Robust Silhouette Shadow Volumes on Contemporary Hardware

Jan Pečiva, Tomáš Starka, Tomáš Milet, Jozef Kobrtek, Pavel Zemčík

Faculty of Information Technology Brno University of Technology Czech Republic *

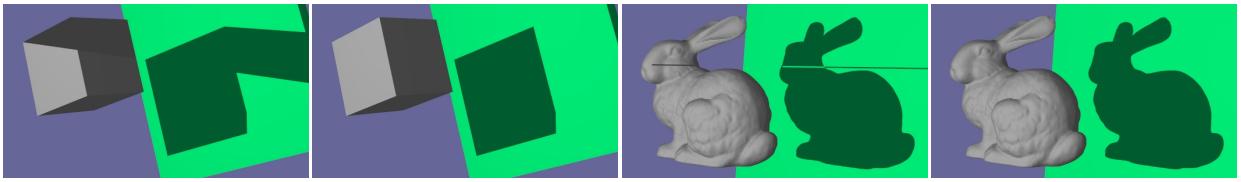

Figure 1: *The Figure shows the difference between the original algorithm and our robust algorithm. The right image of each couple shows result of our robust algorithm. The first couple of images shows very simple model, where artefacts are most visible. The second couple shows artefacts on more complex model, which could appear in real applications.*

Abstract

The paper describes an algorithm, which produces shadow volumes for an arbitrary triangle model without visual artifacts. The algorithm has been implemented, optimized, and evaluated for a number of contemporary hardware platforms. The main contribution of the paper is removal of visual artifacts caused by limited precision of floating point arithmetics. The paper also presents an overview of the implementation and result of the optimizations on individual platforms. Finally, the conclusions are drawn and the future work is outlined.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Radiosity;

Keywords: shadow volumes, silhouette, OpenCL, GPGPU, geometry shader

1 Introduction

Shadow volumes (SV) method is a traditional and popular method for shadow casting in computer graphics. It has been introduced in 1977 [Crow 1977]. Later on, in 1991, SV algorithm was implemented in stencil buffer hardware [Heidmann 1991]. This implementation is generally called z-pass method. This method, however, is not robust. It produces incorrect shadows when the observer is inside a SV itself. This problem was addressed in 2002 through a method called z-fail [Everitt and Kilgard 2002].

An alternative for rendering shadows is shadow mapping [Williams 1978]. It is very frequently used as it generally offers high performance; however, the shadow maps approach suffers from visual imperfections caused by the limited shadow map resolution. The shadow map approach is massively used in game industry where high performance is critical and scenes can be adjusted so that visual artefacts are not too visible or do not occur. On the other hand, these features limit the applicability of the approach in e.g. CAD systems, where the scene is given by the model being constructed and may be incompatible with the requirements of the shadow maps approach to work well.

While the SV approach produces per-pixel correct results, they are affected by some performance issues. In its simple form, the applications were using a SV generated for each triangle of the object geometry, which could be very rasterization demanding. This fact lead to development of more sophisticated methods, which construct a SV only from the possible silhouette edges of the occluder geometry, that has positive impact on fill-rate. The first one to propose an algorithm of silhouette extractions using the consumer graphics hardware was [Brabec and Seidel 2003]. Later on, [McGuire et al. 2003] managed to implement the algorithm solely using vertex shaders. Furthermore, [van Waveren 2005] extracted the silhouette edges using the SSE2 instructions on CPU. Finally in [Stich et al. 2007] the geometry shaders were used for silhouette extraction.

Many of the of these algorithms expect 2-manifold objects as their input. More general objects were considered by [Bergeron 1986], who focused on manifold objects with a boundary edge case. [Aldridge and Woods 2004] further removed constraints on the input model but oriented non-manifold meshes are expected. Finally, [Kim et al. 2008] presented the algorithm that works with any non-manifold mesh objects. The overview of above methods can be found in [Kolivand and Sunar 2013]

Unfortunately, during the implementation of the algorithm presented in [Kim et al. 2008] we found out that the algorithm is not completely robust and often produces result with visual artefacts. This is caused by limited precision of floating point arithmetics. Thus, we developed the proposed robust algorithm, based on [Kim et al. 2008], that is free from visual artifacts. Additionally, we optimized the algorithm for a number of contemporary hardware platforms, such as modern CPUs and GPUs. The paper describes the algorithm, presents an overview of the implementation of the proposed improvements and optimizations and assesses performance. Finally, the conclusions are drawn and the future work is outlined.

