DOI: 10.1111/cgf.142649
EUROGRAPHICS 2021 / N. Mitra and I. Viola
(Guest Editors)

Volume 40 (2021), Number 2

Hierarchical Raster Occlusion Culling

Gi Beom Lee[1], Moonsoo Jeong[1], Yechan Seok[1], and Sungkil Lee[†] [1]

[1]Sungkyunkwan University, South Korea

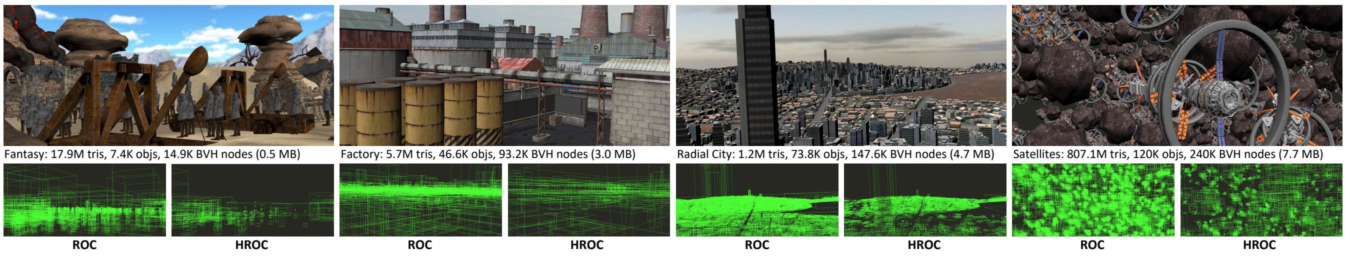

Fantasy: 17.9M tris, 7.4K objs, 14.9K BVH nodes (0.5 MB)
Factory: 5.7M tris, 46.6K objs, 93.2K BVH nodes (3.0 MB)
Radial City: 1.2M tris, 73.8K objs, 147.6K BVH nodes (4.7 MB)
Satellites: 807.1M tris, 120K objs, 240K BVH nodes (7.7 MB)
ROC
HROC
ROC
HROC
ROC
HROC
ROC
HROC

Figure 1: *Screenshots of the four scenes used for our experiments and rasterized bounds for occlusion culling. Unlike the previous Raster Occlusion Culling (ROC), our Hierarchical ROC (HROC) rasterizes coarser bounds of occludee groups, and casts per-pixel rays for fine-grained occlusion tests. This two-phase culling approach attains higher performance and efficiency than the previous techniques.*

Abstract

This paper presents a scalable online occlusion culling algorithm, which significantly improves the previous raster occlusion culling using object-level bounding volume hierarchy. Given occluders found with temporal coherence, we find and rasterize coarse groups of potential occludees in the hierarchy. Within the rasterized bounds, per-pixel ray casting tests fine-grained visibilities of every individual occludee. We further propose acceleration techniques including the read-back of counters for tightly-packed multidrawing and occluder filtering. Our solution requires only constant draw calls for batch occlusion tests, while avoiding costly iteration for hierarchy traversal. Our experiments prove our solution outperforms the existing solutions in terms of scalability, culling efficiency, and occlusion-query performance.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Rasterization; Visibility;***

1. Introduction

Occlusion culling (OC) bypasses rendering of occludees hidden by visible occluders, which is crucial in accelerating geometry rendering without sacrificing quality. Online OC techniques typically test the screen-space bounds of potential occludees against the depth buffer values occluders produce. Classical techniques use axis-aligned screen-space bounds, often with downsampling [ZMHHI97, CT97, Gre96] or reprojection [SKS11, LKE18] of the depth buffer. Their per-bound tests incur nearly constant overhead and scale well. However, efficiency is low due to generally wider bounds. Also, decoupling occluders and occludees is difficult.

An alternative approach is the rasterization of per-object bounds, where individual fragment depths are tested with the depth buffer values. The bounds are tighter, and occluders can be found based on temporal coherence. Efficient techniques rely on Graphics Processing Units (GPUs). Hardware occlusion query (HOQ) is a standard choice, but incurs non-trivial stalls [Sek04]. Hierarchical structures with temporal coherence [CT99, BWPP04, MBW08] largely suppress redundant queries, but require hierarchical iteration based on the last queries. This causes latency from query-and-readbacks for complex scenes. Raster OC (ROC) [KT14, BK15] made a success in completely eliminating stalls owing to early-Z, random access to GPU buffers, and indirect multidraw capability. Though, its per-object iteration becomes costly for complex scenes.

