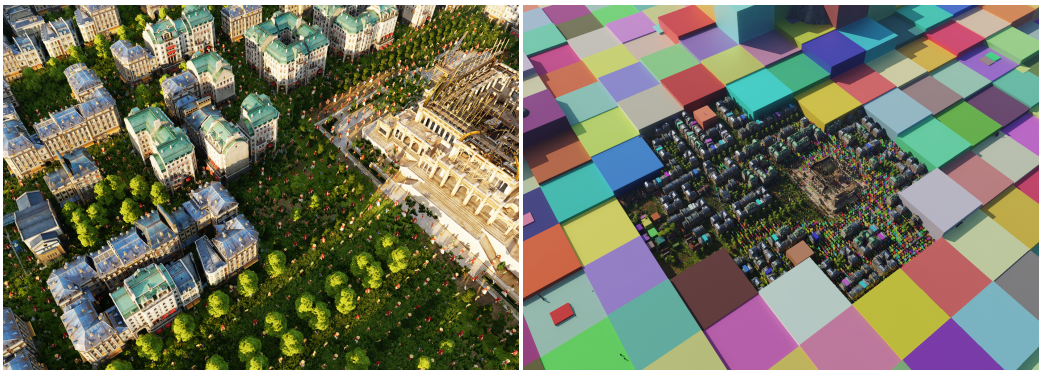
Light Path Guided Culling for Hybrid Real-Time Path Tracing

JAN KELLING, Ubisoft, Germany
DANIEL STRÖTER, Technical University of Darmstadt, Germany
ARJAN KUIJPER, Technical University of Darmstadt, Germany and Fraunhofer IGD, Germany



Hybrid path tracing with LiPaC | Acceleration structure visualization

Fig. 1. Left: City scene with many animated units rendered by our method, 19ms on an AMD RX 7900 XTX at 1080p. Right: Visualization of the top-level acceleration structure used for ray tracing.

Rendering visually convincing images requires realistic lighting. Path tracing has long been used in offline rendering to produce photorealistic images. While recent hardware advancements allow ray tracing methods to be employed in real-time renderers, they come with a significant performance and memory impact. Real-time path tracing remains a challenge. We present light path guided culling (LiPaC), a novel culling algorithm for ray tracing that achieves almost optimal culling results by considering the number of light paths encountered by objects. In addition, we describe a hybrid path tracing pipeline using LiPaC to render large and highly dynamic scenes in real-time on the current generation of consumer hardware.

CCS Concepts: • **Computing methodologies** → **Ray tracing**.

Additional Key Words and Phrases: Ray Tracing, Path Tracing, Real-Time, Culling, Hybrid Rendering, Games

1 INTRODUCTION

To produce visually convincing images of 3D scenes, realistic lighting is required. Improved lighting can result in rendered scenes being more visually pleasing and allow for better perception of the geometry and its materials. Highly accurate lighting can be obtained from path tracing [Kajiya 1986], a technique following arbitrary light rays through the scene. It has been used for many years in offline rendering, where rendering times of several minutes or hours are acceptable. Path tracing methods usually rely on ray tracing to determine intersections of rays with the scene.

Due to its computational cost, ray tracing techniques have traditionally seen limited use in real-time graphics. However, the ever growing complexity of graphics processing units (GPUs) and advancements in hardware accelerated ray tracing during the last years allow to utilize ray tracing techniques to greatly increase lighting quality even in real-time renderers. In comparison to specialized ray tracing effects, such as ray traced diffuse global illumination, path tracing solves

Authors' addresses: Jan Kelling, jan.kelling@ubisoft.com, Ubisoft, Mainz, Germany; Daniel Ströter, daniel.stroeter@gris.tu-darmstadt.de, Technical University of Darmstadt, Darmstadt, Germany; Arjan Kuijper, arjan.kuijper@igd.fraunhofer.de, Technical University of Darmstadt, Darmstadt, Germany and Fraunhofer IGD, Darmstadt, Germany.

the full rendering equation [Kajiya 1986] and can produce accurate lighting even in special cases. However, real-time path tracing is challenging due its runtime and memory demands. One major challenge for all real-time ray tracing techniques is management of acceleration structures in graphics memory. They are needed for fast intersection testing and have to represent the entire scene. This can quickly become a limiting factor, especially for large and highly dynamic scenes. Real-time renderers often employ culling strategies to handle such scenes. However, known strategies such as frustum or occlusion culling cannot directly be applied when using ray tracing. We present LiPaC, a new application-level method for efficient culling of ray traced scenes. The method divides the scene spatially and counts ray intersections in each cell to determine its importance for the lighting of the scene. Each object in the scene has a persistent state deciding over its ray tracing representation and culling. This state is changed by heuristics based on the previously encountered number of light paths. In summary, the contribution of this work consists of

- a general purpose culling method for real-time ray tracing that allows for fine trade-offs between quality and memory/performance
- a GPU-driven acceleration structure management approach that can alleviate CPU bottlenecks observed with real-time ray tracing
- a hybrid path tracing pipeline that allows aggressive acceleration structure culling and is able to path trace highly dynamic scenes in real-time on commodity hardware.

2 BACKGROUND & RELATED WORK

Path tracing originated as a solution to the rendering equation [Kajiya 1986] and has a long history in computer graphics. Most often, ray tracing is used to find intersections between light rays and scene geometry. This process is computationally expensive and acceleration structures such as bounding volume hierarchies [Kay and Kajiya 1986] are used to reduce the number of necessary intersection tests. Recent GPUs include specialized hardware for accelerating ray tracing. Modern graphics APIs expose these capabilities [Hector et al. 2020; Wyman et al. 2018]. They assume a two-level acceleration structures scheme. Individual meshes are capsuled into so-called bottom-level acceleration structures (BLASes) which in turn are used to build a top-level acceleration structure (TLAS) representing the scene. GPU pipelines can then use such a TLAS to compute ray intersections. Ray tracing pipelines introduce additional shader types: For each intersection between ray and geometry, an *anyhit* shader can possibly reject the hit. For the closest valid hit, a *closeshit* shader will be invoked.

