Silhouette Extraction for Shadow Volumes Using Potentially Visible Sets

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

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

Adam Herout
Brno University of
Technology
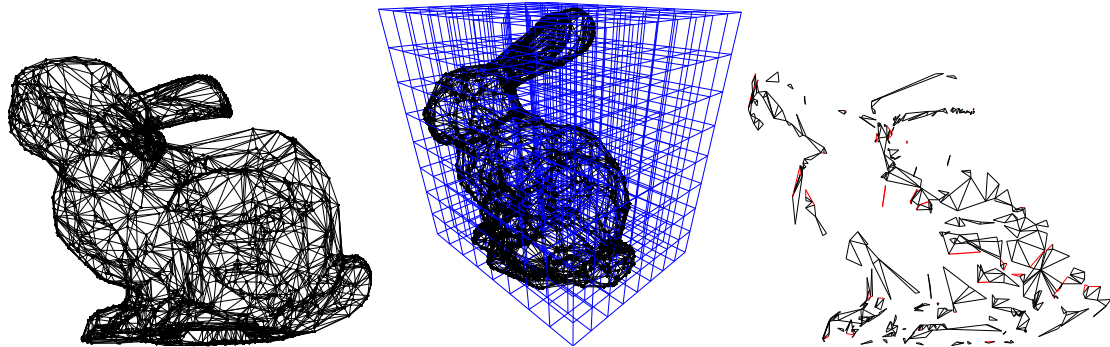Czech Republic
herout@fit.vut.cz

Figure 1: **left:** Wireframe representation of a given model. **middle:** We voxelize the space around the model. One voxel on the lowest octree level is selected, based on the light position, and all potentially-silhouette (need to be tested) and silhouette edges (guaranteed to be silhouette) can be collected by ascending the octree hierarchy. **right:** Red coloured edges are those that are a part of the silhouette after testing the set of potentially silhouette edges (all red and black ones). Only a small subset of model edges need to be tested, which considerably reduces the computational complexity.

ABSTRACT

In this paper, we present a novel approach for accelerated silhouette computation based on potentially visible sets stored in the octree acceleration structure. The scene space, where the light source can appear, is subdivided into voxels. The octree voxels contain two precomputed sets of edges that potentially or always belong to the silhouette. We also propose a novel method of octree compression for reduction of the memory footprint of the resulting acceleration structure. Using our novel technique we were able to considerably decrease the computational complexity of finding the silhouette and reduce its sensitivity to the number of edges.

Keywords

Silhouette Extraction, Octree, Compression, Shadow Volumes

1 INTRODUCTION

Solving surface visibility from a light source (or another point in space in general) is a very fundamental problem of computer graphics. Determining, whether a point on a surface is lit from a point light source has been subject of research for decades, as documented by Woo and Poulin [Woo12]. Over the course of history, two major techniques were developed to address

this problem in the field of rasterization – Shadow Maps and Shadow Volumes. Although the majority of the derived methods of the above mentioned techniques are based on Shadow Maps, Shadow Volumes still provide an important option for scenarios requiring sample-precise shadows, which can be problematic when Shadow Maps are involved due to their discrete nature and limited resolution.

Crucial part of the algorithm of Shadow Volumes is silhouette extraction, i.e. finding the subset of edges that have both visible and non visible triangles connected to them from the light's perspective. Such edges are subsequently extruded as the shadow volume side and rendered into the stencil buffer on the GPU. Usually, all the edges are tested during the rendering of a single frame to determine whether they get extruded or not.

In this paper we focused on improving the silhouette extraction performance of the Shadow Volumes by reducing the number of edges that need to be tested during the Shadow Volumes rendering of an arbitrary triangle soup. Edge indices are stored in an octree-like structure created by voxelizing selected scene space. This octree is later traversed by the GPU to acquire two sets of edges – one set that requires further testing (potentially silhouette edges, **PE**), the second set is known to be silhouette (silhouette edges, **SE**) and edges inside the set are extruded immediately.

The remainder of the paper is organized as follows. Section 2 outlines the previous work, focusing on Shadow Volumes and silhouette extraction. The following section 3 introduces the reader to the details of our algorithm, being divided into octree construction, compression and traversal. Section 4 discusses implementation details and problems. Performance and practical analysis as well as limitations are described in Section 5. Finally, we conclude our findings and results in Section 6.

