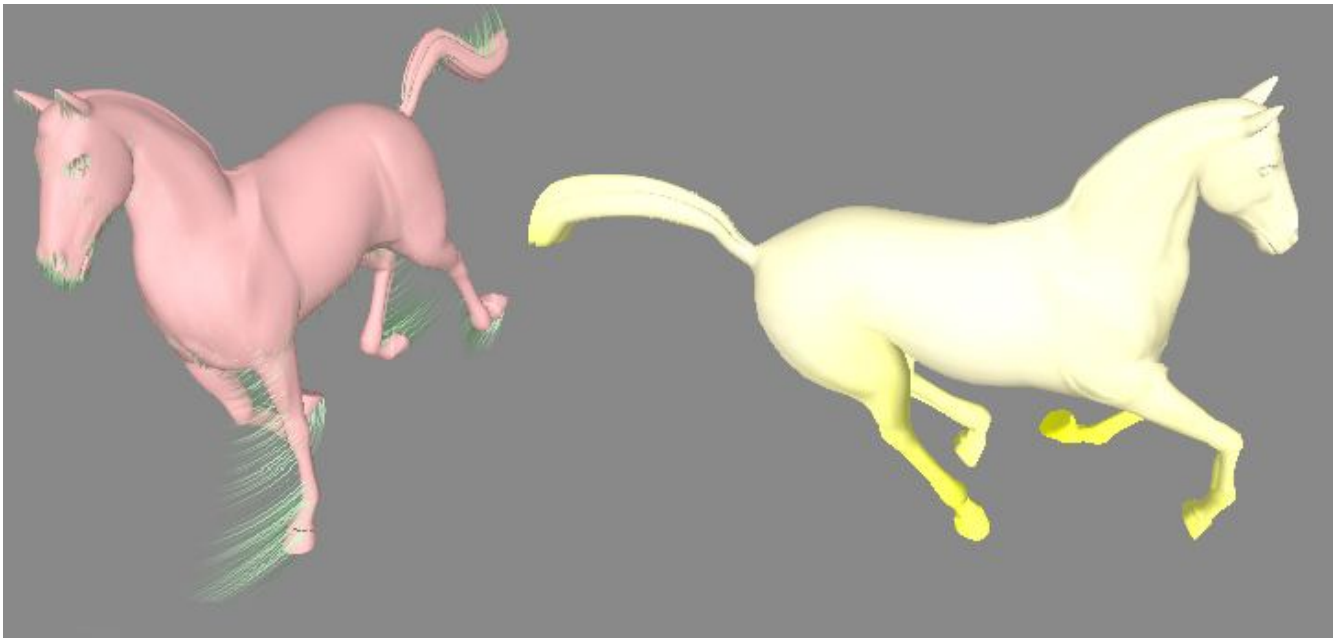
Motion Trails

By: Piotr Bugaj



Introduction

The goal of the project was to create the feeling and to display the idea of motion for a moving animated object through the use of rendering techniques such as drawing motion trails. Example of existing techniques include motion blur and after images, which are often used in animated movies and cartoons. This report will discuss five of such techniques that can be applied, in addition to some useful algorithms created.

Along with the five different techniques implemented, a friendly user interface was created in order to control the various parameters within each technique while testing and applying it to an object in motion. The object used within each animation was limited to the animated mesh of a horse.

Individual Lines

The first technique involved the rendering of thin individual lines being left behind of a moving mesh surface. More specifically, each vertex on a mesh surface, or a chosen number of vertexes, would leave behind a trail as the vertex would move through space, throughout the animation.

As the animation would play, the user, through the use of the user interface, would be able to change the colour of the trails, the interpolation used for constructing them, in addition to being able to alter the length, alpha value and the number of vertexes the trails would be created from to control the density of the lines displayed. This allowed for various experiments to be done while being able to observe the best results for specific parameters chosen.

The general construction of a trail for a given vertex A within a given time frame T, first involved finding the corresponding vertexes A_1, A_2,.. A_n that marked the location of vertex A within the time frames T-1, T-2, and so on, up to time frame T-n, where n would be a value representing the chosen length of the trail. This first step was very straight forward since the vertexes had identical numerical labels for every frame, thus making it easy to find the corresponding vertexes. From these set of vertexes, a line, which will be referred to as a motion trail, would be constructed and rendered along with the animated object.

A motion trail for a specific vertex would be constructed only if the direction of this trail faced the same direction as the normal at that vertex. In other words, the trail would only appear behind a moving surface.

Interpolation

Interpolation was used to smoothen out the edges of the motion trails. Without the application of interpolation, the trails would look very crooked and thus end up looking very unpleasant. The type of interpolation used was the Bezier Spline. The control points used for this spline where the vertexes making up the original motion trail.

The Bezier Spline lead to the display of wonderful results and thus was chosen for two reasons. Firstly, the computation required for constructing the interpolated lines was not very expensive and thus could easily be applied to points in three dimensions.

The second reason regarded the fact that the interpolation did not go through the control points. This was very important as it allowed for different shaped line to be created for each time frame the vertex passed through, thus ensuring that the line moving through the animation did not seem as if it was just moving through space through fixed defined points. The resulting effect looked very good as it gave the impression that the specific motion of the vertex from which the motion trail was being constructed from seemed to control and shift the entire shape of that trail - as oppose to the trail always moving through a fixed curve path. This is explained more clearly in figure 1 below:

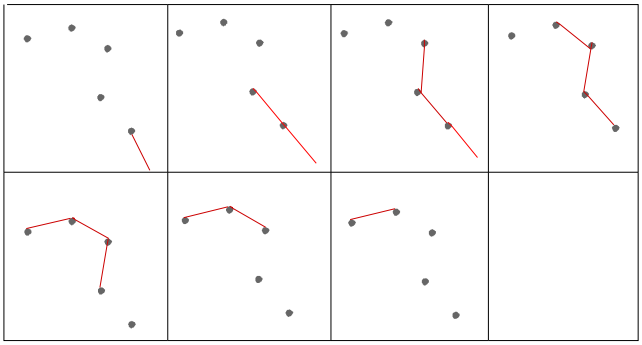


Figure 1 a: Line rendered without interpolation. Here the line moves through the vertexes. Lack of interpolation results in sharp edges, as can be seen in the figure.

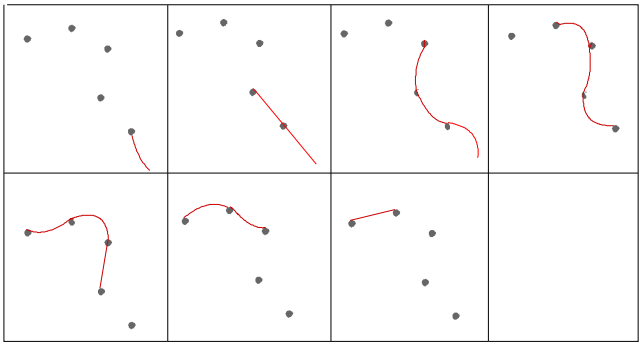


Figure 1b: Lines rendered with the use of interpolation. Here the interpolant ensures that the line passes through the control points. However the resulting effect does not look that great and hence such interpolation was not used.

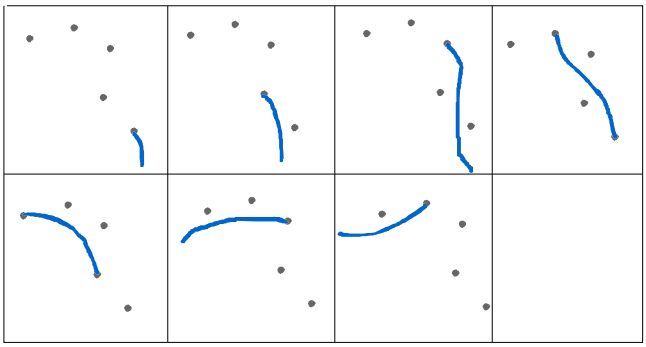


Figure 1c: Here the Bezier Spline is used to interpolate the points. The line no longer goes through the control points but instead is adjusted to fit in between the control points. This results in different parts of the line to constantly change between each time frame.

Altering of Alpha Values

The alpha values of the motion trails decreased from one hundred to zero percent, as the line was rendered from the starting location at its corresponding vertex to the its end point vertex where it became invisible. This gave the effect of a blur as multiple motion trails were drawn with the object in motion. Without the use of the changing alpha value, the motion trails looked very similar to strings being just attached to the moving mesh.

Another way that the alpha values had been assigned to the trails during experimentation was based on their length. Shorter lines where given high alpha values whereas long lines where given smaller values, as oppose to the alpha value decrease along the line. However this result was not very successful as the shorter lines looked very much like strings, whereas longer lines had not been visible at all. The comparison of these results can be seen in figure 4.

Length of the Motion Trails

The length of a motion trail referred to how many previous vertexes a line would be constructed from for a given vertex at the current time frame. If these vertexes were control points and the motion trail was interpolated, the length of the trail would then simply be equal to the number of control points used.

The chosen length of the trails rendered within most of the experiments had been four as it gave the best looking effect for the animated object used, however the user interface also allowed the user to decrease and increase the length. From observation, having the lines too short did not create the best effect as the resulting trails were not very visible. If the trails became to long they would end up blocking parts of the moving object. The comparison of these results can be viewed in figure 5.

Density of the Lines

The density of the lines referred to how many vertexes making up the mesh out of the total would leave behind a trail. Decreasing the density would render very few lines across the mesh, whereas a higher density would result in the trails covering up the view of the mesh and thus slowing down the program. From experiments, every third other vertex had been chosen to leave behind a trail. This gave the best looking result. The user interface also allowed the density value to be increased or decreased.