2 Algorithm description and Robustness improvement

The algorithm, described in [Kim et al. 2008], generates the output shadow silhouette based on the triangular mesh representing the model and the position of light source.

2.1 Description of the algorithm

The algorithm can be briefly described as follows:

Input: model represented as a triangular mesh and light source position.

*e-mail:{pecita,starka,imilet,ikobrtek,zemcik}@fit.vutbr.cz

Output: silhouette represented by a set of edges selected from the input model.
Algorithm:

1. The triangular model is converted into an edge representation. Every edge occurs only once even though it is shared among more triangles. Each edge in the new representation is described by its vertices and list of all opposite vertices (OVs). The OVs are vertices of the triangles sharing the edge that do not belong to the Edge. See Figure 2.
2. For each edge from the edge set, an oriented light plane (LP) is evaluated from edge vertices and the position of the light source.
3. For each OV belonging to the edge, multiplicity is calculated as +1 or -1 depending on which side of LP the OV lies. If the OV lies exactly on the LP, its multiplicity is 0. The final multiplicity of an edge is given by the sum over the multiplicities of every OV. See Figure 2.
4. Finally, the set of edges forming the silhouette is a subset of all the edges such that their multiplicity is not 0.

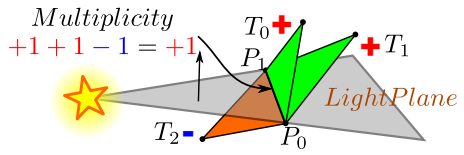

Figure 2: *Multiplicity of Edge P0-P1 for the Opposite Vertices (OVs) T0-T2.*

2.2 Implementation and problems

The above mentioned algorithm processes the model with the "by edge" approach. The multiplicity could also be calculated with the "by triangle" approach (with identical results). In parallel implementation, the "by edge" approach, used in this paper, is better than "by triangle" implementation, although the later may seem more natural. The main reason is that the edges are independent to each other, so this avoids concurrent memory writes. While the "by triangle" approach would lead to usage of atomic operations. Therefore, the "by edge" implementation is exploited.

The algorithm assumes that the evaluation of multiplicity is consistent within each triangle. Unfortunately, this is not the case for floating point arithmetics used in HW. Let us consider an example as shown in Figure 3 In the "by edge" approach, the multiplicities could be evaluated inconsistently for the triangle which is (almost) parallel to the LP. While the error was demonstrated for a single triangle model, such error can occur in a more complex model for individual triangles and ruin the whole silhouette leading into visible artefacts in shadows (see Figure 1).

2.3 The proposed robust algorithm

The proposed algorithm resolves the above issue connected with the inconsistency of triangle edges multiplicity evaluation. The main idea of the improvement is that the triangles, where the inconsistency can occur, are removed from the silhouette calculation. Because these triangles are (almost) parallel with the LP (their shadow volume would be zero), they cannot affect the shape of the resulting shadow. In fact, the removal of the triangles is equivalent to evaluation of its edges multiplicity to 0 which would occur in triangles parallel to the LP if the precision was not limited.

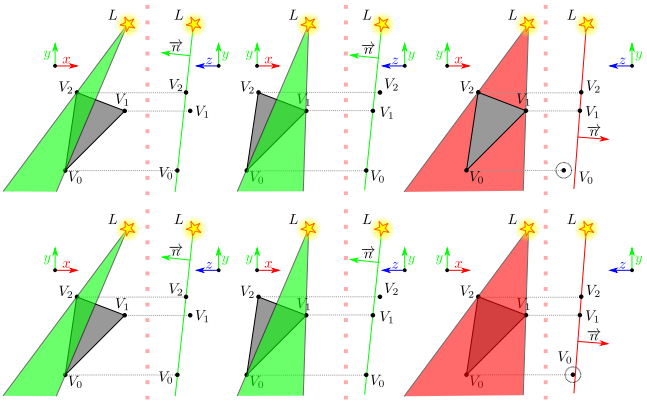

Figure 3: *The grey triangle generates a SV. The green and red triangles represent the two orientations of the light plane (LP). The lower part shows erroneous calculation. In the upper right part, triangle is in front of LP with respect to the viewer. In the lower right, this is incorrectly evaluated, therefore the vertex V0 is assumed to be behind.*