In this paper, we present a GPU-based Hierarchical ROC (HROC) algorithm, which improves the previous ROC using object-level Bounding Volume Hierarchy (BVH) for higher scalability. Unlike the ROC, we do not immediately test the visibilities of individual occludee bounds during their rasterization. Instead, our algorithm

[†] Corresponding author. e-mail: sungkil@skku.edu

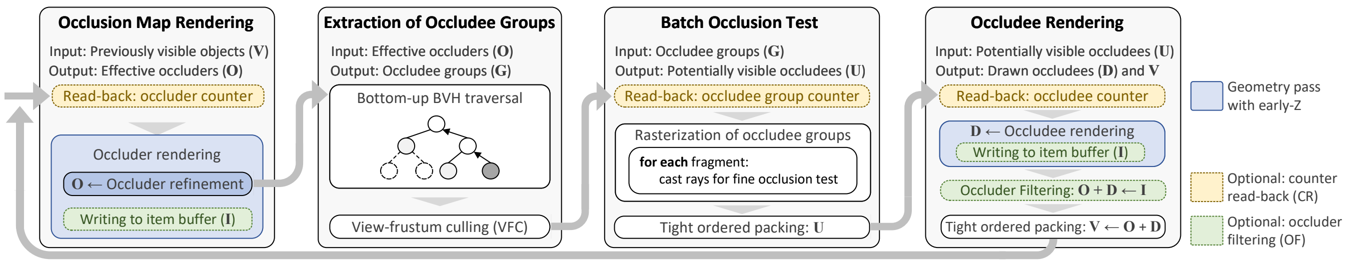

Occlusion Map Rendering
Input: Previously visible objects (V)
Output: Effective occluders (O)
Read-back: occluder counter
Occluder rendering
O ← Occluder refinement
Writing to item buffer (I)
Extraction of Occludee Groups
Input: Effective occluders (O)
Output: Occludee groups (G)
Bottom-up BVH traversal
View-frustum culling (VFC)
Batch Occlusion Test
Input: Occludee groups (G)
Output: Potentially visible occludees (U)
Read-back: occludee group counter
Rasterization of occludee groups
for each fragment:
cast rays for fine occlusion test
Tight ordered packing: U
Occludee Rendering
Input: Potentially visible occludees (U)
Output: Drawn occludees (D) and V
Read-back: occludee counter
D ← Occludee rendering
Writing to item buffer (I)
Occluder Filtering: O + D ← I
Tight ordered packing: V ← O + D
Geometry pass with early-Z
Optional: counter read-back (CR)
Optional: occluder filtering (OF)

Figure 2: *Overview of the four stages of our online rendering pipeline performed for every frame.*

first finds the coarse groups of potential occludees (interior nodes in the hierarchy), and rasterizes their bounds. The visibilities of individual objects are batch-tested using *per-pixel ray casting*, where the hierarchy is implicitly traversed within fragment processing. This two-phase approach better utilizes the fragment processors, and avoids costly iteration over the hierarchy at the host. We also intensively utilize *multidraw* capabilities for a stall-free pipeline, similarly to the ROC. We further accelerate our pipeline with the read-backs of draw counters for packed multidraw and occluder filtering for better temporal coherence. As a consequence, our solution attains higher scalability (from small-scale to gigantic scenes) and culling efficiency than the existing solutions.

Precisely, our major contributions can be summarized as:

- a hierarchical raster occlusion culling algorithm;
- a hybrid culling scheme using both rasterization and ray casting.

2. Related Work

Modern GPUs support early depth test (early-Z) or Hierarchical-Z (Hi-Z) [GKM93] for fragment culling. They can reject hidden fragments early, and reduce fragment processing. However, draw calls still require triggering for primitives. Hence, previous online OC solutions typically aim to operate on a per-object basis.

The depth buffer of occluders is often represented hierarchically (mipmapping [ZMHHI97, HW99, CT97] or N-Buffers [Déc05]). Identifying effective occluders often relies on heuristics, meta-data, and occluder fusion [LG95, KS00, WWS00], which is difficult for complex scenes. This can be improved with warping or reprojection of the depth buffer generated for the previous frame [SKS11, HA15, LKE18]. However, depth disocclusions degrade the culling efficiency, and two-pass culling might be necessary not to miss false negatives [HA15].

For the screen-space depth bounds of potential occludees, classical culling techniques typically use axis-aligned bounds. They scale well, but the axis-aligned bounds are often too conservative, resulting in lower culling efficiency. An alternative is rasterizing individual object bounds (e.g., boxes or k-DOPs [BKS05]) for per-fragment tests on a per-object basis. The tighter object bounds lead to higher culling efficiency, but individual bounds require rasterizing. Temporal coherence regards visible objects in the last frame as still visible [NL12]. A fundamental building block for the per-fragment test is HOQ [BMH98, CCG*01], which counts drawn fragments. The query itself is fast, but query results require reading back to the host to steer object rendering, unless conditional rendering is used. Also, per-object iteration can cause stalls for a complex scene [Sek04]. Software rasterization can eliminate stalls, but is feasible only for low resolutions [CMK*13, HAAM16].

A hierarchy of objects (e.g., BVH) can greatly reduce the amount of queries [BWPP04, WB05, GBK06, MBW08]. Temporal coherence is a key to reduce redundant queries by maintaining cuts in the hierarchy [CT99, BH01]. While generally efficient, the queries require iterating over the hierarchy on a per-object basis, resulting in still too many queries and read-backs for complex scenes. In contrast, ours does not maintain coherent cuts in the hierarchy, enabling a batch test. The use of coherence and hierarchy has been extended for ray tracing in CHC+RT [MBJ*15], which generalizes HOQ with ray-bound intersections. We also use GPU-based ray casting, but the difference is that ours is used to cull sets of *objects*, but CHC+RT culls sets of *rays*. Also, we use object indices at the intersections, while CHC+RT simply counts the number of intersections.

