J Inf Process Syst, Vol.14, No.4, pp.877~891, August 2018
https://doi.org/10.3745/JIPS.02.0091

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

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Motion-Blurred Shadows Utilizing a Depth-Time Ranges Shadow Map

MinhPhuoc Hong* and Kyoungsu Oh*

Abstract

In this paper, we propose a novel algorithm for rendering motion-blurred shadows utilizing a depth-time ranges shadow map. First, we render a scene from a light source to generate a shadow map. For each pixel in the shadow map, we store a list of depth-time ranges. Each range has two points defining a period where a particular geometry was visible to the light source and two distances from the light. Next, we render the scene from the camera to perform shadow tests. With the depths and times of each range, we can easily sample the shadow map at a particular receiver and time. Our algorithm runs entirely on GPUs and solves various problems encountered by previous approaches.

Keywords

Motion-Blurred Shadows, Real-Time Rendering, Space-Time Visibility

1. Introduction

Motion-blurred shadow effects improve the sense of realism in images. However, rendering motion-blurred shadows is difficult in real-time applications and there have been few studies in this area.

For static scenes, a traditional shadow map stores the nearest depth of each pixel and shadow tests are performed in subsequent passes to produce a final image. For animated geometries, a brute force method renders the geometries many times and averages the results to create a final image with motion-blurred shadows. The resulting images often contain banding artifacts, but increasing the number of rendering iterations affects performance significantly. To address this problem and improve performance, stochastic sampling methods utilize multi-sampling, where each sample has a random time. These methods eliminate banding artifacts, but encounter time mismatch problems, which result in self-shadow artifacts. Addressing this issue requires a large number of samples, which consumes a large amount of memory.

To solve these problems and render motion-blurred shadows, our algorithm generates a depth-time ranges shadow map from a light source. For each pixel, each animated geometry is visible for a certain number of frames during rendering. Therefore, our goal is to store all time ranges and their corresponding depth values for each pixel in the shadow map. We then utilize multi-sampling, where

Manuscript received April 3, 2017; first revision May 24, 2017; second revision August 7, 2017; accepted September 15, 2017
Corresponding Author: MinhPhuoc Hong (hmphuoc1985@gmail.com)
* Dept. of Media, Soongsil University, Seoul, Korea (hmphuoc1985@gmail.com, oks@ssu.ac.kr)

each sample has a random time, to render all geometries from the camera and perform shadow tests for visible samples.

The remainder of this paper is organized as follows. First, we discuss related works in Section 2 and describe our algorithm in Section 3. Section 4 discusses the implementation of our algorithm and Section 5 presents experimental results.

2. Related Works

There has been significant research on shadow mapping and motion-blurred rendering. We refer to a book by Eisemann et al. [1] and Woo and Poulin [2] for an overview of shadow mapping. For an overview of motion-blurred rendering, we refer readers to a survey presented by Navarro et al. [3].

An accumulation buffer [4] renders a scene many times by utilizing traditional shadow mapping methods and then averages this results to generate a final image. This approach can render motion-blurred shadows correctly, but requires a huge number of rendering iterations. Otherwise, this approach suffers from banding artifacts.

The stochastic-sampling-based method [5] introduces time-continuous triangles and time-dependent shadow mapping (TSM) to render motion blur and motion-blurred shadows. This method utilizes stratified sampling to divide the total exposure time into uniform time intervals and then randomly selects a time from each time interval. First, this method renders a scene utilizing stochastic rasterization from a light source to generate many shadow maps, where each shadow map represents a random time t_s. Next, the scene is rendered from the camera utilizing stochastic rasterization and a shadow test is performed at a random time t_r. By utilizing stratified sampling, this method ensures that t_s and t_r belong to the same time interval. Therefore, this method eliminates banding artifacts and improves performance. However, it suffers from noise issues and self-shadow artifacts.

McGuire et al. [6] implemented time-dependent shadow maps on current GPUs, but their method encounters the time mismatch problem. To address this issue, during the initial shadow pass, we store a list of visible depth-time ranges for each pixel in a shadow map. Each visible depth-time range represents a period where a particular geometry was visible in the current pixel with two depths corresponding to that range. Therefore, during the lighting pass, we can easily perform shadow tests without the time mismatch problem.

Palmer [8] extended a motion-blurred rendering algorithm [7] that analytically renders motion blur to render motion-blurred shadows. This algorithm finds a period where a particular geometry is visible to the light source and stores this information as a depth-time interval. By sorting and solving the occlusion relationships between all depth-time intervals, this algorithm renders motion blur and motion-blurred shadows analytically. However, this algorithm does not run on GPUs.

