Hong and Oh *Hum. Cent. Comput. Inf. Sci. (2017) 7:22*
DOI 10.1186/s13673-017-0102-9

Human-centric Computing and Information Sciences

RESEARCH Open Access

Efficient motion blurred shadows using a temporal shadow map

MinhPhuoc Hong* and Kyoungsu Oh

*Correspondence: hmphuoc1985@gmail.com
Soongsil University, Room 509, 50, Sadang-ro, Dongjak-gu, Seoul 07027, South Korea

Abstract

In this paper, we propose a novel algorithm that renders motion blurred shadows efficiently on GPUs using a temporal shadow map. Each triangle moves during a frame and for each pixel, it is visible to the light in a range of time. The main idea of our algorithm is to find such visible ranges and store in the temporal shadow map. For each sample which is visible to the camera at a certain time, we can determine whether it is shadowed or lit using visible ranges in the temporal shadow map. Thus, our algorithm solves a time-mismatch problem in the time-dependent shadow mapping algorithm. Furthermore, we use a coverage map to reduce memory footprint used for the temporal shadow map.

Keywords: Real-time rendering, Motion blur, Motion blurred shadows, Visibility

Background

According to recent research described by ThanhBinh [1], Agarwal and Bedi [2], image processing is an important part of modern graphics and motion blur is an essential effect in that field. Instead of processing images, we render images with motion blurred shadows. Motion blurred shadow effect enhances the sense of realism experienced by users. When a geometry is blurred, its shadows should be blurred as well. However, there are few proposed algorithms for rendering motion blurred shadows.

Because the shadow casters, the shadow receivers and the light source can move during a frame, motion blurred shadows rendering is a challenging problem in real-time rendering. For a given pixel which is visible to the camera at a certain time, it is difficult to determine if the current pixel is occluded or not with respect to the light.

A brute force method renders a scene with shadow many times and then averages the results to produce correct motion blurred shadows. However, this approach is extremely slow, so it is not suitable for the real-time rendering. Stochastic sampling based approaches use multi-samples per pixel, with each sample has a unique random time, to render motion blurred shadows. However, time mismatch when generating and sampling a shadow map causes visual artifacts.

Contrary to previous approaches, we seek an approach that finds a range of time when each geometry is visible to the light for a given pixel. In this paper, we introduce a novel algorithm that renders motion blurred shadows efficiently on GPUs using a temporal shadow map. During a frame, at each pixel, each moving triangle is visible to the light

© The Author(s) 2017. This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

source in a range of time. For each pixel of a shadow map, we store all visible time ranges along with depth values of all moving triangles. For a sample which is visible to the camera, we can determine if it is shadowed at a certain time. *Thus, our algorithm renders motion blurred shadows and solves time mismatch problem in the time-dependent shadow map algorithm*. We further extend our algorithm to reduce the total number of visible time ranges stored in the temporal shadow map and simplify the shadow tests.

A summary of this paper is as follows: In "Related works", we briefly review related works. "Motion blurred shadows rendering" and "Extension" present our algorithm and its extension, respectively. Finally, we show comparison results, performance and memory analysis in "Evaluation".

Related works

Many algorithms are proposed for rendering motion blur and shadows. Therefore, we refer readers to Navarro et al. [3] and Eisemann et al. [4] for an overview of motion blur and shadow mapping, respectively.

Haeberli and Akeley [5] render a scene with shadow many times and average the results to produce blurred images with motion blurred shadows. However, this approach has ghosting artifacts at low sampling rates. But increasing the sampling rate impacts performance substantially.

For each pixel in a shadow map, deep shadow map [6] stores a list of semi-transparent surfaces. The visibility of a surface at a given depth is computed as $\prod_{p_z < z_i}(1 - \alpha_i)$, where z_i and α_i are a depth and an opacity of a surface. To render motion blurred shadows, authors assign a random time for each sample and all samples at the same depth are averaged together to an opacity of a surface. Therefore, such surfaces are regarded as transparent blockers. This approach only works for static receivers. As receiver moves, the time dimension is collapsed and motion blurred shadows are rendered incorrectly.

Distributed ray tracing [7] renders motion blur and soft shadows by shooting many rays at a pixel at different times and averaging all visible rays to produce the final image. But the computation cost of this approach is prohibitive. Akenine-Möller et al. [8] use the stochastic rasterization to render motion blurred shadows using time-dependent shadow maps (TSM). This algorithm uses many samples per pixel and each sample has a random time. As rendering from the light source and from the camera, each sample has a random time t_s and t_r, respectively. This algorithm uses the stratified sampling to ensure that t_s and t_i belong to the same segment of the exposure interval. *The time mismatch causes visual artifacts. Samples should be lit are shadowed or samples should be shadowed are lit. Additionally, rendered images have self-shadow artifacts at low sampling rates when geometries move toward the light*. Later, this idea is implemented in the current GPUs by McGuire et al. [9].

Inspiring the idea of Akenine-Möller et al. [8], Andersson et al. [10] render motion blurred shadows using depth layers. This approach generates time-dependent shadow maps and then divide into multiple layers using a method described by Andersson et al. [11]. Subsequently, this approach projects all samples along an average motion vector of each layer, and performs shadow lookups in this representation. Finally, this approach uses a statistical method described by Donnelly and Lauritzen [12] to approximate the visibility of a sample. Therefore, this approach has the same problem with the variance

shadow map. Apart from that problem, artifacts might occur when samples in the same layer move in the different directions and speeds. Authors alleviate but not address completely this problem using a tile-variance approach described by Guertin et al. [13].

Motion blurred shadows rendering

Figure 1 gives an overview of our algorithm. Our algorithm composes of two passes: a shadow pass and a lighting pass. First, we present our main idea and describe the details of the shadow pass in "Shadow pass". Later, we describe the lighting pass in "Lighting pass". Throughout the presentation, we use the term triangle, but it can naturally extend to a general geometry which might have animation data defined by Myeong-Won et al. [14].

