High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

DIRT: Deferred Image-based Ray Tracing

K. Vardis[†1] and A. A. Vasilakis[‡1,2] and G. Papaioannou[§1]

[1]Department of Informatics, Athens University of Economics & Business, Greece
[2]Information Technologies Institute, Centre for Research & Technology Hellas, Greece

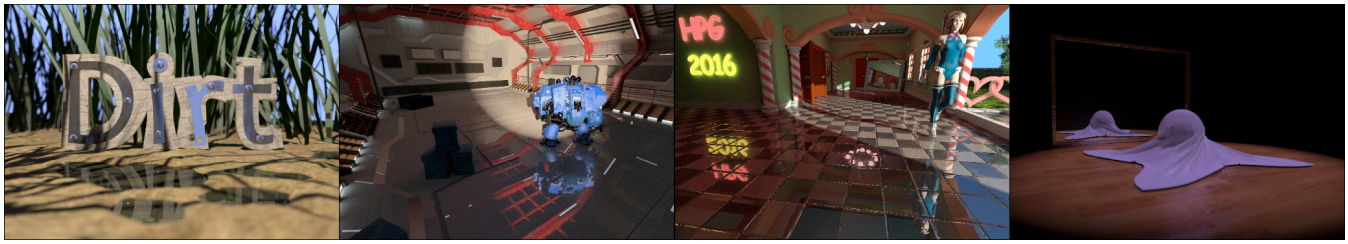


Figure 1: *Our deferred architecture is able to produce high quality results based on image-space ray tracing while maintaining low construction times and memory requirements, making it applicable to environments containing arbitrary complexity and motion. The last two insets demonstrate non-rigid animation.*

Abstract

We introduce a novel approach to image-space ray tracing ideally suited for the photorealistic synthesis of fully dynamic environments at interactive frame rates. Our method, designed entirely on the rasterization pipeline, alters the acceleration data structure construction from a per-fragment to a per-primitive basis in order to simultaneously support three important, generally conflicting in prior art, objectives: fast construction times, analytic intersection tests and reduced memory requirements. In every frame, our algorithm operates in two stages: A compact representation of the scene geometry is built based on primitive linked-lists, followed by a traversal step that decouples the ray-primitive intersection tests from the illumination calculations; a process inspired by deferred rendering and the path integral formulation of light transport. Efficient empty space skipping is achieved by exploiting several culling optimizations both in xy- and z-space, such as pixel frustum clipping, depth subdivision and lossless buffer down-scaling. An extensive experimental study is finally offered showing that our method advances the area of image-based ray tracing under the constraints posed by arbitrarily complex and animated scenarios.

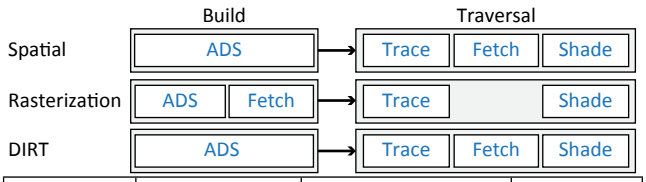
Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Ray tracing is widely considered as the most preferred method for accurately and robustly simulating global illumination phenomena. However, interactive rendering of fully dynamic environments is still an open problem due to the various constraints involved in the process. Typically, renderers involving ray tracing consist of two stages, an acceleration data structure (ADS) construction stage for speeding up the ray-object intersection calculations and a ray traversal loop, as shown in Figure 2. Throughout the years, numerous ADSs have been proposed improving more or less some of the key ray tracing characteristics: construction time, ray traversal, memory requirements and parallelization. An ADS can be classified either as *spatial*- or *rasterization*-based, depending on the approach taken for spatial subdivision.

Spatial ADS methods provide an approximative primitive-prioritized ordering to optimize ray intersection queries by exploiting either spatial subdivisions (grids, octrees, kd-trees), hierarchical clustering of bounding volume representations (BVH),

† kvardis@aueb.gr
‡ abasilak@{aueb,iti}.gr
§ gepap@aueb.gr



Method	Performance		Memory	Quality
	Build	*Traversal*	*Build*	*Traversal*
Spatial	Slow	**Fast**	**Primitives**	**Accurate**
Rasterization	**Fast**	Average	Fragments	Approximate
DIRT	**Fast**	Average	**Tile-based primitives**	**Accurate**

Figure 2: *Tracing pipelines overview. Our approach effectively combines the advantages of rasterization- (fast construction times) and spatial-based methods (analytic intersection tests) in a GPU- and memory-friendly context using a deferred rendering pipeline.*

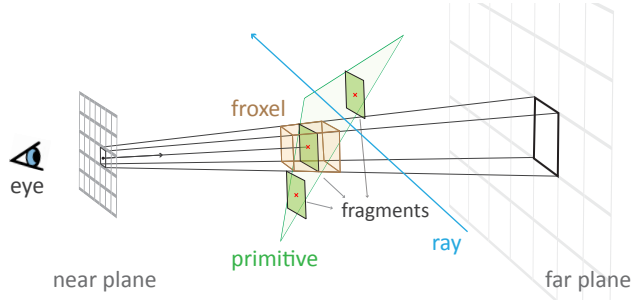


Figure 3: *Rasterization-based methods fail to capture most ray-object collisions due to their sparse geometry discretization. While replacing fragments with frustum-shaped voxels (froxels) can improve hit-ratio, it cannot guarantee accurate intersection behavior.*

or a combination of both strategies [Hav00]. These methods achieve high quality results along with efficient traversal times, but are limited to work with mostly static environments, where the resource-demanding construction stage is performed infrequently. Furthermore, interactivity becomes an issue in cases where geometry is potentially modified (tessellated or deformed) in an unexpected way in every frame since the ADS has to be dynamically updated, or even worse, entirely rebuilt from scratch [WMG*09].

