Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 535–544
DOI: 10.15439/2022F92
ISSN 2300-5963 ACSIS, Vol. 30

Hierarchical data structures in rendering scenes containing a massive number of light sources

Andrzej Lamecki, Krzysztof Kaczmarski, Joanna Porter-Sobieraj
Warsaw University of Technology, Faculty of Mathematics and Information Science
ul. Koszykowa 75, 00-662 Warszawa, Poland
Email: {andrzej.lamecki.stud, krzysztof.kaczmarski, joanna.porter}@pw.edu.pl

***Abstract*—In order to speed up the process of rendering scenes containing many light sources, spatial data structures are used, which allow the number of lights processed for each pixel to be reduced during lighting computation. Examples of algorithms using such data structures are clustered shading and hybrid lighting. Alongside the rendering time, it is important to consider memory consumption resulting from processing a large number of lights. This paper presents a novel modification of the hybrid lighting algorithm using an octree that allows for a significant reduction in the amount of memory required to store the data structure.**

The proposed modification uses an octree to store the information about the rendered space. Detailed analysis of the proposed algorithm, and numerical results obtained for various 3D scenes, as well as different input data, all prove that the proposed method significantly reduces the memory required to store lists of lights used by the algorithm.

I. Introduction

IN RECENT years during the creation of virtual scenes, a lot of emphasis has been put on rendering visually realistic scenes. Rendering scenes containing multiple light sources requires calculating for each pixel a list of lights that affect its color. The most commonly used type of light is a point light with a limited range. Such lights can be represented as spheres in a rendered scene. The process of rendering a scene containing multiple light sources is therefore equivalent to determining for each rendered point which spheres contain this point. Fig. 1 shows an example scene for this problem containing 1 000 000 lights.

In the case of scenes with a large number of lights a naive approach of checking the distance between each rendered point and each light source requires a large number of operations. Improving rendering performance can be achieved by parallelization and by using spatial data structures to approximate light distribution in the scene space. These data structures are used during lighting computations to reduce the number of lights that are processed for each rendered point, which results in lower rendering times.

Another major concern in the rendering process, alongside the number of performed operations, is the memory complexity of the algorithm. Operating with a large amount of memory can often create a bottleneck while processing and visualizing complex scenes.

Research funded by the grant of Faculty of Mathematics and Information Science no. 504/04628/1120, Warsaw University of Technology

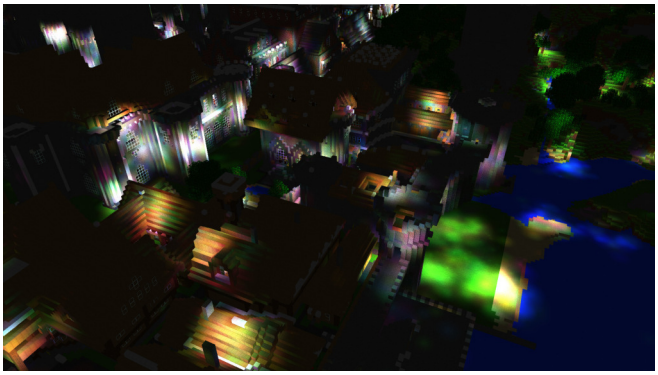

Fig. 1. A fragment of the *Rungholt* scene [1] containing 1 000 000 lights. All lights are point lights with limited range.

II. Related work

Research on the optimization of rendering has been conducted for over 50 years. Most of the algorithms used to efficiently render scenes with multiple lights use an additional data structure, describing scene geometry as well as lights placed in the scene, in order to decrease the time required for the calculation of lighting for each pixel. In these algorithms data structures allow a list of lights that potentially affect rendered geometry to be determined. The final decision on whether or not a light should affect a given pixel is made for each pair (pixel and light), based on the position of the shaded point and the position of the light source and, optionally, the normal vector in this point. The usage of the aforementioned data structures decreases the set of lights that are considered for a given pixel, which results in shorter rendering times.

One of the methods of rendering scenes containing many light sources is an algorithm using g-buffers [2], which are separate buffers used to store partial data, such as world position, normal vector or material properties, used in lighting computations in the final stage of frame rendering in which the final color displayed on the screen is calculated. Areas affected by lights are represented by spheres and each of those spheres is then rasterized onto the screen, and lighting data for each pixel in the area taken up by the light is updated. This solution is not efficient on modern GPUs [3] as rendering each frame requires multiple reads and writes to g-buffers, which significantly slows down the overall rendering process.

One of the most popular algorithms used to render scenes containing many light was the tiled shading algorithm [4], in which the rendered frame is divided into rectangular tiles (Fig. 2). Each tile is associated with a list of lights that affect any part of the scene contained in that tile. This additional data structure is used during final pixel color calculation. However, for each shaded pixel not all lights assigned to a corresponding list affect its color. Increasing the number of tiles can result in fewer lights to process for each pixel but processing more tiles requires additional operations and memory.

The part of the scene being rendered is divided only vertically and horizontally during tile construction. If a tile contains objects that are close to the camera and objects that are far from the camera (an example of such situation is presented in Fig. 3), lights affecting any of these objects will be processed for all of them. This can lead to many operations being performed unnecessarily as these lights are unlikely to affect all of the geometry contained in the tile.

There has been developed a modification of the tiled shading algorithm, a 2.5D culling [5], aiming to reduce the number of lights assigned to each tile in the case of a discontinuous geometry. For each tile, the geometry's range of depth is calculated and this range is divided equally into a fixed number of cells which contain the actual lists of lights.

An extension of the method of dividing space in three dimensions was proposed in the clustered shading algorithm [6]. In this algorithm all of the pixels of the rendered image are divided into groups (clusters) and a list of lights is assigned to each such cluster, in a similar way to the tiled shading algorithm. Clusters are created based on the three-dimensional position of shaded pixels as well as, optionally, a normal vector at that point. As described by Olsson et al. the depth of the rendered scene can be divided uniformly in either screen space or view space, or one can perform an exponential depth division in which resulting cells' dimensions are as equal as possible. During each frame of the rendering process, the lights are organized into a bounding volume hierarchy (BVH), a tree structure that allows for fast queries for all the lights that affect a given part of space. The tree is constructed in parallel, using the bottom-up approach. Then the bounding box of each cluster is used to determine a set of lights that possibly affect pixel samples in the cluster, and the normal vectors of cluster samples are used to discard lights that cannot affect any sample in the cluster.

