DOI: 10.1111/cgf.14882

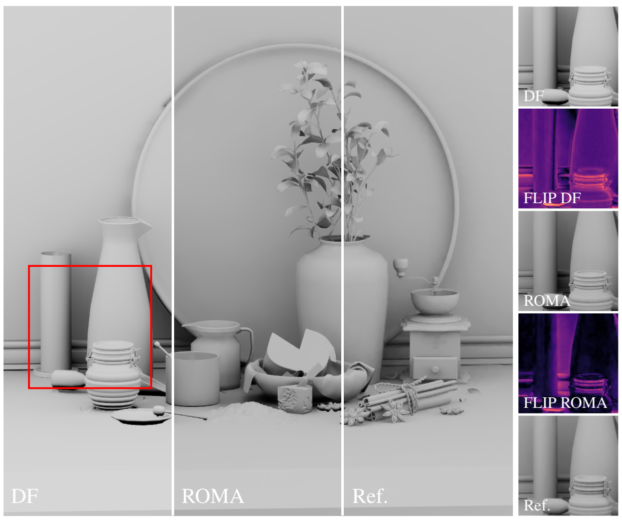
Eurographics Symposium on Rendering 2023
T. Ritschel and A. Weidlich
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 4
e14882 *pp. 1 - 14*

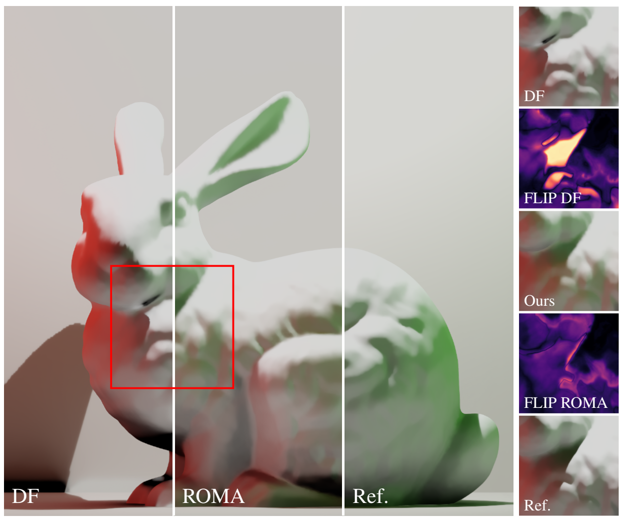
Ray-aligned Occupancy Map Array for Fast Approximate Ray Tracing

Z. Zeng[1], Z. Xu[2], L. Wang[†2], L. Wu[3] and L. Yan[1]

[1]University of California, Santa Barbara, USA
[2]Shandong University, China
[3]NVIDIA, USA



(a) *Ambient occlusion.*



(b) *One-bounce indirect illumination.*

Figure 1: *Our Ray-aligned Occupancy Map Array (ROMA) provides fast and approximate ray tracing. We demonstrate its applications in (a) ambient occlusion and (b) one-bounce indirect illumination. Our ROMA is implemented without any specific hardware units for acceleration, but performs comparably fast as hardware ray tracing. And compared to distance functions (DF) with the same resolution and equal storage, our method is about* $2.5\times$*–*$10\times$ *faster in generation and tracing, and achieves better quality.*

Abstract

We present a new software ray tracing solution that efficiently computes visibilities in dynamic scenes. We first introduce a novel scene representation: ray-aligned occupancy map array (ROMA) that is generated by rasterizing the dynamic scene once per frame. Our key contribution is a fast and low-divergence tracing method computing visibilities in constant time, without constructing and traversing the traditional intersection acceleration data structures such as BVH. To further improve accuracy and alleviate aliasing, we use a spatiotemporal scheme to stochastically distribute the candidate ray samples. We demonstrate the practicality of our method by integrating it into a modern real-time renderer and showing better performance compared to existing techniques based on distance fields (DFs). Our method is free of the typical artifacts caused by incomplete scene information, and is about $2.5\times$*–*$10\times$ *faster than generating and tracing DFs at the same resolution and equal storage.*

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Rendering;***

1. Introduction

Photorealistic rendering is becoming increasingly crucial for real-time applications such as video games, virtual reality, and visual-

† Corresponding author: luwang_hcivr@sdu.edu.cn.

izations. It is generally believed that employing *ray tracing* over *rasterization* is the key to achieving photorealism, which is provably easier and more consistent to simulate global shading effects such ambient occlusion and indirect illumination.

However, with only *hardware ray tracing (HWRT)* using specific hardware acceleration support, one can barely achieve real-time frame rates. More crucially, even today, only high-end platforms support HWRT. So we still need a HWRT alternative, *software ray tracing (SWRT)* solution, as a reasonable approximation to HWRT for providing users with the option to scale down: allowing trade-offs between quality and performance for low-end platforms, e.g., mobile devices and VR headsets.

A widely used SWRT solution is *screen space ray tracing* [BS08; RGS09; Ulu18]. It uses *depth maps*, which are easily acquired from G-buffers, as the proxy scene representation and achieves fast tracing performance with the mip-map support. However, it will miss the geometries out of the screen and behind the nearest surfaces, because the depth map merely provides information about the nearest boundary of scene geometry to the camera. Using screen space ray tracing usually leads to visible artifacts.

Another popular alternative is *distance fields (DFs)* coupled with *sphere tracing* [TDD*22]. DFs provide a good approximation to the scene geometry, and sphere tracing enables fast ray tracing against DFs. However, the major drawback of this solution is that DFs are costly to generate ($1.7\times$–$11.0\times$ slower than our method; see Table 2) and therefore are hard to support dynamic scenes. Moreover, as we will demonstrate later, tracing against DFs is not as fast as expected.

Unlike DFs, *occupancy maps (OMs)* serve as an approximate scene representation which can be generated efficiently from scene geometry [ED06]. By placing a camera and rasterizing the scene, it discretizes the scene into *binary* grid cells, indicating whether each cell contains any geometry or not. Every 32 cells along the camera's z-axis are encoded into a 32-bit integer. In this way, a 3D OM can be compactly represented by a 2D texture map. However, tracing rays against the OM representation is challenging. Thiedemann et al. [THGM11] propose a tracing method that exploits the nature that the cells are encoded in bits: they trace rays in the texels of an OM following the 2D DDA algorithm [AW*87] and compute the visibility along a ray using bit operations. But their method suffers from slow tracing since multiple and inconsistent steps have to be performed to trace a ray.

Our goal is to develop an SWRT solution that achieves fast generation and tracing simultaneously. Inspired by Thiedemann et al. [THGM11], we propose a new SWRT solution built on a novel *ray-aligned occupancy map array (ROMA)*. It combines the advantages of the above solutions: fast generation of the required information from scene geometry (e.g., generating a ROMA with 128^3 resolution takes 0.3 ms, while generating a DF with the same resolution takes 3.3 ms) thus effectively supporting dynamic scenes; fast tracing that is about $5\times$ faster than DFs using sphere tracing. Also, our solution is fully scalable and is easy to integrate with a spatiotemporal scheme to improve performance further and avoid aliasing.

Our key observation is that the *thread divergence* caused by the inconsistent number of algorithm iterations seriously impacts performance. When tracing against the OM in the 2D DDA style, the number of iterations is associated with the projected length of ray onto the texture plane. Hence, our insight is, what if we only have to trace rays along the camera's z-axis? Ray tracing then becomes extremely simple—a ray only travels along a 1D bit array (with zero projected length onto the texture plane). In this way, not only a constant one-step tracing can be achieved (Sec. 4), but the thread divergence can also be minimized.

To achieve this, we need to generate a group of OMs whose z-axes are aligned with all possible ray directions. In practice, we presample an array of uniformly distributed *candidate directions*, and generate ray-aligned OMs only along these directions. During rendering, for any given ray, we replace its direction with the closest candidate direction to achieve fast and low-divergence ray tracing that takes exactly one step. We present a fast generation scheme: instead of rasterizing the scene multiple times, we "rotate" a well-generated *base occupancy map* (BOM) to create our ROMA.

Finally, we provide a fully scalable solution, by tuning the resolution of OMs (positional resolution) and the number of candidate directions (angular resolution). Inspired by Temporal Anti-Aliasing (TAA) [TKD*14], we use jittered samples for both camera pixels when generating the BOM and the candidate directions of ROMA over different frames. This spatiotemporal support not only provides good anti-aliasing but also enables a fully scalable solution to further improve our performance.

In summary, our main contributions are as follows:

- a new SWRT solution that enables fast approximate ray tracing,
- a novel scene representation with spatiotemporal support that offers more options to trade off performance and quality,
- a fast generation method for the new scene representation that effectively supports dynamic objects with deformation or animation, and
- a fast tracing method that enables non-hierarchical and low-divergence tracing in $O(1)$ time.

2. Related Work

Real-time ray tracing for visibility. The occupancy map (OM) is widely used to improve occlusion queries [SBM03] or simulate hair self-shadowing [SA09]. It is also referred to as *voxel bit bricks* in the industry [TDD*22]. Generation of occupancy maps from scene is pervasively studied in *scene voxelization*: it turns scene geometries into a 3D uniform grid and encodes each cell with specific scene information such as occupancy, lighting, or material. Eisemann et al. [ED06] propose a fast binary scene voxelization using rasterization with graphics hardware. It voxelizes and encodes the boundary of scene geometries into bits and save them on a 2D texture. Dong et al. [DCB*04] present a similar approach with three textures are generated to address the problem of holes appears on geometries parallel to viewing direction. Forest et al. [FBP09] propose a novel tracing method built upon the voxelization by first converting scene geometries into a voxel octree and using an adaptive bitmask to directly eliminate the nodes that potentially have intersections and refine the octree. Thiedemann et al. [THGM11] follow this idea and propose to drop the octree and directly tracing against the 2D texture. They also propose a mip-map scheme to accelerate the tracing. While these methods successfully explore fast generation of occupancy map from scene geometries, the tracing method still remains inefficient. Instead, our method propose tracing against a novel ray-aligned occupancy maps array (ROMA) to achieve fast and low-divergence tracing.