Real-time renderers such as video games have been employing ray traced effects for the last years. For instance, diffuse global illumination [Majercik et al. 2019], reflections [Deligiannis and Schmid 2019] or shadows [Boksansky et al. 2019] have been built upon hardware accelerated ray tracing. However, these independent ray tracing effects only solve specific parts of the rendering equation and produced images will still exhibit inaccurate lighting in many cases. Path tracing solves the entire rendering equation by capturing even complex light paths. Hybrid effects mixing rasterization or screen-space ray tracing with hardware accelerated ray tracing have been proposed [Bavoil et al. 2018; Brenham and Szlachtycz 2022; Willberger et al. 2019]. Our pipeline extends these ideas by allowing to completely fall back to rasterization for some scene objects that pose performance problems for ray tracing.

Denoising advancements [Schied et al. 2017; Zhdan 2021] allow to generate visually pleasing images from stochastic ray tracing effects. Denoising algorithms can create smooth images that are close to ground truth with as little information as one ray per pixel per frame for Monte-Carlo solvers. Real-time path tracing for such low sample counts is possible on the powerful ray tracing GPUs of recent years. First real-time renderers relying on path tracing have been

released [Schied 2019], even for current generation games [Burnes 2023]. However, real-time path tracing is computationally expensive and challenging to implement efficiently. We propose multiple performance and memory consumption improvements to GPU-accelerated ray tracing, path tracing in particular.

Utilizing hardware accelerated ray tracing in current generation games poses many problems. Especially for large and highly dynamic scenes, building acceleration structures can become a limiting factor [Gruen 2020]. BLASes can consume high amounts of graphics memory, even more so with many animated or otherwise uniquely-transformed instances that prevent BLAS sharing. There are many known culling techniques for real-time renderers to handle complex, dynamic scenes [Beacco et al. 2016]. For ray tracing, culling instances by not building their BLAS and not including them in the TLAS is an important optimization to reduce acceleration structure building time and memory consumption. However, ray tracing techniques require new culling strategies since objects outside of the frustum or occluded from the camera can still have a big influence on the lighting. Heuristics based on instance size and distance have been proposed [Deligiannis and Schmid 2019] and are commonly used [Sjoholm 2020]. Such heuristics can miss important occluders and might still include many instances that have no noticeable impact on the final image. Furthermore, in scenes with many animated models, updating all included BLASes in every frame is too computationally expensive for real-time renderers. Games have been using stochastic updates with heuristics for prioritization [Choi et al. 2020]. A heuristic based on the number of encountered rays has been proposed [Makarov 2023]. Stochastic updates of BLASes can lead to lighting artefacts and visible stuttering in reflections. Instead of just prioritizing updates, we build upon the idea of ray feedback to present a general culling strategy suitable for path tracing and other ray tracing effects. Even though our culling method is of particular importance for animated models, it improves performance and memory consumption even in static scenes. Especially for new methods of ray tracing highly detailed geometry, the need for such detailed culling has been described before [Benthin and Peters 2023].

Another approach to handle large scenes is lazy acceleration structure building [Hunt et al. 2007]. An adaption to GPU hardware has been presented [Lee and Liktor 2020]. However, it uses a ray tracing pipeline extension known as traversal shaders [Lee et al. 2019] that is not widely available. In addition, it suffers from performance issues by repeated tracing. Our method could be considered similar to lazy acceleration structure building, but split over the course of multiple frames.

3 METHOD

Our method is designed to enable fast ray tracing in arbitrarily large and complex scenes. We achieve this by only building BLASes when they are needed. Our pipeline therefore aims to optimize BLAS builds by batching them efficiently and running on an asynchronous compute queue in parallel to our rasterization workload [Dunn 2019], as shown in Figure 2. The instance buffer for TLAS building is assembled in each frame from scratch on the GPU just before the TLAS is built. The algorithm is described in Section 3.3. Afterwards, path tracing and denoising are run. At the end of a frame, hit feedback generated by path tracing is processed. This step is needed for our culling method and detailed in Section 3.4.

3.1 Hybrid Rendering Pipeline

Using GPUs, rasterization can solve visibility problems significantly faster than ray tracing. However, it is limited to coherent rays and only delivers approximations in certain scenarios. We employ a pipeline combining ray tracing and rasterization to combine their strengths. This is known as hybrid rendering and allows for trade-offs between visual quality and performance suited for games. Our idea is to rely on rasterization not just as a path tracing optimization but as a rendering

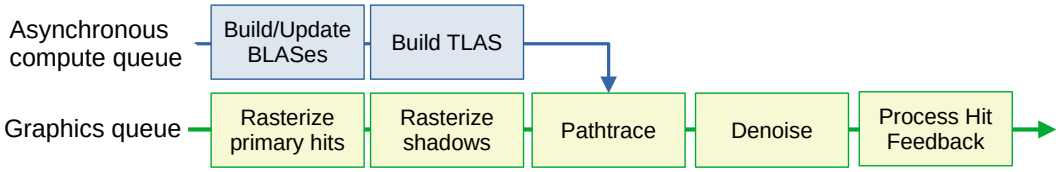


Fig. 2. Pipeline Overview.

fallback for some scene objects and specific visibility problems. This opens up the possibility of aggressive culling algorithms: Objects that are of low importance for the lighting of the scene can be completely omitted from path tracing while still remaining visible and shaded at least with the quality of a rasterizing renderer.

We rely on backward path tracing [Arvo et al. 1986] since it produces viable results even for low sample counts. We shoot one ray per pixel per frame and employ denoising to obtain a visually pleasing result [Zhdan 2021]. Backward path tracing allows to utilize rasterization for the primary rays starting at the camera. Using rasterization for these rays has no impact on visual quality but permits certain effects such as depth-of-field. While these rays are coherent and therefore comparably fast to trace on GPUs, this is still an important optimization. Especially in scenes with many non-opaque surfaces, a high number of *anyhit* shaders might be required to find intersections of rays with the scene. Using rasterization can provide significant speedups. Furthermore, rasterizing primary hits causes objects omitted from ray tracing, i.e. not present in acceleration structures, to still appear in the final image. We rasterize primary hits into *gbuffers* holding surface material information.

Rasterization can accurately provide shadowing information for punctual or directional light sources. However, ray traced shadows have shown to provide higher quality even for such light sources. They can capture smaller geometric details and do not exhibit aliasing patterns caused by shadow maps. Therefore, we utilize rasterized shadows only for some specific objects. We rasterize shadows for scene objects culled from ray tracing into a shadow map. These objects were previously deemed unimportant for the lighting of the scene, see Section 3.4. Furthermore, we rasterize shadows for all vegetation instances. Ray tracing through typical game vegetation assets involves many *anyhit* shader invocations. Rasterizing their shadows and then masking them out for shadow rays provides significant speedups. See Section 4.1 for details on the performance improvements. During path tracing, the shadow from shadow rays is combined with the shadow sampled from the shadow map.