There were also optimization attempts using the graphics pipeline to organize lights into lists assigned to different parts of the scene. The hybrid lighting algorithm [7] uses rectangular billboards to approximate the location of each light and to assign lights to appropriate lists. As with the clustered shading algorithm, the rendered space is divided into cells of a three-dimensional array. The billboards are rendered in a resolution corresponding to the vertical and horizontal array dimensions, and analytical calculations (the intersection of a sphere representing the area affected by the light and a ray originating from the camera) are performed to determine the range of cells in the depth affected by the light.

Complex spatial data structures can be implemented efficiently using graphics cards, allowing for parallel construction which results in lower building times. Tree structures are often used to organize points in space, e.g. for dealing with the level of detail [8] or multi-dimensional data clustering [9]. Trees allow the space to be divided into either regular cells [8], [9] or using hyper-planes which divide the space into two subspaces, each containing a subset of points [10].

The graphics pipeline has also been used to build this type of structure. Crassin et al. [11] describe a parallel algorithm for constructing an octree by inserting leaves into the partially constructed tree. To ensure that no two threads try to insert children nodes into the same parent node at the same time, a mutex for each node is used. The algorithm uses a separate buffer to store all nodes that cannot be inserted at a given moment and iterates until all of the nodes are inserted into the tree.

Another approach to rendering scenes with many light sources is taken in the tiled light trees algorithm [12]. Instead of the discrete division of the rendered space in all three dimensions, the lights are assigned to two-dimensional tiles, in a similar way to the tiled shading algorithm. However, in each tile a "light tree" (a variant of an interval tree), organizing lights in the depth of the scene, is constructed. During the final shading process for a given pixel, a tree from the tile in which the pixel is located is queried in the logarithmic time for the set of lights that can affect the pixel's color.

III. Algorithm description

The developed algorithm is based on the hybrid lighting algorithm [7]. The algorithm's major novelty is its utilization of an octree for lights' spatial organization combined with dynamic analysis of the scene: tree leaves are only created in the presence of a scene's geometry. In this way, the memory needed for the tree's representation is minimized. Another optimization extends the basic tree properties: lights can be stored not only in the leaves but also in the internal nodes. This allows the information about a light to be stored in a single node if it is assigned to the lists of all its children nodes. The tree's creation is therefore faster, since there is just one insertion operation instead of many.

One of the improvements in the tree's construction is the adaptive space division during the tree's creation, instead of equal division in all directions. A bigger tree is created but only a fragment of it is used while the rest, laying outside of the divider region, is ignored. An example of this approach is presented in Fig. 4. The number of tree cells in each direction is equal to the lowest power of 2 equal or greater to the highest array dimension. This solution does not add any significant computation or memory cost due to the sparse structure of the octree used in the implementation.

The octree is constructed in each frame, before the light assignment operation. This operation is split into two steps: determining a list of cells containing the scene's geometry present in the rendered frame and building an octree containing these cells. In order to calculate a list of unique cells, first -

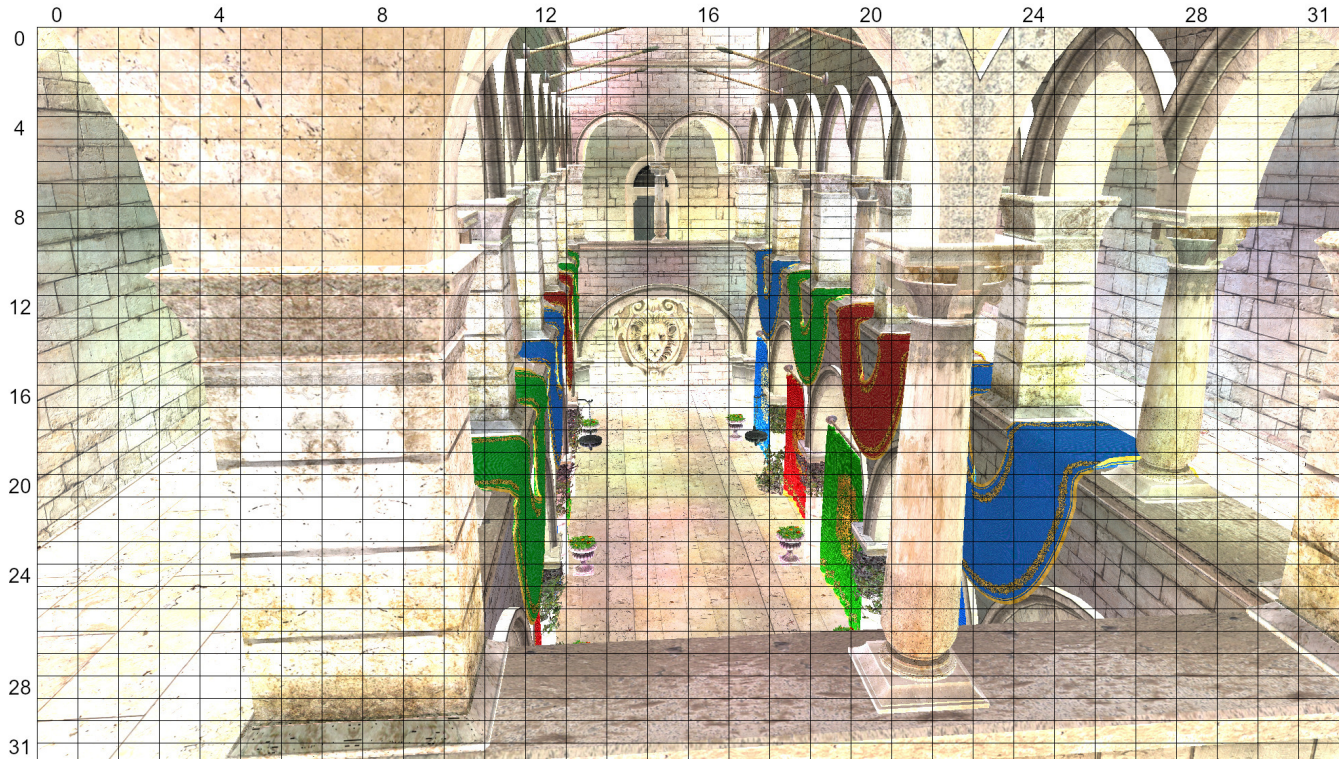

Fig. 2. Division of a rendered frame into 32 tiles vertically and 32 tiles horizontally (*Sponza* scene [1]).