Deep shadow maps [9] store a visibility function for each pixel in the shadow map. When generating the shadow map, each sample is given a random time and, for a given depth, all samples are averaged into a visibility function. Because it does not maintain time information in the shadow map, this algorithm only renders motion-blurred shadows correctly when the receiver is stationary. Similarly, deep opacity maps [10] utilize a layered approach to render shadows from semi-transparent objects. However, this algorithm requires a large number of layers to avoid visual artifacts.

Stochastic transparency [11] renders the shadows of transparent objects by computing a proportional number of shadow map samples passing through the transparent layers. This approach was extended to

render colored shadow maps by McGuire and Enderton [12]. However, similar to the deep shadow maps [9], this extended method requires a stationary receiver.

Andersson et al. [13] rendered motion-blurred shadows utilizing a depth layer approach. This method utilizes stochastic rasterization to generate time-dependent shadow maps, where each sample contains motion vectors. The samples in each tile of the shadow maps are then clustered into depth layers utilizing a depth clustering method [14]. Next, for each layer, this approach calculates an average motion vector d and re-projects all the samples along this vector (to $t = 0.5$). The depth values of these samples are also computed at $t = 0.5$. For each pixel in the shadow map, this algorithm stores two depth moments in a variance shadow map [15], which is defined as a rotation-scaling transformation matrix that is computed utilizing a filter and the average motion vector d. To render motion-blurred shadows, a receiver sample is first re-projected along the corresponding layer's motion vector to $t = 0.5$. Next, variance shadow map visibility values are computed and utilized to approximate the visibility of receiver samples. Therefore, this algorithm renders motion-blurred shadows without self-shadow artifacts. However, this algorithm has some potential problems. First, when all samples in a tile move in different directions, the calculated average motion vector is incorrect. This problem can be alleviated by utilizing using a tile variance approach [16]. Although this algorithm can render images with plausible motion-blurred shadows utilizing a statistical method, it cannot accurately determine if a sample is occluded. Additionally, this algorithm may encounter the same problems as variance shadow maps.

3. Algorithm

3.1 Overview

Fig. 1 presents an overview of our algorithm. Similar to other shadow mapping methods, our algorithm utilizes two main passes: a shadow pass and lighting pass. In the shadow pass, we render all moving triangles at $t = 0$ and $t = 1$ from the light source to generate a depth-time ranges shadow map. In the lighting pass, we perform shadow tests for each sample utilizing this shadow map.

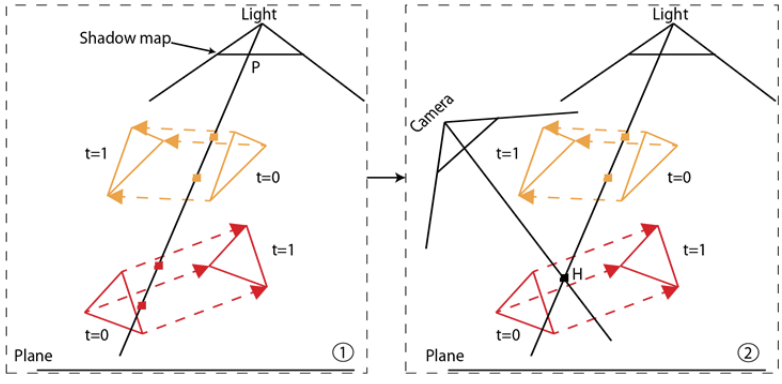


Fig. 1. An overview of our algorithm. Dashed lines indicate movement directions of each triangle. We render all moving triangles at $t = 0$ and $t = 1$ to generate a shadow map in the first pass. The two orange points and two red points indicate the periods during which the orange and the red triangles were visible to the light in pixel P. For each sample H that is visible to the camera, we utilize the shadow map to perform shadow tests in the second pass.

3.2 Shadow Pass

The purpose of this pass is to generate depth-time ranges for each pixel in a shadow map. Fig. 2 demonstrates how to utilize the ability of current GPUs to generate a depth-time range for a moving triangle. We assume a triangle moves linearly from the beginning frame ($t = 0$) to the ending frame ($t = 1$). The positions of this triangle at $t = 0$ and $t = 1$ are PQR and $P'Q'R'$, respectively. To generate a depth-time range for this triangle, we first utilize these two triangles to create a prism, as shown in Fig. 2. Next, we compute a center vertex for each quad by averaging the four corresponding vertices of each quad. We utilize each center vertex to triangulate the corresponding quad. The prism is completely triangulated and we assign time information to each vertex (i.e., PQR has $t = 0$ and $P'Q'R'$ has $t = 1$). When utilizing conventional rasterization, there are two generated fragments for each pixel (F_1, F_2), which are utilized to define a depth-time range. Each fragment has an interpolated time and a rasterized depth.