Recently, the ROC [KT14, BK15] greatly improves GPU culling without read-back stalls. The depth buffer of temporally coherent occluders is first rendered, and occludee bounds are tested. At the visible fragments passing early-Z, their object indices are set in the command buffer, and are multi-drawn finally. However, every object bound requires rasterizing, not scaling well with large scenes. Also, many fragments do redundant work; this has been recently improved with `NV_representative_fragment_test`.

The efficiency of the ROC is shared with our technique. A key distinction from the ROC is the use of a hierarchy for coarser queries, where the rasterization initiates fine-grained queries using ray casting for individual objects. This better utilization of parallel fragment processing avoids iterating for all the potential occludees, making ours less sensitive to a scene complexity. A simpler hierarchy-based approach [LL20] has been explored, but needs non-trivial iteration for read-backs, and the hierarchy is traversed only partially.

3. Algorithms

The overview of our online rendering pipeline is illustrated in Figure 2. For an input to our algorithm, an object-level BVH requires generating offline, which is constructed only for objects, not primitives. Every frame starts with the rendering of potential occluders to generate a depth buffer (occlusion map). Given the occluders, we find the groups of potential occludees. Then, the bounds of the occludee groups are drawn without depth writing, which is equivalent to the ROC on the host side. While the simple ROC directly marks the visibility flags of objects, we cast rays, and find bounding

Algorithm 1 Occludee Group Extraction

```
Input:  T: hierarchy with root $r$, O: occluder indices
Output: G: occludee groups
  procedure TRAVERSE($n$)
     while $n.state \neq visible$ and $n \neq r$ do
        $n.state \leftarrow visible$; $n.sibling.state \leftarrow unknown$
        $n \leftarrow n.parent$
  procedure EXTRACTOCCLUDEES(O)
     if $\mathbf{O} = \emptyset$ then APPEND($\mathbf{G}, r$) return
     for each $n \in \mathbf{T}$ do $n.state \leftarrow nil$
     for each $o \in \mathbf{O}$ do $n \leftarrow$ FINDNODE($o$); TRAVERSE($n$)
     for each $n \in \mathbf{T}$ do
        if $n.state = unknown$ then APPEND($\mathbf{G}, n$)
```

volumes intersecting with rays. The indices of the intersected leaf nodes drive the rendering of the corresponding objects. We describe the details of each stage in what follows.

3.1. Occlusion Map Rendering

Similarly to ROC [KT14, BK15], we also exploit *temporal coherence* to select a set of initial potential occluders **V**, which have been drawn in the previous frame; the first frame considers all the objects as visible. The occluders are directly rendered into the framebuffer. Their depth buffer, which is a byproduct of the rendering, is used as an *occlusion map* for depth tests in the occlusion test stage.

While ROC tests all the objects as occludees, we exclude drawn occluders in occlusion tests. Since not all occluders in **V** are likely to be visible as the view may change, we select only effective occluders **O**, and send their indices to the next stage. To this end, our occluder rendering uses early-Z unlike ROC. For further accelerations, we present how to reduce false positives in **V** (Sec. 4.2).

3.2. Extraction of Occludee Groups

We find a set of the groups of potential occludees, **G**, in the hierarchy for batch query; we construct a BVH hierarchy only for object bounds. Our key idea to extract **G** is based on an observation that all the ancestors of a visible leaf node (occluder) are also visible, and the visibility of remaining nodes are undetermined. A bottom-up traversal from each occluder can find the visible ancestors, and the others are left as visibility-unknown.

Algorithm 1 shows the pseudocode of the occludee group extraction. Given occluders **O**, we traverse the hierarchy **T** (see Figure 3 for example). When **O** is empty, all the objects are considered potential occludees. Otherwise, we find a leaf node $n \in \mathbf{T}$ corresponding to each occluder $o \in \mathbf{O}$; their pairwise mapping is built offline. For every n, we traverse bottom up, until we meet an already visited node v or the root node r. n and its all the ancestors are marked as visible, meaning nodes within a path from r to n are visible. On the other hand, the sibling nodes u of n and v are marked as unknown, which require testing occlusion. After traversing for all the occluders, we collect nodes with the unknown state and add them in **G**; we also perform view-frustum culling (VFC) in the end of the stage.

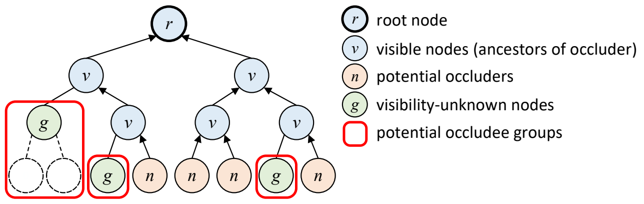
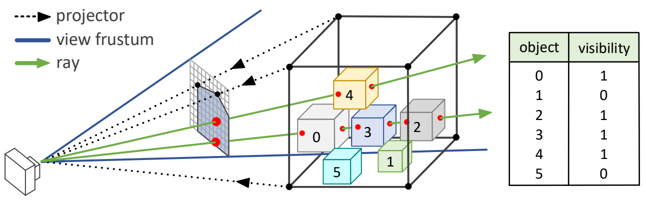


Figure 3: *An example of the bottom-up traversal for extracting occludee groups (g; visibility-unknown nodes), which are the siblings of every ancestor (v) of potential occluders (n).*



object	visibility
0	1
1	0
2	1
3	1
4	1
5	0

Figure 4: *Within the rasterized bound of a coarse occludee group, rays are cast to test fine-grained visibilities of individual occludees.*

3.3. Batch Occlusion Test

We perform occlusion tests in two phases, one in the fixed-function GPU rasterization and the other in fragment processing through our custom shader. As already alluded to, we rasterize the bounds not of individual objects but of the occludee groups. Depth writing is disabled and early-Z is enabled.

