Fast and Robust Tessellation-Based Silhouette Shadows

Tomáš Milet
Brno University of Technology
Czech Republic
imilet@fit.vutbr.cz

Jozef Kobrtek
Brno University of Technology
Czech Republic
ikobrtek@fit.vutbr.cz

Pavel Zemčík
Brno University of Technology
Czech Republic
zemcik@fit.vutbr.cz

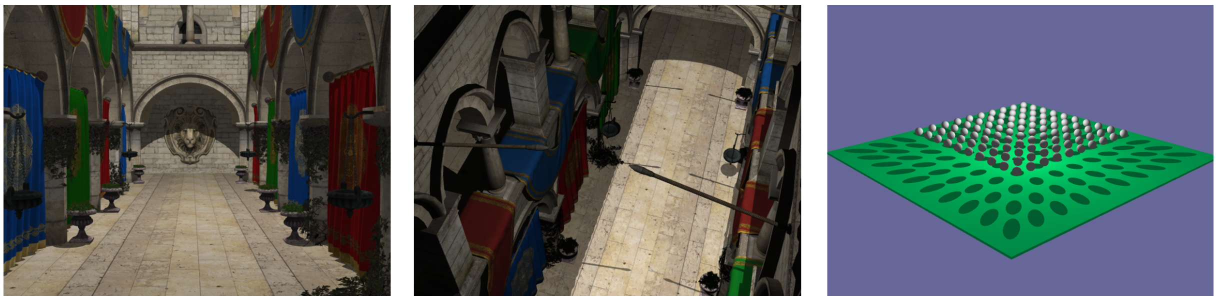
Figure 1: Test scenes - Crytek Sponza and Spheres

ABSTRACT

This paper presents a new simple, fast and robust approach in computation of per-sample precise shadows. The method uses tessellation shaders for computation of silhouettes on arbitrary triangle soup. We were able to reach robustness by our previously published algorithm using deterministic shadow volume computation. We also propose a new simplification of the silhouette computation by introducing reference edge testing. Our new method was compared with other methods and evaluated on multiple hardware platforms and different scenes, providing better performance than current state-of-the art algorithms. Finally, conclusions are drawn and the future work is outlined.

Keywords: shadows, shadow volumes, silhouette, tessellation shaders, geometry shader

1 INTRODUCTION

Shadow Volumes (SV) algorithm was introduced in 1977 by [Crow, 1977], first implementation using hardware support via stencil buffer was carried out by [Heidmann, 1991]. Heidman's implementation is generally called z-pass, but does not produce correct results when observer is in shadow. This problem was eliminated in the z-fail method [Bilodeau and Songy, 1999; Everitt and Kilgard, 2002], which reverses depth test function, but requires shadow volumes to be capped.

Shadow Mapping (SM) algorithm, proposed by [Williams, 1978], is an alternative approach to shadow volumes. It uses depth information from light source stored in a texture. Shadow mapping is nowadays massively used in games thanks to its performance. Due to discrete nature of shadow maps, this technique suffers from spatial and often temporal aliasing problems and produces imperfect shadows because of limited shadow map resolution [Donnelly and Lauritzen, 2006]. Low resolution is not an issue for games, because scenes can be adjusted so that visual artifacts are suppressed or a filtering method is applied, but applications for visualization in architecture or industrial design require pixel-correct shadows for object visualization, thus shadow mapping is not suitable for such applications. Per-sample precise alias-free shadow maps (AFSM) algorithm was proposed by [Sintorn et al., 2008]. Their method stores multiple samples into a list for each shadow map pixel and conservatively rasterizes triangles into shadow map using CUDA. Individual samples stored in lists are then tested against shadow volume of the triangle. As they stated, their per-sample precise method is three times slower than standard shadow mapping with resolution of 8096 by 8096 texels.

While producing per-sample correct shadows, SV are affected by performance issues. In its naive form, when a volume is generated for every triangle in the scene, resulting performance is very low due to rasterization of a large amount of triangles. More efficient way is to construct shadow volumes only from silhouette edges of the occluding geometry, which has positive impact on fill-rate. Silhouette extraction on CPU was first published by [Brabec and Seidel, 2003].

Several silhouette-based methods were published since then, utilising novel hardware features to speed up silhouette calculation. [McGuire et al., 2003] managed to implement the algorithm in vertex shader. [van Waveren, 2005] described CPU implementation of silhouette extraction using SSE2 instruction set in Doom3 game and [Stich et al., 2007] used geometry shaders.

Most of the methods mentioned above are not completely robust and also cannot handle non 2-manifold casters. [Kim et al., 2008] proposed an algorithm for non 2-manifold casters, but unfortunately it is not completely robust. We managed to improve Kim's algorithm in [Pečiva et al., 2013] using deterministic multiplicity calculation, which we further simplified in this paper.

2 METHOD DESCRIPTION

We developed three methods - two per-triangle approaches and robust silhouette method.

Our silhouette method is based on the work of Kim et al. [2008]. This algorithm calculates so-called *multiplicity* of an edge - light plane from light source through the edge is casted and all opposing vertices are tested, if they are above or bellow the plane. According to result of the test, multiplicity is incremented or decremented. Absolute value of multiplicity is the number of times an infinite quad needs to be drawn from this edge.

