Comparison of hierarchies for occlusion culling based on occlusion queries

V.I. Gonakhchyan
pusheax@ispras.ru
Ivannikov Institute for System Programming of the RAS, Moscow, Russia

Efficient interactive rendering of large datasets still poses a problem. Widely used algorithm frustum culling is too conservative and leaves a lot of hidden objects in view. Occlusion culling with hardware occlusion queries is an effective technique of culling of hidden objects inside view. In the paper, we perform the comparative analysis of popular indexing techniques in conformity to occlusion culling.

Keywords: *Occlusion culling, occlusion query, opengl draw call.*

1. Introduction

Occlusion culling algorithms remove occluded objects to enhance frame buffer composition speed. In the paper, object is defined as a mesh that can be rendered using one draw call. Two categories of invisible objects are illustrated in fig. 1: occluded objects and objects beyond camera frustum. Frustum culling quickly discards objects beyond camera frustum but leaves occluded objects inside the camera. We consider the algorithm to find occluded objects. The goal of this paper is to analyze how different hierarchies for scene organization affect occlusion culling and rendering performance.

Occlusion culling algorithms are described in-depth in seminal papers [2][5]. We are interested in online algorithms based on hardware occlusion queries that are widespread today.

Some of the previous algorithms were implemented using OpenGL 2 which is using the immediate rendering mode [10]. In immediate mode all of the geometry is sent each frame resulting in CPU-GPU bandwidth bottleneck and driver overhead caused by excessive number of commands in driver queue. Display lists were used to compile multiple rendering commands once and reduce driver overhead. OpenGL 3 introduced retained rendering mode and immediate mode techniques such as display lists were deprecated and later removed from specification. It raised the question of effective use of object indexing techniques in modern OpenGL. Previously developers had to worry about the amount of geometry passed to GPU. Now developers have to think about driver overhead and state changes as well [18]. In this paper, we analyze performance considerations when using different hierarchies in retained rendering mode.

Although Vulkan API is out of the scope of this paper, it was shown that it gives a better multithreaded performance by utilizing all of the cores of CPU for rendering [3]. It is still based on the same hardware and using retained rendering mode so all of the results in the paper stand.

Hardware occlusion query is GPU technique to find visible faces of a polyhedron [6]. Visibility check stops when the first visible face is found. Checking query result in the frame queries were sent is a blocking function call and it causes CPU starvation. Visibility query result is available without noticeable delay in the next frame because by that time all draw calls required for query execution were sent. So visibility information in given frame is based on the previous frame.

Space coherency is a relationship between nodes in which one node's visibility determines the visibility of other nodes. For example, if building is invisible then objects inside building are also invisible. Time coherency determines visibility in the future by visibility at given time. For example, if object is visible in the given frame then we can consider it visible some number of frames and avoid sending expensive queries.

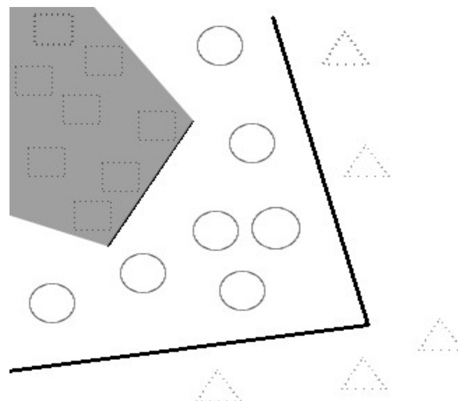

Fig. 1. Different types of culled objects. Frustum culled objects are depicted as triangles. Occluded objects are depicted as rectangles.

2. Previous work

Greene et al. introduced occlusion culling algorithm based on hierarchical z-buffer [7]. Z-buffer is stored as pyramid structure. Z-buffer is divided into 4 components, each one having maximum z value in its region. The process is repeated until pixels are reached. Z-pyramid allows for quick triangle culling by comparing minimum z value of triangle and z value in the corresponding region. This method can be implemented in hardware and software. Software implementation requires expensive rasterisation and pyramid structure updates on CPU. In hardware implementation (ATI Hyper-Z) triangles still have to be transformed and rasterized on GPU which does not replace quick method for geometry culling with a hierarchical structure.