3.2 Light Path Guided Culling

Our culling algorithm aims to remove all instances that have little impact on the lighting from the TLAS. It tracks how many light paths are encountered by each model instance during path tracing and then uses these counters for decisions during acceleration structure management. Every instance in the scene has an associated state based on the number of intersections encountered in previous frames that decides if and how it is represented in the TLAS. Areas of the scene that are rarely encountered by light paths are not added to the TLAS anymore and instead replaced with bounding boxes that only count light path intersections. Therefore, we call the method light path guided culling (LiPaC).

To gather hit feedback, the number of light paths intersecting an instance is counted in *closesthit* shaders during path tracing. Afterward, the hit counters are spatially accumulated into a coarse spatial grid managed on the GPU. The elements in the grid are called **hitboxes** and accumulate

the number of light paths that traverse the associated area. Each hitbox is marked as *active* or *inactive*. For *active* hitboxes, all model instances in their volume are added to the TLAS. *Inactive* hitboxes just add an axis-aligned bounding box (AABB) BLAS instance to the TLAS that accumulates the number of rays passing its volume. As the number of hits a single hitbox encounters passes an activation threshold ϑ_a, it switches to the *active* state. To consider whether a hitbox should be deactivated again, the accumulated hits from all its model instances are compared against a deactivation threshold, ϑ_d.

For the rendering of Figure 1 with our method, large parts of the city and most animated objects were culled from ray tracing and replaced with hitboxes. The smaller boxes in the streets are bounding boxes for animated units. Their only purpose during ray tracing is to accumulate the number of encountered light paths. Some instances outside of the view frustum are still active because they were encountered by many rays. Our implementation of LiPaC does not handle culling of terrain but adjusts the height of the hitboxes to the contained instances. Overall, this culling result could have been achieved with an aggressive simple heuristic as well. The advantage of LiPaC is that it works in scenes with more unpredictable light paths as well. It can cull instances in the frustum and near the camera if they are occluded, effectively realizing occlusion culling for ray tracing efficiently. With reflective or refractive surfaces in the scene, it will also include instances far away from the frustum, if they are needed. This can be observed in Figure 3, where a large mirroring cube was placed in the scene. LiPaC culls instances occluded by the mirror while still including instances far behind the camera, as they are visible in the mirror.

Fig. 3. Visualized hit feedback. Left: path traced views of the scene. Right: the instances present in the TLAS for the associated perspective on the left.

LiPaC realizes a culling strategy that has a much stronger foundation than traditional culling approaches in the context of ray tracing. Instead of relying on *plausible* heuristics based on distance to camera, bounding box size, solid angle or covered pixels on screen, our heuristic is built upon the number of encountered light paths as an estimator for the impact on the final lighting of the image. More sophisticated hit feedback information could improve the quality of this estimation further. An *optimal* culling algorithm would find the best trade-off between performance/memory overhead and impact to lighting of the scene. LiPaC typically achieves good results that are close to this optimal tradeoff, see Section 4.3.

3.3 TLAS Building

To realize this method efficiently, we fill the instance buffer used to build the TLAS entirely on the GPU. The TLAS instance buffer building process that is run for each frame is outlined in Algorithm 1. When the application starts rendering a scene, all hitboxes are initialized to be in the *inactive* state. Their state changes are discussed in Section 3.4. Each model instance is in one of three possible states:

active The instance has an associated BLAS that is added to the TLAS, see line 7. In practice, appending TLAS instances to the buffer is done via an atomic counter.

omitted The instance is in a region of the scene that is important for lighting the scene. However, the instance itself is not hit by many light paths and it is costly to add to the TLAS, for example, because it is animated and needs its unique BLAS. Therefore, instead of the exact geometry, a bounding box of this instance is added to the TLAS. It accumulates the number of encountered light paths to activate the instance when needed.

inactive The instance is in a region of the scene that is not encountered by many light paths, i.e. it is associated with an inactive hitbox. The instance is not added to the BLAS. Instead, the minimum and maximum height of all instances in a hitbox are evaluated in lines 16 and 17. Our algorithm requires the application to store the bounding box of each instance on the GPU. We always just associate an instance with the hitbox at the center of its bounding box, assuming that all instances are significantly smaller than hitboxes. Implementations could increase hitbox size in all dimensions to ensure it covers all instances, if needed.

Hitboxes and instance bounding boxes in the TLAS are realized by having a single, static BLAS that contains the unit cube. The transform matrix of the instance description is used to transform it as needed, as outlined in lines 10 and 23.

Afterward, a GPU thread is dispatched per hitbox, outlined by the loop in line 20. For *inactive* hitboxes, it adds a transformed unit cube instance to the TLAS. The transform will consider the height bounds previously determined by all contained instances. This box instance is needed for inactive hitboxes so that hits reaching the associated area in world space will be registered and can be accumulated. After these steps, we build the TLAS using the newly assembled buffer of instances.

During path tracing, whenever a ray hits a model instance or hitbox, the associated hit feedback counter is increased atomically. Hitboxes and bounding boxes of omitted models have a special *closesthit* shader, indicated by `HITGROUP_HITBOX` in Algorithm 1. That shader will increase the associated counter and set a bit in the ray payload returned to the *raygen* shader causing this hit to be skipped and the ray continued to be traced along its current direction. In practice, we allow each ray to only hit one hitbox, masking out all other hitbox instances in tracing after the first intersection of a ray with a hitbox.

3.4 Hit Feedback Processing

After path tracing, the accumulated hit feedback is processed on the GPU. The relevant functions are outlined in Algorithm 2. First, the number of encountered light paths is summed up from the active model instances into their spatially associated hitboxes in line 4. Then, in a second pass, the hitbox states are reconsidered: If a hitbox is marked as *inactive* but the received number of hits exceeds a threshold ϑ_a, it is marked as *active* in line 9. When the hitbox is marked as *active* but the number of hits received for the instances in its volume is below a threshold ϑ_d, it is marked *inactive* again, see line 11. See Section 3.5 for a discussion of threshold values.

