Parallelization of Shape Diameter Function Computation using OpenCL

Rastislav Kamenicky*
Supervised by: Martin Madaras†

Faculty of Mathematics Physics and Informatics
Comenius University
Bratislava / Slovakia

Abstract

Shape Diameter Function (SDF) is a scalar function that expresses a measure of the diameter of the object's volume in the neighborhood of each point on the surface on an input mesh. It is fundamental in many applications in computer graphics used for consistent mesh partitioning and skeletonization. The algorithm sends several rays inside a cone centered around the point's inward-normal direction and measures the distance at the point of intersection. We have implemented the original algorithm and further extended it on GPU by parallelizing the ray casting process using OpenCL. We have also generalized the algorithm to support non-manifold meshes. The algorithm shows great speedup in terms of timing when compared with the CPU based implementation.

Keywords: Shape Diameter Function, OpenCL, Parallelization

1 Introduction

Analysis of 3D models and processing of spatial data is a fundamental part of computer graphics. However acquired models are often non manifold or lack crucial data i.e. skeletal representation, UV coordinates, etc. Methods as mesh processing and shape analysis are commonly used to fill missing information. Such methods require algorithms that are robust and work fast and effectively.

SDF is a volume-based shape function that can help to process and manipulate families of objects which contain similarities using a simple and consistent algorithm. It can be used for skeleton extraction and mesh partitioning and contraction. SDF remains largely unaffected by pose changes of the same object and maintains similar values in analogue parts of different objects [16]. The diameter measured also relates to the medial axis transform (MAT) [4]. However, unlike the expensive computation and handling of medial axis, SDF is much simpler. It is a scalar field created by sending several rays from every input point on the mesh, measuring the distance at the point of intersection.

Such ray casting is highly parallel algorithm. If processed on the CPU, the task becomes extremely inefficient. Therefore, in our approach instead of tracing one ray at a time, we propose a parallel method for computing SDF that is performed on GPU using OpenCL, exploiting the independence of rays.

The method was originally meant to be used effectively only on manifold structures, but the process could be natively expanded to non-manifolds. And so we propose several changes that could further improve it's support for non-manifold structures. Finally, at the end of this paper, we compare the results of our GPU implementation against the CPU implementation.

Figure 1: Visualization of Shape Diameter Function with values normalized to interval $\langle 0, 1 \rangle$.

*kamenicky8@uniba.sk
†madaras@sccg.sk

2 Related Work

MeshLab implementation, implemented by Baldacci [2] used a different approach for calculating SDF. The method is based on iteratively peeling two or three successive depth layers of the mesh from multiple views around the mesh. This is performed through shaders and uses native GPU support for mesh projection and rasterization. Thanks to uniform memory access, this could achieve better results on larger meshes, but it loses detail on parts that are too close to the camera.

In [6], it is pointed out that SDF can be approximated using only a small subset of data. The remaining data is interpolated via Poisson interpolation. Even though the speedup is very significant, a lot of detail is lost on complex surfaces and areas where different body parts connect. This information can be crucial when connecting parts of skeletons and could lead to improper skeletonization.

In [15], it is mentioned that the outliers removal technique proposed in [16] generates counterintuitive results in some cases. The SDF value calculated by the Shapira et al. method is given by the weighted average of all the values thrown inside the point's cone, which for example in the case of a mesh composed of two parallel (infinite) planes, underestimates the correct diameter due to large cone size. In [15] to resolve the dilemma between a small or large cone, a more conservative estimation of the SDF is introduced by using an adaptive cone size. In the case, for example, of the infinite parallel planes this method converges to a very small cone size giving a correct SDF value equivalent to the distance between the two planes. In our GPU implementation we maintain the original outliers removal approach proposed in [16], leaving the one proposed by [15] for future work, because the adaptive cone size is computationally more expensive than original method. The ray has to be cast multiple times to find the correct cone size.

3 Original SDF Algorithm

Let M be an input mesh surface defining a volumetric object. SDF is a scalar function $f_v: \mathbf{M} \rightarrow \mathbb{R}$ that consists of creating a cone centered around inward-normal direction (the opposite direction of its normal) of every point $\mathbf{p} \in \mathbf{M}$. Inside this cone several rays are sent to the other side of the mesh, measuring the euclidean distance at the point of intersection. Outliers are removed and the remaining values are averaged and smoothed. As a result there is a single value for every point $\mathbf{p} \in \mathbf{M}$. The original algorithm consists of 4 steps.