The previous ROC marks a visibility flag for each bound, but we instead cast rays (Figure 4) when the bound does not correspond to a leaf node. A primary ray is cast at each fragment that passed early-Z; fully hidden groups are implicitly culled. Its intersections are found against **T**. The traversal for intersections starts from the interior node that triggers the rasterization. Objects at intersections are added to the set of potentially visible occludees **U**, which are drawn in the final stage. For efficiency, we avoid repeated tests for already visited nodes. Precisely, we check the sibling of every intersected node is visited, and if so, we tag their parent as visited. We repeat the process until reaching a non-visited sibling or the root. We note the ray casting should be *conservative*. In other words, ray casting should not stop at the nearest intersection as usual, because rays blocked by the conservative bounds are not necessarily blocked by real object geometries.

This strategy greatly reduces the amounts of rasterization, while testing every single object in parallel. Importantly, the pass for occlusion tests is not iterated, not following preceding queries. Also, we can use a single-pass multidraw for all the occludees.

3.4. Occludee Rendering

Finally, we render the actual geometries of the potentially visible occludees **U**. We again resort to early-Z for efficiency, and add objects passing early-Z to the set of drawn objects **D**. After drawing all the potential occludees, we merge **D** with the potential occluders (**O**) into the object indices **V** to initialize the next-frame rendering.

Table 1: *Comparison of culling methods by per-frame timing (ms).*

Scene	NOC	VFC	CHC++	ROC	WOC	IOC	REF	HROC	
								OF	CR+OF
FN	4.8	3.9	8.8	1.3	3.3	28.9	0.8	1.2	1.8
FC	25.5	25.7	9.1	3.9	6.3	20.8	0.5	2.2	1.6
RC	44.3	21.0	32.6	5.9	15.0	14.7	0.9	3.3	2.2
RC′	45.1	20.6	39.6	6.0	13.8	12.7	0.7	3.3	2.4
ST	66.6	37.1	88.1	14.6	15.9	20.7	6.7	11.7	9.2

Table 2: *Comparison of ROC and HROC in* 4096×2160 *resolution.*

Scene	FN	FC	RC	ST
ROC	2.42	4.50	6.78	15.85
HROC (OF)	2.56	3.43	4.76	13.77
HROC (CR+OF)	3.23	3.35	4.10	11.54

4. Accelerations

This section presents two acceleration techniques, which revisit and improve the previous techniques for our purpose.

4.1. Packed Multidraw with Counter Read-Back

Our solution can also work without stalling the pipeline similarly to the ROC. The stall-free pipeline relies heavily on indirect *multidraw* capability of modern GPUs [KT14]. The multidraw processes a command buffer filled for the entire objects in a single batch, which greatly reduces the driver overhead of GPUs. For culling, we set the instance counters for occluded objects to zeros. Unfortunately, modern GPUs still do not well handle such *void* objects, resulting in performance drops for many void objects.

Similarly to [HA15, Wih16], we can tightly pack and render with the command buffer so that only effective objects are handled in multidraw with their counters; e.g., we can use `glMultiDrawElementsIndirectCount` in OpenGL. It is important to preserve the rendering order of objects. Otherwise, we may encounter frame inconsistency resulting from Z-fighting. Tiny objects with the same depth may hide each other but in different orders; note that the ROC is free of this race condition, since it does not use packing. For the ordered packing, we also use a parallel prefix sum [HSO08, Wih16] for **V** and **U**. The counter of the packed objects is also obtained as a byproduct of the prefix sum.

The stall-free pipeline is already efficient with the packing, but we still have to provide another counter `maxdrawcount` that specifies the maximum number of objects we allow. `maxdrawcount` needs to be provided in the host, whereas the draw counter of effective objects to draw can be dereferenced in GPU. A standard usage is providing the total number of objects, but may cause inefficiency to a large extent; the counters of all the objects and BVH nodes require to be provided. To cope with this problem again, we read back the counters to the host, and use them for `maxdrawcount`. In our case, the counters of **V**, **G**, and **U** (Figure 2) can be read. Whereas this stalls the pipeline, it completely removes redundancy in multidraw, and achieves higher performance for complex scenes.

4.2. Occluder Filtering for Better Temporal Coherence

In our basic algorithm (Sec. 3), the early-Z selects occluders for temporal coherence. However, the early-Z inherently includes non-trivial false positives. Invisible objects rendered earlier than their blockers can be classified as falsely visible by the incomplete depth buffer. To filter out these false positives, sorting might matter [HA15], but be costly and difficult for complex scenes.