Bittner et al. considered ways for optimal usage of occlusion queries for hierarchical scenes based on extension NV_occlusion_query [1]. Previous frame visibility results are used in the next frames. Main performance problems come from CPU starvation and GPU starvation. Visibility results are checked in the next frame to remove CPU starvation. Previously visible objects are rendered at the beginning of the current frame to remove GPU starvation. Authors used kD-tree constructed according to the surface-area heuristic [11]. Visibility queries are sent for leaf nodes and results are propagated to upper hierarchy levels.

Guthe et al. observed that in many cases visibility queries make performance worse than frustum culling, proposed probability criterion to minimize the number of queries, performance model which helps to avoid queries when rasterisation is cheaper [9]. Mattausch et al. suggested further ways of minimizing the number of queries like sending one query for group [12]. Authors used p-hbvo (polygon-based hierarchical bounding volume decomposition) [13], which is

well-suited for static scenes and expensive to maintain for dynamic scenes.

Software rasterisation and visibility checks can be used instead of hardware occlusion queries [4]. First, triangles of significant occluders are rendered on CPU, hierarchical z-buffer of specified resolution is created. Then bounding boxes of objects or hierarchy nodes are rasterized to determine their visibility against created z-buffer. This helps to avoid expensive occlusion query read-back but requires occluder selection that is best done manually. Also, occluder rasterisation can be expensive and low level-of-detail models give only approximate results.

Scene preprocessing can be effectively applied to static scenes. Teller et al. proposed to use BSP tree with decomposition along axes for architectural scenes [17]. Achieved hierarchy corresponds to room structure in a building. Visibility information is stored as the graph with rooms and portals. Room visibility can be determined by rasterizing portals. However, that graph is expensive to compute and works effectively only for static scenes. Commercial solution Umbra computes voxel representation of a scene [14]. Empty voxels serve as portals between different parts of a scene. Software rasterisation of portals determines the visibility of parts of a scene. That algorithm effectively finds occluded objects for static 3d scenes and is widely used in video games.

Greene introduced image space algorithm based on precomputed occlusion masks [8]. As input, it takes a list of polygons in front-to-back order. It recursively subdivides image space into quadtree until visibility of polygon can be determined for each quadrant. Main advantages of this approach are small memory requirements and no pixel overwrites. However, it requires special hardware to implement efficiently. Zhang et al. proposed visibility culling based on hierarchical occlusion maps, which is better suited for modern hardware [19]. It constructs occlusion map hierarchy by rendering chosen occluders and then traverses bounding volume hierarchy of the model database to perform visibility culling. The algorithm allows for approximate visibility when opacity threshold is set to value lower than one. The main disadvantage of the algorithm is the sophisticated process of occluder selection which favors large objects with small polygon count for faster construction of occlusion map hierarchy.

3. Occlusion culling algorithm

Hierarchical occlusion culling algorithm in this paper is based on Coherent hierarchical culling [1]. Although our implementation does not include query batching, tight bounding volumes, probabilistic estimation of visibility, it allows comparing benefits and limitations of different subdivision hierarchies.

Pseudocode of the algorithm:

```
function RenderFrame(rootNode, frustum,
queries, sentNodes)
   PerformFrustumCulling(frustum,rootNode)

   for i in 0..queries.size-1
      vis <- GetQueryResult(queries[i])
      SetNodeCulled(sentNodes[i],vis == 0)
   end for
   PropagateVisibilityUpHierarchy()

   nodes <- GetVisibleNodesInFrustum()
   for n in nodes
      for inst in n
         if !InstRendered(inst)
            Render(inst)
         SetInstRendered(inst)
      end for
   end for

   sentNodes <- GetLeafNodesInFrustum()
   for i in 0..sentNodes.size-1
      SendQuery(queries[i],sentNodes[i]
         .boundingBox)
   end for

end function
```

Function "RenderFrame" renders one frame. Function "PerformFrustumCulling" recursively sets frustum culled bit for every node outside the frustum. First for loop checks visibility results of occlusion queries sent in the previous frame. Second for loop renders objects of visible nodes in the frustum. Last for loop sends queries for all leaf nodes in the frustum.

During the first frame, all objects inside frustum are rendered, and all hierarchy leafs inside frustum are queried. During the second frame, query results are checked, and only visible objects are rendered. Hierarchy leafs inside frustum are queried each frame. We propagate visibility up the hierarchy to optimize performance of hierarchy traversal. Space decomposition hierarchies allow multiple nodes per object. Rendered state of each instance is stored in bit array to make sure that all objects are rendered only once.