Shadow pass

We assume a triangle linearly moves from the beginning ($t = 0$) to the end ($t = 1$) of a frame. The position of this triangle at $t = 0$ and $t = 1$ is ABC and $A'B'C'$, respectively. To generate motion blurred shadows for this triangle, a brute force method renders this triangle many times and averages all rendered images. The goal is to find a visible time range of this triangle at each pixel and compute an average color along this time range. At the

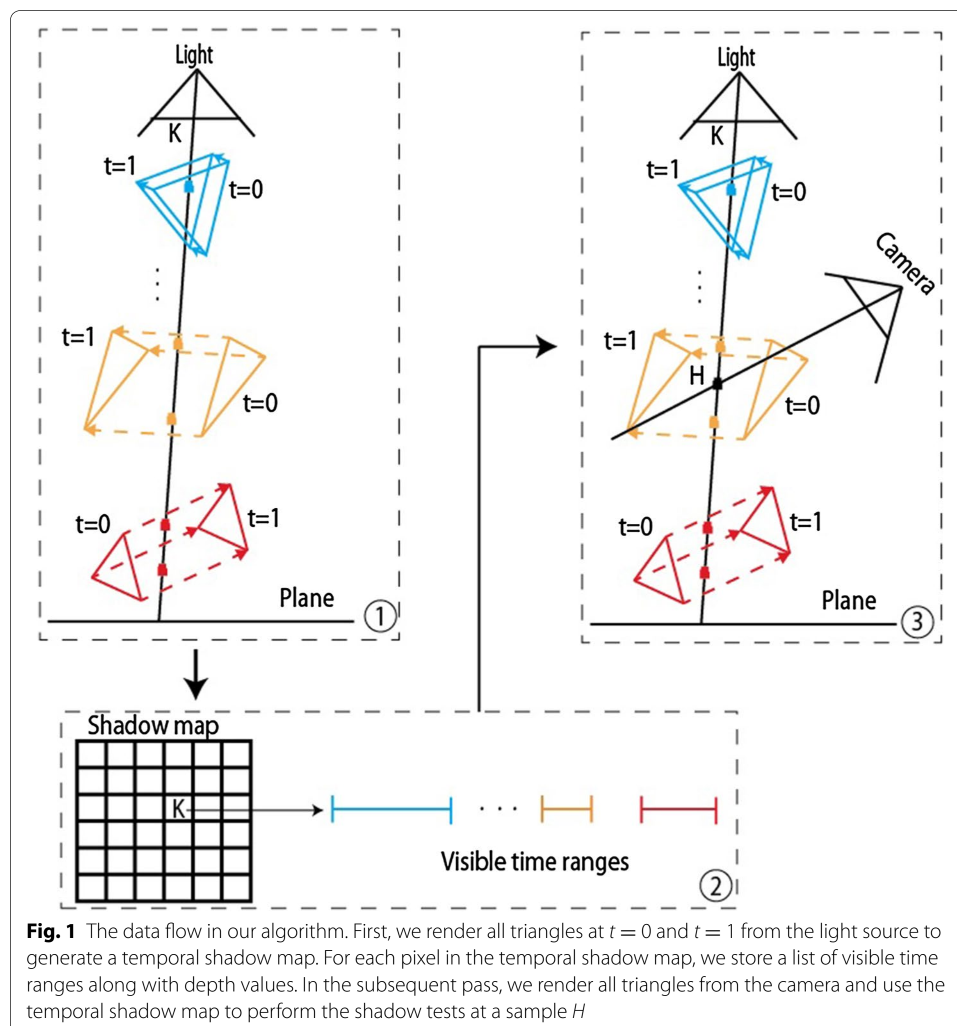

Fig. 1 The data flow in our algorithm. First, we render all triangles at $t = 0$ and $t = 1$ from the light source to generate a temporal shadow map. For each pixel in the temporal shadow map, we store a list of visible time ranges along with depth values. In the subsequent pass, we render all triangles from the camera and use the temporal shadow map to perform the shadow tests at a sample *H*

pixel *P*, this triangle is visible through five intersection points at five times t_1, t_2, t_3, t_4, and t_5, in Fig. 2a. From this observation, our main idea is to render this triangle only once and get a visible time range of this triangle by finding the first and the last intersection points (F_1 and F_2) at the first time (t_1) and the last time (t_5), respectively. So at the pixel *P*, we can compute the visible time range of this triangle and know a depth range from F_1 to F_2.

To implement our main idea, we assign a time to each vertex of two triangles *ABC* ($t = 0$) and *A′B′C′* ($t = 1$). Next, we use *ABC* and *A′B′C′* to form a prism, in Fig. 2b, and then triangulate this prism. For each pixel, GPU generates two points (F_1 and F_2), with each point having an interpolated time and a depth value. These two points form a visible time range of the triangle which can be computed as $|t_1 - t_5|$.

With this main idea, we render a scene from the light to generate a temporal shadow map. For each pixel in the temporal shadow map, we store a list of tuples with five values in the form: ($t_1_t_2$, $z_1_z_2$, *id*), where (t, z) is an interpolated time, a depth value of a generated point such as F_1. "*id*" is an *id* of a triangle in which the current fragment belongs to, and this triangle *id* is used to address self-shadow artifacts in the lighting pass.

Lighting pass

In this pass, we use the stochastic rasterization [5] to render a scene from the camera. A triangle covers a set of pixels when moving from the start ($t = 0$) to the end ($t = 1$) of a frame. We use two positions of this triangle at $t = 0$ and $t = 1$ to make a convex hull to cover all such pixels. There are multi-samples per pixel and each sample has a random time. To check whether the current sample is visible or not, we shoot a ray from the camera through the current sample and then perform a ray-triangle intersection. If there is an intersection, the current sample is visible.