Step 1 - preprocessing: In order to facilitate the ray casting, an acceleration structure is needed. Therefore in preprocessing stage an octree is created.

Step 2 - ray casting: In the ray casting stage rays are cast through octree and euclidean distance at the point of intersection is measured. According to Shapira et al. [16], the ideal number of rays is 30 inside a cone with angle of 120°. The rays are chosen randomly.

Step 3 - outliers removal: After the measured distances are obtained, the rays that are in the same direction as the inverse normal of the mesh they hit (the same direction is defined as an angle difference less than 90°) are ignored. This is performed to remove false intersections with the outside of the mesh. The SDF at a point is defined as the weighted average of all rays lengths which fall within one standard deviation from the median of all lengths. The weights used are the inverse of the angle between the ray to the center of the cone. This is because rays with larger angles are more frequent, and therefore have smaller weights.

Step 4 - smoothing: In order to increase robustness and fill in the values for points that could have ended up with 0 valid rays a smoothing stage is necessary. Anisotropic smoothing is chosen to smooth the values of the points on the mesh.

4 Our Implementation

We have based our implementation upon the original algorithm extending it on GPU by parallelizing the ray casting process using OpenCL. We have inherited all the steps from original algorithm and further extended them on GPU.

4.1 Preprocessing

In the preprocessing stage we have to create an acceleration structure around the mesh that will improve the ray tracing routine. The acceleration structure is one of the most important parts of ray tracing. As noted by [5], the fastest acceleration structure for static scenes is kd-tree, followed by bounding volume hierarchies (BVH). However for the purpose of comparison with the original article, we have chosen to use an octree to be able to compare results, leaving the other ones for future work. We computed the octree on CPU because the preprocessing time was not a concern. It is built in a top-down manner. Triangles that were in the middle of several nodes were detected through Mollers AABB-triangle intersection algorithm [1] and split into multiple nodes. On our test models optimal octree depth ranged from 6 to 12 depending on the number of triangles. We have chosen a maximum depth of 10 for consistent results.

4.2 Ray Casting

There are several ways to perform ray casting on GPU. Carr et al. [3] proposed to generate rays and perform traversal of acceleration structures on CPU, then store the results and perform ray-triangle intersections on GPU. Purcell et al. [14] proposed to store scene geometry and acceleration structure on GPU and perform both traversal and intersection on GPU. Recent efforts in optimizing the algorithms for GPUs have demonstrated to obtain better results on traversal of acceleration structures than a single-core execution on CPU. Therefore in our approach we send data containing triangle indices and acceleration structure to GPU, then we perform a per-ray computation of SDF. Our code is based upon the original implementation of the algorithm extending it to GPU with the help of OpenCL [12]. OpenCL was chosen because it is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, making it an universal solution for multiple platforms.

The computation of SDF is a three step process that consists of generating and casting rays, traversing the chosen acceleration structure and measuring the distance at the point of intersection with the geometry.

Ray Generation: In the first step, we have to generate N rays in a cone centered around inward-normal direction of a given point p. The generation of rays on CPU and then transferring the data to GPU creates unnecessary overhead because we have to store every ray and it's associated weight (inverse of the angle between the ray and the center of the cone). Therefore the rays have to be generated on GPU, but original algorithm generates the rays randomly, which would require defining a pseudo-random generator in OpenCL and store weight per every ray. Rolland [15] has tackled this problem by defining a cone sampling strategy consisting of random rays that are uniformly generated inside the cone. However, we wanted a fast deterministic algorithm that would be uniform for both smaller and larger number of rays and for any given cone. This is to prevent storing weights and to avoid unnecessary bias in the values on the mesh, which can be seen in Figure 2. Therefore, we have decided to evenly distribute rays in a cone with the help of Spherical Fibonacci [10].

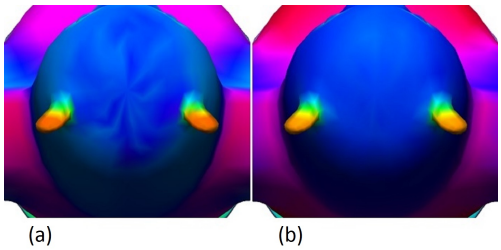

Figure 2: (a) random ray generation, (b) uniform ray generation using Spherical Fibonacci.