Frame buffer composition time for a large number of objects can be approximated by the formula:

$$T_{frame} = T_{check} + T_{render} + T_{queries} ,$$

where T_{check} — time it takes to check query results,

T_{render} — time to render visible objects,

$T_{queries}$ — time it takes to send queries for leafs nodes inside frustum.

When the number of queries is small T_{render} is the bottleneck. When the number of queries is large $T_{queries} + T_{check}$ is the bottleneck. Let's rewrite the formula by expanding the terms:

$$T_{frame} = c_1 N_q + c_3 N_{obj} + c_2 N_q ,$$

where N_q — number of queries (leaf nodes inside frustum),

N_{obj} — number of visible objects inside the frustum. N_{obj} is the function of camera position and hierarchy height for the given object distribution.

Let's consider a common case where a scene is indexed by octree and camera is positioned outside the scene. In the worst case, three sides of the bounding box of the scene are visible. Assuming that objects are distributed uniformly across the bounding box of the scene, the number of objects per octree leaf equals $N_{total}/2^{3h}$, where N_{total} is the total number of objects in the scene, h is octree height. Then the number of visible objects approximately equals $3N_{total}/2^h$. We get the formula for frame duration that depends only on octree height and constants:

$$T_{frame} = (c_1 + c_2)2^{3h} + c_3 3N_{total} 2^{-h}.$$

Optimal octree height for given scenario is $\frac{1}{4}\log_2 \frac{N_{total} c_3}{c_1 + c_2}$. For example, for dataset 1 calculated octree height $h = 4$ gives the best performance in practice because dataset 1 can be described by that theoretical model.

LBVH and BVH SAH perform better on scenes where the density of objects is uneven. Better clustering helps to lower the number of queries to get visible objects inside the frustum.

4. Hierarchies

4.1 Octree

Octree is uniform space decomposition structure that uses three axis-perpendicular planes to simultaneously split the scene's bounding box into eight regions at each step [15]. When object's bounding box intersects the splitting plane, it is either

assigned to the internal node (single reference octree) or propagated below and assigned to multiple leaf nodes (multiple reference octree). Storing geometry in leafs increases clustering quality and as a result reduces number of visible objects. The downside is the increased number of occlusion queries, which are sent for every leaf in the frustum. We performed rendering performance comparison to find out which technique is more effective. Dataset 1 is small enough that GPU can handle rendering and queries quite efficiently (fig. 2). As a result, multiple reference octree gives the best time because of efficient clustering. Many objects intersect upper levels of octree resulting in redundant draw calls in case of single reference octree. Dataset 2 has non-uniform object distribution where several planes occupy half of the scene. It produced a lot of redundant leaf nodes and occlusion queries that degraded performance when storing objects in leafs. Dataset 3 has many buildings with tightly packed objects inside buildings. Even though the number of objects is large, it can be very efficiently subdivided requiring only small number of nodes. Overall, multiple reference octree provides the most efficient occlusion culling of large architectural scenes.

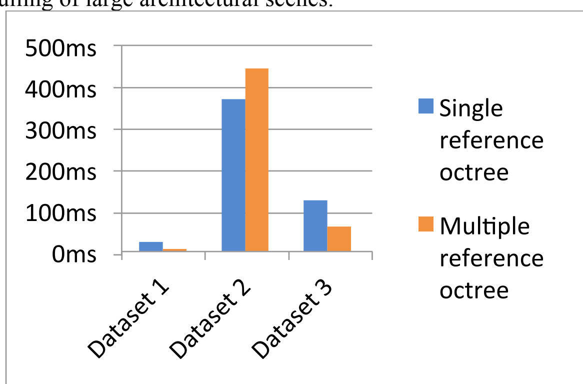

Fig. 2. Average frame rendering time for single and multiple reference octrees.

When rendering a visible node, contained object is skipped if it was already rendered in the current frame. Out of all hierarchical structures considered in the paper, octree gives the least effective clustering because of wasted space without any objects. Octree allows dynamic scenes because visibility results of octree nodes can be used for subsequent frames as they have fixed position in space. Also, it does not need to be rebalanced as other space decomposition hierarchies like kd-tree. Maximum octree level can be restricted depending on GPU performance.

4.2 LBVH

LBVH is primitive space decomposition hierarchy that is based on sorting along space filling curve [15]. All centers of object bounding boxes are sorted along Z space filling curve and grouped hierarchically from bottom to top [16]. LBVH achieves tighter clustering than octree and as a result less number of occlusion queries. Estimating the number of queries is simple because the number of leafs is determined at the start of the construction. LBVH cannot handle dynamic scenes because moving objects make occlusion queries for previously constructed LBVH useless in the current frame.

