J Inf Process Syst, Vol.13, No.4, pp.818~833, August 2017
https://doi.org/10.3745/JIPS.02.0066

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Real-Time Motion Blur Using Triangular Motion Paths

MinhPhuoc Hong* and Kyoungsu Oh*

Abstract
This paper introduces a new algorithm that renders motion blur using triangular motion paths. A triangle occupies a set of pixels when moving from a position in the start of a frame to another position in the end of a frame. This is a motion path of a moving triangle. For a given pixel, we use a motion path of each moving triangle to find a range of time that this moving triangle is visible to the camera. Then, we sort visible time ranges in the depth-time dimensions and use bitwise operations to solve the occlusion problem. Thereafter, we compute an average color of each moving triangle based on its visible time range. Finally, we accumulate an average color of each moving triangle in the front-to-back order to produce the final pixel color. Thus, our algorithm performs shading after the visibility test and renders motion blur in real time.

Keywords
Motion Blur, Real-Time Rendering, Visibility

1. Introduction

Motion blur effect gives us a sense of speed and movement in images. This effect is generated when we capture an image and objects move relative to the camera during the exposure time. The motion blur effect is widely used in off-line rendering but it is a difficult problem in real-time rendering because it requires the visibility problem to be solved in the space-time dimensions [1]. During one frame rendering, there are many animating objects in a scene. Time at the start and the end of a frame is t=0 and t=1, respectively. At a certain time $t \in [0, 1]$, we need to determine which geometry is visible for a given pixel. And solving the visibility problem in the space-time dimensions is performing such visibility determination at many different times and averaging results.

A moving triangle only occupies a pixel in a range of time during one frame rendering. Many motion blur rendering algorithms have been proposed to approximate or analytically find such a range of time. But those algorithms produce ghosting artifacts or noisy images at low sampling rates. Increasing the sampling rate impacts performance significantly.

In this paper, we propose a new algorithm that renders motion blur using triangular motion paths. First, we triangulate a motion path of each triangle then use the ability of GPU to find a visible time range of this triangle. Thereafter, we sort visible time ranges in the depth and time dimensions and then use bitwise operations to resolve the occlusion problem. Finally, we blend all moving triangles in the

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.
Manuscript received January 6, 2017; first revision February 28, 2017; accepted April 7, 2017.
Corresponding Author: MinhPhuoc Hong (hmphuoc1985@gmail.com)
* Dept. of Media, Soongsil University, Seoul, Korea (hmphuoc1985@gmail.com, oks@ssu.ac.kr)

Copyright© 2017 KIPS

front-to-back order based on their visible time range. Thus, our algorithm can render noise-free and blurred images at interactive frame rates.

Our contributions are:

- Using triangular motion paths and the power of GPU to find a range of time that a triangle visible to a pixel during one frame rendering.
- Supporting the mipmapping technique to compute an average color of a moving triangle in its visible time range.
- Using a sorting algorithm in the depth-time dimensions and bitwise operations to solve the occlusion problem between moving triangles.

This paper is organized as follows. We discuss related works in Section 2. Section 3 describes our algorithm and Section 4 describes the implementation. Finally, we present results and discuss some issues in Section 5.

2. Related Works

Post-processing methods: Rosado [2] and Sousa [3,4] describe algorithms that render motion blur in real time and these methods are widely used in interactive games. However, these algorithms render geometries at a certain time and then use a filter to produce blurred images. Therefore, the visibility problem in the space-time dimensions is improperly handled. Later, McGuire et al. [5] use the scatter-as-gather based approach to generate blurred images. Scatter operations are operations that scatter a pixel's color to its neighbor, and gather operations are operations that gather color samples from neighboring pixels. A scatter-as-gather based approach is a post-processing method that converts scatter operations into gather operations in a post-processing pass. A scene has many animating geometries so each geometry might cover many pixels. Therefore, instead of scattering a pixel's color to its neighbor, a scatter-as-gather based approach gathers samples from neighboring pixels based on the current pixel's velocity. McGuire et al. [5] generate two intermediate buffers from a velocity buffer. For each pixel, the first intermediate buffer (TileMax) stores a velocity vector that has the largest magnitude in every $k \times k$ tile; and the second intermediate buffer (NeighborMax) stores a velocity vector that has the largest magnitude from the neighboring pixels. For each pixel, this method extracts a maximum velocity vector from the NeighborMax buffer and then samples along this vector. Consequently, result images have artifacts as objects move at different speeds and directions. Guertin et al. [6] improve an algorithm described by McGuire et al. [5] and produce high-quality blurred images by sampling along a maximum velocity vector and a pixel's velocity vector. Yet this algorithm does not completely solve the problem. We refer readers to McGuire et al. [5] for some older post-processing algorithms which use extruded geometry to render motion blur but those algorithms have the same visibility problem of post-processing approach.

Most algorithms assume an object moves linearly so it might produce artifacts as rendering fast motion. A scattering based rendering approach described by Guertin and Nowrouzezahrai [7] renders motion blur effect for non-linear motion. They represent a curve by a list of lines with each line has a constant color. Furthermore, they store a color and a weight at each pixel then use additive alpha blending to store all scatter operations. Finally, a color of each pixel is computed based on the contribution of the other pixels in the post-process pass. This approach has a high performance and partially address the visibility.

Brute-force methods: Haeberli and Akeley [8] and Korein and Badler [9] render geometries many times and average result images at each pixel using accumulation buffering. These algorithms have discrete ghosting artifacts due to under-sampling, but increasing the number of samples per pixel significantly impacts performance. Therefore, they are not suitable for real-time rendering.

