Afterimage Toon Blur: Procedural Generation of Cartoon Blur for 3D Models in Real Time

by

Robert L. Smith

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2016

Author.....
Department of Electrical Engineering and Computer Science
January 29, 2016

Certified by.....
Andrew Grant
Lecturer, MIT Game Lab
Thesis Supervisor

Accepted by.....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Afterimage Toon Blur: Procedural Generation of Cartoon Blur for 3D Models in Real Time

by

Robert L. Smith

Submitted to the Department of Electrical Engineering and Computer Science on January 29, 2016, in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

One of the notable distinctions of traditional animation techniques is the emphasis placed on motion. Objects in motion often make use of visual stylistic effects to visually enhance the motion, such as speed lines or afterimages. Unfortunately, at present, 2D animation makes much more use of these techniques than 3D animation, which is especially clear in the stylistic differences between 2D and 3D videogames. For 3D videogame designers fond of the look and feel of traditional animation, it would be beneficial if 3D models could emulate that 2D style. In that regard, I propose two techniques that use the location history of 3D models to, in real time, construct non-photorealistic motion blur effects in the vein of 2D traditional animation. With these procedural techniques, designers can maximize the convenience of 3D models while still retaining an aesthetic normally constrained to 2D animation.

Thesis Supervisor: Andrew Grant
Title: Lecturer, MIT Game Lab

Acknowledgments

I would like to express my gratitude towards my advisor, Professor Andrew Grant for his insight and support during my work on this thesis. Whenever I felt lost or hopeless on how to proceed, which happened often, especially early on, it was he that helped to re-center my perspective and to see a direction for progress. For that, I consider myself lucky to have such a calm and positive advisor.

I would also like to express similar sentiment to fellow researcher Chong-U Lim, for being a constant source of support from which I could share and receive feedback on ideas, results, and concerns throughout the thesis. Not only that, but also, he helped to provide me a workspace that I could use to make effective progress on this thesis throughout the spring semester of 2015. Indeed, that was the time period in which many algorithm details were finally successfully implemented, so I am extremely grateful for his help.

Lastly, I want to thank my friends and family for continuously giving me encouragement to start, continue, and eventually finish this thesis. I surely could not make it here without all of their support.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 State of the Game

It's remarkable how much video game graphics have changed in the past 20 years. When I was young, games were generally in 2D, and they made use of hand-drawn frame-by-frame animations, aka sprites. Nowadays, I've found that an increasing number of video games make use of 3D models and environments, even if the game is played on a 2D plane. It really seems that the evolution of computer graphics, at least for games, has been set on a three-dimensional course.

But personally, I have always been fonder of the hand-drawn, 2D sprite aesthetic than realistic 3D models, particularly the motion effects. Animations, like a punch, a bat swing, or even an explosion, just seem be more impactful when they're hand-drawn. Hand-drawn animations are blurred, or otherwise deformed, which augments their visual impact. And I know I'm not alone; hand-drawn animation still occupies a niche position as a stylistic visual form for video games. The motion effects within games like *Skullgirls* and *Persona 4 Arena* (Figure 1-1) make use of 2D sprites to great aesthetic effect.

Figure 1-1: Sprites from *Persona 4 Arena* [1] Note the stylish curve as Akihiko throws his punch.

Nevertheless, there are times where designers wish to have 2D, sprite-like aesthetic, but dimensional constraints make design difficult. For example, hand-drawn 2D sprites are often inconsistent when the sprite changes horizontal direction. All asymmetric aspects are flipped, such as eyepatches or weapon-handedness. The reason this occurs is generally for efficiency's sake; maintaining consistency often requires extra sprites, which can be difficult due to either constraints on time or limits on space within the game to store these extra assets.

Figure 1-2: Classic examples of limitations with sprite animations. DeeJay's 'MAXIMUM' was purposely chosen because of its vertical symmetry. On the other hand, Sagat's eyepatch and scar both shift to the other side of his body when changes direction. Sprites from *Street Fighter Alpha* [2].

Many of the setbacks of 2D design can be resolved by shifting to 3D models, which also allows for greater flexibility with character design. For example, one of the greatest advantages 3D games still have over 2D variants is the potential of character customizability. Games such as *Street Fighter IV* [3] and *Super Smash Bros.* [4] have 3D character models allow for their character to have multiple, distinguishable outfits. *Tekken* [5] games allow for user-generated outfits by allowing players to select from a variety of tops, bottoms, and accessories. *Soul Calibur* [6] goes one step further and even allows character creation by in essence allowing a player to select a character skeleton in addition to creating their own costumes.

For 2D games to feature character customization of a comparable level, the amount of sprites needed would increase exponentially with the number of potential equipment available to the player. There are ways to mitigate the need for so many sprites, such as drawing sets of clothing and weapons that can be overlaid over a character model like a paper doll. However, the task remains difficult if the character has many distinct motions or if there are multiple characters with differing animations. But perhaps with hybrid models, sprite-like games could support customizability.

1.2 Hybridization

What would be ideal is a hybrid of 2D and 3D design elements. Specifically, a game whose models effectively simulate the aesthetic of hand-drawn sprites, but can also enjoy the functionality of 3D models. *Guilty Gear Xrd*, Arc System Works's newest installment in their *Guilty Gear* fighting game series, exemplifies the potential of this idea. Models are rendered in a way that makes them appear anime-esque, yet game mechanics allow for players to enjoy a distinctly 3D perspective when employing certain attacks.

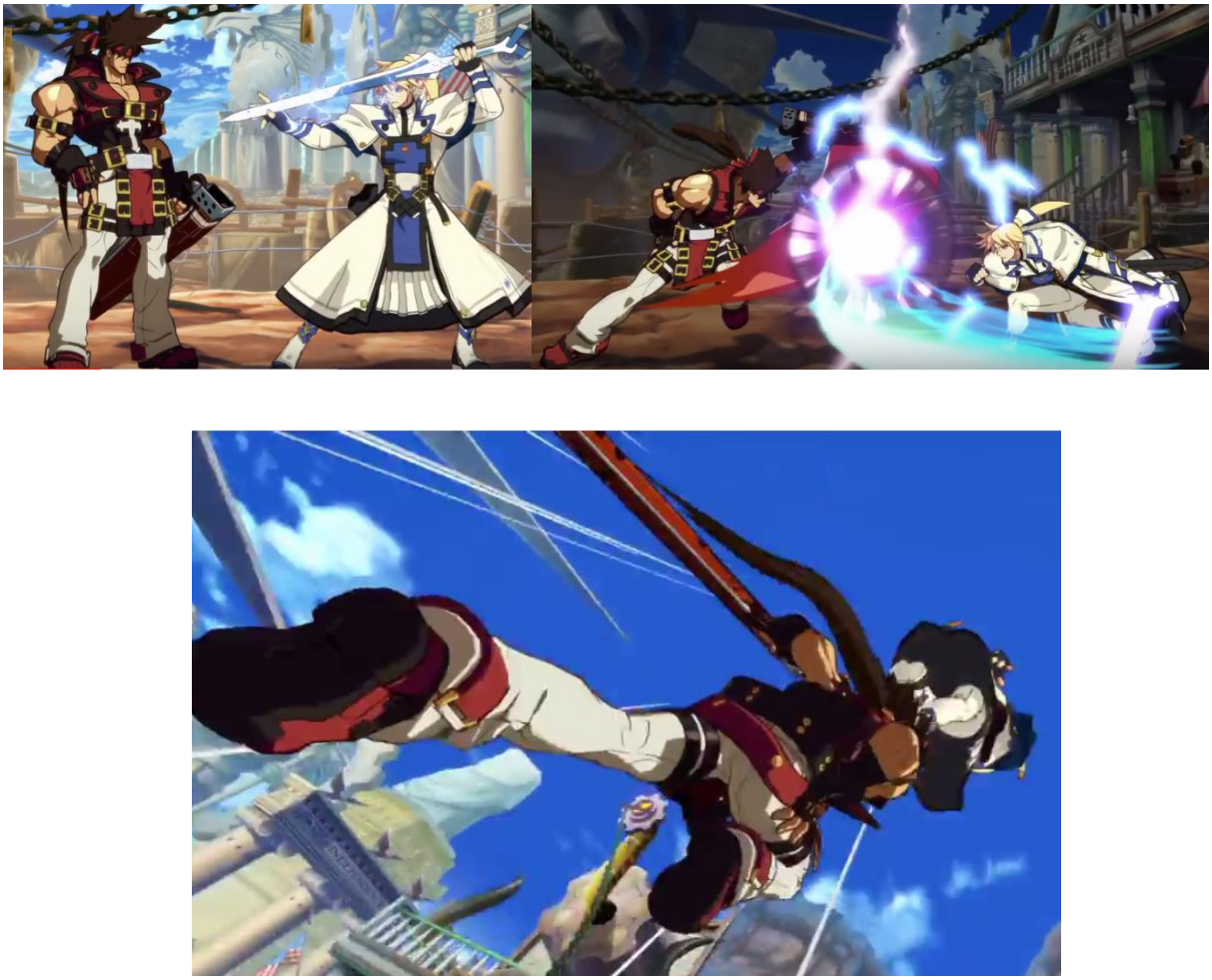

Figure 1-3: Screenshots from *Guilty Gear Xrd* [7]. The top images imply a traditional 2D design, while the bottom helps to establish the 3D nature of the game.

