A Multiview and Multilayer Approach for Interactive Ray Tracing

Kostas Vardis* and Andreas A. Vasilakis† and Georgios Papaioannou‡
Department of Informatics, Athens University of Economics & Business, Greece

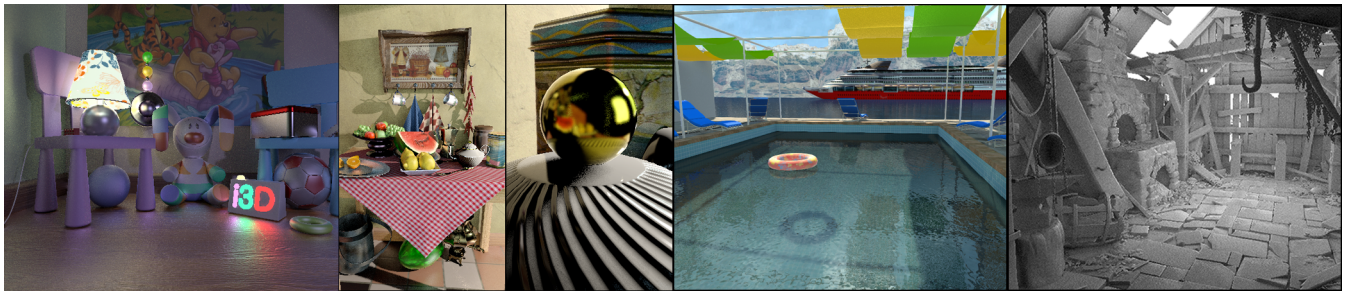


Figure 1: *Our multiview-layered approach produces visually convincing path tracing and ambient occlusion results in complex and dynamic environments, where both near and distant information as well as off-screen geometry is properly captured.*

Abstract

We introduce a generic method for interactive ray tracing, able to support complex and dynamic environments, without the need for precomputations or the maintenance of additional spatial data structures. Our method, which relies entirely on the rasterization pipeline, stores fragment information for the entire scene on a multiview and multilayer structure and marches through depth layers to capture both near and distant information for illumination computations. Ray tracing is efficiently achieved by concurrently traversing a novel cube-mapped A-buffer variant in image space that exploits GPU-accelerated double linked lists, decoupled storage, uniform depth subdivision and empty space skipping on a per-fragment basis. We illustrate the effectiveness and quality of our approach on path tracing and ambient occlusion implementations in scenarios, where full scene coverage is of major importance. Finally, we report on the performance and memory usage of our pipeline and compare it against GPGPU ray tracing approaches.

Keywords: rasterization, ray tracing, cubemap, A-buffer, GPU

Concepts: •**Computing methodologies** → **Rasterization; Ray tracing;**

1 Introduction

The estimation of global illumination via ray tracing methods is a heavy computational process that requires accurate ray-object intersections for a massive number of rays that propagate through the virtual environment. For efficiency reasons, common CPU and GPGPU ray tracing methods require the construction and maintenance of acceleration data structures in order to achieve interactive framerates. However, this procedure is still not suited for fully dynamic environments, where the entire scene can potentially change in every frame [Bittner and Meister 2015]. Alternatively, rasterization-based methods, which commonly operate either in image- or volume-space can elegantly support dynamic environments. However, image-based methods are prone to view-dependent artifacts, resulting in significant energy loss, while volume-based methods are limited in the type and frequency of phenomena they are able to capture. Hybrid methods attempt to bridge the gap between rasterization-based and ray-tracing approaches. They can provide efficient results but require multiple data representations and are limited either to tightly bounded scenes or in the type of effects they are able to reproduce.

This paper proposes an image-space ray-tracing method that effectively combines the advantages of voxel-based approaches (complete geometry) and image-based methods (highly detailed geometric data, fully-dynamic environments), for interactively approximating global illumination phenomena. We perform multifragment rendering in a cubemap configuration in order to efficiently store geometric information in high-detail for the entire scene. This way, view-dependent errors are reduced significantly. Near and distant geometric information is captured by adapting screen-space ray tracing [McGuire and Mara 2014] on a multiview and multilayered scheme. By moving the computational overhead of a global acceleration structure to a per-fragment basis and relying entirely on the rasterization pipeline, which GPUs are highly optimized for, we are able to efficiently render arbitrary *dynamic* environments of any complexity. Furthermore, the realization can benefit from features such as multisample antialiasing, tessellation, displacement, instancing and primitive deformation.

To this end, we introduce a novel A-buffer that exploits decoupled bucket-aware double-linked lists. This structure provides fast layer sorting, efficient bi-directional list traversal (with respect to the camera) and empty-space skipping. Since our method is generic, any multifragment variant can be used as a trade-off between performance and memory consumption, as demonstrated by our comprehensive measurements using several common A-buffer variants. Finally, a performance and quality evaluation is provided on path tracing and ambient occlusion implementations in a collection of complex environments demonstrating that our approach is highly suitable for rendering dynamic scenes at interactive rates and fast previewing of a scene during editing.

*kvardis@aueb.gr
†abasilak@aueb.gr
‡gepap@aueb.gr

To summarize, the main contributions of this work are:

- A generic multilayer approach for image-space ray tracing, based on a cubemapped representation of the entire scene that lifts many of the restrictions of previous approaches.
- A search-aware A-buffer method using *decoupled* storage and *double linked lists* for fast construction and efficient bi-directional ray traversal suitable for a large number of layers. Per-fragment *empty space skipping* is achieved by arranging the fragments into uniformly divided depth-based buckets.

This paper is organized as follows: Section 2 offers a detailed summary of prior work. Section 3 describes our cubemap A-buffer architecture and generalizes ray tracing in a multilayer and multiview pipeline. Section 4 reports on the efficiency and robustness of our method by providing visual results on several global illumination effects as well as a comprehensive analysis and comparison between different multifragment methods. Finally, Section 5 offers conclusions and identifies promising research directions.