Stochastic sampling methods: Cook et al. [10] distribute rays in the space-time dimensions and then averages all visible rays for the pixel color. Akenine-Moller et al. [11] have a similar idea but they use time-continuous triangles and require a new hardware for implementation. McGuire et al. [12] implement an idea described by Akenine-Moller et al. [11] on GPU using multi-sample antialiasing and 2D screen-space convex hulls or bounding boxes as space-time bounding volumes to render motion blur. Each sample has random time information which is used to locate the position of a moving triangle and then the visibility of this sample is determined through a ray-triangle intersection test. All visible samples are shaded and averaged to make the final pixel color. Thus, this method reduces the geometry rendering cost but suffers from noise at low sampling rates. Authors use multi-sampling to reduce to perform shading once per pixel, this improves performance but introduces an error as samples in the same pixel belong to different triangles. Furthermore, large bounding shape might result in a large number of intersection tests. A method described by Munkberg et al. [13] significantly reduces the amount of this test using hierarchical tile test with temporal overlap. However, this technique is implemented in a software rasterization. To reduce the number of ray-triangle tests, Laine et al. [14] construct a screen-space bounding for (u, v, t) and then determine t and (u, v) intervals for each tile. Subsequently, they test all samples in the current tile within those intervals. However, the computation cost for each tile impacts performance substantially.

Analytical methods: Our research is inspired by the analytical visibility approaches described by Korein and Badler [9], Newell et al. [15], and Grant [16]. Korein and Badler [9] compute a covering interval of each geometry for a given pixel. Then, they handle the interval occlusion and resolve these intervals for the final pixel color. We similarly find a time interval of each moving triangle by exploiting the standard rasterization then removing occluded intervals using a sorting in the depth-time dimensions and bitwise operations. Our sorting algorithm is analogous to a method described by Newell et al. [15] and polyhedron clipper described by Grant [16].

Gribel et al. [17] describe a rasterizer, based on time-dependent edge equations, to analytically solve the visibility in the space-time dimensions for motion blur rendering. This algorithm uses a software rasterizer so it does not utilize the power of GPUs.

Other methods: We refer readers to other algorithms described by Ragan-Kelley et al. [18] and Clarberg and his colleagues [19,20] for decoupling sampling from the visibility and algorithms described by Andersson et al. [21], Clarberg and Munkberg [22] for reusing shading or Johannes et al. [23] for a different effect since these methods are beyond the scope of this paper.

3. Algorithm

3.1 Background

When a triangle moves from a position in the start of a frame to another position in the end of a frame, it creates a motion path. The brute force method renders many times along the motion path of a

moving triangle then averages all rendered images to produce a blurred image, Fig. 1. In other words, this method samples a triangle uniformly in time dimensions. For instance, the triangle is sampled every 0.01 second. As a result, this method has ghosting artifacts at low sampling rates.

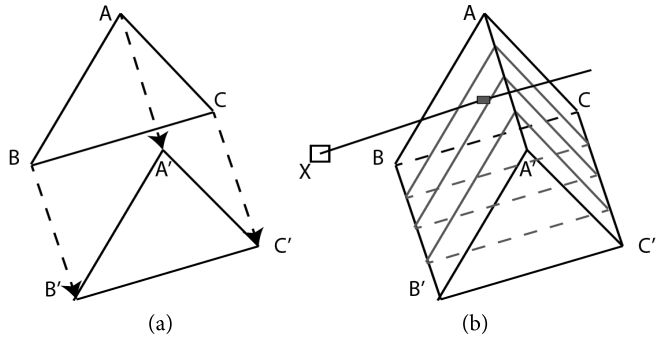


Fig. 1. (a) A triangle with its motion path from *ABC* to *A'B'C'*. (b) The brute force method generates a blurred image by rendering many times. At a given pixel *X*, all rendered images are averaged to produce the final color.

Contrary to the brute force method, stochastic sampling based methods render the moving triangle once using two positions at the start and the end of its motion path. These methods assume that two positions at the start and the end of a motion path are defined at time t=0 and t=1, respectively. From two triangles at t=0 and t=1, these methods build a space-time bounding volume in Fig. 2. McGuire et al. [12] use such a bounding volume as a convex hull. To do the visibility test and render motion blur, these methods use multi-sampling with each sample has a random time $t \in [0, 1]$. At a random time, t, vertices are linearly interpolated along edges. For example, A_t, B_t and C_t are linearly interpolated along *AA'*, *BB'* and *CC'* in Fig. 2, respectively. After that, the visibility determination is done through ray-triangle intersection tests. A ray is shot from the camera through the current pixel. If there is an intersection between the ray and $A_tB_tC_t$, the current sample is visible and its color is computed. Finally, all visible samples are averaged to produce the final pixel color. By this way, these methods randomly sample a triangle in the space-time dimensions. Therefore, a blurred image has noise at low sampling rates.

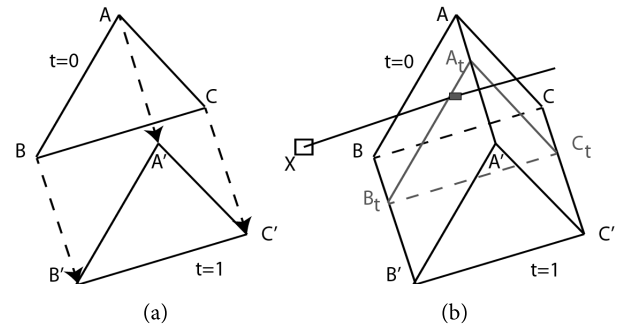


Fig. 2. (a) A triangle moves linearly from the start (t=0) to the end of frame (t=1). (b) A space-time bounding volume of the triangle is made from *ABC* and *A'B'C'*. The red highlighted triangle is the triangle's position at a random time *t*. A ray-triangle intersection test is performed at a given pixel *X*.

Since above methods do not know a visible time range of the moving triangle at the pixel *X*, so the brute force method and stochastics sampling methods use uniform sampling and randomly sampling in the time dimension, respectively. In our work, we use the standard rasterization of GPU to generate such a visible time range at the pixel *X*. Hence our algorithm avoids rendering many times and does not perform the visibility tests manually. The main idea of our algorithm will be described in the next section.