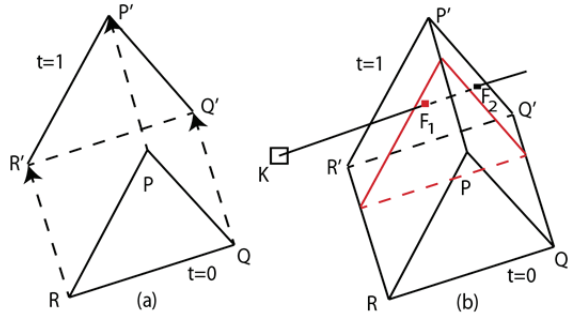


Fig. 2. (a) A triangle moves from a beginning frame ($t = 0$) to an ending frame ($t = 1$). At $t = 0$ and $t = 1$, this triangle is defined by PQR and $P'Q'R'$, respectively. (b) A prism is rasterized by utilizing conventional rasterization techniques and there are two generated fragments F_1 and F_2 for the pixel K, which are utilized to define a depth-time range. At pixel K, the triangle is visible in the range of $[t, 1]$.

3.3 Lighting Pass

In this pass, we utilize stochastic rasterization to render a scene from the camera. A triangle covers a set of pixels when moving from the beginning frame ($t = 0$) to the ending frame ($t = 1$). We utilize the two positions of this triangle at $t = 0$ and $t = 1$ to create a convex hull that covers all such pixels. There are multiple samples per pixel and each sample has a random time t. To determine a triangle's position at a given time t, we linearly interpolate the two positions of that triangle at $t = 0$ and $t = 1$, as shown in Fig. 3. To determine if the current sample is visible or not, we perform a ray-triangle intersection [17], where the ray is shot from the camera through the current sample. If there is an intersection between the ray and triangle, the current sample is visible.

To perform a shadow test for a visible sample, we first project the sample onto the shadow map and retrieve each visible depth-time range (t_s, t_e, d_s, d_e). Next, if the visible sample's time t_s is within the visible time range $[t_s, t_e]$, we calculate a depth value at t_s utilizing linear interpolation along the depth range $[d_s, d_e]$ and compare the interpolated depth to the sample's depth, as shown in Fig. 3. Finally, we perform shading and average all sample colors within a pixel based on the depth comparison results. To address the self-shadow artifacts present in TSM [5], we ensure that the current sample and triangle do not have the same triangle index prior to shadow testing.

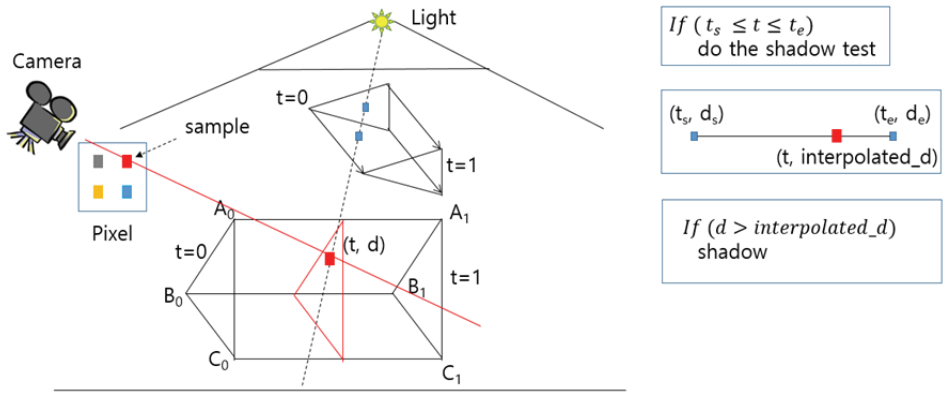


Fig. 3. Shadow test utilizing a depth-time range. The red sample is visible to the camera at time t. We retrieve a depth-time range from the shadow map to perform a shadow test for this sample.

4. Implementation

Our algorithm utilizes two main passes: a shadow map generation pass and lighting pass. The shadow generation pass consists of two sub-passes, which are shown in Fig. 4, and the lighting pass is a geometry pass that is implemented utilizing stochastic rasterization [6]. We refer readers to the original paper on stochastic rasterization [6] for the details of this process and to our website for pseudo-code (Appendix 1).

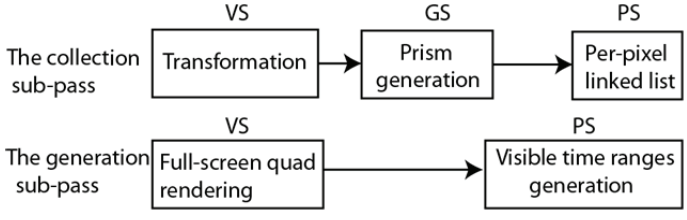


Fig. 4. Flowchart of the shadow map generation pass.