2 Related Work

Image-based methods. In general, image-based approaches take advantage of the fragment information in the available G-buffers and use this as an approximation of the entire scene. They can capture contact details, produce convincing results at real-time frame rates and simulate various effects, such as ambient occlusion [Mittring 2007], refractions [de Rousiers et al. 2011] or even photon tracing [McGuire and Luebke 2009]. *Multilayer* approaches capture missing information inside the view frustum, either by using multiple layers [Nießner et al. 2010; Bauer et al. 2013; Mara et al. 2014] or tessellation [Nalbach et al. 2014]. Finally, *multiview* approaches attempt to retrieve the missing information by relying on other views, such as, the already available shadow maps [Vardis et al. 2013].

Despite their fast performance, the result is view-dependent due to the missing information inside and outside the viewing frustum, leading to significant energy loss. An illustrative example is shown in Fig. 2(a). Front-facing surface information present in the depth buffer and its back-facing or subsequent layers can be efficiently retrieved using single layer (green sides) and multilayer (blue sides) methods respectively. However, surfaces outside the primary view (red sides) are missed.

Close to our work, Ganestam and Dogget [2015] use a hybrid approach to render perfect reflections and refractions. A boundary volume hierarchy is created to store the geometry near the camera and a cubemap of G-buffers, each with a field of view of 90 degrees, for the rest. Their method, which requires both GPGPU and rasterization pipelines, can deliver interactive frame rates but is still prone to view-dependent artifacts due to the single layer information stored in each face. Very recently, Widmer et al. [2015] utilize their adaptive acceleration structure on a cubemapped variation to offer multilayer reflections by off-screen objects at interactive frame rates, but their results are limited to two-layered representations, opaque geometry and diffuse surfaces. Contrary to these approaches, our method is not limited to a particular type of surface or effect, is applicable to a large number of layers and can capture multilayer information inside and outside the viewing frustum, limiting view dependencies to a minimum.

Volumetric methods. Volume-based methods provide a view-independent representation of a bounded volume within the environment and make the assumption that a voxel uniformly represents a neighborhood of geometric elements and corresponding light interactions, limiting the detail of the stored information.

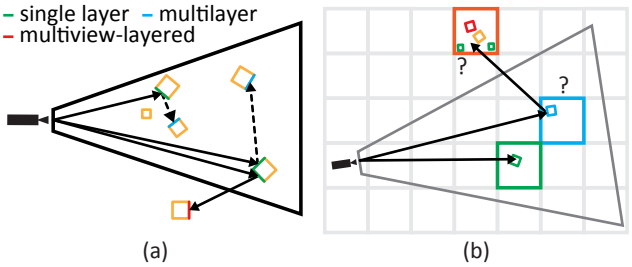


Figure 2: *View-dependency problems manifest from missing geometric information inside and outside the viewing frustum in image-based methods (a), while a coarse scene approximation limits the quality of information that can be stored in voxel-based ones (b).*

As a result, energy exchange is approximate and contact details as well as small-scale surface information cannot be accurately captured (see Fig. 2(b)), thus limiting the frequency of the properly reproduced phenomena.

In the spirit of ray tracing, Crassin et al. [2011] compute single bounce indirect illumination for Lambertian and Phong BRDFs, where illumination is estimated at the shaded points by cone marching in a sparse voxel octree structure. Zirr et al. [2014] perform object-order ray tracing by exploiting a coarse voxel grid that stores links to all rays which intersect each cell, but their method is limited to single bounce indirect illumination. Hu et al. [2014] proposed a full ray tracing solution using only the rasterization pipeline. The scene is projected orthographically onto three A-buffers along with a low resolution volume for fast ray-fragment intersection tests. The volume, which contains the fragments' average normal, is traversed with a 3D Digital Differential Analyzer (DDA) algorithm to quickly identify a general hit location, and the A-buffers are then traced to locate the intersected fragment. However, the authors assume that each voxel contains one smooth surface and their method is therefore applicable to scenes of limited extent, in order to properly capture the underlying geometric detail.

Multifragment Rendering. The A-buffer can objectively be considered as the most preferred multilayer framework for maintaining all generated per-pixel fragments in correct depth order. In contrast to visibility determination applications (e.g. order-independent transparency) where only depth information is necessary, additional surface properties are required for global light transport computations [Mara et al. 2014]. While a multitude of GPU-accelerated A-buffer structures exist, including variable arrays [Vasilakis and Fudos 2012] and the use of one or more per-pixel linked lists [Yang et al. 2010; Vasilakis and Fudos 2013], none of them focuses on fast ray-fragment intersections. In this work, we introduce an A-buffer variant optimized for screen space ray tracing by offering a bi-directional traversal mechanism, multiple per-pixel double linked lists and decoupled visibility-shading data.

3 Method Description

3.1 Overview

Our algorithm operates in three main steps, as illustrated in Figure 3. Initially, we create 6 views, aside from the primary view frustum, using the center of projection of the latter. The near and far planes of each of the 6 view frusta are shifted to ensure full coverage of the entire scene (see Sec. 3.2 and Fig. 4). Then, generated fragments for each one of the views are subsequently captured via multifragment rendering. Instead of utilizing one of the widely accepted multifragment alternatives, we introduce a new

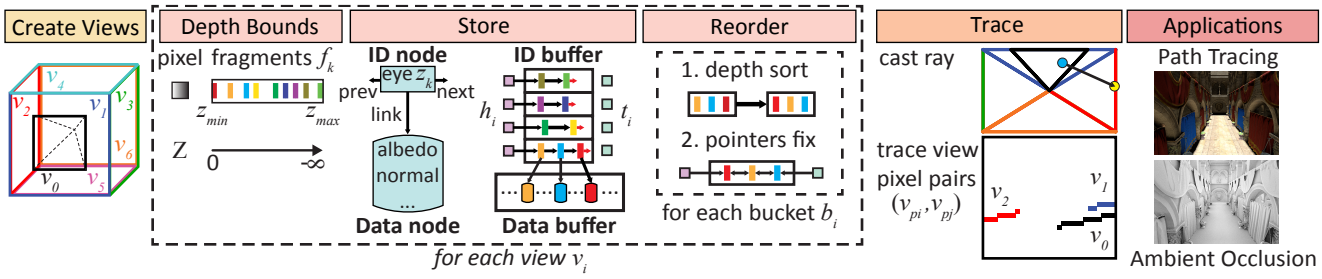


Figure 3: *Diagram of our ray-tracing pipeline. Camera view and cubemap configuration (left). Construction of our novel A-buffer. For each pixel of every view, depth min/max bounds are computed, geometry is rasterized, uniformly subdivided, stored in decoupled fashion and finally sorted (middle). Views of the A-buffer cubemap are potentially traced for the production of global illumination effects (right).*