The algorithm generates the rays in a sphere from top to bottom. The generation was restricted to a given sphere cap and once a ray is generated, it is transformed into world coordinates by multiplication it with tangent-binormal-normal matrix specified by the point's inward-normal and two orthogonal unit vectors spanning the tangent plane of the point p. It is important to note that only valid points are used to generate the rays from (valid point is defined as having a non-zero normal, tangent and binormal vectors). Using triangle centers can have advantages over vertices in non-manifold models, where the normal of some points can not be properly determined. And it helps to reduce necessary data transfer thanks to the fact normal, tangent and binormal can be calculated.

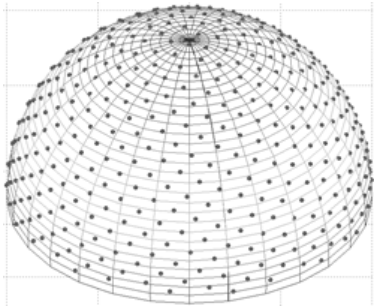

Figure 3: Generating rays uniformly with Spherical Fibonacci, image from [10].

Octree Traversal: In the second step we traverse our acceleration structure. Every ray is computed separately, one ray per work item. This is faster on modern cards that have thousands of cores. Unfortunately, we could not avoid random memory access that slows down the entire process. The octree structure and triangles are sent to GPU and each ray is cast separately. Several octree traversal methods are mentioned by Kristof et al. [7], notably neighbor pointer, kd-restart and short-stack approaches. Our implementation is based on Laine et al. [8] stack-based approach, but unlike [8] the tree is traversed in a top-down fashion all the way to the leaves. The nodes are traversed until we find a valid intersection in the third step.

Triangle Intersection: In the third step we compute ray-triangle intersection between the cast ray and triangles that belongs to a given node. As mentioned by Philippe et al. [13], one of the best algorithms for ray-triangle intersection is Moller and Trumbore [11] because it uses mainly dot and cross product that is fast on current graphical hardware. To improve the algorithm on non-manifolds, we have to skip intersections that are too close to the ray origin, like in the case of self intersecting mesh. We define minimum_closeness using the max dimension of model (max_size) as $mininum\_closeness = max\_size * 2.0 * 0.00001$; Once the intersection is found, we check if the ray is valid by comparing normal at the point of intersection and the ray direction as mentioned

in original algorithm [16]. In a case when no intersection is found, the distance is set to -1 and the ray is ignored. Afterwards the measured distance is stored in memory.

Data Management: Mesh triangles are stored in a 32bit RGBA texture. Coordinates (X, Y, Z, W) of each vertex of a given triangle are stored in a single texel. Normals and other necessary information is calculated on GPU. We encode the topology of the octree using 2 arrays. First one contains 32-bit child descriptors, each corresponding to a single node. The child descriptor contains a 24-bit child pointer and a 8-bit bit-mask that tells whether each of the child slots actually contains a node. In a case when the node is a leaf the child pointer points to the second array containing data for leaf nodes. Each leaf has to store the number of triangles it contains and their pointers to the triangle texture. This is all stored in the second array which can be seen in Figure 4.

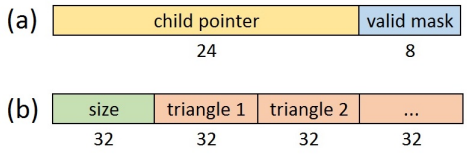

Figure 4: (a) 32bit child descriptor, (b) leaf data.

In a case when we are using triangle centers as the points from which we are casting the rays and measuring the distance, no additional information is necessary. Otherwise, we have to store the point's origin, normal and tangent or other information like triangle / vertex neighbors from which we can fill in the data. At last we have to store our results. They are stored in a single array with size $=$ number_of_points $*$ number_of_rays. Figure 5 shows an example of how the octree topology is stored in memory.

4.3 Outliers Removal

Once we collected all values for our points and their rays, another program is executed on GPU. We do not have to send the data to GPU because they are already there from previous step. For output we create 1 array containing value for every point. The work is split in a way that 1 work item processes data of one point. Rays with lengths which do not fall within one standard deviation from the median of all lengths are removed and the rest is averaged using weights that can be calculated again thanks to our uniform sampling. After we get our final value, we send the data to CPU memory. While we are saving the data, we normalize them, this is very fast and does not needs to run on GPU because we would need another array for second output.