2.1 Per-Triangle Methods

These methods require no pre-processing and work with arbitrary triangle soup. In the first variant, input patch has 3 points, which are original points of the triangle. Tessellation factors are 3 (inner) and 1 (outer, for all sides), equal spacing and reversed triangle winding. The resulting patch can be seen in the picture 2b.

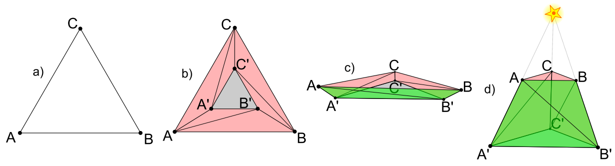

Figure 2: Creating semi-enclosed shadow volume from a triangle. Initial triangle in $a)$ is tessellated using outer factors (1, 1, 1) and inner (3) $b)$. Points A', B', C' are given positions of points A, B, C $c)$ and then pushed to infinity to form a volume with back cap $d)$. Front cap is absent and must be rendered in second pass.

We construct a simple volume in evaluation shader as in Algorithm 1. Basically, points A', B' and C' are moved to positions of A, B, C and then pushed to infinity, forming a back cap. Vertex ordering must be reversed in order the sides face outwards. In the second pass, light caps are rendered in order to close the volumes for z-fail.

```
Data: original points $\mathbf{P}[3]$, light position $\mathbf{L}$, tessellation
      coordinates $\mathbf{T} = (x,y,z)$, $x,y,z \in \langle 0,1 \rangle$
Result: world-space coordinates $X$
$c = x \cdot y \cdot z$;
if $c == 0$ then
    $\mathbf{X} = \mathbf{P}[0] \cdot x + \mathbf{P}[1] \cdot y + \mathbf{P}[2] \cdot z$;
    $\mathbf{X}_w = 1$;
else
    $i = getIndexOfLargestVectorElement(\mathbf{T})$;
    $\mathbf{X} = l_w \cdot \mathbf{P}[i] - \mathbf{L}$;
    $\mathbf{X}_w = 0$;
end
```
Algorithm 1: Evaluation shader in two-pass per-triangle method

We also designed a single-pass version for z-fail. This method takes a triangle as an input, but adds one more point to form a quad (four control shader invocations per patch). This quad is then tessellated using outer factors (1, 5, 1, 5), inner (5,1) and fractional odd spacing, resulting in a shape seen in Fig. 3b.

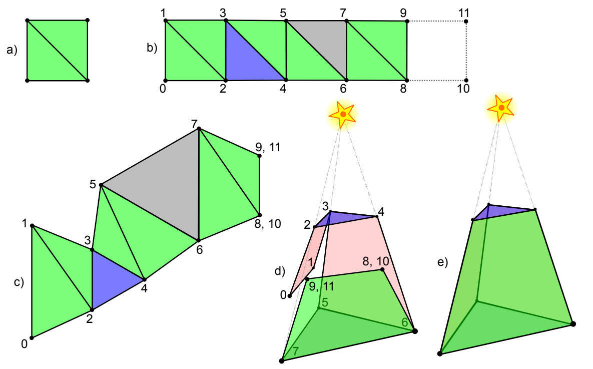

Figure 3: Single-pass per triangle method, a full shadow volume is created in a single pass. One point is added to the triangle in order to form a quad $a)$ which is then tessellated using factors (1, 5, 1, 5),(5, 1) $b)$. Points 10 and 11 are merged with 8, 9. Light cap is visualized as blue, dark cap grey $c)$. Then we join points 0-7, 1-5, 2-9, 4-8 and push points 5, 6, 7 to infinity $d)$ to make the volume $e)$.

Evaluation shader then twists the shape in order to create a volume, note Figure 3.

2.2 Silhouette Method

The method finds silhouette edges by looping over every edge in the model. Each edge is processed in parallel in Tessellation Control Shader where multiplicity is computed. An input patch primitive is composed of two vertices that describe an edge, one integer that contains number of opposite vertices and n opposite vertices, see Figure 4. Because patch the size must be constant, some positions are not used.

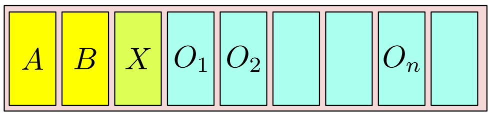

Figure 4: Input patch for tessellation control shader

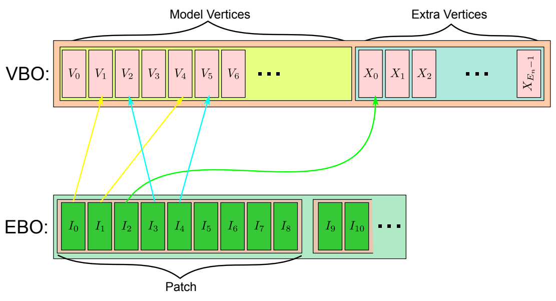

Figure 5: Combining per-edge patch data using Element Buffer Object. Vertex Buffer Object only needs to be extended by E_n (number of edges) vertices, where X_i is number of adjacent triangles to edge E_i. Because input vertices are 4-component vectors and X_i is scalar value, only a single value per vector is used.