In the collection sub-pass, we render all geometries at $t = 0$ and $t = 1$ from the light source to store all visible fragments for each pixel in the shadow map. First, all vertices are transformed into the view space by the vertex shader. We then utilize six vertices in the view space to create a prism in the geometry shader (Section 4.1). In the pixel shader, we store all fragments for each pixel as a per-pixel linked list (Section 4.2).

The generation sub-pass is a full-screen sub-pass. For each pixel, we load a list of fragments to create a list of visible depth-time ranges (Section 4.3).

4.1 Prism Generation

The input for this step is a set of six vertices in the view space and the output is a prism in the clip space. Fig. 5 illustrates this step. Each moving edge generates a side surface. For example, an edge

moving from PQ ($t = 0$) to $P'Q'$ ($t = 1$) generates a side surface $PQQ'P'$. For each side surface, we average its four vertex locations to derive a center vertex and then assign $t = 0.5$ to the center vertex. For example, H is the center vertex of a bilinear surface $PQQ'P'$. We then divide each side surface into four triangles utilizing its center vertex. Finally, we emit 12 triangles from the three triangulated surfaces and two input triangles. Note that all triangles emitted from the same geometry shader have the same triangle index.

Each moving edge creates a pair of edges at $t = 0$ and $t = 1$. For example, the first moving edge creates a pair of edges PQ and $P'Q'$. When there is an intersection between two edges of a pair in the screen space, we find the intersection point and utilize it to emit two triangles for the side surface, as shown in Fig. 6. We will briefly describe how to find the intersection point between two edges of a pair in the screen space and refer readers to the method presented in [17] for details.

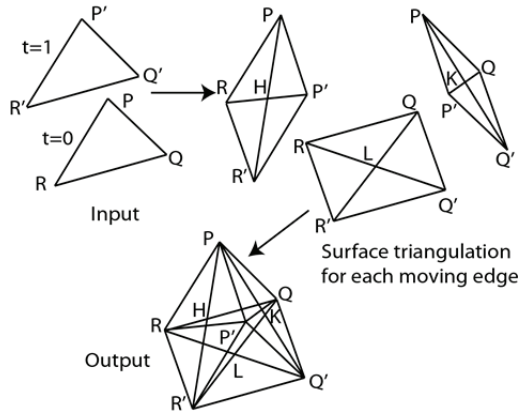


Fig. 5. Prism generation in the geometry shader. Three edges moving from $t = 0$ to $t = 1$ generate three side surfaces. The vertices PQR have $t = 0$ and the vertices $P'Q'R'$ have $t = 1$. All center vertices (HKL) have $t = 0.5$.

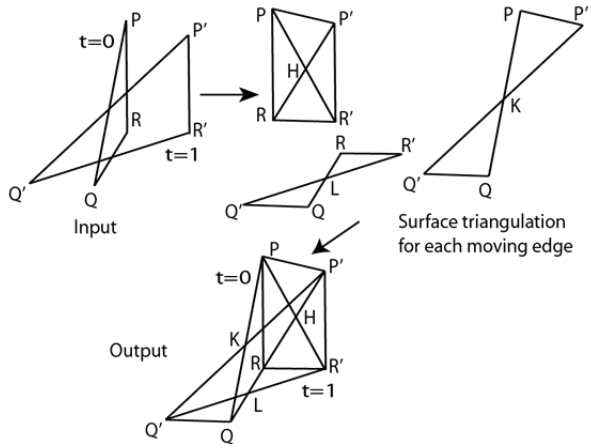


Fig. 6. A special case when generating a moving triangle in the geometry shader. (Left) Six input vertices (PQR at $t = 0$) and ($P'Q'R'$ at $t = 1$). (Right) Three side surfaces are triangulated based on an intersection check between two edges: (PQ, $P'Q'$), (PR, $P'R'$), and (QR, $Q'R'$). We then emit these triangles from the two input triangles.

Consider two edges *AB* and *CD*. We calculate the perp dot product (*PDP*) of *AB* and *CD* to determine 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° counterclockwise. The value returned by *PDP*(*AB*, *CD*) is the area of the parallelogram spanned by *AB* and *CD*, as shown in Fig 7. Therefore, if *PDP*(*AB*, *CD*) is equal to zero, there is no intersection.

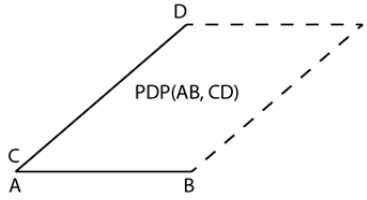


Fig. 7. *PDP* of two edges.