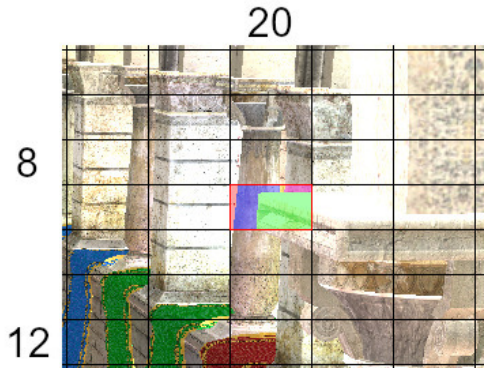

Fig. 3. Close-up of one of the tiles (outlined in red) of an image rendered using the tiled shading algorithm (*Sponza* scene [1]) in which there are many geometry discontinuities. Continuous parts of the geometry are highlighted in the same color.

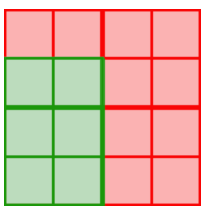

Fig. 4. Division of the space to 2×3 cells, based on 4×4 quadtree. Green cells represent a space fragment, red cells are outside and are ignored.

for each pixel of a rendered scene - the cell which the pixel belongs to is determined and its index is stored in a list. Then, repetitive values are removed from the list.

Two methods of removing repetitive elements have been proposed. The first method uses two functions commonly used in parallel computing: sorting and removing consecutive repeating elements on a list. One possible optimization of this process entails also removing consecutive repeating elements from the list before sorting. This optimization exploits the fact that all cell indices are written to the list row by row and many consecutive indices on the list are equal. This results in a certain amount of the list's elements being removed, which reduces the time needed to sort the entire list.

The second method of removing repeating elements from the list exploits a fact that all indices are from a small, finite range, bounded by the array dimensions. This makes it possible to use bitmasks to store the information about the presence of a cell in a frame. Moreover, the cells' indices are correlated with the two-dimensional tile of the image the pixel belongs to. This fact allows for easier parallel processing of the cells: all pixels from a given tile are processed by threads from the same warp, and the atomic operations (synchronized between threads in a single warp) are used to mark the presence of a given cell. The bitmasks are therefore stored in a separate list, each 32-bit element of the list representing 32 consecutive cells within the same two-dimensional tile. In the list containing the bitmasks, a parallel scan counting the

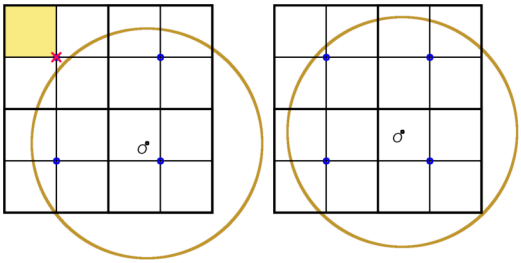

Fig. 5. The process of checking if a light can be assigned to the inner node of a quadtree. The node represents a group of 4×4 cells. The distance is checked from the light source O to each cell corner marked in blue. The light's range is represented by the yellow circle. In the example shown in the figure on the left, one of the points (marked in red) is farther from the light source than the light's range, therefore this light isn't assigned to the leaf marked in yellow so it cannot be assigned to the processed node. In the example shown in the figure on the right, all marked points are within range of the light, thus this light can be assigned to the processed node.

number of set bits is performed to determine the starting index in the resulting list under which the cells' indices should be written. In the final step, the list of unique cells present in the rendered frame is filled using the bitmask list.

The approach to building an octree described by Crassin et al. [11] has been adapted to the `CUDA` framework. All of the cells from the list computed in the previous step are inserted into the tree. This tree is then used during the assignment of lights to lists associated with each cell.

The process of assigning lights to lists is similar to the original algorithm, in which the lights' locations and ranges are approximated by billboards. The main difference is that after determining the cells which the light should be assigned to, for each cell a path in the octree (from the root to the leaf corresponding to the cell) is traced to check whether or not the light can be assigned to one of the internal nodes. For each node on this path, the distance from the light source to the innermost corners of the outermost cells of a group is compared to the light range. If all of the corners are within the range of the light, a light is assigned to this inner node and the rest of the path to the leaf is ignored. A 2D example of this process is shown in Fig. 5. All the calculations are performed in view-space because then the corner of each of the cells has a constant position. Moreover, it is possible to calculate the view-space position of each cell corner in advance and read it from the additional buffer instead of calculating it every time it is used.

During the final shading process the lights are read from all the lists corresponding to nodes on a path from the octree root to the leaf representing the cell the shaded pixel is in.

IV. Results

The described algorithm was subjected to a series of tests to determine its effectiveness in comparison to the clustered shading and hybrid lighting algorithms. We concentrated on comparing the memory required by algorithms, as minimizing the memory requirements was the main purpose of the proposed modification. All reported results were obtained on a PC

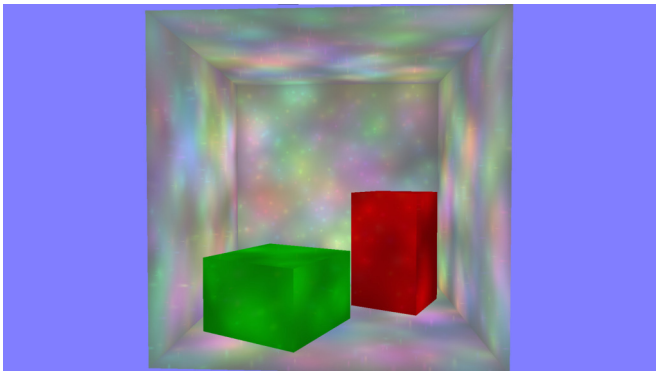

Fig. 6. *Cornell Box* test scene with 2 000 lights spaced uniformly in the scene volume.