What is perhaps most distinctive about the design of *Guilty Gear Xrd* are its motion effects. Models are often blurred to enhance the feel of high-speed motion. The blurs are created by deforming the actual model. In contrast, other 3D fighters, such as *Super Smash Bros* or *Soul Calibur*, motion is often accentuated with colorful strokes, but models maintain their rigidity.

1.3 Vision

It is clear that the motion effects present in *Guilty Gear Xrd* were created by hand. But what if such motion effects were generated procedurally? If so, then models could be customized in the vein of traditional 3D games, and then have motion effects applied to them, thereby achieving the desired hybrid aesthetic. By decoupling the motion effects from the mesh itself, it would be possible for any kind of mesh to be blurred, deformed, or otherwise modified, without requiring such assets to be constructed beforehand. In addition to customization, procedurally generated motion effects could also be useful when preprocessed assets are difficult to create, whether due to lack of artistic resource or due to the number of assets being too cumbersome to produce.

The high-level goal of this thesis is develop techniques for procedurally generating motion effects. Unfortunately, it is not such a purely technical endeavor, as generated content should be visually convincing enough to be a practical substitute for the real thing. In addition, unlike pre-rendered animation, the nature of video games requires that these effects also be performable in real-time. Even with all of that in mind, it turns out that constructing 2D sprites from 3D models in the vein of traditional animation is a bit of a lofty goal. Thus, the general issue tackled by this thesis has been made much smaller. Instead of attempting to handle all aspects of animation, I decided to mainly focus on high-speed motion; motion that *blurs.* In 2D, characters moving fast tend to have a deformed look: the sprite can stretch and bend, or it appears to be dragged in the direction opposite of motion. I refer to this kind of visual affect as “toon blur,” as it is also commonly found in cartoons.

In this thesis, I will introduce and illustrate techniques for creating different blur effects that can be applied to meshes in real-time. Analysis of these techniques will explore methods to enhance customization, as well as limitations with current algorithms.

Specifically, Chapter 2 discusses relevant research that formed the basis for Afterimage Toon Blur.

Chapter 3 formally introduces the technique of Afterimage Toon Blur, specifically defining what exactly an “Afterimage” is.

Chapters 4 and 5 explore Afterimage Toon Blur in more detail, namely in the form of two techniques.

Chapter 6 discusses engine-specific considerations that were made to produce a working algorithm.

Chapter 2

Background

2.1 Disney Principles of Animation

The task was to emulate a kind of blurring often depicted within traditional animation. Of course, among artists there are different interpretations of toon blur, so I found it necessary to develop a basis for exactly what I wished to emulate. A particularly commonly referenced basis comes from the techniques and thought processes concerning Disney Animation.

Officially there are 12 "principles" [8] all of which center around exaggeration of a body in motion. Arguably, the most well-known principle is known as "Squash & Stretch," and it refers to strong deformations made to an object as it responds to forces upon it. However, another commonly applied principle is called "Slow In Slow Out.'' It refers to how frames representing transitions between keyframes should not be as detailed as frames near the keyframes themselves. It is the combination of these two principles that forms the technique responsible for creating the effect commonly known as cartoon blur.

Another motion effect I was interested in emulating dealt with arcing motions, like sword swings. As a result, I looked into a third principle, appropriately named "Arcs." Arcs deal with the tendency of non-linear motions to follow a curved path. Arcs are therefore important to consider when attempting to blur high-speed, non-linear motions.

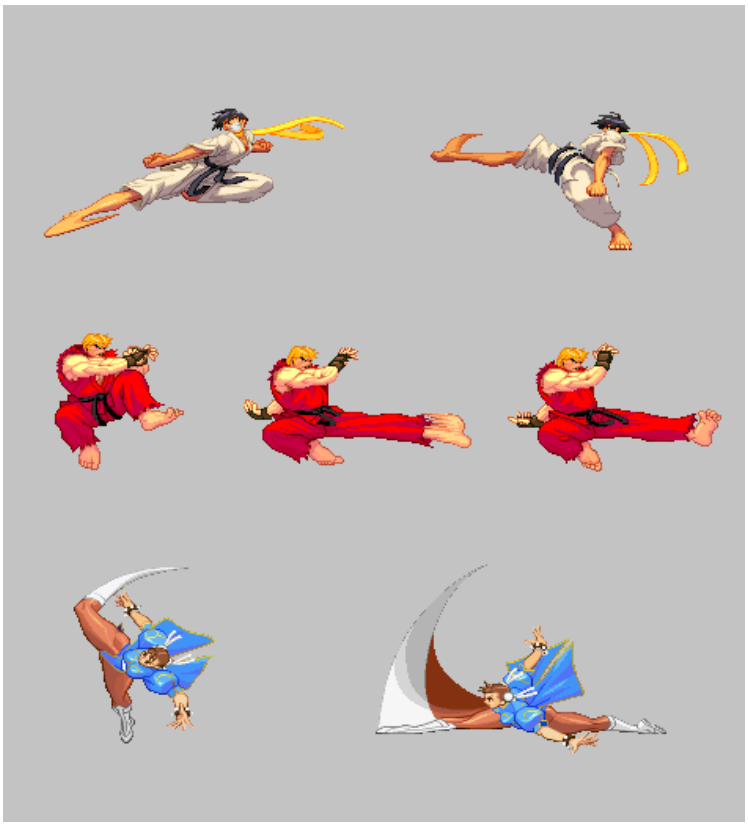

Figure 2-1: Examples of Disney Principles in 2D fighting game sprites. Shown from top to bottom: Stretch and Squash, Slow In Slow Out, Arcs. Sprites from Street Fighter III [9, 10]

2.2 Cartoon Blur

Research done by Kawagishi et.al resulted in a technique for the nonphotorealistic rendering of motion blur [11]. The crux of the technique is an algorithm that determines which parts of an image's outline refers to its "rear" as it relates to its direction of motion. This "rear edge" is detected by finding the dot product between a vertex's direction of motion and a vector perpendicular to an edge of the image. After detecting the rear-edge, various techniques were developed to stylize it, such as drawing speedlines, afterimages, or a deformed blur, as shown in Figure 2-2.

Figure 2-2: Different stylizations of the rear edge. Shown from left to right: Speedlines, Afterimages, Deformation.

While the original paper discussed the techniques in relation to 2D images, it was eventually adapted to work with 3D models [12]. However, the techniques were not applied to the model directly; rather, they were applied to a projection of the model to a 2D plane. From here, I began to devise similar techniques that would construct blurs directly on the models themselves, rather than just their projections.

Chapter 3

Afterimage Toon Blur

The general way that toon blur visually represents high-speed motion is to have some parts of an object lag behind as the model moves through space. This gives the object a stretched look, where the stretch occurs along the motion's path.

In order to procedurally generate this effect, locations of vertices are recorded each frame. These snapshots of recorded vertex locations are called Afterimages. A spline representing the motion path is then created by using these recorded locations as control points. In order to visualize the blur along this path, a mesh's vertices are translated to these recorded locations. In order for this process to work effectively, there must be a 1-to-1 relationship between a vertex and a single recorded position.

The following chapters discuss two distinct techniques: the first restricts the history to a single Afterimage which is used to produce linear blurs, whereas the second uses multiple Afterimages to create a non-linear blur in the form of a trailing arc.

Chapter 4

Single Afterimage Toon Blur

The simplest kind of toon blur is one that follows the contour of the image in a straight line. Single Afterimage Toon Blur, hereafter referred to SAT Blur, is the process for emulating this on a 3D model. As the name implies, a single Afterimage is recorded, which provides location data that is used to construct the blur.

4.1 Algorithm

Formally, the blur is created by translating specific vertices throughout the mesh. A key feature of this algorithm is determining which vertices to translate; not all points can be translated, because enough points must remain untranslated so that the mesh retains its shape. Ideally, only vertices farthest behind along the path of motion should be modified. As such, the SAT Blur algorithm can be summarized in the following steps, which are executed each frame:

1. Find the direction of motion for each vertex.
2. Find the rear-facing vertices of the mesh.
3. Translate these vertices opposite the direction of motion.

The following sections explain these steps in further detail.

4.1.1 Recording Vertex Motion

This step is relatively simple, as it merely involves recording vertex positions at each frame. A vertex's velocity at a time t can be expressed as follows,

$$m_i^t = p_i^t - p_i^{t-1}$$

where p_i^t refers to the position of the i^{th} vertex at time t.