Conversely, rasterization ADS methods, which operate mainly in the *image* domain [MM14, WKP*15, VVP16] or, at a lesser degree, jointly in a discrete *volume* domain [HHZ*14], are able to achieve real-time construction times by exploiting the hardware rasterization pipeline. While they can elegantly support dynamic scenes, they are, in general, prone to three major issues. First, for image-space methods, the captured information is sampled in a view-dependent uniform grid, which is potentially a sub-optimal acceleration structure for efficient ray traversal. Second, fragment-based approaches result in poor sampling of oblique geometry, leading to rays passing between fragments and subsequently intersecting with the wrong ones. This is illustrated in Figure 3. As a result, the estimated radiance is approximate. Last but not least, a potentially large and possibly wasted amount of storage is allocated due to their strategy to fetch the shading properties of each incoming fragment (or voxel), regardless if they are a part of the illumination computations or not; a memory issue that, in this paper, we refer to as *over-fetching*.

In this work, we attempt to bridge the gap between these approaches, by unifying their strengths and lifting their limitations (see table in Fig. 2). We introduce *Deferred Image-based Ray Tracing* (DIRT), a generic solution for screen-space ray tracing, able to simulate environments of arbitrary complexity in an accurate, memory- and GPU-friendly manner. Opposite to prior approaches, we apply the deferred tracing scheme of spatial-based methods in a rasterization-based ray tracing framework. This is achieved by disassociating the ADS with the shading data, which are only required for illumination calculations. This modification makes also feasible the conversion of the ADS construction from a per-fragment to a per-primitive basis, hence, improving: (i) *quality*: ray-triangle intersections can analytically be performed in screen-space, (ii) *memory*: lighting attributes are only fetched for primitives that are intersected by a ray and (iii) *performance*: build and traversal times are reduced since the ADS can be created in a compressed representation and the excessive fragment sorting stage during construction is not required.

Our ADS is initially built by performing multilayer rendering in a cubemap configuration inspired by the work of Vardis et al. [VVP16]. Contrary to fragment-based storage, the entire information is captured on multiple linked lists of pixel-clipped primitives. A coarser representation of this structure is also explored through the use of conservative rasterization, providing notable improvements in storage cost and rendering times without sacrificing the final quality. During traversal, and for each light bounce, rays are concurrently traced by efficiently skipping empty space regions both in image and depth space [Ulu14, VVP16]. Concerning quality, analytic intersection tests are achieved by adapting screen-space ray tracing [MM14] to our primitive-based data structure. A deferred pass is subsequently performed, only for the pixels that contain intersected primitives, gathering material and intersection properties in an auxiliary shading buffer. Finally, a resolve pass is responsible for computing the final illumination based on the available shading information.

To summarize, the main contributions of this work are:

- A novel deferred approach to image-space ray tracing ideally suited for the efficient rendering of arbitrary animated environments by explicitly using the rasterization pipeline.
- An analytic solution for screen-space ray tracing against a primitive buffer including several optimizations for the early termination of rays, such as primitive-based hierarchical traversal, bucket storage and lossless buffer down-scaling.

The paper is organized as follows: Section 2 includes a detailed summary of prior art. Sections 3 and 4 present the outline and implementation details of our architecture respectively. Section 5 reports on the efficiency and robustness of our method and discusses limitations and finally, Section 6 offers conclusions and future research directions.

2. Prior Art

Spatial ADS methods. While for non-interactive applications, where geometry is fixed [VHB15], the ADS choice can easily be determined solely based on its traversal stage complexity (build stage can be seen as a preprocessing offline step), new issues and challenges rise when attempting to support dynamic geometry. We refer readers to a comprehensive survey by Wald et al. [WMG*09] that sketches the general problem environment as well as covers the competing goals that even a present ray tracer has to face. Though a large variety of different refitting and updating strategies have been introduced for accommodating new geometry to the previous frame's ADS, most of them suffer from excessive ADS deterioration and require the animation sequence to be either restricted to coherent motion types and fixed connectivity or known a priori [Gar09, KIS*12, BM15]. Thus, the support of real-time animations, dynamically generated from interactive manipulation tools or streamed from a shared distributed virtual environment, boils down to the simple solution of completely rebuilding the ADS for each incoming frame [Kar12]. Despite the impressive build performance speedup of recent approaches, with the most popular being the hierarchical linear BVH (HLBVH) [PL10, GPM11] and the treelet-based BVH (TRBVH) [KA13], the construction process remains computationally intensive for arbitrarily complex environments.

With interactive applications in mind, uniform grids are able to achieve low construction times through regular subdivision of the environment. While easy to implement, the lack of empty space skipping available in hierarchical data structures can significantly increase traversal times, especially for incoherent rays. Therefore, they are mainly applicable to scenes with uniform distribution of primitives. Second level adaptive subdivision [KBS11] can speed up traversal on scenes with moderately non-uniform geometry distribution, while perspective grids [GN12] improve the coherence on primary and shadow rays.

Rasterization ADS methods. By relying entirely on the rasterization pipeline, a user-centered regular grid, such as the *deep G-buffers* (DGB) generalized by Mara et al. [MMNL14], can be considered as one of the fastest ADS to build. Although the limited uniform discrete sampling rate of DGB can be mitigated by replacing each fragment sample with a fixed-size frustum-shaped voxel (a.k.a. *froxel*) [MM14], inaccurate hits and misses cannot be totally avoided as illustrated in Figure 3. Several variants of DGB have been utilized for screen-space ray tracing throughout the years: G-buffer [SKS11], depth-peeling [MMNL14, MM14], *k*-buffer [WKP*15], cubemap G-buffer [GD15], three orthographically projected A-buffers [HHZ*14] and cubemap A-buffer [VVP16], listed here in a low-to-high build time, memory consumption and result quality. Moreover, a binary voxelization of the entire scene is often exploited to quickly identify a general hit location followed by a traversal step to locate the intersected fragment [HHZ*14]. In contrast to DGB, which is always available as part of the rasterization pipeline, a voxelization structure requires additional effort for the generation and processing of the underlying data.