3.2 Main Idea

The same triangle in Figs. 1 and 2 is used to illustrate the main idea of our algorithm in Fig. 3. We also assume that a triangle linearly moves from the start to the end of a frame. At the start and the end of a motion path, the triangle is *ABC* and *A'B'C'*, respectively. We use two triangles *ABC* and *A'B'C'* to make a motion path. Each vertex has a time (*t*) and texture coordinates (*uv*). We triangulate the motion path and send to the next stage in GPU. For each pixel GPU, there are two points generated; each point has an interpolated time (*t*) and interpolated texture coordinates (*uv*). For instance, such two points are P_1 and P_2 in Fig. 3. Consequently, for a given pixel we know a range of time that this triangle is visible at the pixel *X*. We then compute opacity and color of this triangle for the pixel *X*. Opacity indicates a visible time range of a moving triangle at a pixel, and at the pixel *X* the moving triangle is only visible from t=0 to t=α so opacity is α. If we call texture coordinates of P_1 and P_2 are uv_1 and uv_2, respectively, the color of this moving triangle can be approximated by averaging texture colors along a line segment between uv_1 and uv_2. In this way, our method solves the visibility problem analytically and render motion blur.

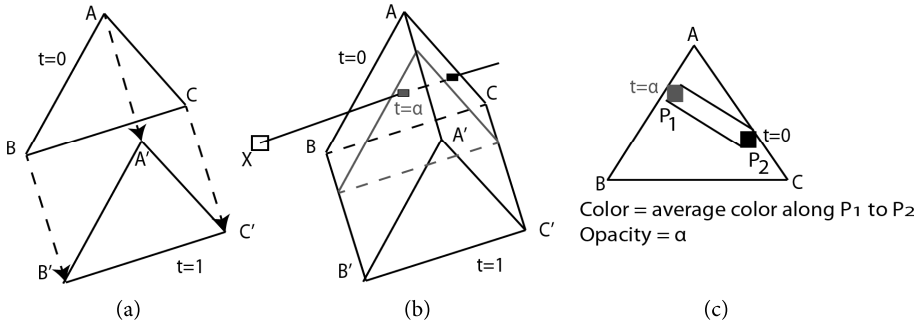


Fig. 3. (a) A triangle moves linearly from t=0 to t=1. (b, c) A color and opacity of an extruded triangle for a given pixel *X*. P_1 and P_2 are two visible fragments at the pixel *X*. P_1 has (t=α, uv_1) and P_2 has (t=0, uv_2).

When there are many moving triangles in a general case, we make many triangular motion paths and solve the occlusion problems using a sorting algorithm in the depth-time dimensions, in Section 3.3. Finally, average color of each motion path is accumulated, in Section 3.4.

3.3 A Sorting Algorithm in the Depth-Time Dimensions

In Section 3.2, we described our main idea to render motion blur using a triangular motion path when there is only one a moving triangle. Yet in a general case, there are many triangles covering the same pixel during one frame rendering. Therefore, we need to make and sort triangular motion paths in

the front-to-back order prior to the blending. To that end, we use an interval to represent a triangular motion path and then apply a sorting algorithm which is modified from Grant [16] on intervals. Finally, we use bitwise operations to solve the occlusion problem.

Interval generation: From now on, we use an interval to denote a visible time range of a moving triangle for a given pixel during one frame rendering. One interval is made from two fragments which are generated from a triangular motion path by GPU. For each interval, we store a minimum depth, a maximum depth and time information as shown in Fig 4.

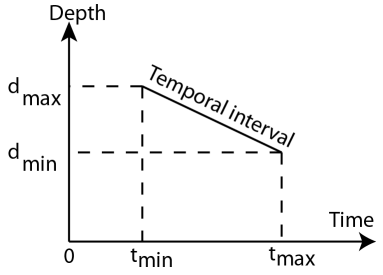


Fig. 4. An interval in the depth-time dimensions.

Note that if two triangles at t=0 (ABC) and at t=1 ($A'B'C'$) twist, there are four generated fragments at a pixel in Fig. 5. In such cases, we can make two intervals from four fragments of a triangular motion path. We first sort these four fragments by the distance from the camera and then use every two fragments to make an interval.

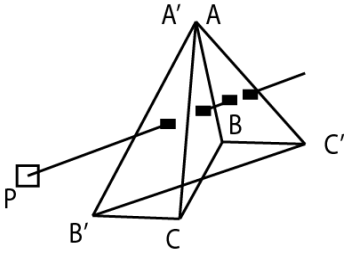
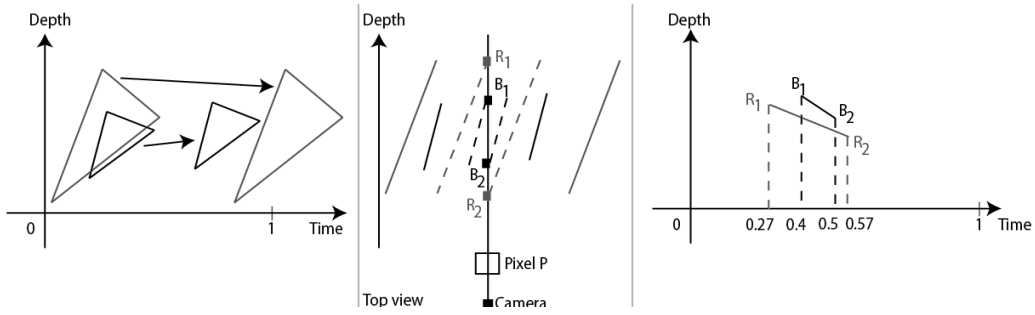


Fig. 5. A triangle rotates from ABC at t=0 to $A'B'C'$ at t=1. For a given pixel P, instead of two fragments, GPU generates four fragments.

Sorting: We compute a line equation for each interval using its depth and time. Then, we represent an interval as a 2D line segment in the depth-time dimensions. And we sort these intervals in the front-to-back order using a modified depth sorting algorithm which can be briefly summarized as follows: first, we sort all intervals by the minimum depth in ascending order. If two intervals intersect, we find and use the intersection point to divide these two intervals into four shorter intervals. Thereafter, we check the following conditions with each pair of intervals (H, K).

1. Check whether there is no depth overlap between H and K.
2. Check whether there is no time overlap between H and K.
3. Check whether K is entirely behind H from the camera. (Use H's line equation).
4. Check whether H is entirely front of K from the camera. (Use K's line equation).

If any condition is true, we keep the current order and advance to the next pair. Otherwise, we swap these two intervals.