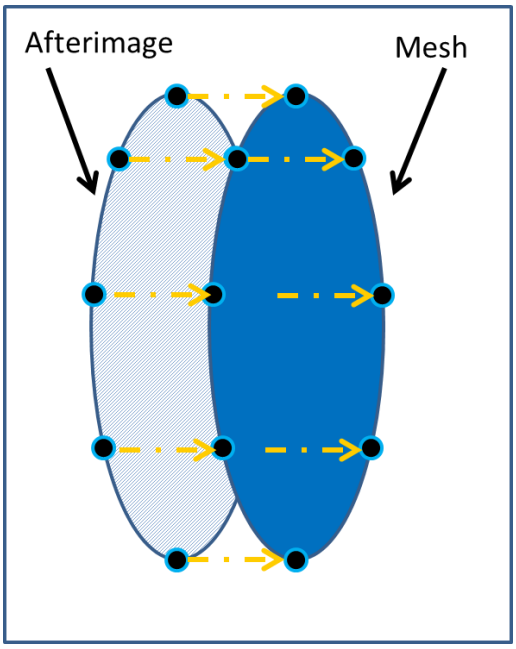

Figure 4-1: Calculation of vertex velocities. Velocities are calculated by taking the difference of vertex positions between two frames (Velocity vectors depicted by yellow arrows.)

4.1.2 Determining the Rear Surface

The notion of "rear-facing" vertices refers to Kawagishi et.al's research in detecting the back edge on an image [11], although the approach has been extrapolated from 2D to 3D. To provide an intuition, the approach for determining the rear-surface vertices is very similar to how a shader determines which sides of a mesh to apply lighting, and which to ignore. Namely, it uses the cosine of the angle between a vertex's normal and a vector that describes a direction of motion. For lighting, this vector is comes from a directional light source. If the cosine is negative, then that vertex is facing away from the light source, and should not be lit. For SAT Blur, the vector is a vertex's direction of motion, which is obtained by calculating the vertex's displacement in a single

frame. If the cosine is negative, then that vertex is facing away from direction it's moving in; in other words, it's part of the rear-facing surface.

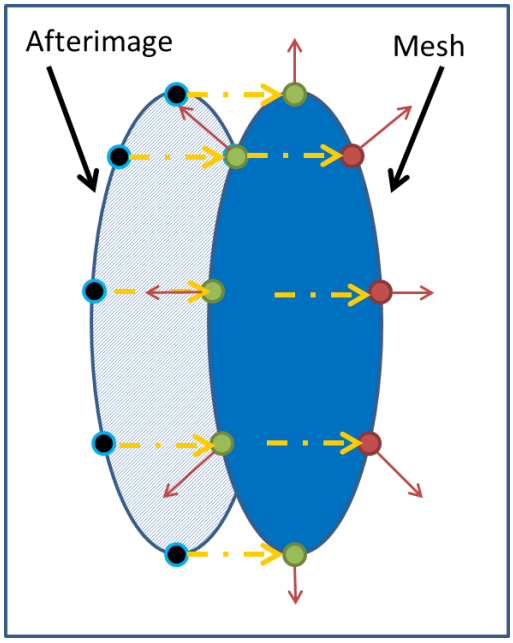

Figure 4-2: Determining the rear-face vertices. These vertices (shown in green) are detected by calculating the cosine of the angle between vertex velocity (shown in yellow) and its normals (shown in red).

4.1.3 Vertex Translation

Once the rear-facing vertices have been found, all that is left is to translate them. In general, translations are performed opposite direction of motion, and a configurable Strength Factor (denoted as f) controls how far the translation should be. Thus, the translation for a vertex i at time t is calculated formally as:

$$T_i^t = R_i^t * f * -m_i^t$$

where R_i^t is a random value between 0 and 1 (more on this can be found in Section 4.3). This value helps to create non-uniform blurs, such as the one shown in Figure 4-3b.

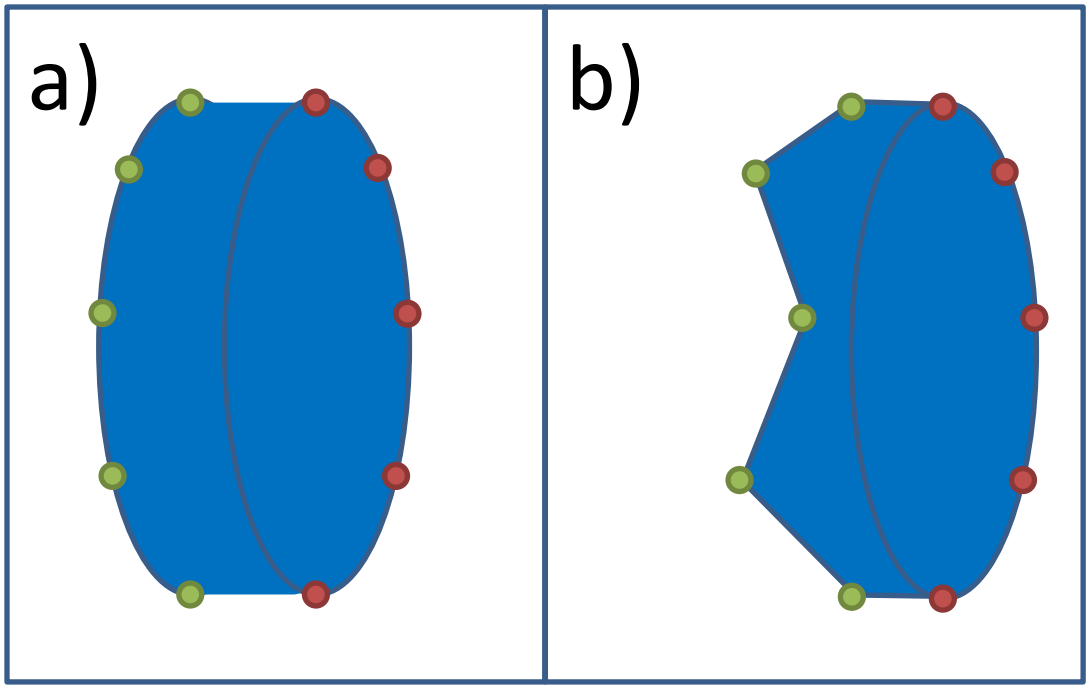

Figure 4-3. Uniform (a) and Random (b) translation of rear-facing vertices.

4.2 Results

In this section, I discuss the results of applying SAT Blur to a typical mesh (Figure 4-4). In addition, I highlight the customizability of SAT Blur and the different effects that can be applied on the same model.

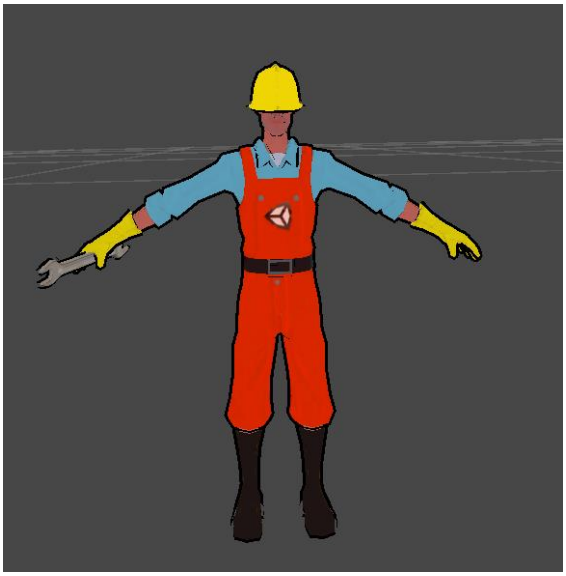

Figure 4-4: Model of a Construction Worker (shown in his bind pose).

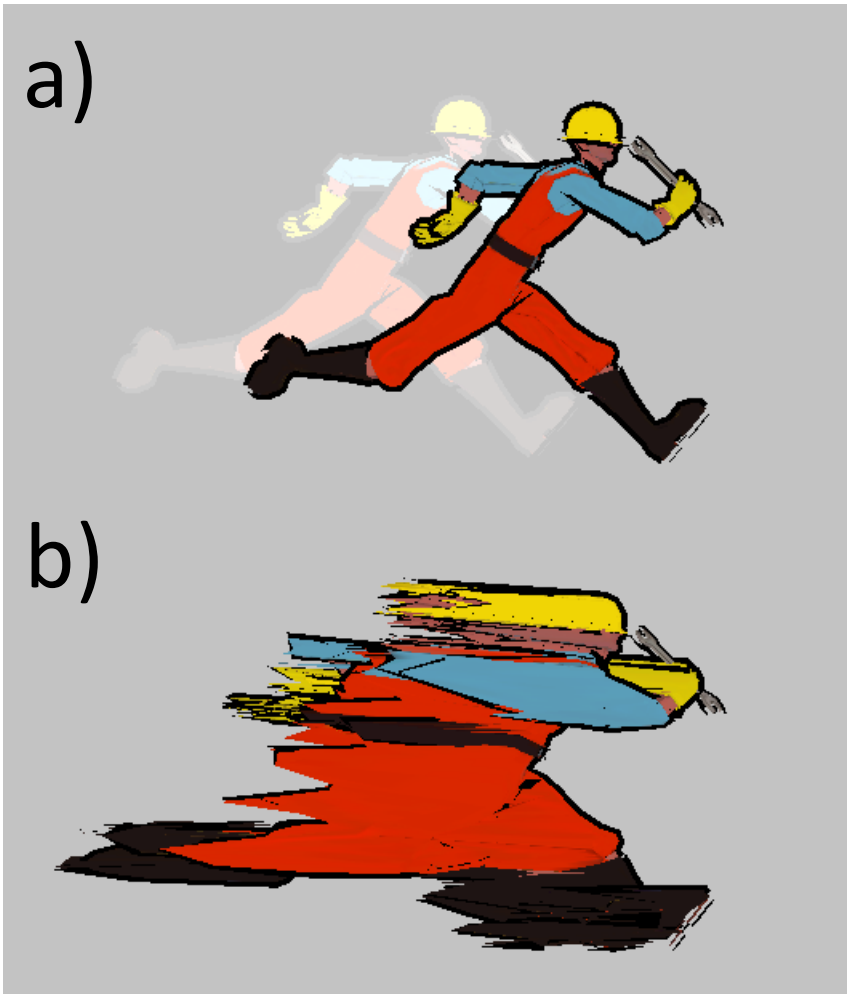

Figure 4-5: Example of Single Afterimage Toon Blur. a) shows an unmodified model and it's recorded Afterimage behind it. b) shows the resultant blur created from the location data provided by the Afterimage.