2 RELATED WORK

Shadow Volumes were first introduced by Crow [Crow77]. The core of the algorithm is casting rays from camera to the scene and incrementing/decrementing their value on intersection with extruded shadow volume sides. When geometry is hit by the ray, the fragment is considered lit or shadowed based on the ray value being zero or non-zero. The first GPU implementation came with the introduction of stencil buffer by Heidmann [Hei91]. The drawback of this method is that when the camera is in the shadow, the stencil test must be inverted. The camera problem of Heidmann's method was solved simultaneously by Everitt and Kilgard [Eve02] and by Bilodeau and Songy [Bil99] in the so-called "z-fail" method, which reverses the stencil test, but requires the shadow volumes to be capped. In order to draw an arbitrary triangle soup, Kim et al. [Kim08] introduced the concept of edge multiplicity, so a single quad cast from an edge is rendered multiple times.

Silhouette extraction methods can be divided into 3 categories – image space, object space, and hybrid (computing in object space, displaying in image space), as categorized by Isenberg et al. [Ise03]. The majority of these methods were used to provide object contours for non-photorealistic rendering, but some of the object-based method are interesting from the perspective of Shadow Volumes. Johnson and Cohen [Jon01] use a hierarchy of normal cones to determine edge visibility. Olson and Zhang [Ols06] propose octree as an acceleration structure to store Hough transform of a 3D mesh. Gooch [Goo99] and Benichou and Elber [Ben99] designed a preprocessing method based on projecting face
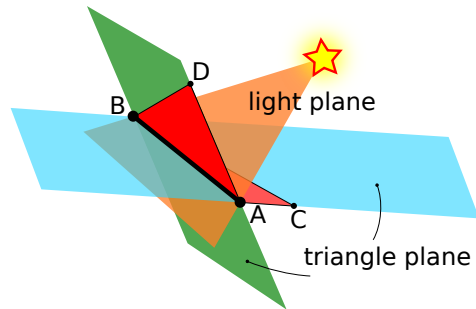

Figure 2: Silhouette edges. Edge AB is not a silhouette edge because triangles ABC and ABD do no lie on the same side of the light plane. Two triangles partition the world space into four subspaces.

normals onto a Gaussian sphere. However, all of these methods are limited to 2-manifold objects.

With the introduction of programmable graphics pipeline, research focused more on the ad-hoc algorithms, mostly due to the fact that the new pipeline provided mechanisms to determine the silhouette during the rendering process with zero or minimal preprocessing. Silhouettes can be computed in almost any programmable shader stage, specifically vertex [Bren02, Mil14], geometry [Sti07], requiring practically no preprocessing, tessellation [Mil15] or compute (OpenCL) [Peč13] shaders.

Gerhards et al. [Ger15] use BSP trees constructed from per-triangle frusta. Fragments are then tested against this structure, whether they are lit or shadowed. This method, however, needs to rebuild the data structure each time the light source or geometry is changed by – even a rigid – transformation.

The proposed method does not require rebuilding (unless the light source moves outside the targeted space) and it is also invariant to affine transformation – the correct voxel in the octree is selected by applying an inverse transformation of the object to the light source and then traversing from the corresponding voxel.

3 ALGORITHM

Our algorithm is based on the concept of the *potentially visible set* (PVS) introduced by Airey et al. [Air90]. It precomputes the results of brute force silhouette extraction for a discrete set of world-space voxels. The brute force extraction process therefore does not need to be executed on all scene edges but only on a small subset that cannot be precomputed, see Figure 1. This section will describe the construction of a compression structure for storing the PVS in an effective manner. It will also describe the modified extraction process.

The algorithm can be broken down into two major stages: **construction** and **traversal**, but first let us summarize the brute force silhouette extraction process.
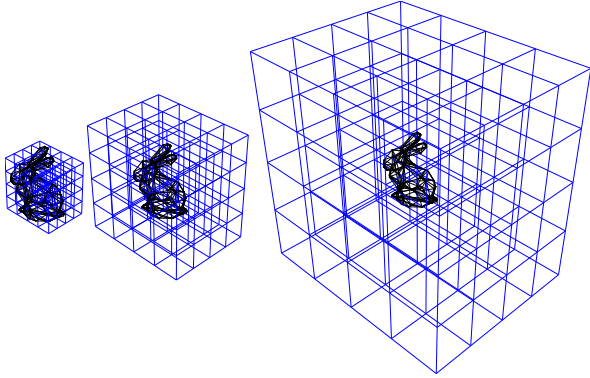
Figure 3: The algorithm supports custom scales of the scene bounding box. If a larger scale is selected, the light can be moved farther from the model. The image shows three different scales with the same voxelization level (in this case 2 levels of octree, $4 \times 4 \times 4$ voxels).