Distance fields (DFs) is another form of scene representation widely used in computer graphics. It can be used for ray tracing [TDD*22; LM22], in which case it is an approximation to the scene geometries and typically generated using *jump flooding algorithm (JFA)* [RT06] with occupancy maps as seeds. *Sphere tracing* is used to perform fast tracing against DFs to find nearest intersection. However, due to the complexity of JFA, DFs are generally considered costly to update on a per-frame basis and thus hard to support dynamic objects. In contrast, we target at fast generation and strong support for dynamic objects while keeping tracing reasonably fast and low-divergence. DFs can also be employed to compose neural implicit representations [TLY*21; SJ22; MESK22]. However, the inference performance limits its practicality in real-time applications where 1ms is considered prohibitively expensive.

Real-time global illumination. There are many approaches to GI achieving real-time frame rates by utilizing rough approximations of scene geometry, visibility, lighting, or materials. Crassin et al. [CNS*11] propose to generate a hierarchical voxel octree representation on the fly from scene geometries and use cone tracing to estimate visibility and incoming energy. Ritschel et al. [RGK*08] propose a coarse approximation to scene visibility coupled with *virtual point lights (VPLs)* to achieve real-time GI. Each VPL generates an *imperfect shadow map* from a subset of the rough point-based representation of the scene. This inspires us that it can be practical to generate our ray-aligned occupancy map for each possible direction. Ritschel et al. [REG*09] propose to perform final gathering at visible surface on *micro-buffer*, which is rasterized from a hierarchical point-based scene representation. Szirmay-Kalos et al. [SP98] propose global ray-bundles to first rasterize scene surfaces into planar patches for a group of global ray directions, and then exchange radiance between visible patches to approximate global illumination. Hermes et al. [HHGM10] further combine global ray-bundles with a k-buffer [BM08] to achieve radiance exchanges for all patches of the scene instead of the visible ones only. Recent work on irradiance probes [MGNM19; MMSM21; MMK*21] approximate dynamic global illumination with precomputed probes in a 3D grid and is ready to use inside the NVIDIA RTXGI SDK.

That said, our method focuses on computing visibility queries and is orthogonal to real-time GI techniques. In Section 7, we demonstrate that our method can be integrated with one of the real-time GI techniques and computes single-bounce diffuse indirect illumination efficiently.

Spatiotemporal scheme. Using a spatiotemporal scheme to distribute computational workload spatially and temporally to improve performance while achieving good quality has became a defacto solution to many applications. Temporal Anti-Aliasing (TAA) [TKD*14] jitters the sample position in pixels in current frame, and blend them with pixel values from previous frames to produce anti-aliased result. Reservoir-based SpatioTemporal Importance Resampling (ReSTIR) [BWP*20] generally only sample one candidate for each reservoir and then spatiotemporally combines reservoirs to decrease variance. Similar to these methods, we also employ a spatiotemporal scheme to further improve performance while avoid significant alias, using a small angular resolution (the number of OMs in ROMA) for each frame with different candidate directions for different frames. Note specifically that being able to employ spatiotemporal approaches to distribute the workload is unique to our method – the trait of fast generation makes this practically possible.

3. Background and Problem Analysis

In this section, we briefly review the occupancy map technique and explain how to trace rays against occupancy maps to perform visibility tests.

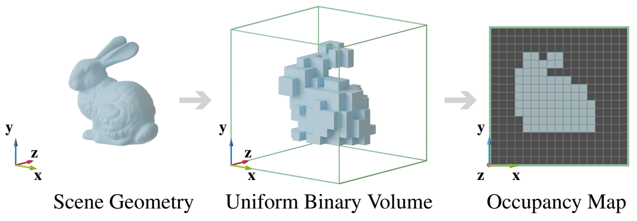


Figure 2: *Example of an occupancy map with its corresponding scene geometry and 3D (uniform) binary volume.*

3.1. Occupancy Map, Distance Field, and Ray Tracing

An occupancy map (OM) is essentially a 3D (uniform) binary volume that approximately represents scene geometries (Figure 2). Each grid cell contains a binary value indicating whether it intersects with the scene geometry or not. To store an OM compactly, every 32 cells along one dimension, normally the z-axis, of the grid are encoded into a 32-bit integer. As a result, 3D OMs can be compactly represented by 2D texture maps [ED06].

Generation of the above-mentioned OMs can be easily accomplished by rasterization using graphics hardware [ED06] (see Sec. 6 for more details). This means it is possible to generate the OMs from scratch for every new frame, which enables support for dynamic objects with deformation or animation.

For visibility computations between OM and rays, there is an intuitive but inefficient solution: performing a 3D DDA [AW*87] to traverse and check all cells one by one along the ray. Unsurprisingly, this performed poorly on most of the hardware, simply due to the typically large and inconsistent number of iterations to advance rays. To improve it, one idea is to uniformly sample and only check a small and fixed number of cells between the endpoints of the ray [RBA09]; however, this may miss occupied cells and therefore cause artifacts. Instead of using this 3D tracing scheme, Thiedemann et al. [THGM11] proposes a method that exploits the nature of cells being encoded as bits along the z-axis inside a texel. Instead of directly tracing the rays in 3D space to check all cells, they first project the ray onto the 2D texture map—the xy-plane; then, they perform a 2D DDA to traverse all texels along the ray:

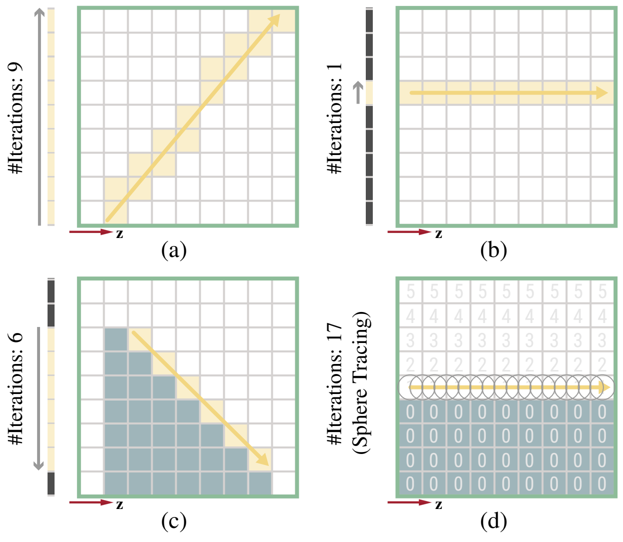


Figure 3: *Examples of tracing rays against occupancy maps and distance fields in 2D. In this case, 2D occupancy maps are represented by 1D texture maps. When traversing the 1D texture map, a group of cells along z-axis can be checked at once. The number of algorithm iterations depends on the projected area of the cells through which the ray should have traversed. (a) A ray travels from the lower left corner to the upper right corner, which leads to the maximum number of iterations. (b) In contrast, when a ray is traveling along z-axis, it only requires one single iteration. (c) A typical failure case when using mip-maps to hierarchically traverse the occupancy map: it always fails to advance to the next mip-level. (d) Failure case of sphere tracing the distance fields when rays are at grazing angles. Since each step size the sphere tracing method takes depends on the distance to the closest surface, the algorithm has to take a large number of small steps to advance the ray.*

for each texel, a group of cells along z-axis can be checked at once with one texture fetch and one bit operation. See Figure 3 for examples in 2D. Moreover, similar to the hierarchical-Z screen space ray tracing technique [Ulu18], to further reduce the number of necessary iterations, mip-maps of OM can be generated for faster skipping of empty space during hierarchical traversal. However, such hierarchical traversal scheme always has difficulties in progressing to a higher mip level when tracing, especially in the case shown in Figure 3(c).

It is also possible to accelerate tracing by incorporating a proximity map into OM and utilizing sphere tracing to march towards the nearest occupied cell. This is commonly referred to as *distance fields (DFs)*. However, sphere tracing the distance fields are still too slow for long and incoherent rays (computed with different numbers of iterations per thread) [TDD*22], especially for grazing angle rays shown in Figure 3(d). When rays are close to grazing angles, the sphere tracing method has to take a large number of small steps to advance the ray and therefore result in a large number of iterations. Besides, distance fields are costly to generate. In real-time applications, distance fields are typically generated either from OMs using *jump flooding algorithm (JFA)* [RT06; LM22] on the fly, or by closest point queries [TDD*22] in a pre-computation manner. However, considering the cost of these methods, and the time required to filling a 3D array of floating numbers, distance fields are more expensive to support per-frame update for dynamic objects than OMs.

Apart from requiring multiple iterations to advance rays, all these tracing methods also suffer from thread divergence, which comes from the inconsistent number of iterations per thread, as illustrated in Figure 3(a) and (b). This means if rays are sampled randomly and uncorrelated, which is a common case when simulating global illumination, the method will always suffer from thread divergence. Mip-maps acceleration can scale down the possible number of iterations, but it is ineffective to avoid the thread divergence [THGM11].

3.2. Analysis and Motivation

We intend to further speed up ray tracing against OMs and better utilize the trait of fast generation for dynamic scene objects. As shown in Figure 3, our key observation is that when a ray is tracing along the z-axis, i.e., when the projection of that ray on the texture plane is entirely inside a single texel, the tracing algorithm requires only a single iteration. If all of our rays are tracing along the z-axis, in other words, when the OMs are *aligned with the rays*, we will not only minimize the thread divergence but also achieve $O(1)$ tracing performance. Considering that the ray directions in real-time applications are fully random, the most intuitive way to achieve this goal is, for each possible ray direction, to precompute a *ray-aligned occupancy map* whose z-axis of the normalized device coordinate (NDC) space aligned with the ray direction.

This immediately introduces two problems. Firstly, we need an infinite number of ray-aligned OMs, which is intractable in practice. Secondly, generating each ray-aligned occupancy map invokes one pass of rasterization for the entire scene, resulting in significant generation overhead.

For the first problem, we have a key insight that it is unnecessary to generate ray-aligned occupancy maps for all possible ray directions. We can instead generate only for a subset of *candidate directions*, which are distributed evenly on a hemisphere. After that, for any ray to be traced, we select the most appropriate ray-aligned occupancy map based on the similarity between its direction and the candidate directions; then, instead of tracing the (still possibly skewed) ray, we always trace the ray straight through the selected ray-aligned OM, along its z-axis. Essentially, we enforce our rays to be sampled only from a discrete set of directions to reduce the necessary number of ray-aligned occupancy maps. Intuitively, this may cause inaccuracies. However, we find that for every frame, the candidate directions can be sampled sparsely and differently. Together with a spatiotemporal denoising approach (Section 4.3), we are able to distribute the computation over time, and achieve clean results.