As shown in Figure 4-6, the blur created by this algorithm is completely 3D:

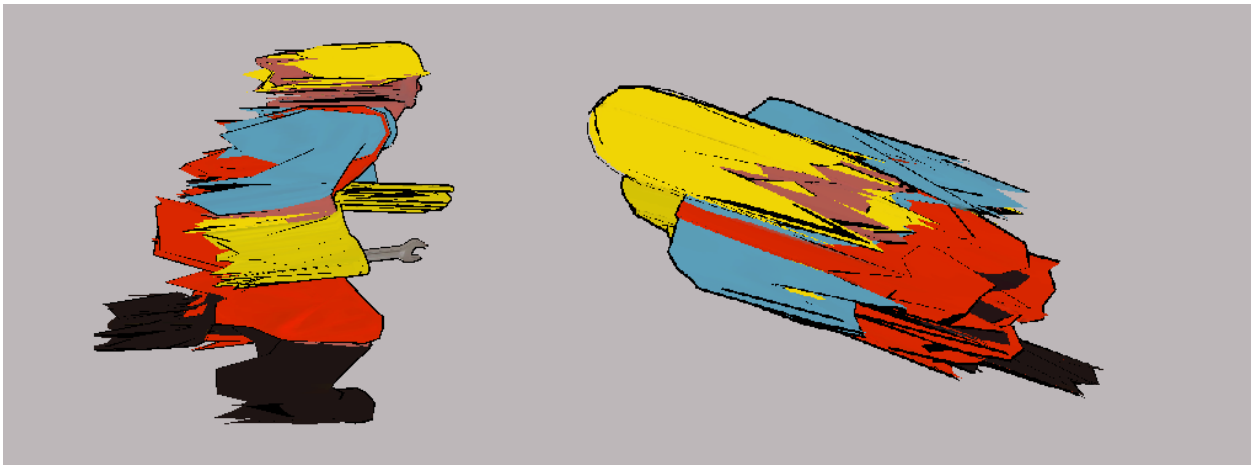

Figure 4-6: SAT Blur shown at varying angles.

4.3 Customizability

One of the main goals of procedural blur is for it for it to have genuine aesthetic appeal, which will most likely differ from designer to designer. In response to this, the algorithm presented was constructed in a way as to make it highly configurable. The main configurable settings produced for this thesis are as follows:

- **Translation Strength**: This value affects how much influence motion has on the resultant blur. Intuitively, larger strength values correspond to longer blurs. There are two separate strength values that control influence due to different kinds of motion
    - **Uniform**: A motion in which the entire mesh is moved in unison. A typical uniform motion is caused by a single force being applied to the model. The uniform motion seen in Figures 4-5 and 4-6 was generated by having the model follow a mouse around the screen.
    - **Non-Uniform**: A motion where different parts of a mesh are moved in different ways. Animations are a typical example of non-uniform motion.

Because it is possible to have both these motions occur simultaneously on a model, a direction for each kind of motion is recorded. By providing 2 separate values, a designer has more control over the extent to which different kinds of motion can influence the produced blur. Specifically, vertex translation is slightly modified to be calculated via the following formula:

$$T_i^t = R_i^t * (f^U * -m^{U}{}_i^t + f^V * -m^{V}{}_i^t)$$

Where the subscript f^U and $m^{U}{}_i^t$ correspond to the contribution to translation from uniform motion, and f^V and $m^{V}{}_i^t$ correspond likewise to non-uniform motion. The calculation of the motions is functionally the same, except that non-uniform motion is calculated via a mesh's position in local space, while uniform motion is calculated via position in world space, albeit with motion due to internal factors subtracted away.

- **Randomization**: It is possible to modify the style of the blur by tweaking the way the algorithm uses it's random influence factor (r):
    - **R-Value Bounding**: By default, the translation distance is multiplied a random value $r \in [0,1]$. By changing the bounds, different blur styles can be created. In general, standard blur effects are generated when there is a reasonable difference between the lower and upper bounds. However, when they are close to the same, the blur follows the contour of the object.
    - **R-Value Thresholding**: Another way to modify r is to threshold it. So, given a pseudorandom function $R(x)$ and a threshold $r_t \in [a, b]$:

$$R(x) = \begin{cases} 0, & x < r_t \\ x, & x \geq r_t \end{cases}$$

      The effect of this is similar to bounding in that blurred vertices correspond to values in the interval $[r_t, b]$. The difference is that now not every vertex is blurred, and

that as r_t increases, the resulting blur "sharpens," because for any blurred vertex, it becomes less likely that its immediate neighbors are blurred as well. Thresholding in this way has the side effect of increasing the length of the blur as the threshold increases, and it may be appropriate for thresholding to occur independently of x instead.

Figure 4-7: Default blur vs Threshold blur. Note how the blur on the right is more jagged and less of the model is deformed than the one on the left is.

- **Tapering**: By default, with the exception of randomization, all vertices are translated proportional to values independent of their position in the mesh. As shown in Figure 4-8, the resulting blur has a distinctly rectangular shape. One desirable modification is for the blur to better follow contours of the mesh. One way to do this would be to make the translation to be proportional to the direction of the vertex normal. Essentially, the larger the angle between the direction of motion and the vertex's normal, the less that vertex should be translated.

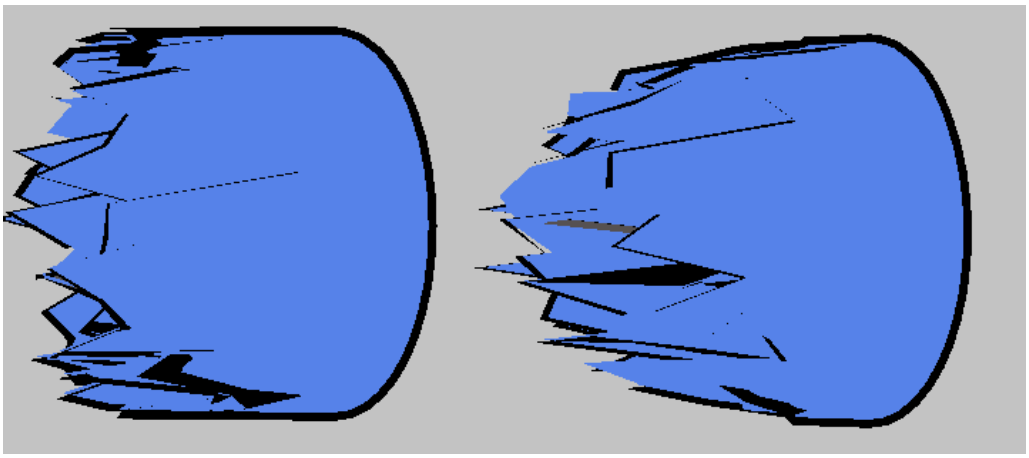

Figure 4-8: Default blur vs Tapered blur on an ellipsoid. Note how the blur on the left is relatively parallel to direction of motion, while the on the right seems to come to a point.

- **Speedlines**: Normally, this algorithm modifies the mesh directly by translation specific vertices. However, speedlines can be created by instead drawing edges from the rear-surface vertices to their translated locations.

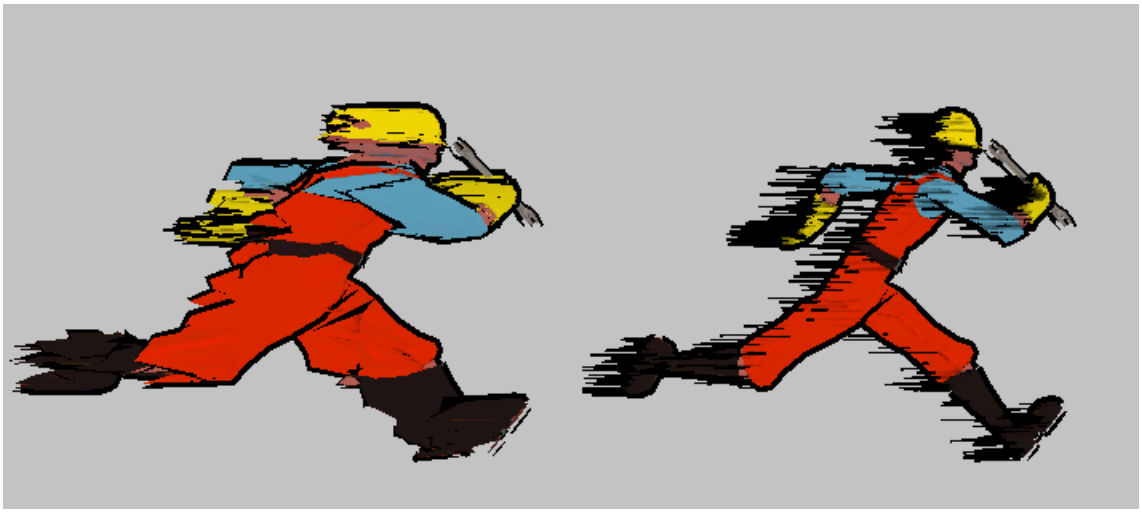

Figure 4-9: Normal SAT Blur vs speedlines generated by the same algorithm.

