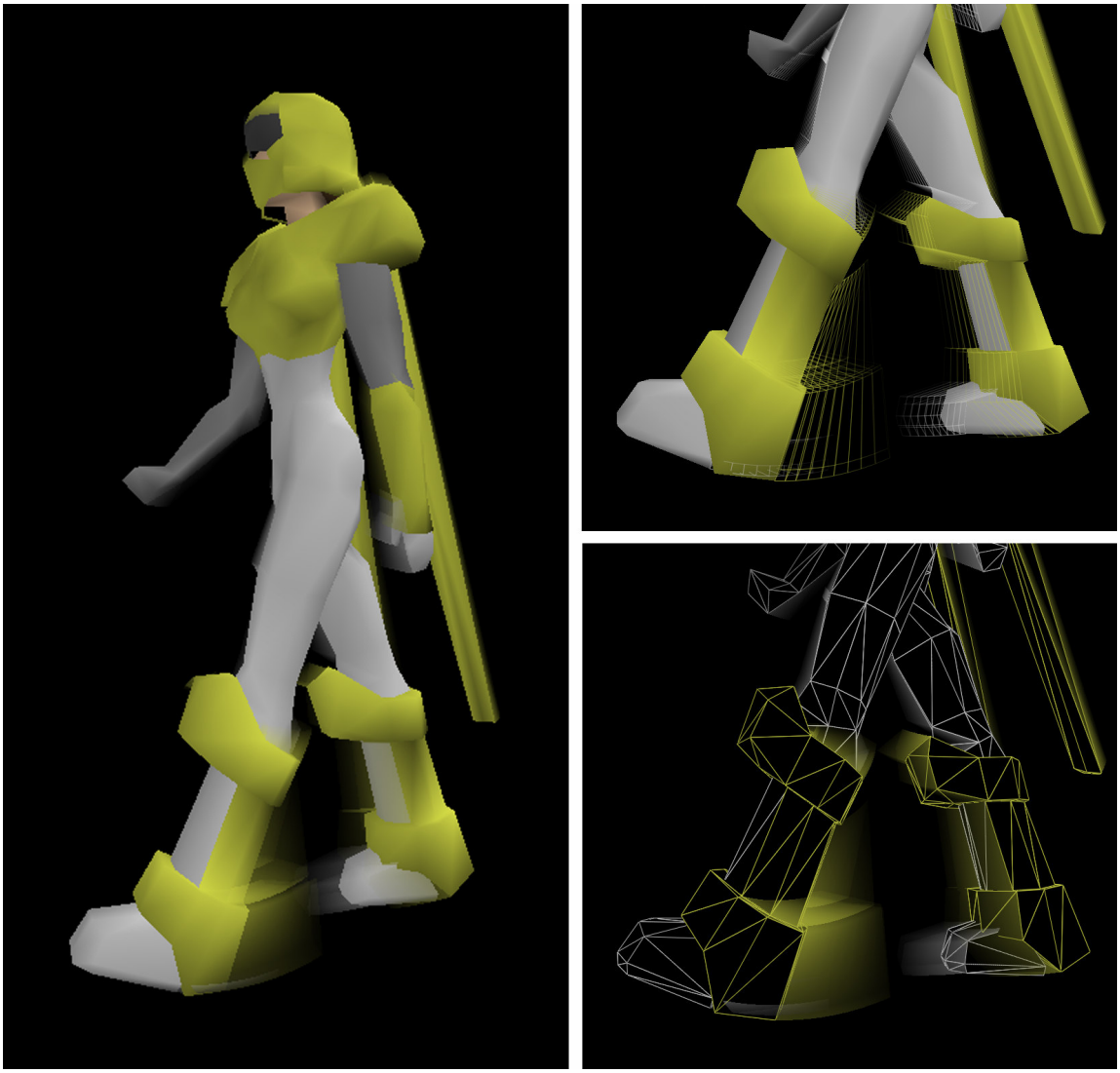

Figure 14. A view of a character's legs using the motion blur technique found within this thesis