Considering ray traversal in screen-space, linear 3D ray marching by Sousa et al. [SKS11] is the most widely-used method in the games industry. A hit is identified if the projected point, sampled from the marched ray in 3D space, lies behind the depth value stored in the G-buffer. McGuire and Mara [MM14] addressed under- and over-sampling limitations of this method by ensuring perspective-correct interpolation of the traced ray. An early-z skipping strategy accelerated this process by representing the ADS in a hierarchical fashion [Ulu14]. Several simple multilayer and multiview methods sacrifice performance [HHZ*14, MMNL14, WKP*15] to counter the significant information loss from inside and outside the viewing frustum when a single depth layer is used. To this end, Vardis et al. [VVP16] introduced a cubemap-based A-buffer variant with Z-interval buckets, enabling the tracing of rays in multiple layers and off-screen directions at interactive framerates.

Last but not least, these methods suffer more or less from over-fetching, resulting at a waste of storage and high possibility of fragment overflows. While a multitude of GPU-accelerated geometry-based structures offer decoupled deferred shading [BH13, SD15], none of them consider handling multiple layers. Finally, a deferred scheme with analytic intersection tests has been recently proposed [ZRD14], but requires an undetermined number of geometry passes and is limited to single bounce indirect illumination. To the best of our knowledge, DIRT can be considered as the first practical implementation of an accurate image-space ray tracing system, supporting global illumination effects in fully-dynamic scenarios of arbitrary depth complexity.

3. DIRT Architecture Overview

Akin to any ray tracing pipeline, our image-based algorithm operates in two broad stages: the *Build* stage, where geometry primitives are recorded in image-space data structures and the *Traversal* stage, where arbitrary rays are traced and the light transport is resolved in a breadth-first, iterative manner. By using geometric primitives instead of sampled fragment data, analytic tests can be performed, resulting in accurate image generation. Similar to Guntury et al. [GN12], we also exploit per-view perspective grids. However in our case, cells are irregularly sized in the depth dimension, based on the pixel's depth bounds, and hierarchically organized in the image domain, enabling early space skipping that significantly improves traversal in arbitrary environments.

To compute the contribution of each event, when a ray path is traced through a virtual environment, we use the well-known three-point light transport formulation and only allocate storage for the previous, current and next hit points encountered. This

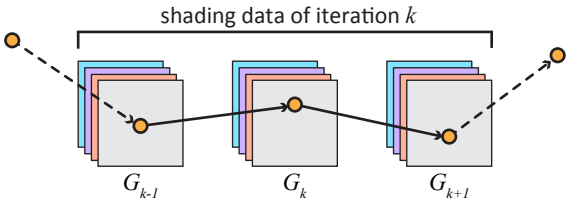


Figure 4: *To compute the illumination for point G_k, our shading structure contains information only for the three points contributing to the current event iteration k.*

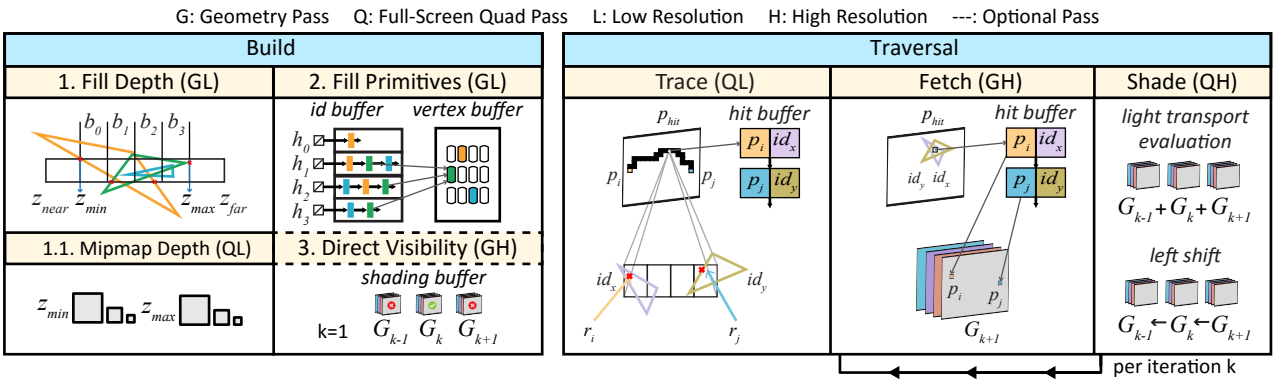


Figure 5: *A detailed and illustrative diagram of the DIRT architecture stages. (left) Build: Geometry is clipped (i) to compute the mipmapped depth bounds and, in a subsequent pass, (ii) to perform primitive occupancy discretization. (iii) Optionally, a deferred pass is performed to resolve direct visibility. (right) Traversal: Rays (r_i, r_j) are analytically traced in screen-space (starting from p_i, p_j) storing intersection attributes at the hit pixel location (p_{hit}). A geometry pass is subsequently performed at the masked pixels (p_{hit}) to fetch the shading properties of all hit primitives (id_x, id_y). Finally, the illumination contribution at path node (iteration) k is computed and hit point data are shifted in preparation for the next event ($k+1$). Note that several passes can be performed in lower resolution via conservation rasterization.*

way, we overcome the over-fetching issues of prior image-based approaches. Instead of storing fragment-based shading data for the entire environment before the tracing operation, we only retain minimal information about the primitives registered in the image buffers that is required exclusively for analytic ray-primitive intersection tests. After the valid hit point of each ray has been determined, shading information is updated *only once* for each event (see Fig. 4). For multiple events, this update operation involves shifting the currently stored shading attributes of the three path points and appending the newly discovered hit attributes.

Build Stage. The construction of the ADS captures the geometric information of the entire scene by rasterizing the scene in a multi-view setup. Two main and one optional steps are required. First, a *Fill & Mipmap Depth* step is employed, which stores mipmapped per-pixel minimum and maximum depth values based on the incoming primitives. This operation is required for two reasons: (i) screen-space ray tracing can be performed in a hierarchical manner, significantly reducing the cost of screen-space traversal and (ii) uniform depth subdivision can be exploited, allowing for efficient empty space skipping in the depth dimension. Next, a *Fill Primitives* pass captures the detailed geometric information by storing vertex information and primitive indices. Optionally, a *Direct Visibility* pass can be executed in order to cache surface data for direct lighting calculations, when camera shading effects such as depth of field are not enabled. Note that in order to accommodate dynamic environments, the Build stage occurs in every frame. However, for the multiple view setup, if the geometry is static, only the (trivial) Direct Visibility pass needs to be executed per frame, as the captured image buffers already contain all scene primitives. The entire construction process is explained in detail in Section 4.1.