A-buffer structure that aims at maximizing the efficiency of the performance-critical ray tracing stage at the expense of extra memory allocation (Sec. 3.3). In the final step, our layered multiview structure is exploited to perform accelerated computations on tracing applications by ray marching in screen-space increments and switching between different viewports, when required (Sec. 3.4). Ray fragment intersections are computed by keeping track of the ray's eye-space Z entry and exit values at each marching step and checking whether the stored fragments are within this range.

Regarding the per-pixel fragment storage, we evolved the bucketed linked-list A-buffer [Vasilakis and Fudos 2013] to increase the queries performance in the specific case of the ray marching paradigm: (i) An adaptive scheme is explored, where all buckets can handle an arbitrary number of rasterized fragments, avoiding potentially large and unused pre-allocated storage. (ii) A double linked-list is constructed for fast bi-directional list traversal. In a generic ray tracing setup, the A-buffer is traversed in any order, according to the direction of the rays. The benefits of this modification are especially noticeable in scenes of high depth complexity. (iii) Fragments are assigned in buckets in linear instead of post-projective space. This way, overloading the last few buckets is avoided, a common occurrence due to the perspective projection for large far-over-near ratios. (iv) Depth and shading per-fragment data are stored separately in order to improve both fragment sorting and traversal by significantly reducing the latency of global GPU memory. Note that employing a bucketed scheme provides huge ray traversal benefits by performing efficient empty space skipping at arbitrary pixel locations.

3.2 Cubemap Configuration

The first stage of our method is responsible for configuring the necessary view frusta $v_j \in [v_0, \ldots, v_6]$. The first one, v_0, corresponds to the typical camera view with all its associated parameters, such as the field of view (FOV), near/far planes and resolution. Usually, the latter one is set to the framebuffer resolution.

The 6 additional views, $v_1, \ldots, v_6$, are distributed in an world-space axis-aligned cubemap centered at the camera position with FOV $= 90^o$ and square dimensions. This cubemap configuration is simple to implement and results in an overall acceptable distortion. In order to cover the entire scene, the far clipping planes are adjusted to the corresponding extent of the scene's bounding box (left in Fig. 4).

To cover the clipped volume behind the near planes of the frusta, each cubemap frustum center of projection is slightly shifted backwards according to the near clipping distance. This creates some frustum overlap, but considering typical near clipping plane distances, this results in negligible redundancy. Figure 4 (right) demonstrates this modification, where the near plane distances are deliberately set far enough for illustration purposes.

The main advantage of keeping the camera view, at the cost of the extra memory required, is that it disassociates the user's FOV from those of the cubemap faces. Thus, the direct lighting pass can be efficiently executed without casting extra rays between the camera position and the first hit location (unless anti-aliasing or depth of field effects are explored). Furthermore, it allows a downgrade of the cubemap resolution, independent of the geometry captured in the primary buffer, for performance or memory savings.

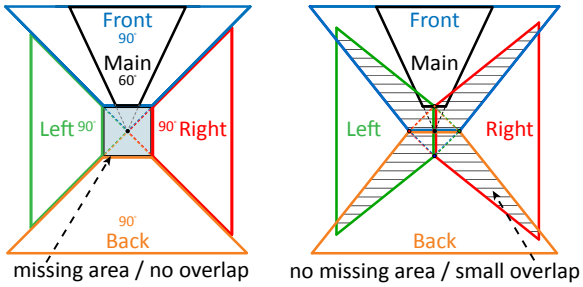


Figure 4: *Our 7-view camera setup (5 are illustrated for clarity). The clipped volume bounded by the near clip planes (left) is eliminated by offsetting each frustum backwards (right).*

3.3 A-buffer Structure

Once the view information has been generated, our A-buffer structure is constructed in three stages: Initially, we divide the depth range of each pixel p (of each view v_j) into B uniformly consecutive subintervals and assign each span to one $b_i \in [b_0, \ldots, b_{B-1}]$ bucket (*depth bounds* stage). Then, we concurrently store each one of the per-pixel fragments in its bucket by exploring a layered and decoupled data representation that connects nodes via GPU-accelerated double linked lists (*store* stage). A final step sorts fragments by their depth and assigns the appropriate pointers to the correct fragment nodes (*reorder* stage).

Data Structures. Our approach initially requires $2 + 2B$ textures (32-bit unsigned integers) for each view. These include (i) a *depth bounds* texture responsible for storing the min/max depth values $[z_{min}(p), z_{max}(p)]$ of pixel p as well as (ii) B *head* $h_i \in [h_0, \ldots, h_{B-1}]$ and *tail* $t_i \in [t_0, \ldots, t_{B-1}]$ pointers linking the fragments between each view's buckets. To avoid traversal performance bottlenecks that arise from GPU memory latency, fragment attributes are stored in two separate buffer objects, an *ID buffer* whose nodes are connected via double linked lists, and a *Data buffer* (see Fig. 3). Each node of the ID buffer contains (i)

a 32-bit floating point *eye space Z* value (the essential attribute for depth-ordering) and (ii) two 32-bit unsigned integers for the *next/prev* pointers between nodes. The index of each ID node serves as a *link*, mapping the corresponding fragment to its shading data node. The remaining attributes required for shading computations (which depends on the application) are stored in each data node and are read when a ray hit has been identified. Each node (from both buffers) is placed in the next available memory address location according to a single specialized atomic counter (instead of using one counter per bucket [Vasilakis and Fudos 2013]), resulting in an adaptive memory-aware fragment storage scheme.