3.1 Silhouette Extraction

A model is composed of vertices that are connected by edges/triangles. An edge is considered as belonging to the silhouette if all triangles adjacent to this edge lie on the same side of the light plane, see Figure 2. In general, from 1 to N triangles can be connected to a single edge. Kim et al. [Kim08] proposed a technique that computes the difference in the number of triangles on the left and the right side of the light plane called edge multiplicity $m \in [-N, N]$.

Without loss of generality, edges with more then 2 connected triangles can be transformed into several simpler edges by splitting and duplicating. If an edge is connected to only one triangle, it is considered a silhouette edge in every case. Our method works with edges having maximum 2 adjacent triangles connected to them.

3.2 Octree Construction

We base the voxelization space on scaling the scene's axis-aligned bounding box (AABB) by a user-specified scaling factor, as seen in Figure 3. The scaling factor depends on the user's needs and on the type of the scene (closed-space scenes will do even with factor 1, open scenes or simple models require larger factors, around 5–10).

This scaled bounding volume circumscribes all the possible light positions. The user can then choose the maximal level of the octree hierarchy, see Figure 4. AABB scaling and maximum octree depth define the octree granularity and voxel size on the deepest level.

We found that depth level of 3–5 is suitable in most scenarios. Larger scales tend to consume too much memory (as described in Chapter 5.1, each octree level increases the amount of memory by a factor of 4). The next step is to find two sets (**SE** and **PE**) for every voxel in the lowest level of the octree. The algorithm tests each edge against all voxels on the lowest level
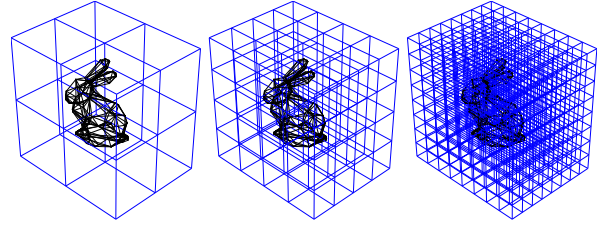

Figure 4: The algorithm supports a custom level of voxelization. The image shows three levels (1,2,3) of depth of the octree for the same scale of the scene bounding box.

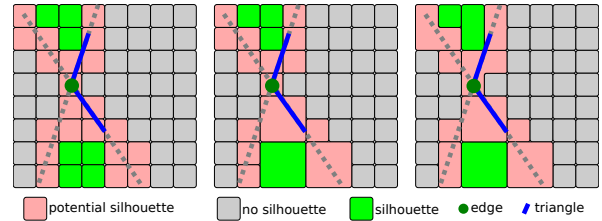potential silhouette   no silhouette   silhouette   ● edge   ╱ triangle

Figure 5: Overview of the proposed approach in 2D. The image shows voxels for one edge. The left side of the images shows the first step of the voxel building algorithm. Voxels are classified into 3 categories – no silhouette, silhouette and potentially silhouette. The next step is to propagate this classification into higher levels (middle image). The right image shows the improvement of compression stage of building algorithm. The octree is transformed into a tree with nodes containing many different subsets of edges defined by bitmasks.
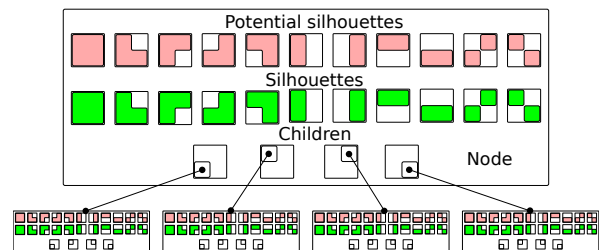

Figure 6: Node data in 2D space for the 8-bit compression. One node contains sets of silhouette and potentially silhouette edges, each addressed by its bitmask value. If a set shape does not intersect the triangle planes of an edge, the edge is stored into the set. The largest set shape is chosen if multiple set shapes do not intersect the triangle planes. A node also contains pointers to child nodes.

of the octree, as seen in Figure 5. If any plane constructed from the triangles adjacent to edge E intersects the voxel, E is considered a PE. If none of the triangle planes intersects the voxel and multiplicity of E is non-zero, it is stored among **SE** (set of silhouette edges). The multiplicity can be computed against any point inside the voxel because the whole voxel lies within one of the four subspaces, as demonstrated in Figure 2.