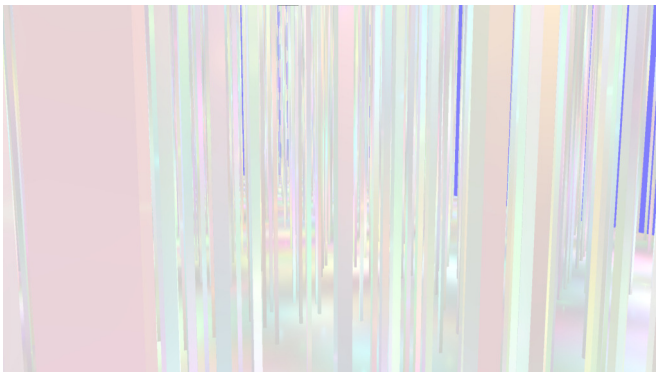

Fig. 7. *Bars* test scene with 50 000 lights spaced uniformly in the scene volume.

with an Intel Core i7-9750H CPU 2.6 GHz and 16 GB RAM, supplied with an NVIDIA GeForce RTX 2060 Mobile (1920 CUDA cores) 6 GB GDDR6 with CC 7.5. All algorithms were implemented using `C++17` with `Direct3D 11` [13] and `CUDA` [14] libraries and were implemented for the deferred shading pipeline.

The algorithms were tested on 4 different scenes with 3 different distributions of light positions. Images were rendered at a resolution of 1920×1080 pixels. In each test the camera moved along a predetermined path. To minimize the impact of the operating system's background work on rendering times, each test was repeated 5 times, and the results were averaged for each frame.

Three of the four test scenes, *Cornell Box*, *Sponza* [1] and *Rungholt* [1], are commonly used as a benchmark for evaluating rendering algorithms, and the other one, *Bars*, contains many vertical bars, and was created to test the algorithms on a scene containing many geometry discontinuities. All scenes, along with the light distribution used in them, are shown in Figs. 6, 7, 8 and 9.

The lights in all test scenes were distributed randomly, using one of three distributions: uniform distribution in the scene's volume; uniform distribution on the scene's geometry; groups of lights distributed uniformly in the scene's volume. The

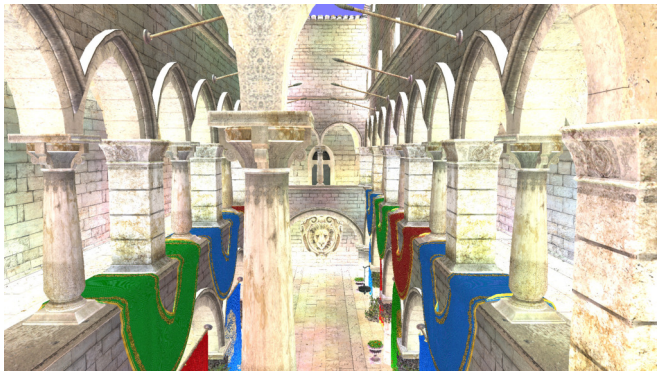

Fig. 8. *Sponza* [1] test scene with 50 000 lights spaced uniformly on the scene's geometry surface.

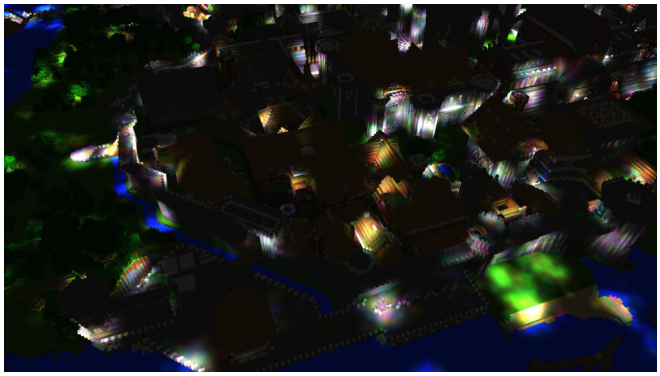

Fig. 9. *Rungholt* [1] test scene with 1 000 groups each containing 1 000 lights spaced uniformly in the scene volume.

lights' ranges were generated based on a uniform distribution between a minimum and maximum range, different for each scene. For the uniform distribution around the scene's volume, each coordinate of the light's position was generated independently of another, based on the scene's bounding box. In the uniform distribution around the scene's geometry, a random triangle from a fixed number (1 000 in the case of the performed tests) of the biggest triangles was selected at random, with the probability of being chosen proportional to the triangle's area. Then, a random point on a chosen triangle was generated and a light was placed in a position within the predefined distance of the chosen point along the triangle's normal vector. To generate groups of lights, the positions of the groups' centers were generated around the scene's volume. Then, for each group, a fixed number of lights' positions were generated using the truncated normal distribution [15] with the expected value equal to the generated group's center.

A number of tests were performed to determine the impact of each parameter on the average rendering time of each scene. In each test case, parameter configurations differed by a single parameter. As a base configuration we assumed:

- the division of the rendered image into 30×17 tiles;
- exponential depth division in the view-space;
- the unique cell list determination method using bitmasks;

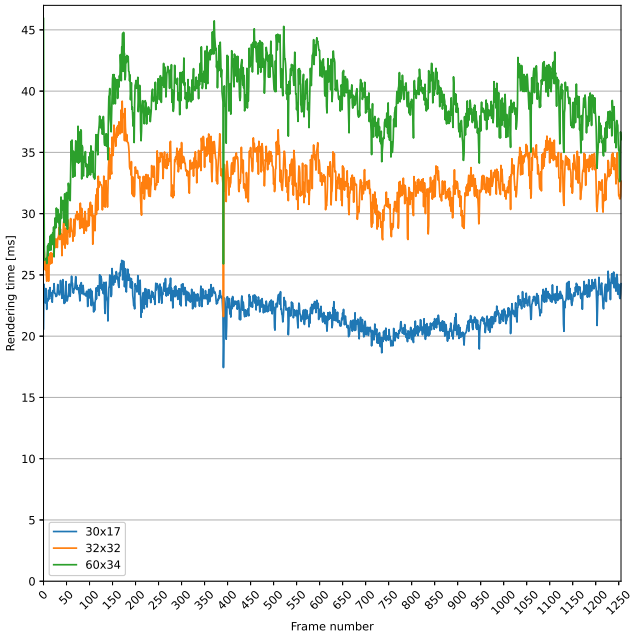

Fig. 10. Rendering time for each frame of the *Bars* test scene with different tile counts.

- precomputing the positions of cell corners.