Traversal Stage. This stage is executed in an iterative manner, requiring three passes for each path event: a *Trace*, *Fetch* and *Resolve* pass. The key idea here is that during each event k, per-pixel material information for the three required points (G_{k-1}, G_k, G_{k+1}) is stored in a shading buffer instead of the ADS (see Fig. 4), significantly reducing the memory requirements. Rays are generated based on the current point G_k, while any identified hits are fetched through a rasterization process and stored at G_{k+1}. At the end of each event, a left shift operation is performed to set the identified hit as the starting location of the next event (see Fig. 5 - Shade).

Initially, in the *Trace* pass, rays are generated based on either the cached data from the *Direct Visibility* pass or the camera lens. Tracing is performed hierarchically in screen space on the mipmapped per-pixel depth bounds. When a depth-based ray intersection is found, the primitive linked lists associated with the corresponding pixel are traversed until the closest ray-primitive collision is located. At each bounce, all intersections with a particular pixel from multiple rays are stored in an auxiliary buffer (the *hit buffer* in Fig. 5 - Trace), in the form of a linked-list associated with the hit pixel's location. In the next pass, called the *Fetch* pass, each stored hit is retrieved, rasterized and the resulting attributes are stored in the shading buffer. Finally, the *Shade* pass accumulates the lighting contribution at these points based on the properties stored in the shading buffer, according to the three-point form of the light transport equation, and prepares its contents for the next path event (iteration). Further details regarding the Traversal Stage are discussed in Section 4.2.

Data structures. Our deferred approach requires six structures in total, three for the ADS construction and another three for traversal. The ADS consists of (i) a *depth bounds texture*, storing the per-pixel min-max depth information created during the Fill & Mipmap Depth pass, (ii) a linked-list structure storing primitive indices, called the *id buffer* and (iii) a *vertex buffer* holding per-primitive vertex attribute information. The last two buffers are filled during the Fill Primitives pass. Note that if no new primitives are generated within the rasterization pipeline, e.g. through a tessellation or geometry shader, the vertex buffer already available as part of the rasterization process can be used instead. During tracing, intersection and shading information is associated to two new storage units: a *hit buffer*, containing ray-primitive hit information; and a *shading*

buffer, storing per-pixel material properties only for the three hit points (G_{k-1}, G_k, G_{k+1}) associated with the path event (iteration) k.

Acceleration Techniques. Similar to spatial ADS approaches, efficient exclusion of empty space is crucial for fast ray traversal. To this end, we apply several Z-culling optimizations in both the image plane and depth direction by adapting Hierarchical-Z (Hi-Z) traversal [Ulu14] and uniform depth subdivision [VF13] in our primitive-based pipeline. This is accomplished through primitive clipping operations against each pixel's frustum boundaries during the Build stage. A notable improvement in the efficiency of Hi-Z is also obtained by downscaling the ADS through conservative rasterization, where each group of shading pixels is represented as a *tile* in the ADS, reducing the total image-space steps required during ray marching in the Trace pass. Last but not least, a noticeable performance optimization is achieved by a pixel rejection scheme; all pixels with no associated intersections are marked in a *mask texture* and are subsequently discarded during the Fetch pass, where shading takes place. The entire pipeline is illustrated in Figure 5 and discussed in further detail in the following section.

4. Method Details

4.1. Build Stage

Overview. Briefly, the first stage of DIRT samples the scene primitives in a user-centric manner as follows: First, 6 views $v_j, j = 0\ldots5$ arranged in a cubemap configuration are created covering the entire scene extents, using the multiview setup proposed by Vardis et al. [VVP16]. Second, the ADS is efficiently constructed by performing two geometry passes for all views: the first one is responsible for computing the depth bounds texture that is subsequently mipmapped to retain the aggregate min/max depth values per texel. The second pass fills the vertex and id buffers. Finally, an optional geometry pass, executed only for the primary camera view, is employed to initialize the shading buffer with the direct lighting results. This pass can be omitted when camera shading effects such as depth-of-field are present, increasing the Traversal stage steps by one.

ADS downscaling. To reduce the redundant data generated at neighboring pixels corresponding to the same rasterized primitive, a more compact lossless representation of the ADS is constructed via conservation rasterization. This mechanism, exposed as an OpenGL extension for the NVIDIA Maxwell architecture, allows rasterization to generate fragment samples for all pixels intersected by a primitive. Thus, the ADS can be constructed (Fill & Mipmap Depth and Fill Primitives passes) and traced (Trace pass) at a lower resolution $R_l = R_h/S$ by associating each high-resolution pixel p_h, that belongs to the framebuffer's resolution R_h, to a pixel tile p_l with size $S \times S$, $S = 2^m$, $m \in \mathbb{Z}_{\geq 0}$. While this data structure holds the same primitives as the high-resolution ADS due to conservative rasterization, a smaller number of fragments are generated. This is because although a tile S intersects more primitives, only a single copy of a pixel-clipped primitive is generated and stored (Sec. 4.1.2), reducing the memory requirements and construction times. However, due to the reduced portability of conservative rasterization, downscaling can be optionally omitted ($m = 0$).