To perform the shadow lookup at a visible sample, we do as follows. First, we project this sample to the temporal shadow map and load each tuple ($t_1_t_2$, $z_1_z_2$, *id*). If the visible sample's time (t_s) is inside the visible time range [t_1, t_2], we find a depth value at t_s using the linear interpolation along the depth range [z_1, z_2] and compare the interpolated

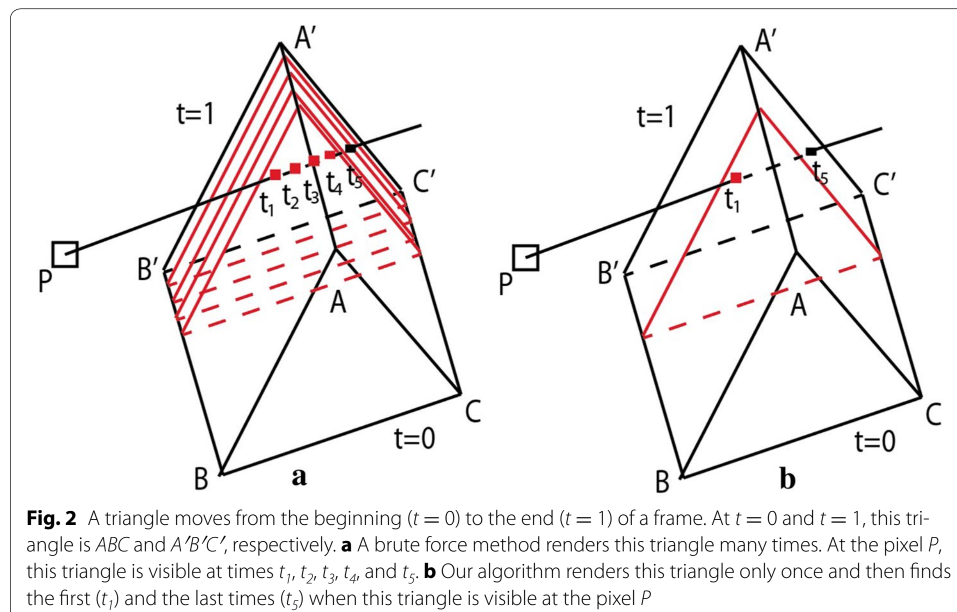

Fig. 2 A triangle moves from the beginning ($t = 0$) to the end ($t = 1$) of a frame. At $t = 0$ and $t = 1$, this triangle is *ABC* and *A′B′C′*, respectively. **a** A brute force method renders this triangle many times. At the pixel *P*, this triangle is visible at times t_1, t_2, t_3, t_4, and t_5. **b** Our algorithm renders this triangle only once and then finds the first (t_1) and the last times (t_5) when this triangle is visible at the pixel *P*

depth with the sample's depth. Finally, we perform shading and average all samples' color in a pixel. To address the self-shadow artifacts in TSM, we check if the current sample does not belong to the current triangle prior the shadow test.

Extension

In our algorithm, the temporal shadow map consumes a lot of graphics memory by storing all visible time ranges. In this section, we describe how to use a coverage map to reduce graphics memory used by the temporal shadow map. The idea of this step is to find the nearest triangle that continuously covers a pixel in a temporal shadow map, Fig. 3. Therefore, we do not need to store all visible time ranges at this pixel and thereby the graphics memory is reduced. The data flow in this extension is shown in Fig. 4.

The first pass

We render all triangles at the start ($t = 0$) and the end ($t = 1$) of a frame from the light source and each triangle is assigned a triangle id. Then, we use two positions (at $t = 0$ and $t = 1$) of each triangle to form a prism, in Fig. 4a. After that, we find the nearest triangle using the conventional z-buffer and then store a triangle id and time of this triangle to a *nearest map*. The nearest map has the same resolution as the temporal shadow map and each pixel of this map stores two 32-bit floating point values, i.e., one for a triangle id and the other for a time. In this pass, we use a depth test function (LESS), enable the depth write, and disable the stencil operations. Note that we only clear a depth map in the first pass.

The second pass

Again, we render all triangles at $t = 0$ and $t = 1$ from the light source to find an overlap region between two positions of each triangle, Fig. 4b. Such a region denotes an area where a triangle continuously covers a pixel during a frame. To this end, we use a *coverage map* which has the same resolution as the temporal shadow map, and there is a one-to-one

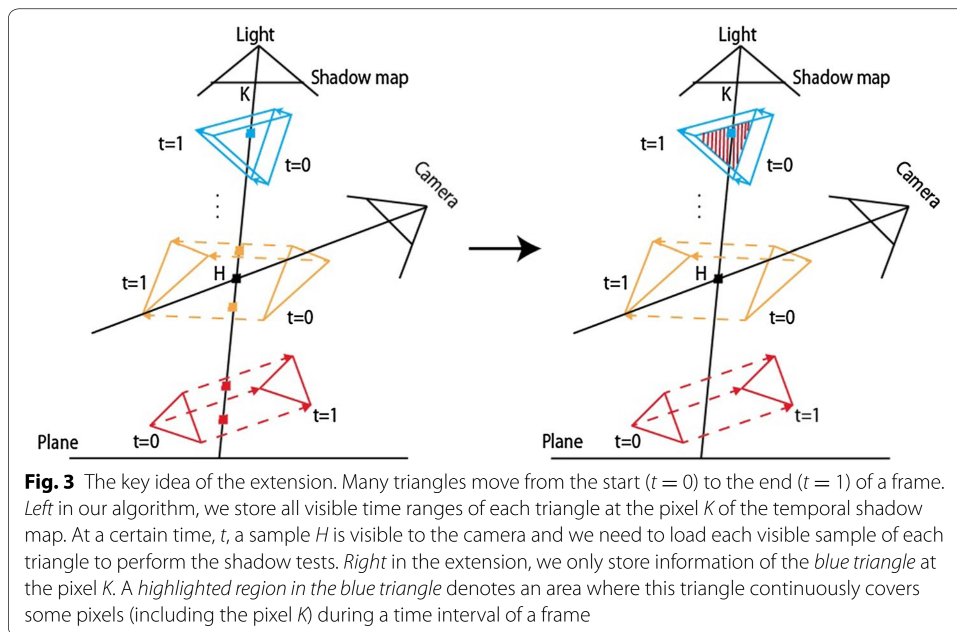

Fig. 3 The key idea of the extension. Many triangles move from the start ($t = 0$) to the end ($t = 1$) of a frame. *Left* in our algorithm, we store all visible time ranges of each triangle at the pixel *K* of the temporal shadow map. At a certain time, *t*, a sample *H* is visible to the camera and we need to load each visible sample of each triangle to perform the shadow tests. *Right* in the extension, we only store information of the *blue triangle* at the pixel *K*. A *highlighted region in the blue triangle* denotes an area where this triangle continuously covers some pixels (including the pixel *K*) during a time interval of a frame