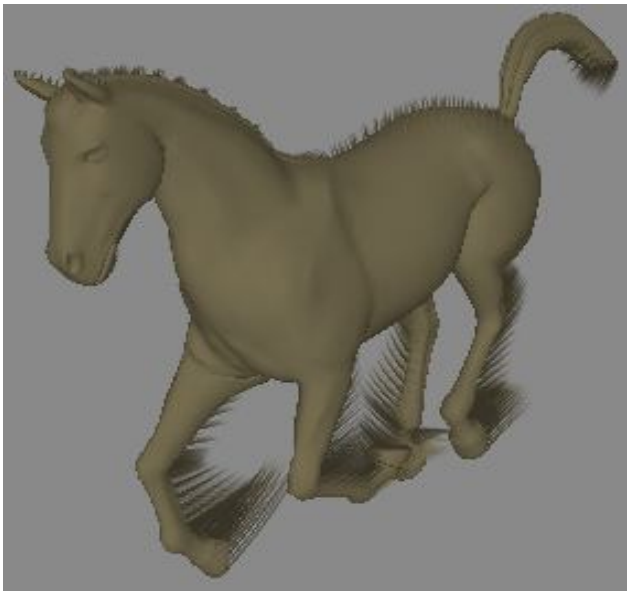


Figure 2a: Example of lines drawn for every vertex

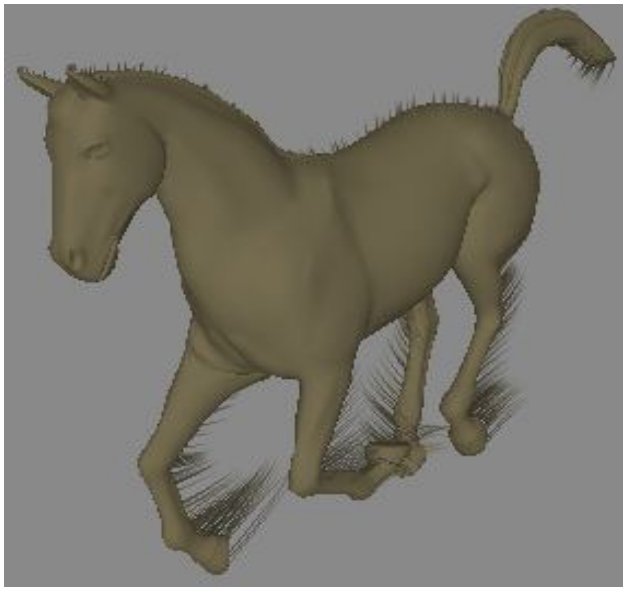


Figure 2b: Example of lines drawn every three vertexes

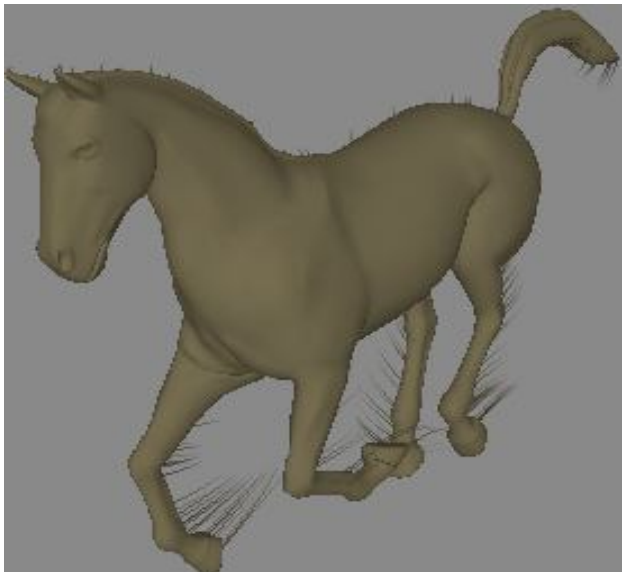


Figure 2c: Example of lines drawn every ten vertexes

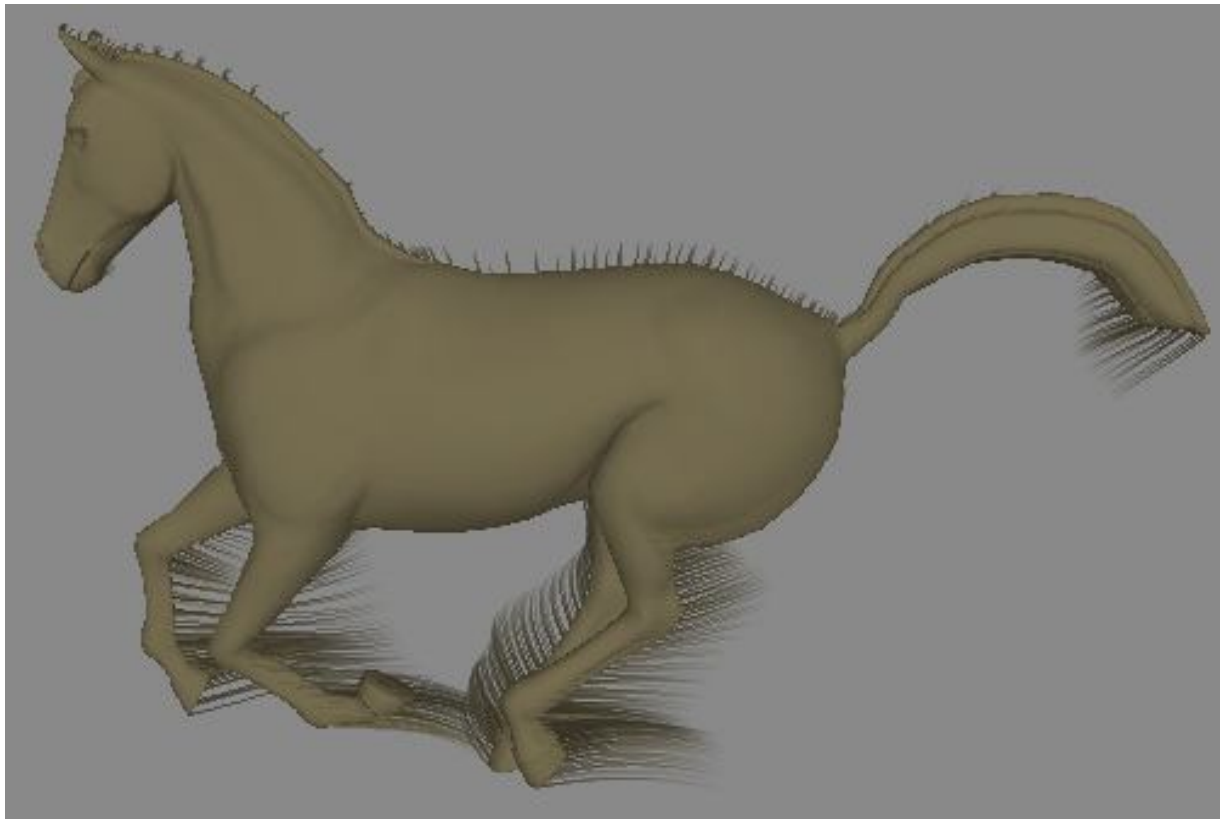


Figure 3a: Lines rendering with the use of the Bezier Spline interpolant.

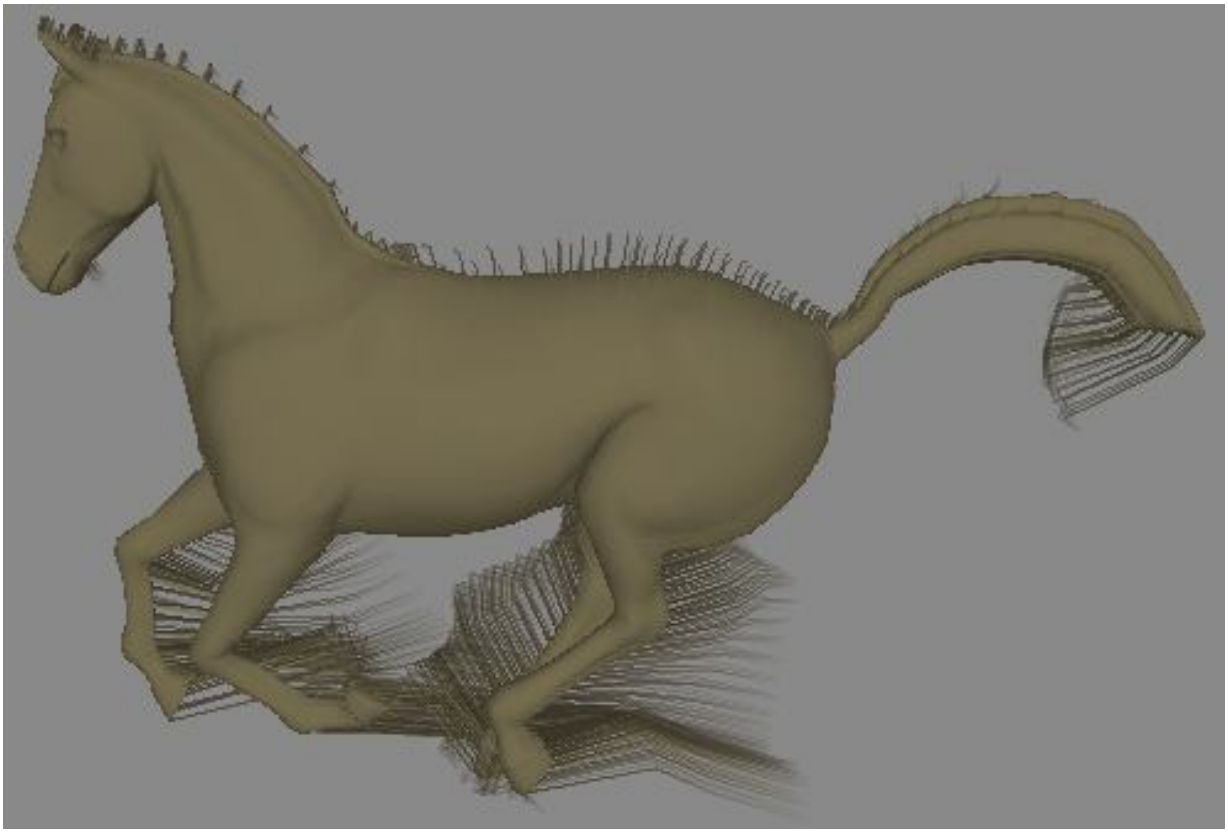


Figure 3b: Lines rendered without the use of any interpolation.

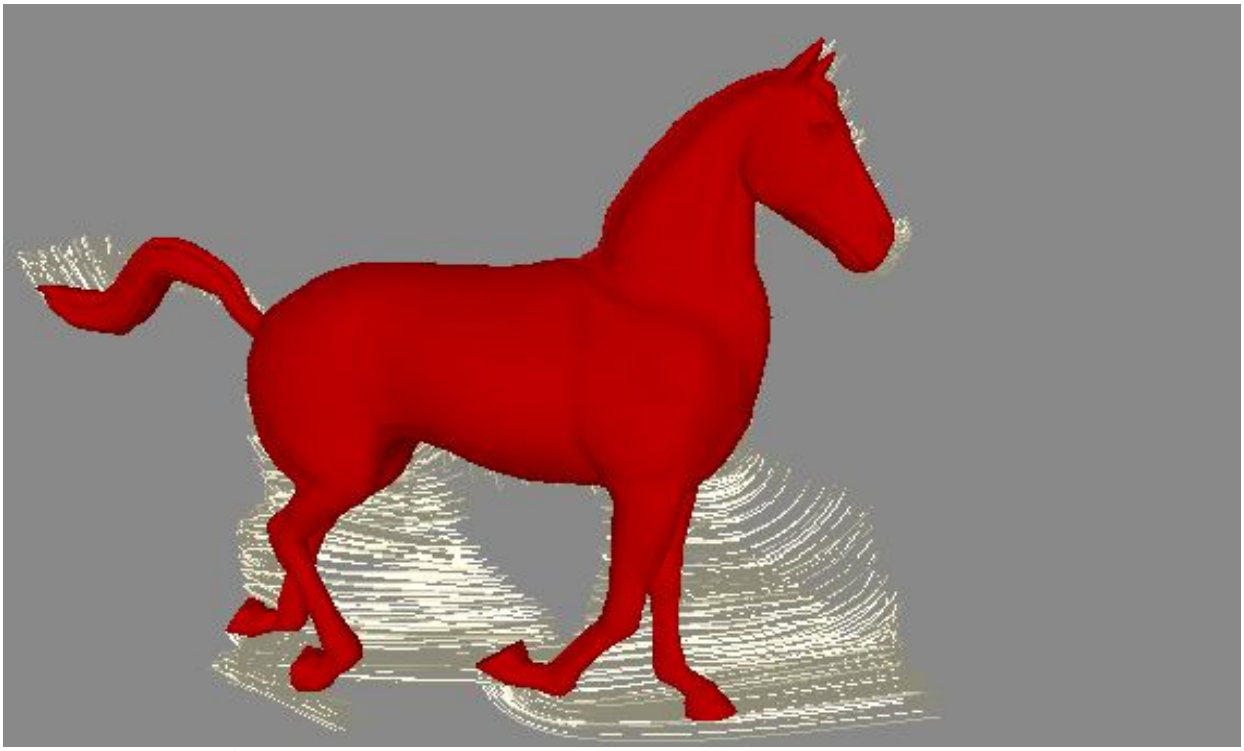


Figure 4a: Rendering of lines without changing the alpha values at all.