Depth Bounds Stage. Initially, the depth bounds texture is cleared and a quick rendering pass over the geometry is performed by setting depth testing off. This can either be a simplified but accurate depth-only pass over the actual geometry or an approximate and faster pass, where each object is replaced by its bounding box. During the rendering pass, each fragment assigns its eye-space Z value to the depth bounds texture using atomic min and max operations.

Store Stage. A full geometry pass is performed at this stage in order to capture all rasterized fragment information in an unsorted sequence. To avoid executing different geometry passes for each view, the geometry shader is used to emit each primitive to every view. Each fragment $f_k \in [f_0, \ldots, f_{n-1}]$ in a pixel p is mapped to its corresponding bucket by checking the pixel's depth range with the fragment's interpolated eye space Z value z_k such as $b(f_k) = \lfloor B \cdot (z_k - z_{min}(p)) / (z_{max}(p) - z_{min}(p)) \rfloor$. The storage location in both ID and Data buffers is pointed by atomically incrementing the *next* counter. An atomic exchange operation sets the next pointer of the ID node to the head pointer, which is afterwards replaced by the current ID node address. Finally, the Data node is filled with the fragment shading information. At this stage, no previous node or tail pointers are assigned since they will be set after correcting depth order at the reorder stage. The head and tail textures as well as the next counter have been initially cleared to zero values.

Reorder Stage. Finally, a full-screen quad rendering pass is initiated, reordering the ID nodes for each bucket via insertion or shell sort. Then, the corrected next/head pointers as well as the newly assigned previous node/tail pointers are set for each fragment/pixel respectively.

3.4 Ray Tracing

Image-space ray tracing. Once the multilayer cubemap construction has been completed, arbitrary ray tracing is performed in a manner similar to the *single view* 2D DDA algorithm which iteratively samples pixel locations until a ray-fragment collision is detected [McGuire and Mara 2014]. In that work, given a ray r in 3D space with starting position $\mathbf{x}_r$ and direction $\mathbf{d}_r$, the ray's end point for the current view $\mathbf{y}_r$ is computed by clipping the ray against the viewing frustum. Then, the start and end pixel locations $\mathbf{x}_p$, $\mathbf{y}_p$ are computed by projecting $\mathbf{x}_r$, $\mathbf{y}_r$ to screen space, respectively. For each sampled pixel location on the segment $\overrightarrow{\mathbf{x}_p\mathbf{y}_p}$, the corresponding eye-space Z coordinates $z_{min}(r), z_{max}(r)$ of the intersection of the ray segment with the pixel boundaries are computed and a hit is found when the Z coordinate of the closest fragment in eye space for this pixel is in the range $[z_{min}(r), z_{max}(r))$.

It should be noted that for performance reasons, a predetermined number of ray marching steps can be used instead of exhaustively visiting consecutive pixels. In this case, samples are jittered using a fixed offset per ray. Alternatively, a hierarchical approach can be explored instead [Uludag 2014].

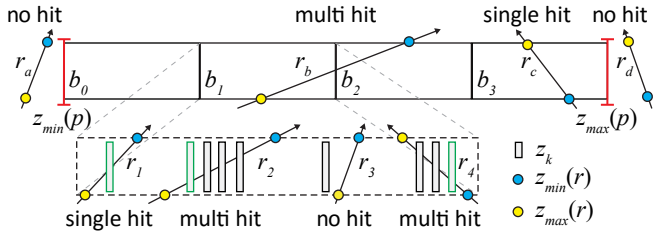


Figure 5: *Different ray-bucket (top) and ray-fragment (bottom) hit cases. Rays passing outside the pixel/bucket boundaries can efficiently skip fragment traversal ($r_a, r_d / r_b, r_c$). Rays crossing multiple fragments (r_2, r_4) use their direction to determine the hit.*

For multiple layers (limited to 4 in the original paper), a sequential front-to-back traversal of the fragment list is performed. However, that algorithm does not always exhibit error-free behavior, as the correct hit does not depend only on the ray extents but also on the ray's Z direction (e.g. ray r_4 in Fig. 5). Returning the correct hit is crucial for illumination computations as these fragments may belong to entirely different surfaces.

To correctly handle fragment lists of any length, arbitrary ray traversals and multiple views, in this work we augment and revise screen-space ray tracing, as explained below. Furthermore, decoupled storage and empty space skipping optimizations are employed in order to alleviate the performance bottlenecks that arise from the different ray-fragment collision tests.

Single view ray tracing. Figure 5 illustrates the five different ray-fragment hit cases that arise; a ray may pass outside pixel depth boundaries in either side (r_a, r_d), it may cross one (r_c) or more (r_b) buckets, having zero (r_3), one (r_1) or multiple fragment hits (r_2, r_4). We identify two cases where we can reduce ray-fragment intersection tests; for rays that do not cross any fragment since they reside outside pixel depth boundaries (r_a, r_d), we can perform an *early skip*. Secondly, for rays that do not cross the entire pixel depth range but intersect only a bucket subset (r_b, r_c), we can perform *fragment bucket skipping*.

The fragment intersection algorithm consists of three steps: Initially, we discard rays that reside outside the pixel's depth boundaries by performing a fast interval comparison, where the ray's min/max Z values are tested against the pixel's max/min depth bounds: $z_{max}(r) \leq z_{min}(p)$ or $z_{min}(r) > z_{max}(p)$ (step a). If the ray is not culled, we compute which bucket(s) it intersects in constant time (step b). Finally, we linearly iterate through all fragments in each intersected bucket and check which fragments are within the ray interval. Rays that move towards the near plane are checked in back-to-front order (through the previous node pointer), while rays towards the far plane are checked in the opposite order (via the next node pointer) (step c). In case of multiple hits, the returned fragment is the first one encountered based on the direction of list traversal, given by the sign of the Z ray direction.