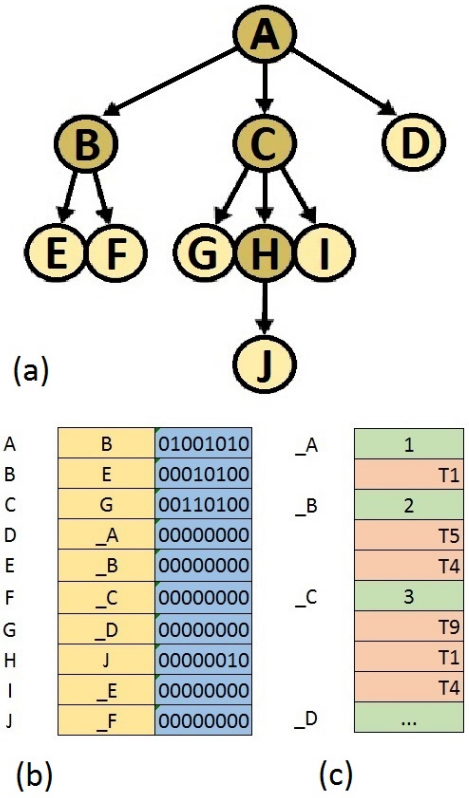

Figure 5: Octree topology on an example. (a) octree structure, (b) 1st array with octree nodes, (c) 2nd array with leaf data.

4.4 Smoothing

As mentioned in the original paper [16], to overcome errors in the measure caused by pose changes or complex surface geometry, a smoothing operation is necessary. The method chosen is directly related to the result we are trying to obtain because various methods can lead to significantly different values. Therefore we propose 3 approaches that can be used to smooth the SDF values in various ways.

Smoothing on Mesh: In first approach we smooth values in mesh, by defining k-ring neighborhood Gaussian smoothing. The k specifies the blur radius given by our connectivity in mesh, which can be seen in Figure 6. We start with a chosen vertex, for which the $k = 0$; In first iteration, we create a list of vertices that share an edge with our first vertex. These vertices have $k = 1$; In every next iteration we create a new list of vertices that share at least one edge with vertices from previous iteration, but only those that we have not yet chosen. This is repeated until we reach our desired k. We then create a 1D Gaussian matrix for our k and perform weighted averaging of all the values from vertices using data from our matrix as weights. This is repeated for every vertex in mesh. To ensure consistency and continuity during smoothing, duplicated vertices have to be merged into one. This ensures

that the k-ring neighborhood Gaussian smoothing will not fail to find neighboring vertices. This method of smoothing does not preserve values at corners. In a case when the values should be preserved, smoothing can be performed on triangles and the dihedral angle between the connected triangles from 2 consecutive iterations can be used as additional weight. This method can be parallelized, but it is not suited for GPU because it requires large dynamic arrays for the vertices and is mostly based on accessing memory than doing mathematical calculations.

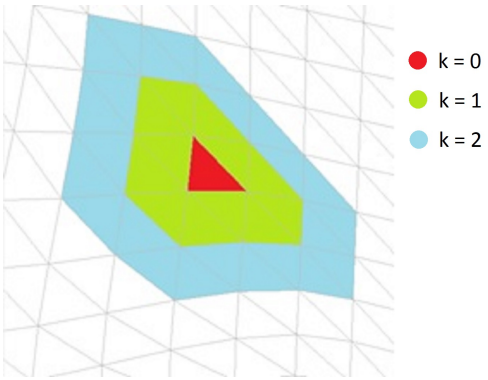

Figure 6: Smoothing by defining a k-ring neighborhood on the mesh.