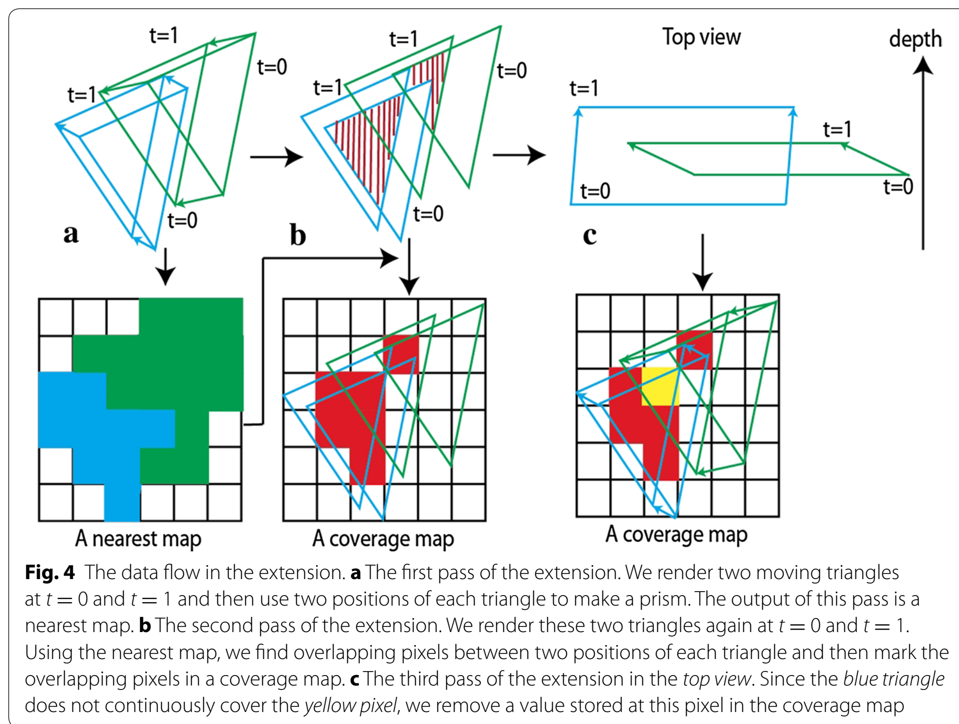

Fig. 4 The data flow in the extension. **a** The first pass of the extension. We render two moving triangles at $t = 0$ and $t = 1$ and then use two positions of each triangle to make a prism. The output of this pass is a nearest map. **b** The second pass of the extension. We render these two triangles again at $t = 0$ and $t = 1$. Using the nearest map, we find overlapping pixels between two positions of each triangle and then mark the overlapping pixels in a coverage map. **c** The third pass of the extension in the *top view*. Since the *blue triangle* does not continuously cover the *yellow pixel*, we remove a value stored at this pixel in the coverage map

correspondence between pixels in the coverage map and pixels in the temporal shadow map. The coverage map holds a 32-bit floating point value at each pixel. If this value is "−1", there is no triangle that continuously covers the current pixel during a time interval of a frame. Otherwise, this value is a triangle id of a triangle occupying the current pixel continuously. In this pass, we disable the depth test, enable the depth write. The following pseudo code shows how to update a value of the coverage map at the current pixel.

```
Coverage_MapSecond_Pass ( The current triangle id (id), a time (t), the
nearest map (NM))
{
        (stored_id, stored_t) = NM.Load(the current pixel's position)
        If (id != stored_id)
            Discard the current pixel
        If (|stored_t – t| < 1)
            Discard the current pixel.
        Store id to the coverage map at the current pixel's position.
}
```

The third pass

We render all triangles at $t = 0$ and $t = 1$ and then use two positions of each triangle at $t = 0$ and $t = 1$ to make a prism. The blue triangle does not continuously covers the yellow pixel. Therefore, we need to reset a value of the coverage map to "-1", Fig. 4c. In this pass, we use a depth test function (LESS), disable the depth write. The following pseudo code shows how to perform this pass.

```
Third_Pass( The current triangle id (id), the coverage map(CM))
{
    cover_id = CM.Load(the current pixel's position)
    If (id == cover_id)
        Discard the current pixel and do not change the coverage map.
    Reset the coverage map at the current position to "-1".
}
```

For a given pixel in the temporal shadow map, we can skip storing the number of visible time ranges using the coverage map. To do this, we check a value of the coverage map at the current pixel before storing visible time ranges. If this value is positive, we exit and do not store any visible time ranges. Otherwise, we insert each visible time range to the current pixel of temporal shadow map. And the following pseudo code shows how to perform the shadow tests using the coverage map and the temporal shadow map.

```
Final_Shadow_Test( A sample position (X), the current triangle id (id), a
time (t), the coverage map(CM))
{
    Y = project X to the light space.
    covered_id = CM.Load(Y).
    If (covered_id == id) // The triangle cannot cast shadows on itself
        The current sample is lit.
    If (covered_id > -1)
        The current sample is shadowed.
    If (covered_id == -1)
        Load each visible time range and perform the shadow tests at t.
}
```

Evaluation

Our algorithm is implemented using DirectX 11, HLSL 5.0 with a GTX 980 Ti 6 GB graphics card. In the shadow pass, we generate and store a temporal shadow map in graphics memory using a per-pixel linked list described by Barta et al. [15], Burns [16] and Salvi et al. [17]. In the lighting pass, we use the stochastic rasterization described by McGuire et al. [5] with a fast ray-triangle intersection [18] and multi-sampling. For comparisons, we implement a brute force method [3] using 3000 samples to generate reference images and the time-dependent shadow mapping (TSM) using the stochastic rasterization [5]. In all rendered images, the shadow map used in the TSM have 1024×768 resolution.