In a third pass, a GPU thread is dispatched per model instance to consider if the state of this model instance needs to be changed, depending on the state of the associated hitbox and the number

Algorithm 1 Build TLAS Instance Buffer

```
1:  for every model instance M do                          ▷ Executed in parallel on GPU
2:      if M.state = active then
3:          X: TLAS instance description
4:          X.blas ← M.blas
5:          X.transform ← M.transform
6:          X.hitGroup ← M.hitGroup
7:          Append X to TLAS instance buffer
8:      else if M.state = omitted then
9:          X: TLAS instance description
10:         X.blas ← global static unit cube BLAS
11:         X.hitGroup ← HITGROUP_HITBOX
12:         Set X.transform such that it maps unit cube to M.aabb
13:         Append X to TLAS instance buffer
14:     else if M.state = inactive then
15:         Get hitbox H at position of M
16:         atomicMin(H.minY, M.aabb.min.y)
17:         atomicMax(H.maxY, M.aabb.max.y)
18:     end if
19: end for

20: for every hitbox H do                                  ▷ Executed in parallel on GPU
21:     if H.state = active ∧ H.minY < H.maxY then
22:         X: TLAS instance description
23:         X.blas ← global static unit cube BLAS
24:         X.hitGroup ← HITGROUP_HITBOX
25:         Set X.transform such that it maps unit cube to H bounds
26:         Append X to TLAS instance buffer
27:     end if
28: end for
```

of encountered light paths. On state change, an encoded command value is appended to a buffer intended for CPU reading, as seen in lines 17, 25, 31, and 34. The buffer holding the commands is afterward copied and processed on the CPU. For activated model instances, it is ensured that they have a valid BLAS associated. For model instances that are deactivated, associated resources such as unique BLASes are deallocated. As mentioned above, model instances with a unique BLAS are handled separately. When the associated hitbox is activated, they are put into the *omitted* state first, as seen in line 23. It is only activated when the bounding box added to the TLAS (see Algorithm 1 line 13) encounters a number of light paths above a threshold σ_a, as shown in line 32.

3.5 Culling Heuristic

We use threshold values of $\vartheta_a = 1000$, $\vartheta_d = 100$, $\sigma_a = 1000$, $\sigma_o = 100$. These values were empirically determined for our path tracing algorithm at 1080p. Setting the activation threshold significantly higher than the deactivation threshold proved a good setup to avoid frequent state cycles caused by many rays hitting the hitbox when deactivated but few rays hitting the actual instance when activated. Additionally, we only activate hitboxes when their hit counter has exceeded the threshold for $n_a = 4$ frames, or exceeded $2 \times \vartheta_a$ in a single frame. We have chosen to deactivate hitboxes

Algorithm 2 Process Hit Feeback

```
1:  for each model instance M do                                  ▷ Executed in parallel on GPU
2:      if M.state = added ∨ M.state = omitted then
3:          Get hitbox H at position of M
4:          atomicAdd(H.hitcount, M.hitcount)
5:      end if
6:  end for

7:  for each hitbox H do                                          ▷ Executed in parallel on GPU
8:      if H.state = inactive ∧ H.hits > ϑ_a then
9:          H.state ← active
10:     else if H.state = active ∧ H.hits < ϑ_d then
11:         H.state ← inactive
12:     end if
13: end for

14: for each model instance M do                                  ▷ Executed in parallel on GPU
15:     Get hitbox H at position of M
16:     if M.state = added ∧ H.state = inactive then
17:         Append remove(M) command to readback buffer
18:         M.state ← removed
19:     else if M.state = omitted ∧ H.state = inactive then
20:         M.state ← removed
21:     else if M.state = removed ∧ H.state = active then
22:         if M needs a unique BLAS then
23:             M.state ← omitted
24:         else
25:             Add add(M) command to readback buffer
26:             M.state ← added
27:         end if
28:     end if
29:     if H needs unique BLAS then
30:         if M.state = omitted ∧ M.hitCount > σ_a then
31:             Add add(M) command to readback buffer
32:             M.state ← added
33:         else if M.state = added ∧ M.hitCount < σ_o then
34:             Add remove(M) command to readback buffer
35:             M.state ← omitted
36:         end if
37:     end if
38: end for
```

only after their hit counter is below ϑ_d for $n_d = 15$ frames. While this further reduces unnecessary cycles, it can increase memory consumption for extended periods when the camera is moving quickly. Furthermore, we still observed unnecessary state cycles in cases where the bounding box approximates the mesh poorly. Resolution and path length impact the number of rays encountered by objects, thresholds could be adjusted dynamically to account for this. Importance sampling

strategies can also impact the number of intersections per object. This can be seen in Figure 3. We use importance sampling of the specular BRDF and therefore only objects visible in the mirror get activated. Light path guiding algorithms [Lu et al. 2024] and re-use approaches like ReSTIR [Bitterli et al. 2020] behave in a similar way. They trace rays more often that are expected to have a bigger influence on the lighting of the scene. This only improves the effectiveness of LiPaC. However, care has to be applied when mixing LiPaC with such approaches. In theory, when certain paths are sampled rarely, objects on it might not get activated, thus changing inputs for future guiding, possibly leading to a negative feedback loop.

3.6 Latency Mitigation

Our method is explicitly designed to be GPU-driven. This has a major advantage: No CPU-processing of all (visible) model instances in a scene is required. This allows us to maintain many millions of model instances. On the GPU side, iterating over all instances in the scene, no matter if visible or not is no performance concern in practice.

However, one problem of the feedback approach is the latency between detecting that instances should be active and having them influence the lighting of the final image. The GPU-driven nature of our method amplifies this problem. The readback buffer containing which instances to add or remove can only be read on the CPU side later when execution on the GPU is terminated. For instance, when the camera suddenly jumps to a new position, no instance at this position might be added to the TLAS. In the first frame that is path traced at this new position, all hitboxes will receive high numbers of hits and thus activate all of their model instances. While this means that instances could already be active in the next frame, their associated BLASes might not have been built yet. In that case, they will only become visible after the readback buffer has been processed on the CPU and their BLASes have been created. This will potentially cause instances to appear with a delay of multiple frames after the camera position, viewing direction, or elements in the scene change.

An extension of our method solves this worst case: On the CPU we ensure for each frame that all instances newly appearing in the frustum have an associated BLAS and are marked as active. This check also covers newly created instances in the frustum. While this might activate too many instances, unimportant ones are quickly made inactive again by hit feedback processing. Therefore, we do not have any latency at all for objects directly in the view frustum. On the other hand, significant camera changes might lead to spikes in the number of active instances and BLAS builds. Thus, an additional heuristic based on instance distance to camera and size selects only instances considered crucial for indirect lighting. Furthermore, the visible latency of several frames still persists for appearing instances that are not in the frustum and are only viewed through mirroring or refracting surfaces.