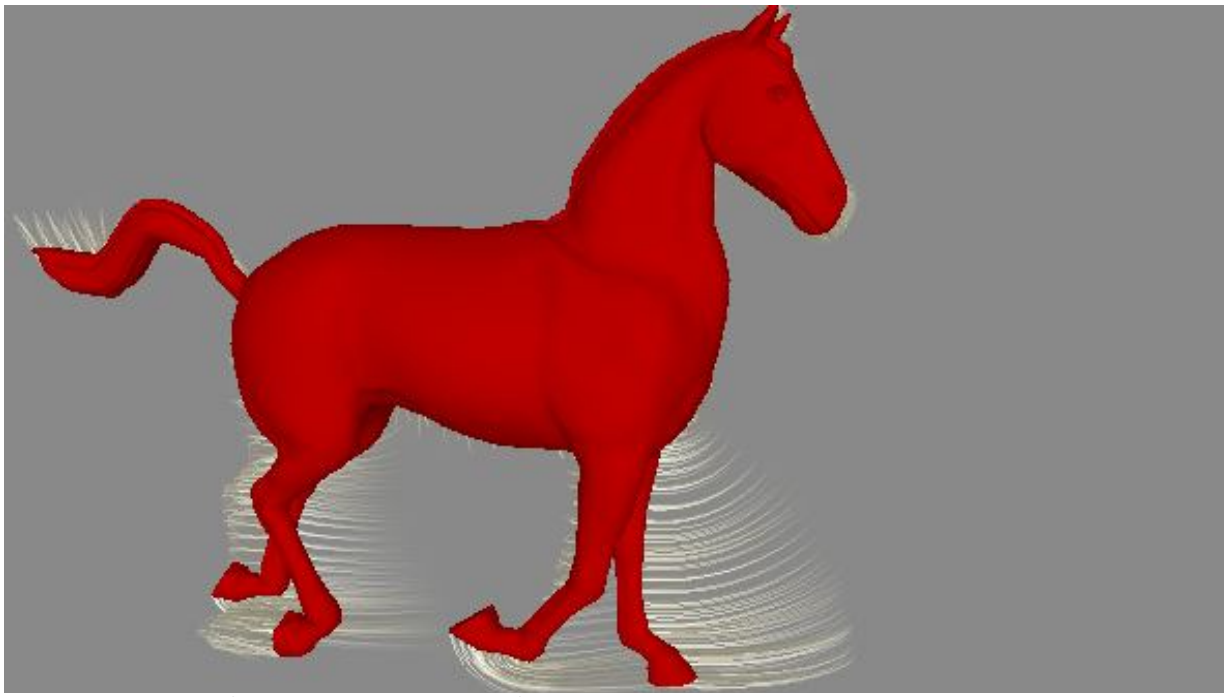


Figure 4b: Rendering of lines with alpha decreasing from one hundred percent alpha to zero.

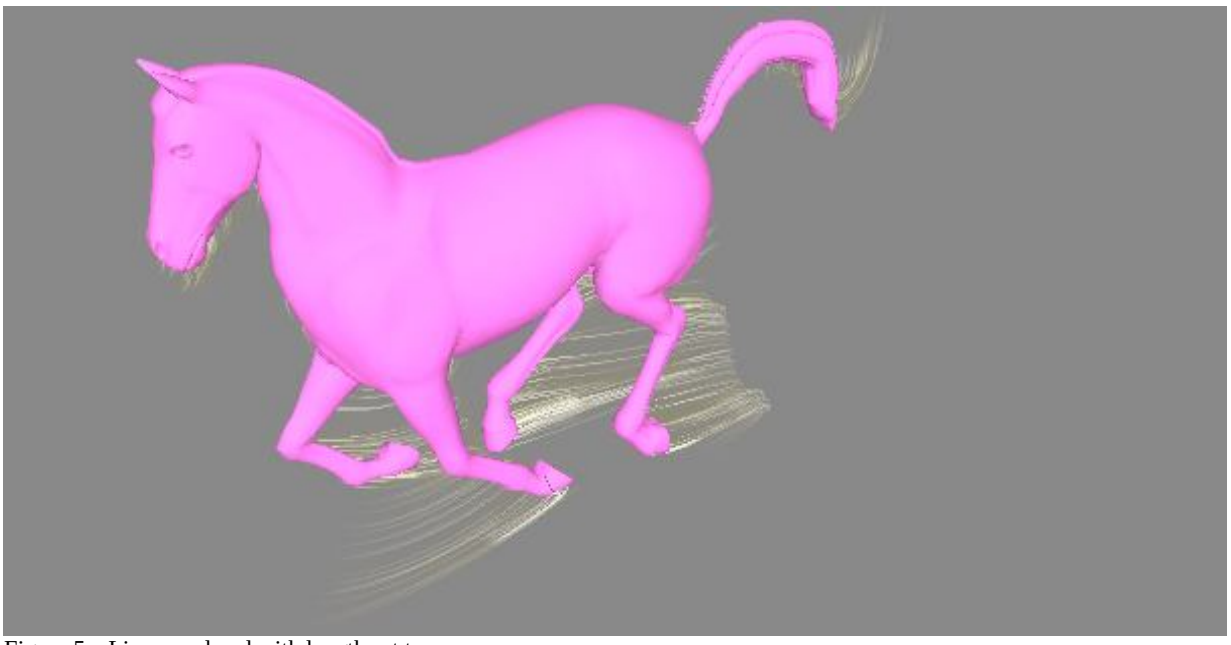


Figure 5a: Lines rendered with length set to seven.

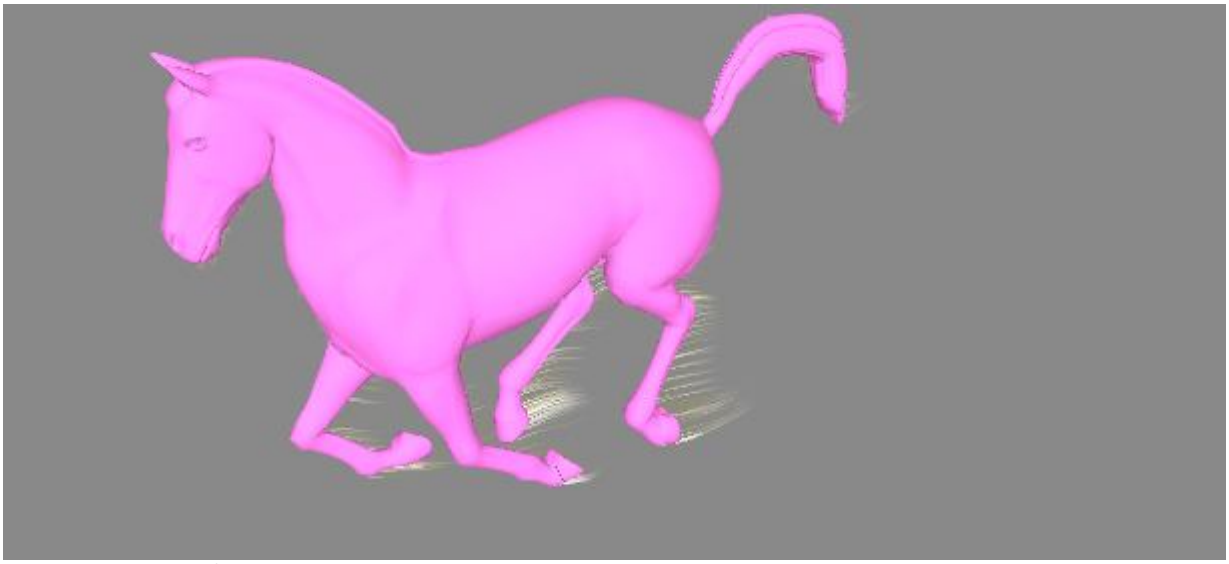


Figure 5b: Examples of lines rendered with length set to two.

Constructed Lines

The second technique developed was similar to the first one described above. Instead of a chosen vertex leaving behind a trail, a much larger line would be constructed from a triangular surface, known as a base, that would make up a small portion of the area of the triangular mesh of the moving object of interest.

Similar to the first technique, the user would be able to alter the length of the line, as well as the colour, interpolation, and the size of the base that the line would be constructed from.

Triangular Base

The triangular base is defined as a mesh surface of a circular shape belonging to the overal mesh of the moving object at the current time frame. From this triangular base, a line is constructed, with its starting point being that base. Thus, the lines constructed have a much larger diameter than the lines described in the first technique, where the lines are simply constructed from the diameter of one vertex.

Construction of Triangular Bases

The triangular bases where constructed by an iterative and recursive algorithm that would consider all the faces belonging to the mesh of an animated object, and return the required output. Then for a given time frame, bases that contained at least one face oriented in the opposite direction of the current motion vector would have a line be constructed from them. A more detailed explanation of this construction will be described in the next section, title; Construction of Lines.

The algorithm for constructing these bases, which will be referred to as Algorithm A, worked as following: It would first label all the faces, belonging to the to the animated object at an arbitrary time frame, as having been unmarked. It would then iterate through all the faces, and when encountering an unmarked face, it would send it to a recursive algorithm, called Algorithm B. Algorithm B would create a base from the face it had been given and thus execute. Algorithm A would then iterate to the next unmarked triangle and call Algorithm B again to create the next base.

Algorithm B would work by first taking the triangle given by Algorithm A, marking it as having been visited, labeling it with a recursive depth of zero, and then pushing it onto a stack. It would then continuously pop triangles of the stack until it being empty. When popping a triangle, Algorithm B would proceed as follows:

It would recursively visit each of the neighbouring triangles of the current triangle popped. If a neighbouring triangle is unmarked and the current triangle popped has a recursive depth label with value less that a specific numerical value, equal to the size of the base to be constructed, that triangle would be marked, labeled with a recursive depth value one values greater than that of the current triangle popped, and be pushed onto the stack.

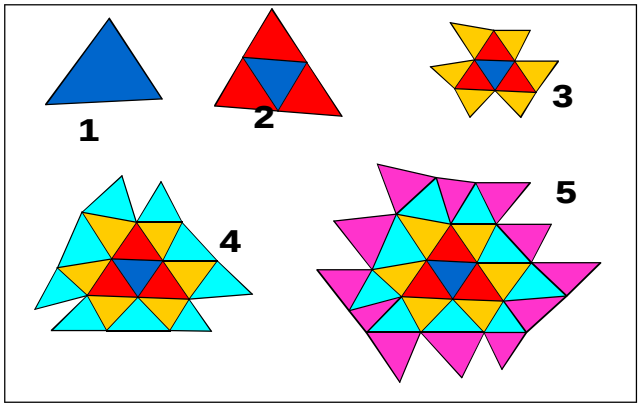


Figure 6: Example of bases of varying size, labeled from one to five. Here the blue triangle has a recursive depth value of zero, whereas the pink triangles represent a recursive depth value of four. Thus, triangle base one has a size zero, whereas base five is classified to have a size of four.

Once no more triangles are pushed onto the stack, Algorithm B would return a triangular base, that is, the triangles it has marked at the given iterative step of Algorithm A. Figure 7 shows the pseudo code example.

```
Algorithm A: Iterate through all triangles {
  if (triangle is not marked) {
    run AlgorithmB on triangle and store
    the triangular base created by Algorithm B
  }
}

Algorithm B (triangle start):
Label triangle start with recursive depth value
of zero, mark it as having been visited, and
push it onto the stack.

while ( stack is not empty) {

  pop triangle of the stack
  if (triangle is labelled with a recursive depth
     less than size) {

     visit each unmarked neighbour of
     current triangle and push them onto the
     stack, while labelling them with a recursive
     depth one greater than that of the triangle
     just popped
  }
}
Construct a base from the triangles visited
and return it to the function call made by
Algorithm A
```

Figure 7: Pseudo code for the algorithm used for constructing the triangular bases.