When geometries animate, their shadows should be blurred as well. Therefore, we render two scenes having animation characters to compare with TSM in terms of image quality, in Figs. 5 and 6. Since the quality of blurred shadows is better when increasing the number of samples per pixel, we also compare with TSM in terms of the rendering time by varying the number samples per pixel, in Fig. 7.

Our algorithm vs. stochastic rasterization algorithm

Figures 5 and 6 show image quality comparisons between our algorithm and TSM using multi-sampling with the same number of samples per pixel. Due to a small number of samples per pixel, images rendered by our algorithm and TSM have noise. However, images rendered by TSM have visual artifacts (a green highlighted inset in Fig. 5 and

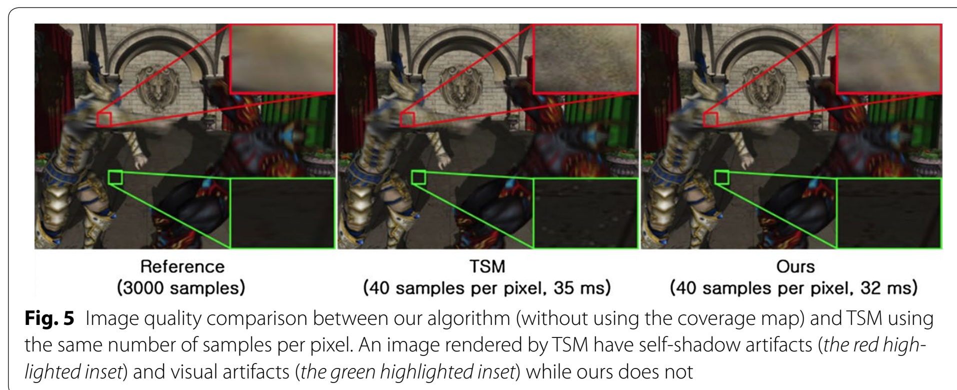

Fig. 5 Image quality comparison between our algorithm (without using the coverage map) and TSM using the same number of samples per pixel. An image rendered by TSM have self-shadow artifacts (*the red highlighted inset*) and visual artifacts (*the green highlighted inset*) while ours does not

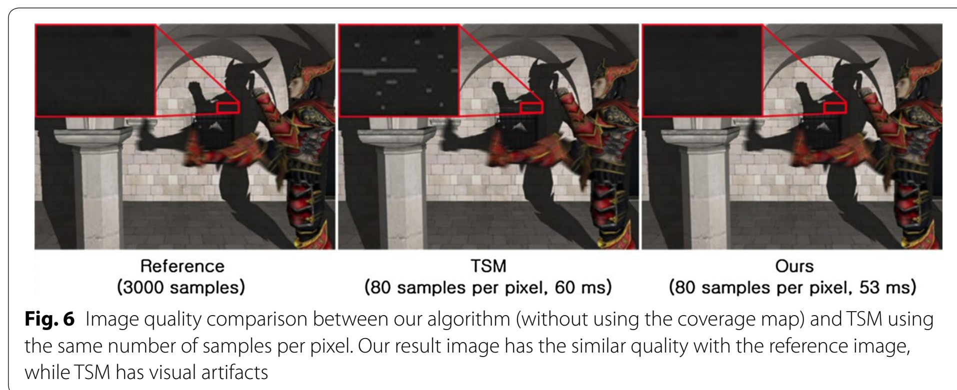

Fig. 6 Image quality comparison between our algorithm (without using the coverage map) and TSM using the same number of samples per pixel. Our result image has the similar quality with the reference image, while TSM has visual artifacts

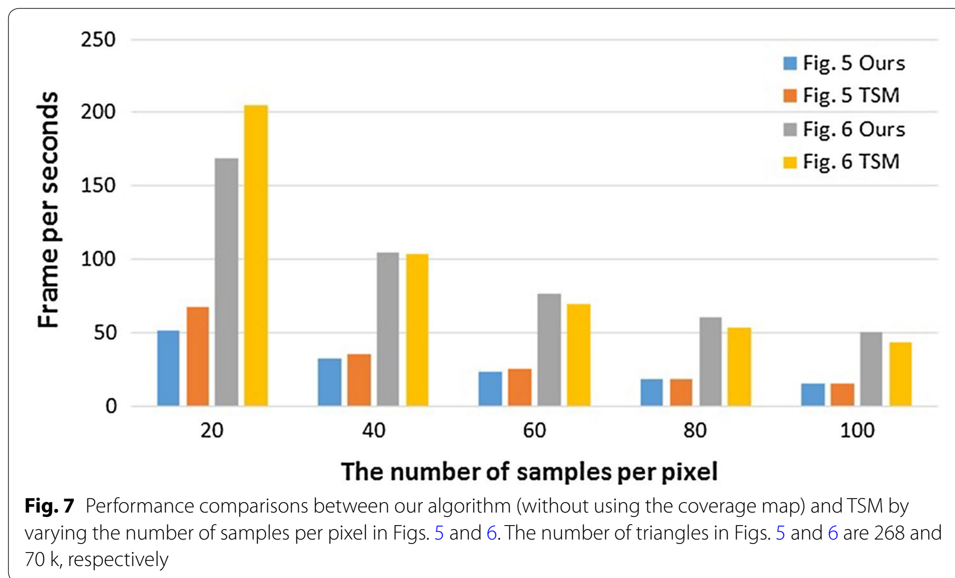

Fig. 7 Performance comparisons between our algorithm (without using the coverage map) and TSM by varying the number of samples per pixel in Figs. 5 and 6. The number of triangles in Figs. 5 and 6 are 268 and 70 k, respectively