For the second problem, we observe that a well-generated occupancy map contains complete 3D geometric information from every viewing angle (unlike a depth map). So, instead of re-generating this information based on a new view direction aligned with a candidate ray through rasterization, we only need to rasterize the scene once to a base occupancy map (BOM), then simply "rotate" the information for the new view direction within a *compute shader*, resulting in fast generation of our *ray-aligned occupancy map array (ROMA)*.

4. Method

In this section, we present our tracing method for fast visibility and intersection computations between rays and occupancy map (OM) (Figure 4).

Our high-level idea is to first perform a fast generation from scene geometry (Section 4.1): we rasterize a high-quality *base occupancy map* (BOM), select a set of candidate directions, rotate the BOM to get ray-aligned OMs, and then save them to our *ray-aligned OM array* (ROMA). For each ray to be traced, we find the "closest" candidate direction and trace the corresponding ray-aligned OM to achieve $O(1)$ performance (Section 4.2). Since generating a ROMA is fast, in each frame we re-sample *a different set of* candidate directions and generate a new ROMA to improve accuracy and alleviate aliasing artifacts (Section 4.3).

4.1. Generating Ray-aligned Occupancy Map Array

Algorithm 1: Algorithm to generate a ray-aligned occupancy map (OM) given the base OM. **Inputs:** $\mathbf{M}'_{viewproj}$: view-projection matrix of the base OM. $\mathbf{M}^{-1}_{viewproj}$: inverse of view-projection matrix of the ray-aligned OM. uv: the uv coordinates of the currently processed texel. **Output:** $OM_{aligned}(uv)$: Texel value of the ray-aligned OM.

```
Function Generate($\mathbf{M}'_{viewproj}$, $\mathbf{M}^{-1}_{viewproj}$, $uv$) : $OM_{aligned}(uv)$ is
    $\mathbf{x}^{start} \leftarrow (uv, 0.5/32)$, $\mathbf{x}^{end} \leftarrow (uv, 1.0 - 0.5/32)$;
    $\mathbf{x}^{start}_{world} \leftarrow$ ToWorldSpace($\mathbf{x}^{start}$, $\mathbf{M}^{-1}_{viewproj}$);
    $\mathbf{x}^{end}_{world} \leftarrow$ ToWorldSpace($\mathbf{x}^{end}$, $\mathbf{M}^{-1}_{viewproj}$);
    $\mathbf{x}^{start}_{base} \leftarrow$ FromWorldSpace($\mathbf{x}^{start}_{world}$, $\mathbf{M}'_{viewproj}$);
    $\mathbf{x}^{end}_{base} \leftarrow$ FromWorldSpace($\mathbf{x}^{end}_{world}$, $\mathbf{M}'_{viewproj}$);
    $\mathbf{v}_{step} \leftarrow (\mathbf{x}^{end}_{base} - \mathbf{x}^{start}_{base})/31$;
    $U_{aligned} \leftarrow 0$;
    for $i \leftarrow 0$ to 31 do
        $uv_{base} \leftarrow \mathbf{x}^{start}_{base}$.xy;
        $t \leftarrow$ Floor($\mathbf{x}^{start}_{base} \times 32$).z;
        $U_{base} \leftarrow OM_{base}(uv_{base})$;
        $U \leftarrow (U_{base} << t) >> 31$;
        if $U > 0$ then
            $U' \leftarrow 1 << (31 - i)$;
            $U_{aligned} \leftarrow U_{result} | U'$;
        $\mathbf{x}^{start}_{base} \leftarrow \mathbf{x}^{start}_{base} + \mathbf{v}_{step}$
    $OM_{aligned}(uv) \leftarrow U_{aligned}$;
```

A straightforward way to generate a ROMA is computing all the maps by rasterization, but it is inefficient due to the limited time budget. Instead, we generate a BOM by rasterization and use it to "rotate" to other ray-aligned OMs (Figure 4). As displayed in Algorithm 1, for each cell of a new ray-aligned OM associated with a specific candidate direction and camera pose, we first transform the cell to the world space, further transform it to the NDC space of the BOM, and fetch the value. This only involves simple coordinate transformations and queries in the BOM, which can be efficiently implemented only using compute shaders and is decoupled from scene complexity. Note that in Algorithm(s) 1 (and 2), we demonstrate our method using a single 32-bit integer for simplicity. It is straightforward to extend it to support more bits; please refer to our code for details.

To improve the accuracy of our approximate scene representation, we choose candidate directions to evenly distributed on the hemisphere, because these directions are used for generating ray-aligned OMs and will later be selected for replacing each newly sampled ray during tracing. We use 2D stratified sampling based on the concentric mapping [SC97]. This technique maps concentric squares to concentric circles on the hemisphere and preserves fractional areas.

Discussion: difference to imperfect shadow map. The imperfect shadow map [RGK*08] also generate a group of approximate scene representation – depth maps from multiple views and save onto a single atlas. The key difference is that the depth map only provides the nearest boundary of the scene geometries, thus they can not generate another depth map from the existing one for a different view. So they have to rasterize the same scene primitives for many times, which is in practice replaced by distributing a point-represented scene over different views, trading performance with "imperfection".

4.2. Tracing against Ray-aligned Occupancy Map Array

With the ROMA generated in Section 4.1, achieving fast and low-divergence tracing is easy.

Given a newly sampled ray for visibility or intersection computations, the first step is to find the "closest" candidate direction. This process can be done by comparing dot products of the ray direction and candidate directions, which is intuitive but inefficient, especially when considering the limited budget. Our key observation here is that when using stratified sampling, for any sampled ray direction, we can immediately find its corresponding stratum; and the candidate direction in that stratum is already close to the sampled ray direction. Therefore, as illustrated in Figure 4, instead of finding the "closest" candidate direction for sampled ray direction, we find a close enough one by using stratified sampling and employing the concentric mapping [SC97].

Then, as described in Section 3.2, to minimize the number of iterations and the thread divergence, instead of directly tracing the newly sampled ray, we "snap" the ray to the selected candidate direction before tracing it. Then, we trace against the ray-aligned OM which corresponds to that candidate direction in ROMA (See Figure 4).

Having this ray and its aligned OM, visibility or intersection

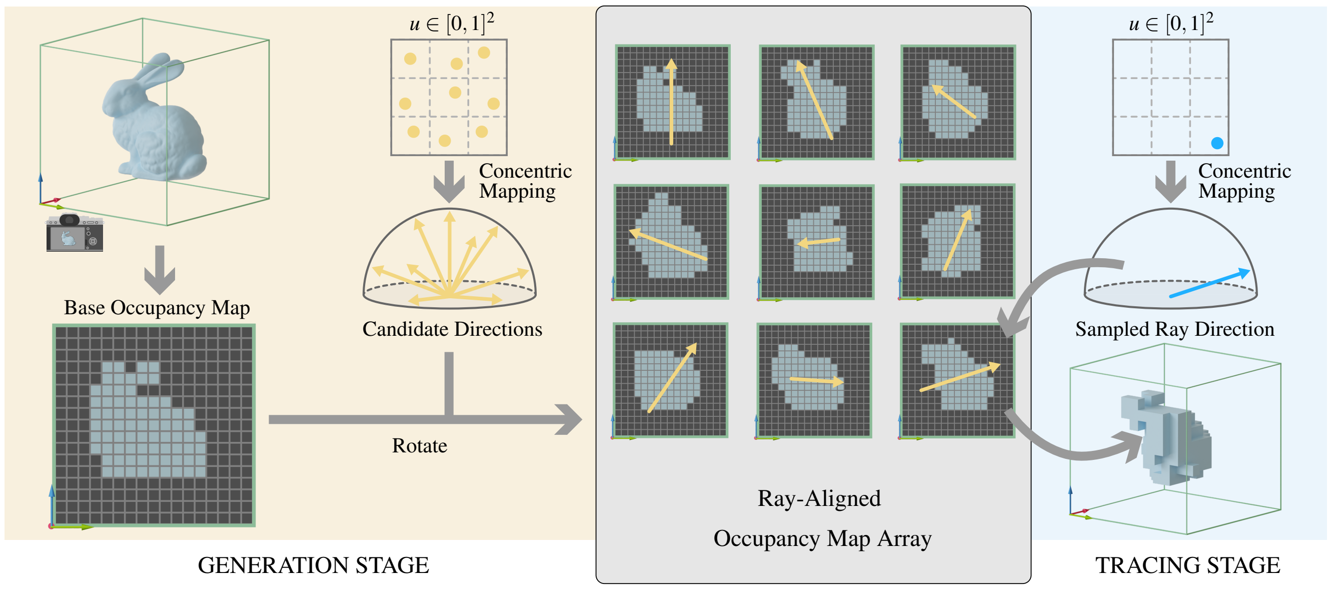


Figure 4: *Algorithm overview. Given the geometries, we first rasterize them into a regular Base Occupancy Map (BOM). Then we rotate it towards stochastically selected candidate directions to build our Ray-aligned Occupancy Map Array (ROMA). With ROMA, given any ray to be traced against the geometries, we find its closest candidate direction and the corresponding OM from ROMA, to perform fast O(1) ray tracing using bit operations.*

computations can be simply done using bit operations (Algorithm 2). For visibility, we only need to check if there is an intersection along the ray (an *any hit* query). Given the ray's origin and direction (either along z or $-z$), an *any hit* query can be accomplished in two steps. First, we left shift (or right shift, depends on the direction) the bit values saved on the OM – the resulting bit values are belonging to cells where the ray would have passed through; then check if the resulting bit values is beyond zero – in that case, there is an intersection. For finding the exact intersecting position (a *closest hit* query), there is only one extra step: performing a *low-bit* operation (or *formost-bit* operation, again depending on the ray's direction) to locate the right-most (or left-most) occupied cell where the ray would have intersected first.