4 EVALUATION

We test our method on a machine with an AMD Ryzen 5900x, 64 GB of RAM and Windows 10, using Direct3D 12. Rendering performance of various GPUs is tested, as stated per test case. The renderer is written in C++ and compiled using MSVC 14.36. We evaluate our technique for a large and dynamic city scene from a current generation city building game, see Figure 1. Path tracing in city building games is especially challenging due to user-built scenes with many objects, fast camera movement through the scene, and a high degree of dynamic elements. Many of the buildings are animated or uniquely deformed to fit the terrain they are placed on. Our test scene contains additional vegetation in the streets and more than six thousand animated units. The ground truth path tracing technique we compare our method against does not employ any rasterization but includes denoising and all of our path tracing optimizations such as Russian Roulette termination [Rath et al. 2022]. It relies

on a trivial culling heuristic that conservatively includes instances based on their distance to the camera and bounding box size.

4.1 Runtime

On capable GPUs of the current generation, our method allows path tracing even large game scenes in real-time at 1080p, see Figure 4 for path tracing time comparison in a static scene. We use *frame time* to describe the time required by the GPU to render a single frame. On the most capable consumer hardware, path tracing with our method is even possible in 60 fps or higher resolutions. However, less capable and older GPUs do not achieve real-time capable frame rates even with aggressive culling. Lowering the resolution or falling back to specialized ray tracing effects that do not solve the entire rendering equation allows relatively high lighting quality even on such GPUs.

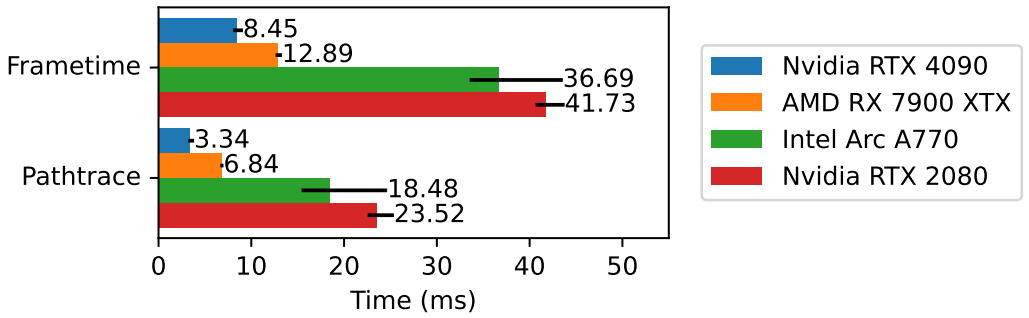


Fig. 4. Total frame time and the path tracing portion of it on various GPUs. City scene without animated units rendered at 1080p.

While LiPaC improves performance and memory consumption in static scenes, its impact is even bigger in highly dynamic scenes. For the scene in Figure 1, it is responsible for ensuring real-time performance.

Speedup of path tracing is achieved by rasterizing primary hits and vegetation shadow, see Figure 5. Rasterizing primary hits and using rasterized directional shadows for vegetation as described in Section 3.1 decreases path tracing time by a comparable speedup each. Enabling LiPaC slightly increases path tracing timings by writing out hit feedback. Shadow rays cannot just accept the first hit and rays have to be retraced when they encounter a hitbox as closest hit. In our implementation, every single intersection increases the respective hit counter. We could not notice any performance improvements by accumulating only some intersections stochastically. In the scene with many animated units, employing LiPaC reduces BLAS building time per frame from around 16ms to below 0.1ms, thus reducing overall frame time significantly. With LiPaC enabled, rasterizing primary hits for the entire scene and rasterizing shadows of omitted instances is required. The GPU time needed for assembling the TLAS buffer as described in Section 3.3 and hit feedback processing from Section 3.4 was negligible even with more than a hundred thousand instances. Timings never exceeded 0.1ms respectively on the AMD RX 7900 XTX.

Avoiding frame timing spikes is important to ensure a smooth experience in interactive renderers. LiPaC lazily activates instances and builds BLASes that are needed for indirect lighting of the scene. Thus, during camera movement, many BLASes are built and GPU buffers are updated. To evaluate this impact, we measured timings while quickly moving the camera through the scene, see Figure 6. In its initial resting pose, the camera is high above the city, looking down. Then, it moves down,

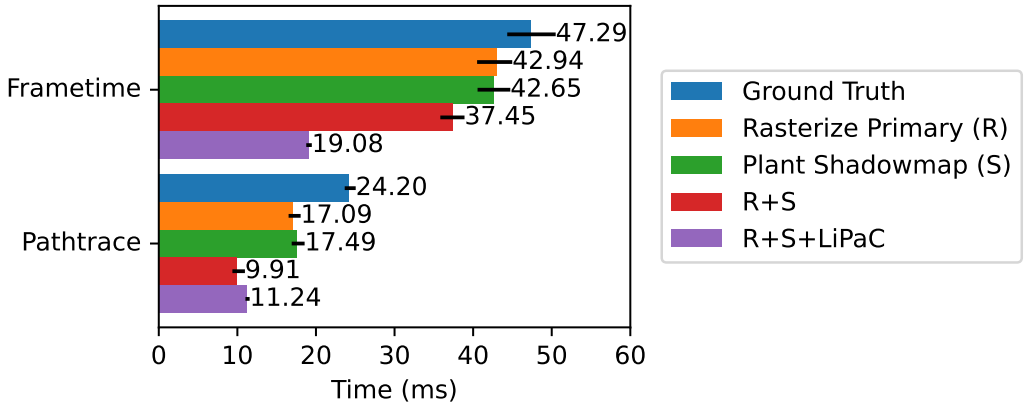


Fig. 5. Impact of the various optimizations in our method. Animated city scene from Figure 1. 1080p, AMD RX 7900 XTX. Our optimizations achieve a speedup factor greater than two.

hovering close to a busy street. Thereafter, it zooms slightly out and moves horizontally through the city. Finally, the camera is tilted to look toward the horizon, its final resting pose. All this happens in less than ten seconds, as the camera is moving quickly.