4.3 BVH

BVH with surface area heuristic for choosing the splitting plane to minimize the number of ray and bounding box intersection tests was developed for ray tracing, but it also gives efficient clustering of primitives for occlusion culling [13][15]. BVH construction is a top-down recursive process; on each step, we create two axis aligned bounding boxes. Triangles are sorted by the longest scene dimension and splitting plane with minimum cost is taken according to surface area heuristic.

Because of top-down construction, BVH SAH sometimes creates clusters that cannot be subdivided into two nodes. We try to subdivide such cluster for each axis in order, and in case of failure leave it as a leaf node.

We compare rendering performance when using BVH SAH for storing three types of primitives: objects, subdivided objects, triangles. Object is set of triangles that can be rendered with one draw call. Subdivided object is an object that was subdivided into multiple objects to achieve better triangle clustering. Storing each triangle as an object generates hierarchy with the best clustering. For rendering efficiency, large number of triangles is stored in a leaf node, render state changes are avoided when encountering triangles of the same object. Storing triangles in BVH gives many additional draw calls creating a CPU bottleneck (fig. 3). Storing subdivided objects gives much faster performance. However, clustering efficiency increase is not enough to cover for additional draw calls for considered datasets using simple shader. Let's consider the difference in rendering time of dataset 1 for objects (8.7ms) and subdivided objects (45.6ms). Even though object subdivision helped to lower average number of query calls N_q from 192 to 171, it raised the average number of draw calls N_{obj} from 4546 to 19452 (tests were conducted for $bvh\ height = 10$).

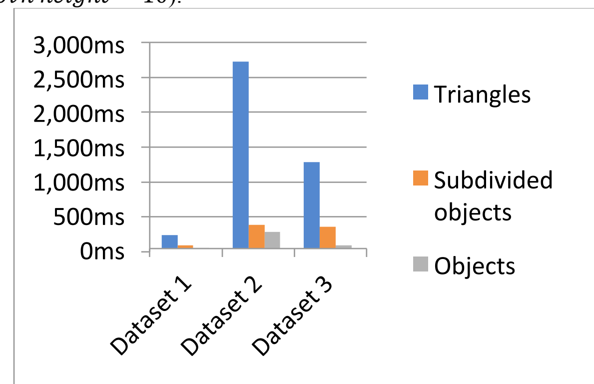

Fig. 3. Average frame rendering time for different types of primitive clusters when using occlusion culling on BVH SAH.

5. Performance comparison

5.1 Datasets

For rendering performance comparison we took three large datasets (figs. 4–6):

1. Dataset 1: 5,012,582 triangles, 50,521 objects. Building is tightly packed with objects having small variation in size.
2. Dataset 2: 10,827,713 triangles, 71,961 objects. Scene has many relatively large objects, half of the scene's volume is occupied by several planes.
3. Dataset 3: 10,154,304 triangles, 221,796 objects. Artificial test scene with 36 buildings. Each building has cluster of objects that can be culled after rendering exterior consisting of small number of objects.

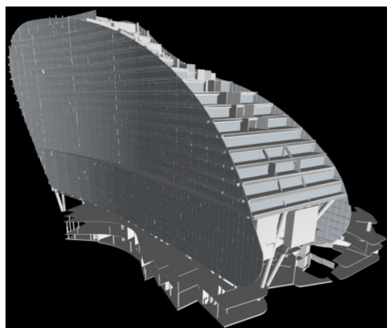

Fig. 4. Dataset 1 – architectural scene with 5 million triangles.

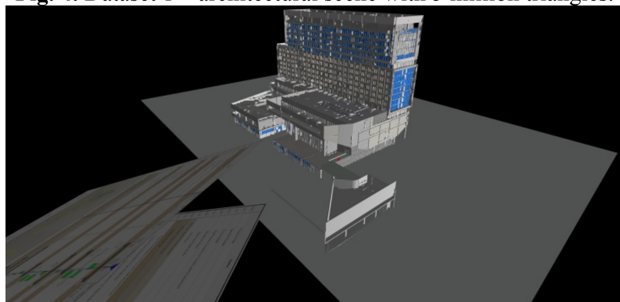

Fig. 5. Dataset 2 – architectural scene with 10.8 million triangles.