Discussion: "snap" or not. Although "snapping" the sampled rays to the candidate directions is the best way to perform ray tracing with ROMA, note specifically that ROMA also benefits directly from tracing the sampled rays (un-"snapped" rays). This is because, for any sampled ray, the corresponding ray-aligned OM chosen from ROMA is already a good OM to be directly traced against: the sampled ray direction is close to the candidate direction, so it only requires a few iterations to cross the entire OM. This means, considering that the "snapped" rays do not introduce artifacts only the original sampled rays are distributed on the entire hemisphere, for applications that require the sampled rays pointing to specific directions (for example, soft shadows in Section 5), we can choose not to "snap" the rays to trade some performance for better visual quality. In our experiments, we found using 8 iterations in ROMA tracing is sufficient for simulating visually pleasing soft shadows.

4.3. Spatiotemporal

With our tracing method performed on ROMA, as reported in Table 2, we can already achieve fast generation from scene geometry and fast tracing. However, we can further boost its performance while alleviating aliasing by employing a spatiotemporal scheme.

Since the OM is a discrete representation of scene geometries, the resolution of each OM in our ROMA, we call it the *positional resolution* of our method, is critical for representing the scene well and avoiding aliasing. Also, the number of candidate rays or the number of OMs in a ROMA, the *angular resolution* of our method, determines how many directions we can really trace; insufficient number of directions will lead to artifacts on final rendering (See Figure 9). However, the extra time and space costs incurred by utilizing higher positional and angular resolutions are always unacceptable.

Inspired by the idea of Temporal Anti-Aliasing (TAA) [TKD*14], we propose to use a spatiotemporal scheme to distribute the computational workload spatially and temporally. More specifically, for each new frame: when generating the OMs, we jitter objects' geometric positions for each OM; before rotating to ROMA, we resample candidate directions using stratified sampling. Coupled with a spatiotemporal denoiser [SPD18], we increase the effective positional and angular resolutions within the limited budget and therefore alleviate aliasing artifacts. Tunable positional and angular resolutions make our method fully scalable on quality and performance.

Algorithm 2: Algorithm to trace a ray-aligned OM from ROMA. **Inputs:** $OM_{aligned}$: the selected ray-aligned OM that is a 32-bit texture map. $\mathbf{M}_{viewproj}$: view-projection matrix of the ray-aligned OM. $\mathbf{d}_{world}$: the world-space ray direction. $\mathbf{o}_{world}$: the world-space ray origin. **Output:** hitInfo: the result of visibility or intersection computation.

```
Function AnyHit($U$: int) : hitInfo is
    if $U > 0$ then
        return Occluded;
    else
        return Missed;
Function ClosestHit($U$: int, $z$: float) : hitInfo is
    if $U > 0$ then
        if $z > 0$ then find foremost bit
            return 32 - Floor($\log_2 U_{result}$) - 0.5
        else find lowest bit
            return 32 - $log_2\left(U_{result} \& (-U_{result})\right)$
    else
        return Missed;
Function Trace($OM_{aligned}$, $\mathbf{M}_{viewproj}$, $\mathbf{d}_{world}$, $\mathbf{o}_{world}$, $uv$) :
  hitInfo is
    $\mathbf{o}_{aligned} \leftarrow FromWorldSpace(\mathbf{o}_{world}, \mathbf{M}_{viewproj})$;
    $\mathbf{d}_{aligned} \leftarrow FromWorldSpace(\mathbf{d}_{world}, \mathbf{M}_{viewproj})$;
    $uv \leftarrow \mathbf{o}_{aligned}.xy$;
    $t_{start} \leftarrow Floor\ (\mathbf{o}_{aligned} \times 32)$;
    if $\mathbf{d}_{aligned}.z \geq 0$ then
        $t_{end} \leftarrow 31$;
    else
        $t_{end} \leftarrow 0$;
    $U_{result} \leftarrow 0$;
    $U_{aligned} \leftarrow OM_{aligned}(uv)$;
    $t_{min} \leftarrow Min\ (t_{start}, t_{end})$, $t_{max} \leftarrow Max\ (t_{start}, t_{end})$;
    $U_{result} \leftarrow (U_{aligned} << t_{min}) >> (31 - t_{max} + t_{min})$;
    return AnyHit($U_{result}$) or
      ClosestHit($U_{result} << (31 - t_{max})$, $\mathbf{d}_{aligned}.z$);
```

5. Applications

Our method can accelerate real-time rendering applications by efficiently answering the ray intersection queries, i.e., *any hit* queries and *closest hit* queries. In this section, we first demonstrate two representative applications using sampled rays distributed on the entire hemisphere: *ambient occlusion* with fast *any hit* queries, and *one-bounce diffuse indirect illumination* with fast *closest hit* queries. Then, we show another representative application using sampled rays pointing to specific directions: soft shadows from area lights with *any hit* queries.

Ambient occlusion. One of the most straightforward application employing visibility test is ambient occlusion, which captures the occlusion at $\mathbf{x}$ with normal $\mathbf{n}$, calculated from the visibility $V(\mathbf{x}, \omega)$ of incident rays from all directions ω on hemisphere Ω:

$$AO(\mathbf{x}) = \frac{1}{\pi} \int_{\Omega} V(\mathbf{x}, \omega)(\mathbf{n} \cdot \omega)\, \mathrm{d}\omega. \tag{1}$$

We compute the visibility function $V(\mathbf{x}, \omega)$ using ROMA with *any hit* queries with an infinite t_{max} in constant time (Algorithm 2).

One-bounce diffuse indirect illumination. Starting from the primary shading point $\mathbf{x}$ on a diffuse surface with albedo ρ, we need to trace secondary rays to compute the one-bounce indirect illumination $L_o(\mathbf{x})$:

$$L_o(\mathbf{x}) = \int_{\Omega} \frac{\rho}{\pi} Li(\mathbf{x}, \omega)(\mathbf{n} \cdot \omega)\, \mathrm{d}\omega. \tag{2}$$

To compute the incident radiance $L_i(\mathbf{x}, \omega)$, the first thing is finding the closest intersection point along the secondary ray direction ω. ROMA with *closest hit* queries can be employed here to find the closest intersection (Algorithm 2), i.e., the secondary shading point. Then, there are many ways to inject lighting and calculate illumination at the secondary shading point, including radiance cache [MRNK21], mesh cards [TDD*22], and reflective shadow map (RSM) [DS05]. In this paper, we use the RSM technique. RSMs are generated for spot and point lights with *radiance*, *position*, and *normal* in each pixel. For each secondary shading point we found, we transform and project it into the RSM to fetch the scattered radiance $L_i(\mathbf{x}, \omega)$ to primary shading point from its position.

Rays from the above-mentioned two applications are distributed over the entire hemisphere; each single visibility query does not have to be very precise as long as the final integral approximates well. Therefore, we can safely use "snapped" rays without introducing artifacts. Next, we show one application that requires precise visibility.

Soft shadows. Having chosen a particular area light to calculate direct lighting from, we need to trace shadow rays from the shading point $\mathbf{x}$ to a point $\mathbf{x}'$ sampled on the area light to simulate soft shadows:

$$L_0(\mathbf{x}) = \int_{\mathcal{M}} \frac{\rho}{\pi} L_e\left(\mathbf{x}' \to \mathbf{x}\right) G\left(\mathbf{x}' \leftrightarrow \mathbf{x}\right) \mathrm{d}A(\mathbf{x}'). \tag{3}$$

We compute the visibility function $V\left(\mathbf{x}' \leftrightarrow \mathbf{x}\right)$ in the geometry term $G\left(\mathbf{x}' \leftrightarrow \mathbf{x}\right)$ using ROMA with *any hit* queries with a finite t_{max}. Note that since simulating soft shadows requires precise visibility, i.e., rays have to point to specific directions, we choose not to "snap" the ray for accuracy.

Discussion: range of applications that can be handled. As shown in Table 1, ROMA supports the above-mentioned three applications well. However, for applications using rays pointing to a single specific direction like hard shadows from point lights and pure specular reflection, although we can choose not to "snap" the ray to use the original sampled direction, there will still be light leaking caused by the limited positional resolution. Such applications are challenging to all voxel-based methods including ROMA since it requires an extremely high positional resolution. Applying the spatiotemporal scheme here can only alleviate but fails to eliminate the artifacts.

6. Implementation

We implement our algorithm with the Slang [HFF18] shading language inside the NVIDIA Falcor [KCK*22] renderer. We will release our code upon publication.

Sampled rays' direction	Applications	Support	"Snap"
Entire hemisphere	Ambient occlusion Diffuse reflection	✓	✓
Specific directions	Soft shadows Glossy reflection	✓	✗
One specific direction	Hard shadows Pure specular reflection	✗	N/A

Table 1: *Range of applications that can be handled by our method. For applications using rays distributed on the entire hemisphere, we recommend to use "snapped" rays to maximize performance. For applications using rays pointing to specific directions, we recommend not to "snap" the rays to avoid artifacts. Same as other voxel-based methods, we do not support applications that have rays pointing to one specific direction like hard shadow and pure specular reflection, because the limited positional (grid) resolution will lead to light leaking artifacts.*

ROMA generation. By default, we use a positional resolution of 128^2 (effectively representing a 3D binary grid with a resolution of 128^3 because of bit compression) for the base occupancy map (BOM) and the ray aligned OMs in ROMA. The default angular resolution for ROMA is 4^2, i.e., we sample 4^2 candidate directions. When simulating effects that require precise visibility like soft shadows, we use an angular resolution of 8^2 to increase tracing accuracy.

The quality of the BOM is vital for generating our ROMA with complete geometric information. One major drawback of the BOM generation strategy described in Section 3 is that surfaces with slopes close to the grid's z-axis will not be rasterized into fragments and thus will not occupy any cells. To address this issue, for all our results, we perform three times of rasterization with cameras placed towards three axis directions: x, y, and z, and then compute the union of the three resulting BOMs, as suggested by Forest et al. [FBP09]. It improves accuracy while adding some overhead.

To avoid potential race conditions caused by multiple fragments being sent to the same pixel, we use *rasterizer order views (ROVs)* when generating the BOM using rasterization.

Tracing ROMA. To avoid shadow acne caused by self-intersections due to the insufficient positional resolution, for the starting points of bounced rays, we perturb them along the surface normal directions by $1.5\times$ ROMA's grid cell size in the world space.