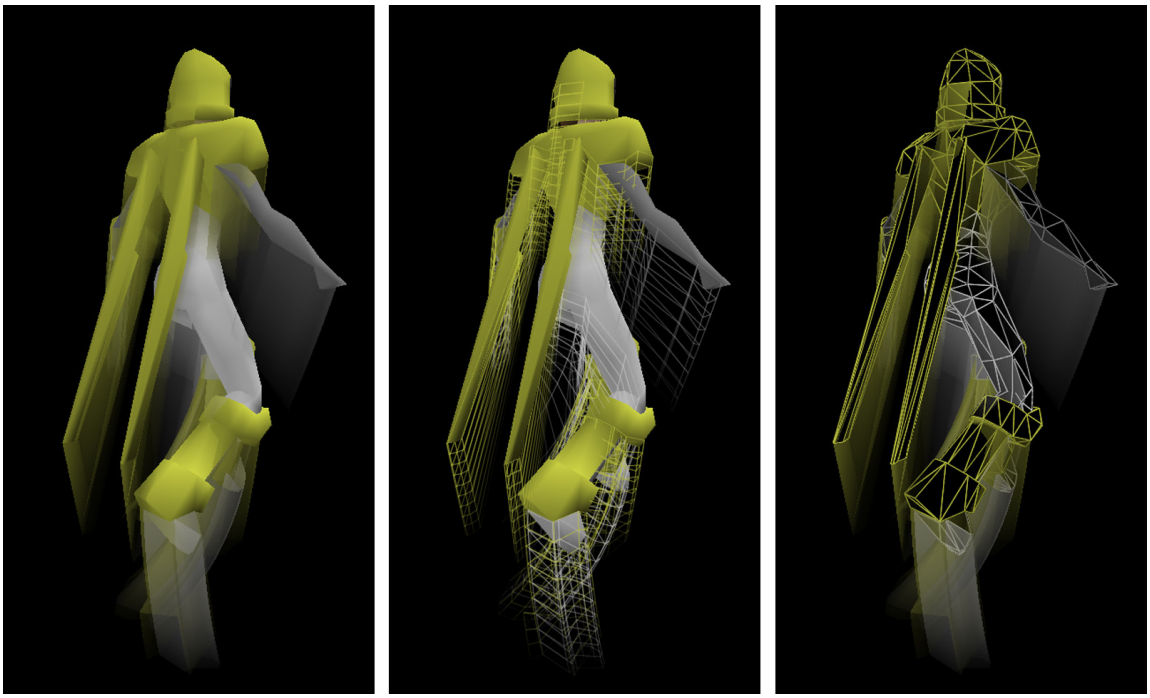

Figure 15. A series of views of the same frame of animation showing the motion blur, the blur's wireframe, and how the blur connects to the model.

The four images in Figure 16 again all show a single frame. However, the bottom right image is of just the blur polygons; the character's mesh has been completely removed. This view is useful as it shows how by just using the motion blur, one can show motion, even without an object to portray it. The upper left hand image is shown larger in Figure 17. In this figure it is possible to see the non-connected blur polygons caused by a model that does not have a continuous mesh. The vertices of the character's feet are not actually attached to its lower legs, causing the silhouette of motion to be discontinuous, as discussed in section 3.3.