A vertex buffer of model has to be extended by E_n vertices, which is the number of edges in the model. We used element buffer to reduce memory requirements, as can be seen in Fig. 5.

Byungmoon's algorithm [Kim et al., 2008], as in its core proposal, has a flaw that multiplicity is not calculated in a deterministic way. In older approach [Pečiva et al., 2013], it was fix this by calculating multiplicity per triangle and if the 3 results troughout all 3 edges were not consistent, we discarded the triangle from further processing, because it meant that the triangle is almost parallel to the light and does not cast a shadow. We further improved this approach - multiplicity is now computed only once for each opposite vertex using *reference edge*.

A choice of reference edge has to be the same for all occurrences of a triangle. This can be achieved for example by introducing vertex ordering - Equations 1 and Algorithm 2.

$$\begin{aligned}
\mathbf{A} < \mathbf{B} &\Leftrightarrow \text{Greater}(\mathbf{A},\mathbf{B}) < 0 \\
\mathbf{A} = \mathbf{B} &\Leftrightarrow \text{Greater}(\mathbf{A},\mathbf{B}) = 0 \\
\mathbf{A} > \mathbf{B} &\Leftrightarrow \text{Greater}(\mathbf{A},\mathbf{B}) > 0 \qquad (1)
\end{aligned}$$

In order to guarantee consistency, reference edge of a triangle in our algorithm is constructed using smallest and larges vertex of a triangle, as in Algorithm 2. More options for such method are available, but as mentioned above, evaluation per each triangle occurance must be consistent in order to get correct results.

```
Data: Vertices $\mathbf{A},\mathbf{B}$
Result: Result $r$ of comparison
$\mathbf{S} = \text{sgn}(\mathbf{A}-\mathbf{B})$;
$\mathbf{K} = (4,2,1)$;
$r = \mathbf{S}\cdot\mathbf{K}$
```

Algorithm 2: Function Greater($\mathbf{A},\mathbf{B}$) used for vertex ordering.

To simulate behaviour of Byungmoon's algorithm (edge casts a quad as many times as it has multiplicity), we tessellate the casted quad from the edge 6using inner tessellation levels $(Multiplicity \cdot 2 - 1, 1)$ and then we bend the tessellated quad in evaluation shader in a way to create m overlapping quads, as seen in Fig. 6, which demonstrates edge A-B having multiplicity of 3.

The procedure of multiplicity calculation is described in Algorithm 3 and 4.

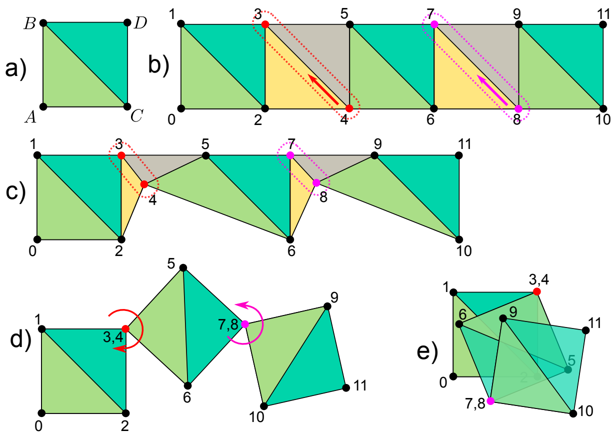

Figure 6: The image shows the transformation of a quad into three overlapping shadow volume sides. The transition from part a) to part b) is tessellation of quad with Multiplicity = 3. Only green and blue triangles will be drawn. Yellow and gray triangles will be degenerated. The transition from part b) over part c) to part d) shows degeneration process. Red and purple vertices 3, 4 and 7, 8 from part a) form only one vertex in part d). The transition from part d) to part e) shows rotation around red and purple vertices. This transformation creates three overlapping sides of shadow volume. Positions of vertices A, B, C, D that form initial quad, can be computed according to equations 2.

After tessellation, we have to transform tessellation coordinates into vertex position of the shadow volume quad in the evaluation shader. The algorithm for its implementation is described in Algorithm 5 and Equations 2.

```
Data: Edge $\mathbf{A},\mathbf{B}$, $\mathbf{A} < \mathbf{B}$, set $\mathfrak{O}$ of opposite vertices
      $\mathbf{O}_i \in \mathfrak{O}$, light position $\mathbf{L}$ in homogeneous
      coordinates
Result: Multiplicity $m$
$m = 0$;
for $\mathbf{O}_i \in \mathfrak{O}$ do
    if $\mathbf{A} > \mathbf{O}_i$ then
        $m = m + CompMultiplicity(\mathbf{O}_i, \mathbf{A}, \mathbf{B}, \mathbf{L})$;
    else
        if $\mathbf{B} > \mathbf{O}_i$ then
            $m = m - CompMultiplicity(\mathbf{A}, \mathbf{O}_i, \mathbf{B}, \mathbf{L})$;
        else
            $m = m + CompMultiplicity(\mathbf{A}, \mathbf{B}, \mathbf{O}_i, \mathbf{L})$;
        end
    end
end
```