4. 4 Limitations

Rear-face vertex detection works best on smooth surfaces, because adjacent vertices tend to have their normals oriented in a similar direction. As a result, meshes with sharp corners tend to have

those features blurred relatively poorly. This is especially apparent for meshes of few vertices, or where the vertices are generally located along these sharp boundaries. Figure 4-9 shows the effect on a simple cube:

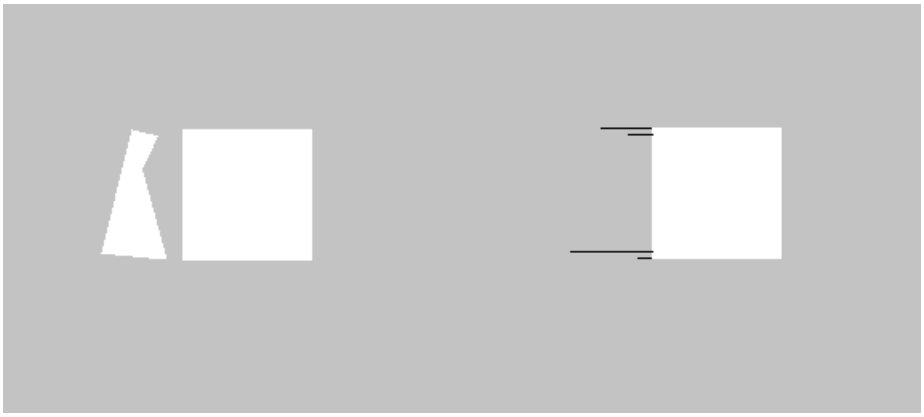

Figure 4-10: SAT Blur performed on a 24-vertex cube (3 vertices at each corner). Speedlines shown for comparison.

In these cases, speedlines are an effective alternative, as there is no risk of terribly deforming the mesh in question.

4.5 Future Work

SAT Blur is designed as an algorithm that runs in real-time, and as such, performance is very important to keep in mind. Currently, the process runs suitably well, although slowdown is apparent with multiple meshes, or even a single mesh with a large number of vertices. However the reason for this slowdown may have less to do with the algorithm and more to do with the engine it was programmed in (see Chapter 6 for more detail). Nevertheless, applying the algorithm via a shader seems to be a viable alternative; the simplicity of SAT Blur lends itself well to parallelization on the GPU, as many elements to its algorithm are per-vertex, such as the determining the rear-face and performing the translation. The only tricky bit seems to be calculating vertex velocities, as it

requires storing past vertex locations. While I was unable to explore this alternative in much detail, some progress was achieved by applying a single velocity to all vertices, which simulates blur applied via a strictly external motion.

Chapter 5

Multi-Afterimage Toon Blur

While SAT Blur is able to produce decent blurs as long as they are linear, the main goal of this thesis is to represent non-linear deformations as well, particularly those due to arcing and spinning motions. In this chapter, I propose an extension to SAT Blur that makes use of data recorded by multiple Afterimages to create curved blurs.

5.1 Algorithm

In order to support multiple Afterimages, it must be possible to both record and translate vertex positions along a non-linear path. Recording is simple; simply increase the size of the history buffer. Translation, however, is not so trivial. By default, a mesh's vertex can only correspond to one recorded position at a time. Supporting extra vertices typically requires some augmentation to the mesh. The current approach preprocesses the mesh to support more Afterimages, although a real-time augmentation may also be feasible. Below, I enumerate the steps for Multi-Afterimage Toon Blur, hereafter referred to as MAT Blur:

0. Preprocess Mesh to support multiple Afterimages.
1. Record vertex positions over a number of frames.
2. Determine back-facing vertices.
3. Translate these vertices.

Notably, the last 3 steps are almost the same as they are for SAT Blur. Indeed, much of the complexity of MAT Blur actually comes in the form of augmenting the mesh. The following sections describe this first step in more detail.

5.1.1 Vertex Duplication

In order to support m multiple Afterimages for a mesh with n vertices, $n * (m - 1)$ extra vertices are added to a mesh. These extra vertices have the same default position and normal as the original vertex.

5.1.2 Triangulation and UVs

In order for this augmented mesh to render properly, these new vertices have to be connected into triangles. Triangulation has to be don=e in a way such that the translation of multiple vertices gives the shape of a blur. The triangulation of these extra vertices is as follows:

1. Construct an adjacency graph that maps each vertex to all other vertices it shares a triangle with.
2. Given m Afterimages, for each vertex, there are $m + 1$ points (hereafter referred to as "frame vertices") that initially have the same location. The vertex i_{j+1} will be directly connected to vertex i_j .
3. For each vertex i_j, there are k adjacent vertices. For each of these adjacent points ai_j, get its succeeding frame vertex ai_{j+1}.
4. These 4 points $\{i_j, i_{j+1}, ai_j, ai_{j+1}\}$ correspond to a rectangle (Figure 5-1), from which two triangles can be constructed.

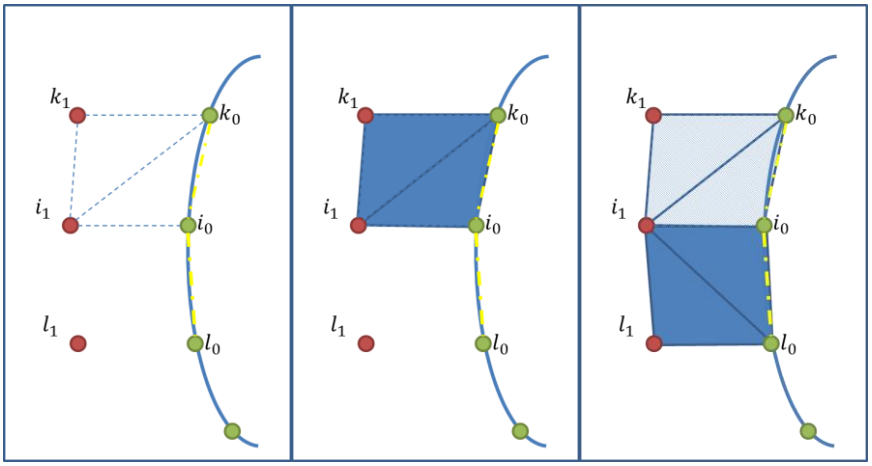

Figure 5-1: Visual depiction of triangulation process (namely steps 2-4). For reference, green dots represent original mesh vertices, red dots represent duplicated vertices, yellow lines adjacency of original vertices, and the blue triangles represent the newly constructed triangles.

Alongside determining new triangles, these added vertices have to be given uv-coordinates in the case they are meant to be used with a texture. By default, vertex $i_{j>0}$ shares the same coordinates as vertex i_0. The effectiveness of these chosen uv-coordinates depends heavily on the type of texture used. For textures with large continuous details, the blur is passable, but when there are many fine details in the texture, the effect can be very poor, as shown in Figure 5-2.