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

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

There is an intersection between *AB* and *CD* only if *s*, *u* ∈ [0, 1]. Finally, we emit all triangles created by intersections. For example, 10 triangles are emitted in Fig. 6.

4.2 Per-Pixel Linked List

In this section, we briefly describe how to store the fragments at each pixel as a per-pixel linked list. Our method is similar to those proposed by Salvi et al. [18] and Burns [19]. We refer readers to the studies by Barta et al. [20], Yang et al. [21], and Maulea et al. [22] for additional details. The main difference in our method is the structure of each fragment. In our method, each fragment stores a depth, time, triangle index, and pointer. To create this structure, we utilize two buffers: a fragment buffer and index buffer.

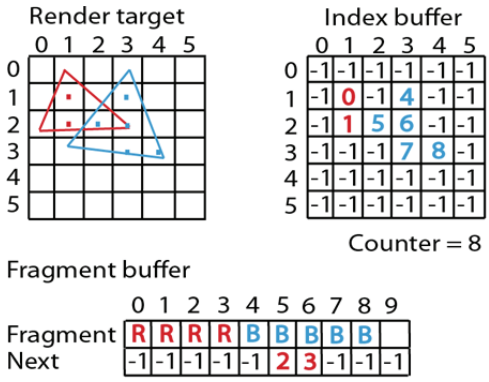


Fig. 8. The insertion orders are highlighted in red and blue. The index buffer stores a starting index for each pixel that points to the most recently inserted fragment in the fragment buffer. The most recently inserted fragment points to the previous fragment with a "next" pointer in the fragment buffer. "Counter" is a global value that is incremented when there is a new incoming fragment. "R" and "B" are fragments of the red and blue triangles, respectively. "-1" indicates the end of a list.

The index buffer has the same resolution as the render target. For each pixel, the buffer stores a starting index. This index points to a location in the fragment buffer, which stores all rasterized fragments. When there is a new incoming fragment, this index is updated to point to the new fragment, which receives the previous index. Therefore, each fragment points to the previous fragment and all fragments are stored in the form of a linked list. Fig. 8 presents an example of this step.

4.3 Generation Sub-pass

The purpose of this sub-pass is to improve model performance. Because we utilize multi-sampling in the lighting pass, for each visible sample in a pixel, we must load a list of fragments and create a list of visible depth-time ranges many times prior to the shadow tests. For each pixel, the list of visible time ranges does not change, so we utilize this sub-pass to load a list of fragments and create a list of visible depth-time ranges once for each pixel.

To that end, we render a full screen and then load a list of fragments for each pixel. We then iterate through the list of fragments to find two fragments that have the same triangle index, then utilize them to derive a visible depth-time range. If there are four fragments with the same triangle index, we sort these fragments by depth in ascending order and then generate two visible depth-time ranges. After deriving all visible depth-time ranges, we store them in a new per-pixel linked list. To accomplish this, we utilize two more buffers: one for the new index buffer and another for the range buffer.

5. Experimental Results

Our algorithm was implemented utilizing DirectX 11 and the High-Level Shader Language version 5.0 on a GTX 980 Ti 6 GB GPU. For comparison, we implemented the brute force method [3] utilizing 3,000 samples to generate reference images and TSM utilizing stochastic rasterization [6] with a fast ray-triangle intersection test [23]. All result images were rendered at a resolution of 1024×768 pixels.

Image Quality: Figs. 9 and 10 present image quality comparisons between the results of our algorithm and TSM utilizing the same number of samples per pixel. To highlight the self-shadow artifacts in the TSM results, we only utilized eight samples per pixel in Fig. 9 and 20 samples per pixel in Fig. 10. Therefore, both result images contain noise, but the images rendered by TSM contain self-shadow artifacts, while the images rendered by our method do not. The cause for these artifacts in the TSM results is the time mismatch issue, which results in incorrect shadow tests.

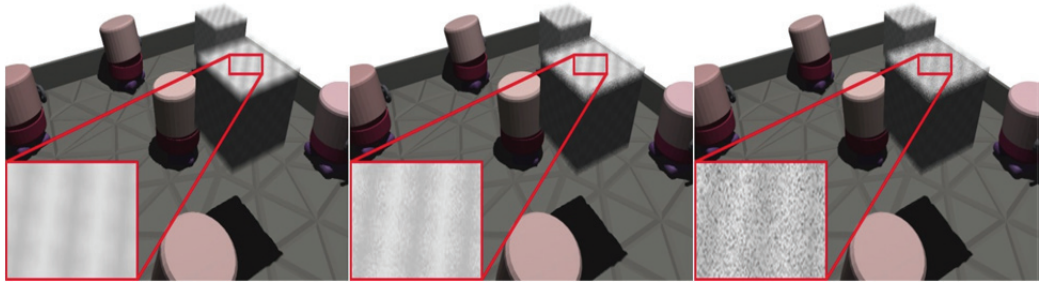
Performance: Fig. 11 presents the timing for each step in our algorithm and TSM when rendering the scene in Fig. 10. As the number of samples per pixel increases, TSM becomes increasingly slower compared to our algorithm. Although we store all visible depth-time ranges for each pixel in a shadow map, our algorithm performs the shadow pass faster than TSM. The reason for this is that the overhead for draw calls and state changes in TSM is larger than that in our method. In general, the more samples per pixel, the slower the model performance. However, the shadow test in our algorithm takes more time than that in TSM because for each sample, we must load each visible depth-time range and perform a time check prior to the depth comparison.