Real-time GI approximation. We use reflective shadow maps (RSMs) to approximate the real-time single-bounce diffuse indirect illumination. For each spot light and point light, We generate a 512×512 RSM to inject lighting and compute indirect illumination. We apply the same techniques as those for generating the OMs, to generate RSMs: we jitter the light position to avoid aliasing; when testing the texel in RSM to accumulate lighting, we perturb the points to avoid shadow acne of RSMs.

Post-processing. We use the ReLAX denoiser in the NVIDIA Real-Time Denoisers (NRD) library [NVI] for our results at 1 sample per pixel every frame, followed by a Temporal Anti-Aliasing (TAA) [TKD*14] filter to compress residual noise and reduce spatiotemporal aliasing.

Distance fields. We implement real-time distance fields as one of the baseline methods. For the DF generation, we apply the 3D Jump Flooding algorithm [RT06] on our BOM and create a 3-level mipmap of the resulting global distance field's grid with a resolution of 128^3. Each grid cell records the distance to the center of the nearest occupied cell and the distance is saved in float16 for best performance (equal storage to ROMA with an angular resolution of 4^2). Hardware trilinear interpolation is used between the cells. For tracing, we use the sphere tracing accelerated by mipmaps [Aal18] with up to 64 iterations. To achieve the best visual quality and prevent artifacts such as light leaks and self-intersection, for each scene we manually tweaked the hyperparameters including the minimum ray propagation distance, the threshold value of ray intersection, and the offset distance along the surface normals.

7. Results

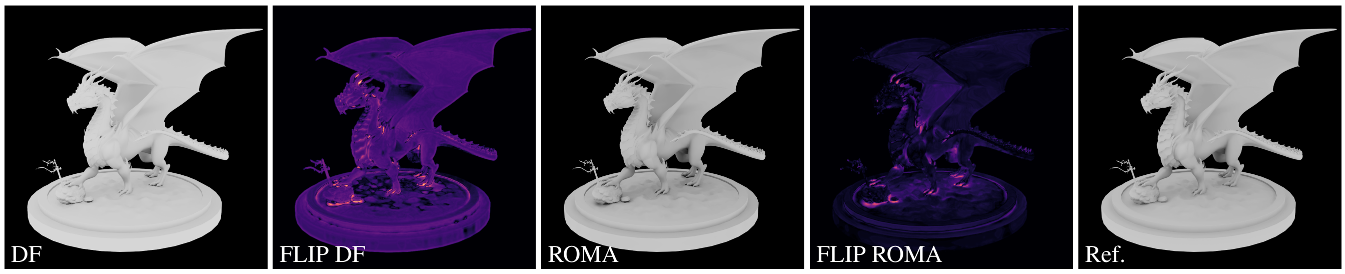
In this section, we present our results of ambient occlusion and single-bounce diffuse indirect illumination. We also compare our rendering results with distance fields (DF) and hardware ray tracing (HWRT). All experiments and timings are conducted on a desktop with a 3.70 GHz Intel i9-10900K and an NVIDIA GeForce RTX 3080 Ti. We use the FLIP [ANA*20] image metric as the visual difference evaluator. All results are rendered using only one sample per pixel and are denoised further by ReLAX denoiser and TAA. Please refer to the supplementary video for the results on dynamic scenes.

7.1. Main Results

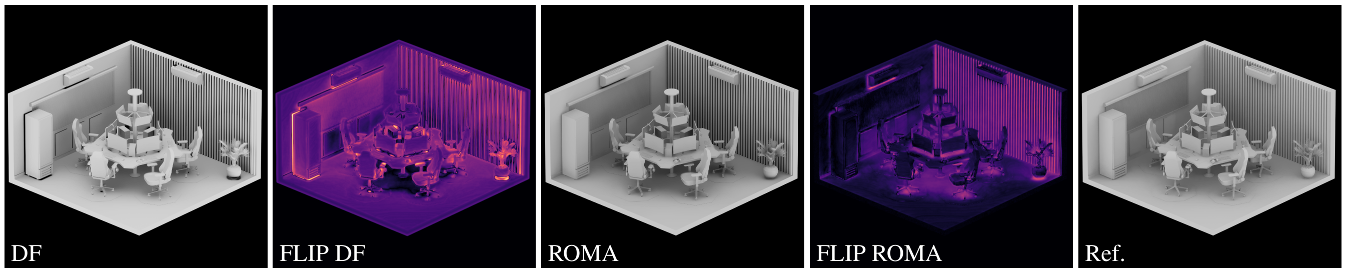
Ambient occlusion (AO). As shown in Figure 1(a) and Figure 5, we compare our method (ROMA) with DF on computing AO for five scenes. We use the AO simulated by HWRT as the reference. The comparison indicates that our ROMA achieves better visual quality than DFs on all test scenes with around $2.5\times$–$10\times$ speedup. The contact shadows appear over-darkening caused by self intersections in both ROMA's and DF's results. This is fundamentally due to the limited positional resolution and the inappropriate ray origin perturbation used by ROMA and DFs.

Single-bounce diffuse indirect illumination. As shown in Figure 1(b) and Figure 6, we use three scenes with spot lights for comparing the single-bounce diffuse indirect illumination among our method (ROMA), DF, and HWRT. Both ROMA and DF use reflective shadow maps (RSMs) to inject lighting into the scene, while HWRT directly simulates global illumination using path tracing with next event estimation (NEE). We show both direct illumination and single-bounce diffuse indirect illumination in the results. The comparison indicates that ROMA achieves comparable and even better visual quality. We also observe the over-darkening issue that appears in AO here, especially near the bottom of objects.

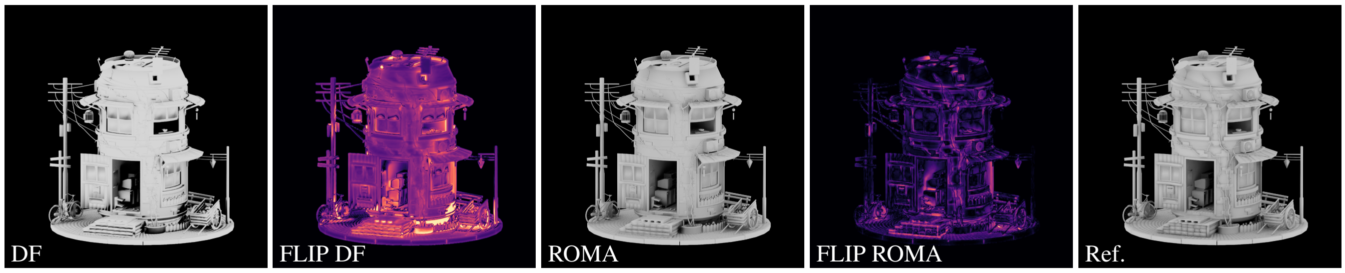
Soft shadows. As shown in Figure 7, we use three dynamic scenes with area lights on the ceilings for comparing soft shadows among our method (ROMA), DF, and HWRT. We show both



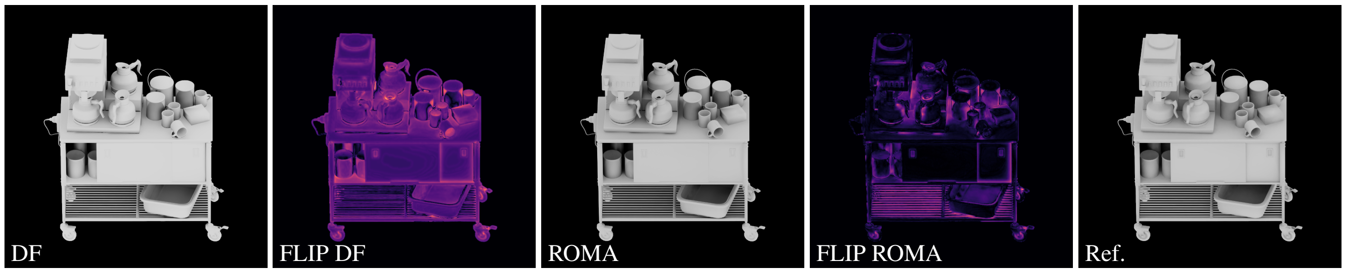
(a) DRAGON scene.



(b) GAMING STATION scene.



(c) CAN HOUSE scene.

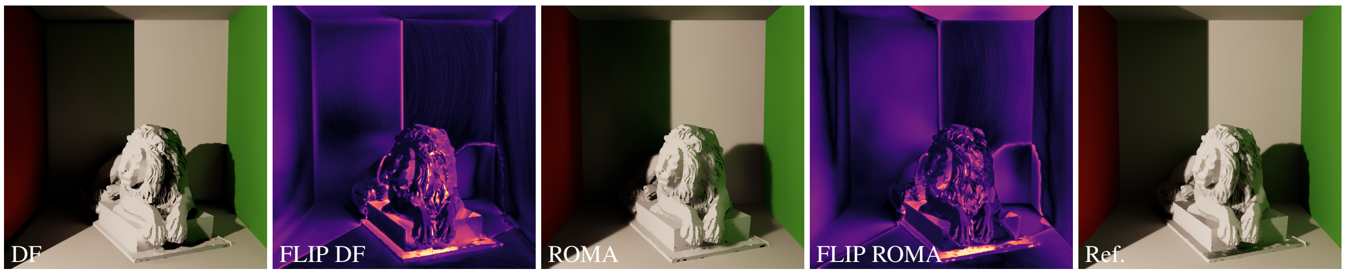


(d) COFFEE CART scene.

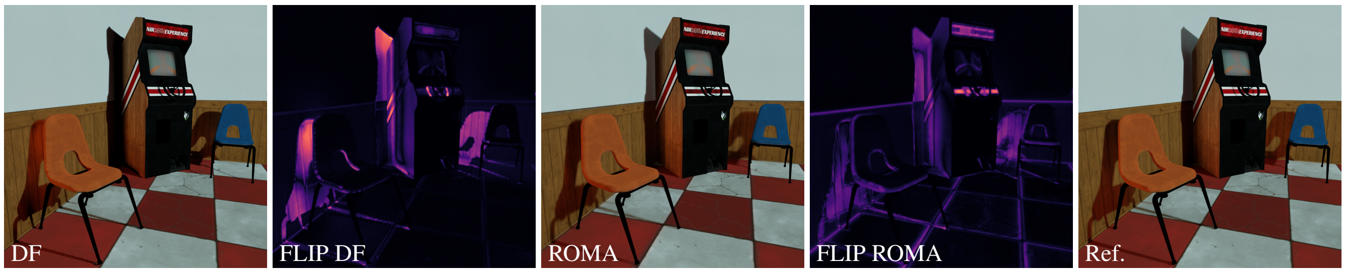
Figure 5: *Comparison between our method (ROMA), distance field (DF), and hardware ray tracing (Ref.) on simulating ambient occlusion. We use the FLIP image as the visual difference evaluator. ROMA achieves better visual quality on all four scenes, with around* 2.5-10× *speed-up to DF.*

direct illumination with soft shadows and single-bounce diffuse indirect illumination (same as Figure 6) in the results. MORPHING SPOT scene and MORPHING SPIKE scene have deformations, while BRAINSTEM scene has skinned animations. The comparison indicates that ROMA achieves comparable visual quality. Please see the supplemental video for animated comparisons.