Using figure 7, a base of size three, for example, would consist of a base constructed recursively from triangles located by algorithm B up to a recursive depth value of three.

Construction of the Lines

Given a triangular base, the line would be constructed as following. First the vertexes making up the boundary edges of the base would be found. These vertexes would be marked as belonging to a set called level zero.

Next, for every vertex vi0 belonging to set level zero, a corresponding vertex vi1 marking the location of vertex vi0 within the previous time frame would be found, and these newly located vertexes would be marked as belonging to a set called level one. The creation of these levels would continue until the n^{th} level set would be created, where n would be the length of the line currently being constructed.

After the creation of each level, the centroid would be calculated for each level set of vertexes. Then, within each set, the vertexes would be moved loser and closer to their centroid, as the level would increase, from one to n, until finally, at level n, the vertexes within that set would all be touching their centroid. The following figure shows these steps visually:

Step 1: Construct a base of size k

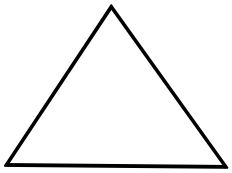
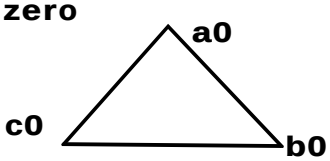


Figure 8 a: Step one of the line construction. The construction of the base is as described before.

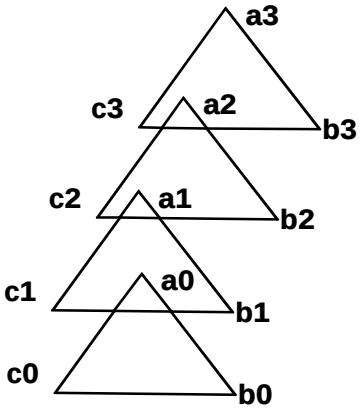
Step 2: Locate the vertexes defining the boundary of the base and put them in a group called level zero



level zero = a0, b0, c0

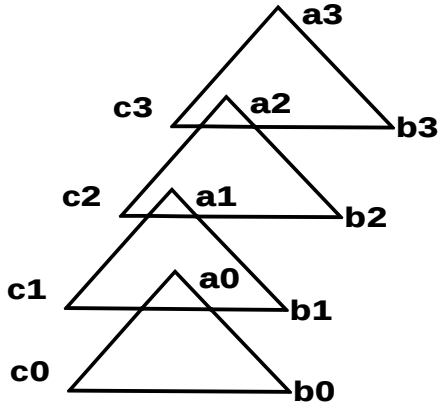
Figure 8b: Vertexes a0, b0, and c0 defining the boundary edges of the base.

Step 4: Interpolate any additional points between each level and replace the existing levels with the interpolated levels, if needed, if the spline does not go through the control points



Create a spline using x0, .. xn as control points, where x = a, b, or c and store the new interpolated levels

Step 3: Find the corresponding vertexes marking the location of the vertexes in level zero at previous time frames



level i = ai, bi, ci

Figure 8c: a0, b0, and c0 belong to the current time frame t. Vertexes ai, bi, and ci, are the vertexes that mark the previous locations of vertexes a0, b0, and c0 at time frame t-i.
Figure 8d

Step 5: Find the centroid, pi, for each level

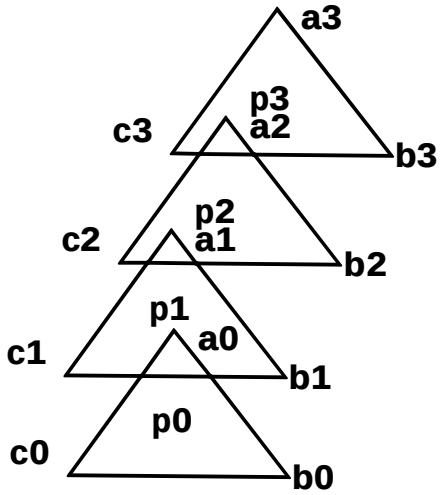


Figure 8e: Finding the centroid for each set of the vertexes, create for level zero to level n, here n being equal to three.

Step 6: Move the vertexes within each level closer and closer to the centroid as the level increases

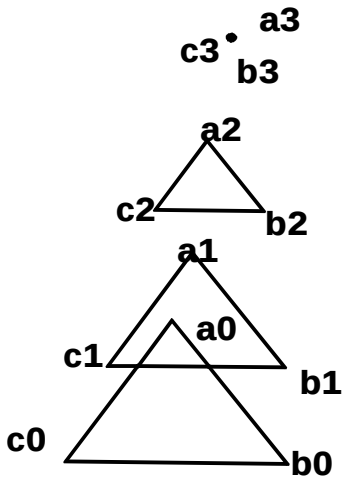
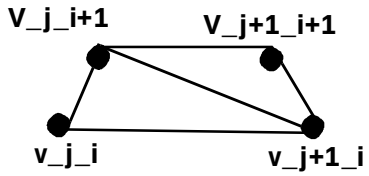


Figure 8 f: Step 6: Altering the location of each vertex with each set, moving each vertex closer and closer to the centroid within each level, as the level increase to n, here n being equal to three. The rate at which the vertexes get near the centroid is controlled by a function. In the experiments performed, the function was linear.

Step 7: Draw the line



drawTriangle(v_j_i, v_j+1_i, v_j_i+1)

drawTriangle(v_j_i+1, v_j+1_i, v_j+1_i+1)

Figure 8e: Here, j represents the numerical label of a vertex within a given level. Thus i marks the current level the vertex belongs to.

Base Size

The base size used was four. However the user interface also allowed other values for the size to be selected. Smaller values increased the number of lines being constructed from a single surface and hence the number of constructed lines actually rendered had to be reduced to decrease the overall density. Larger sizes were also very interesting as the lines rendered seemed to be a part of the mesh itself when displayed with a similar colour.

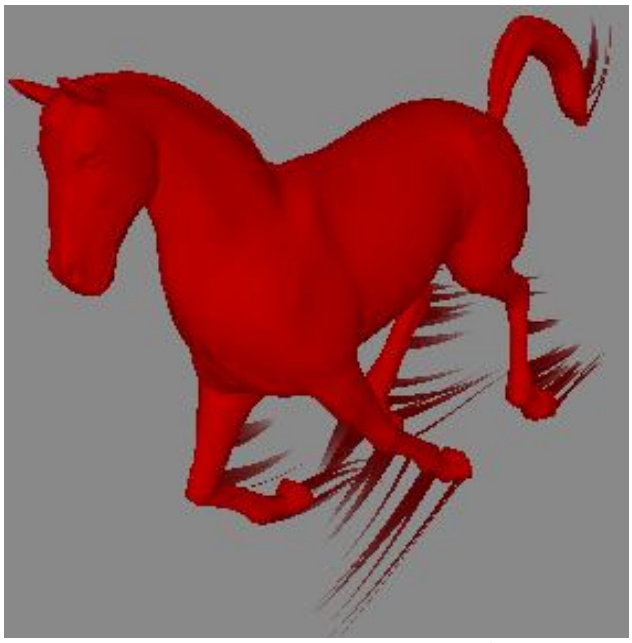


Figure 9 a: Lines rendered with a triangular base of size two. Density: every six of the constructed lines are rendered.

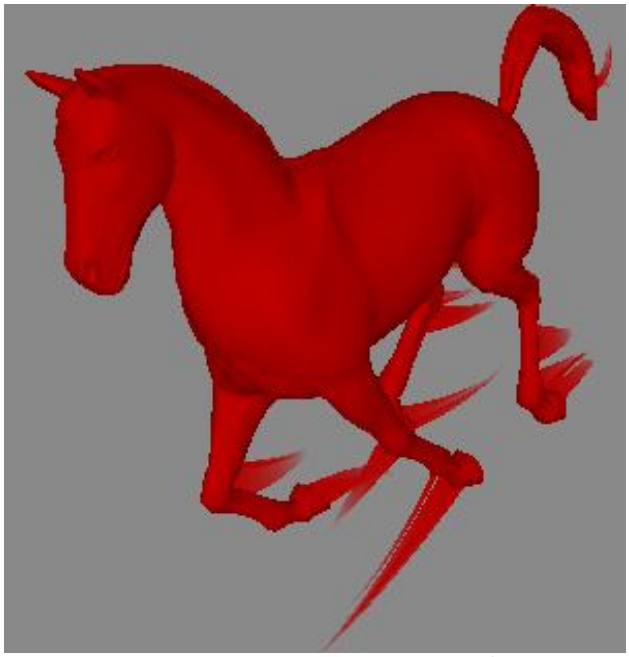


Figure 9 b: Lines rendered with a triangular base of size ten. Density: every two of the constructed lines are rendered.

Drawing Lines on Slanted Surfaces

One thing considered when drawing the lines is how they would appear together when being drawn on a surface that is on an angle with respect to the direction of motion, versus a surface that has a reverse normal normal equal to the direction vector. The problem with drawing lines on such a surface is that lines drawn at the part of the surface that is at an angle would end up being drawn more closely together. This would result in a bad visual effect. Figure 10 shows this example:

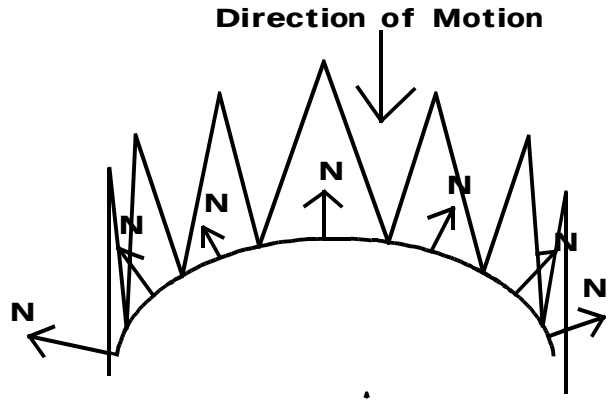


Figure 10 a: The side view of the lines being drawn on a curved saddle. Clearly, lines in the center are more spaced out versus the lines that are drawn at the surfaces where the face normals are more perpendicular to the direction of motion.

Figure 10 b: Same image as the one above in figure 10 a, however the front curve of the saddle is shown instead. As can be seen, from this view point, because much more lines are drawn at the slanted side of the surface, the density of the lines at that region becomes very high.

To fix this problem, the bases from which the lines have been constructed from have been altered such that: for each vertex making up the boundary edge of the base, the distance between that vertex and the centroid of that base was set to the minimum such distance. For example, if the vertexes making up a base are a, b, and c, with distance of one, two and three units away from the centroid, then vertexes b and c would be moved closer to the centroid such that their new distance from the centroid is also one unit away. Figure 10c shows the new result.

Figure 10 c: With the bases altered, the lines drawn at the sides are now much tinner, hence the unwanted density is reduced as wanted.

Shading Lines

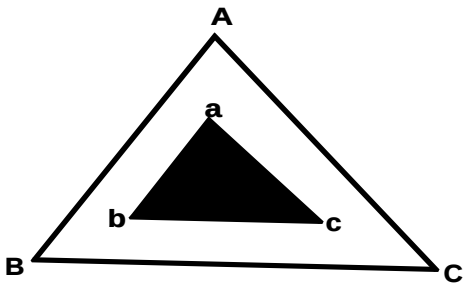
This technique involved using the motion trails to create the shade for the moving object. The moving object itself would be drawn with a constant shade and thus its texture would be depended on the motion trails the lines create.