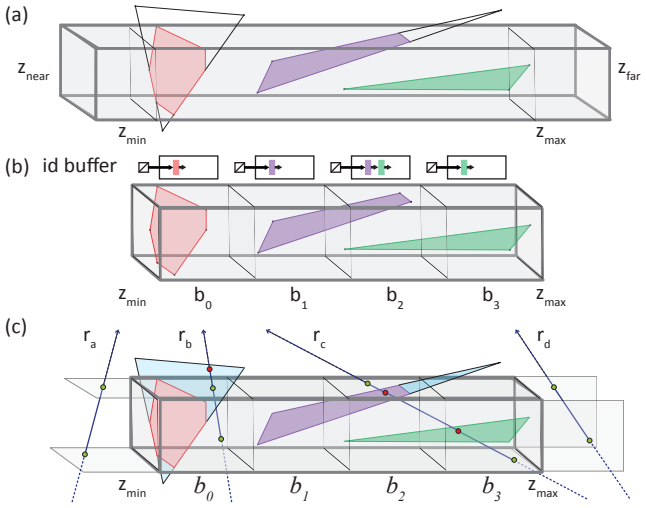


Figure 6: *Handling primitives in screen-space. Per-pixel primitive clipping is exploited for correct ADS building including (a) depth range computation and (b) multi-bucket placement. (c) Empty-space skipping is performed when either rays pass outside depth boundaries (r_a, r_d) or a valid hit is found in a bucket where traversal order is defined by the ray's direction (r_c). Note that intersections detected outside the pixel frustum are discarded (r_b).*

4.1.1. Depth Bounds Passes

Fill Depth. The first geometry pass is responsible for generating a depth bounds texture, which stores the depth extents $z_{min}(p_l)$, $z_{max}(p_l)$ of all primitives spanning each pixel p_l of every view. Since the ADS stores primitive ids instead of depth-ordered samples, correct calculation of depth extents is achieved by clipping primitives against each pixel's boundaries (see Fig. 6(a)). This operation is required for two reasons. First, uniform depth subdivision (bucketing) can be exploited, where primitives are assigned to multiple linked-lists spanning equally sized depth intervals, during the Fill Primitives Pass (Sec. 4.1.2). A primitive is assigned to all buckets its depth bounds overlap with (see Fig. 6(b)). Second, Hi-Z can be performed after constructing a mipmapped version of the depth bounds buffer (Sec. 4.2.1).

From an implementation point of view, a geometry shader outputs each primitive pr_k to the fragment shader, which clips it against the pixel's frustum planes, resulting in a new primitive pr_l. Then, the new primitive's depth extents update the respective pixel's depth bounds via atomic min/max operations. The geometry shader is required for emitting primitives to multiple views and is not required in a single-view implementation.

Mipmap Depth. The depth bounds texture is downscaled and filtered, independently for both the min and max values, into multiple levels d by performing a fast full-screen quad rendering pass. The modest increase in texture memory usage ($\simeq 1.5\times$) provides a significant ray tracing speedup.

4.1.2. Fill Primitives Pass

Once the mipmapped depth bounds texture is computed, a second geometry pass is employed to construct the core structure of our ADS: the vertex buffer, a linear array that contains the three per-vertex attribute structures $\mathbf{v}_i(id_k), i = 0,1,2$ of each incoming primitive pr_k with unique identifier id_k; as well as the id buffer, an unsorted multiple linked-list structure that contains the unique identifiers for all primitives rasterized in every pixel p_l of every view v_j. Specifically, at the geometry shader invocation, the incoming vertex information (such as positions, normals, texture coordinates, etc.) are stored sequentially in the vertex-buffer. The value of id_k can be based either on the unique primitive identifier provided by the corresponding API for single draw call operations or on a global atomic counter, when multiple draw calls are required. In the following fragment shader invocation, each pixel's p_l depth range $z_{range}(p_l) = z_{max}(p_l) - z_{min}(p_l)$ is split into B uniform sub-intervals $b_0, \ldots, b_{B-1}$. Then, each incoming rasterized primitive is clipped against pixel's p_l frustum boundaries and the resulting (clipped) primitive pr_l is atomically added to all buckets in the range $[b_{min}, b_{max}]$ overlapping its depth extents (see Fig. 6(b)) :

$$b_{min}(p_l, pr_l) = \lfloor B \cdot (z_{min}(pr_l) - z_{min}(p_l))/z_{range}(p_l) \rfloor \quad (1a)$$

$$b_{max}(p_l, pr_l) = \lfloor B \cdot (z_{max}(pr_l) - z_{min}(p_l))/z_{range}(p_l) \rfloor \quad (1b)$$

4.1.3. Direct Visibility Pass

When simple perspective primary rays are generated, a quick rendering pass is performed first, storing the material properties of the tracing starting point for each pixel p_h in the camera view shading buffer G_1. The primary ray hits are created via direct rendering and stored in the shading buffer. In our implementation, this buffer write operation exploits a semaphore-based spin-lock mechanism since the shading buffer contains structures and is therefore not a regular frame buffer, where conventional fragment depth testing operations can be performed. After this step, we also initiate a Shade pass to quickly compute the direct illumination and then continue iteratively the Traversal stage as discussed in the following section.

4.2. Traversal Stage

Overview. This stage is executed in an iterative manner, computing the illumination contribution of each scattering event per iteration $k \geq 1$. Briefly, it consists of three different passes:

- Rays are traced in screen space by traversing the downscaled ADS until the closest ray-primitive intersection is analytically found and captured in the hit buffer. The hit buffer stores intersection data for *all rays* intersecting a particular pixel during iteration k. This information includes the primitive ID, intersection position and barycentric coordinates. Since the number of registered hits is unknown a priori, a per-pixel linked-list structure was preferred as a storage container.
- The shading properties for each identified hit are fetched and stored at the shading buffer by performing a geometry rendering pass. Conceptually, the shading buffer's contents correspond to three points for each pixel: G_{k-1}, holding the previous point's position, G_k, pointing to the current event's shading attributes (essentially the starting ray location); and finally, G_{k+1}, storing the new intersected point's fetched material properties. A pixel-rejection mechanism is also exploited to prevent unnecessary invocations of non-intersected pixels.
- A full-screen quad rendering pass is finally employed to compute the shading of the current event by evaluating the three-point light transport formulation at the shading buffer. To support a complete path tracing implementation, an additional *operators* texture is employed during the Trace and Shade passes, storing the updated probability of path segment k as well as the cumulative transport operators respectively.