(a) (b) (c)

Fig. 6. (a) Two triangles move from t=0 to t=1. (b) The top view of two moving triangles. The two red dashed lines intersect with a ray shot from the camera at two certain times and two intersection points are R_1 and R_2. It is the same with the two black dashed lines. (c) Intersection points of the red and the black triangle with the ray are denoted by two line segments in the depth and time dimensions.

Fig. 6 shows the basic idea of our sorting algorithm with two moving triangles at the start (t=0) and at the end (t=1) of a motion path. From the top view, we can see that the red triangle intersects with a ray shot from the camera through a pixel P at R_1 and R_2. It means that the red triangle is visible at the pixel P at two certain times which are 0.27 and 0.57. Similarly, the black triangle is visible at the pixel P at 0.4 and 0.5. First, we use depth and time information of R_1 and R_2, B_1 and B_2 to make two line segments (intervals). Then, we apply the sorting on these two intervals in the depth and time dimensions. As can be seen in the rightmost of Fig. 6, two intervals overlap in both depth and time dimensions but the interval B_1B_2 is completely behind the interval R_1R_2 with respect to the distance from the camera. Therefore, the interval R_1R_2 is closer to the camera than the interval B_1B_2.

3.3 Bitwise Operations in Blending

After sorting, we traverse the sorted interval list in the front-to-back order to compute and remove occluded regions in each interval. Then, we do the blending based on each interval. An interval is a time range that a moving triangle is visible at the current pixel after resolving the occlusion problem. To calculate a visible time range of each moving triangle, we convert time information of an interval to an unsigned integer and then use bitwise operations to solve the occlusion problem. In an array of bits, '0' bit means invisible and '1' means visible; i^{th} bit is the visibility in $[i/n, (i+1)/n]$ where n is the total number of bits. In Fig. 7, the "accum_mask" is a mask used for accumulating visible time ranges of previous moving triangles and the "curr_mask" is a visible time range of the current moving triangle before and after solving the occlusion problem, respectively. These two masks overlap at some bits which are highlighted in the "curr_mask". It means that a time range of the current moving triangle is occluded at those bits. We call such highlighted bits as occluded bits. Therefore, to solve the occlusion problem we just change all '1' bits to '0' bits at occluded bits using NOT and AND operators. We then compute and add the current moving triangle's average color to the pixel color based on its visible time range. Thereafter, we use OR operator to accumulate the overall visible time range. During blending if

all bit values of "accum_mask" are '1' we can stop the traversal. After blending all intervals if the 'accum_mask' is not 1, we add the background color to the pixel color based on the number of '0' bits in the "accum_mask".

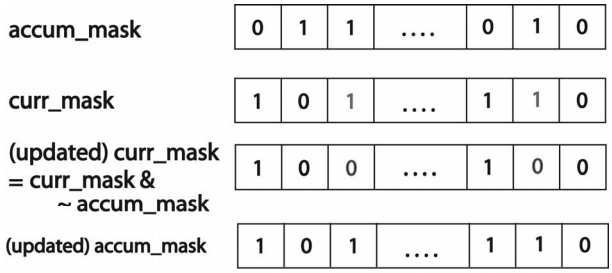


Fig. 7. The occlusion problem between intervals is solved using bitwise operations.

4. Implementation

4.1 System Overview

Fig. 8 shows the flowchart of our algorithm. In the first pass, all vertices are transformed from the object space into the view space in the vertex shader. Then, the geometry shader uses six vertices at the start and the end of a frame to make a triangular motion path, in Section 4.2. In the pixel shader, we store all fragments into a buffer as a per-pixel linked list using the current fragment's information, in Section 4.3.

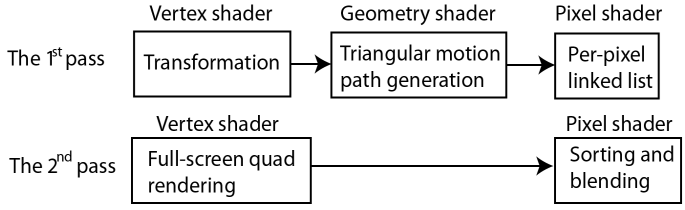


Fig. 8. A flowchart of our algorithm.

In the second pass, we do the same thing as another OIT algorithm [24] to load all fragments belonging to the current pixel then sort (in Section 3.3) and blend all intervals (in Section 3.4). Since the cost of swapping intervals is more expensive than the cost of comparison as sorting, we use the selection sort instead of insertion sort.

To compute an average color of an interval, we linearly interpolate texture coordinates (uv) many times from uv_1 to uv_2 of the current interval and then do the texture sampling using a computed mipmap level. Subsequently, we average all texture colors to produce the average color of the current interval. For clarity of the presentation, we describe how to compute the mipmap level in Section 4.4.

When solving the occlusion problem between intervals, we observe that varying the number of bits in a coverage mask from 32 to 128 only increases the render time about 0.3 ms and there is no big difference in blurred images. Therefore, we use 128 bits for a coverage mask.

In the second pass, we fix the maximum number of intervals per pixel at 90. Rendering a high depth complexity scene requires a large number of intervals. If the required number of intervals is larger than the maximum number of intervals per pixel, the visibility is solved incorrectly. Because there is not enough space to store some visible intervals, thereby we cannot accumulate these intervals' color to the pixel color. But in practice, we observe that using 90 intervals per pixel produces plausible blurred images. Note that in low depth complexity scenes, we can reduce the maximum number of intervals per pixel to increase performance.

4.2 Triangular Motion Path Generation

This section describes how to make and output a triangular motion path in the geometry shader. The input of this step is six vertices in the view space and the output is a triangular motion path in the clip space. Fig. 9 illustrates this step. Each moving edge generates a bilinear surface, for example, an edge moving from AB (t=0) to $A'B'$ (t=1) generates a bilinear surface $ABB'A'$. For each bilinear surface, we average its four vertices to make a center vertex and then assign t=0.5 for the center vertex. For instance, H, L and K are center vertices of bilinear surfaces $ABB'A'$, $BB'CC'$ and $AA'C'C$. As computing center vertices, we average not only vertices' positions but also texture coordinates and normal vectors. Thereafter, we use a center vertex of each bilinear surface to subdivide it to four triangles. Finally, we emit 12 triangles from three triangulated surfaces and two input triangles.