We exploit a simpler yet effective strategy to reduce the false positives in temporal coherence. In our basic scheme, occluders **V** in frame t are the union of **O** and **D** in frame $t-1$, including false positives. To select pure true positives, we use the item-buffer technique [WHG84, KS01]. While rendering real geometries for **O** and **D**, we write their indices in another render target, **I** (i.e., the item buffer). Then, we collect the indices of objects that pass the regular depth test and mark them as visible in the command buffer just before the packing of **V** for frame t (Sec. 4.1); hence, we do not read the item buffer back to the host, unlike [KS01]. Thereby, less potential occluders (**V**) are selected for frame t, and more potential occludees (including the false positives) can be disoccluded, tested, and culled; nonetheless, this does not lower the culling efficiency.

5. Results

This section reports our experimental results and comparisons with existing methods, assessed in terms of performance and efficiency.

5.1. Method

We implemented and experimented our algorithm on an Intel Core i7-7800X 3.6GHz machine with NVIDIA GeForce GTX 2080 Ti, using OpenGL API in Windows 10. Unless noted, all the tests were performed at 1920×1080 (full-HD) resolution.

Four scenes and their BVHs (generated offline) are used for the experiments (Figure 1). The Fantasy scene (FN) is simple with a small number of complex objects. The Factory scene (FC) is a widespread medium-scale scene with detailed objects. The Radial City scene (RC) has many low-polygon objects. The Satellites scene (ST) is challenging in terms both of geometric details and the number of objects. Each scene defines an animated sequence of the duration of 10 s; RC defines another sequence RC′ (for the same scene) to evaluate faster rotations and backward camera walking. Frame time measurements are averaged for the sequences; the first four frames are excluded not to reflect the lazy initialization of OpenGL.

HROC is parameter-less, and requires no particular per-scene parameters, except for the BVH generation. Our experiments always use the packed command buffer, and optionally use the Counter Read-back (CR; Sec. 4.1) and Occluder Filtering (OF; Sec. 4.2); CR+OF indicates both CR and OF are used.

We compare our solution with NO-Cull rendering (NOC), VFC, and an ideal REFerence rendering (REF). REF pre-records all the visible objects offline for the animated sequences, and multidraws them in a batch. We also implemented and compared four existing

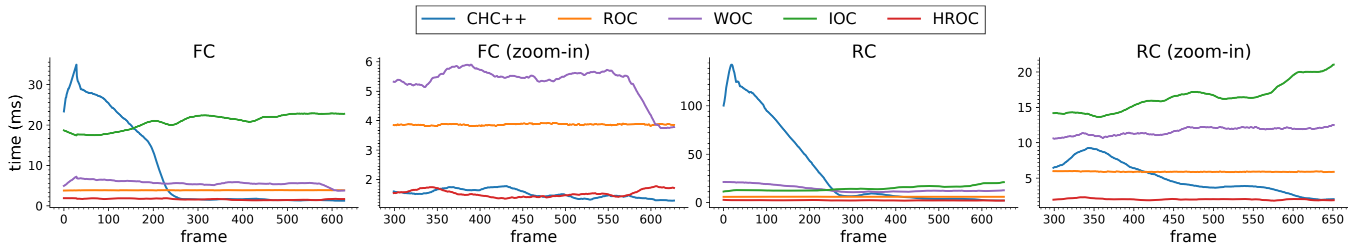


Figure 5: *Per-frame time variations of our solution (CR+OF) along the frames of animated camera sequences (FC and RC).*

Table 3: *Effects (timings measured in ms) of the Counter Readback (CR) and Occluder Filtering (OF) against No-Acceleration rendering (NA). The numbers in parentheses are speedup factors with respect to NA.*

Scene	NA	CR	OF	CR+OF
FN	1.37	2.21 (0.62)	1.17 (1.17)	1.83 (0.75)
FC	2.25	1.86 (1.21)	2.17 (1.04)	1.63 (1.38)
RC	3.35	2.37 (1.41)	3.27 (1.02)	2.20 (1.52)
ST	14.00	11.74 (1.19)	11.66 (1.20)	9.22 (1.52)

solutions, including CHC++ [MBW08] (the state-of-the-art hierarchical culling), the original ROC [BK15] (the state-of-the-art raster culling), Warping-based Occlusion Culling (WOC) [LKE18] (the state-of-the-art culling using axis-aligned bounds), and a recent Iterative ROC [LL20] (IOC); we excluded techniques that manually select occluder/occludee sets in our choice. Our implementation of CHC++ uses the same BVHs that we use for HROC, and includes batch, randomization, tighter bounds of inner nodes, and multi-queries with the following parameters: n_{av}=10, b=4000, d_{max}=4, and s_{max}=1.4 [MBW08]; we note that b and d_{max} are optimized for our scenes. WOC uses a single-layer warping with conservative holes; only WOC uses a deferred pipeline to perform its depth warping. IOC uses the same BVHs, and its BVH depth ranges are $[6,8]$, $[11,13]$, $[9,11]$, and $[14,17]$, for FN, FC, RC, and ST, respectively, which are manually tuned for optimal performance.