Two different tile counts were compared with the base one: 32×32 and 60×34 tiles. In all test scenes, the lowest average time was achieved in the configuration using 30×17 tiles. The rendering times for each frame of the scene *Bars* are shown in Fig. 10. Table I shows the average results for all test scenes.

Two configurations with uniform depth division in the screen-space were tested: into 128 and into 256 cells. In the case of the *Cornell Box* and *Sponza* scenes, notably higher average rendering times were seen in the configuration with a higher cell number, whereas in the other two scenes a higher cell count resulted in lower average rendering times. Uniform depth division into 128 and 256 cells in the view-space was also tested. As with the screen-space division, whose cell count resulted in lower average rendering times, the rendering times differed between scenes. Notably, for the *Rungholt* scene the differences between the average rendering times were negligible. Table II shows the averaged results for all of the four described depth division methods for all test scenes.

Figs. 11, 12, 13 and 14 show the rendering times of each frame for the three tested methods of depth division. For the uniform divisions, a division resolution resulting in the lowest average rendering times was chosen. For the *Bars*, *Sponza* and *Rungholt* scenes, screen-space division resulted in significantly higher rendering times than the exponential division in the view-space. Uniform division in the view-space resulted in similar rendering times to exponential division for the *Cornell Box* and *Rungholt* scenes, but the rendering times for uniform division were higher than the exponential one for the *Bars* and *Sponza* scenes. In the case of the *Rungholt* scene

TABLE I
AVERAGE TIME [MS] FOR FRAME RENDERING USING THE DESCRIBED ALGORITHM FOR DIFFERENT IMAGE TILE COUNTS.

Test scene \ Tile count	30×17	32 × 32		60 × 34	
Cornell Box	11.07	15.61	+40.9%	18.71	+69.0%
Bars	22.37	32.59	+45.7%	38.96	+74.1%
Sponza	78.02	143.42	+83.8%	197.19	+152.8%
Rungholt	63.32	84.10	+32.8%	108.58	+71.5%

TABLE II
AVERAGE TIME [MS] OF FRAME RENDERING USING THE DESCRIBED ALGORITHM FOR FOUR DIFFERENT DEPTH DIVISIONS.

Test scene \ Cell count in depth	128 (screen-space)	256 (screen-space)	256 (view-space)	256 (view-space)
Cornell Box	12.02	14.39	12.46	11.43
Bars	40.97	39.78	20.03	26.23
Sponza	99.44	106.25	60.14	73.66
Rungholt	149.20	136.82	61.16	61.19

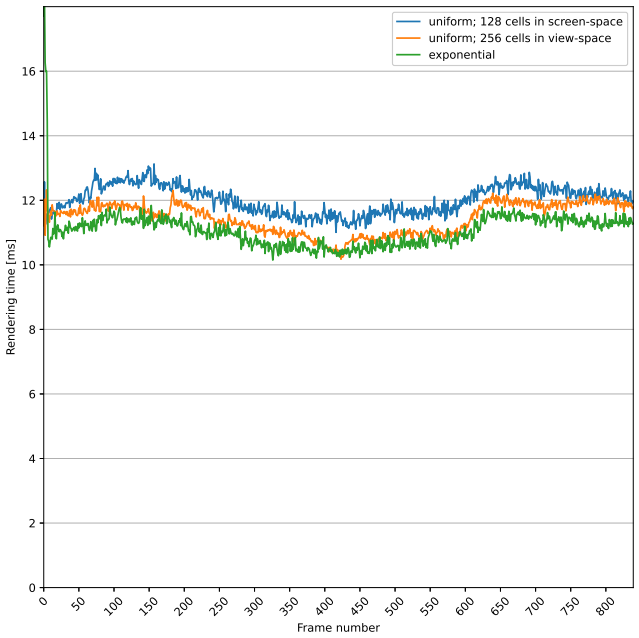

Fig. 11. Rendering time of each frame for the *Cornell Box* test scene for the three depth division methods.

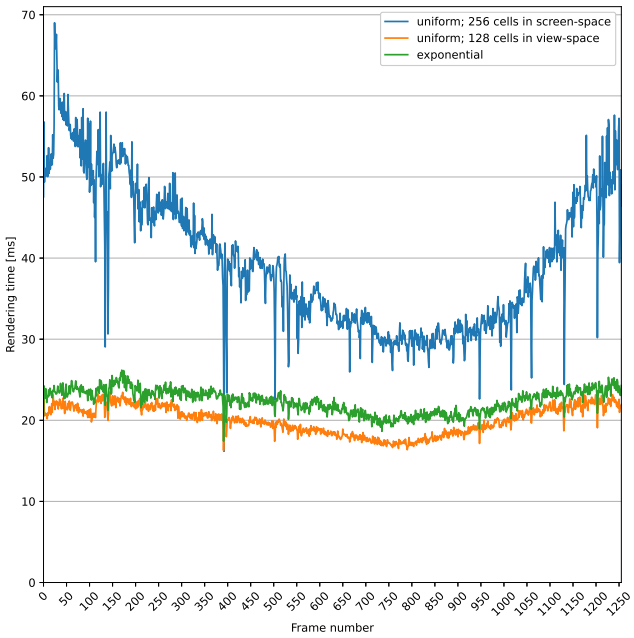

Fig. 12. Rendering time of each frame for the *Bars* test scene for the three depth division methods.

(Fig. 14), there were three parts of the test (at the beginning, in the middle and at the end), in which screen-space division resulted in significantly higher rendering times compared with the other methods. In these parts of the test, the camera was far from the scene geometry and a large area of the scene was visible.