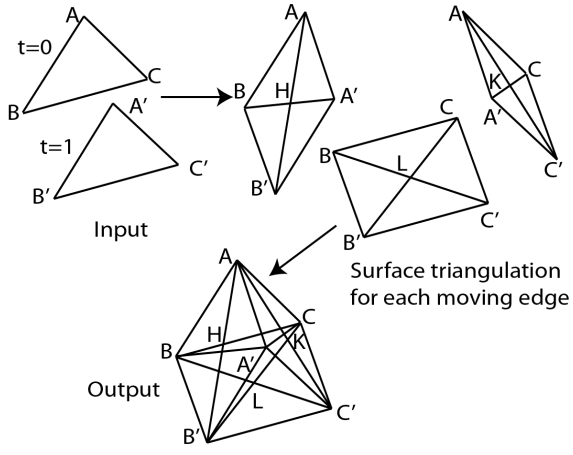


Fig. 9. Triangular motion path generation in the geometry shader. Three edges moving from t=0 to t=1 generate three bilinear surfaces. We compute a center vertex for each bilinear surface then use the center vertex to triangulate the bilinear surface. Vertices A, B, C have t=0 and vertices $A'B'C'$ have t=1. All center vertices (H, K, L) have t=0.5.

Each moving edge is represented by a pair of edges, i.e., one at t=0 and the other at t=1. For example, the first moving edge is represented by a pair of edges (AB, $A'B'$). When there is an intersection between two edges of a pair in the screen space, a moving edge does not generate a bilinear surface anymore. In such cases, we find an intersection point then use it to emit two triangles for the side surface. We briefly describe how to find an intersection point between two edges of a pair in the screen space and we refer readers to a method described by Akenine-Moller et al. [25] for details.

Consider two segments (edges) $AB = A + \overrightarrow{AB} * s$ and $CD = C + \overrightarrow{CD} * u$. We use a perp dot product (PDP) of AB and CD to check if AB intersects CD. A PDP of two vectors is a dot product where the first vector is replaced by a perpendicular vector rotated 90° to the left. A value returned from PDP(AB, CD) is an area of the parallelogram spanned by AB and CD, in Fig 10. So if PDP(AB, CD) equals to 0, there is no intersection.

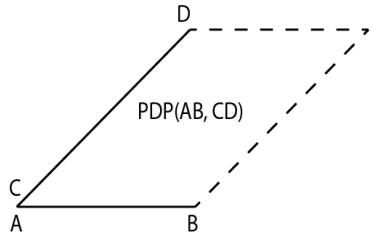


Fig. 10. Perp dot product (PDP) of AB and CD.

Otherwise, we compute two variables (s and u) as follows:

$$s = \frac{PDP(CD,AC)}{PDP(AB,CD)},\ u = \frac{PDP(AB,AC)}{PDP(AB,CD)} \tag{1}$$

There is an intersection and it only insides AB and CD if s, $u \in [0, 1]$.

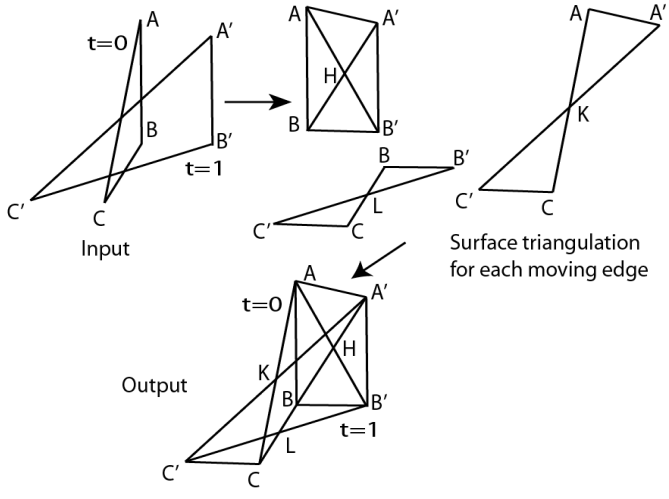


Fig. 11. A special case of generating a triangular motion path in the geometry shader. Left: Six input vertices (A, B, C at t=0) and (A', B', C' at t=1). Right: Three side surfaces are triangulated based on the intersection check between two edges (AB, $A'B'$), (AC, $A'C'$), (BC, $B'C'$).

If there is an intersection between two edges of a pair, we compute all intersections' information using the perspective interpolation along corresponding edges. Finally, we emit all triangles. In this case, 10 triangles are emitted in Fig. 11.

4.3 Per-Pixel Linked List

In this section, we briefly describe how to store and load all fragments as a per-pixel linked list. It is analogous to Barta et al. [24], Burns and Hunt [26], and Salvi et al. [27]. In our work, a fragment stores a depth, texture coordinates, a time, a normal vector, a texture id and a next pointer. To achieve this, we use two buffers in total: a fragment buffer and a start-index buffer.

The fragment buffer stores rasterized fragments and each fragment has an index pointer which is used to access the next fragment in the fragment buffer. If an index pointer is '-1', this is an end of a list.

The start-index buffer has the same resolution with the render target and stores start-index values. Thus for each pixel, it maps exactly to one element in the start-index buffer.

Fig. 12 shows an example of this step. At the beginning, a counter is initialized to '0', all values in the start-index buffer and the "Next" pointer in the fragment buffer is initialized to '-1'. As rendering the red triangle, we map to a position in the start-index buffer then store the counter's value and get the original index value. Considering the first fragment of the red triangle, we store '0' at (1, 1) in the start-index buffer then get '-1'. And we store this fragment to the fragment buffer at '0' index then assign the "Next" pointer to '-1'. Finally, we increase the counter's value one unit using an atomic operation. In this way, we store all four fragments of the red triangle, Fig. 12. As rendering the blue triangle, we store all fragments of the blue triangle in the same way. However, we get '2' and '3' when storing '5', '6' at positions (2, 2), (2, 3) in the start-index buffer, respectively. The '2' and '3' values are used to assign to the "Next" pointer in Fig 12. Thus, we can link and store fragments in a per-pixel linked list.