Switching between different views. In a single view approach, no valid hit is detected if the ray exits the frustum boundaries. In a connected multiple view setting, rays must switch frusta as they cross their boundaries until the far clipping plane of one of them is encountered. A brute-force approach would iterate through all frusta in order to find the next view the ray traverses. However, this simple approach suffers from two limitations. First, as clipping occurs within half-pixel intervals, the ray might not entirely exit the current frustum. This is resolved by snapping the ray to the outside of the viewport by a small offset. Second, if the ray exits through any frustum side (except the near clipping plane), the next

view is well-determined and no iteration is required (e.g. exiting the bottom view via the bottom plane switches the ray traversal to the back view). In that case, we switch between views in constant time by keeping viewport edge connectivity information. However, in the very infrequent case of the ray exiting the near plane, an iteration through all frusta cannot be avoided. A multiview tracing example is illustrated in Figure 6.

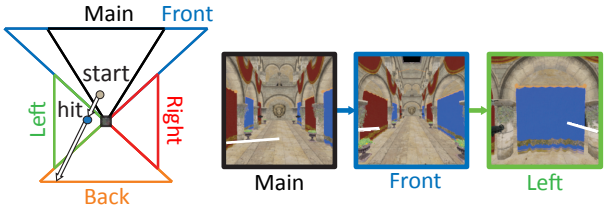


Figure 6: *Ray tracing example on the Sponza scene. Left: Overview of the multiview frusta. Right: Screen-space ray traversal. The ray, shown in white color, starts on the main view, continues to the front cubemap face and finally finds a hit in the left frustum.*

4 Results and Discussion

Typical illumination algorithms that rely on ray tracing and use information of the entire scene can benefit from our approach. Instead of providing results on isolated global illuminated effects, such as reflections, refraction etc. we demonstrate our method on a full path tracing implementation as well and as ambient occlusion (Sec. 4.1). An experimental analysis of our method against general purpose GPU ray tracing implementations is provided, both in terms of quality and performance (Sec. 4.2). We also compare our specialized multifragment structure against prior A-buffer variants in terms of performance and memory usage for ray-fragment collision queries involving arbitrary pixel fragments (Sec. 4.3). To our knowledge, such a test has not been conducted so far on a multilayer method.

Experiments setup. We evaluated our method with respect to performance and quality on a variety of testing scenarios. These include environments containing static geometry as well as rigid and deformable mesh animations. The images were rendered at a resolution of 720×480 for the main view and 480^2 for the cubemap faces on a Geforce GTX 780 Ti. For the stills shown, the number of samples (paths) per pixel (spp) was set to 200 and the number of sampling locations for each ray (ray marching steps) was set to 100 for each view, unless specified otherwise. This testing configuration was chosen in order to keep the rendering times as close to the real-time domain as possible. Note that all timings include the direct lighting calculations, which come at a minimal cost due to rasterization. The number of buckets was fixed to $B = 4$, which provided a good balance between performance and memory consumption. We also provide as supplemental material the shader source code for the construction of our multiview A-buffer with the path tracing and ambient occlusion implementations as well as the single view construction of the A-buffer variants shown in Figure 12.

4.1 Applications

Path tracing (PT). A complete path tracing algorithm for fully dynamic environments can be easily implemented using our approach. All fragment data needed for shading are stored in the Data buffer. Therefore, material information such as normals, surface roughness, reflectivity and index of refraction are only involved and accessed during shading, without causing a bottleneck in the A-buffer construction and tracing apart. We trace a single ray per pixel using BSDF importance sampling, starting from the nearest visible fragment and trace one additional ray per light bounce to form a path. Since tracing shadow rays comes at the same cost as tracing path segments, we rely on the shadow maps instead. Multiple samples per pixel are accumulated as separate passes, progressively or at a fixed rate. The current number of samples is stored in the alpha channel of the output frame buffer so that adaptive sampling can be easily performed for each pixel individually.

Ambient occlusion (AO). Implementing ambient occlusion is a straightforward simplification of the path tracing mechanism. For each surface, we send ray samples around the hemisphere of directions and use our tracing method to identify a hit. Again, multiple visibility samples are gathered with multiple one-ray passes. The only change relative to the multiview construction is the information stored in the Data buffer, which requires only the normal vectors. It should be noted that since the shading information stored is minimal, a decoupled representation has a much smaller performance benefit compared to path tracing.

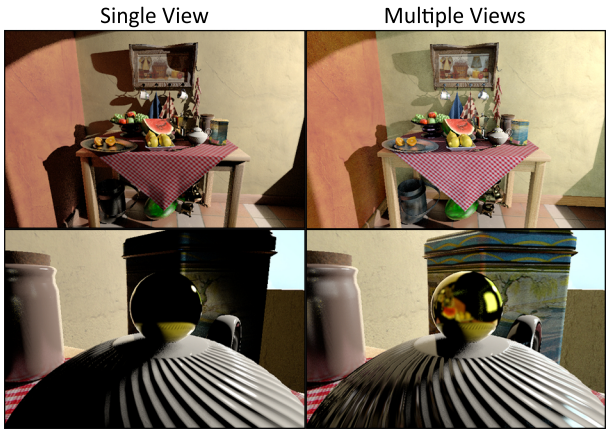


Figure 7: *1-indirect bounce path tracing comparison between single and multiple views on the Dead Nature scene. The energy loss in the single view case (left) is clearly shown in both view points, but becomes significant when visually important illumination comes from parts of the scene outside the primary view (bottom). Images rendered at 17ms, 37ms (top) and 20ms, 57ms (bottom) per ray path respectively.*

4.2 Quality and Performance Analysis

Multiple views / multiple layers. The visual impact of using a layered multiview representation in PT is shown in Figures 7 and 8. A single view and/or single layer setting can exhibit significant energy loss, since visually important illumination is present only in certain views/layers. This becomes even more apparent in scenarios where only a fraction of the scene is available in the primary view (bottom row in Fig. 7). Note than even using multiple views with a single depth layer may still miss important surfaces (bottom left in Fig. 8) since the reflected/refracted rays can hit surfaces in the deeper layers. Similarly, Figure 9 demonstrates AO on the Smithy model from the Blacksmith environment package of Unity game engine. Note that geometry layers residing outside the viewport can only affect the result with our approach (bottom right inset).