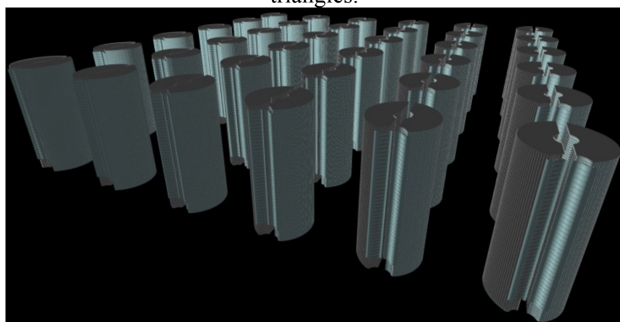

Fig. 6. Artificial test scene with buildings with 10.2 million triangles.

5.2 Clustering

Let's calculate the average number of rendered objects to compare object clustering efficiency of considered hierarchies for occlusion culling. Better object clustering should result in fewer visible nodes and fewer draw calls. During tests, we fixed the number of leafs for all hierarchies. BVH has the most efficient clustering of objects (fig. 7). It could be better but top-down subdivision process leads to the scenario where relatively long objects are gathered in a node and cannot be subdivided efficiently. In dataset 2 we encountered clusters with 40–70 objects where subdivision by any axis produced singleton. Algorithm based on octree issues more draw calls when space decomposition gives bounding volumes with a lot of empty space. Octree is more efficient for dataset 1 because it has very little empty space. It gives worst clustering in spacious dataset 3 because it cannot decompose it as efficiently using fixed number of leafs.

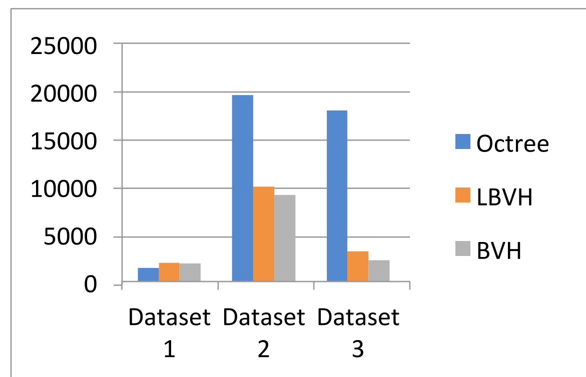

Fig. 7. Average number of rendered objects during scene walkthrough for considered hierarchies.

5.3 Frame rendering

All scene geometry is uploaded once at the beginning, shader with one directional light is used. Test results were produced on the system: AMD FX 8320 Processor, 24GB DDR3 RAM, AMD Radeon HD 6770 1GB.

Camera walkthrough is performed diagonally from the lower left to the upper right corner of a scene. Average and maximum frame rendering times are measured along the camera path (figs. 8, 9). Fig. 8 shows average rendering performance of all considered hierarchies on three datasets. Octree showed the fastest time for dataset 1 because it produced the best clustering (fig. 7). It performed better than expected for dataset 3 because most of the objects can be culled with relatively small number of queries. Dataset 2 was problematic for all hierarchies because it has most of the objects in one building. For efficient rendering careful balance of draw calls and occlusion queries is required. BVH SAH showed the fastest time because of efficient clustering. LBVH is close in performance to BVH SAH for all datasets.

Occlusion culling may give worse performance than frustum culling when GPU can efficiently render all of the objects inside the frustum. However, frustum culling shows worst performance on datasets 2 and 3 because of the large number of visible objects inside the frustum. Note that occlusion culling algorithm in the paper is not state-of-the-art and can be improved further to reduce the number of queries using visibility prediction and multiqueries [9][12].

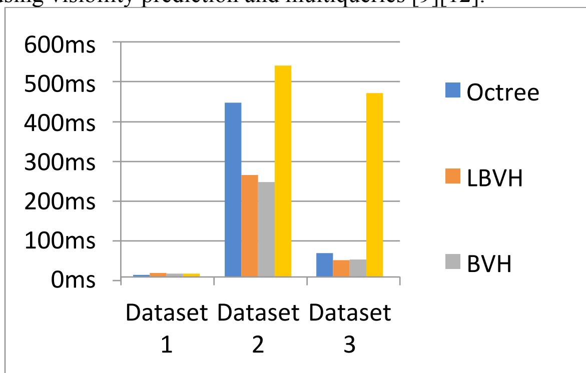

Fig. 8. Comparison of average frame rendering times for all datasets and hierarchies.