4.2.1. Trace Pass

Screen-space ray tracing. This step starts with the generation of a new ray r_h for each pixel p_h, based on the contents of shading buffer G_1. G_1 stores either the camera position or the attributes stored in the direct visibility pass during the Build stage, depending on whether direct rendering is used for the primary rays or not. Rays are clipped against the viewing frustum and subsequently traced via ray marching up to the screen-space projection of their clipped endpoint until the closest intersection is found in the id buffer. If no valid hit is found within the current view, the process is repeated for each subsequent view the ray intersects. Readers are referred to [MM14, VVP16] for further details on screen-space ray tracing in one or more views respectively. Generally, the image-space ray tracing performance relies heavily on how well the empty space regions of the scene can be avoided, by exploiting the available min-max depth maps. Efficient empty space skipping is achieved by moving in hierarchical steps on the image plane and by avoiding uniform sub-intervals in the depth domain.

Hi-Z tracing. Single-layer hierarchical screen-space tracing is commonly carried out by switching between tiles of different sizes, belonging to different mip levels $d \in [0, d_h]$ of the high-resolution (R_h) Z-buffer [Ulu14]. We revise accordingly this operation with respect to our compact primitive-based pipeline with two modifications: (i) reaching the lowest mip level does not define an intersection, but initiates a lookup procedure in the id buffer and (ii) the lowest mip level d_l depends on the resolution of the ADS buffer: $d_l \in (0, d_h)$. The last modification essentially, trades traversal overhead in the image-space with larger lists in the depth domain. Specifically, starting at mip level $d = d_h$, the ray's eye-space Z coordinates are compared against the tile's depth extents. For each successful intersection, the mip level is decreased and iteratively refined until reaching the lowest value d_l, where primitive traversal in the id buffer is initiated (see Alg. 1, Ln $1-6$). Otherwise, the current tile is skipped and the mip level is increased for the next tracing iteration (see rays r_a, r_d at Fig. 6(c)).

Bucket tracing. Once the process enters the primitive traversal phase, analytic ray-primitive intersection tests are performed between the ray and the primitives stored in each bucket of the id buffer. To ensure the closest valid hit, the following operations take place. First, we identify the range of pixel bucket IDs $b_{range}(p_l, r_h) = [b_{min}(p_l, r_h), b_{max}(p_l, r_h)]$ intersected by the ray via equations 1a and 1b. Rays that move towards the near plane are checked in back-to-front bucket order, while rays towards the far plane are checked in the opposite order. Note that while this operation is similar to Vardis et al. [VVP16], it is applied here in

order to skip any remaining buckets during the traversal operation rather than to ensure correct multi-hit behavior of ray-fragment collisions. Second, we linearly iterate through all primitives of each intersected bucket in order to find the closest ray-primitive collision. For each successful intersection test between a ray r_h and a primitive pr_k, we keep the hit if both of the following conditions are met:

- The hit point lies within the candidate pixel tile $p_l^{d_l}$. We check this by projecting the hit location in the image space of the tile's associated view (see ray r_b at Fig. 6(c)).
- The intersection distance t_k is the shortest one acquired up to this point ($t_k < t_{hit}$).

For any accepted hit, we maintain the primitive ID id_k, the barycentric coordinates br_k and intersection position pos_k. If at least one valid hit is found in one bucket, the remaining buckets are omitted (see ray r_c at Fig. 6(c)). The details of this method are shown in Algorithm 1 (Ln $7-22$).

Final Storage. After testing all primitives in the intersected bucket for ray-triangle collisions, the closest hit location pos_{hit} is projected to the high-resolution hit buffer location p_{hit}, where a new hit record is atomically inserted in the linked list at that location. Each record contains the hit data information acquired during tracing, the view v_j in which the intersection occurred and the shading buffer pixel location p_h. The latter is the pixel location in which the tracing started ($G_k[p_h]$) and also the location where the fetched data will be stored at the next pass ($G_{k+1}[p_h]$). The view information is required since lighting calculations are performed in eye space. To reduce the overhead of unnecessary fragment invocations of non-intersected pixels, a pixel-rejection scheme is employed, by flagging the pixel p_{hit} as occupied, through the use of a mask texture. In terms of implementation this can be either a depth or a stencil texture, essentially used as an early culling testing mechanism.

Algorithm 1 int *analytic_ssrt* (Pixel p, Ray r, Mip-level d)

```
1:        ▷ (1) Hi-Z tracing
2:  if z_min(r^d) > z_max(p^d) or z_max(r^d) < z_min(p^d) then
3:      return no_hit;                          ▷ skip complete pixel tile
4:  else if d > d_l then
5:      return invalid_level;                   ▷ large mip-level, decrease & retry
6:  end if
7:        ▷ (2) Bucket tracing
8:  b_range = [b_min(p,r), b_max(p,r)];
9:  for each bucket b_i ∈ b_range do
10:     for each primitive pr_k ∈ b_i with id id_k do
11:         {t_k, br_k} ← hit(r, pr_k);         ▷ ray-primitive intersection test
12:         pos_k ← orig(r) + t_k · dir(r);
13:         p_k ← project(pos_k);
14:         if t_k ∈ (0, t_hit) and p_k = p^d then   ▷ new best hit found
15:             {t_hit, id_hit, br_hit, pos_hit} ← {t_k, id_k, br_k, pos_k};
16:         end if
17:     end for
18:     if hit found then
19:         break;                              ▷ skip remaining buckets
20:     end if
21: end for
22: return {pos_hit, br_hit, id_hit};           ▷ return closest hit info
```

4.2.2. Fetch Pass

A full geometry rendering pass is responsible for retrieving the shading information for all intersected primitives identified in the previous pass. For each incoming rasterized primitive, we search the linked-list at that hit buffer location for a match between the stored primitive ID and the incoming one. For every successful comparison during each iteration, we compute the vertex attributes using the stored barycentric coordinates, thus avoiding the attribute extrapolation problem of conservative rasterization. The primitive's shading data are then retrieved through common texture fetching operations. Finally, the primitive's shading data are stored in the shading buffer, at G_{k+1}.