Process BLAS commands measures the CPU time spent in each frame processing the commands written out by hit feedback processing on the GPU, possibly (de-)activating instances. *Handle appearing instances* measures the CPU timing of the latency mitigation approach described in Section 3.6. This step is optional to avoid latency inside the frustum on quick camera movements. *Update Raytrace Buffers* describes the CPU time needed to record updates of our GPU data structures, e.g. for new, changed or (de-)activated instances. *Build BLASes* measures the time needed to record BLAS build commands on the CPU and execute them on the GPU, respectively. Frame timing spikes can be observed. They could be reduced by processing a smaller number of commands in each frame, delaying BLAS builds over multiple frames, or not including the additional check for appearing instances inside the frustum on CPU side. Note that these optimizations would come at the cost of a potentially increased latency.

4.2 Memory Consumption

Memory consumption of ray tracing resources is dominated by BLASes and TLAS. In large scenes such as cities, meshes are often used multiple times throughout the scene, keeping memory consumption of mesh buffers reasonable. In that case, the BLASes can be shared between instances and inserted into the TLAS multiple times. However, when instances are animated or otherwise uniquely transformed, this is not possible anymore. In our evaluation setting, buildings might transform dynamically to the terrain they are placed on, requiring a unique BLAS per instance instead of per mesh in many cases. Additionally, the many animated units each require their own BLAS as well. LiPaC is important in such a case to achieve low memory consumption, see Figure 7. LiPaC reduces the number of instances and therefore TLAS memory consumption. Some meshes are not required at all, reducing the memory occupied by shared BLASes as well. But especially the memory consumed by unique BLASes is almost completely freed with LiPaC. In our test case, unique BLASes were created for all the allocated units but also for parts of some buildings.

In total, our culling method reduces graphics memory needed for ray tracing resources on the AMD RX 7900 XTX from 496 MiB to 173 MiB, thus achieving a reduction factor greater than 2.5.

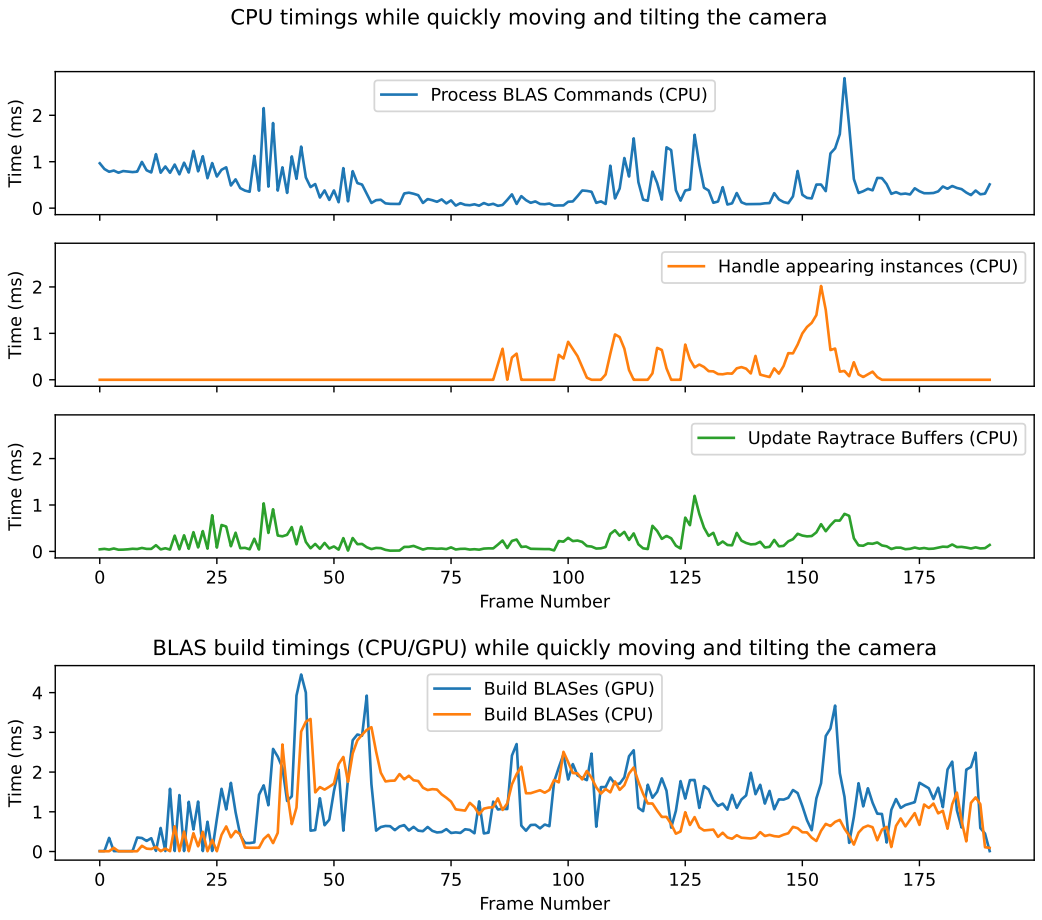


Fig. 6. The most relevant acceleration structure management GPU and CPU timings while moving the camera through the city scene. AMD RX 7900 XTX at 1080p.

The data structures introduced by LiPaC consume little memory. Our hit feedback buffer requires 4 byte per model instance, thus consuming just 4 MiB for 1 million model instances. We require 12 bytes per hitbox and have each hitbox cover roughly 16x16m in world space. Even with a scene size of 8x8km, this amounts to a memory consumption of just 3 MiB for hitboxes.

4.3 Visual details

While our method of omitting costly model instances allows us to reduce consumed memory and BLASes that need to be updated or rebuilt each frame, it impacts lighting by omitting scene elements from path tracing. Thus, the method can be seen as a trade-off between visual quality and memory/performance constraints. Instances encountered by a small number of light paths, such as animated models in the distance, are not crucial for the indirect lighting of the scene but have a significant cost, making this trade-off reasonable in the context of real-time path tracing.

With a hybrid pipeline, we can ensure that omitted elements indeed only impact indirect lighting. Using overly high activation thresholds (σ_a = 5000), the indirect lighting impact can be noticed in

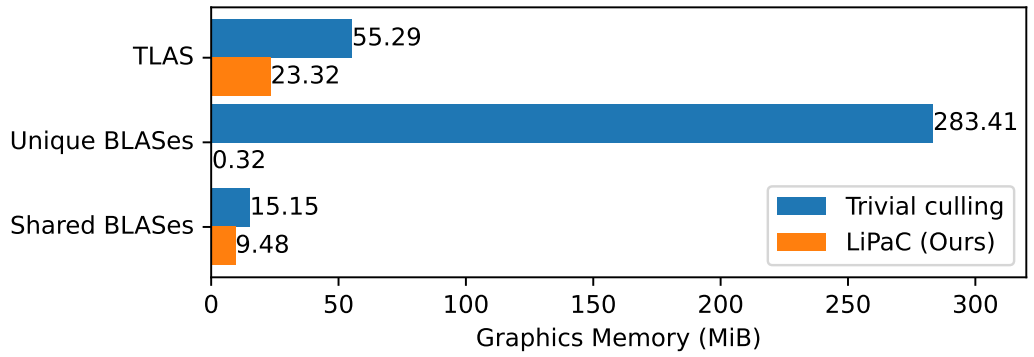


Fig. 7. GPU memory consumption for city scene with high number of animated units. AMD RX 7900 XTX at 1080p.