Smoothing on Projected Points: Our next approach is based on projecting the points in their inward normal direction to a distance which is half of their SDF value. This creates a cloud like structure inside the mesh that resembles medial axis. Then for every point we perform nearest neighborhood search within the radius of the given point's SDF value, acquire the SDF value of every detected point and average the result. But due to the fact that many meshes have round, spheroidal parts where thousands of points can occupy small space, this would lead to a time complexity of $O(n^2)$ in worst case where n is the number of points. Therefore, for efficiency we have to join the points whose distance from each other is too small. This can be done by creating an octree structure, that will store the average value of projected points and their count in it's nodes. And instead of searching nearest points within the radius of the given point's SDF, we search nearest octree nodes, which can be seen in Figure 7. We perform this by traversing the tree from top to bottom, checking if nodes are within the radius of our SDF value. If a node is fully inside our radius, we do not traverse this node further. At the end, we average values from the nodes using weighted averaging. As weight we use the multiplication of number of points in each collected node and an approximate percentage of how much of the cube lies inside the radius. This method can be parallelized on GPU. Besides the standard octree structure that must be send to GPU, we have to include number of points and SDF value for every node. Leaves do not need to contain any additional pointers. Other information as the projected points and their SDF values can be obtained from data that remained from ray casting steps. Octree node positions and dimensions can be interpolated from the position and dimension of root node.

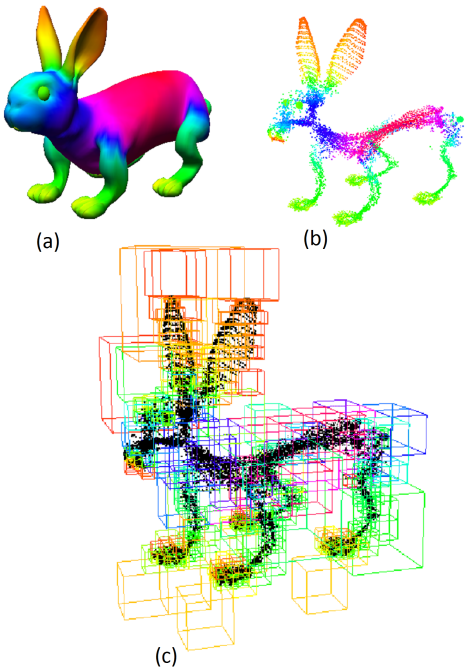

Figure 7: (a) mesh with SDF values, (b) projected points resembling medial axis, (c) octree nodes containing the average SDF values.

Smoothing in Texture: In our third approach we have chosen a more traditional method by smoothing the SDF values in texture. This requires that the mesh has a proper UV coordinates. In a case when the parametrization is missing, but we have a manifold model, it is possible to use Skeleton Texture Mapping [9] to create necessary UV coordinates. If performed on GPU, the fastest way is to use shaders. We bind our texture to a Frame Buffer Object (FBO), then we create a 2D orthogonal projection that has in the bottom-left corner coordinate (0,0) and in the top-right corner coordinate (1,1). Then we use OpenGL to draw the triangles into this texture using their UV coordinates and their SDF values as color. Afterwards we run our shader program that performs the smoothing operations. There are various methods that can be used, from Gaussian to bilateral, median or anisotropic filtering. After the smoothing is performed, we retrieve our results from texture. In a case when we end up with a vertex that has multiple UV coordinates, we can average the result. Result of Gaussian filtering can be seen in Figure 8.

Discussion: Smoothing on mesh: The parameter k is set manually depending on the number of triangles / vertices the mesh contains. We have tested the effect of various parameter settings on the consistency of the SDF on many

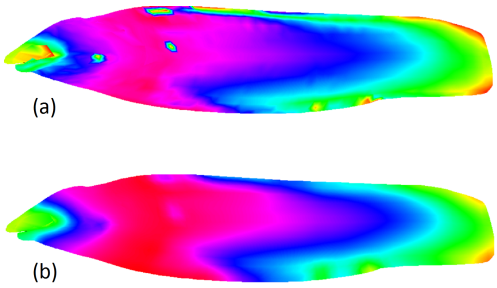

Figure 8: (a) before smoothing, (b) after smoothing.

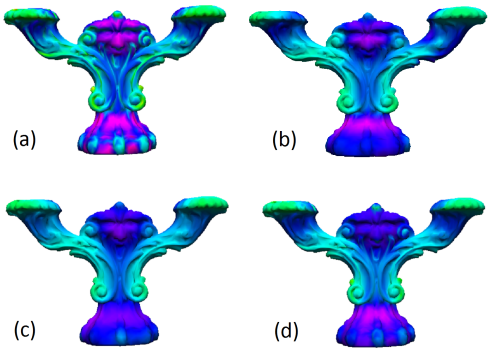

Figure 9: (a) no smoothing, (b) smoothing in texture, (c) smoothing on mesh, (d) smoothing on projected points.

meshes. In practice smaller meshes up to 20k triangles required radius of 2, while larger meshes like point cloud scans with 500k+ triangles radius of 5+. This smoothing approach gave best results when compared to others, however it was the slowest. It fails when the connectivity in mesh is not well defined or when the triangles does not have approximately the same sizes, in this case a lot of detail can be lost.