In terms of performance, we observe that applying multiple views comes at an increased cost of $3 - 5\times$ on average, depending on the number of views required for each ray to hit a fragment or exit the scene boundaries. Figure 10 illustrates AO and PT in scenes with deformable geometry, showcasing the fact that our method is suitable for interactive preview of arbitrarily dynamic scenes.

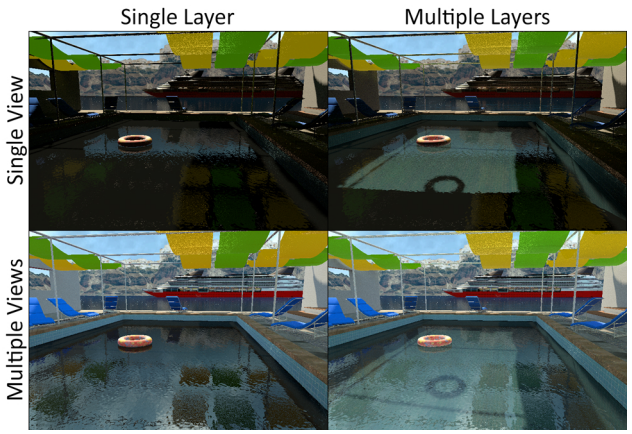


Figure 8: *2-indirect bounce path tracing of the Pool scene with different view/layer settings. Even though most refracted rays can be captured with a multilayer approach, reflections require a multiview-layered approach. Images rendered at 30ms, 58ms (top) and 110ms, 245ms (bottom) per path respectively.*

High-quality renderings (1000spp) of our method are also shown in Figure 1, which demonstrate that contact details, distant geometry and off-screen information are properly captured on a variety of environments. Note that while all scenes are illuminated by one point light source, the Bunny scene (first image) is further lit by emissive geometry. Measurements regarding the performance and memory of these images are shown in Table 1. The main observations are: (i) the construction times depend on the overall complexity of the scene, (ii) the tracing performance is directly related to the efficiency of empty space skipping as well as the number of buckets used and (iii) the average number of layers traversed during tracing remains relatively constant, independent of the number of indirect bounces. Regarding the number of buckets, in complex scenes such as the Smithy model, we can further improve tracing performance by increasing B to 8-12, at the expense of the extra construction cost during the resolve stage and the increased memory requirements.

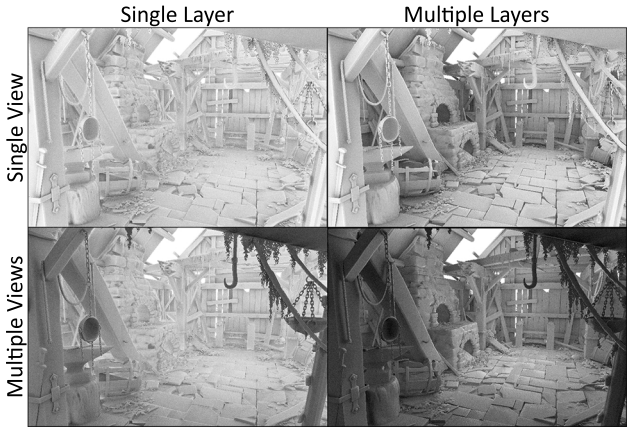


Figure 9: *Ambient occlusion with a range of 3m on the Smithy model with different view/layer settings. Note how the missing occlusion is rectified by including multiple layers (right) and views (bottom). Images rendered at 8ms, 13ms (top) and 28ms, 35ms (bottom) per ray path respectively.*

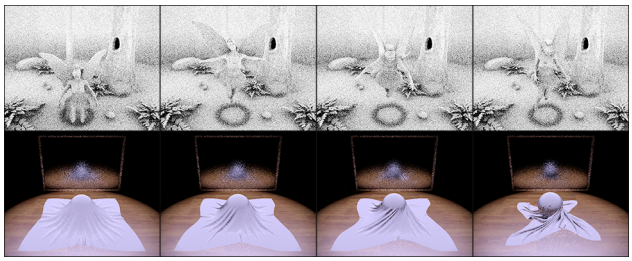


Figure 10: *Interactive examples demonstrating object deformation in the Fairy Forest (top) and the Cloth-ball (bottom) scenes at 4spp per frame. Average construction/tracing times per frame: 6/89ms (top) and 3/97ms (bottom).*

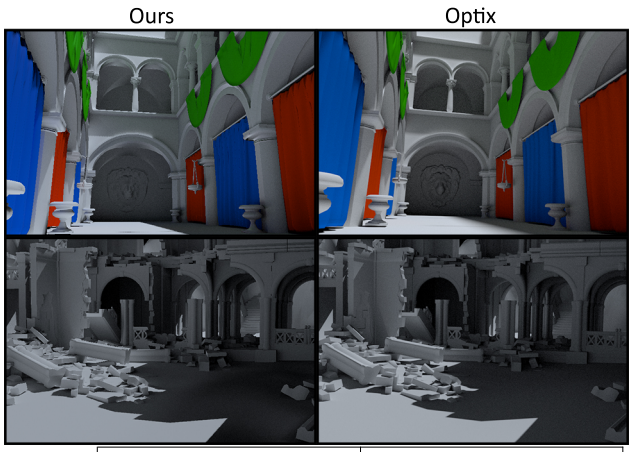
Comparison with GPGPU ray tracing. Figure 11 shows a comprehensive comparison between our method and a simple but optimized GPGPU path tracer implemented using NVIDIA OptiX ray tracing engine [Parker et al. 2010], tested with both SBVH and TRBVH acceleration structures. Due to inconsistencies in the lighting models between the GPGPU implementation and our method, the scenes were stripped of any textures, lit with a single point light and shaded with a diffuse-only model.