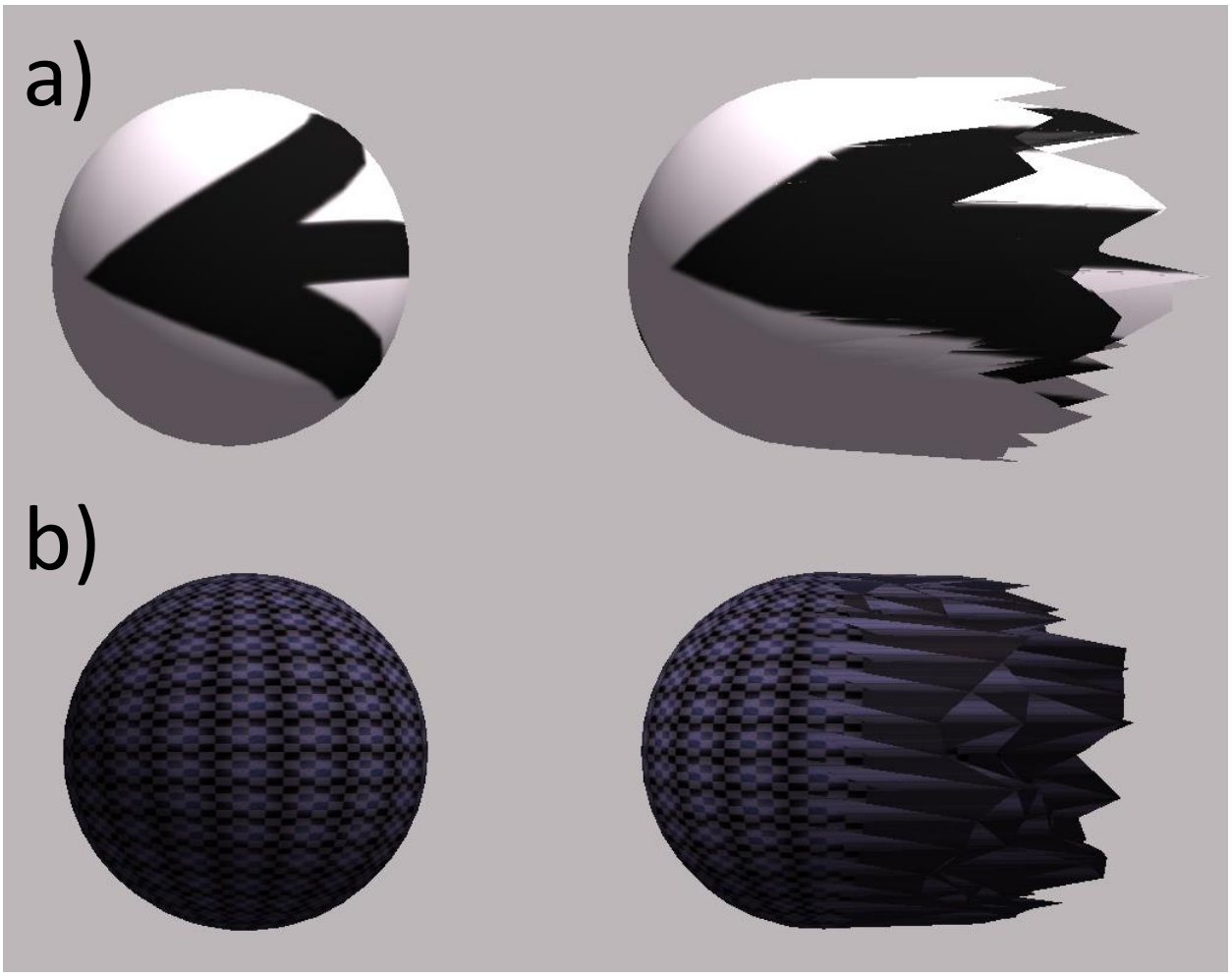

Figure 5-2: Blurring of textured spheres (originals on left, blurred versions on right). For a) the chosen uv-coordinates allow for a decent blurring effect. Unfortunately, as seen in b), those uv-coordinates are not as effective for small details.

5.2 Results

As MAT Blur multiplies the number of vertices in a mesh by a significant amount, it is more suitable for smaller meshes, like weapons or other hand held objects. The main mesh used to illustrate the effect was a samurai sword (Figure 5-3).

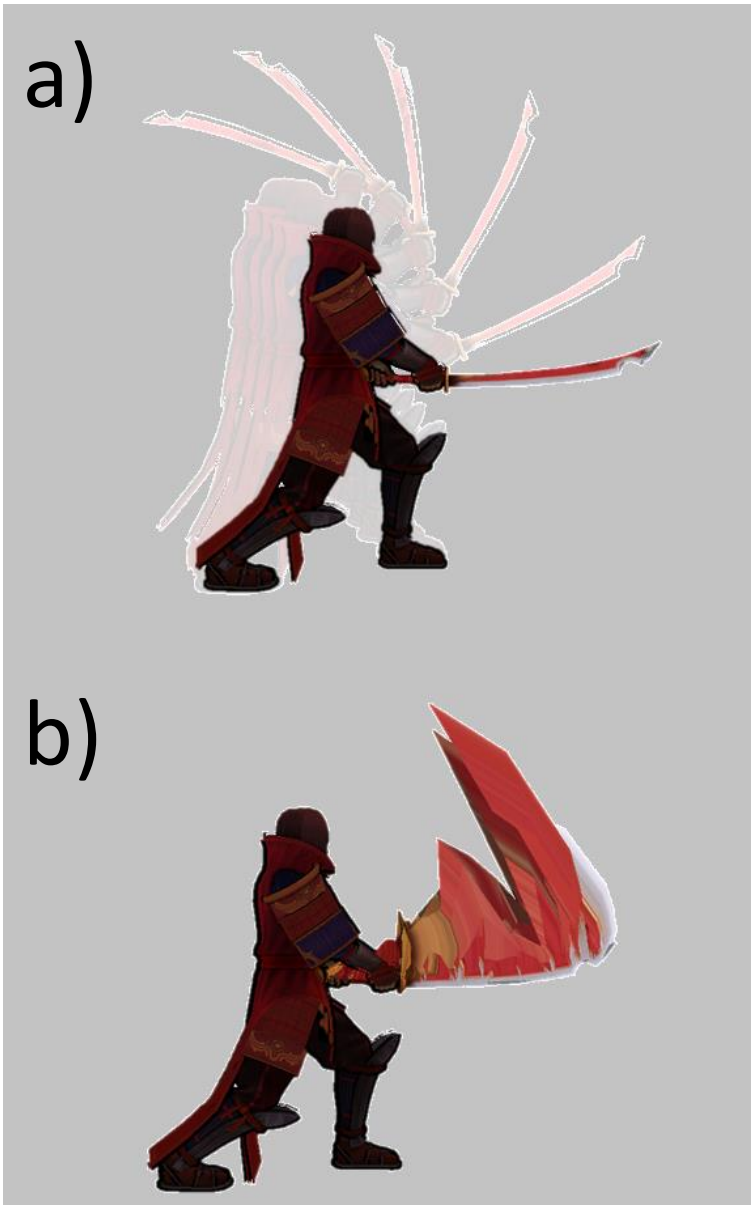

Figure 5-3: Example of Multi-Afterimage Toon Blur. a) shows 5 recorded Afterimages (note that the blur is performed on just the sword). b) shows the resultant blur from the location data of the recorded Afterimages.

5.3 Customizability

One of the main goals of procedural blur is for it for it to have genuine aesthetic appeal. However, the desired visual will most likely differ from designer to designer. In response to this, the

algorithm presented was constructed in a way as to make it highly configurable. The main configurable settings developed in this this thesis are as follows:

- **Influence**: By default, MAT Blur will create arcs that visualize the entire mesh sweeping through space. However, as shown in Figure 5-4a, there are times where this alone is not really a convincing blur; a sharper, steeper-looking arc would be more desirable. While normally each vertex in the original mesh is treated the same, an alternative would be for vertices that travel shorter distances (i.e. at closer to the center of rotation) to be less pronounced than vertices that move farther away. This introduces the idea making displacement of vertices proportional to some "influence value. " Formally, influence can be represented as any mathematical function $I(X)$, where X is a series of inputs, and $I(X)$ has an output range of [0, 1]. Below, I give a brief analysis of two example influence functions.
    a) A simple influence function used in this thesis was

$$I(D_i, D_{max}, k) = \left(\frac{D_i}{D_{max}}\right)^k$$

where D_i is the total distance traveled by a vertex with respect to its history locations, D_{max} is the maximum distance traveled by any vertex with respect to the entire history buffer, and k is a nonnegative modifier that enhances the intensity of the influence. The raising to a power was used because it increases the variety of the output arc while still keeping the influence values bounded between 0 and 1. Unfortunately, for high values of k, all vertices with a $D_i < D_{max}$ quickly degenerates to 0 influence, leading to arcs like the one shown in Figure 5-5a.

    b) A slightly more complex influence function that I used was

$$I(D_i, D_{max}, n, j, k) = \left(1 - \left(1 - \frac{D_i}{D_{max}}\right) * \frac{j}{n}\right)^k$$

where j refers to the j[th] afterimage that is being translated, and n is the number of

Afterimages per vertex. The result of this function is that earlier Afterimages are not as affected by relative distance traveled (influence remains close to 1) as later Afterimages (influence converges to $\left(\frac{D_i}{D_{max}}\right)^k$).

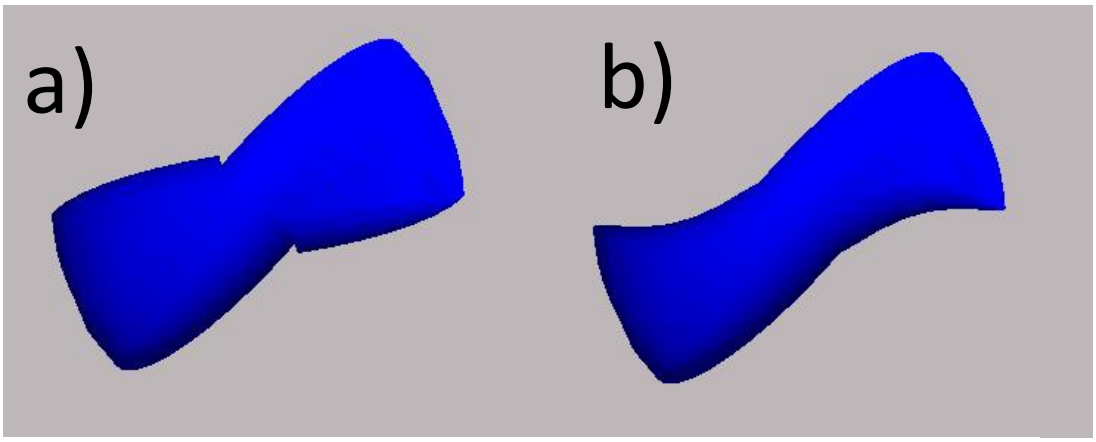

Figure 5-4: Ellipsoids blurred as they spin around their center. a) shows a default blur, while b) shows a blur with a modified influence function.