Three methods of obtaining the list of cells filled with geometry were compared: using bitmasks, sorting and removing consecutive repeating elements, and removing consecutive repeating elements before sorting. These configurations were tested on the *Bars*, *Sponza* and *Rungholt* scenes. Fig. 15 shows the obtained results for the *Rungholt* scene. Table III shows

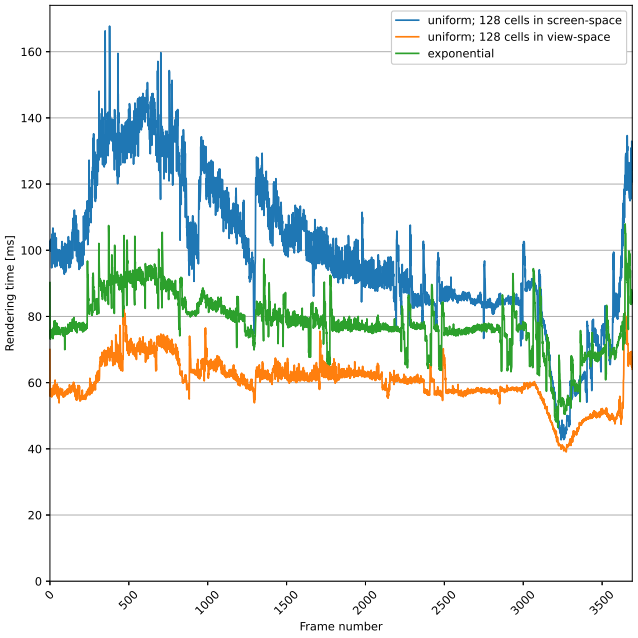

Fig. 13. Rendering time of each frame for the *Sponza* test scene for the three depth division methods.

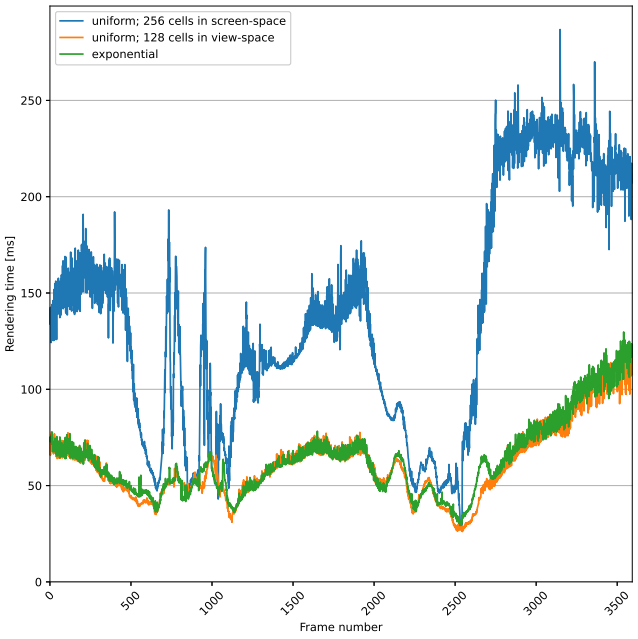

Fig. 14. Rendering time of each frame for the *Rungholt* test scene for the three depth division methods.

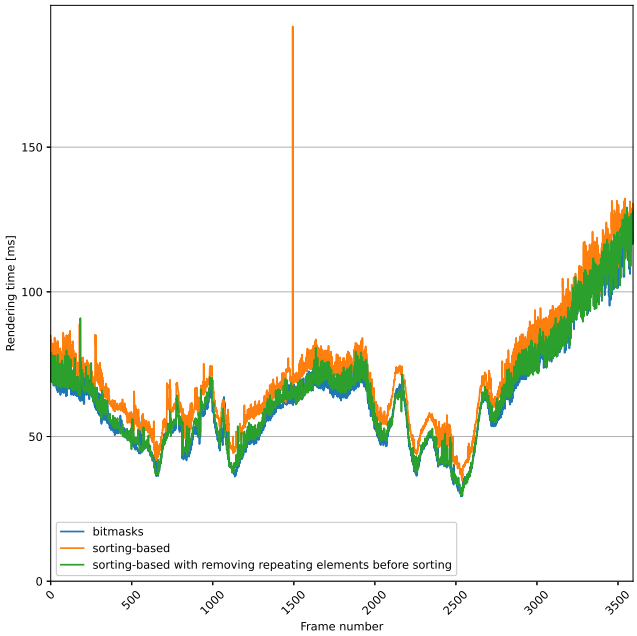

Fig. 15. Rendering time of each frame for the *Rungholt* test scene for the three methods of obtaining the list of unique cells.

the averaged results for all the test scenes. In all cases the lowest average time was achieved using the bitmasks and the highest time was seen using the sorting-based method without removing the repeating elements beforehand.

A version of the algorithm in which the cells' corner positions were precomputed was also compared with a version in which the positions were calculated each time they were used. These configurations were also tested on the *Bars*, *Sponza* and *Rungholt* scenes. For all of these scenes, precomputing the cells' corner positions resulted in significantly lower average rendering times than calculating them every time. Fig. 16 shows the results for the *Sponza* scene. Results for all of the tested scenes are shown in Table IV.

The proposed algorithm was compared in terms of rendering time and memory occupancy with the clustered shading and the original hybrid lighting algorithms. The algorithms were compared using the *Sponza* and *Rungholt* scenes with a variable number of lights, with lights being placed on the scene's geometry in the *Sponza* scene, and with groups of lights in the *Rungholt* scene. Two configurations of the hybrid algorithm and the octree-based modification were tested, differing in the number of tiles used to divide the rendered image: 30×17 and 60×34. This resulted in tiles of 32×32 and 64×64 pixels respectively. In the case of the clustered shading algorithm, tile sizes of 16×16 and 32×32 pixels were used. Depth was divided using the exponential division method, and a configuration of the clustered shading algorithm without using normal vectors during cluster creation was chosen.

Figs. 17 and 18 show the average rendering times for each tested algorithm variant. The octree-based algorithm

TABLE III
AVERAGE TIME [MS] OF FRAME RENDERING USING THE DESCRIBED ALGORITHM FOR THE THREE METHODS OF OBTAINING THE LIST OF UNIQUE CELLS.

List obtaining method / Test scene	bitmasks	sorting-based		sorting-based with the removal of repeating elements before sorting	
Bars	22.37	29.13	+30.2%	24.07	+7.6%
Sponza	78.02	84.29	+8.0%	78.78	+1.0%
Rungholt	63.32	70.66	+11.6%	64.55	+1.9%

TABLE IV
AVERAGE TIME [MS] OF FRAME RENDERING USING THE DESCRIBED ALGORITHM FOR TWO METHODS USED TO CALCULATE CELLS' CORNER POSITIONS.