The question is "What is the most efficient way to remove such triangles?". Note that while the "by triangle" approach permits to simple discard the computed triangle, the "by edge" approach does not. One possible solution would be to evaluate "how close to parallel" the triangle is to the LP but in this case, the evaluation would have to be consistent for each triangle edge leading more or less to the same problem. Therefore, a solution was taken to "simulate" evaluation of the "other two edges" of the triangle formed by the edge and each OV. Our modification to the original algorithm is modification of step 3:

3. For each OV belonging to the edge a triangle is formed from OV and the edge. The multiplicity is evaluated for **every side** of this triangle and its remaining vertex as +1 or -1 depending on which side of LP the vertex lies. If the vertex lies exactly on the LP, its multiplicity is 0. If the evaluation of the multiplicity is inconsistent, the triangle is disregarded (the OV multiplicity is set to 0). Note also, that the same order of vertices and edges in triangles must be preserved for each edge evaluation. The final multiplicity of an edge is given by the sum over the multiplicities of every OV. See Figure 2.

The actual multiplicity evaluation for the edge AB for the light source position $\mathbf{L}$ and set $\mathfrak{O}$ of all OV, each in homogeneous coordinates, is as follows:

The LP itself is defined as:

$$\begin{aligned}\mathbf{V} &= (\mathbf{L}_x - \mathbf{A}_x\mathbf{L}_w, \mathbf{L}_y - \mathbf{A}_y\mathbf{L}_w, \mathbf{L}_z - \mathbf{A}_z\mathbf{L}_w)\\ \mathbf{N} &= \text{normalize}((\mathbf{A} - \mathbf{B}) \times \mathbf{V})\\ \mathbf{LP} &= (\mathbf{N}_x, \mathbf{N}_y, \mathbf{N}_z, -\mathbf{N} \cdot \mathbf{A})\end{aligned} \tag{1}$$

The multiplicity of the edge is:

$$m = \sum_{\mathbf{o} \in \mathfrak{O}} \text{sgn}(\mathbf{LP} \cdot \mathbf{o}) \tag{2}$$

Where $|m|$ denotes the number of times the side of SV, extruded from this particular edge, is actually drawn/rendered.

Of course, the evaluation of the above set of expressions, for each edge of the triangle (instead of just once for each triangle), introduces a computational overhead. While some subexpression results can be reused, a significant overhead remains. However, it turns

	GF650Ti	AMD7950	HD3000	HD4000
CPU	4.9 / 5.0	12.9 / 13.9	9.9 /11.2	6.3 / 5.9
AVX+OMP	4.9 / 5.1	11.8 / 13.5	10.0 / 12.4	6.6 / 8.6
GS	30 / 40	92 / 98	n/a	4.4 / 8.7
OpenCL	42 / 41	83 /80	n/a	7.6 / 7.7

Table 1: *Results of experiment (the first value shows FPS of the robust implementation, the second value is the original implementation)*

out that the cost of additional arithmetics, especially in case of exploitation of powerful computational platform, is less costly than increased memory traffic or synchronization operations needed in alternative approaches.

3 Experiments and Results

The experiments were conducted in order to evaluate the achieved results, to assess feasibility of exploitation of the presented approach at the above mentioned platforms and in different applications, and to verify that the approach works well.

The performance of the algorithm was tested under these conditions: the robust version compared to the original version with no robustness, objects of simple shape compared to more complex shapes with similar number of triangles, varying levels of geometric complexity of the same object and scenes containing several separated objects.

3.1 Robust versus traditional implementation

The purpose of this test was to evaluate whether at all and how much the implementation of the robustness of the algorithm adversely affects performance. The conditions for the test were made similar to the real applications conditions in terms of the size of the scene ($\sim 10^5$-10^6 triangles). The test was performed on a scene that did not exhibit significant occurrence of visual artefacts caused by the traditional implementation not being robust.