Algorithm 3: Modified algorithm for computation of final multiplicity of edge $\mathbf{A},\mathbf{B}$

```
Data: Vertices $\mathbf{A},\mathbf{B},\mathbf{C}$; $\mathbf{A} < \mathbf{B} < \mathbf{C}$; light position $\mathbf{L}$ in
      homogeneous coordinates
Result: Multiplicity $m$ for one opposite vertex
$\mathbf{X} = C - A$;
$\mathbf{Y} = (l_x - a_x l_w, l_y - a_y l_w, l_z - a_z l_w)$;
$\mathbf{N} = \mathbf{X} \times \mathbf{Y}$;
$m = \operatorname{sgn}(\mathbf{N} \cdot (\mathbf{B} - \mathbf{A}))$;
```

Algorithm 4: $CompMultiplicity(\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{L})$ function used in algorithm 3

$$\begin{aligned}
\mathbf{A} &= (a_x, a_y, a_z, 1)^T \\
\mathbf{B} &= (b_x, b_y, b_z, 1)^T \\
\mathbf{C} &= (a_x - l_x, a_y - l_y, a_z - l_z, 0)^T \\
\mathbf{D} &= (b_x - l_x, b_y - l_y, b_z - l_z, 0)^T \qquad (2)
\end{aligned}$$

Because caps are not generated, this method can also be used with simpler z-pass algorithm.

2.3 Implementation

All our methods were implemented in Lexolights open-source multi-platform program based on OpenScene-Graph and Delta3D, using OpenGL.

Single-pass per-triangle method suffers from inconsistent rasterization of two identical triangles at the same depth but with different winding - depth of fragments from both triangles differs, which resulted in z-fighting artifacts. We had to manually push the front cap's fragments into depth of 1.0f, so they would fail the depth test, otherwise we observed self-shadowing artifacts. Bypassing early depth test in rasterization due to assigning depth values in fragment shader causes performance loss over two-pass method. On the other hand,

```
Data: Vertices $\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}$, tessellation coordinates
      $x, y \in \langle 0, 1 \rangle$ and multiplicity $m$
Result: Vertex $\mathbf{V}$ in world-space
$\mathbf{P}_0 = \mathbf{A}$;
$\mathbf{P}_1 = \mathbf{B}$;
$\mathbf{P}_2 = \mathbf{C}$;
$\mathbf{P}_3 = \mathbf{D}$;
$a = \text{round}(x \cdot m)$;
$b = \text{round}(y)$;
$id = a \cdot 2 + b$;
$t = (id \mod 2)\,\hat{}\,(\lfloor id/4 \rfloor \mod 2)$;
$l = \lfloor (id + 2)/4 \rfloor \mod 2$;
$n = t + l \cdot 2$;
$\mathbf{V} = \mathbf{P}_n$;
```

Algorithm 5: This algorithm transforms tessellation coordinates into the vertex of side of shadow volume. Vertices $\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}$ are computed using Equation 2.

this method served as basis for silhouette-based approach.

For caps generation in silhouette-based method, we used gemetry shader and multiplicity calculation, using which we calculated triangle's orientation towards light source via reference edge. It was also necessary for keeping discarding calculations consistent throughout the rendering process of shadow volumes.

Because tessellation factors are limited, at the time of writing, to 64, there is also a limit of maximum multiplicity per edge that this algorithm is able to process. Acording to equation to calculate tessellation factor $Multiplicity \cdot 2 - 1$, maximum multiplicity of an egde is 32, which should be more than enough for majority of models. But for example well-known Power Plant model (12M triangles) has some edges, which have multiplicity of 128. In that case, they would have to be splitted into more edges.

3 EXPERIMENTS

We compared our methods against already available shadow volumes implementations on modern hardware - robust geometry shader implementation and standard shadow mapping, using which we also tried to evaluate performance against Sintorn's AFSM [Sintorn et al., 2008]. We also tested two-pass per-triangle method against similar geometry shader implementation. For shadow volumes approaches, z-fail was used; shadow map resolution was set to 8k x 8k texels.

Testing platform had following configuration: Intel Xeon E3-1230V3, 3.3 GHz; 16GiB DDR3; GPUs: AMD Radeon R9 280X 3 GiB GDDR5, nVidia GeForce GTX 680 2 GiB GDDR5; Windows 7 x64; driver version: 13.12 (AMD), 334.89 (nVidia).

Sponza	R280				G680			
Resolution	TS		GS		TS		GS	
800x600	112	130	51	49	175	178	62	61
1024x768	116	124	49	50	177	178	61	60
1366x768	107	116	48	48	159	160	60	61
1920x1080	95	101	47	46	122	126	60	59

Table 1: Performance of two deterministic methods measured in FPS on Sponza scene at different resolutions. First column for each method denotes original algorithm (3 edges) and second value denotes our new method using a single reference edge.

3.1 Testing Scenes

We created a camera fly-through in two testing scenes, each having one point light source. The scenes can be seen in Fig. 1

- Sphere scene: synthetic scene containing adjustable number of spheres (typically 100) with configurable amount of detailness. Fly-through has 16 seconds.
- Crytek Sponza: popular model used to evaluate computer graphics algorithms. 262 267 triangles, 40 seconds.