Fig. 8. Comparison of denoised diffuse irradiance. Left: Ground Truth. Right: LiPaC with high activation thresholds.

Figure 8. With LiPaC, animated instances near the camera occlude the ground correctly (green box). However, the animated models in the back do not influence lighting of the ground when LiPaC is enabled (orange box). Interestingly, by starting path tracing at the rasterized primary hits, the instances omitted from ray tracing still receive accurate lighting from their environment, it is just their impact on the indirect lighting of the scene that is culled.

Omitted models are rasterized into a shadow map that is sampled during path tracing in addition to shadowing rays. Using once again overly high activation thresholds, this effect is visualized in

Figure 9. Only animated models in the distance will throw rasterized shadows (orange box), other shadows are path traced (green box). Missing contact shadows can be noticed when sampling from the shadow map. While the rasterized shadow is of lower quality than a path traced shadow and only possible for simple light sources, it ensures a visual quality baseline.

Fig. 9. Lighting and shadow comparison. Left: Ground Truth. Right: LiPaC with high activation thresholds, falling back on rasterized shadows for omitted models.

In comparison to existing prioritization methods for BLAS updates, our hybrid path tracing method will ensure that rays are never traced against outdated BLASes. Methods just omitting updates [Choi et al. 2020; Makarov 2023] can easily introduce artefacts by a mismatch between rasterized and ray-traced geometry. Our method will never encounter this case. The rasterized geometry is a strict super-set of the geometry present in the acceleration structures. Instances are either fully and correctly considered for ray tracing or not at all. And, depending on the heuristic chosen to fulfill memory and performance constraints, the latter case might only happen for instances that are almost irrelevant to the lighting of the scene anyways.

5 CONCLUSION

We described a hybrid path tracing pipeline achieving real-time performance on commodity hardware, even for difficult cases of current generation games. We introduced LiPaC, a new culling method for real-time ray tracing that leverages the advantages of a hybrid pipeline. LiPaC significantly reduces the acceleration structure memory consumption in large scenes. Especially for highly dynamic scenes, it is crucial to achieve real-time performance and fit acceleration structures into the graphics memory of commodity hardware. Utilizing rasterization allows to cull aggressively with only minor impact on the final image. While our method focuses on path tracing, LiPaC can be combined with specialized ray tracing methods as well. It allows to employ real-time ray tracing methods even more broadly, allowing games to unlock beautiful visuals on a wide range of hardware.

Even with our method, real-time path tracing is still limited to the powerful current generation GPUs. However, future GPU generations can be expected to allow it even on average consumer hardware. More powerful GPUs and acceleration structure building algorithms will likely still benefit from efficient culling algorithms but allow for more detailed geometry or lower activation thresholds.

5.1 Limitations

Latency. Our method has an inherent latency that can be observed in some cases. The extension of our method described in Section 3.6 ensures that all objects appearing in the frustum in a given frame contribute to the lighting of the scene. However, objects outside of the frustum that contribute significantly to the image will only be considered with a delay of multiple frames after a camera jump or fast movement. For instance, objects only visible in mirrors or large occluders outside of the frustum might cause undesirable pop-in artefacts.

Non-trivial light sources. While rasterization of shadow maps ensures an approximation of shadows for instances culled from ray tracing, shadow maps only work for trivial light sources. Rasterizing shadows for light-emissive surfaces is not possible in the general case. Culled instances will not have any shadows for such light sources, possibly leading to pop-in artefacts.

5.2 Future Work

Combination with other path tracing methods. Screen space path tracing [Willberger et al. 2019] could be used for the first bounce with a fallback to proper ray tracing. Our method already produces rasterized buffers with material information for the primary rays. Besides a reduction in tracing times, using screen space information could add indirect lighting details even for instances omitted from ray tracing. Our culling method could also be adjusted to work with bidirectional path tracing or photon mapping techniques.

Advanced spatial data structures. Instead of the grid of boxes used to cull entire scene areas, more advanced data structures could be used. For instance a quadtree [Samet 1984] spanning the scene could be subdivided dynamically depending on the light paths reaching its nodes. This could improve performance and memory consumption overhead of LiPaC in very large scenes by successively scaling hitboxes in areas never encountered by any light paths.

Improved heuristics. Activation and deactivation thresholds could be coupled to memory and TLAS/BLAS build time budgets. Shadow rays to light sources could consider whether the light source has rasterized shadows to ensure instances occluding non-trivial light emitters are given a higher priority. Instead of just counting rays, ray type and ray differentials could be used to estimate the contribution of rays and missing instances to the final image. When limiting the number of BLAS builds per frame, LiPaC could provide prioritization based on the number of encountered light paths.

More nuanced instance states. Instead of the three presented instance states, our culling algorithm could provide finer differentiation. Animated instances hit by a low number of light paths could instead be represented by a static approximative representation for ray tracing, possibly while still rasterizing their shadows. Hit feedback information can also be used to drive mesh level of detail (LOD) selection.

Evaluation with opacity micromaps. The presented pipeline is especially tailored to allow for efficient path tracing of vegetation assets, avoiding some overhead caused by the high number of *anyhit* shader invocations. The recent advancement of opacity micromaps [Fenney and Ozkan

2023] allows more efficient ray tracing of non-opaque geometry, possibly rendering the presented trade-offs suboptimal.

ACKNOWLEDGMENTS

We extend gratitude to the Anno team of Ubisoft Mainz for allowing us to use the city scene for evaluation. We thank the Anno Research and Development team, especially Aaron Scherzinger for their feedback and discussions.

REFERENCES

James Arvo et al. 1986. Backward ray tracing. In *Developments in Ray Tracing, Computer Graphics, Proc. of ACM SIGGRAPH 86 Course Notes*. 259–263.