5.2. Performance

Table 1 summarizes the overall performance, which compares ours (HROC) with NOC, VFC, CHC++, ROC, WOC, IOC, and REF. In all the scenes, HROC performs best, which proves the scalability of our algorithm. Our solution is less sensitive to the number of objects owing to the hierarchical approach and per-pixel ray casting. Specifically, the OF-only rendering (stall-free rendering without CR) performs best for FN, and CR+OF performs best for the remainder (FC, RC, RC′, and ST). The speedup factors of ours (OF for FN and CR+OF for the remainder) against CHC++, ROC, WOC, and IOC are 7.5/1.4/2.9/24.8×, 5.5/2.4/3.9/12.7×, 14.8/2.7/6.8/6.7×, and 9.6/1.8/1.7/2.2×, respectively. For RC′, CHC++, ROC, and HROC perform slightly worse than RC, but the penalty from the lower temporal coherence is not much. While ours scales well from small-scale to complex scenes, others manifest themselves in particular configurations. Up to medium-scale scenes (FN, FC, and RC),

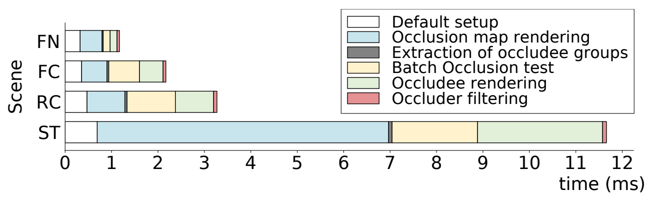


Figure 6: *Per-stage performance breakdown of HROC (OF).*

ROC performs best except ours. WOC performs well for large-scale scenes (ST). CHC++ performs relatively well up to medium-scale scenes (FN and FC), but suffers from large-scale scenes (RC and ST). To our experiences, CHC++ performs very well in highly-occluded scenes. IOC performs well only for large-scale scenes (RC and ST).

Table 2 compares ROC and HROC in 4096 × 2160 resolution. The tendency is similar to those for the full-HD resolution, but the speedup here (0.9, 1.3, 1.7, and 1.4× for FN, FC, RC, and ST, respectively) is slightly lower. This shows HROC is still scalable in a higher resolution (in particular for per-pixel ray casting).

Table 3 reports the effects of our acceleration techniques (OF and CR), which are measured against No-Acceleration (NA) in terms of the average per-frame times (ms). At a glance, NA already performs better than the existing techniques (except for FN), but further speedups are made with OF and CR. OF improves performance by culling more occludees, and CR gains speedup for complex scenes (except FN) by precluding irrelevant (exceeding the draw counter) objects in the indirect multidraw. The speedups of OF, CR, and CR+OF against NA become more apparent as the scene complexity increases; CR+OF reaches up to 1.52 for RC and ST. Based on the experiments, ours is likely to perform optimally when using the stall-free pipeline (OF) for small-scale scenes and CR+OF for medium to large-scale scenes, respectively.

Figure 5 shows timing variations along the camera sequences. HROC (CR+OF), ROC, and WOC are insensitive to camera movements, where the performance slightly fluctuates without strong peaks. IOC is slower but also insensitive to camera movements. CHC++ gradually improves over time, and converges to those of HROC as objects are occluded more in close-up views.

Figure 6 shows the detailed timings for our solutions, OF and CR+OF; "default setup" indicates baselines for non-culling cost. Overall, the occlusion map rendering, occludee rendering, and occlusion test take the majority of the cost. The cost for the extraction of occludee groups is negligible. HROC is lower in occlusion-test

Table 4: *Comparison of the average culling efficiency of ours (NA and OF) against VFC, CHC++, ROC, WOC, IOC, and REF. C, N, and R indicate the numbers of culled objects for each method, the total objects, and the culled objects of REF, respectively.*

Scene	VFC		CHC++		ROC		WOC		IOC		REF		HROC (NA)		HROC (OF)	
	C/*N*	*C*/*R*	*C*/*N*	*C*/*R*	*C*/*N*	*C*/*R*	*C*/*N*	*C*/*R*	*C*/*N*	*C*/*R*	*C*/*N*	*C*/*R*	*C*/*N*	*C*/*R*	*C*/*N*	*C*/*R*
FN	0.59	0.65	0.88	0.97	0.88	0.97	0.87	0.96	0.63	0.69	0.91	1.00	0.75	0.83	0.88	0.97
FC	0.41	0.41	0.99	1.00	0.99	1.00	0.99	1.00	0.89	0.90	0.99	1.00	0.91	0.92	0.99	1.00
RC	0.83	0.85	0.96	0.99	0.96	0.99	0.95	0.98	0.90	0.92	0.97	1.00	0.96	0.98	0.97	0.99
RC$'$	0.82	0.86	0.94	0.98	0.94	0.98	0.92	0.97	0.86	0.90	0.95	1.00	0.94	0.98	0.94	0.99
ST	0.77	0.81	0.93	0.98	0.94	0.98	0.92	0.97	0.92	0.96	0.96	1.00	0.92	0.96	0.94	0.98