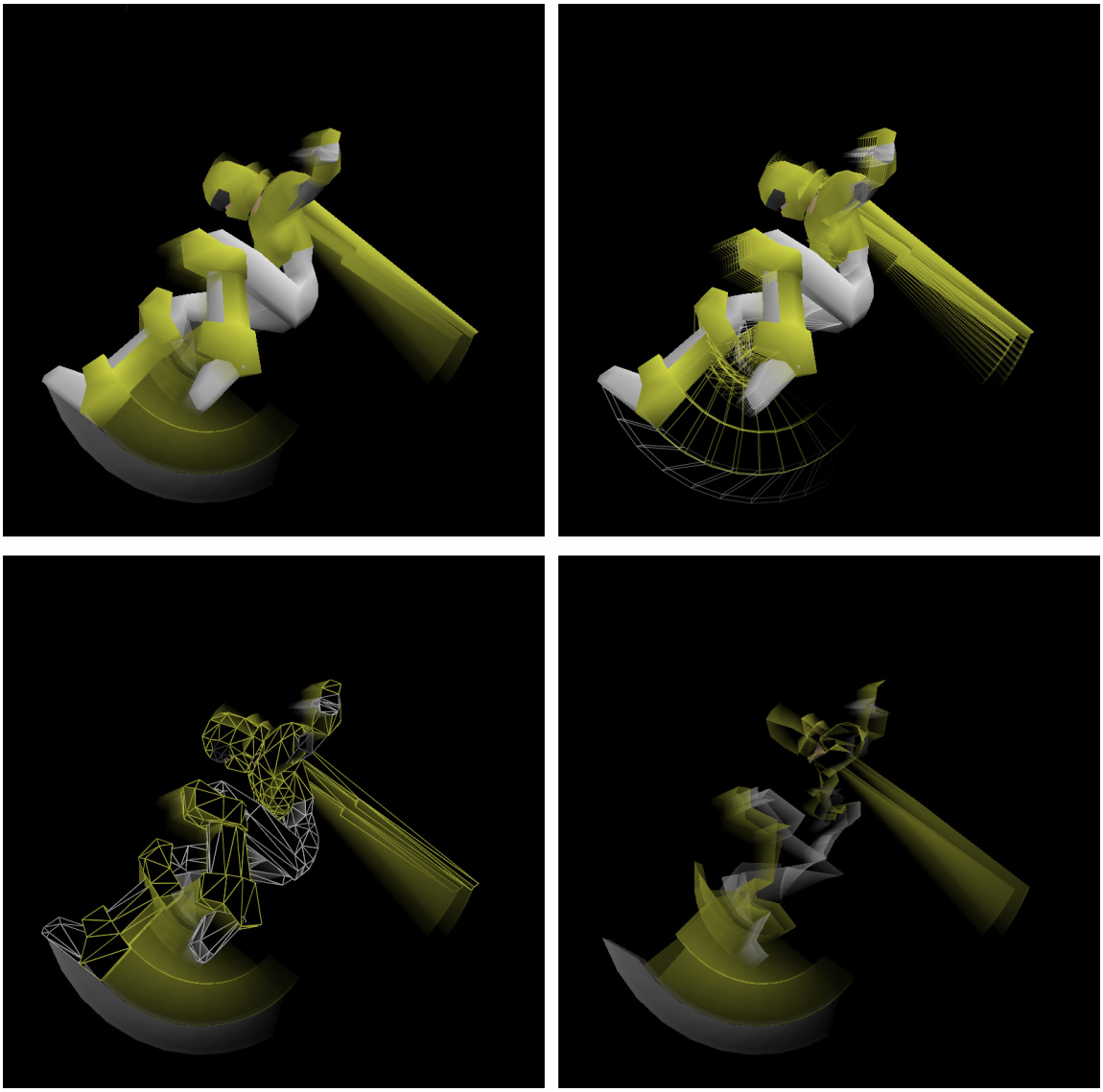

Figure 16. A number of images from the same frame of a jumping animation. The bottom right image just displays the blur polygons.