The next step is to propagate **PE** and **SE** into higher levels of the octree. An edge can be propagated to
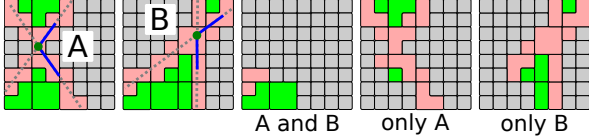
Figure 7: The first two images show two edges – *A* and *B*. Each edge partitions all voxels into voxel shapes for silhouette case and of potential silhouette case. If some voxel shapes are the same for both edges, the edge subsets of those voxel shapes contain both edges (middle image). Otherwise, voxel shapes contain only one edge.

the parent node if it is contained in all of its children. Both types of edges are propagated. The propagation process already significantly reduces the memory footprint. We refer to this propagation scheme as "basic compression".

The last optional step in octree construction is advanced compression. It extends the propagation step by allowing edges to be moved to their parent node even if not all of them are contained in all of its children. These sets of edges are marked with bitmasks corresponding to voxel shapes, see Figure 6. Every subvoxel in these voxel shapes contains the same set of edges, see Figure 7. We call this extended propagation "8-bit compression" as we propagate the edges from children to parent and the bitmask is 8-bit integer. Edges can also be propagated into grandparents (from 64 sibling voxels) which can further improve the compression ratio. This compression scheme is referred to as "64-bit compression". Octree node data are shown in Figure 6.

3.3 Traversal

The traversal part of the algorithm has to copy **SE** and **PE** subsets from the octree into two continuous buffers. The light position determines which subsets of edges have to be copied to the linear buffers, see Figure 8.
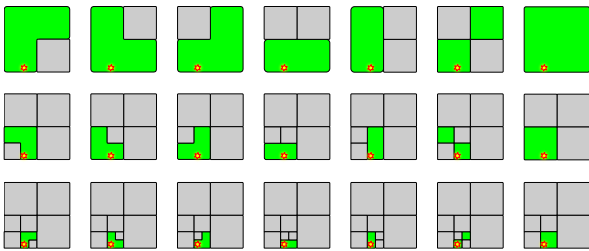

Figure 8: 2D illustration of all edge subsets that contain silhouette edges for a given light position. The hierarchy level is 3. The union of all subsets forms the set of all precomputed silhouette edges. Similar subsets are selected for all potentially silhouette edges. Note that some subsets could be empty. A single edge is contained only in one of the subsets (the largest possible).

The **PE** linear buffer is in the final part of the brute force silhouette extraction process. However, the **PE** set is

very small compared to the set of all edges which leads to performance improvement.

4 IMPLEMENTATION

We implemented the whole process both on CPU and an GPU (OpenGL). The construction process relies on two compute shaders: the first one is used to load the data to the octree, the second one for propagation to upper levels.

Hypothetically, every voxel of the octree could contain more than 50 % of all the edge indices in both **PE** and **SE**. Provided we use Crytek's Sponza model as the reference which breaks down to 431 246 edges with maximum multiplicity of 2, octree maximum depth of 5 ($8^5 = 32\,768$ leaf voxels) and indices stored as 32-bit integers, we may end up, in theory, with more than 50 GB of memory. For larger models, adding edges to the lowest level of the octree runs in batches. The batch size is limited by the GPU memory size.

Usually, not all edges get stored neither in **PE** or **SE** buffer of a voxel, thus their size can be limited to a percentage of total edge count (we found that a factor of 0.8 works for most cases and can be seen for example in Table 1). This increases the batch size and speeds up the building process.

Data are then copied back to the system memory. Before the edge propagation, which is also implemented in a compute shader, the algorithm needs to sort the edges, which is carried out on the CPU in parallel.

We first tried the compression as a multi-core post-processing of the octree, but such implementation, although parallelized, was $20 - 170\times$ slower than 8-bit compression, based on particular scene. For the 8-bit scenario, we moved the advanced compression to the compute shader which tests the edges against octree voxels, because it is in this very step that the bitmask is already known. However, porting the 64-bit compression scheme to GPU seemed problematic as the potential number of sub-voxels is 2^{64}, which would lead to excessive memory footprint, thus we perform the 64-bit compression as CPU post-processing. Due to implementation reasons, compressed nodes using bitmasks other than all-bits-set are located only in `max_depth-1` for 8-bit compression or `max_depth-2` for 64-bit.