4.2.3. Shade Pass

The last step of the iteration initiates a quick full-screen quad rendering pass, which computes the lighting contribution of the current scattering event k. Using the shading buffer contents and the operators texture, the three-point light transport is evaluated and stored in the framebuffer. Afterwards, a left shift operation on the shading-buffer is performed, essentially setting the current intersection point as the starting ray position for the next iteration: $G_{k-1} = G_k, G_k = G_{k+1}$.

4.2.4. Notes on conservative rasterization

If conservative rasterization is not available, the entire pipeline is slightly modified: (i) the comparison during the Fetch pass requires both primitive and view information and (ii) barycentric coordinates are not required since attribute interpolation is based entirely on the rasterizer.

5. Results and Discussion

We have implemented DIRT solely on the OpenGL rasterization pipeline, which is compatible with common graphics engines. Due to the design philosophy of the algorithm, a full path tracing solution was implemented and evaluated. We offer a detailed evaluation in terms of quality (Sec. 5.1), performance (Sec. 5.2) and memory (Sec. 5.3) against: (i) the most recent rasterization-based ray tracing DGB method by Vardis et al. [VVP16] since it offers the best quality to date but also suffers from the highest memory consumption and over-fetching issues and (ii) two fast and highly optimized spatial-based data structures, HLBVH [PL10, GPM11] and TRBVH [KA13] on a GPGPU path tracer implemented with NVIDIA OptiX [PBD*10]. We have run our tests on an NVIDIA GTX 980 Ti GPU and all images were rendered at a resolution of $R_h = 1024 \times 512$, $R_l = R_h$ unless specified otherwise. The node size in bytes for each of our buffers was $size(node_{id}) = 8$ for the ADS and $size(node_{hit}) = size(node_{sb}) = 32$ for the hit and shading buffers respectively. Note that the size of the shading buffer is specific to our path tracing implementation, containing packed material attributes such as diffuse and specular reflectivity, opacity and emissive parameters. All quality results were rendered at ≈ 1000 spp, where each sample corresponds to a complete tracing path, including both the direct and indirect illumination events. The Dirt, Candy, Hangar and Cloth scenes are presented in Figure 1 while the Bunny scene is shown in Figures 7 and 9.

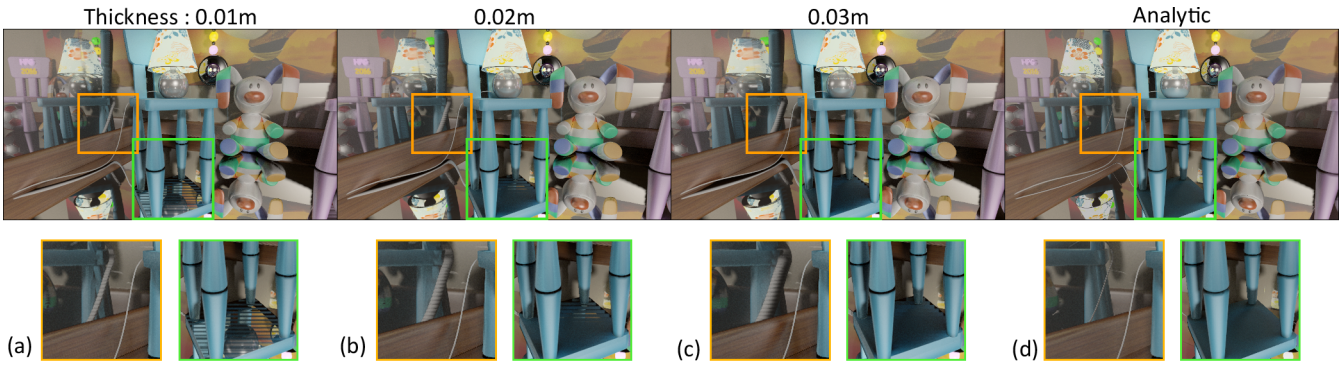


Figure 7: *Quality comparison (2-indirect bounce lighting) between fragment-based collisions (a,b,c) and our analytic approach (d). High frequency phenomena cannot be accurately captured by manually adjusting the view-dependent thickness parameter. This is mostly visible in the reflections of the lamp power cord (orange inset) and the cyan chair (green inset) at the floor and walls, which appear stripped or extruded.*

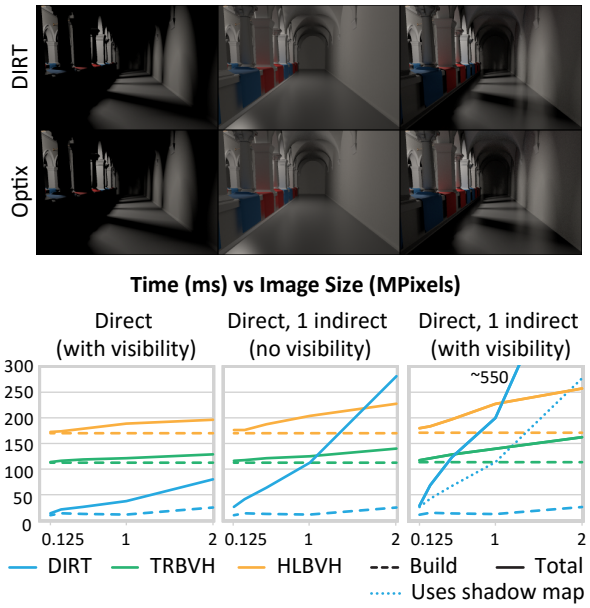


Figure 8: *Comparison with two spatial-based data structures in the Sponza Atrium, where construction occurs on every frame.*

5.1. Quality Evaluation

Fragment-based ray tracing. Tracing methods, which approximate the environment based on discrete samples (fragments), are able to produce plausible results in most typical scenarios. However, they are susceptible to view dependencies due to rays passing through sparsely sampled geometry. While this can be mitigated by assuming each sample is a frustum-shaped voxel of non-zero *thickness*, inaccurate misses cannot be totally avoided. Even worse, the thickness parameter is view-dependent and requires manual adjustment. As a result, these methods are approximate and their error is mostly visible when high frequency phenomena are present. Figure 7 demonstrates this on a scene, where perfect reflections are dominant. Rays which would intersect with the seat of the cyan chair end up passing through (a). Increasing the thickness eventually resolves this issue but results in extruding the reflected objects (b, c), which is especially noticeable when compared with the correct result captured by our method (d - see for instance the power cord).