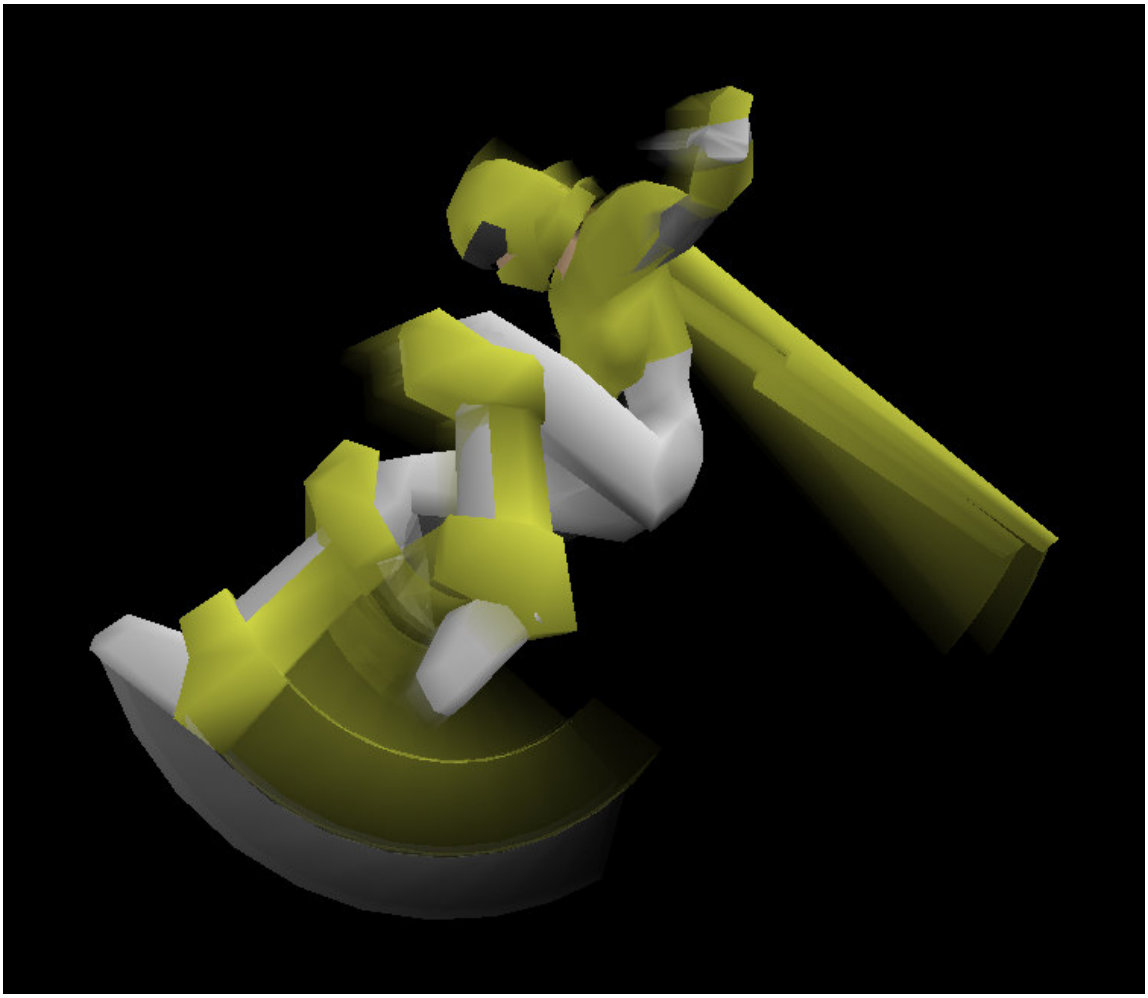

Figure 17. A close up view of the jumping frame showing the disconnected blur produced by the foot

As stated earlier in section 4.3, the number of subdivisions controls how smooth the created motion blur is. Two different blur qualities can be seen in Figure 18. The top two images show a high quality blur, while the bottom shows the wireframe created by a lower quality blur.

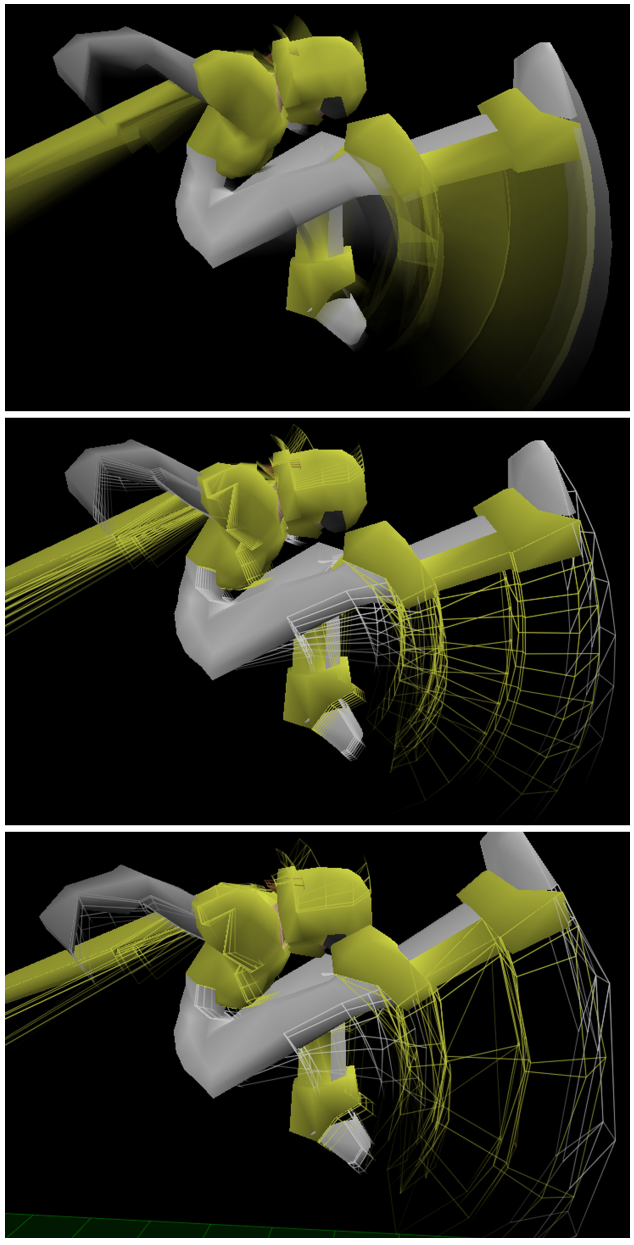

Figure 18. Varying blur qualities

In addition to seeing the blur on an animated character, it is useful to see the effect being used on a moving sphere. Figure 19 shows two views of the same blurring sphere. The top shows a close up of the blur, and how it attaches to the sphere; in this view it can also be seen that the polygons are attached to the silhouette of motion. The bottom figure shows the blur in wireframe. Here the past silhouettes of motion can be seen, they are the rings that are visible in the trailing blur.