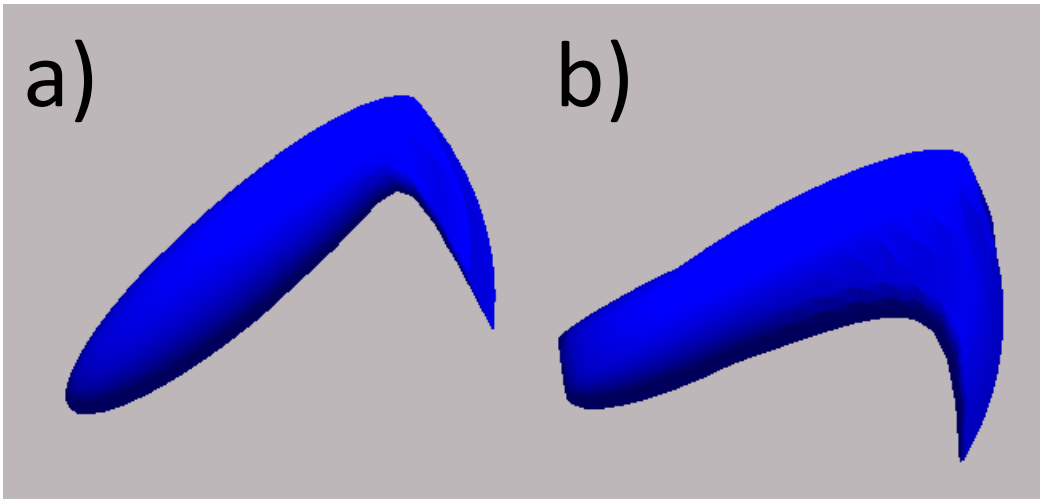

Figure 5-5: Spinning ellipsoids blurred with their respective influence functions (a) and (b). In both cases k-factor = 10.

- **Randomization**: Like with SAT Blur, MAT Blur makes use of a randomization factor to enhance the effect of the blur. Like influence, any pseudorandom generator can potentially be used; the main constraint on the function is consistency. To be specific, a randomization

factor R_i for a vertex i should be time and spatially invariant. Time-invariance is important mainly for meshes blurred at stand-still, so that the blur does not rapidly flicker while the game is paused, for example. Spatial-invariance is important for blurs to remain consistent not only for linear motions with a constant direction, but also as rotations.

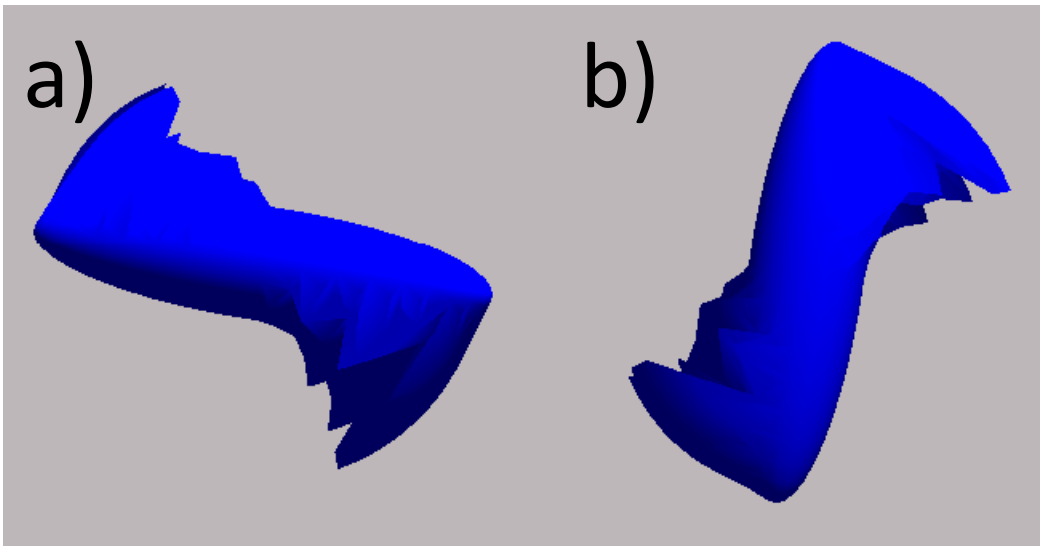

Figure 5-6: Spinning ellipsoid blurred with randomization. Note the consistency of the blur's translated vertices as the object rotates.

- **Thresholding**: Sometimes, it is not desirable to blur every vertex possible. In order to contrast the high-speed parts of a mesh to its parts that are moving at a reasonable speed, it would be reasonable to set a threshold so that only the fastest vertices to be blurred.
    - *Relative*: If a mesh is designed to blur every frame, it may be the case that only a portion of that mesh gets blurred at any one time. For example, in previous images that depict a samurai swinging a sword, the pommel and hilt are usually blurred along with the blade. However, due to the arcing motion of the swing, the vertices corresponding to the blade typically move faster than these parts. Figure 5-7 shows a relative threshold that permits only the blade to be blurred.

- *Absolute*: Another possibility is that parts of a mesh only blur if they move fast enough. In this case, it might be better to have a threshold that is independent of vertex velocities. This is useful for highlighting a specific motion, such as the firing of a gun or even throwing a punch.

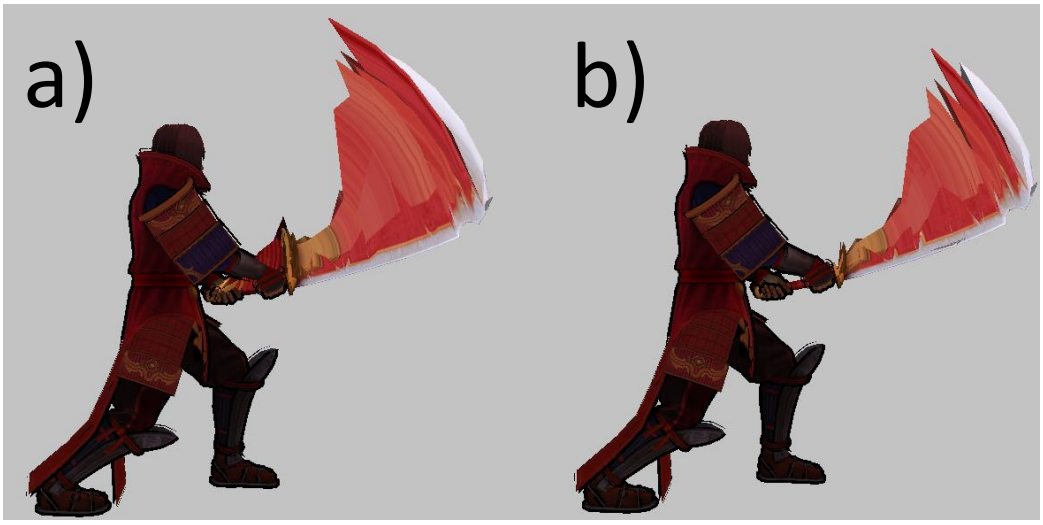

Figure 5-7: Side-by-side depiction of normal MAT Blur (a) and MAT Blur with a relative threshold of .65 (b). Note how the sword on the right only has its blade blurred, whereas the both the blade and hilt are blurred for the left sword.

5.4 Limitations

One of the drawbacks to this algorithm is that, while every vertex has triangles between frames, not all vertices are translated at once, which can lead to some bizarre artifacts, such as overlapping triangles. One possible solution is to determine the list of triangles to draw at each update step, which is currently not being done for performance reasons.

Another consideration is that this algorithm assumes that all vertices that share the same location share adjacent vertices as well. While is it not a bad assumption to make, it is not always true, and same-location vertices do not always share triangles between them. The issue is that the duplicate

vertices are often used so that a single point has multiple normals, and therefore can be lighted on more than one side. But because this algorithm depends on vertex normals, it is possible for one vertex to be translated and other to not. In this case, if they do not share triangles, then the resulting blur can become disconnected from the original mesh. As shown in Figure 5-8, the samurai sword used in this thesis suffered from this issue. To work around it, if a vertex was determined to translate, all other vertices that share that location were translated as well. However, in general, this is an issue with the mesh's design rather than the algorithm.