During traversal, the algorithm first determines the (X, Y, Z) voxel coordinates within the octree from the light position (flooring the floating point coordinates), which are then converted to linear voxel index. If a light source lies on the boundary between two or more voxels, only one voxel is selected according to eq. (1) where l is the light position and A and B are minimal and maximal corners of the voxel:

$$l_x \in [A_x, B_x) \wedge l_y \in [A_y, B_y) \wedge l_z \in [A_z, B_z). \quad (1)$$

The traversal process depends on the compression level. For non-compressed and 8-bit compression scenarios, we first traverse the octree and compute an exclusive scan of sizes of all necessary sub-buffers, which serves as the input to the second stage that performs the actual data copy from selected subsets to two linear buffers, one for **PE**, the other for **SE**. Traversal for 64-bit compression splits the pre-processing and prefix scan into two steps as the amount of sub-voxels increases to several thousands. Splitting the stage also helps reducing the count of the global memory reads.

5 RESULTS

Evaluation took place on the following test setup: Intel Core i5 6500, 16GB of DDR4, nVidia GeForce RTX 2080Ti (11GB of GDDR6, driver version 419.17), Windows 10 Pro x64. The test application was built using Visual Studio 2015 x64.

5.1 Build and Compression Tests

We conducted comprehensive build tests on the Šibenik, Conference, and Sponza scenes. The aim was to evaluate the building time and the size of the octree structure, based on the octree deepest level, size of the voxelization area, and compression type.

Tables 1, 2, and 3 show the performance evaluation of the building process under various octree settings, with respect to the selected light source position inside the scene's bounding box. We compared 3 types of the build – with basic compression only (first two coloured columns), 8-bit GPU compression (c8) and 64-bit CPU compression (c64).

One of the first things the reader may notice is that 8-bit compression on GPU performs actually faster than the non-compressed version of the algorithm. This is due to fact that GPU compression occurs in the very first stage of the algorithm thus the following stages have to process a smaller amount of data. However, the 64-bit compression is performed as a postprocessing step and it happens on the CPU, thus being very slow, even though the algorithm was written using OpenMP. We tested the 64-bit compression also on the AMD Thread-Ripper system with 24 cores, which improved the 64-bit compression build time by around 60 %, but other two methods performed significantly slower, probably due to different architectures of the two processors.

It can be seen that the amount of memory required to store the octree increases with a factor of 4 with each octree level, but also the average amount of extracted SE increases by around 10 % and the average number of PE that needs to be tested is almost halved with increasing octree depth, for each of the tested scenes.

This test, however, shows the biggest weakness of the algorithm, the memory consumption which, for a particular model, is strongly dependant on the algorithm's

settings. For practical use, the 8-bit compression scheme seems to be the the best choice, in terms of both the building speed and the size of the resulting octree structure.

The memory consuption can be approximated using eq. (2), where S is an approximation of the resulting size of the octree structure in MB, e is the number of edges in millions, d is octree depth and c is compression ratio:

$$S(e,d,c) = e \cdot 8^d \cdot V_d \cdot c \qquad (2)$$

Based on the results in Tables 1, 2, and 3, we estimated the average compression ratios to 0.32 for 8-bit compression scheme and 0.11 for 64-bit scheme by dividing the compressed octree size with non-compressed octree. Values V_d define the approximate size of a single voxel per 1 million edges. These values were calculated as $V_d(d,e) = S_m/8^d/e$, where S_m is the measured size of non-compressed octree. Values obtained by this equation are $V_3 = 0.93$, $V_4 = 0.53$ and $V_5 = 0.30$.

The average relative deviation of eq. (2) is 6 %.

5.2 Silhouette Extraction Tests

We compared our new method (with 8-bit compression) to a brute-force compute shader implementation of silhouette extraction, based on an OpenCL implementation and multiplicity theorem described in [Peč13]. Both methods output edge indices as their result. For this test, we compiled 26 models in total, mixing popular models (Sponza, Šibenik, Buddha, Conference, Gallery, Bunny[1]) with two types of synthetic scenes that we created – the first type were scenes consisting of uniform grid of increasing amount of spheres, having 33 750 to 1 574 640 edges. The second type consisted of randomly positioned spheres differing in numbers, having 124 200 to 933 120 edges. We evaluated our algorithm with two levels of octree depth (3 and 5) posing as best and worst case, and scene scales 1, 2, 4, 8, and 16.

In a single test run, we moved the light source through the octree volume in a $10 \times 10 \times 10$ grid, both for our method and the brute-force approach. From each light position we evaluated the traversal time as average of 5 repetitions. In total, we made 75 000 measurements in each scene: 50 000 for our approach and 25 000 for bruteforce method. As mentioned above, we tested 2 octree levels; that is why our method has twice as many measurements per scene.