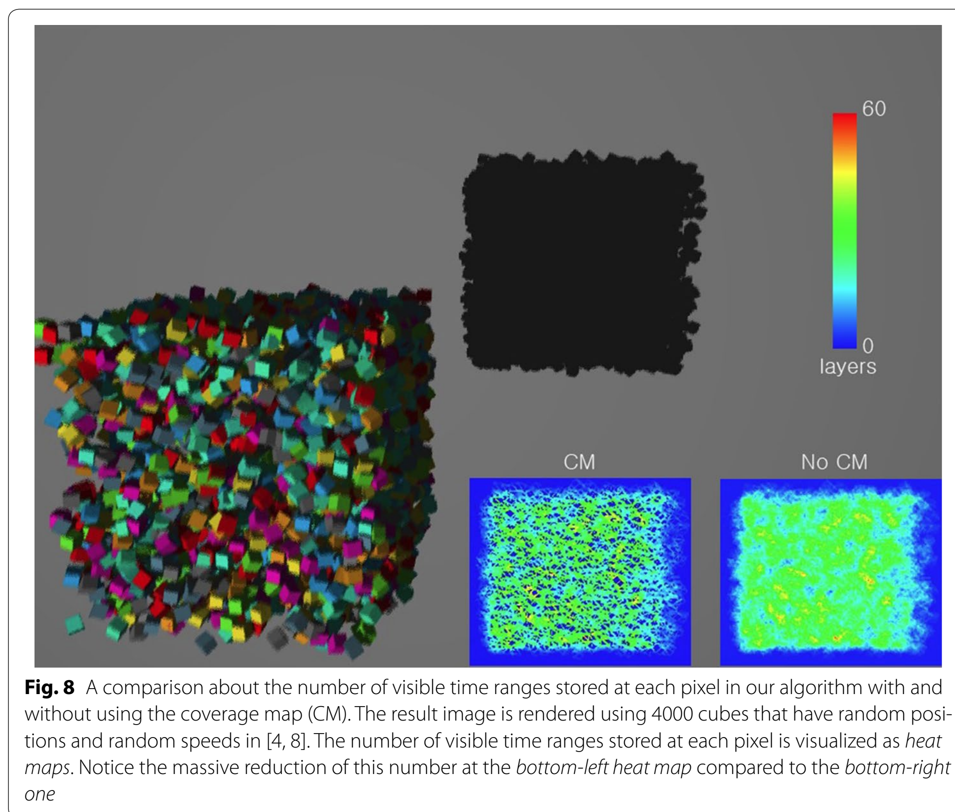

Fig. 8 A comparison about the number of visible time ranges stored at each pixel in our algorithm with and without using the coverage map (CM). The result image is rendered using 4000 cubes that have random positions and random speeds in [4, 8]. The number of visible time ranges stored at each pixel is visualized as *heat maps*. Notice the massive reduction of this number at the *bottom-left heat map* compared to the *bottom-right one*

a red highlighted inset in Fig. 6) in the shadow areas while ours does not. The reason for this is that TSM uses two random times, t_s and t_r, for the same sample. t_s and t_r are used when rendering from the light and from the camera, respectively. Time mismatch results in incorrect shadow tests. Additionally, the red highlighted area in Fig. 5 shows that TSM has self-shadow artifacts.

Figure 7 shows the performance comparison between our algorithm and TSM by varying the number of samples per pixel. As increasing the number samples per pixel, the rendering time in both algorithms increases. But in the shadow pass, the overhead of draw calls and state changes in TSM is higher than ours. The reason for this is that the number of draw calls in TSM is proportional to the number of samples per pixel. For generating the shadow map, TSM renders a scene many times, while our algorithm renders the scene once.

Memory and performance

Generally, scenes have a relative high depth complexity which requires more graphics memory for the temporal shadow map. Therefore, we render a large number of cubes that have random positions and random moving speeds to evaluate our algorithm in terms of performance and graphics memory. We do the evaluation in two different scenarios. In the first scenario, we vary the number of random cubes and each cube's moving speed is randomized in a fixed speed range, Figs. 9 and 10. In the second scenario, we increase the moving speed of each cube, Figs. 11 and 12.

We use the nearest map and the coverage map to reduce the graphics memory used for the temporal shadow map. Both the nearest map and the coverage map have the same resolution as the shadow map (1024×768). For each pixel, the nearest map stores two 32-bit floating point values and the coverage map holds a single 32-bit floating point value. So it requires about 9 MB memory for both maps. The graphics memory used for the temporal shadow map relies on the number of visible time ranges stored at each pixel.

Figure 8 illustrates that using the coverage map significantly reduces the total number of visible time ranges stored in the temporal shadow map. Memory comparisons and

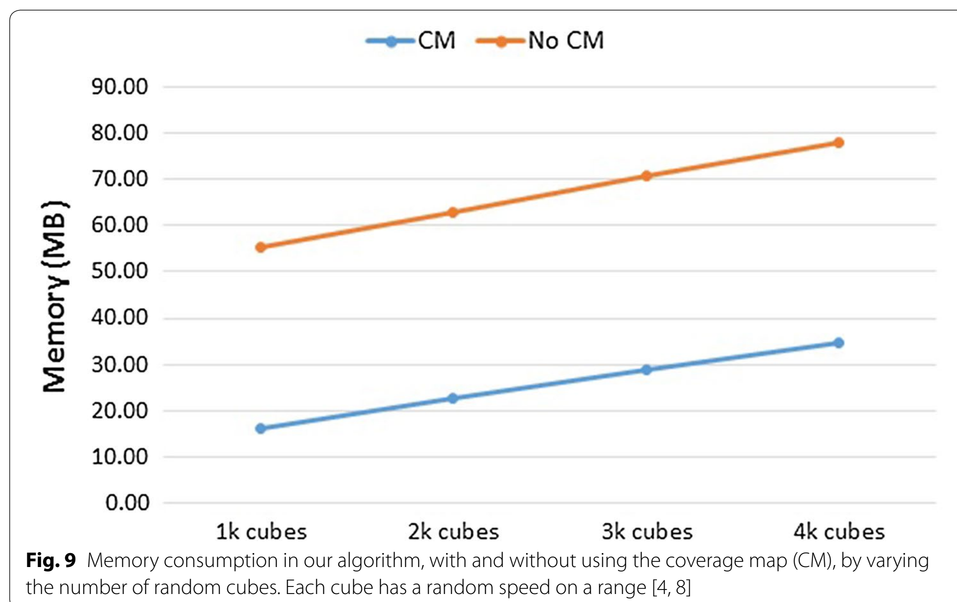

Fig. 9 Memory consumption in our algorithm, with and without using the coverage map (CM), by varying the number of random cubes. Each cube has a random speed on a range [4, 8]