Memory: Fig. 12 presents a memory comparison between our algorithm and TSM when generating

shadow maps. The required memory for the shadow map in our algorithm depends on scene complexity, while that for TSM depends on the number of samples per pixel. In our algorithm, scene complexity is determined as follows. First, we check the number of fragments N_f stored by the collection sub-pass and the number of visible depth-time ranges from the generation sub-pass N_r. We then compute the required memory for the shadow map as

$$Memory = N_f * fragment\_size + N_r * range_{size} + 2 * screen\_resolution * 4 \quad (2)$$

where "2 ∗ screen_resolution ∗ 4" is the size of two index buffers in which each index is a 4-byte unsigned integer. The value of "*fragment_size*" is 16 (a depth, time, triangle index, and pointer) and the range size is 24 (two times, two depths, a triangle index, and pointer). In TSM, the shadow map has the same resolution as the screen resolution and the required memory for the shadow map is the product of the number of samples per pixel, number four (4 bytes per depth value), and shadow map resolution.



Reference | Our method (8 samples per pixel) | TSM (8 samples per pixel)

Fig. 9. Image quality comparison between our algorithm and TSM utilizing the same number of samples per pixel. The light source is on the top and the two cubes move toward the light source. The images rendered by TSM contain severe self-shadow artifacts, while those rendered by our method do not. The total number of triangles is 12,000.

Reference | Our method (20 samples per pixel) | TSM (20 samples per pixel)

Fig. 10. Image quality comparison between our algorithm and TSM utilizing the same number of samples per pixel. The light source is in front of the character and the leg is moving toward the light source. The images rendered by TSM contain severe self-shadow artifacts, while those rendered by our method do not. The total number of triangles is 265,000.

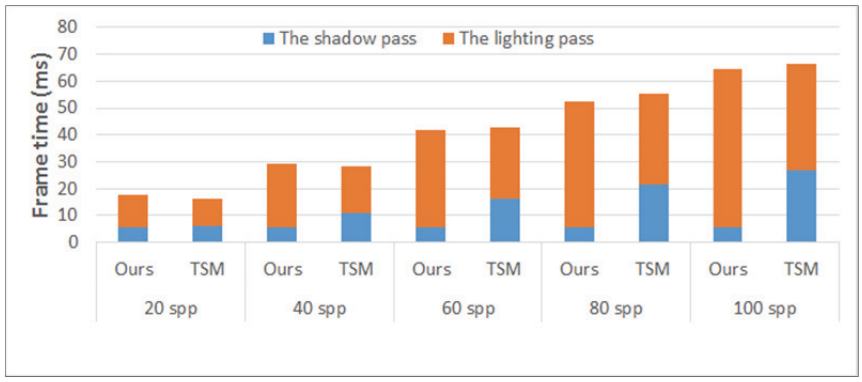


Fig. 11. Timing for each step of our algorithm and TSM when rendering the scene in Fig. 9.

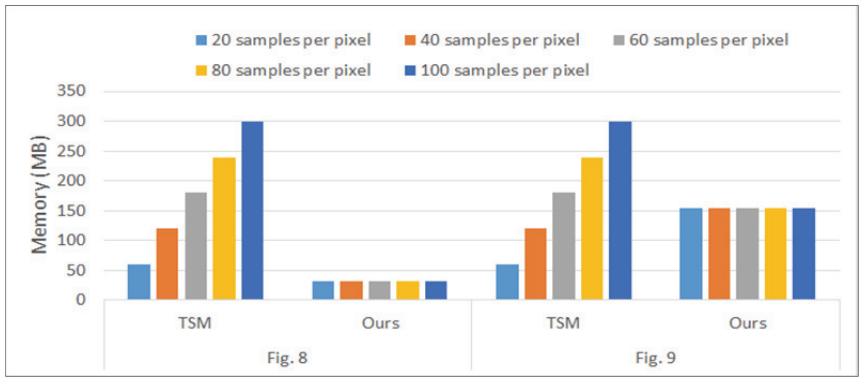


Fig. 12. Measurements of required memory for the shadow map in our algorithm and TSM with various numbers of samples per pixel.