This test is critical in demonstrating the tradeoff between ray tracing performance and acceleration structure construction speed. Under the assumption of fully dynamic environments, the construction process potentially occurs in every frame buffer update. The images show that our method exhibits comparable quality with respect to GPGPU path tracing, within the limits of the rasterization sampling rate. In terms of performance, a global acceleration structure is naturally faster with respect to tracing rays but takes considerably longer to construct than our method, showing a clear benefit from adopting a rasterization-based approach for applications with dynamic geometry changes (e.g. modelling software).

Limitations. Since rasterization is based on discretized samples at specific locations, the information available is limited by the viewport resolution and the pixel samples. Geometry parallel to the view directions is not captured, even with the use of conservative rasterization or multisampling approaches. This leads to rays passing between fragments of sparsely sampled triangles. An example of this can be observed in Figure 8, where some of the water reflection rays miss the yellow/green shades and hit the sky dome instead. The problem of sparse rasterization of oblique geometry can be alleviated via triangle tesselation and point sampling (e.g. as in Nalbach et al. [2014]).

Table 1: *1- and 2- indirect bounce path tracing performance evaluation on scenes with increasing depth complexity.*

Scene	Bunny	Dead Nature	Pool	Smithy
Memory (MB)	128	197	241	209
Fragments	2.4M	4.9M	6.6M	9.4M
Avg/max layers	2/17	3/30	4/20	6/49
Construction (ms)	4	12	9	20
	1 indirect bounce tracing			AO (3m range)
Avg layers/views	2/2	3/1	4/2	6/2
Empty space efficiency	98%	89%	44%	77%
1spp / skipping. off (ms)	21/121	37/85	89/141	35/77
	2 indirect bounce tracing			
Avg layers/views	2/2	3/2	4/4	-
Empty space efficiency	98%	86%	47%	-
1spp / skipping. off (ms)	54/284	96/189	245/372	-



	Construction (ms)			1spp Tracing (ms)		
Scene	Ours	SBVH	TRBVH	Ours	SBVH	TRBVH
Sponza	**10**	2183	148 (15×)	36	**17** (2.1×)	18
Ruins	**18**	4103	1414 (78×)	27	**24** (1.1×)	25

Figure 11: *1-indirect bounce quality comparison of our method (left) with an NVIDIA Optix path tracing implementation (right) on the Sponza (top) and Ruins (bottom) scenes at 1000spp. Our method achieves comparable quality results with significantly lower construction times.*

Furthermore, minor quality issues arise from the quantization of the geometric information and the non-conservative DDA ray traversal algorithm. This can be mitigated by assuming that each fragment is a frustum-shaped voxel of a certain thickness [McGuire and Mara 2014]. However this approximation adds a relatively small view dependency. Finally, it should be noted that while our method does not distinguish between the type of environments compared to a global acceleration structure, our overall performance is obviously resolution-dependent due to its sequential marching nature, increasing linearly both the memory budget and rendering times. To improve the latter, the hierarchical-z approach to screen-space ray tracing [Uludag 2014] was also implemented by computing a mipmapped representation of the depth bounds texture. However, since this method requires frequent lod changes and accurate snapping of rays at pixel boundaries, a fixed number of sparse samples is mainly preferred when speed over quality is desired.

4.3 Analysis of multifragment alternatives

Performance. Figure 12 presents an extensive experimental performance evaluation of our approach (DLLB) against competing A-buffer variants in single view configuration when moving from low-to-high depth complexity scenes. Experimenting in multiple views proved unnecessary since the computational cost is proportional. The average/max depth complexity of the three scenes is $3/29$, $7/26$ and $16/96$ respectively. We performed 20 random-access per-pixel operations by artificially generating random rays. If a hit was detected, the stored color value was fetched from the Data buffer. We do not include a detailed comparison of A-buffer construction times, as a small difference between variants was observed. For a fair comparison against our method, we implemented our decoupled scheme and empty-space skipping optimizations on several competing A-buffer techniques, including a double linked lists A-buffer method (DLL).

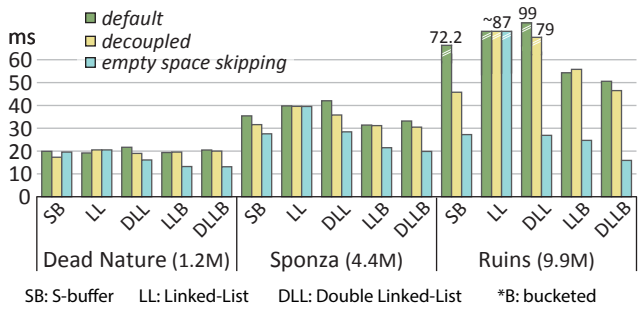


Figure 12: *Ray tracing performance evaluation of single view A-buffer variants with and without decoupled shading data and empty space skipping on scenes with different fragment complexity at 800^2 resolution. Numbers in parentheses denote fragment count.*

While decoupling can be adopted by all cases, gaining on average 15% improved construction times, per-pixel fragment queries and empty-space skipping depend on each variant's memory layout. The SB [Vasilakis and Fudos 2012] allocates fragments using continuous memory segments and traversal can be performed by either linear or binary search based on the number of generated fragments. Despite its logarithmic complexity, binary search is mostly preferred for traversing large per-pixel fragment sequences (> 16). On the other hand, linked-lists are limited to linear search but can take advantage of bucketing for fast fragment bucket skipping. Culling outside the depth boundaries is available in all methods, except LL [Yang et al. 2010], where the early skip on the far side is not available (e.g. r_d in Fig. 5).

The results showed an average increase for each scene in the decoupled versions by 5, 8 and 17% respectively as well as a further increase of 21, 30 and 210% by enabling the empty space skipping. We can further make the following observations: (i) bucket-based methods exhibit much better performance compared to the rest ones due to the fragment bucket skipping, (ii) bi-directional traversal in conjunction with early space skipping outperforms all other approaches and finally (iii) the average performance improvement is exponential with respect to fragment complexity.