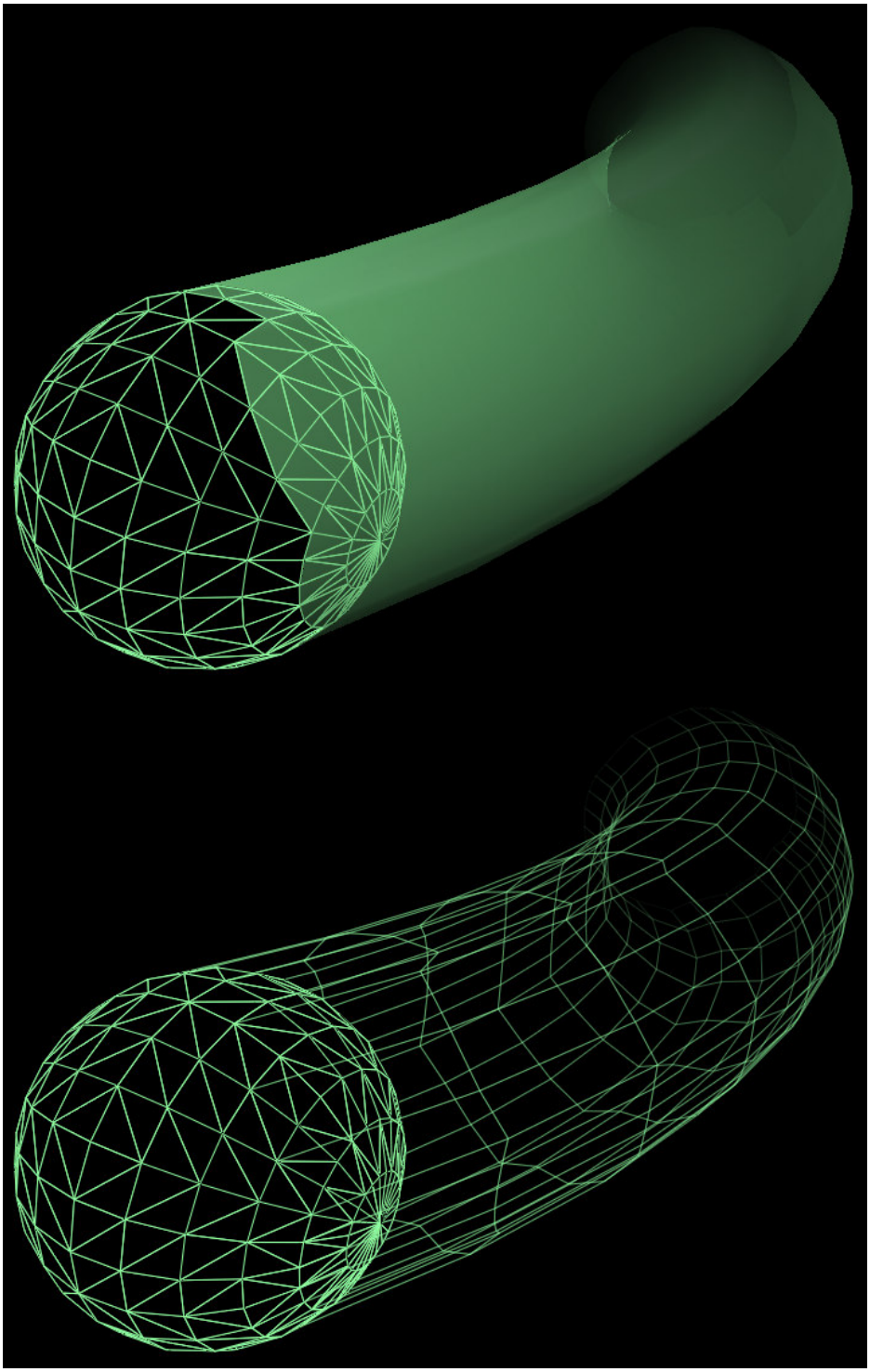

Figure 19. Blur created on a moving sphere, its connection to the model, and the past silhouettes of motion

# 5 FUTURE WORK

There are a number of potential future improvements to the motion blur technique described in this thesis. Most of these involve implementation specific changes.