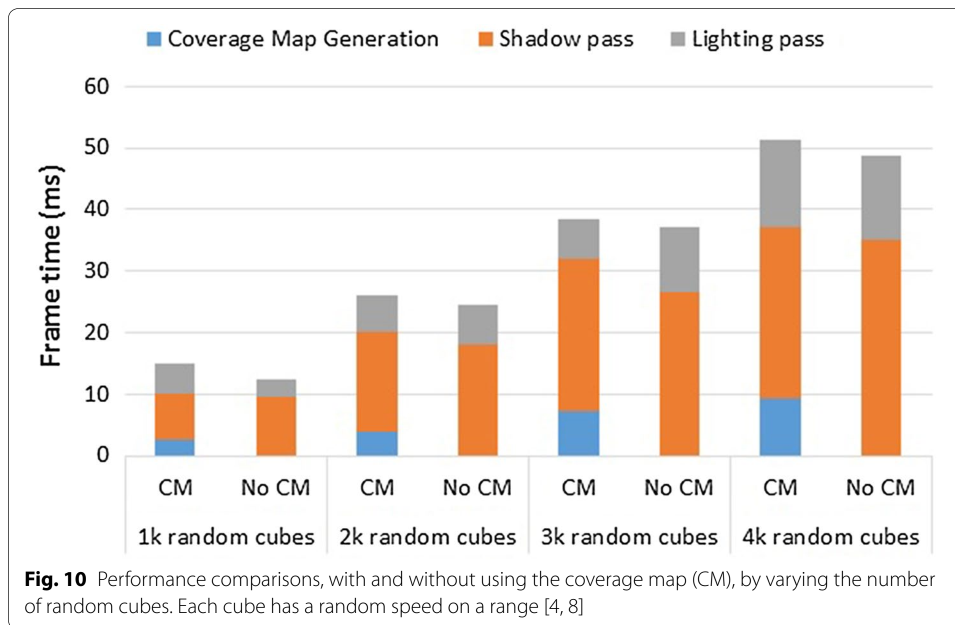

Fig. 10 Performance comparisons, with and without using the coverage map (CM), by varying the number of random cubes. Each cube has a random speed on a range [4, 8]

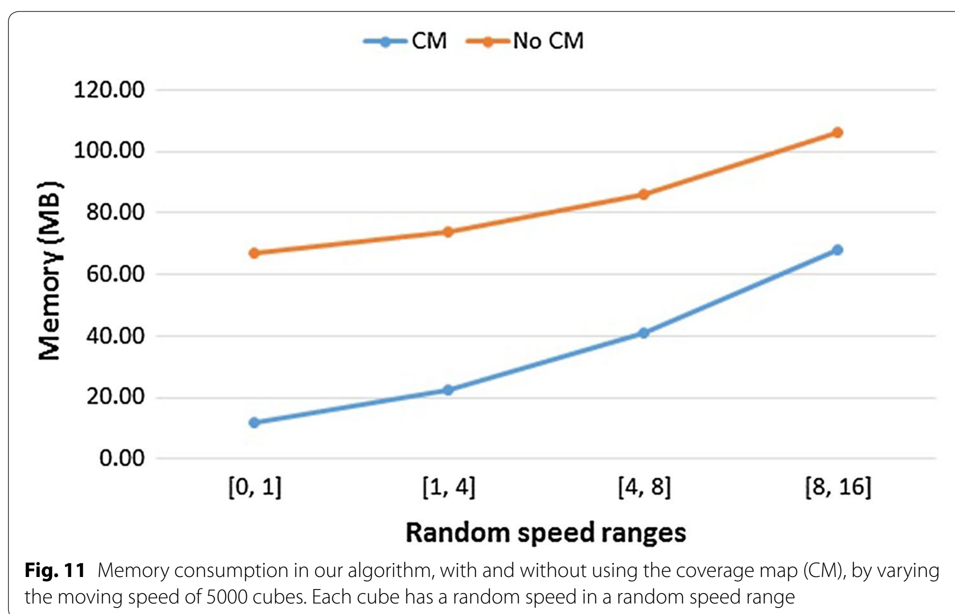

Fig. 11 Memory consumption in our algorithm, with and without using the coverage map (CM), by varying the moving speed of 5000 cubes. Each cube has a random speed in a random speed range

performance comparisons are shown from Figs. 9 to 12. Graphics memory used for the temporal shadow map in our algorithm is varied from low to high when increasing moving speed of cubes, Figs. 9 and 11. However, using the coverage map massively reduces the memory footprint while remaining the similar rendering time. The reason for this is that three geometry rendering passes in the extension take some time to generate the nearest map and the coverage map.