6. Conclusions

We presented a novel algorithm that renders motion-blurred shadows utilizing a depth-time ranges shadow map. First, we generate a shadow map, which stores many visible depth-time ranges for each pixel. Each time range expresses a period with two depths from the start and end of a frame in which a particular geometry was visible to the light source. In a second pass, we utilize stochastic rasterization to render the scene from the camera to produce motion-blurred shadows. For each visible sample, we project it into the light space then load each visible depth-time range to perform shadow tests. All test results are averaged to calculate the final pixel color.

There are various methods to store all visible depth-time ranges in a per-pixel linked list [23]. In our current implementation, we allocate at a fixed amount of memory for storing all visible depth-time ranges, meaning some memory is wasted when only a small portion of the fixed amount is utilized. Therefore, in the future work, we will apply the method from [24] to our algorithm to allocate memory dynamically. Additionally, a sample is occluded if there is any blocker on the path from the light to that sample. Therefore, we will find a way to reduce the number of visible depth-time ranges that are stored by the generation sub-pass that do not affect the shadow test's results.

Acknowledgement

Models were downloaded from the Utah 3D Animation Repository and McGuire Computer Graphics Archive (http://casual-effects.com/data/). This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2018R1D1A1B07050099).

Appendix 1. Pseudo-code

```
//-----------------------------------------------The shadow pass-----------------------------------------------
    //-------------------vertex shader----------------------------
    vertex_shader(float3 vertex_position)
    {
        Transforming vertex positions in the previous frame and in the current frame to the view space

    }


    // Input: Six vertices of a triangle, which represents two positions of the triangle in the previous
    frame and in the current frame
    // Output: Triangles (up to 14) of a prism
    geometry_shader(vertex_posision input[6])
    {
        If ( three vertices of the current triangle is outside or behind the view frustum )
            Do nothing and exit

        If ( the current triangle is back-faced in the previous frame and in the current fame )
            Do nothing and exit

        Emitting two triangles at the previous and the current frames.

        // For each edge pair(PQ, P'Q'), where PQ and P'Q' are two edges in the previous and in the
    current frame, respectively
        // We check if there is an intersection between these two edges

        for (edge_index = 0, edge_index < 3, edge_index ++)
        {
            Check and find a 2D intersection between two edges at the previous frame and the current
    frame.

            If ( there is no intersection )
                Find a center point and use it to emit four triangles.
            else
                Using the intersection point to emit two triangles.
        }

    //--------------------pixel shader----------------------------
```

```
pixel_shader()
{
    //Increasing the current counter in the fragment_buffer to 1 uint
    counter = counter + 1

    Using the current fragment's position to compute an index value. We use the index to locate a
linked_list in the index_buffer.

    Making a new element using a depth value of the current fragment, a triangle id that the
current fragment belongs to, and time when is the current fragment is visible.

    Storing the new element to the to the linked list.
}


//-----------------------------------------------The generation subpass-----------------------------------------
    // a full-screen rectangle
    vertex_shader()
    {
        Transforming each vertex of a full-screen rectangle to the clip space.
    }

    //------------------pixel shader----------------------
    // process each pixel
    pixel_shader()
    {
        Loading all elements stored the current fragment using the current fragment' position.


        Iterating through all elements and find two fragments belonging to the same triangle (having
    the same triangle id). Then, using these two fragments to make an interval.

        Storing all intervals, as a linked-list, to the shadow map at the current pixel's position.
    }


//-----------------------------------------------The lighting pass-----------------------------------------------
// The lighting pass in our paper is extended from a paper named "Real-Time Stochastic Rasterization
on Conventional GPU Architectures", a pseudo code of this paper is on authors' homepage

    //------------------vertex shader----------------------
    vertex_shader()
    {
        Transforming each vertex to the view space.

        Transfering texture coordinates and a normal vector to the geometry shader.
    }


    //------------------geometry shader----------------------
```

```
ST_GS()
{
    Computing normal vectors of the current triangle in the previous frame and in the current
frame.

    If ( the current triangle is back-faced in the previous frame and in the current fame )
        Doing nothing and exit

    // We make a convex hull which contains all vertices
    // and handle all cases of a convex hull in the view space.

    Computing the minimum depth and the maximum depth of vertices in the view space.

    If ( all vertices are behind or outside the view frustum )
       Do nothing and exit.

    else if ( the convex hull is partially behind the view frustum)
    {
        Projecting vertices of the convex hull, which behind the view frustum, to the near plane.
    }
    else
    {
        Making and emitting a convex hull.
    }
}

//------------------pixel shader----------------------
ST_PS(PS_INPUT input)
{
    Using the current fragment's position to compute a sample positions in the current pixel.

    Using the sample position to load a random time (t) from a texture.

    Finding the current position of a moving triangle at time t using linear interpolation.

    Shooting a ray from camera through the current sample.

    Finding an intersection between the ray and the current triangle at t.

    if ( there is no intersection)
        Do nothing and exit.

    Converting a depth value of the intersection point in the view space.

    // Projecting the intersection point to the light space
    if ( The intersection point is inside the view frustum of the light )
    {
        Using the intersection point's xy position to load each depth-time range from the shadow
map.
```

```
            For each depth-time range
            {
                If ( the current intersection point is occluded the current depth-time range at time t )
                {
                    //The current intersection point is in shadow.
                    Computing color for the current intersection point.
                    Stopping the loop.
                }
            }

            Outputting the final color
        }
    }
```