Description of Algorithm

The algorithm for creating the shading lines would first start of by iterating through the faces of the moving mesh and mark the faces visible to the camera. Out of these visible faces, it would locate the faces that have a normal facing the direction of the reversed motion vector, and faces that have a normal at an angle of more than 90 degrees away from the reversed motion vector. These faces would be marked as 'Trailed Faces', and 'Untrailed Faces'. 'Trailed Faces' would be the faces that would have a trail being left behind them if used in the first and second techniques described before. Whereas 'Untrailed Faces' would have a normal that is in less than 90 degrees at an angle from the reverse direction vector of motion, and hence would have no visible trail.

The algorithm would then find the light intensity at the 'Untrailed Faces' and 'Trailed Faces'. The light intensity would be computed by first defining a specific light vector pointing into a similar direction as the direction of the camera. Then the light intensity at a face would be computed by taking the normal at the face, and dotting it with the light

vector, of course, with the vectors being normalized first.
After finding the light intensity at each of the faces, the algorithm would draw a black triangle of varying size in the center of each of the 'Untrailed Faces', such that the ratio of the area of the black triangle over the area of the 'Untrailed Face' in which the black triangle is being drawn inside equals to the light intensity. Figure 11a show an example of this:



light intensity at triangle ABC =
area(triangle abc)/area(triangle ABC)

Figure 11a

For the 'Trailed Faces', the algorithm would draw a black triangle in the center of the 'Trailed Face'. It would then located the previous position of the 'Trailed Face' with respect to the current time frame, find the centroid of that previous 'Trailed Face', and then draw lines from the vertexes of the current 'Trailed Face' to the centroid. These lines would form the overal shading line. Figure 11b shows this step:

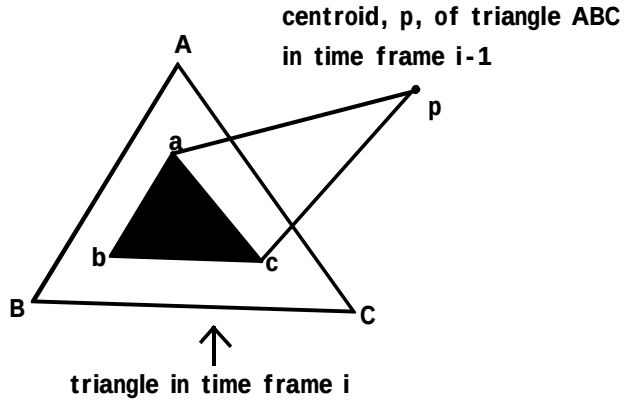


Figure 11 b

Following the representation in figure 11 b, the algorithm would then label the black triangle abc as region A. It would then find the intersection of lines ap and cp with line AB and label the intersecting points as i1 and i2. Finally it would mark the resulting quadrilateral , a, i1, 12, b, as region B. Figure 11 c shows this step with region A coloured in blue and region B coloured in yellow:

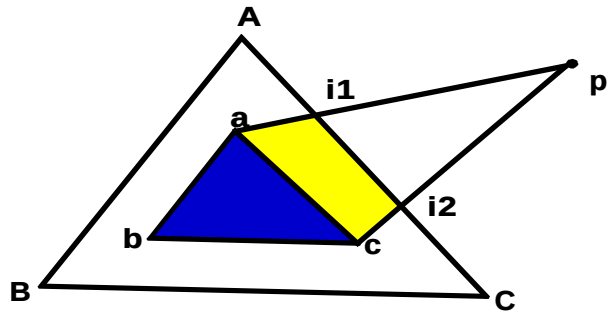


Figure 11 c

The technique would render region A at one hundred percent alpha, region B at fifty percent alpha, and triangle i1, i2, p at ten percent alpha. Thus the algorithm would adjust the size of triangle abc, such that the ratio of the sum of the area of triangle abc and the area of region B divided by two, divided by the area of triangle ABC would be equal to the light intensity.

$$\text{light intensity} = \frac{\text{area(region A)} + 1/2*\text{area(region B)}}{\text{area(triangle ABC)}}$$

Dealing With Shade From Overlap

The area of region B is divided by two since it is rendered with a colour at only fifty percent alpha. Triangle i1,i2,p would be rendered with ten percent alpha so that it would not create too much shade over any 'Trailed Faces' and 'Untrailed Faces' it might be overlapping that are neighbouring triangle ABC. Thus, it was also important, when choosing the 'Trailed Faces' and 'Untrailed Faces', that these faces themselves would be of a large size, constructed from smaller triangles, in order to reduce the total number of triangles making up the surface of the moving object the shading lines would be drawn on top of. With bigger triangles, the density of the lines would be smaller and thus the problem of triangle i1,i2,p overlapping other triangles would become very insignificant to the overal visual result.

Rendering Examples

Figure 12a shows the rendering of a horse with no shading applied. Figure 12b shows the same horse without shading, but with black triangles drawn within the visible faces, treating each visible face as an 'Untrailed Face'. Finally Figure 12d shows the rendering of the horse with trails drawn and with the technique taking account regions A and B for the 'Trailed Faces'.

Figure 12 a: Horse rendered with a constant shade with no shading lines applied.

Figure 12 b: Horse rendered with black triangles drawn inside the regions of the 'Trailed' and 'Untrailed' faces.

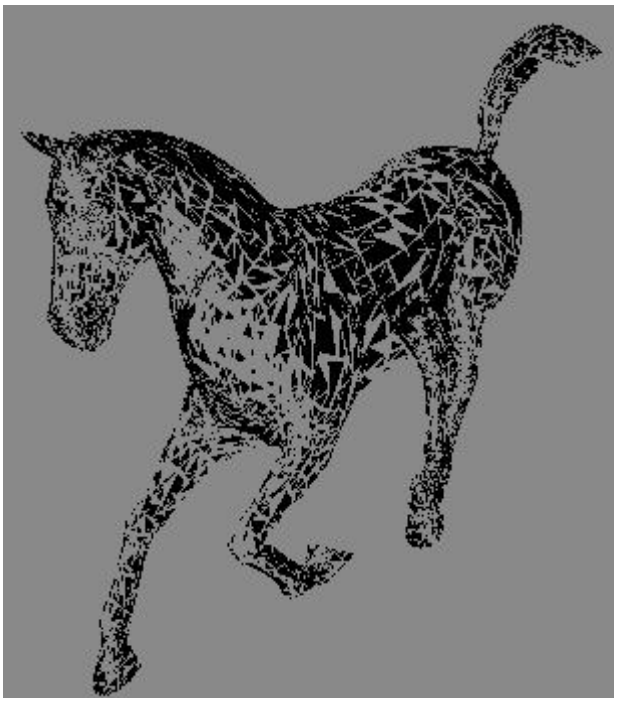


Figure 12 c: Same as figure 12 b but with no mesh visible. As can be seen,the black triangles applied form a nice shade across the surface of the horse.

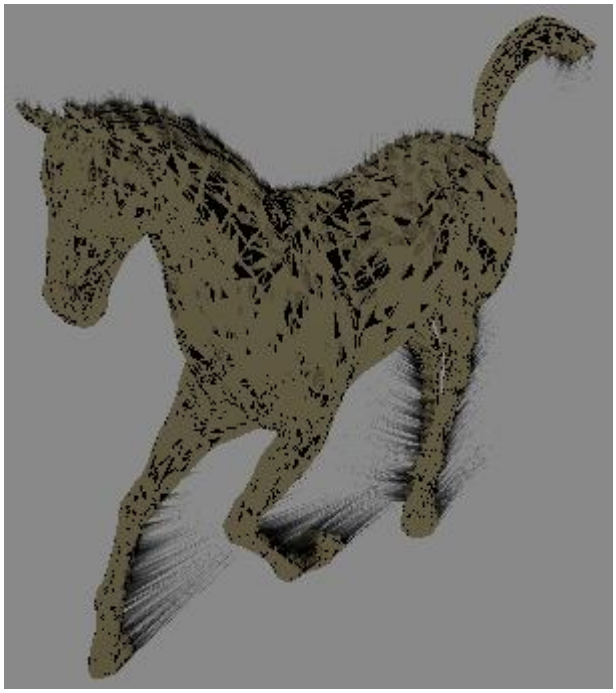


Figure 12 d: Rendering of the horse with the full technique applied to the creation of the trails.

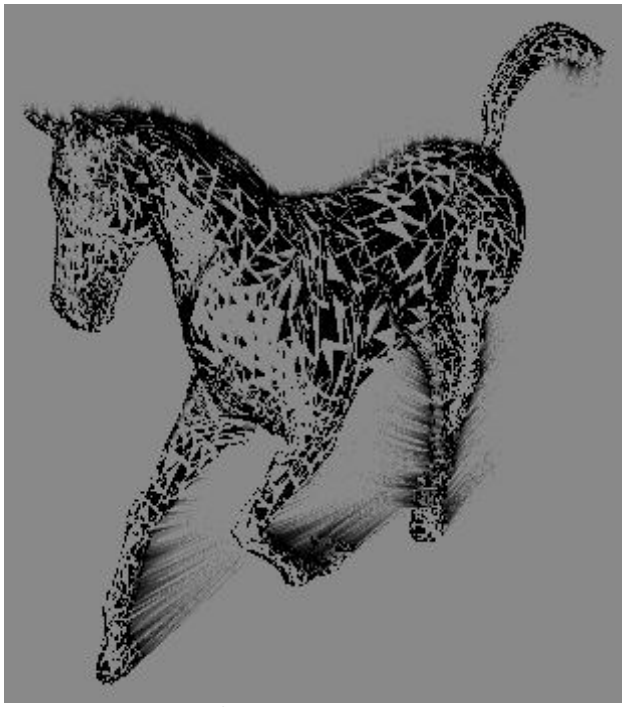


Figure 12 e: Same as figure 12 d but with no mesh created.

Choosing the size of the Inner Black Triangles

Choosing the size of black triangles within the 'Untrailed Faces' was very simple. First the light intensity at that face would be calculated. Next the algorithm would store a table of ratios. Each ratio would be equal to the area of a small triangle divided by the area of a larger triangle surrounding it. These ratios would vary from zero to one, and the size of the table would contain about a hundred of these values, though any bigger number would also be reasonable. Then the algorithm would store a second table, corresponding to the first one, that would store a second set of ratios. These ratios would store the length of an arbitrary side of a small triangle divided by the corresponding length of the bigger triangle, forcing the assumption that all the black triangles drawn within the region of the bigger triangles are similar to those bigger triangles.

Finally, using these two tables of ratios, whenever the algorithm would need to draw a small black triangle inside of a bigger triangle such that the areas of the triangles would match the light intensity, it would look into the second table and figure out the length of the sides of the smaller triangle with respect to the length of the sides of the bigger triangle. Thus, it would create the small triangle using those side lengths.

The ratios within the tables varied depending on how the triangles where shaped, depending on whether they had a small or large obtuse angles, for example, however this variance was extremely small, and hence the results looked successful with the use of these precomputed ratios.

When trying to figure out the size of the inner black triangles for the 'Trailed Faces', the task was a lot more difficult as the size of region B would vary based on the location of the centroid of the previous inner triangle with respect to the time frame. Thus when choosing these triangles, their size would constantly be adjusted until the ratio of the area of region A added to half the area of region B, divided by the area of the outer triangle would much the light intensity within a given number of while loop iterations.