3.2 Results

In the first set of our tests we used Crytek Sponza scene (first and second part of Fig. 1), being a popular model to evaluate performance of computer graphics algorithms. This scene has fixed amount of triangles (262K), so we evaluated our method in different resolutions, compared it to the geometry shader implementation and also both implementations with older variant of deterministic multiplicity calculation. as seen in Table 1 and Figure 7. Geometry shader implementation provides more stable frames per second (FPS) with change of resolution, but fails to outperform our new tessellation algorithm on both graphic cards. Moreover, our new determinism method is faster in majority of cases, with some exceptions in geometry shader implementation, where it is on-par with older type of determinism, using all 3 edges.

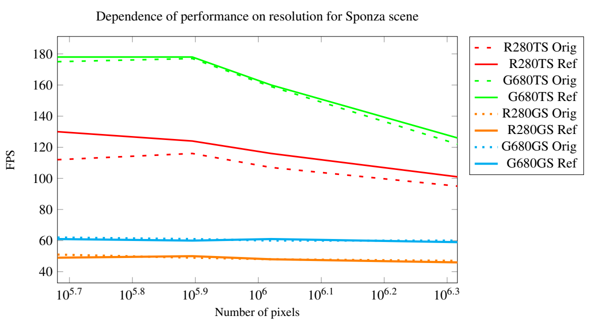

Figure 7: Dependence of performance (FPS) on resolution of original and new method, measured on Sponza scene.

Sponza	R280				G680			
Variant	TS		GS		TS		GS	
tex	121	130	48	49	176	185	60	62
notex	112	123	51	52	197	219	64	63

Table 2: Performance of two deterministic methods measured in FPS on Sponza scene, but without textures. First column for each method denotes original algorithm (3 edges) and second value denotes our new method using a single reference edge.

Spheres10x10	R280				G680			
Triangles	TS		GS		TS		GS	
32400	984	**995**	490	484	739	**825**	542	540
67600	921	**963**	488	487	624	**667**	494	513
102400	615	**729**	484	479	491	**555**	372	402
360000	203	233	270	**272**	218	**228**	131	135
1081600	72	88	104	**110**	82	**94**	46	49
1440000	56	72	84	**91**	67	**81**	36	39
1960000	34	41	59	**62**	49	**58**	26	28

Table 3: Performance of two determinism methods measured in FPS on a scene with 10x10 spheres at different triangle count.

We also tested the same scene without textures in resolution of 800x600 to speed up fragment processing, so more processors on the GPU are assigned to geometry/tessellation shader executions. Radeon R9 280X, when using tessellation, surprised us with higher performance when using textured model. Currently, we have no explanation for this behaviour. Apart from that, geometry shader methods benefited only slightly from lack of texturing, but tessellation on GeForce GTX680 was speeded up by up to 18%.

Majority of our tests was performed on a scene with spheres (Fig. 1 on the right), in which we can control the amount of geometry. First, we made a flythrough in a scene containing 100 spheres with different amount of triangles per scene, the results can be seen in Table 3 and graph in Fig. 8.

On GTX680, tessellation using reference edge is the fastest, no matter the number of triangles, although the performance gaps gets smaller with increasing number of triangles in scene. The situation was different on R9 280X - tessellation was more than 2x faster when the scene contained only 32K triangles but at approximately 300K, geometry shader method took lead.

We also tested a scene with only 1 box (12 triangles) and a sphere, triangle amount of which we increased in steps, results can be seen in the Table 4 and Figure 9. The reults have the same characteristic as those from a scene having 100 spheres - tessellation is dominant on GTX680, whereas on Radeon R9 280X it is geometry shader. In previous measurement, geometry shader became dominant at about 3500 triangles per sphere, which also observable in this measurement, so both cards keep their performance tendency, no matter the object count.

We further extended this test to performance dependency on number of objects in a scene while maintainig

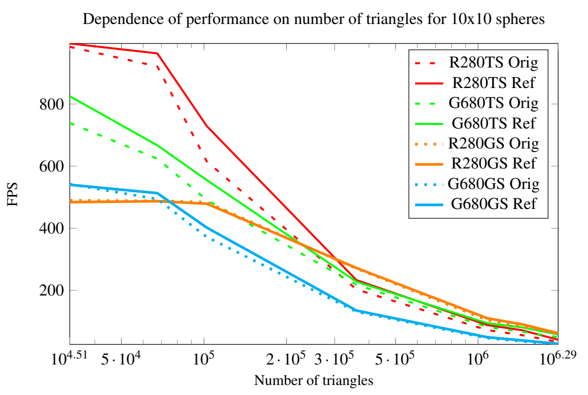

Figure 8: Dependence of performance (FPS) on number of triangles on a scene with 10x10 spheres using original and new deterministic method.