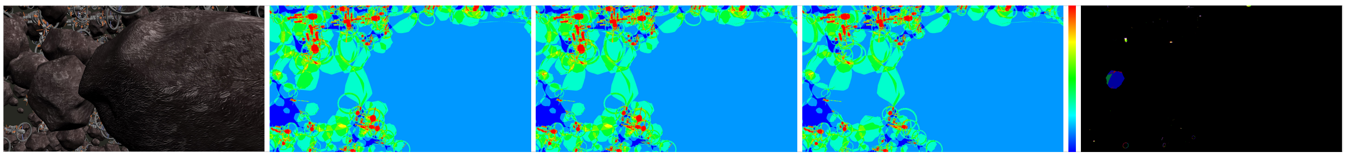


Figure 7: *Comparison of overdrawing of ROC (second), HROC (third) and REF (fourth), and the difference between ROC and HROC (fifth).*

overhead than ROC. The pure OC-only overhead of HROC (for occludee group extraction, occlusion test, and occluder filtering) are 0.23, 0.76, 1.16, and 2.0 ms for FN, FC, RC, and ST, respectively, while those of ROC are 0.25, 1.22, 1.97, and 3.3 ms.

Our technique consumes additional GPU memory mainly for BVHs and item buffer. The memory consumption of BVHs is insignificant (0.5–7.8 MB; Figure 1), and those of the item buffer are 8 and 32 MB for the full-HD and 4K resolutions, respectively.

We report further statistics for the per-pixel ray casting as follows. The maximum BVH depths are given as 19, 20, 22, and 25 for FN, FC, RC, and ST, respectively. The traversals from the occludee group (interior) nodes start at the depths of 12.1, 12.3, 14.3, and 16.3 on average. The average heights from the group nodes to the leaf nodes are 1.2, 4.1, 4.5, and 2.5, visiting 4–6 nodes on average.

5.3. Culling Efficiency

Table 4 compares culling efficiencies, which are measured in terms of relative fractions with respect to the total number of objects (N) and the number of culled objects in REF (R). HROC (OF) achieves the highest efficiencies close to those of REF by removing most of the false positives included in the occluders of HROC (NA); unlike ROC, HROC occlusion-tests only potential occludees. CHC++ and ROC are similar to HROC (OF). WOC gains further lower efficiency, resulting from its axis-aligned bounds with depth holes. IOC has consistently lower efficiencies due to its limited hierarchy traversal. RC$'$ identifies that temporally less coherent motions result in slightly degradation. Obviously, VFC has the lowest efficiency, since it does not deal with occlusions. Figure 7 compares ROC and HROC (OF) in ST in terms of overdrawing. The overdrawing is very low for both ROC and HROC, which is close to that of REF.

6. Conclusion and Discussions

We presented a scalable online hierarchical raster occlusion culling algorithm, which improves the previous raster occlusion culling in terms of scalability and culling efficiency. Our algorithm rasterizes coarser groups of occludees, while casting rays for fine-grained occlusion tests for individual objects. Our solution outperforms most of the state-of-the-art culling techniques.

One of the inherent limitations of our algorithm, which is shared with other hierarchy-based solutions, is that we assume a static hierarchy and dynamic objects require handling independently. A potential solution is a hybrid culling solution that uses the HROC and ROC for static and dynamic objects, respectively. Dynamic potential occludees can be directly fed into our batch occlusion test as they are individual occludee groups, and their visibilities are tagged without ray casting as ROC does. This would work because the number of dynamic objects is likely to be less than static objects in many cases and ROC can work well for dynamic objects.

The counter read-back could be infeasible for practical rendering pipelines. A viable alternative to get rid of the CPU-GPU interops is an approximation of the counters with a reasonable margin. Also, our in-house ray casting with OpenGL and hierarchy construction can be improved with the recent RTX extension in Vulkan.

Another performance penalty rarely occurs in case that a camera lies inside the bound of a large occludee group. In such a case, our algorithm may cast rays for the entire screen, which is excessive. A potential solution can be an adaptive grouping, which subdivides the box down to its children to avoid testing against the large bound.

Acknowledgments

3D Models of FC, RC, and ST scenes are provided through the courtesy of {Dmitrii Kutsenko, monstranim, SkyhawkTron, xedvenom} at cgtrader.com, and FN scene, {terryzhang, marco-santoriello, inalaatzu} at cgtrader.com and soidev at turbosquid.com, respectively. This work was supported in part by Mid-career R&D program through the NRF grants (2019R1A2C2002449) funded by the Korea Government and the Samsung Research Funding & Incubation Center of Samsung Electronics (SRFC-IT1901-01). Correspondences concerning this article can be addressed to Sungkil Lee.

References

[BH01] BITTNER J., HAVRAN V.: Exploiting coherence in hierarchical visibility algorithms. *The Journal of Visualization and Computer Animation 12*, 5 (2001), 277–286. 2

[BK15] BOUDIER P., KUBISCH C.: GPU Driven Large Scene Rendering. NVIDIA GPU Technology Conference, 2015. 1, 2, 3, 5

[BKS05] BARTZ D., KLOSOWSKI J. T., STANEKE D.: *Tighter bounding volumes for better occlusion culling performance*. Technical Report WSI-2005-13, Wilhelm-Schickard-Institut, 2005. 2

[BMH98] BARTZ D., MEISSNER M., HÜTTNER T.: Extending graphics hardware for occlusion queries in OpenGL. In *Proc. Graphics hardware* (1998), pp. 97–ff. 2