The size of the inner triangle was adjusted by first setting the ratio of the side length of the smaller triangle over the side length of the larger triangle to be equal to one half. Next the algorithm would see if the resulting size of the small triangle would lead to creating regions A and B, such that the ratio of the areas would equal to the light intensity, plus or minus an epsilon. If successful, the algorithm would execute. Otherwise it would increase or decrease the side length of the smaller triangle such that the ratio of the sides would increase or decrease by one forth, and then by one eight, and so on, until the while loop would reach a certain counter or the resulting regions A and B would yield the correct ratio equal to the light intensity, plus or minus epsilon. The algorithm would decrease the side length of the small triangle if the ratio of the areas was greater than the light intensity, and increase the side length if the area was smaller than the light intensity. These iterations were very successful and the algorithm was usually able to find the correct ratio of the areas within ten or less iterations. The maximum iterations allowed was twenty.

Calculating Areas of Regions A and B

The algorithm had to consider five different cases when calculating the areas of these regions depending on where the centroid, p, of the previous 'Trailed Face' was located with respect to the location of the 'Trailed Face' in the current time frame from the perspective of the camera vector. These cases are described in figure 13:

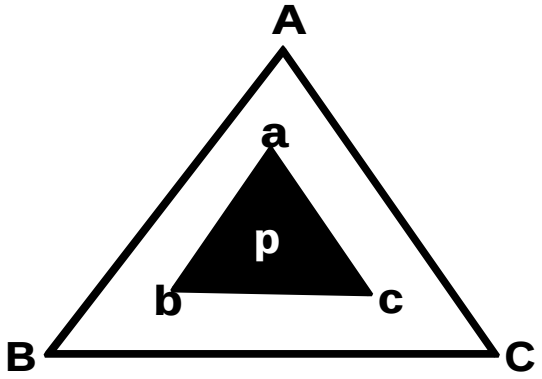


Figure 13 a: The centroid p is located on top of triangle abc. In this case only the area of region A is calculated.

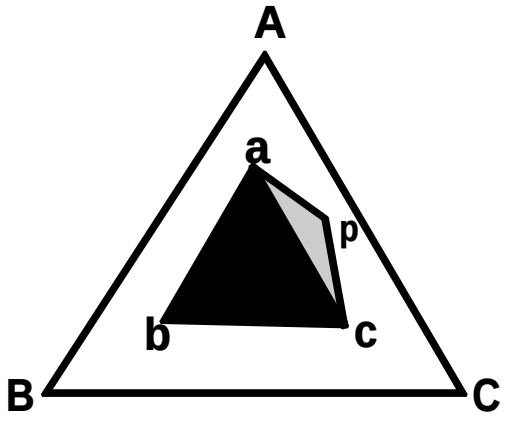


Figure 13 b: Case where e the centroid p lies outside of triangle abc, inside triangle ABC, and appears on the one side of either the side ac, ab, or bc.

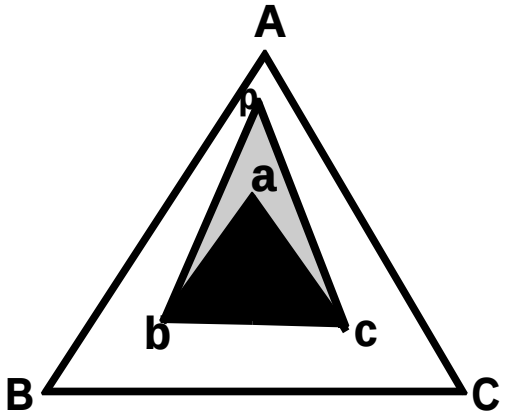


Figure 13 c: Case where the centroid p lies outside of triangle abc, inside triangle ABC, and appears on the two sides of either the sides ab and ac, ba and bc, or ca and cb

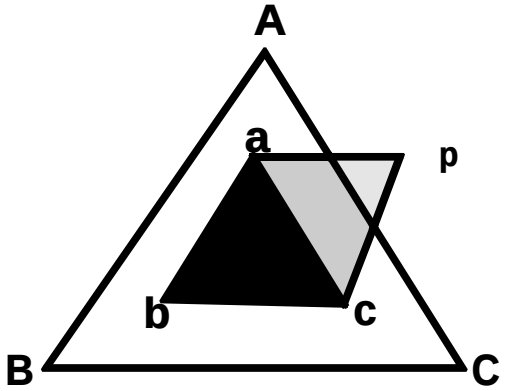


Figure 13 d: Case where p lies outside of triangle ABC and lies on one side of either side ac, bc, or ab

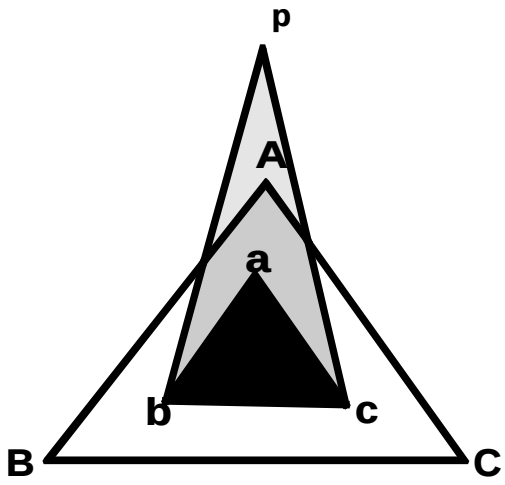


Figure 13 e: Case where p lies outside of triangle ABC, and appears on two sides of either the sides AB and AC, BA and BC, or CA and CB

Choosing the Boundary of Region B

One of the difficulties encountered when rendering the shading lines was locating the exact point along the line where the boundary of region B would exist. This boundary was not easy to render on the line as it would sometimes be required to be drawn at varying angles across the line. Figure 14 shows this example where the line being rendered is coloured in green and the boundary of region B is marked in orange:

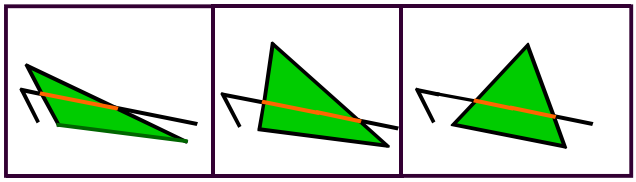


Figure 14

Rendering the line with the alpha changing exactly at the boundary was therefore very difficult and computationally expensive, as the line had be divided into smaller triangles. This computation was done, however the rendering was not perfect due to the lack of time required for debugging.

Reducing the Contrast of the Inner Triangles

A second problem encountered while testing the technique, resulted from the fact that the dark triangles drawn within the visible faces to create the shade where sharply visible on top of the mesh when no visible trail was leaving them. I.e., the centroid p existed inside the region of triangle abc or triangle ABC.

To help fix this, the moving object was first shaded normally, as oppose having just a constant shade applied. This was done by choosing a constant colour C, and then assigning each face F the colour A(F). A(F) would be equal to colour C multiplied by the light intensity at face F.

Next, the colour assigned to face F would be changed to be closer to colour C based on the length of the trail leaving that face. So for example, if a face had a short trail, or it is the case where the centroid p is within the region of triangle abc, the colour of face F would remain to be A(F). However if the face would have a long trail leaving it, its colour would be changed to be closer to that of the constant colour C.

Similarly, the shading line leaving a face F would be assigned the colour A(F) if they were extremely short. However, as the length of the shading line would increase, its colour would become darker and darker, and differ more and more from the colour A(F).

This resulted in small black triangles that no longer appeared on the surface of the mesh where no shading lines where being rendered. The following results are shown in figure 15:

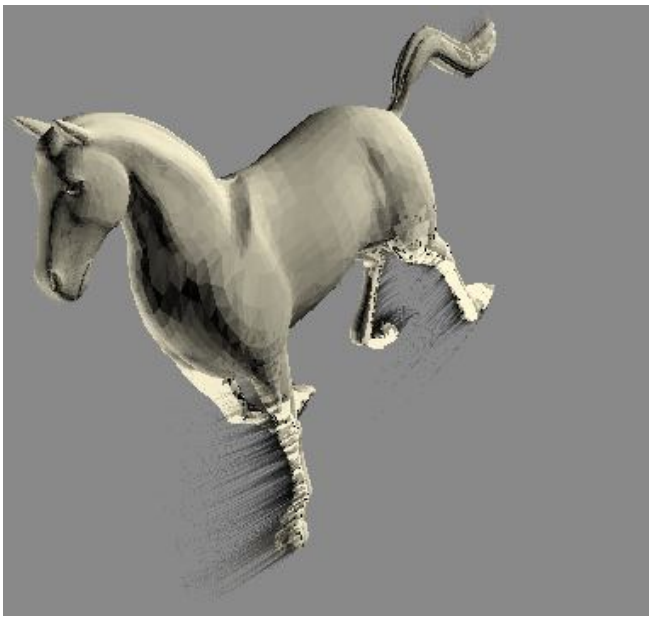


Figure 15 a: Example of shade lines applied where the colour of the lines is more similar to the actual shade at that face, if the line is short. Also, as can be seen, parts of the mesh where long trails appear have a much lighter colour applied, that is closer to the constant colour C.

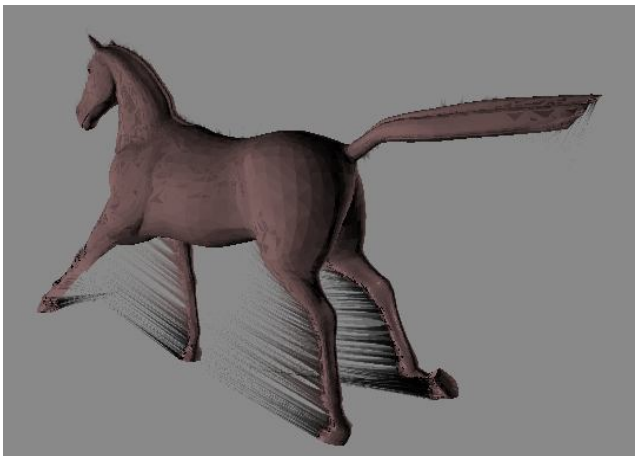


Figure 15 b: A second example. The shorts trails here are drawn with a much similar colour as used for the mesh. Whereas parts of the mesh with longer trails are drawn with a much lighter constant colour, with the trails themselves being much darker.

Motion Sensors

The fourth technique involved changing the colour and alpha value of the mesh in areas where the motion of the vertexes is much greater than average, and keeping the colour and alpha value constant in areas where there is no motion at all. The implementation of this was very simple and the result where interesting to view. Figure 16 shows examples of this technique used:

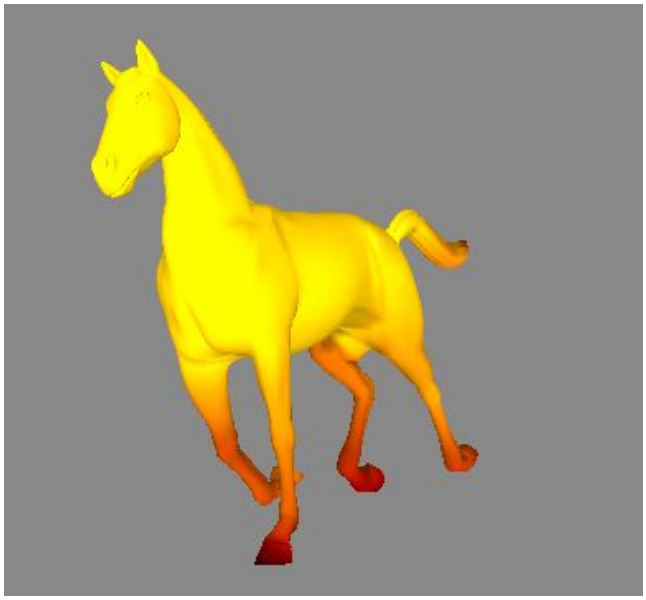


Figure 16 a: Example of the application of the Motion Sensors technique. The parts of the horse which are in motion (the legs), are rendered with a different colour then the rest of the mesh.