The result can be seen in Figure 9. Our accelerated approach has reduced the sensitivity of the algorithm to the number of edges, compared to the bruteforce approach. Our method performs always better on models

---

[1] freely available at https://casual-effects.com/data/

Octree Depth	Scale	Size (MB)	Build (s)	Size c8 (MB)	Build c8 (s)	Size c64 (MB)	Build c64 (s)	Pot Avg	Sil Avg	Pot % Avg	Sil % Avg
3	1	52	0.62	16	0.58	5	11.39	19668	16105	16.76	72.20
	2	57	0.60	18	0.57	5	14.69	21586	15649	18.40	69.91
	4	58	0.61	18	0.57	5	17.12	22088	15514	18.82	69.25
	8	58	0.61	18	0.57	5	17.98	22179	15494	18.90	69.26
	16	58	0.61	18	0.57	6	18.35	22147	15498	18.87	69.19
4	1	235	1.77	77	1.54	27	21.37	10291	18801	8.77	84.28
	2	256	1.75	84	1.55	29	29.13	11359	18531	9.68	82.85
	4	262	1.75	86	1.54	30	32.38	11610	18469	9.89	82.51
	8	263	1.75	86	1.53	30	35.10	11688	18444	9.96	82.37
	16	263	1.74	86	1.53	30	35.32	11690	18448	9.96	82.42
5	1	1022	7.67	341	6.82	127	62.17	5304	20405	4.52	91.52
	2	1122	7.71	376	6.78	139	76.00	5876	20266	5.01	90.62
	4	1147	7.78	385	6.78	141	81.60	6008	20246	5.12	90.45
	8	1155	7.69	388	6.79	143	83.72	6038	20236	5.15	90.41
	16	1155	7.75	388	6.77	142	86.69	6051	20237	5.16	90.39

Table 1: Build test of Sibenik scene, consisting of 117 342 edges. We evaluated the build times and resulting octree size under various voxel sizes and scales. The 3rd and 4th columns contain results for octree build with basic compression scheme. Columns tagged "c8" and "c64" show build times and sizes when using 8-bit or 64-bit advanced compression schemes. The numbers in "Pot Avg" and "Sil Avg" columns show the average number of PE and SE acquired during octree traversal, tested from each lowest level voxel. The second column from the last tells the average amount of edges from the full edge count that needs to be tested, the last column describes the average amount of SE acquired from octree as the percentage of all silhouette edges observed from light position in the middle of each lowest level voxel.

Octree Depth	Scale	Size (MB)	Build (s)	Size c8 (MB)	Build c8 (s)	Size c64 (MB)	Build c64 (s)	Pot Avg	Sil Avg	Pot % Avg	Sil % Avg
3	1	80	0.84	25	0.79	8	34.95	34338	19426	17.61	65.87
	2	93	0.84	29	0.78	9	50.55	39952	18481	20.49	62.37
	4	96	0.84	30	0.79	10	59.29	41323	18288	21.19	61.56
	8	97	0.84	31	0.78	10	62.89	41656	18265	21.36	61.46
	16	97	0.84	31	0.78	11	64.99	41794	18243	21.43	61.32
4	1	379	2.55	121	2.54	42	69.81	18675	23055	9.58	78.19
	2	431	2.64	139	2.53	48	106.65	21857	22317	11.21	75.35
	4	443	2.62	144	2.52	50	124.19	22595	22187	11.59	74.71
	8	446	2.64	145	2.53	51	134.23	22783	22154	11.68	74.54
	16	447	2.65	146	2.54	52	136.04	22832	22149	11.71	74.51
5	1	1786	10.66	581	8.80	209	150.24	9894	25738	5.07	87.29
	2	2044	11.49	668	8.92	238	222.80	11698	25234	6.00	85.18
	4	2102	11.12	687	8.61	245	265.10	12123	25151	6.22	84.69
	8	2120	11.23	694	8.81	248	284.60	12219	25141	6.27	84.58
	16	2121	11.28	694	8.96	249	291.80	12241	25135	6.28	84.56

Table 2: Build test of Conference scene, consisting of 195 019 edges. Check table 1 for column description.