The results of experiments are shown in the Table 1. Note that performance of the robust algorithm was slightly worse than in the traditional implementation (the adverse effect of additional calculations was less than 10% at all platforms) but this is true only in case of scenes exhibiting no or little visual artefacts. In scenes with larger amount of artefacts, performance of the robust implementation was mostly better and the larger the amount of artefacts, the worse the traditional implementation perform also from the computational time point of view probably due to the fact that that the artefacts caused increase in the fill rate. Overall, quite surprisingly, the decrease of performance in the robust method is not a problem on any platform.

3.2 Simple versus complex shapes

The purpose of the next test was to evaluate how the shape of the objects influences the rendering times. Therefore, scene consisting of simple shapes, spheres, and scene consisting of complex shapes, bunnies, similarly large in terms of triangles, were compared. The size of the scene in this case was about the usual application size ($\sim 6.5 \cdot 10^5$ triangles). One measurement was performed for the scene consisting of more (10) objects, one for the scene consisting of a single object but with the same complexity as in the previous case; this was done in order to check whether the number of objects affects the calculation speed.

	GF650Ti		AMD7950		HD3000		HD4000	
CPU	4.9	5.0	12.9	7.8	9.9	**11.0**	6.3	5.4
	5.5	5.5	13.3	9.6	12.2	**10**	9.6	5.5
AVX+OMP	4.9	5.1	12.9	4.3	9.9	6.2	6.3	6.0
	5.5	5.5	12.8	5.2	**12.8**	5.1	**11.9**	**6.2**
GS	30	30	**92**	95	n/a	n/a	4.4	n/a
	34	34	**156**	124	n/a	n/a	6.0	n/a
OpenCL	**42**	**52**	83	**96**	n/a	n/a	**8.1**	**8.2**
	54	**67**	142	**147**	n/a	n/a	11.3	6.0

Table 2: *Results for one particular GPU and one platform consisting of 4 tests on different scenes: 10 individual bunnies (top left), 10 baked bunnies (top right), 10 individual spheres (bottom left), and 10 baked spheres (bottom right).*

	GF650Ti		AMD7950		HD3000		HD4000	
CPU	3.9	35	5.1	81	**5.3**	**34**	**37**	20
	3.4	27	8.4	61	6.5	32	4.4	18.5
AVX+OMP	3.9	35	5.7	81	4.9	32	4.2	**23**
	3.4	28	9.3	63	**6.7**	**34**	**5.0**	**24**
GS	23	165	85	**650**	n/a	n/a	3.7	15
	18.7	**100**	**84**	**482**	n/a	n/a	3.1	19.3
OpenCL	**43**	**192**	**91**	430	n/a	n/a	**6.0**	19.3
	13.4	26	22	54	n/a	n/a	4.4	9.1

Table 3: *Computational performance based on the number of triangles in the scene. Results for one particular GPU and one platform consisting of 4 tests on different scenes: one sphere with 10^6 (top left), one sphere with 10^5 triangles (top right), 10x10 spheres each with 10^4 (bottom left), and 10x10 spheres with 10^3 triangles (bottom right).*

3.3 Number of triangles in the scene

The consequent test was focused on the behavior of the algorithm in rendering of the scene depending on the number of triangles in the scene. The goal was to learn how the algorithm performs when the number of triangles changes from relatively low ($\sim 10^5$ triangles) to relatively high ($\sim 10^6$ triangles).

It can be seen that, as expected, performance of the algorithm decreases with the increased size of the object in terms of triangles. However, what was not as expectable is the fact that on different platforms, performance does not decrease uniformly and also that performance is not affected uniformly in different implementations on the same platforms. Additionally, in cases where the number of triangles is large, performance is also adversely affected by subdivision of the scene into separate objects as described below.

3.4 Number of isolated objects in the scene

The goal of this final test was to demonstrate how the algorithm performs in dynamic scenes where typically a scene is composed from a number of ($\sim$10-100) independently movable objects and cannot be represented by a single object to enable for easy independent motion of the objects. The test was performed along with the above mentioned testing of performance with changing number of triangles in the scene. It shows that the number of isolated objects does have impact on results especially in case of some platforms, probably due to synchronization and data transfer issues.

3.5 Hardware platforms

The implementation was tested on following platforms:

- AMD Radeon HD 7950 (driver version: 13.1)
- GeForce 650 Ti (driver version: 314.07)
- HD3000 integrated Intel GPU (driver version: 9.17.10.3062)
- HD4000 integrated Intel GPU (driver version: 9.18.10.3071)

First three GPUs were tested with Intel i7-2600K with 16GB RAM. The HD4000 was tested with Intel i5-3570K with 8GB RAM. All the measurements were carried out on Windows 7 x64 SP1.