Performance. In Table 2, we report and compare the average computation cost in milliseconds among ROMA, DF, and HWRT. The timings are measured on the BUNNY scene (Figure 1) rendered in 1080P for AO. At the generation stage, our ROMA is built within 1 ms, even with a positional resolution of 128^2. In contrast, DF requires 3.3 ms to build at the same resolution and with equal storage, which is 11.0× slower than ours. This is mainly due to the time complexity of the 3D Jump Flooding algorithm [RT06]. The speedup shrinks as the resolution decreases, but our generation speed is consistently 1.7×–11.0× faster than DF. At the tracing stage, our method using ROMA can answer ray intersection queries in 0.16 ms regardless ROMA's resolutions, thanks to our constant-time tracing method. Tracing is around 3.4×–8.1× faster than DF,



(a) LION scene.



(b) ARCADE scene.

Figure 6: *Comparison between our method (ROMA), distance field (DF) and hardware ray tracing (Ref.) on simulating one-bounce diffuse indirect illumination. We use the FLIP image as the visual difference evaluator. Each scene curtains a spot light. Both ROMA and DF use reflective shadow maps (RSMs) to inject lighting, while HWRT directly using path tracing with next event estimation (NEE). We show both direct illumination and single-bounce diffuse indirect illumination in the results.*

Pos. Res.	32^2		64^2		128^2	
Ang. Res.	4^2	8^2	4^2	8^2	4^2	8^2
GENERATION						
ROMA	0.14 ms	0.18 ms	0.20 ms	0.31 ms	0.30 ms	0.69 ms
Distance field	0.31 ms		0.55 ms		3.31 ms	
(Speed-up)	(2.2×)	(1.7×)	(2.7×)	(1.7×)	(11.0×)	(4.8×)
(Storage)	(1×)	(4×)	(1×)	(4×)	(1×)	(4×)
HWRT	0.08 ms					
TRACING (1 sample per pixel)						
ROMA	0.16 ms	0.16 ms	0.16 ms	0.16 ms	0.16 ms	0.16 ms
Distance field	0.55 ms		0.84 ms		1.30 ms	
(Speed-up)	(3.4×)		(5.25×)		(8.1×)	
HWRT	0.31 ms					
(Speed-up)	(1.9×)					
TOTAL						
ROMA	0.30 ms	0.34 ms	0.36 ms	0.47 ms	0.46 ms	0.85 ms
Distance field	0.86 ms		1.39 ms		4.61 ms	
(Speed-up)	(2.9×)	(2.5×)	(3.9×)	(3.0×)	(10.0×)	(5.4×)

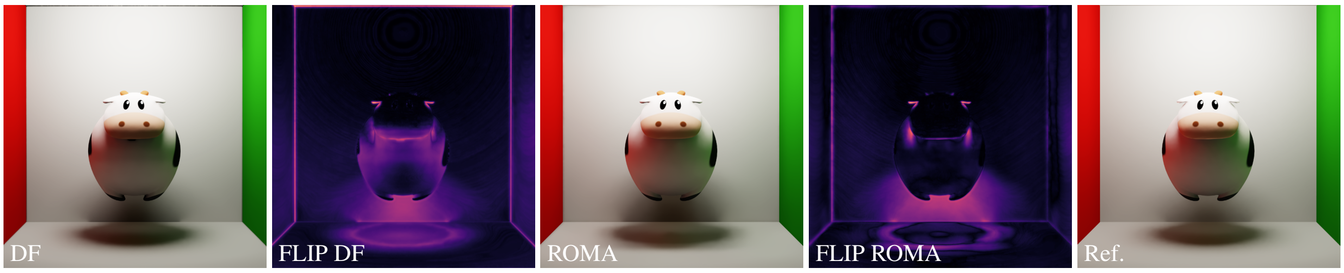
Table 2: *Runtime breakdown between our method (ROMA), distance field (DF), and hardware ray tracing (HWRT). The times are measured on the* BUNNY *scene (Figure 1) rendered in 1080P for ambient occlusion. Compared with DF, our method is consistently faster in both generation and tracing. Compared with HWRT, generating ROMA is slower but tracing is* 1.9× *faster.*

and in total we are 2.5×–10× faster than DF combined with the generation time. Although ROMA generation is slower than the hardware BVH construction/update for HWRT, the tracing speed with ROMA is about 1.9× faster than HWRT.

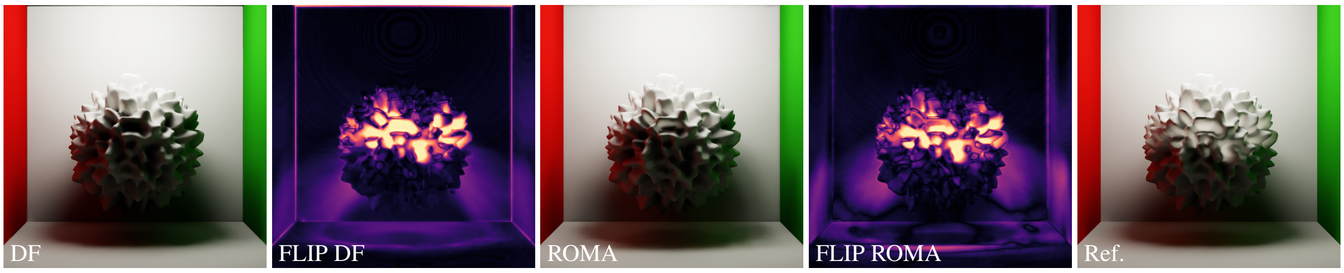
We also measure the average timings on the MORPHING SPOT scene, MORPHING SPIKE scene, and BRAINSTEM scene simulating soft shadows. At the generation stage, the situation is the same: our ROMA is built within 0.8 ms, while DF requires 2.9 ms to build at the same resolution, which is 3.6× slower than ours. At the tracing stage, tracing un-"snapped" rays using ROMA with a maximum of 8 iterations only takes 0.23ms, which is 1.7× faster than tracing DF with a maximum of 16 iterations(0.40ms) and 1.3× faster than HWRT (0.30ms).

7.2. Discussions

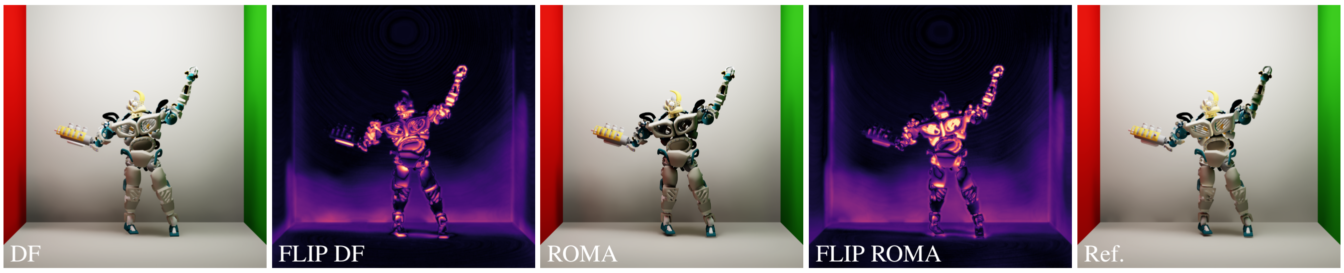
Artifacts by "snapping" the ray. As described in Section 4, given the newly sampled ray for ray intersection query, instead of directly tracing the new ray, we "snap" the ray to the selected candidate direction before tracing it. This idea boosts performance while do not introduce extra visual artifacts for applications like AO and diffuse indirect illumination. As shown in Figure 8 (a) on the MARBLE scene for one-bounce diffuse indirect illumination, there is no visible difference between "snapping" and not "snapping". Besides, when "snapping" the ray, the tracing stage takes 0.24 ms, and when not doing it, the tracing stage takes 2.05 ms with a maximum of 64 iterations (for getting the comparable color bleeding), which means we can gain a 8.5× speed-up when choose to "snap" the ray. Also, as discussed in Section 4.2, for applications like soft shadows, as shown in Figure 8 (b) on MORPHING SPOT scene, "snapping" the ray would cause artifacts since the "snapped" shadow rays may point to wrong places. So, we choose to directly trace the sampled rays for better visual quality and use a higher angular resolution of 8^2 for maintaining the good performance. We can achieve 0.23ms



(a) MORPHING SPOT scene.



(b) MORPHING SPIKE scene.



(c) BRAINSTEM scene.

Figure 7: *Comparison between our method (ROMA), distance field (DF), and hardware ray tracing (Ref.) on simulating soft shadows. We use the FLIP image as the visual difference evaluator. We show both direct illumination with soft shadows and single-bounce diffuse indirect illumination in the results.* MORPHING SPOT *scene and* MORPHING SPIKE *scene have deformations, while* BRAINSTEM *scene has skinned animations. Please make sure to check out our accompanying video for much clearer comparison in dynamic. ROMA achieves comparable quality, with around* $3.6\times$ *speed-up in generation and* $1.7\times$ *speed-up in tracing than DF.*

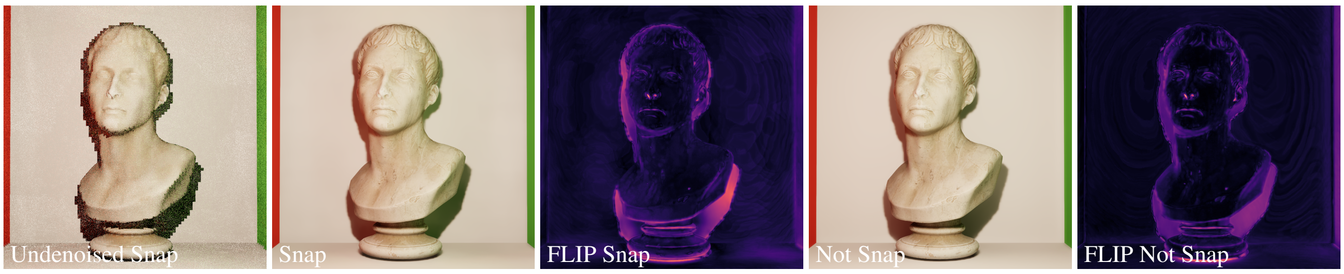
in tracing with a maximum of 8 iterations (for simulating visually pleasing soft shadows), in comparison to $0.15ms$ when "snapping".