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3 (2004), 615–624. 1, 2

[CCG*01] CUNNIFF R., CRAIGHEAD M., GINSBURG D., LEFEBVRE K., LICEA-KANE B., TRIANTOS N.: *ARB occlusion query*. White paper, NVIDIA and ATI, 2001. 2

[CMK*13] CHANDRASEKARAN C., MCNABB D., KUAH K., FAUCONNEAU M., GIESEN F.: *Software Occlusion Culling*. White paper, Intel Corporation, 2013. 2

[CT97] COORG S., TELLER S.: Real-time occlusion culling for models with large occluders. In *Proc. Symp. Interactive 3D graphics* (1997), pp. 83–ff. 1, 2

[CT99] COORG S., TELLER S.: Temporally coherent conservative visibility. *Computational Geometry 12*, 1-2 (1999), 105–124. 1, 2

[Déc05] DÉCORET X.: N-buffers for efficient depth map query. *Computer Graphics Forum 24*, 3 (2005), 393–400. 2

[GBK06] GUTHE M., BALÁZS Á., KLEIN R.: Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In *Proc. Eurographics Symp. Rendering* (2006), pp. 207–214. 2

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical z-buffer visibility. In *Proc. ACM SIGGRAPH* (1993), pp. 231–238. 2

[Gre96] GREENE N.: Hierarchical polygon tiling with coverage masks. In *Proc. ACM SIGGRAPH* (1996), pp. 65–74. 1

[HA15] HAAR U., AALTONEN S.: GPU-driven rendering pipelines. SIGGRAPH Advances in Real-Time Rendering in Games course, 2015. 2, 4

[HAAM16] HASSELGREN J., ANDERSSON M., AKENINE-MÖLLER T.: Masked software occlusion culling. In *Proc. High Performance Graphics* (2016), pp. 23–31. 2

[HSO08] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). Addison-Wesley, 2008, ch. 39, pp. 851–876. 4

[HW99] HO P. C., WANG W.: Occlusion culling using minimum occluder set and opacity map. In *Proc. IEEE Information Visualization* (1999), pp. 292–300. 2

[KS00] KLOSOWSKI J. T., SILVA C. T.: The prioritized-layered projection algorithm for visible set estimation. *IEEE Trans. Visualization and Computer Graphics 6*, 2 (2000), 108–123. 2

[KS01] KLOSOWSKI J. T., SILVA C. T.: Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Trans. Visualization and Computer Graphics 7*, 4 (2001), 365–379. 4

[KT14] KUBISCH C., TAVENRATH M.: OpenGL 4.4 scene rendering techniques. NVIDIA GPU Technology Conference, 2014. 1, 2, 3, 4

[LG95] LUEBKE D., GEORGES C.: Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proc. ACM Symp. Interactive 3D graphics* (1995), pp. 105–ff. 2

[LKE18] LEE S., KIM Y., EISEMANN E.: Iterative Depth Warping. *ACM Trans. Graphics 37*, 5 (2018), 177:1–13. 1, 2, 5

[LL20] LEE G. B., LEE S.: Iterative GPU Occlusion Culling with BVH. In *High Performance Graphics Posters* (2020). 2, 5

[MBJ*15] MATTAUSCH O., BITTNER J., JASPE A., GOBBETTI E., WIMMER M., PAJAROLA R.: CHC+RT: Coherent Hierarchical Culling for Ray Tracing. *Computer Graphics Forum 34*, 2 (2015), 537–548. 2

[MBW08] MATTAUSCH O., BITTNER J., WIMMER M.: CHC++: Coherent hierarchical culling revisited. *Computer Graphics Forum 27*, 2 (2008), 221–230. 1, 2, 5

[NL12] NIESSNER M., LOOP C.: Patch-based occlusion culling for hardware tessellation. In *Proc. Computer Graph. Int.* (2012). 2

[Sek04] SEKULIC D.: Efficient occlusion culling. In *GPU Gems*, Fernando R., (Ed.). Addison-Wesley, 2004, ch. 29, pp. 375–390. 1, 2

[SKS11] SOUSA T., KASYAN N., SCHULZ N.: Secrets of CryENGINE 3 graphics technology. SIGGRAPH Advances in Real-Time Rendering in Games course, 2011. 1, 2

[WB05] WIMMER M., BITTNER J.: Hardware occlusion queries made useful. In *GPU Gems 2*, Pharr M., (Ed.). Addison-Wesley, 2005, ch. 6, pp. 375–390. 2

[WHG84] WEGHORST H., HOOPER G., GREENBERG D. P.: Improved Computational Methods for Ray Tracing. *ACM Trans. Graphics 3*, 1 (1984), 52–69. 4

[Wih16] WIHLIDAL G.: Optimizing the Graphics Pipeline With Compute. GDC, 2016. 4

[WWS00] WONKA P., WIMMER M., SCHMALSTIEG D.: Visibility preprocessing with occluder fusion for urban walkthroughs. In *Proc. Eurographics Workshop on Rendering* (2000), pp. 71–82. 2

[ZMHHI97] ZHANG H., MANOCHA D., HUDSON T., HOFF III K. E.: Visibility culling using hierarchical occlusion maps. In *Proc. ACM SIGGRAPH* (1997), pp. 77–88. 1, 2