Smoothing on projected points: When creating an octree additional restrictions can be applied to further reduce the depth, while keeping the detail. In practice, the projected points are much closer to each other than the triangles in mesh and so we can use a smaller maximum depth, 8 was satisfying in most cases. Also minimum number of points for a node to branch can be increased to about 0.2% of all points. We can also compare the average value of the points against the octree dimensions and if it is bigger, then we do not branch. This method is faster than the first one, but the parallelization on GPU leads only to a slightly better results due to a lot of memory access. It also has an advantage that we do not have to set any radius value, because it is automatically acquired from the SDF values. Stronger smoothing can be obtained by iteratively running the method, which can even yield better results. It fails when the mesh surface is too irregular and the points are projected randomly, not forming a cloud like structure. Also it keeps the values at corners, which can be unwanted in some cases.

Smoothing in texture: This method is by far the fastest one, done in terms of ms even on large textures like 2048×2048. The need to have UV coordinates can be contraproductive because the automatic methods to create them are very slow. It can fail in multiple cases. First one is when the texture parametrization does not divide the mesh into logical parts. Second one is the texels of different parts are within the smoothing radius. Third one is when the projected triangles have different sizes. In practice we set the radius to 2 for a 256×256 texture and multiply as necessary, but it can vary depending on the model.

In Figure 9 the effect of all the methods on single model can be seen. Smoothing in texture performed similarly to smoothing on mesh, while smoothing on projected points kept more detail in corners.

5 Results

The testing was done on Intel Core i5, 2,67GHz with 4GB RAM and AMD Radeon R9 290, using 30 rays for each point. The algorithms were implemented in C++ in Visual Studio 2012 using standard OpenCL API.

Table 1 show basic performance of ray casting routine, outliers removal and smoothing. As for the smoothing chosen in the Table 1, we decided smoothing on projected points was the most suitable because it did not require any additional parameters. Table 2 shows octree creation. The time was measured on CPU. Table 3 shows smoothing on mesh using various k-ring area settings. The time was measured on CPU because we did not had a GPU implementation. Table 4 shows smoothing in a 2048×2048 texture using Gaussian Filter with various radius settings. Figure 10 shows percentage difference between GPU and CPU implementation when used on the same model with varying level of detail.

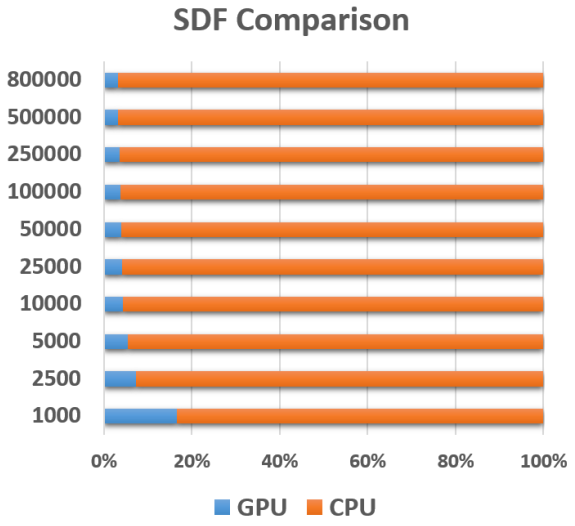

Figure 10: Benefits of GPU implementation with varying level of detail. Comparison was performed on model of Stanford Dragon.

Model	Faces	SDF Computation		Outliers Removal		Smoothing		Total	
Lizard	1 000 000	555,9	18,049	3,3	0.078	7,098	1,166	566,298	19,293
Stanford Dragon	500 000	234,1	8,721	1,32	0.047	3,385	0,576	238,805	9,344
Davy Jones' head	260 000	143,0	2,449	0,843	0,032	1,139	0,334	144,982	2,815
Skeleton	100 000	41,209	0,967	0,521	0,024	0,437	0,155	42,167	1,146
Buzz Lightyear	40 000	11,122	0,499	0,141	0,016	0,312	0,047	11,575	0,562
S-shape	20 000	4,274	0,219	0,063	0,015	0,171	0,016	4,508	0,250
Rabbit	15 000	3,026	0,156	0,047	0,008	0,109	0,015	3,182	0,179
Bottle	2 500	0,316	0,031	0,016	0,006	0,015	0,012	0,347	0,049
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU

Table 1: Results for both CPU and GPU computation of SDF. Listed times are in seconds. Total time does not include preprocessing.