Choice of resolutions. We compare the AO results in Figure 9 when choosing different positional and angular resolutions. A positional resolution of 128^2 and an angular resolution of 4^2 can already achieve visually pleasing results and fast performance (according to Table 2), while avoiding obvious light leaking; thus, we make it the default choice. When simulating effects that require precise visibility like soft shadows, we can increase the angular resolution to 8^2 to boost tracing accuracy.

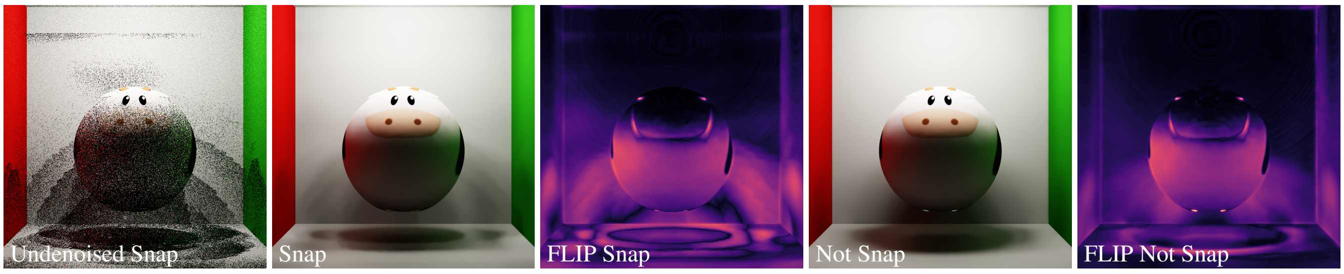
ROMA as a HWRT alternative. The purpose of ROMA is to provide a different way to perform ray tracing. Though much faster in building and tracing, ROMA is not designed as an alternative to DFs, for that DFs offer other convenient properties other than ray tracing, e.g., the simplicity to compute surface normal, differentiability [VSJ22] and their unique way to compute soft shadows [Wri15].

The same conclusion also applies to GI. Since ROMA only offers a fast ray tracing solution for secondary rays in GI, ROMA itself is not a GI solution. Therefore, any methods that provide caches to the outgoing radiance or incident illumination, e.g., Voxel Global Illumintaion [CNS*11] and Neural Radiance Caching [MRNK21], is orthogonal to what ROMA does. And their compatibility with ROMA depends on how fast these methods perform to build and update the cache, and whether they also support dynamic objects or not.

Fast hardware ray tracing. Note that in Table. 2, the BVH building process using HWRT is faster than our ROMA. We would like to specifically note that, although building the BVH is not hardware-accelerated, it is specially optimized and handled by drivers. For instance, BVH allows partial updates (BVH refitting) instead of rebuilding from scratch when the scene changed. For tracing the BVH, HWRT is accelerated with *specific-purpose* hardware (e.g. RT cores from NVIDIA RTX GPUs). Hence, it is already extraordinary that ROMA traces rays faster than HWRT. We

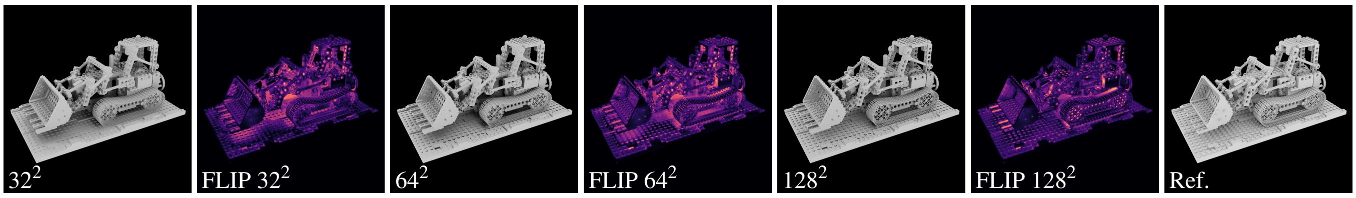


(a) MARBLE *scene. We show both direct illumination from RSM and one-bounce diffuse indirect illumination traced using ROMA.*

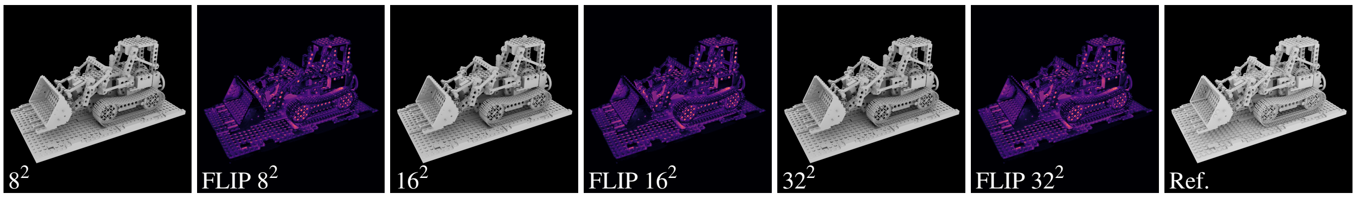


(b) MORPHING SPOT *scene. We show both direct illumination with soft shadow and one-bounce diffuse indirect illumination traced using ROMA.*

Figure 8: *Comparison between "snapping" the ray and not "snapping" the ray on* MARBLE *scene (aiming at showing one-bounce indirect illumination traced using ROMA) and* MORPHING SPOT *scene (aiming at showing soft shadow traced using ROMA). We use the FLIP image as the visual difference evaluator to the ground truth. For applications like diffuse indirect illumination, "snapping" the rays boost performance* (0.24*ms, in comparison to* 2.05*ms when not "snapping") while do not introduce extra visual artifacts. Also, for applications like soft shadows, "snapping" the rays would cause artifacts; so we choose to trace un-"snapped" rays for better visual quality while using a higher angular resolution for maintaining good performance* (0.23*ms, in comparison to* 0.15*ms when "snapping").*



(a) *Different positional resolution (with a same angular resolution of* 4^2*).*



(b) *Different angular resolution (with a same positional resolution of* 128^2*).*

Figure 9: *Ambient occlusion on* LEGO *scene rendered with positional and angular resolutions. We use the FLIP image as the visual difference evaluator.*

believe that with equal hardware support, ROMA could be built and run even faster. A note to SDF practitioners: tracing SDF is not necessarily faster than HWRT (as we have also demonstrated). We quote the following text from the design document of Lumen [TDD*22] as one of the important reason to use DF: "Hardware Ray Tracing is great and it is the future, but we need options to scale down. In the PC market there are still plenty of video cards that don't support hardware ray tracing, and console Hardware Ray Tracing is not that fast".

Light leaking. ROMA can possibly has few cells missing in each view. This is due to the aliasing from the "rotation" step. But these missing cells are located in different places cross views and frames; with our spatiotemporal scheme, we do not observe any obvious light leaking caused by this. However, like all other voxel-based representations with limited positional resolutions, both ROMA and DF suffer from light leaking caused by thin objects that fit within a voxel; for example, the light leaking on the morphing cow's foot in Figure 7. Similar to shadow acne, this can be resolved by perturbation of a voxel-sized bias.

Temporal artifacts. As with other ray tracing techniques for dynamic scenes at 1 sample per pixel, if we do not carefully tune the denoiser, our method will suffer from ghosting, lagging, or noise. We have carefully tested by finding parameters of NRD to guarantee that it accumulates up to 10 frames, which is good enough to suppress most noise and lagging. Besides, considering that a high FPS will hide most of these temporal artifacts, we provide the results showing temporal quality at 60 FPS in the accompanying video to prove that ROMA does not need a high FPS to converge better. Please make sure to check out.

Scalability to larger scenes. Same as DFs, ROMA cannot be directly used on larger scenes due to the limited positional resolution; but we believe it can be extended to support larger scenes. Following the previous extending ideas on DFs, potential solutions are as follows: one simple solution is to use cascades [LM22]: partition the scene into multiple areas according to the distance from the camera and use different resolutions of ROMAs for these areas. Similar to Lumen of using two levels of DFs [TDD*22], another solution is to use two levels of ROMAs: precise mesh ROMAs for near-field tracing and a coarse global ROMA for far-field tracing. One solution can be to refer to AMD's Brixelizer [Kra23] to employ local mesh ROMAs coupled with AABB tree traversal.

8. Conclusion and Future Work

We have presented Ray-aligned Occupancy Map Array (ROMA), a new software solution that enables fast approximate ray tracing, by producing multiple rotated versions of a scene/object. ROMA is fast to generate, requiring only one base occupancy map (BOM) to be rasterized then rotated, therefore effectively supports dynamic objects. ROMA is also fast to perform visibility queries, providing an $O(1)$ ray-aligned coherent tracing scheme. Moreover, by tuning different positional and angular resolutions of ROMA, it offers a fully scalable solution to balancing the performance and quality in a spatiotemporal way.

While we believe in the bright future when full hardware ray tracing (HWRT) will take over, it is still uncertain how long we will have to wait before it happens. During the interim, we also believe it worthy to study HWRT alternatives and their hybrid solutions. As for our ROMA solution, in the near future, it would be of immediate interest to look for hardware support for ROMA to boost its performance. Meanwhile, using geometry shaders to improve the performance of BOM generation can also help – until the performance is only related to its positional resolution rather than the number of triangles. A hybrid solution like combining screen-space ray tracing for near-field tracing and using ROMA only for far-field tracing could also improve the practicality of ROMA.