Louis Bavoil, Edward Liu, Peter Shirley, and Morgan McGuire. 2018. Hybrid Ray-Traced Ambient Occlusion. , 4 pages. https://casual-effects.com/research/Bavoil2018AO/index.html Poster at HPG18 ACM SIGGRAPH / Eurographics High Performance Graphics.

A. Beacco, N. Pelechano, and C. Andújar. 2016. A Survey of Real-Time Crowd Rendering. *Computer Graphics Forum* (2016). https://doi.org/10.1111/cgf.12774

Carsten Benthin and Christoph Peters. 2023. Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail. *Computer Graphics Forum* 42, 8 (2023), e14868. https://doi.org/10.1111/cgf.14868 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14868

Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. 2020. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 39, 4 (7 2020). https://doi.org/10/gg8xc7

Jakub Boksansky, Michael Wimmer, and Jiri Bittner. 2019. *Ray Traced Shadows: Maintaining Real-Time Frame Rates: High-Quality and Real-Time Rendering with DXR and Other APIs*. 159–182. https://doi.org/10.1007/978-1-4842-4427-2_13

Stephanie Brenham and Ihor Szlachtycz. 2022. *Performant Reflective Beauty: Hybrid Ray Traced Reflections In Far Cry 6*. https://gpuopen.com/gdc-presentations/2022/GDC_Performant_Reflective_Beauty_Hybrid_Ray_Traced_Reflections_In_Far_Cry_6.pdf

Andrew Burnes. 2023. *Cyberpunk 2077: Technology Preview Of New Ray Tracing Overdrive Mode Out Now*. https://www.nvidia.com/en-us/geforce/news/cyberpunk-2077-ray-tracing-overdrive-update-launches-april-11/

Jiho Choi, Jim Kjellin, Patrik Willbo, and Dmitry Zhdan. 2020. *Ray Traced Reflections in 'Wolfenstein: Youngblood'*. https://www.gdcvault.com/play/1026723/Ray-Traced-Reflections-in-Wolfenstein

Johannes Deligiannis and Jan Schmid. 2019. *"It Just Works": Ray-Traced Reflections in 'Battlefield V'*. https://www.gdcvault.com/play/1026282/It-Just-Works-Ray-Traced

Alex Dunn. 2019. *Tips and Tricks: Ray Tracing Best Practices*. https://developer.nvidia.com/blog/rtx-best-practices/

Simon Fenney and Alper Ozkan. 2023. Compressed Opacity Maps for Ray Tracing. In *High-Performance Graphics - Symposium Papers*, Jacco Bikker and Christiaan Gribble (Eds.). The Eurographics Association. https://doi.org/10.2312/hpg.20231133

Holger Gruen. 2020. *Are we done with hardware ray tracing?* https://www.highperformancegraphics.org/slides20/monday_gruen.pdf

Tobias Hector, Joshua Barczak, and Eric Werness. 2020. *Ray Tracing In Vulkan*. https://www.khronos.org/blog/ray-tracing-in-vulkan

Warren Hunt, William Mark, and Don Fussell. 2007. Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. *IEEE/EG Symposium on Interactive Ray Tracing 2007*, 47–54. https://doi.org/10.1109/RT.2007.4342590

James T. Kajiya. 1986. The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*. Association for Computing Machinery, New York, NY, USA, 143–150. https://doi.org/10.1145/15922.15902

Timothy L. Kay and James T. Kajiya. 1986. Ray Tracing Complex Scenes. *SIGGRAPH Comput. Graph.* 20, 4 (8 1986), 269–278. https://doi.org/10.1145/15886.15916

Won-Jong Lee and Gabor Liktor. 2020. Lazy Build of Acceleration Structures with Traversal Shaders. In *SIGGRAPH Asia 2020 Technical Communications* (Virtual Event, Republic of Korea) *(SA '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 4 pages. https://doi.org/10.1145/3410700.3425430

Won-Jong Lee, Gabor Liktor, and Karthik Vaidyanathan. 2019. Flexible Ray Traversal with an Extended Programming Model. In *SIGGRAPH Asia 2019 Technical Briefs* (Brisbane, QLD, Australia) *(SA '19)*. Association for Computing Machinery, New York, NY, USA, 17–20. https://doi.org/10.1145/3355088.3365149

Haolin Lu, Wesley Chang, Trevor Hedstrom, and Tzu-Mao Li. 2024. Real-Time Path Guiding Using Bounding Voxel Sampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 43, 4 (2024). https://doi.org/10.1145/3658203

Zander Majercik, Jean-Philippe Guertin, and Morgan McGuire. 2019. Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields. In *Journal of Computer Graphics Techniques Vol. 8, No. 2.* https://www.jcgt.org/published/0008/02/01/paper-lowres.pdf

Evgeny Makarov. 2023. *Practical Tips for Optimizing Ray Tracing.* https://developer.nvidia.com/blog/practical-tips-for-optimizing-ray-tracing/

Alexander Rath, Pascal Grittmann, Sebastian Herholz, Philippe Weier, and Philipp Slusallek. 2022. EARS: efficiency-aware russian roulette and splitting. *ACM Trans. Graph.* 41, 4, Article 81 (jul 2022), 14 pages. https://doi.org/10.1145/3528223.3530168

Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (6 1984), 187–260. https://doi.org/10.1145/356924.356930

Christoph Schied. 2019. *Video Series: Path Tracing for Quake II in Two Months.* https://developer.nvidia.com/blog/path-tracing-quake-ii/

Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/3105762.3105770

Juha Sjoholm. 2020. *Best Practices: Using NVIDIA RTX Ray Tracing (Updated).* https://developer.nvidia.com/blog/best-practices-using-nvidia-rtx-ray-tracing/

Thomas Willberger, Clemens Musterle, and Stephan Bergmann. 2019. *Deferred Hybrid Path Tracing: High-Quality and Real-Time Rendering with DXR and Other APIs.* 475–492. https://doi.org/10.1007/978-1-4842-4427-2_26

Chris Wyman, Shawn Hargreaves, Peter Shirley, and Colin Barré-Brisebois. 2018. Introduction to DirectX Raytracing. In *ACM SIGGRAPH 2018 Courses* (Vancouver, British Columbia, Canada) *(SIGGRAPH '18)*. Association for Computing Machinery, New York, NY, USA, Article 9, 1 pages. https://doi.org/10.1145/3214834.3231814

Dmitry Zhdan. 2021. *ReBLUR: A Hierarchical Recurrent Denoiser.* Apress, Berkeley, CA, 823–844. https://doi.org/10.1007/978-1-4842-7185-8_49