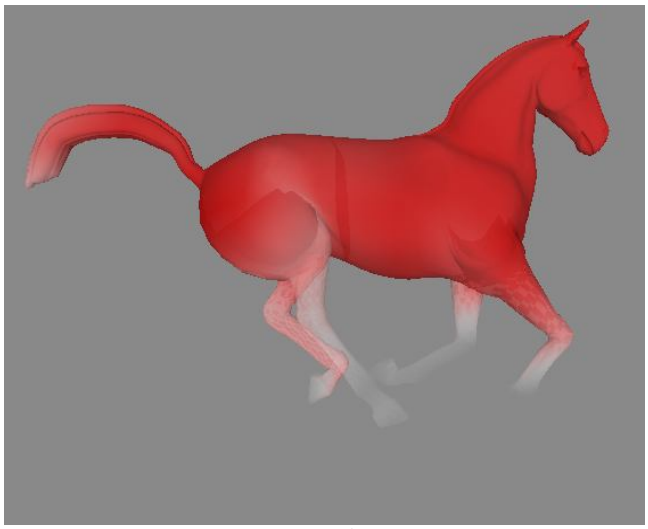


Figure 16 b: Another application of the motion sensors technique with the alpha value reduced at parts of the mesh with a lot of motion.

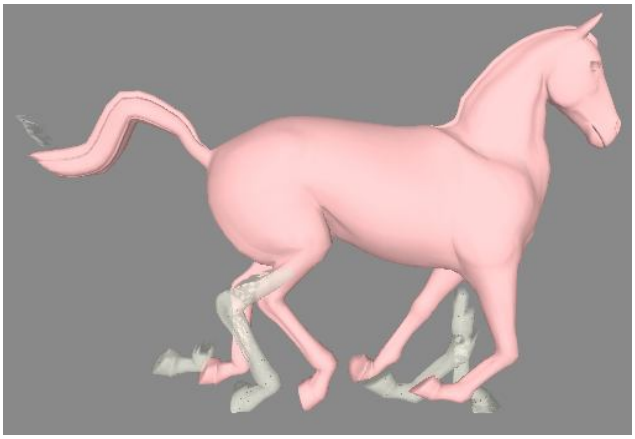


Figure 17: Example use of the after images technique. Here, additional mesh surfaces rendered in white are drawn behind the moving legs of the horse, which are the parts of the mesh undergoing the most motion.

As can be seen, the results of rendering the after images had been very successful. However one problem encountered is that when the entire object moved from one frame to another, the after image might end up representing more parts of the mesh than the user would like to see, hence creating an effect not looking that great. To fix this, when measuring the distance between all the faces between the current and previous time frame, the average distance would be subtracted from the position of each vertex making up the triangles in the current time frame. After this, even if the entire horse was to be moving in figure 17, for example, as oppose to it being still with just its legs being in motion, the after images rendered would still only represent the moving legs.

After Images

The fifth and final technique implemented had been the creation of mesh surfaces being rendered behind a mesh surface in motion. These mesh surfaces would represent parts of the entire mesh that existed in the previous time frames with respect to the current time frame. More specifically, these mesh surfaces rendered for a given time frame T would represent parts of the entire mesh that underwent a certain degree of motion from the previous time frame to the current time frame T.

To find these mesh surfaces, which will be defined as after images, the algorithm would first find all the distances between the faces of the mesh in the current time frame T from the corresponding faces in the previous time frame T-1. It would then take the average of these distances and compute the standard deviation.

The algorithm would then go through the faces in belonging to the mesh at the current time frame and locate all the faces that traveled a distance greater than the average distance plus one and half the standard deviation value. After finding these faces, the algorithm would render them with the object in motion. Figure 17 shows an example of this technique.

Multiple After Images

Rendering a trail of two or more after images has also been considered. To create this trail, the procedure was very similar to the creation of just one after image. The algorithm would first compare triangles between the current time frame T and the previous time frame T-1, and draw the mesh surfaces belonging to time frame T-1 that under-went the most motion. It would then compare triangles between time frame T-1 and T-2, and then draw the mesh surfaces belonging to time frame T-2. The algorithm would continue this until it would compare triangles between time frame T-n+1 and T-n, where n would be the length of the trail of after images.

One thing that was considered however is that the after images drawn from time frame T-i would have to contain triangles that existed in time frame T-i+1. This ensured that the after images rendered for the ith previous time frame did not appear out of no where without any trail of after images existing between the after image at time frame T-i and time frame T-1. Figure 18 shows an example of multiple after images being used:

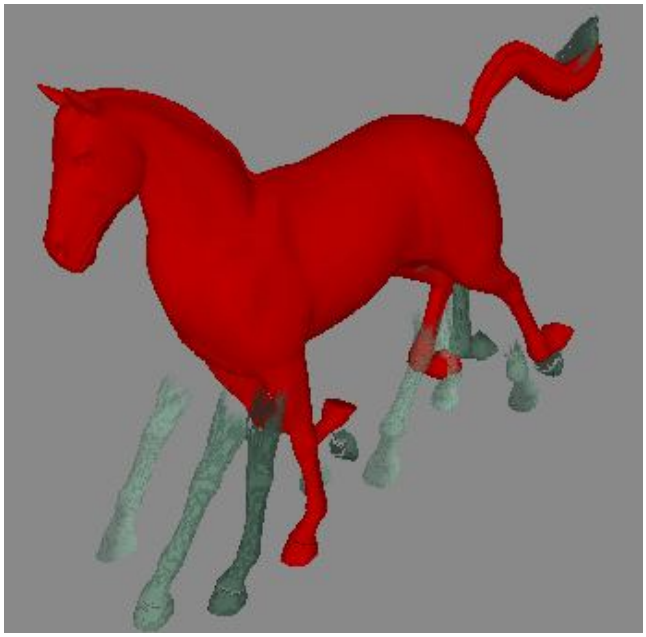


Figure 18

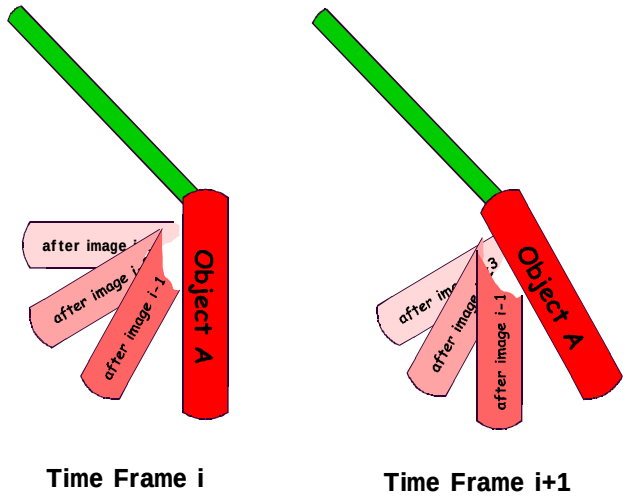


Figure 19 b: Example of the after images being shifted more closely together towards the current motion vector of object A.

Compression and Expansion of the Trail

Another thing considered when creating a trail of after images was an idea very similar to the one presented in the first technique. When the trail of after images was drawn without any modification, the after images seemed to be replacing their location within the previous time frame they existed in. To fix this, as they trail of after images grew9nger, the after images themselves would be shifted closer or further apart from themselves in the direction of corresponding mesh surface moving in the current time frame. Figure 19 explains this more clearly:

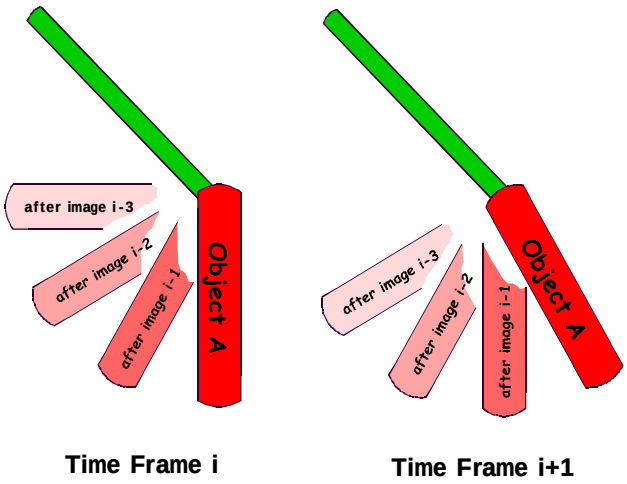
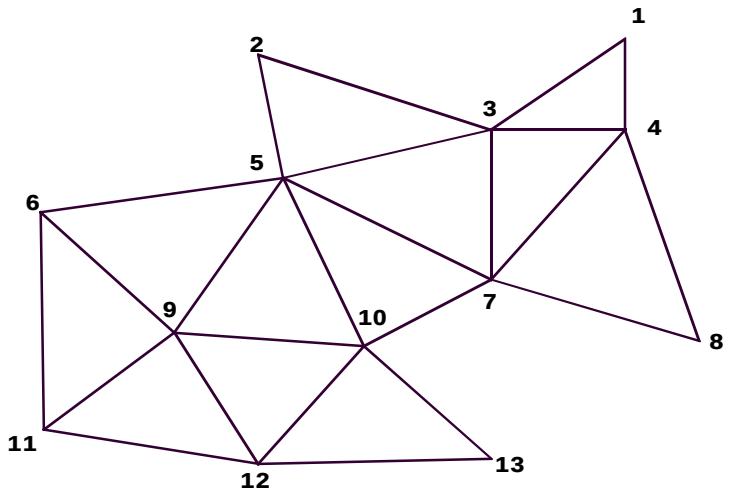


Figure 19 a: Example of a trail of after images being created behind a moving object A. As can be seen, after image i-3 in time frame i replaces the exact location of after image i-2 when rendered in time frame i+1. This resulted in an effect that did not look very good.

Reconstruction of Faces Using Breadth First Search

One useful algorithm developed had been the reconstruction of faces, given just the vertexes and the corresponding edges connecting them. The goal of the algorithm was to run in linear space with respect to the number of triangles, using the assumption that the triangles created a mesh surface. I.e., the number of neighbouring triangles for each face was assumed to be three and each vertex was assumed to be connected to at most a constant number c of neighbouring vertexes. Hence, given n faces to construct, the run time of the algorithm would be O(n*c).

The algorithm was quite simple and worked as follows: It would have a queue Q containing the vertexes to visit next, and an array A for storing the vertexes and some, but not all, of its neighbouring vertexes, and another array keeping track of which edges have been marked. Figure 20 below shows an example of the vertexes, their edges, and an array A that might be created by the algorithm:



Array A:

vertex	neighbours
1	null
2	3
3	1,4
4	1
5	3, 2
6	5
7	3, 4, 5
8	4, 7
9	5, 6
10	5, 7, 9
11	6, 9
12	9, 10
13	10

Figure 20

The algorithm would then pick a starting vertex S and push it onto the queue Q, and place it onto the array A. Then, as long as the queue is not empty, the algorithm would pop the next vertex of the queue and label it as vertex N.