Memory. Table 2 presents memory requirements (in bytes) for a wide range of methods that implement a decoupled-based A-buffer behavior. We observe that SB, LL, and DLL outperform LLB and DLLB due to their bucket-free construction nature. Note that our method requires slightly more storage per pixel m_p (B tail pointers) as well as per fragment m_f (previous node pointer) than its predecessor, the bucketed linked-list A-buffer (LLB) [Vasilakis and Fudos 2013].

Table 2: *Memory formulation of decoupled A-buffer versions. n_p is the number of fragments per pixel, w_j, h_j are the dimensions of the j-th view and $data_f$ is the per-fragment size of the shading data.*

Memory	SB	LL	DLL	LLB	DLLB
$m_p : \forall p \in v_j$	8	4	8	$8+4B$	$8+8B$
$m_f : \forall f \in p$	8	8	12	8	12
	$+ \text{size}\{data_f\}$				
Total	$\sum_{j=0}^{6} w_j h_j \cdot (m_p + n_p \cdot m_f)$				

Discussion. Considering completeness, we have investigated shifting bucket boundaries in order to achieve uniform fragment distribution inside bins. In this case, however, the ray-bucket intersection computation switches from $O(1)$ to $O(B)$ resulting in a noticeable reduction of the empty space skipping efficiency.

Regarding storage, our implementation suffers from potentially large and possible wasted memory demands due to its strategy to allocate data nodes for fragments that may not be hit. Note that it is not mandatory to generate a complete A-buffer for the main camera view, since this information is already present in the cubemap as well. However, since the resolution of the cubemap faces can be scaled down, high frequency effects exploiting the high-resolution main view will not be captured in the best possible detail.

5 Conclusions

We have presented a complete multifragment solution for interactive ray tracing. Our method reduces view dependencies significantly as it captures information on the entire scene without the need for additional data structures, supports large and dynamic environments and is able to effectively capture near and distant geometry. This allows the implementation of ray tracing-based algorithms, such as path tracing and ambient occlusion, without being restricted by any surface type, BSDFs or to individual phenomena. A wide spectrum of testing scenarios has been explored illustrating the high-quality images generated using this work. While our pipeline is independent of the underlying multifragment technique, our analysis has demonstrated the performance superiority of our A-buffer variant compared to prior solutions. Note that further directions may be explored for tackling the two main limitations of our framework: the sparse rasterization of oblique geometry and the extra memory consumption which is currently required.

Acknowledgements

The Cloth-ball simulation and Fairy Forest animations were obtained from the UNC Dynamic Scene Benchmarks and The University of Utah 3D Animation Repository, respectively. The Sponza model was downloaded from Morgan McGuire's Computer Graphics Archive. The Smithy model is included in "The Blacksmith" Environments' package, available from Unity Technologies. The Ruins scene was downloaded from AUEB Computer Graphics Group model collection.

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: ARISTEIA II-GLIDE (grant no.3712).

References

BAUER, F., KNUTH, M., KUIJPER, A., AND BENDER, J. 2013. Screen-space ambient occlusion using A-buffer techniques. In *International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics 2013)*, 140–147.

BITTNER, J., AND MEISTER, D. 2015. T-SAH: Animation optimized bounding volume hierarchies. *Computer Graphics Forum 34*, 2, 527–536.

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum 30*, 7, 1921–1930.

DE ROUSIERS, C., BOUSSEAU, A., SUBR, K., HOLZSCHUCH, N., AND RAMAMOORTHI, R. 2011. Real-time rough refraction. In *Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '11, 111–118.

GANESTAM, P., AND DOGGETT, M. 2015. Real-time multiply recursive reflections and refractions using hybrid rendering. *The Visual Computer 31*, 10, 1395–1403.

HU, W., HUANG, Y., ZHANG, F., YUAN, G., AND LI, W. 2014. Ray tracing via GPU rasterization. *The Visual Computer 30*, 6-8, 697–706.

MARA, M., MCGUIRE, M., NOWROUZEZAHRAI, D., AND LUEBKE, D. 2014. Fast global illumination approximations on deep G-buffers. Tech. Rep. NVR-2014-001, NVIDIA Corporation.

MCGUIRE, M., AND LUEBKE, D. 2009. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 77–89.

MCGUIRE, M., AND MARA, M. 2014. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT) 3*, 4 (December), 73–85.

MITTRING, M. 2007. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, 97–121.

NALBACH, O., RITSCHEL, T., AND SEIDEL, H.-P. 2014. Deep screen space. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '14, 79–86.

NIESSNER, M., SCHÄFER, H., AND STAMMINGER, M. 2010. Fast indirect illumination using layered depth images. *The Visual Computer 26*, 6-8, 679–686.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: A general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July), 66:1–66:13.

ULUDAG, Y. 2014. Hi-z screen-space cone-traced reflections. In *GPU Pro 5*, W. Engel, Ed. CRC Press, 149–192.

VARDIS, K., PAPAIOANNOU, G., AND GAITATZES, A. 2013. Multi-view ambient occlusion with importance sampling. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '13, 111–118.

VASILAKIS, A. A., AND FUDOS, I. 2012. S-buffer: Sparsity-aware multi-fragment rendering. In *Proceedings of Eurographics 2012 Short Papers*, EG '12, 101–104.

VASILAKIS, A.-A., AND FUDOS, I. 2013. Depth-fighting aware methods for multifragment rendering. *IEEE Transactions on Visualization and Computer Graphics 19*, 6, 967–977.

WIDMER, S., PAJAK, D., SCHULZ, A., PULLI, K., KAUTZ, J., GOESELE, M., AND LUEBKE, D. 2015. An adaptive acceleration structure for screen-space ray tracing. In *Proceedings of the 7th Conference on High-Performance Graphics*, ACM, New York, NY, USA, HPG '15, 67–76.

YANG, J. C., HENSLEY, J., GRUN, H., AND THIBIEROZ, N. 2010. Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum 29*, 4, 1297–1304.

ZIRR, T., REHFELD, H., AND DACHSBACHER, C. 2014. Object-order ray tracing for fully dynamic scenes. In *GPU Pro 5*, W. Engel, Ed. CRC Press, 419–438.