References

[1] E. Eisemann, M. Schwarz, U. Assarsson, and M. Wimmer, *Real-Time Shadows*. Boca Raton, FL: CRC Press, 2011.

[2] A. Woo and P. Poulin, *Shadow Algorithms Data Miner*. Boca Raton, FL: CRC Press, 2012.

[3] F. Navarro, F. J. Seron, and D. Gutierrez, "Motion blur rendering: state of the art," *Computer Graphics Forum*, vol. 30, no. 1, pp. 3-26, 2011.

[4] 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.

[5] T. Akenine-Moller, J. Munkberg, and J. Hasselgren, "Stochastic rasterization using time-continuous triangles," in *Proceedings of ACM SIGGRAPH-Eurographics Symposium on Graphics Hardware*, San Diego, CA, 2007.

[6] 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.

[7] 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.

[8] J. Palmer, "Analytical motion blurred shadows," Master's thesis, Lund University, Sweden, 2011.

[9] T. Lokovic and E. Veach, "Deep shadow maps," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, New Orleans, LA, 2000, pp. 385-392.

[10] C. Yuksel and J. Keyser, "Deep opacity maps," *Computer Graphics Forum*, vol. 27, no. 2, pp. 675-680, 2008.

[11] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, "Stochastic transparency." *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 8, pp. 1036-1047, 2011.

[12] M. McGuire and E. Enderton, "Colored stochastic shadow maps," in *Proceedings of Symposium on Interactive 3D Graphics and Games*, San Francisco, CA, 2011, pp. 89-96.

[13] M. Andersson, J. Hasselgren, J. Munkberg, and T. Akenine-Moller, "Filtered stochastic shadow mapping using a layered approach," *Computer Graphics Forum*, vol. 34, no. 8, pp. 119-129, 2015.

[14] M. Andersson, J. Hasselgren, and T. Akenine-Moller, "Depth buffer compression for stochastic motion blur rasterization," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, Vancouver, Canada, 2011, pp. 127-134.

[15] W. Donnelly and A. Lauritzen, "Variance shadow maps," in *Proceedings of the 2006 Symposium on Interactive 3D Graphicsa Games*, Redwood City, CA, 2006, pp. 161-165.

[16] J. P. Guertin, M. McGuire, and D. Nowrouzezahrai, "A fast and stable feature-aware motion blur filter," in *Proceedings of High-Performance Graphics*, Lyon, France, 2014, pp. 51-60.
[17] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-Time Rendering*. Boca Raton, FL: CRC Press. 2008.
[18] M. Salvi, J. Montgomery, and A. Lefohn, "Adaptive transparency," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, Vancouver, Canada, 2011, pp. 119-126.
[19] C. A. Burn, 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.
[20] 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*, Vinicn, Slovakia, 2011.
[21] J. C. Yang, J. Hensley, H. Grun, and N. Thibieroz, "Real-time concurrent linked list construction on the GPU," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1297-1304, 2010.
[22] M. Maule, J. L. Comba, R. P. Torchelsen, and R. Bastos, "A survey of raster-based transparency techniques," *Computers & Graphics*, vol. 35, no. 6, pp. 1023-1034, 2011.
[23] S. Laine and T. Karras, "Efficient triangle coverage tests for stochastic rasterization," NVIDIA *Technical Report No. NVR-2011-003*, 2011.
[24] A. Vasilakis and I. Fudos, "S-buffer: sparsity-aware multi-fragment rendering," in *Proceedings of the 33rd Annual Conference of the European Association for Computer Graphics (Eurographics),* Cagliari, Italy, 2012, pp. 101-104.

MinhPhuoc Hong https://orcid.org/0000-0003-2638-1932

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

Kyoungsu Oh https://orcid.org/0000-0002-7106-7865

He received his B.S. degree and Ph.D. from Seoul National University. His research interests include real-time rendering, shadows, global illumination, and motion blur.