It would then look through all the neighbouring vertexes of vertex N, and for each of these vertexes, i, it would check if the edge connecting the vertex i to vertex N is already marked. If it is, it would ignore this vertex and move onto the next neighbour. Otherwise it would check if this neighbouring vertex i already exists in the array A. If it does, it would add the vertex N as an additional neighbour of the vertex i. If the vertex i does not yet exist within the array, the vertex would be added to this array with its first neighbour being set to be vertex N. Finally the algorithm wold add this vertex i onto the queue and marked the edge i-N as having been visited. The algorithm would then iterate to the next neighbouring vertex of vertex N. Once all the neighbouring vertexes of vertex N have been iterated through, the algorithm would pop the next vertex of the queue.

Finally the algorithm would go through each vertex stored in array A. If the vertex has more than two neighbouring vertexes, it would try out all the possible pairs from those neighbours and check if the pair plus the vertex being iterated through forms a triangle, and store it as a triangle if that triangle has not yet been found.

This algorithm was very similar to just looking through all the possible neighbouring vertexes for each vertex and then trying out all the possible pairs from these neighbours to see whether or not the pair plus the vertex currently being iterated forms a triangle that has not yet been created. The only difference is that this procedure is a lot more computationally expensive since each vertex usually has about six to seven neighbouring vertexes, and hence, the number of combinations becomes large. However, with the use of the breadth first search, each vertex in the array A would only have a few of its neighbours assigned to it out of the total. Thus, the sum of the neighbouring vertexes for each vertex in array A would be equal to the total number of edges, and each vertex would usually only have between zero and four vertex to which the combination of all pairs would have to be created from.

Clearly this algorithm is correct as it marks every single edge during the breadth first search, and hence ensures that when constructing the triangles, all the edges have been taken into consideration.

Visual Description of the Algorithm

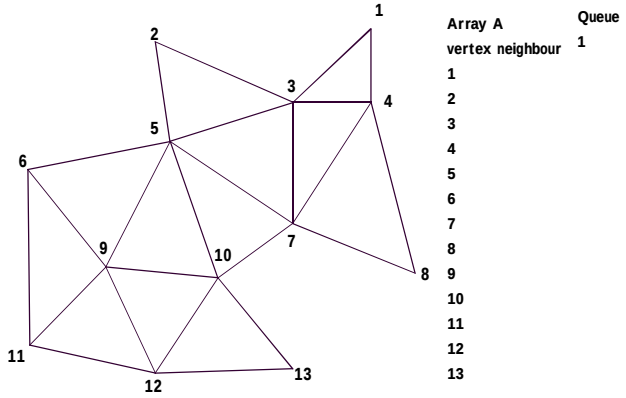


Figure 21 a

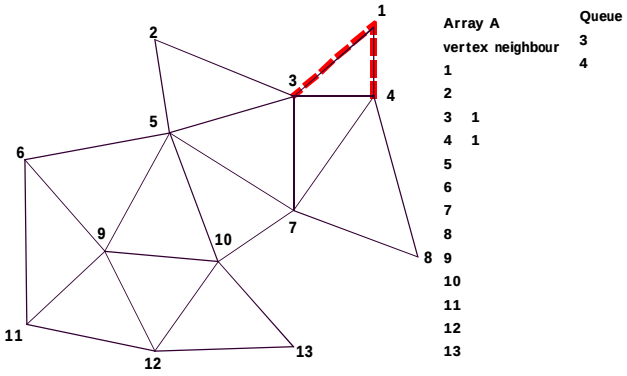


Figure 21 b

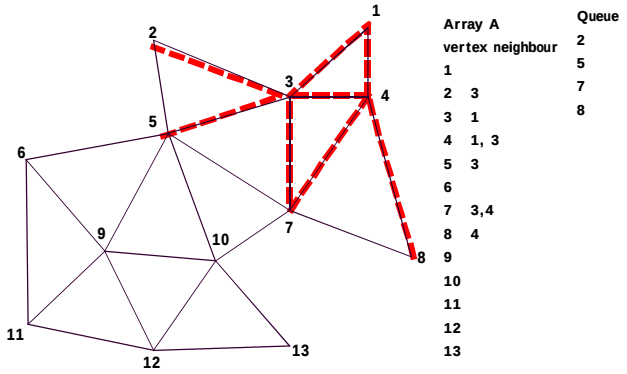


Figure 21 c

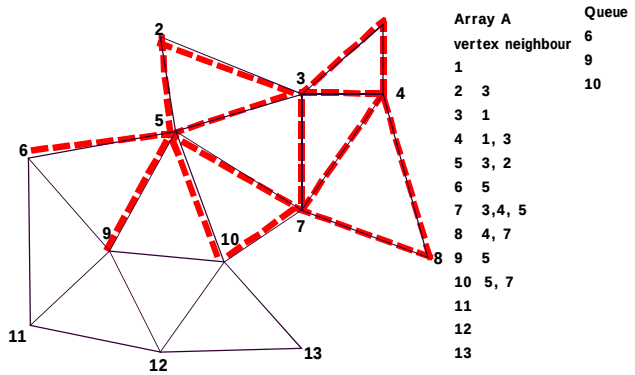


Figure 21 d

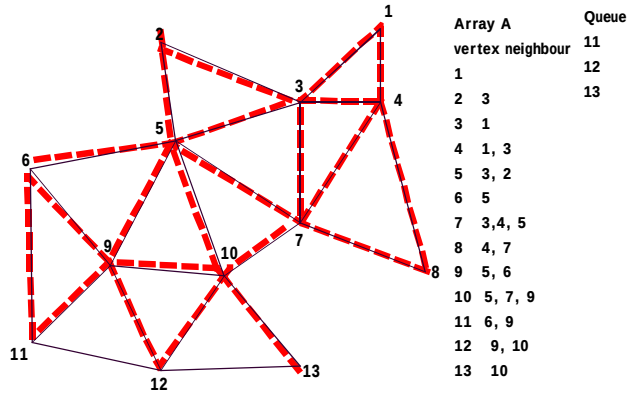


Figure 21e

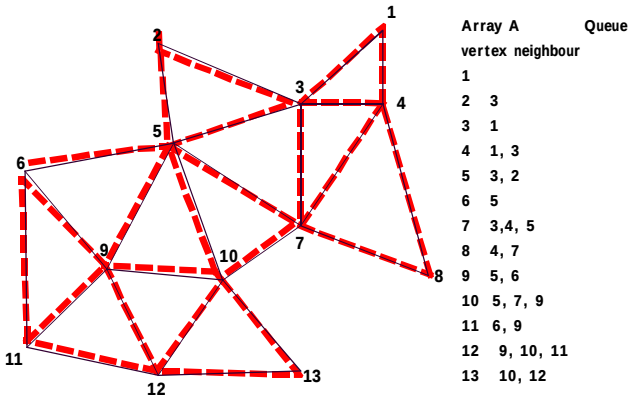


Figure 21 f

Locating Rigid Mesh Surfaces Between Subsequent Time Frames

Another useful algorithm developed was used for locating rigid mesh surfaces between two subsequent time frames. This algorithm was not used often due to the expensive computations required, and hence was not used in any technique described above, however it did well at segmenting parts of the mesh. For example, when using this algorithm on the mesh of the horse, the algorithm would segments the moving legs and the neck.

The algorithm worked by first iterating through all the vertexes making up a mesh object in the current time frame T. For the current vertex being iterated, the algorithm would check if this vertex has been marked (originally all the vertexes are left unmarked). If the vertex is already marked, the algorithm would iterate to the next vertex. Otherwise it would perform the following steps:

It would label the current iterated vertex as vertex S. It would then iterate through all the neighbouring vertexes of vertex S and find all the pairs of neighbours that form a plane with vertex S. For example, if the pair of vertexes are A and B, then the angle ASB should be at least smaller that 170 degrees, as a straight or near to a straight line cannot be used to create a plane. The algorithm would then check if the vertexes A, B, and S are oriented with respect to each other in the same way as they are oriented in the previous time frame T-1. This orientation would be checked by the comparison of distances and angles. For example, if vertexes A_M, B_M and S_M belong to time frame M, and vertexes A_L, B_L and S_L belong to a subsequent previous time frame N, marking the previous positions of vertexes A_M, B_M and S_M correspondingly, the algorithm would check that;
$|A\_M-A\_L| < epsilon1$, $|B\_M-B\_L| < epsilon1$, $|S\_M-S\_L| < epsilon1$, and that the angle formed by A_M, B_M, and S_M, minus the angle formed by A_L, B_L, and S_L, is less than epsilon2.

Once having found such a pair, that forms a plane with the stating vertex S and has similar orientation, the algorithm would use the pair plus vertex S to begin the creation of a cluster. A cluster is defined to be a group of vertex that form rigid mesh surfaces between two subsequent time frames.

The algorithm would expand this cluster by looking through each of the vertexes at the boundary of that cluster. A boundary of the cluster is defined to be the set of vertexes that have been last added to the cluster and have been visited by following edges leading from a previous boundary vertex. Thus a cluster just created has boundary vertexes making up that entire cluster.

For each boundary vertex of the cluster, the algorithm would iterate through all the neighbouring vertex that do not yet belong to the cluster. It would then check if a neighbouring vertex is oriented with respect to the cluster similarly in time frame T and time frame T-1. This orientation would be checked by first selecting two neighbouring vertexes of the boundary vertex that already belong to the cluster and that form a plane with that boundary vertex. Next the new vertex being considered would be check if its oriented to these three vertexes similarly in angle and distance in both the current and previous time frame. If satisfied, it would be added to the cluster and become the new boundary vertex.

Once the cluster is not longer expanded, is creation is executed. The algorithm would then check if the cluster has been expanded at all. If not, it will disregard this cluster and move onto the next satisfying pair of vertex neighbours for the starting vertex S. Otherwise it would label all the vertexes in the cluster as having been marked and it would store that cluster for rendering.

This algorithm worked quite well, however it was quite slow and hence not used. The choices of epsilon1 and epsilon2 was chosen from experimentation and statistical gathering of data involving the comparison of meshes between two time frames. Figure 22 shows the result of the algorithm:

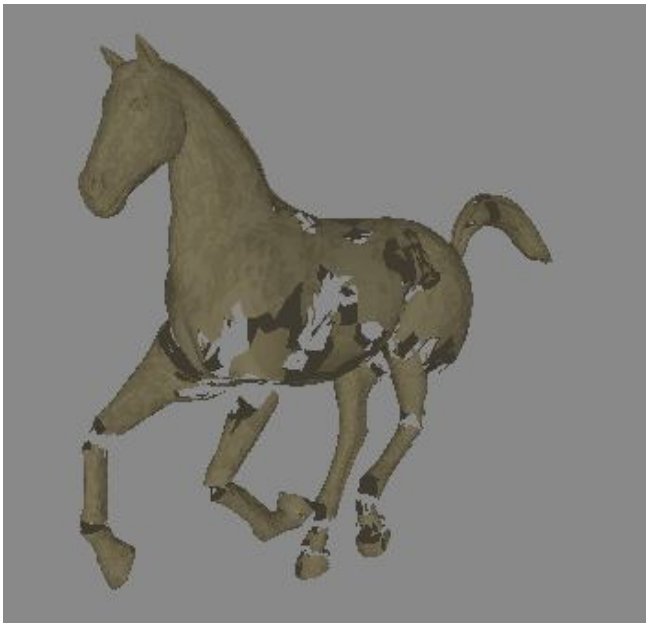


Figure 22: Example of a time frame with the rigid mesh surfaces rendered.