One such improvement would be to use today's graphics hardware to better manage the blur shell and SoM updates. In addition to the examples discussed in section 4.2, one could use programmable graphics hardware to detect SoM edges. This would allow for much faster SoM detection and rendering, and would require less overhead when communicating with the graphics pipeline.

Another extension to the 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 3.3.

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

# 6 CONCLUSION

Today, computer graphics is an important field within the computer science industry, and it will continue to grow in importance as interactive or entertainment applications become more widely accepted. Industry professionals and researchers have spent years developing the current state of computer graphics. Whether this is physical simulation, high-speed rendering techniques, special effects, exploiting hardware tricks, or just plain using hacks is irrelevant. What matters is that developers are trying to increase the immersion, enjoyment, and utility of real-time graphics applications; and they need every tool at their disposal to do this.

One such tool that is gaining in popularity with real-time application developers, and their users, is motion blur. It is a versatile tool that can used to increase the realism of the rendered scenes, show the path of an object, produce special effects, or simply draw a viewer's attention to something within the scene. Due to the increasing popularity of motion blur, it is a topic currently being researched by a number of groups and companies, including ATI and NVidia; and it will continue to draw research for many years to come.

The thesis above describes a method for producing a motion blur in real-time. It allows for an object with a deforming polygonal mesh to be motion blurred by creating a set of motion blurring polygons through the finding and use of the silhouette of motion over the span of a few updates. It is a method that could be expanded upon and refined, but the scheme provided is generic enough to work with nearly any application that uses polygonal models to display its entities.

Though not being graphically impressive enough for offline graphics – it is extremely hard to compete with the results generated by temporal anti-aliasing – the method provides a visually appealing solution to a current research topic in the area of real-time and interactive graphics. It is quick, easy to understand, and useful for any developers who want to add graphical flair to their application.

For these reasons, the algorithm presented in this thesis is a positive addition to the field of computer graphics, and gives designers one more tool to use when creating motion blurs in their real-time applications.

# REFERENCES

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

Brostow, G. and Essa, I. 2001. Image-Based Motion Blur for Stop Motion Animation. In *Proceedings of Siggraph 2001*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM.

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

Dippe, M., and Wold, E. 1985. Antialiasing Through Stochastic Sampling. In *Proceedings of SIGGRAPH 1985*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM.

Glisse, M. 2003. *On the Theoretical Complexity of the Silhouette of a Polyhedron.* Master's Thesis, INRIA Lorainne, Nancy.

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

Korein, J, and Badler, N. 1983. Temporal Anti-Aliasing in Computer Generated Animation. In *Proceedings of SIGGRAPH 1983*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM.

Max, N. and Lerner, D. 1985. A Two-and-a-half-D Motion Blur Algorithm. In *Proceedings of SIGGRAPH 1985*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM.

NVidia. 2001. Developer Relations Notes. http://developer.nvidia.com/object/motion_blur.html

Potmesil, M., and Chakravarty, I. 1983. Modelling Motion Blur in Computer-Generated Images. In *Proceedings of SIGGRAPH 1983*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM.

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

# VITA

Nathaniel Earl Jones

16310 Heatherdale

Houston, TX 77059

After attending Clear Lake High School in Houston Texas, Nathaniel Jones began his undergraduate degree at Texas A&M University. At A&M he studied computer science, with a focus area in art. After graduating in 2002, he began study for his degree in computer science in A&M's master's program, with a primary focus in computer graphics. He will be graduating from A&M in May 2004.

While attending college, Nathaniel's desire to work with computer graphics and real-time simulation grew to a passion. While researching for a class project, he decided to work in the area of motion blur, an important topic in real-time computer graphics. The prime focus of his coursework and research has been in the use of graphics within computer games. With this focus, he helped to found a new student organization at Texas A&M, the Texas Aggie Game Developers. This club is designed to help students get additional education and experience that pertains to the game industry; something they likely would not get through their standard coursework.

Throughout his college career Nathaniel has worked for a number of companies, but currently is employed by the computer science department as a web application programmer.