Acknowledgement. This work was supported by the National Key R&D Program of China (2020YFB1708903), the National Natural Science Foundation of China (62272275). Ling-Qi Yan is supported by gift funds from Adobe, Intel, Meta and XVerse.

References

[Aal18] AALTONEN, SEBASTIAN. "GPU-based clay simulation and ray-tracing tech in Claybook". *San Francisco, CA* 2.5 (2018) 8.

[ANA*20] ANDERSSON, PONTUS, NILSSON, JIM, AKENINE-MÖLLER, TOMAS, et al. "FLIP: A Difference Evaluator for Alternating Images." *Proc. ACM Comput. Graph. Interact. Tech.* 3.2 (2020), 15–1 8.

[AW*87] AMANATIDES, JOHN, WOO, ANDREW, et al. "A fast voxel traversal algorithm for ray tracing." *Eurographics*. Vol. 87. 3. 1987, 3–10 2, 3.

[BM08] BAVOIL, LOUIS and MYERS, KEVIN. "Deferred rendering using a stencil routed k-buffer". *ShaderX6: Advanced Rendering Techniques* (2008), 189–198 3.

[BS08] BAVOIL, LOUIS and SAINZ, MIGUEL. "Screen space ambient occlusion". *NVIDIA developer information: http://developers. nvidia. com* 6.2 (2008) 2.

[BWP*20] BITTERLI, BENEDIKT, WYMAN, CHRIS, PHARR, MATT, et al. "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting". *ACM Transactions on Graphics (TOG)* 39.4 (2020), 148–1 3.

[CNS*11] CRASSIN, CYRIL, NEYRET, FABRICE, SAINZ, MIGUEL, et al. "Interactive indirect illumination using voxel cone tracing". *Computer Graphics Forum*. Vol. 30. 7. Wiley Online Library. 2011, 1921–1930 3, 11.

[DCB*04] DONG, ZHAO, CHEN, WEI, BAO, HUJUN, et al. "Real-time voxelization for complex polygonal models". *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.* IEEE. 2004, 43–50 2.

[DS05] DACHSBACHER, CARSTEN and STAMMINGER, MARC. "Reflective shadow maps". *Proceedings of the 2005 symposium on Interactive 3D graphics and games*. 2005, 203–231 7.

[ED06] EISEMANN, ELMAR and DÉCORET, XAVIER. "Fast scene voxelization and applications". *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. 2006, 71–78 2, 3.

[FBP09] FOREST, VINCENT, BARTHE, LOIC, and PAULIN, MATHIAS. "Real-time hierarchical binary-scene voxelization". *journal of graphics, gpu, and game tools* 14.3 (2009), 21–34 2, 8.

[HFF18] HE, YONG, FATAHALIAN, KAYVON, and FOLEY, TIM. "Slang: language mechanisms for extensible real-time shading systems". *ACM Transactions on Graphics (TOG)* 37.4 (2018), 1–13 7.

[HHGM10] HERMES, JAN, HENRICH, NIKLAS, GROSCH, THORSTEN, and MUELLER, STEFAN. "Global Illumination using Parallel Global Ray-Bundles." *VMV*. 2010, 65–72 3.

[KCK*22] KALLWEIT, SIMON, CLARBERG, PETRIK, KOLB, CRAIG, et al. *The Falcor Rendering Framework*. https://github.com/NVIDIAGameWorks/Falcor. Aug. 2022. URL: https://github.com/NVIDIAGameWorks/Falcor 7.

[Kra23] KRAMER, LOU. "Real-time Sparse Distance Fields for Games". *Game Developers Conference (GDC)*. 2023 13.

[LM22] LINIETSKY, JUAN and MANZUR, ARIEL. *Godot Engine*. https://godotengine.org/, Last accessed on 2023-01-17. 2022 3, 4, 13.

[MESK22] MÜLLER, THOMAS, EVANS, ALEX, SCHIED, CHRISTOPH, and KELLER, ALEXANDER. "Instant neural graphics primitives with a multiresolution hash encoding". *arXiv preprint arXiv:2201.05989* (2022) 3.

[MGNM19] MAJERCIK, ZANDER, GUERTIN, JEAN-PHILIPPE, NOWROUZEZAHRAI, DEREK, and MCGUIRE, MORGAN. "Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields". *Journal of Computer Graphics Techniques (JCGT)* 8.2 (June 2019), 1–30. ISSN: 2331-7418. URL: http://jcgt.org/published/0008/02/01/ 3.

[MMK*21] MAJERCIK, ZANDER, MÜLLER, THOMAS, KELLER, ALEX, et al. "Dynamic Diffuse Global Illumination Resampling". (Dec. 2021). Computer Graphics Forum, 13. URL: https://casual-effects.com/research/Majercik2021Resampling/index.html 3.

[MMSM21] MAJERCIK, ZANDER, MARRS, ADAM, SPJUT, JOSEF, and MCGUIRE, MORGAN. "Scaling Probe-Based Real-Time Dynamic Global Illumination for Production". *Journal of Computer Graphics Techniques (JCGT)* 10.2 (May 2021), 1–29. ISSN: 2331-7418. URL: http://jcgt.org/published/0010/02/01/ 3.

[MRNK21] MÜLLER, THOMAS, ROUSSELLE, FABRICE, NOVÁK, JAN, and KELLER, ALEXANDER. "Real-time neural radiance caching for path tracing". *arXiv preprint arXiv:2106.12372* (2021) 7, 11.

[NVI] NVIDIAGAMEWORKS. *NVIDIAGameWorks/Raytracingdenoiser: Nvidia ray tracing denoiser*. URL: https://github.com/NVIDIAGameWorks/RayTracingDenoiser 8.

[RBA09] REINBOTHE, CHRISTOPH K, BOUBEKEUR, TAMY, and ALEXA, MARC. "Hybrid Ambient Occlusion." *Eurographics (Areas Papers)* 5 (2009) 3.

[REG*09] RITSCHEL, TOBIAS, ENGELHARDT, THOMAS, GROSCH, THORSTEN, et al. "Micro-rendering for scalable, parallel final gathering". *ACM Transactions on Graphics (TOG)* 28.5 (2009), 1–8 3.

[RGK*08] RITSCHEL, TOBIAS, GROSCH, THORSTEN, KIM, MIN H, et al. "Imperfect shadow maps for efficient computation of indirect illumination". *ACM transactions on graphics (tog)* 27.5 (2008), 1–8 3, 5.

[RGS09] RITSCHEL, TOBIAS, GROSCH, THORSTEN, and SEIDEL, HANS-PETER. "Approximating dynamic global illumination in image space". *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, 75–82 2.

[RT06] RONG, GUODONG and TAN, TIOW-SENG. "Jump flooding in GPU with applications to Voronoi diagram and distance transform". *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. 2006, 109–116 3, 4, 8, 9.

[SA09] SINTORN, ERIK and ASSARSSON, ULF. "Hair self shadowing and transparency depth ordering using occupancy maps". *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, 67–74 2.

[SBM03] STANEKER, DIRK, BARTZ, DIRK, and MEISSNER, MICHAEL. "Improving occlusion query efficiency with occupancy maps". *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003.* IEEE. 2003, 111–118 2.

[SC97] SHIRLEY, PETER and CHIU, KENNETH. "A low distortion map between disk and square". *Journal of graphics tools* 2.3 (1997), 45–52 5.

[SJ22] SHARP, NICHOLAS and JACOBSON, ALEC. "Spelunking the deep: guaranteed queries on general neural implicit surfaces via range analysis". *ACM Transactions on Graphics (TOG)* 41.4 (2022), 1–16 3.

[SP98] SZIRMAY-KALOS, LÁSZLÓ and PURGATHOFER, WERNER. "Global ray-bundle tracing with hardware acceleration". *Rendering Techniques' 98: Proceedings of the Eurographics Workshop in Vienna, Austria, June 29—July 1, 1998 9*. Springer. 1998, 247–258 3.

[SPD18] SCHIED, CHRISTOPH, PETERS, CHRISTOPH, and DACHSBACHER, CARSTEN. "Gradient estimation for real-time adaptive temporal filtering". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1.2 (2018), 1–16 6.

[TDD*22] TATARCHUK, NATALYA, DUPUY, JONATHAN, DELIOT, THOMAS, et al. "Advances in Real-Time Rendering in Games: Part I". *ACM SIGGRAPH 2022 Courses*. SIGGRAPH '22. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2022. ISBN: 9781450393621. DOI: 10.1145/3532720.3546895. URL: https://doi.org/10.1145/3532720.3546895 2–4, 7, 12, 13.

[THGM11] THIEDEMANN, SINJE, HENRICH, NIKLAS, GROSCH, THORSTEN, and MÜLLER, STEFAN. "Voxel-based global illumination". *Symposium on Interactive 3D Graphics and Games*. 2011, 103–110 2–4.

[TKD*14] TATARCHUK, NATASHA, KARIS, BRIAN, DROBOT, MICHAL, et al. "Advances in Real-Time Rendering in Games, Part I (Full Text Not Available)". *ACM SIGGRAPH 2014 Courses*. SIGGRAPH '14. Vancouver, Canada: Association for Computing Machinery, 2014. ISBN: 9781450329620. DOI: 10.1145/2614028.2615455. URL: https://doi.org/10.1145/2614028.2615455 2, 3, 6, 8.

[TLY*21] TAKIKAWA, TOWAKI, LITALIEN, JOEY, YIN, KANGXUE, et al. "Neural geometric level of detail: Real-time rendering with implicit 3D shapes". *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, 11358–11367 3.

[Ulu18] ULUDAG, YASIN. "Hi-Z screen-space cone-traced reflections". *GPU Pro 360 Guide to Lighting*. AK Peters/CRC Press, 2018, 237–280 2, 4.

[VSJ22] VICINI, DELIO, SPEIERER, SÉBASTIEN, and JAKOB, WENZEL. "Differentiable signed distance function rendering". *ACM Transactions on Graphics (TOG)* 41.4 (2022), 1–18 11.

[Wri15] WRIGHT, DANIEL. "Dynamic occlusion with signed distance fields". *ACM SIGGRAPH*. Vol. 3. 2015 11.