Octree Depth	Scale	Size (MB)	Build (s)	Size c8 (MB)	Build c8 (s)	Size c64 (MB)	Build c64 (s)	Pot Avg	Sil Avg	Pot % Avg	Sil % Avg
3	1	192	1.56	60	1.40	19	270.59	83074	31640	19.26	58.98
	2	202	1.57	63	1.38	20	332.78	86744	30845	20.11	57.30
	4	205	1.59	65	1.37	20	361.90	87829	30661	20.37	57.08
	8	206	1.62	65	1.37	21	373.43	88157	30644	20.44	57.08
	16	206	1.60	66	1.38	21	379.24	88382	30618	20.49	57.02
4	1	893	5.31	289	5.10	96	586.34	44817	39567	10.39	73.72
	2	930	5.36	302	4.62	101	705.98	47044	39029	10.91	72.58
	4	943	5.40	306	4.62	102	783.30	47414	38933	10.99	72.34
	8	946	5.42	306	4.64	103	819.03	47559	38906	11.03	72.34
	16	948	5.41	307	4.61	103	837.29	47636	38899	11.05	72.34
5	1	4111	21.17	1359	15.45	498	1210.22	23895	45123	5.54	84.18
	2	4305	21.37	1425	15.90	504	1517.91	25094	44867	5.82	83.47
	4	4345	21.20	1438	16.18	522	1635.41	25321	44782	5.87	83.34
	8	4351	21.25	1441	16.36	511	1735.49	25374	44819	5.88	83.30
	16	4357	21.13	1443	16.26	519	1789.33	25376	44782	5.88	83.31

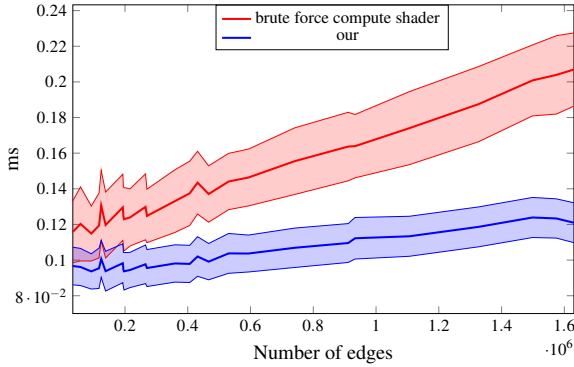Table 3: Build test of Sponza scene, consisting of 431 246 edges. Check Table 1 for column description.

Figure 9: Average extraction times for compilation of 26 scenes (sorted by the number of edges). Red line represents bruteforce compute shader method, blue line represents our new proposed method. The area around the lines represents (+-) mean absolute deviation.

Compression	Average (ms)	Max Abs Deviation (ms)
basic	0.098	0.011
8-bit	0.102	0.012
64-bit	0.134	0.013

Table 4: Comparison between compression levels on Sponza scene. Average octree traversal time calculated from 1000 different light positions in the scene and maximum absolute deviation from the average.

having more than 200 000 edges and it is also more stable. Its average absolute variance is almost half, compared to the bruteforce method.

We also evaluated the performance difference when using different compression ratios. We used the Sponza model and moved the light source around in the same way as described in the previous test. The results can be seen in Table 4. The complexity of the 64-bit com-
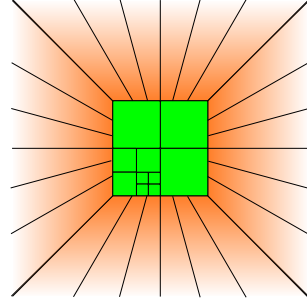

Figure 10: One of the possible extensions to the algorithm. Instead of using one hierarchical structure, the algorithm would use two – one for the close vicinity of the model and one for all the other space around. The green part shows the hierarchical structure as presented, the orange parts is the second hierarchical structure. The second structure uses angles instead of voxels.

pression traversal outweighs its benefits in the form of lower size, thus the 8-bit compression scheme seems to be the best choice.

We can estimate the extraction time for brute force approach as $t_b = E \cdot K$ where E is the number of edges and K is extraction complexity. Our method yields $t_t = P \cdot E \cdot K + T$ where T is traversal cost and P is the ratio of potential edges, which can be seen in 2nd last column of Tables 1 - 3. Based on the build test results, we estimated the P to be 0.2, 0.1 and 0.05 for octree levels 3, 4 and 5. According to our measurements, T was 0.075 ms in average and was not dependant on the number of edges.

6 CONCLUSION

We presented a novel approach to accelerated silhouette extraction by storing pre-computed PVS in an octree, as well as novel octree compression schemes.