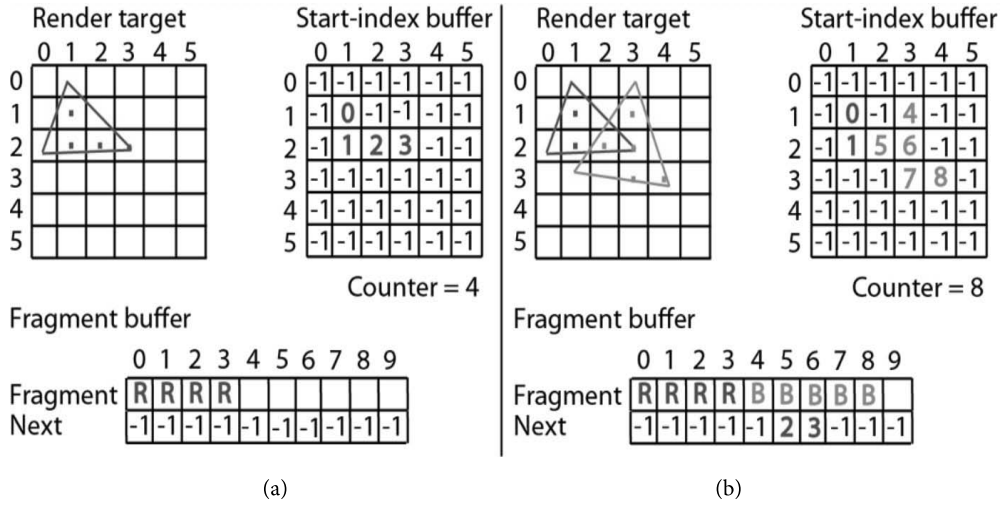


Fig. 12. Demonstration for storing all fragments in a per-pixel linked list. (a) Rendering and storing fragments of the first (red) triangle. (b) Rendering and storing fragments of the second (blue) triangle. "R" and "B" are generated fragments of the red and blue triangles, respectively. The start-index buffer stores start-index values which are used to extract fragment in the fragment buffer. The "Next" pointer stores an index of the next fragment, '-1' value represents the end of a list.

To load all fragments at a given pixel, we extract a start-index value from the start-index buffer and then use the start-index value to get a fragment in the fragment buffer. Thereafter, we use a next pointer

of the current fragment to get the next fragment in the fragment buffer and continuously get all fragments until the end of a list (pointer value is –1).

4.4 Mipmapping

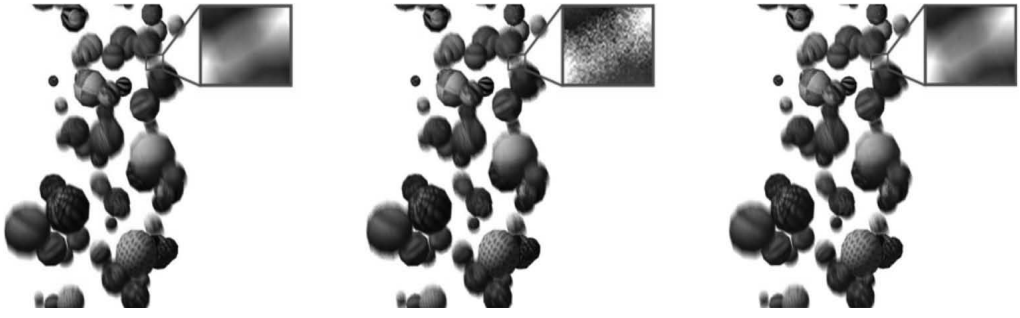
In this section, we describe how to apply the mipmapping technique as computing an average color of each interval. Generally, GPUs choose the mipmap level by computing finite differences across four pixels. In our method, we store a list of intervals at each pixel so such differences may compare different intervals. Therefore, the mipmap level may be chosen incorrectly.

To choose the mipmap level we do as follows. First, we compute the longest distance in *uv* axes in the texture space then divide this distance by the number of texture samples. Next, we use the result to calculate the mipmap level using the logarithm. Thus, our blurred images may have the over-blurred problem as using a few texture samples. In practice, we observe that using 16 texture samples is sufficient to address the over-blurred problem. Computing the mipmap level in this way does not measure exactly finite differences in adjacent pixels but it guarantees that the same interval is used for computing the mipmap level. This may result in error but increase the blurred image quality.

5. Discussion and Results

All result images are rendered at 1024×768 pixels using GTX 980 6GB with DirectX 11, HLSL 5.0, and Phong Shading. We implement the stochastic rasterization method (ST) using fast ray-triangle intersection test [28] and multi-sampling. We use 8 samples per pixel for performance comparisons and 12 samples per pixel for image quality comparisons.

Figs. 13 and 14 show the image quality comparisons between our method and the stochastic rasterization method at the similar render time. It is difficult to make a comparison at the similar time so we increase the render time in our method using 128 texture samples. While our result image has the similar quality with the reference image, an image rendered by the stochastic rasterization have noise.



Reference | The stochastic rasterization method | Our method

Fig. 13. Image quality comparison between our method and the stochastic rasterization method at the similar render time (8.6 ms). The reference image is rendered using the accumulation buffer with 3,000 samples. The stochastic rasterization method uses multi-sampling 12 samples per pixel. Our result image is rendered using 90 intervals, 128 texture samples.