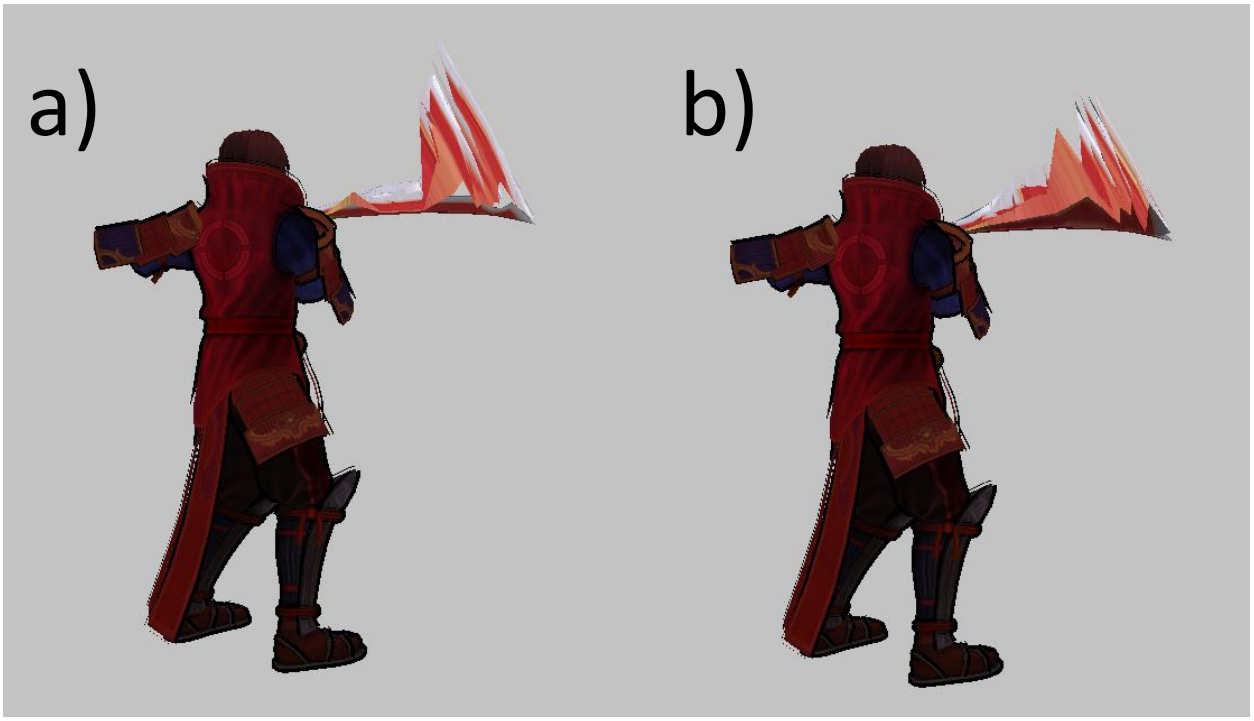

Figure 5-8: Side-by-side view of MAT Blur on a samurai sword. a) is the original blur, which has the blur disconnected from part of blur, whereas b) shows a blur created by forcing same-location vertices to be blurred together.

On a related note, as MAT Blur used the same principle as SAT Blur to translate vertices, it too suffers from the same normal-based limitations.

5.5 Future Work

As with SAT Blur, performance of the MAT Blur algorithm is very important. However MAT Blur suffers from slowdown to a far greater extent than SAT Blur does, as the number of vertices increases linearly with the number of desired Afterimages. Like SAT Blur, MAT Blur is currently calculated on the CPU, although due to the similarities between the two algorithms, I would argue that a GPU shader has great potential for providing a speedier alternative here as well.

However, another possible way to improve performance while still running the algorithm on the CPU is to reduce the number of original vertices that are used to calculate the blur each frame via vertex simplification. Blurs inherently lack detail, so vertex simplification techniques seem like a reasonable approach for reducing calculation time, as long as the produced blur has enough vertices to suitably follow the contour of the object.

Chapter 6

Programming Considerations with Unity3D

All techniques concerning Afterimage Toon Blur were implemented in Unity3D, as it provided a nice graphical interface for visually verifying the effectiveness of toon blur techniques on 3D models. While the algorithm for AIM Blur is simple conceptually, implementing it in Unity turned out to be less intuitive than expected, mainly due to way that Unity stores and renders model data. This chapter details the Unity-specific considerations that were necessary to produce a working algorithm.

6.1 Calculating Vertex Transforms

In Unity, as with most graphic engines, while the direction a vertex moves is calculated in the mesh's world space, the modifications to that mesh have to be done within its bind pose. This is because a model's position is only stored in bind-pose space; it is dynamically transformed each frame to get the world-space position. Ultimately, for the calculations to work correctly, a transform between bind pose space and world space is necessary.

Unfortunately, Unity does not provide these transforms immediately to the user. Instead, they have to be calculated each frame from bone weight data, in a manner equivalent to performing one's own Skeletal Subspace Deformation (SSD). The issue with this is that these transforms for SSD are normally calculated on the GPU, but in order to use the calculated transforms with the algorithms, they had to be computed on the CPU instead. Needless to say, per-vertex transforms can be expensive to compute, which caused significant drag on the framerate.

6.2 Attempts at Optimization

In truth, as long as SSD transforms are computed on the CPU, the practical utility of Afterimage Toon Blur is questionable. However, it was important to ascertain the just the effect of producing the blur on runtime. As a result, a significant effort was made to optimize the SSD transform calculations as much as possible.

6.2.1 Reducing Number of Calculated Transforms

Originally, the algorithm performed the motion direction calculation in world space. This required transforming both the vertices and their respective normals from bind pose space to world space. Then the translated vertices had to be transformed back from world space to bind pose space for Unity to properly render the blur. The resulting process was very slow, averaging an FPS of 15-20 for just the SAT Blur of single mesh of ~2000 vertices. For perspective, 60 FPS is the typical framerate on most video games, and drop to even 50 FPS can be noticeable and jarring. Fortunately, it was discovered that much time could be saved by transforming the direction of motion, which was normally calculated in world space, to a bind pose direction for each vertex, and then doing the translation within that space. The significant difference was that normals did not have to be transformed at all, resulting in essentially a 2x speedup.

6.2.2 Caching

While significant time was spent on optimization of the SSD, there is a limit to what can be done on the CPU. However, one observation made during this time was that vertex locations repeated themselves as their animations loop. As such, their respective transforms must repeat as well. Caching these transforms were therefore a potential alternative. Without caching, SAT Blur

on a single 2000 vertex mesh topped out at about 30 FPS. However, with caching, the FPS eventually reached 80 FPS after about 5 seconds of runtime.

Unfortunately, caching too has its limitations. For one, 3D animations are not discrete like 2D sprite-based ones are. As such, there is a potential for even looping animations to have slightly different transforms as they repeat, defeating the effectiveness of the cache. Animations, therefore, have to be discretized, as to limit the number of stored transforms. I settled on 100 transforms per second per animation per vertex, as I felt that number to be high enough to capture enough different local-space orientations for a single animation, but low enough for the cache to find hits more quickly. Another drawback to caching that it is based on the current animation. By being animation specific, it is possible to have a decent performance for one animation after it has run for a bit, but then for the framerate to slow down significantly after switching to a new animation for the first time.

Chapter 7

Conclusion

In this thesis, I have introduced a technique for procedurally creating cartoon-style blur on 3D models in real-time. In addition, I have outlined ways in which the algorithm can be modified to suit personal preferences. While at present my work is merely proof-of-concept, the long-term goal is to create a system that game designers can use to generate toon blur on their own meshes. As these blur algorithms are written with the Unity3D engine, the most practical next step would be to consolidate these blur techniques into a Unity Asset that can be injected into projects and used by other developers.

References

[1] *Persona 4 Arena*. Arc System Works. August 7 2012. Video game.

[2] *Street Fighter Alpha.* Capcom. February 7 2006. Video game.

[3] *Street Fighter IV.* Capcom. July 7 2009. Video game.

[4] *Super Smash Bros. Brawl.* Nintendo. March 9 2008. Video game.

[5] *Tekken 6.* Namco. October 27 2009. Video game.

[6] *Soul Calibur IV.* Namco. July 29 2008. Video game.

[7] *Guilty Gear Xrd.* Arc System Works. December 16 2014. Video game.

[8] Thomas, F. & Johnston, O. Disney animation: the illusion of life. (Abbeville Press,1981).

[9] *Street Fighter III.* Capcom. February 1997. Video game.

[10] Style Study: Motion Blur in Street Fighter 3 « Final Form Games. At <http://www. finalformgames. com/uncategorized/style-study-motion-blur-in-street-fighter-3/>

[11] Kawagishi, Y. , Hatsuyama, K. & Kondo, K. Cartoon blur: nonphotorealistic motion blur. in Comput. Graph. Int. 2003 Proc. 276–281 (2003). doi:10. 1109/CGI. 2003. 1214482

[12] Syoichi Obayashi, Kunio Kondo, Toshihiro Konma, and Ken-ichi Iwamoto. 2005. Non-photorealistic motion blur for 3D animation. In *ACM SIGGRAPH 2005 Sketches* (SIGGRAPH '05), Juan Buhler (Ed.). ACM, New York, NY, USA, , Article 93 . DOI=http://dx. doi. org/10. 1145/1187112. 1187224