The majority of the building process was implemented on GPU using OpenGL and compute shaders. The building process is reasonably fast when using 8-bit compression scheme, as it processes less data then the basic compression scheme. The resulting octree can be stored in a file to avoid repetitive builds in subsequent runs. Experimentally, we were able to reduce the octree size using 64-bit compression to around 12% of basic compression scheme, but the compression itself was used as a post-processing step on CPU, thus not performing as fast as the GPU implementation. In terms of performance, 64-bit compression also lagged behind due to having a more complicated traversal. The 8-bit compression scheme provides the best results in terms of octree size and traversal speed.

Compared to the brute-force approach, our method is less sensitive to the number of edges. It was also more stable, in terms of maximal absolute deviation. The biggest drawback of the method is its memory consumption and also spatial limitation due to the nature of voxelization. The method also would not work well on scenes with dynamic geometry (f.e. morphing).

This method could be further improved by storing triangle indices instead of edges, which, in theory, could reduce memory footprint of the method even more. The whole structure does not need to reside on the GPU but can be streamed as needed. Future research could also evaluate usage of homogeneous coordinates, which may create hierarchy with unlimited spatial span, see Figure 10.

7 ACKNOWLEDGMENTS

8 REFERENCES

[Mil14] Milet, T., Kobrtek, J., Zemčík, P, Pečiva, J. Fast and Robust Tessellation-Based Silhouette Shadows. In WSCG 2014 - Poster papers proceedings, 2014.

[Peč13] Pečiva, J., Starka, T., Milet, T., Kobrtek and Zemčík P., Robust Silhouette Shadow Volumes on Contemporary Hardware. In Conference Proceedings of GraphiCon'2013, pp 56–59, 2013.

[Wil78] Williams, L., Casting curved shadows on curved surfaces. In SIGGRAPH Comput. Graph., pp 270–274, 1978.

[Jon01] Johnson, D. E. and Cohen, E., Spatialized Normal Cone Hierarchies. In SI3D '01, pp 129–134, 2001.

[Crow77] Crow, F. C., Shadow algorithms for computer graphics. In SIGGRAPH '77, pp. 242–248, 1977.

[Woo12] Woo, A. and Poulin, P., Shadow Algorithms Data Miner. CRC Press, ISBN 978-1-4398-8023-4, 2012.

[Mil15] Milet, T., Tóth, M., Pečiva, J., Starka, T., Kobrtek, J. and Zemčík, P., Fast robust and precise shadow algorithm for WebGL 1.0 platform, In ICAT-EGVE 2015, pp 85–92, 2015.

[Ols06] Olson, M., Zhang, H., Silhouette Extraction in Hough Space, In Computer Graphics Forum 25(3), pp 273–282, 2006.

[Sti07] Stich, M, Wächter, C, Keller, A., Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders, In Nguyen, R. (Ed.) GPU Gems 3, ISBN 978-0321515261, pp 359–378, 2007.

[Bren02] Brennan, C., Shadow Volume Extrusion using a Vertex Shader, In Engel, W. E. (Ed.) ShaderX, pp 188-194, 2002.

[Hei91] Heidmann, T., Real shadow real time, In Iris Universe 18, pp 23–31, 1991.

[Eve02] Everitt, C. W., Kilgard, M. J., Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering, In CoRR, 2002.

[Bil99] Bilodeau, B. and Songy, M., Real time shadows, Creativity, 1999.

[Ise03] Isenberg, T., Freudenberg, B., Halper, N., Schlechtweg, S., Strothotte, T., A developer's guide to silhouette algorithms for polygonal models, In IEEE Computer Graphics and Applications, vol. 23, no. 4, pp. 28-37, July-Aug. 2003.

[Kim08] Kim, B., Kim, K., Turk, G., A Shadow Volume Algorithm for Opaque and Transparent Non-Manifold Casters, In Journal of Graphics Tools 13, pp 1–14, 2008.

[Goo99] Gooch, B., et al., Interactive Technical Illustration, In Interactive 3D Graphics, pp. 31–38, 1999.

[Ben99] Benichou, F. and Elber, G., Output Sensitive Extraction of Silhouettes from Polygonal Geometry, In Proc. 7th Pacific Graphics Conf., pp. 60–69, 1999.

[Ger15] J. Gerhards, J., Mora, F., Aveneau, L., Ghazanfarpour, D., Partitioned Shadow Volumes, In Computer Graphics Forum, volume 34 issue 2, pp 549–559, 2015

[Air90] Airey, John M. and Rohlf, John H. and Brooks,Jr., Frederick P., Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments, SIGGRAPH Comput. Graph., volume 24, 1990