Cells' corner position calculation method / Test scene	precomputing	calculating each time	
Bars	22.37	26.56	+18.8%
Sponza	78.02	95.66	+22.6%
Rungholt	63.32	69.05	+9.1%

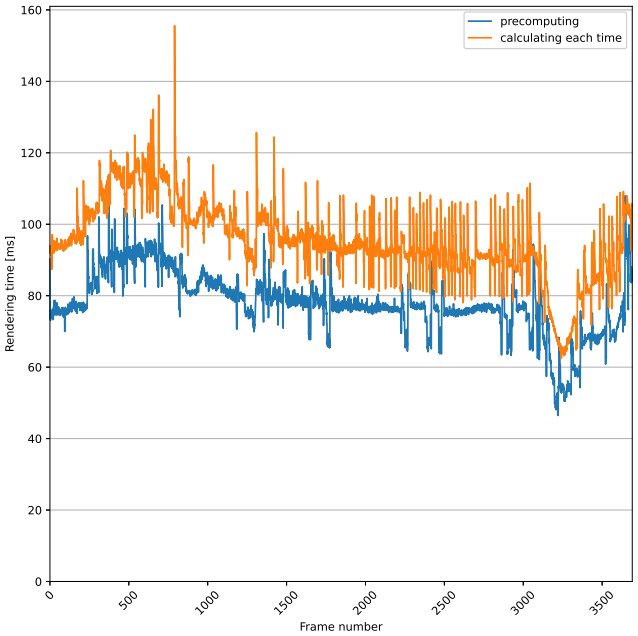

Fig. 16. Rendering time of each frame for the *Sponza* test scene depending on the method used to calculate the cells' corner positions.

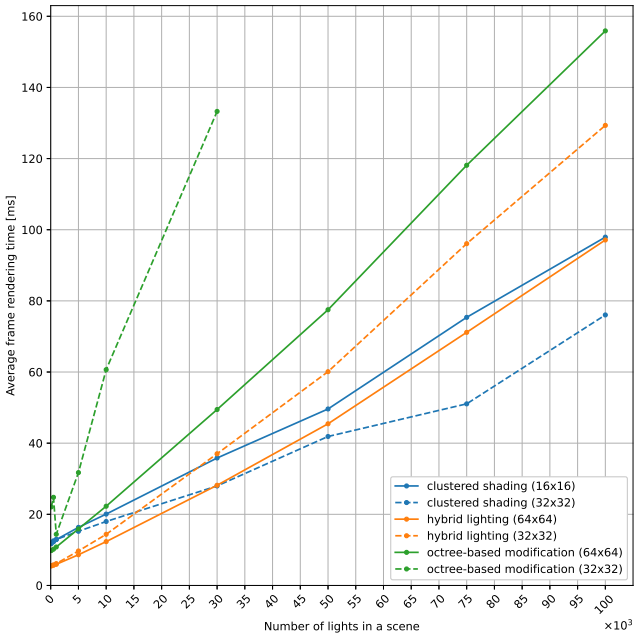

Fig. 17. Average rendering time of the *Sponza* scene depending on the total number of lights in the scene, for each tested algorithm. The tile size, in pixels, is written in parentheses.

wasn't tested for the *Sponza* scene for the light count above 30 000 because of a rapidly rising rendering time for the increasing number of lights. In both test scenes the octree-based algorithm with a tile size of 32×32 saw significantly higher (by $11\% - 392\%$) rendering times than the rest of the algorithms. The configuration with tiles of size 64×64 pixels, for up to 5 000 lights for the *Sponza* scene and up to 90 000 for the *Rungholt* scene, achieved lower (by $16\% - 17\%$) rendering times than both configurations of the clustered shading algorithm. For higher numbers of lights, the octree-based algorithm was slower than the clustered shading algorithm by up to 131% for the *Sponza* scene and up to 75% for the *Rungholt* scene.

Figs. 19 and 20 show the average sum of lights on the lists of lights depending on the total number of lights in a scene. As each light can be assigned to more than one list, the sum of light list elements may be bigger than the number of lights in a scene.

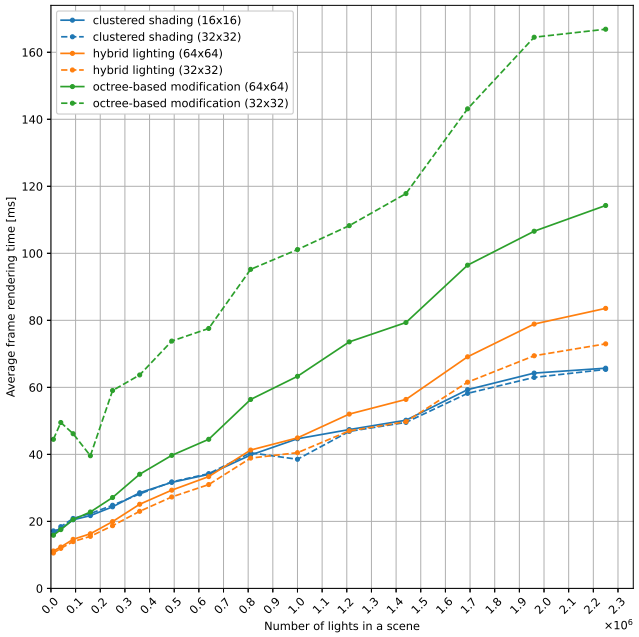

Fig. 18. Average rendering time of the *Rungholt* scene depending on the total number of lights in the scene, for each tested algorithm. The tile size, in pixels, is written in parentheses.

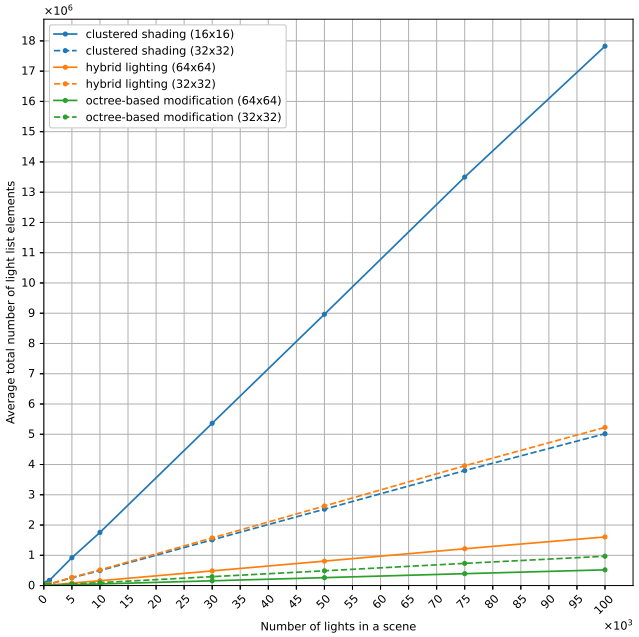

Fig. 19. Total number of light list elements averaged for all frames (*Sponza* scene) depending on the total number of lights in a scene, for each tested algorithm. The tile size, in pixels, is written in parentheses.