Spheres1x1	R280				G680			
Triangles	TS		GS		TS		GS	
336	3000	**3010**	2830	2750	2940	**2990**	2780	2780
1308	2950	**2980**	2730	2700	2830	**2860**	2650	2630
15612	1820	2050	2540	**2650**	1960	**1980**	1460	1570
102412	613	741	945	**995**	822	**992**	465	503
360000	196	245	299	**323**	281	**353**	149	163
577600	122	155	186	**203**	180	**224**	94	103
1000000	73	92	111	**119**	106	**134**	55	60
1960000	37	46	56	**60**	54	**68**	28	31
3686400	19	25	30	**32**	29	**36**	0.5	0.5

Table 4: Performance of two deterministic methods measured in FPS on a scene with 1 sphere having different amount of triangles. First value denotes original method and second value denotes our new method. On GTX 680, GS and 3.6M triangles, the GPU ran out of it's memory and performance dropped to 0.5 FPS.

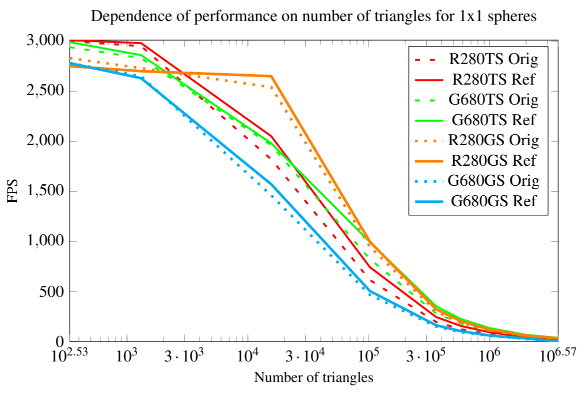

Figure 9: Dependence of performance on number of triangles for one tessellated sphere for original and new method.

constant amount of geometry. This measurement was carried out on Sphere scene, having 1 million triangles (with deviation max 2%) in every case. No hardware instancing was used, every object was drawn via separate draw call. Results can be seen in Table 5 and graph in Figure 10.

This test was biased by CPU overhead of draw calls. Contrary to previous measurements, tessellation was faster on R9 280X, starting from 10^3 objects, although

Spheres 1M	R280				G680			
Objects	TS		GS		TS		GS	
1	73	92	111	**120**	106	**134**	55	60
4	74	94	113	**121**	101	**126**	53	58
25	64	76	97	**101**	76	**88**	46	50
64	68	76	**90**	89	61	**66**	40	43
100	64	70	**84**	82	58	**62**	39	42
240	58	55	**70**	64	**50**	49	35	36
399	53	48	**61**	54	**36**	36	27	28
625	43	38	**53**	46	**29**	27	22	22
851	40	44	46	**50**	24	**25**	19	20
1250	35	**37**	28	31	**19**	19	16	16
2500	**23**	19	15.1	15.4	**12**	11	10.8	10.1
3116	21.2	**21.5**	12.8	12.5	11.1	**11.2**	9.1	9.2
3920	**15.7**	14	10.1	10.12	**9.1**	8.7	7.7	7.5
5100	**14.8**	14.2	7.8	7.75	**8.2**	8.2	6.7	6.8
15600	**7.45**	6.45	3.07	3.14	**10.5**	9.1	3.6	3.6

Table 5: Dependence on number of objects for spheres scene with 1M triangles. Bold values represent the fastest algorithm/implementation for respective number of objects, per GPU.

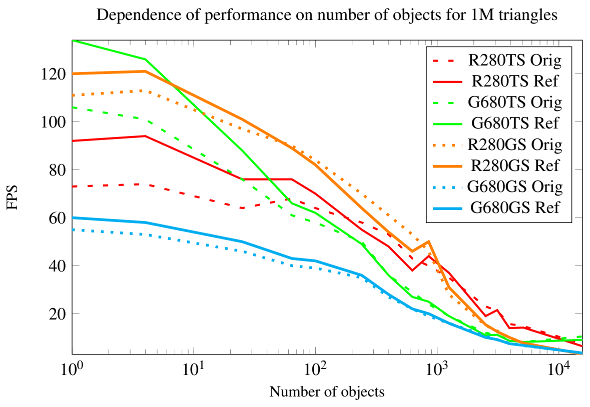

Figure 10: Dependence on number of objects for spheres scene with 1M triangles.

Spheres10x10	R280		G680	
Triangles	TS	SM	TS	SM
32400	**995**	252	**825**	245
67600	**963**	250	**667**	237
102400	**729**	244	**555**	225
360000	**233**	219	**228**	190
1081600	88	**168**	94	**135**
1440000	72	**155**	81	**115**
1960000	41	**120**	58	**103**

Table 6: Shadow Mapping vs Tessellation Silhouettes, 10x10 sphere scene, FPS

reference edge was faster only in 40% cases. Moreover, as can be seen in Fig. 10, there is a slight increase in FPS in both geometry shader and tessellation implementations at about 1000 objects on Radeon. On GTX680, tessellation method was faster in every case; eferenge edge provided increased performance only in a half of measurements, but in all other cases the difference was only 1-3 FPS.

Sintorn in his AFSM paper Sintorn et al. [2008] stated that his per-pixel precise shadow maps are 3-times slower than standard 8Kx8K shadow mapping. In order to evaluate our algorithm against AFSM, we conducted a measurement against shadow mapping having resolution metioned above, results of which are in table 6 and graph 11.