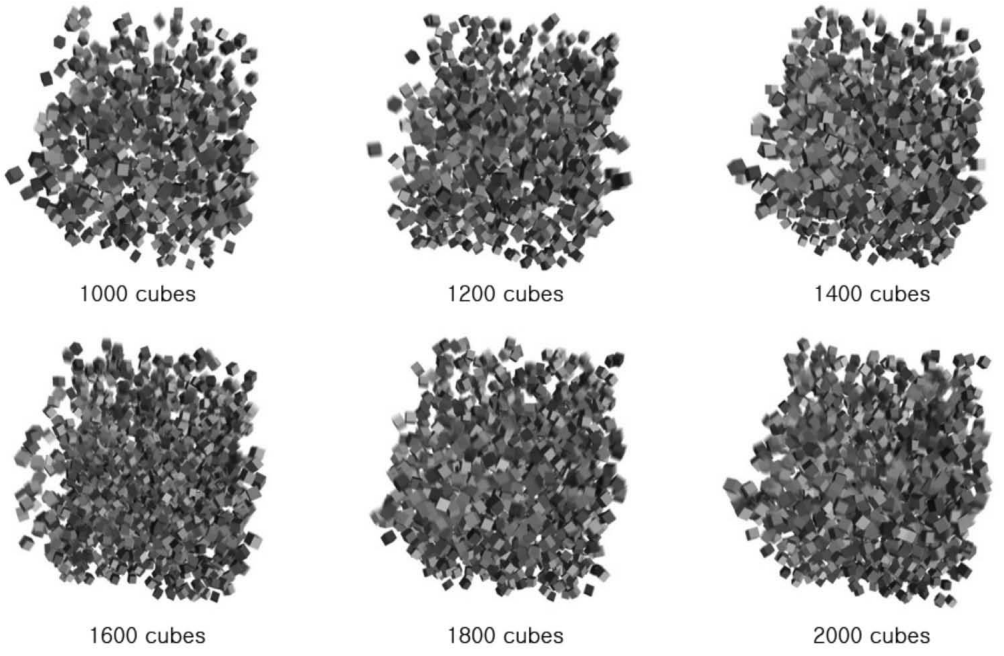


Fig. 14. Our main test scenes with the depth complexity increase from the left to the right and from the top to the bottom.

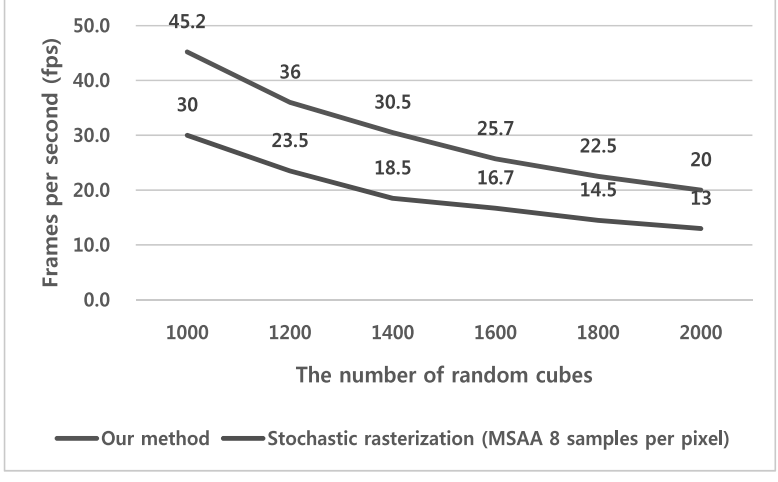


Fig. 15. Performance comparison between our method and the stochastic rasterization algorithm by varying the number of random cubes (Fig. 14) from 1,000 to 2,000. From the left to the right, the average number of intervals per pixel is 1.67, 2.02, 2.33, 2.73, 3.04 and 3.39, respectively.

Fig. 15 shows the performance comparison between our method and the stochastic rasterization method. In this comparison, performance in both methods is impacted by a large number of draw calls. In our method, the number of draw calls is the same as the number of random cubes, while ST requires more draw calls to generate blurred images. For instance, ST needs to double the number of draw calls to use multi-sampling eight samples per pixel because GPU only generates four samples per pixel for a

draw call. Thus, the performance in ST is much impacted than ours, especially when using a large number of samples per pixel. In theory, the sorting in our method takes $O(n^2)$ with n is the number of intervals. But in this comparison, since n is small so the sorting does not impact the overall performance significantly.

Fig. 16 shows the impact of the sorting on performance using the test scenes in Fig. 15. This measurement is done by toggling on/off the sorting in the second pass. As the depth complexity increases, the impact of the sorting also increases. So the sorting is the main bottleneck in our method.

Due to the memory limitation in GPUs, we cannot store all shading attributes and in the current implementation, we assume that all shading attributes are constant as performing shading.

Our method assumes that geometry moves linearly from the start to the end of a frame so we will have visual artifacts when rendering geometries with fast motion. In order to address this problem, we can split a motion into smaller parts then process each part using our method, and we leave this for the future work.

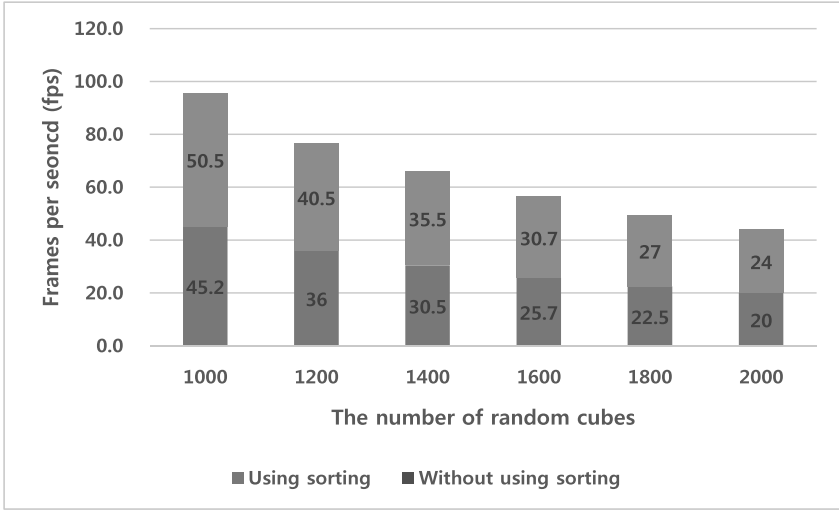


Fig. 16. The impact of the sorting on performance using the test scenes in Fig. 14. From the left to the right, the average number of intervals per pixel is 1.67, 2.02, 2.33, 2.73, 3.04 and 3.39, respectively.

6. Conclusion

We have presented a new method to render motion blur in real time using triangular motion paths. The basic idea is that a motion path of a moving triangle allows us to find a visible time range of this moving triangle for a given pixel. To produce blurred images, we triangulate a motion path then use the hardware rasterization to obtain such a visible time range. Subsequently, we use an interval to represent a visible time range. And, we solve the occlusion problem by using the sorting algorithm in the depth-time dimensions in the front-to-back order and applying the bitwise operations on sorted intervals. Finally, we compute and accumulate an average color of each interval based on its (updated) visible time range to produce the final pixel color. Additionally, we also support the mipmapping technique as computing an average color of each interval.