Maximum Depth	Octree Creation
14	13,931
12	8,923
10	3,525
8	0,998
6	0,421
4	0,218
2	0,063

Table 2: Octree creation in preprocessing stage with varying maximum depth. Listed times are in seconds. We used model of Stanford Dragon with 500 000 triangles.

k-ring	Smoothing
8	26,567
7	20,015
6	14,882
5	10,764
4	7,566
3	5,024
2	3,120
1	1,762

Table 3: Smoothing on mesh using various k-ring areas. Listed times are in seconds. We used model of Stanford Dragon with 500 000 triangles.

6 Conclusion

In Section 3, we described the present state of methods used in the in the original paper [16]. In Section 4, we proposed our OpenCL implementation of the algorithm. We described all the steps necessary to perform the ray casting, outliers removal and smoothing on GPU. The various smoothing techniques which we subsequently developed (see Figure 9) can be used to increase robustness and overcome unwanted variations on mesh. Finally, in Section 5 we compared our GPU implementation against the CPU implementation and shown great speedup in terms of timing.

Radius	CPU smoothing	GPU smoothing
64	15,756	0,160
32	10,842	0,117
16	8,361	0,106
8	7,192	0,101
4	6,599	0,093
2	6,318	0,083

Table 4: Smoothing in a 2048×2048 texture using various radius settings. Listed times are in seconds. We used model of Stanford Dragon with 500 000 triangles.

Overall, in this paper we have proposed a parallel method for computing ray casting, outliers removal and smoothing steps of Shape Diameter Function that is performed on GPU using OpenCL. We have maintained the accuracy of the results while noticeably increasing it's speed.

References

[1] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[2] Andrea Baldacci. Gpu - accelerated shape diameter function filter for meshlab, 2011.

[3] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[4] Hyeong In Choi, Sung Woo Choi, and Hwan Pyo Moon. Moon: Mathematical theory of medial axis transform. *Pacific J. Math*, 1997.

[5] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical En-

gineering, Czech Technical University in Prague, November 2000.

[6] Maurizio Kovacic, Fabio Guggeri, Stefano Marras, and Riccardo Scateni. Fast Approximation of the Shape Diameter Function. *Proc. Workshop on Computer Graphics, Computer Vision and Mathematics (GraVisMa)*, Vol. 5, 2010.

[7] Peter Moller-Nielsen Kristof Rmisch. *Sparse Voxel Octree Ray Tracing on the GPU*. Ph.d. thesis, Department of Computer Science, Aarhus University, Denmark, September 2009.

[8] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 55–63, New York, NY, USA, 2010. ACM.

[9] Martin Madaras and Roman Ďurikovič. Skeleton texture mapping. In *Proceedings of the 28th Spring Conference on Computer Graphics*, SCCG '12, pages 121–127, New York, NY, USA, 2013. ACM.

[10] R. Marques, C. Bouville, M. Ribardire, L. P. Santos, and K. Bouatouch. Spherical fibonacci point sets for illumination integrals. *Computer Graphics Forum*, 32(8):134–143, 2013.

[11] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, October 1997.

[12] Khronos OpenCL and Aaftab Munshi. The opencl specification version: 1.0 document revision: 48, 2013.

[13] Daniel Schweri Philippe C.D. Robert. Gpu-based ray-triangle intersection testing. Technical report, Research Group on Computational Geometry and Graphics, Institute of Computer Science and Applied Mathematics, University of Bern, 2004.

[14] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, July 2002.

[15] Xavier Rolland-Nevière, Gwenaël Doërr, and Pierre Alliez. Robust diameter-based thickness estimation of 3d objects. *Graphical Models*, 75(6):279–296, 2013.

[16] Lior Shapira, Ariel Shamir, and Daniel Cohen-Or. Consistent mesh partitioning and skeletonisation using the shape diameter function. *Vis. Comput.*, 24:249–259, March 2008.