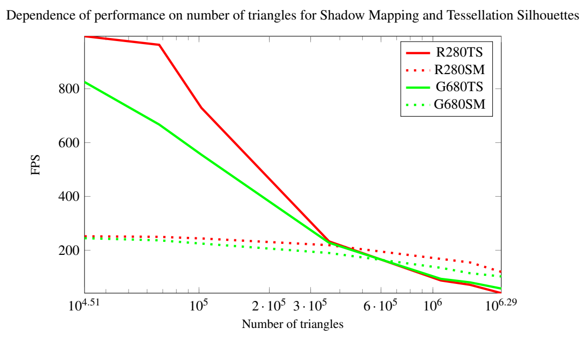

Figure 11: Shadow Mapping vs Tessellation Silhouettes on a scene with 10x10 spheres, measured in frames per second.

Sponza	R280				G680			
Variant	TS		GS		TS		GS	
Spheres1x1	1.24	1.3	1.36	1.4	1.19	1.25	1.13	1.11
Spheres50x50	1.65	1.98	1.77	2.07	3.81	5.04	3.16	3.35

Table 7: Performance speed up for non-visible shadow volumes. First column denotes original method and second one our new method using reference edge.

Not only we managed to outperform shadow mapping with triangle count up to ~400K triangles, but at almost 2M triangles our method was on par or faster than AFSM - R9 280X dropped to 34% of SM performance whereas GTX680 was only 44% slower than 8K shadow mapping.

Throughout the test we noticed different behaviour on both cards when dealing with camera position, where the scene is not visible. Despite the fact that OpenSceneGraph, on which our testing program was based, culls non-visible geometry, shadow volumes were not culled from rendering process and we noticed that both cards deal with such situation differently, as can be seen from Table 7. The values are ratios between views when whole scene is visible and when there is no scene geometry present. The same view location and direction was used in all respective cases.

GTX680 was more effective when it comes to discarding non-visible shadow volume geometry from rasterization, with speedup up to 5-times compared to a full scene view, whereas R9 280X was only 2x faster.

We also compared silhouette methods with two-pass per-triangle tessellation implementation and 8K shadow mapping (only on sphere scene, our framework does not support omnidirectional shadow mapping), results in table 8 and graph 12.

One can observe that per triangle tessellation method is even faster than both geometry shader methods running on Sponza scene. It is also worth noting that per-triangle geometry-shader-based method provides more performance on this scene than silhouette-based approach. On GTX680, the difference between silhouette and per-triangle tessellation method is 122%, whereas on R9 280X card it is only faster by 27%.

	R280		G680	
Method	Spheres	Sponza	Spheres	Sponza
TS Triangle	5.8	102	7.9	83
TS Silhouette	23.7	130	32	185
GS Triangle	3.1	51	4.9	73
GS Silhouette	34	49	14.8	62
SM	93	0	74	0

Table 8: Overall comparison of GS, TS methods and classic 8K shadow mapping on testing scenes - Sponza, and Spheres with 4M triangles. Shadow mapping was not evaluated on Sponza scene (zeros).

Spheres 100k	R280				G680			
Objects	TS		GS		TS		GS	
1	630	765	962	**1025**	825	**1003**	470	510
4	561	657	881	**920**	660	**742**	410	441
25	534	583	630	**645**	445	**453**	340	352
64	443	448	**485**	475	**297**	289	273	273
100	485	**531**	403	408	234	**237**	211	212
225	**307**	258	199	203	**147**	145	124	125
900	99	**103**	48	49	**61**	61	45.5	45.7
2500	34	**36**	16	17	28.5	**28.6**	17	16.8
3600	24	**25**	11	12	**20.3**	20	11.8	11.7

Table 9: Dependence on number of objects for spheres scene with 100k triangles.

With increased amount of geometry in our synthetic test scene, the situation turns around in favor to silhouette methods. Also performance difference between shadow mapping and tessellation on Radeon drops under 1/3 ratio, but GeForce is still able to maintain 43% of SM performance.

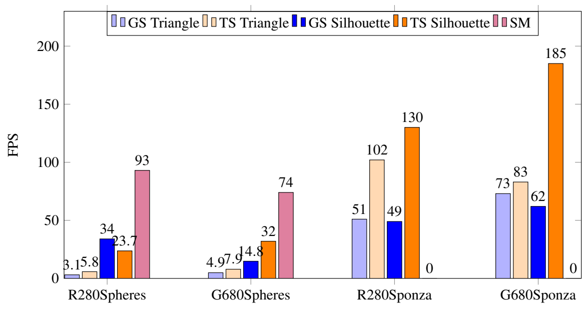

Figure 12: Overall comparison of methods on testing scenes - Sponza and Spheres with 4M triangles.