References

[1] K. Sung, A. Pearce, and C. Wang, "Spatial-temporal antialiasing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 2, pp. 144–153, 2002.

[2] G. Rosado, "Motion Blur as a post processing effect," in *GPU Gems 3*. Upper Saddle, NJ: Addison-Wesley, 2008, pp. 575–581.

[3] T. Sousa, "Crysis next gen effects," in *Game Developer Conference*, San Francisco, CA, 2008.

[4] T. Sousa, "CryENGINE 3 rendering techniques," in *Talk at Microsoft Game Technology Conference (GAMEfest)*, Birmingham, UK, 2011.

[5] M. McGuire, P. Hennessy, M. Bukowski, and B. Osman, "A reconstruction filter for plausible motion blur," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Costa Mesa, CA, 2012, pp. 135–142.

[6] J. P. Guertin, M. McGuire, and D. Nowrouzezahrai, "A fast and stable feature-aware motion blur filter," in *Proceedings of ACM SIGGRAPH/EuroGraphics High Performance Graphics*, Lyon, France, 2014, pp. 51–60.

[7] J. P. Guertin and D. Nowrouzezahrai, "High performance non-linear motion blur," in *Eurographics Symposium on Rendering - Experimental Ideas & Implementations*. Luxembourg: The Eurographics Association, 2015.

[8] P. Haeberli and K. Akeley, "The accumulation buffer: hardware support for high-quality rendering," *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4, pp. 309–318, 1990.

[9] J. Korein and N. Badler, "Temporal anti-aliasing in computer generated animation," *ACM SIGGRAPH Computer Graphics*, vol. 17, no. 3, pp. 377–388, 1983.

[10] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 137–145, 1984.

[11] T. Akenine-Moller, J. Munkberg, and J. Hasselgren, "Stochastic rasterization using time-continuous triangles," in *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, San Diego, CA, 2007, pp. 7–16.

[12] M. McGuire, E. Enderton, P. Shirley, and D. Luebke, "Real-time stochastic rasterization on conventional GPU architectures," in *Proceedings of the Conference on High Performance Graphics*, Saarbrucken, Germany, 2010, pp. 173–182.

[13] J. Munkberg, P. Clarberg, J. Hasselgren, R. Toth, M. Sugihara, and T. Akenine-Moller, "Hierarchical stochastic motion blur rasterization," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, Vancouver, Canada, 2011, pp. 107–118.

[14] S. Laine, T. Aila, T. Karras, and J. Lehtinen, "Clipless dual-space bounds for faster stochastic rasterization," *ACM Transactions on Graphics*, vol. 30, no. 4, article no. 106, 2011.

[15] M. E. Newell, R. G. Newell, and T. L. Sancha, "A solution to the hidden surface problem," in *Proceedings of the ACM Annual Conference*, Boston, MA, 1972, pp. 443–450.

[16] C. W. Grant, "Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space," *ACM SIGGRAPH Computer Graphics,* vol. 19, no. 3, pp. 79–84, 1985.

[17] C. J. Gribel, M. Doggett, and T. Akenine-Moller, "Analytical motion blur rasterization with compression," in *Proceedings of the Conference on High Performance Graphics*, Saarbrucken, Germany, 2010, pp. 163–172.

[18] J. Ragan-Kelley, J. Lehtinen, J. Chen, M. Doggett, and F. Durand, "Decoupled sampling for graphics pipelines," *ACM Transactions on Graphics*, vol. 30, no. 3, article no. 17, 2011.

[19] P. Clarberg, R. Toth, J. Hasselgren, J. Nilsson, and T. Akenine-Moller, "AMFS: adaptive multi-frequency shading for future graphics processors," *ACM Transactions on Graphics*, vol. 33, no. 4, article no. 141, 2014.

[20] P. Clarberg, R. Toth, and J. Munkberg, "A sort-based deferred shading architecture for decoupled sampling," *ACM Transactions on Graphics*, vol. 23, no. 4, article no. 141, 2013.

[21] M. Andersson, J. Hasselgren, R. Toth and T. Akenine-Moller, "Adaptive texture space shading for stochastic rendering," *Computer Graphics Forum*, vol. 33, no. 2, pp. 341–350, 2014.

[22] P. Clarberg and J. Munkberg, "Deep shading buffers on commodity GPUs," *ACM Transactions on Graphics*, vol. 33, no. 6, article no. 227, 2014.

[23] S. Johannes, W. S. Robert, B. Huw, and M. Gross, "Programmable motion effects," *ACM Transactions on Graphics*, vol. 29, no. 4, article no. 227, 2010.

[24] P. Barta, B. Kovacs, S. L. Szecsi, and L. Szirmay-kalos, "Order independent transparency with per-pixel linked lists," in *Proceedings of the 15th Central European Seminar on Computer Graphics (CESCG2011)*, Vinicn, Slovakia, 2011.

[25] T. Akenine-Moller, E. Haines, and N. Hoffman, "Line/line intersection tests," in *Real-Time Rendering*, 3rd ed. Boca Raton, FL: CRC Press, 2008, pp. 780–781.

[26] C. A. Burns and W. A. Hunt, "The visibility buffer: a cache-friendly approach to deferred shading," *Journal of Computer Graphics Techniques*, vol. 2, no. 2, pp. 55–69, 2013.

[27] M. Salvi, J. Montgomery, and A. Lefohn, "Adaptive transparency," in *Proceedings of the Conference on High Performance Graphics*, Vancouver, Canada, 2011, pp. 119–126.

[28] S. Laine and T. Karras, "Efficient triangle coverage tests for stochastic rasterization," NVIDIA Research, *Technical Report No. NVR-2011-0003*, 2011.

MinhPhuoc Hong

He received the B.S. degrees in Software Engineering from University of Science Ho Chi Minh in 2009. His research interest is real-time rendering, motion blur and global illumination.

Kyoungsu Oh

He received the B.S. and Ph.D. degrees in Seoul National University. His research interest is real-time rendering, shadow mapping, global illumination and motion blur.