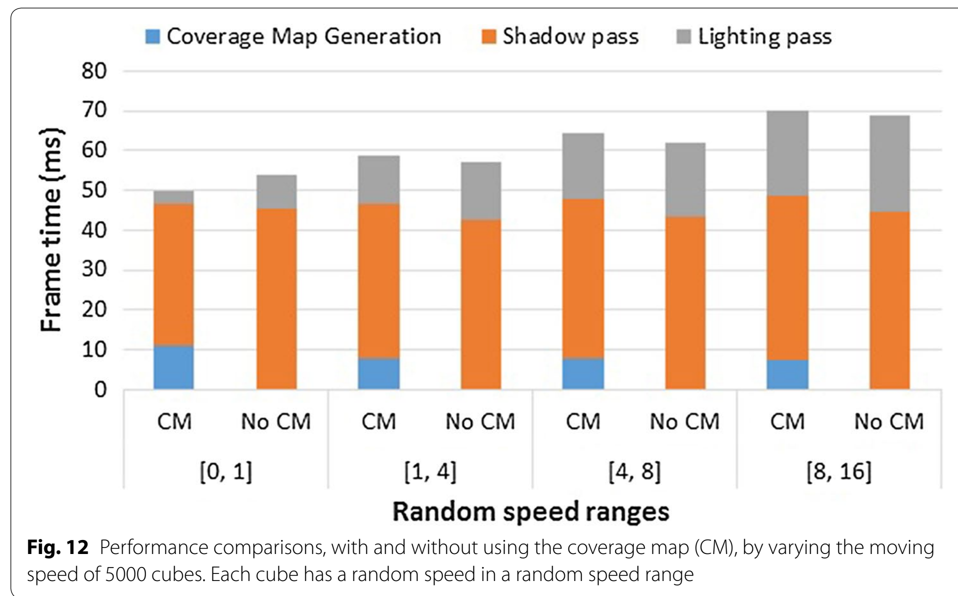

Fig. 12 Performance comparisons, with and without using the coverage map (CM), by varying the moving speed of 5000 cubes. Each cube has a random speed in a random speed range

Conclusion and future work

We have presented a hybrid algorithm that renders motion blurred shadows efficiently on GPUs using a coverage map. First, we generate the temporal shadow map which stores many time ranges at each pixel. Each time range represents a period of time that a geometry is visible to the light for a given pixel. In the second pass, we use multisampling with each sample has a random time to render motion blur and motion blurred shadows. For each visible sample, we project to the light space and then load each visible time range along with depth values to perform the shadow tests. All test results are averaged to produce the final pixel color. We not only reduce the memory footprint but also simplify the shadow tests using the coverage map.

The current implementation can be optimized using an approach described by Vasilakis and Fudos [19] to allocate memory dynamically every frame if the total number of visible time ranges changes. This approach allows to store all visible time ranges linearly in a one-dimensional array instead of a per-pixel linked list. In the future, we would like to find a method for generating the coverage map in a single rendering pass.

Abbreviations
TSM: time-dependent shadow maps; NM: the nearest map; CM: a coverage map.

Authors' contributions
The first author mainly contributes to the research. The second author suggests the idea and give discussions. Both authors read and approved the final manuscript.

Competing interests
Both authors declare that they have no competing interests.

Availability of data and materials
Background in Figs. 5 and 6 are downloaded from Computer Graphics Archive, http://graphics.cs.williams.edu/data. All data in comparisons are generated from our algorithm.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 15 March 2017 Accepted: 30 May 2017
Published online: 13 July 2017

References

1. ThanhBinh N (2015) Image contour based on context aware in complex wavelet domain. Hum-centric Comput Inf Sci 5:14
2. Agarwal J, Bedi S (2015) Implementation of hybrid image fusion technique for feature enhancement in medical diagnosis. Hum-centric Comput Inf Sci 5:14
3. Navarro F, Serón FJ, Gutierrez D (2011) Motion blur rendering: state of the art. Comput Graph Forum 30(1):3–26
4. Eisemann E, Schwarz M, Assarsson U, Wimmer M (2011) Real-time shadows. AK Peters Ltd./CRC Press, Natick
5. Haeberli P, Akeley K (1990) The accumulation buffer: hardware support for high-quality rendering. In: ACM SIGGRAPH computer graphics '90, vol 24. New York, pp 309–318
6. Lokovic T, Veach E. (2000) Deep shadow maps. In: Proceedings of SIGGRAPH, ACM, pp 385–392
7. Cook RL, Porter T, Carpenter L (1984) Distributed ray tracing. In: ACM SIGGRAPH computer graphics'84, vol 18. pp 137–145
8. Akenine-Möller T, Munkberg J, Hasselgren J (2007) Stochastic rasterization using time-continuous triangles. In: Graphics hardware, pp 7–16
9. McGuire M, Enderton E, Shirley P, Luebke D (2010) Real-time stochastic rasterization on conventional GPU architectures. In: High performance graphics, pp 173–182
10. Andersson M, Hasselgren J, Munkberg J, Akenine-Möller T (2015) Filtered stochastic shadow mapping using a layered approach. Comput Graph Forum 34(8):119–129
11. Andersson M, Hasselgren J, Akenine-Möller J (2011) Depth buffer compression for stochastic motion blur rasterization. In: High performance graphics, pp 127–134
12. Donnelly W, Lauritzen A (2006) Variance shadow maps. In: Symposium on interactive 3D graphics and games, pp 161–1659
13. Guertin JP, McGuire M, Nowrouzezahrai D (2014) A fast and stable feature-aware motion blur filter. In: High performance graphics, pp 51–60
14. Myeong-Won L, Chul-Hee J, Min-Geun L, Brutzman B (2015) Data definition of 3D character modeling and animation using H-Anim. J Converg 6(2):19–29
15. Barta P, Kovacs B, Szecsi SL, Szirmay-kalos L (2011) Order independent transparency with per-pixel linked lists. In: Proceedings of CESCG
16. Burns CA (2013) The visibility buffer: a cache-friendly approach to deferred shading. J Comput Graph Tech 2(2):55–69
17. Salvi M, Montgomery J, Lefohn A (2011) Adaptive transparency. In: High performance graphics, pp 119–126
18. Laine S, Karras T (2011) Efficient triangle coverage tests for stochastic rasterization, technical report, NVIDIA
19. Vasilakis A, Fudos I (2012) S-buffer: sparsity-aware multifragment rendering. In: Eurographics conference, pp 101–104

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com