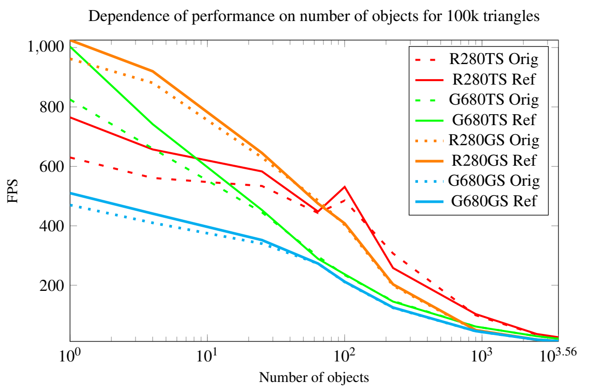

Figure 13: Dependence on number of objects for spheres scene with 100k triangles.

4 CONCLUSIONS

We have developed new methods for computing shadow volume silhouettes using tessellation shaders.

Our two-pass per-triangle tessellation method is, in some cases, faster than silhouette algorithm implemented in geometry shader, but loses performance as geometry amount in the scene grows. Compared to geometry shader per-triangle implementation, the tessellation method proved to be faster in all our tests, no matter the amount of scene complexity.

The silhouette method is more efficient, and as we have proven in our measurements, mostly in scenes with higher amount of geometry. GeForce GTX680 benefited mostly from this algorithm, being faster than geometry shader silhouette method in every case. As for Radeon R9 280X based on GCN architecture, geometry shader method is more suitable. Tessellation method on Radeon proved to be faster in Sponza scene, but our synthetic tests on scene with configurable amount of spheres and level of detail showed that it's performance is dominant only up to ~300K of triangles when having multiple objects in the scene, or only up to 15K triangles when only a single detailed object was drawn. In less detailed scenes it was able to outperform nVidia-based card, but only up to aforementioned 300K triangles.

We also tested hardware's culling capabilities. R9 280X does not cull non-visible volumes so efficiently as it's counterpart, being able to speed up rendering of virtually empty scene by only 2-times max, whereas GTX680was able to gain 5-times higher frame rate.

Our robust algorthm was sped up by using a novel method of multiplicity computation, which was able to provide up to 31% performance gain in tessellation method (13.5% in average), maximum speedup in geometry shader was 10.7% with average of 3.4%.

In comparison to standard SM and Sintorn's Alias-Free Shadow Maps (AFSM), our tessellation method provides better performance than 8K shadow maps up to ~400K triangles and then fall to 43% performance of shadow mapping at 4M triangles on GeForce, 34% on Radeon, which is on par or better than AFSM (it's 3-times slower than 8K SM) and is also simplier to implement.

In the future, we would like to see an arbitrary $\pm$ stencil operation in hardware, configurable in shaders, which would allow us to increase the speed of our method even more, due to a lower number of triangles being drawn. We also encountered inconsistent rasterization of two identical triangles but with different winding, on both GPUs, while experimenting with single-pass per triangle method. Fixing this issue yields a large performance penalty, although not mentioned in measurements, this method is 5-times slower than two-pass per-triangle tessellation. We also want to evaluate more hardware platforms and explore GPGPU potential in the filed of shadow volumes calculation.

ACKNOWLEDGEMENTS

The work has been made possible thanks to the co-funding by the IT4Innovations Centre of Excellence, Ministry of Education, Youth and Sports, Czech Republic, MŠMT, ED1.1.00/02.0070, V3C - Visual Computing Competence Center, Technology Agency of the Czech Republic, TAČR, TE01020415V3C, and RODOS - Transport systems development centre, Technology Agency of the Czech Republic, TAČR, TE01020155.

REFERENCES

Bilodeau, B. and Songy, M. (1999). Real time shadows. *Creativity 1999.*

Brabec, S. and Seidel, H.-P. (2003). Shadow volumes on programmable graphics hardware. *Computer Graphics Forum (Eurographics)*, 2003:433–440.

Crow, F. C. (1977). Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '77, pages 242–248, New York, NY, USA. ACM.

Donnelly, W. and Lauritzen, A. (2006). Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 161–165. ACM.

Everitt, C. and Kilgard, M. J. (2002). Practical and robust stenciled shadow volumes for hardware-accelerated rendering.

Heidmann, T. (1991). Real shadow real time. pages 28–31. IRIS Universe.

Kim, B., Kim, K., and Turk, G. (2008). A shadow-volume algorithm for opaque and transparent nonmanifold casters. *J. Graphics Tools*, 13(3):1–14.

McGuire, M., Hughes, J. F., Egan, K., Kilgard, M., and Everitt, C. (2003). Fast, practical and robust shadows. Technical report, NVIDIA Corporation, Austin, TX.

Pečiva, J., Starka, T., Milet, T., Kobrtek, J., and Zemčík, P. (2013). Robust silhouette shadow volumes on contemporary hardware. In *Conference Proceedings of GraphiCon'2013*, pages 56–59. GraphiCon Scientific Society.

Sintorn, E., Eisemann, E., and Assarsson, U. (2008). Sample based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering 2008)*, 27(4):1285–1292.

Stich, M., Wächter, C., and Keller, A. (2007). Efficient and robust shadow volumes using hierarchical occlusion culling and geometry shaders. In Nguyen, H., editor, *GPU Gems 3*, pages 239–256. Addison Wesley Professional.

van Waveren, J. (2005). Shadow volume construction.

Williams, L. (1978). Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274.