3.6 Interpretation of results

Some of the results observed in the above tests are especially worth mentioning and they are listed below:

- Performance of "OpenCL" implementation is very good on all the platforms for which OpenCL is available at all. The current solution heavily suffers from the synchronization between OpenCL and OpenGL contexts. This mainly occurs in scenes containing many separate objects because synchronization must be performed for each object. Unfortunately, OpenCL solution is not supported by the HD3000.
- The "Geometry Shader" implementation performs well in scenes with many separate objects. The performance declines with increasing complexity of models more than in OpenCL implementation.
- CPU+AVX+OMP performance is sometimes surprisingly less than the standard CPU implementation. On the other hand, measurements carried out on i5-3570K (Ivy Bridge architecture) showed 4-30% performance increase (16% in average), compared to standard one. Despite the fact that our CPU is capable of processing 8(4) threads concurrently, maximum performance increase is only 30% in its peak. The reason is that not all parts of the algorithm can be parallelized or rewritten using AVX intrinsics.

It is most suitable to use OpenCL implementation (where available) for scenes which do not contain too many (less than 100) objects. Geometry shader solution can be used in situations where the scene contains larger number of separated objects. CPU+AVX+OMP implementation should be used on modern CPUs in situations where the above solutions are not available. Standard CPU implementation should be used otherwise.

4 Conclusions

This paper presented a novel approach to Shadow Silhouette Shadow Volumes that leads into a more robust implementation, which has been tested and evaluated on a number of different hardware platforms.

The proposed approach proved to be working well and producing quality shadows with no visual artifacts. At the same time, it exhibits high performance in variety of hardware platforms. As shown in the paper, it can be efficiently implemented in CPU both using the traditional instructions and the SIMD instructions as well as in GPU using Geometry Shaders as well as using OpenCL.

The most efficient achieved implementation was in OpenCL for a scene containing 10^6 triangles, followed by the Geometry Shader implementation that is usable also with Intel HD 4000 platform. However, the OpenCL implementation suffers from synchronization slowdowns in case the scene is divided into more independent objects. As for the CPU implementations, while they are generally lower performance than the GPU ones, the SIMD instructions (AVX) and parallelism boosts performance on the latest CPU architectures.

Overall, the proposed approach performs very well and at the same time it is robust and precise in terms of per pixel precision of the shadows. Therefore, it represents a very good alternative to shadow methods.

Future work includes improvements of the OpenCL implementation in terms of synchronization in scenes containing more objects, general improvements of the triangle tests. The future work also includes more thorough evaluation on more platforms and more scenes.

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

ALDRIDGE, G., AND WOODS, E. 2004. Robust, geometry-independent shadow volumes. In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, ACM, New York, NY, USA, GRAPHITE '04, 250–253.

BERGERON, P. 1986. A general version of crow's shadow volumes. *IEEE Computer Graphics and Applications 6*, 17–28.

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*, ACM, New York, NY, USA, SIGGRAPH '77, 242–248.

EVERITT, C., AND KILGARD, M. J. 2002. Practical and robust stenciled shadow volumes for hardware-accelerated rendering.

HEIDMANN, T. 1991. Real shadow real time. IRIS Universe, 28–31.

KIM, B., KIM, K., AND TURK, G., 2008. A shadow volume algorithm for opaque and transparent non-manifold casters.

KOLIVAND, H., AND SUNAR, M. S. 2013. A survey of shadow volume algorithms in computer graphics. *IETE Tech Rev 2013 30*, 38–46.

MCGUIRE, M., HUGHES, J. F., EGAN, K., KILGARD, M., AND EVERITT, C. 2003. Fast, practical and robust shadows. Tech. rep., NVIDIA Corporation, Austin, TX, Nov.

STICH, M., WÄCHTER, C., AND KELLER, A. 2007. Efficient and robust shadow volumes using hierarchical occlusion culling and geometry shaders. In *GPU Gems 3*, Addison Wesley Professional, H. Nguyen, Ed., 239–256.

VAN WAVEREN, J., 2005. Shadow volume construction.

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