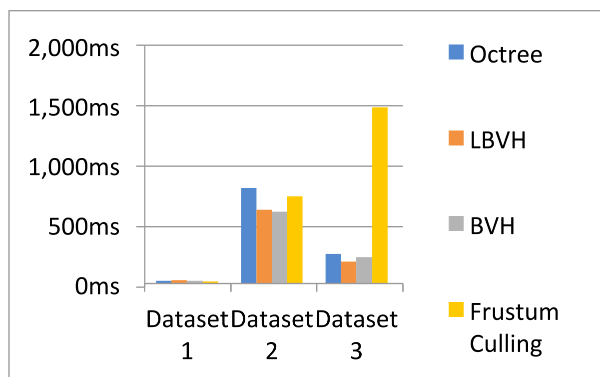

Fig. 9. Comparison of maximum frame rendering times for all datasets and hierarchies.

6. Conclusion

We performed the comparison of frame rendering performance when using different types of primitives and found that using objects instead of subdivided objects is more effective (fig. 3).

Octree efficiently handles datasets where most of the scene's volume is occupied by objects (fig. 7, dataset 1). Although storing objects in interior nodes of octree helps to select large objects and get better performance (fig. 2, dataset 2), storing objects in leafs is overall more effective and can be used to determine the number of leafs by scene's volume.

BVH SAH gives the most effective clustering of objects (fig. 7), and it positively affects frame rendering time (fig. 8). LBVH is close in performance to BVH SAH. Also, it is faster to construct, and bottom-up construction is better suited to get the optimal number of leafs.

7. References

[1] Bittner, J. et al, 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. In Computer Graphics Forum, Vol. 23, No.3, pp. 615–624.

[2] Bittner, J. and Wonka, P., 2003. Visibility in computer graphics. Environment and Planning B: Planning and Design, Vol. 30, No.5, pp.729–755.

[3] Blackert, A., 2016. Evaluation of multi-threading in Vulkan.

[4] Chandrasekaran C. et al, 2013–2016. Software Occlusion Culling. https://software.intel.com/en-us/articles/.

[5] Cohen-Or D. et al, 2003. A survey of visibility for walkthrough applications. IEEE Transactions on Visualization and Computer Graphic,Vol. 9, No. 3, pp. 412–431.

[6] GLAPI/glBeginQuery. https://www.opengl.org/wiki/GLAPI/glBeginQuery.

[7] Greene, N. et al, 1993. Hierarchical Z-buffer visibility. Proceedings of the 20th annual conference on Computer graphics and interactive techniques. ACM.

[8] Greene, N., 1996. Hierarchical polygon tiling with coverage masks. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. pp. 65–74.

[9] Guthe, M. et al, 2006. Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries. In Eurographics Symposium on Rendering. pp. 207–214.

[10] Legacy OpenGL - OpenGL Wiki. https://www.khronos.org/opengl/wiki/Legacy_OpenGL.

[11] Macdonald, J. D., Booth, K. S., 1990. Heuristics for ray tracing using space subdivision. Visual Computer, Vol. 6, No. 6, pp. 153–165.

[12] Mattausch, O. et al, 2008. CHC++: Coherent Hierarchical Culling Revisited. EUROGRAPHICS, Vol. 27, No. 3.

[13] Meissner, M. et al, 2001. Generation of Decomposition Hierarchies for Efficient Occlusion Culling of Large Polygonal Models. In Vision, Modeling, and Visualization, Vol. 1, pp. 225–232.

[14] Next Generation Occlusion Culling. http://www.gamasutra.com/view/feature/164660/sponsored_feature_next_generation_.php?print=1.

[15] Pharr, M. et al, 2016. Physically based rendering: From theory to implementation. Morgan Kaufmann.

[16] Samet, H., 2006. Foundations of multidimensional and metric data structures. Morgan Kaufmann.

[17] Teller, S., Sequin, C., 1991. Visibility preprocessing for interactive walkthroughs. Computer Graphics (Proceedings of SIGGRAPH 91), Vol. 25, No. 4, pp. 61–69.

[18] Wloka, M., 2003. Batch, batch, batch: What does it really mean. Presentation at game developers conference.

[19] Zhang, H. et al, 1997. Visibility culling using hierarchical occlusion maps. Proceedings of the 24th annual conference on Computer graphics and interactive techniques. pp. 77–88.

About the authors

Gonakhchyan Vyacheslav Igorevich, junior researcher at the department of System integration and multi-disciplinary collaborative environments of Ivannikov Institute for System Programming of the RAS. His email is pusheax@ispras.ru.