On average, the lowest total number of light list elements was achieved in the case of the octree-based algorithm with a tile size of 64×64 pixels. For the *Sponza* scene, there were at least 65% fewer elements compared with the other two algorithms, and for the *Rungholt* scene – at least 23% fewer. In the case of the *Sponza* scene, the proposed method with tiles measuring 32×32 pixels resulted in a lower total number of list elements (by $30\% - 40\%$) than in the case of the original, unmodified version of the hybrid lighting algorithm with tiles that were twice as big.

V. Summary

The results obtained in the tests show that both the scene geometry and the lights' distribution are important factors impacting the rendering time.

Using an octree to store the lists of lights allows for a significant reduction (up to 65%) in the number of elements on the lists compared with the other tested algorithms. This resulted in fewer list insertion operations that needed to be performed for each frame. However, additional checks performed in order to determine whether or not a light could be stored in an internal node resulted in an overall slower algorithm than the original version.

The approach of using billboards to approximate the lights' positions and ranges resulted in many calculations that were repeated by multiple threads. As each billboard's pixel is potentially processed by a different thread, each thread has to check if the light affects all nodes in a group independently of other threads. One idea for modifying the described algorithm

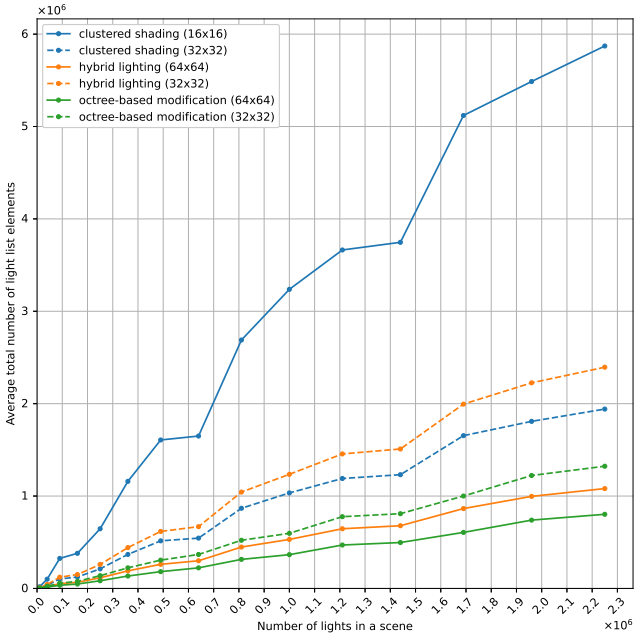

Fig. 20. Total number of light list elements averaged for all frames (*Rungholt* scene) depending on the total number of lights in a scene, for each tested algorithm. The tile size, in pixels, is written in parentheses.

is to adapt this method so that one light is processed entirely by a single thread.

Another modification that could result in shorter rendering times is to replace the sparse octree representation with full three-dimensional arrays, each representing one octree level. This modification would make it possible to read information about any node, without the necessity of tracing a path from the octree's root.

Another aspect of rendering scenes with many lights is accounting for multiple shadow sources. Shadow rendering poses a serious challenge, especially in the presence of many light sources, as both the rendering times [16] and shadow quality [17] have to be considered.

REFERENCES

[1] M. McGuire, "Computer graphics archive," July 2017. [Online]. Available: https://casual-effects.com/data

[2] A. Lauritzen, "Deferred rendering for current and future rendering pipelines," *SIGGRAPH Course: Beyond Programmable Shading*, pp. 1–34, 2010.

[3] O. Olsson, E. Persson, and M. Billeter, "Real-time many-light management and shadows with clustered shading," in *ACM SIGGRAPH 2015 Courses*, 2015, pp. 1–398.

[4] O. Olsson and U. Assarsson, "Tiled shading," *Journal of Graphics*, vol. GPU, pp. 235–251, 11 2011.

[5] T. Harada, "A 2.5 d culling for forward+," in *SIGGRAPH Asia 2012 Technical Briefs*, 2012, pp. 1–4.

[6] O. Olsson, M. Billeter, and U. Assarsson, "Clustered deferred and forward shading," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Citeseer, 2012, pp. 87–96.

[7] J. Archer, G. Leach, P. Knowles, and R. van Schyndel, "Hybrid lighting for faster rendering of scenes with many lights," *The Visual Computer*, vol. 34, no. 6, pp. 853–862, 2018.

[8] J. Dupuy, J.-C. Iehl, and P. Poulin, *Quadtrees on the GPU*, 10 2018, pp. 211–222.

[9] D. Wehr and R. Radkowski, "Parallel kd-tree construction on the gpu with an adaptive split and sort strategy," *International Journal of Parallel Programming*, vol. 46, no. 6, pp. 1139–1156, 2018.

[10] J. R. Jørgensen, K. Scheel, and I. Assent, "Gpu-inscy: A gpu-parallel algorithm and tree structure for efficient density-based subspace clustering." in *EDBT*, 2021, pp. 25–36.

[11] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel cone tracing," in *Computer Graphics Forum*, vol. 30, no. 7. Wiley Online Library, 2011, pp. 1921–1930.

[12] Y. O'Donnell and M. G. Chajdas, "Tiled light trees," in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2017, pp. 1–7.

[13] Microsoft, "Direct3d 11 website," 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11

[14] NVIDIA Corporation, "Cuda toolkit website," 2022. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[15] J. Burkardt, "The truncated normal distribution," *Department of Scientific Computing Website, Florida State University*, pp. 1–35, 2014.

[16] O. Olsson, E. Sintorn, V. Kämpe, M. Billeter, and U. Assarsson, "Efficient virtual shadow maps for many lights," in *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 87–96. [Online]. Available: https://doi.org/10.1145/2556700.2556701

[17] K. Kluczek, "Quality metric for shadow rendering," in *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2016, pp. 791–796.