GPGPU ray tracing. Figure 8 (top) provides a quality comparison of our method against two spatial-based data structures, based on NVIDIA OptiX. We show results with primary and shadow rays (left), primary and secondary rays without shadow rays (middle) and primary and secondary rays with shadow rays (right). We are able to achieve identical results with the reference images, demonstrating that our method can also be employed for quality renderings, a fact that is also validated through all experiments shown in the paper.

5.2. Performance Evaluation

Figure 10 (bottom) presents performance results under different depth subdivision settings and tile sizes as well as the corresponding cost of the Fetch and Shade passes for three test environments. Since these two depend only on the resolution R_h, they are also reported separately (right). Note that we do not provide measurements for different path lengths ($k > 1$) as our tests showed that the increase is linear in closed environments and, as expected, sub-linear in open environments, due to excessive ray misses.

With respect to the number of buckets, we observed an exponential benefit in traversal times while construction is increased linearly by a slope of $\approx 2\%$ due to primitives overlapping more buckets. Regarding changes in the tile size, the performance benefit depends on the number of generated primitives during construction and on the tile *density* of the generated primitives during traversal. Therefore, an increase in the tile size provides notable traversal improvements in scenes where the geometry tessellation is low or medium. Environments containing many finely tessellated objects, however, such as the foliage in the Dirt scene, can benefit from a larger tile size mainly on high resolutions.

Impact of ray coherence. Table 1 presents the performance on four scenes with different types of rays, such as visibility, perfect reflection, highly glossy and pure diffuse. These measurements were taken in practical scenarios by changing the material properties of the objects in each scene and rendered with our prototype path tracing implementation by spawning rays. Note that we performed our tests on the first path event since the cost of primary rays is minimal due to rasterization. Visibility rays were spawned towards a predetermined location outside the scene's extents and terminated at the first intersection, specular rays towards the reflection direction, glossy rays through BSDF importance sampling and finally, diffuse rays using cosine hemisphere sampling. The performance drop observed when moving from specular to glossy and, finally, diffuse rays is justified by the increasing incoherence caused by rough surfaces. The latter is also an indication of the improvement we can achieve with further research in the area.

Table 1: *Mrays/sec of DIRT under different ray types for the first path event.*

Scene/Ray	Primitives	Visibility	Specular	Glossy	Diffuse
Cloth	93k	65.5	26.7	26.6	15.2
Bunny	156k	40.3	24.9	14.58	11.6
Hangar	266k	39.9	14.7	9.6	9.6
Candy	420k	24.9	14.9	12.0	9.6

Fragment-based ray tracing. Table 2 presents a detailed comparison against fragment-based ray tracing in terms of both performance and memory consumption, where the thickness parameter was adjusted to produce the optimal quality result for each scene. For a fair performance comparison against DIRT, conservative rasterization was disabled ($R_l = R_h$) and the number of buckets B was the same as in the fragment-based approach in order to produce a similar number of fragments and clipped primitives. We observed that: (i) the memory was reduced by $7-21\%$ while the generated fragments (n_p) was increased by $\approx 5\%$ due to our multi-bucket placement, (ii) the construction times dropped by 15% since no fragment sorting occurs and finally (iii) tracing times dropped by an average of 15% in scenes where the average distances between objects were larger than the thickness parameter. The latter did not occur in the Bunny scene where many rays terminated prematurely, due to the thickness extrusion (see also Fig. 7). Enabling conservative rasterization, which is required by our method, doubles in most cases the tracing times due to the significantly larger number of fragments generated, which consequently results in an irregular increase of the primitive lists in cases of oblique geometry.

Table 2: *Performance and memory comparison of a DGB-based method and DIRT.*

	[VVP16] / DIRT			
Scene	Fragments (M)	Memory (MB)	Build (ms)	Traversal (ms)
Bunny	3/3.1	161/150	8/6.5	28/53
Hangar	4.5/4.7	201/184	13.7/11	57/47.9
Candy	8.9/10.2	317/250	18/16.8	75/62

GPGPU ray tracing. Figure 8 (bottom) provides a performance comparison against two spatial-based data structures, TRBVH and HLBVH, provided by the NVIDIA OptiX. We report times on the rendered stills shown at the top, with one primary and one shadow ray (left), a primary and a secondary ray (middle) and one primary, one secondary and two shadow rays (right). Note that the tile size is increased gradually with the resolution from 1x1 to 8x8 pixels. Our results show that our method can maintain much lower construction times (dashed lines) even at high resolutions. In cases of incoherent rays, here the secondary camera path segments and their shadow rays, we observe a steeper increase in traversal times which diminishes the benefit of low construction as the resolution increases.

Single-view ray tracing. Our approach can be also used in a single-view configuration, where image accuracy is traded for speed by skipping out-of-view geometry and respective view traversals. Performance can be further improved by replacing visibility calculations with shadow maps (see Fig. 9).

5.3. Memory Evaluation

Table 3 presents the total memory requirements for all structures used in our pipeline, where $c \in [0,B]$ is the average intersected buckets per primitive and n_p is the number of clipped primitive samples. We omit the storage cost of the vertex buffer since it is already required as part of the vertex buffer creation in a common rasterization pipeline. Also note that the mipmap generation of the depth bounds texture slightly increases the memory consumption. By following a deferred tracing strategy, our method reduces significantly the memory required by the large amount of fragment information generated, when the entire scene shading information is captured, instead. This is achieved by (i) bounding the tracing information ($node_{sb}$) and disassociating it from the scene's geometric complexity ($node_{id}$) and (ii) down-scaling the ADS.

Table 3: *Per-pixel memory formulation of the DIRT architecture.*

Memory	Requirements (in bytes)	Unit
depth bounds texture	$11 \approx 8 \times 1.33$	$\forall p_l \in v_j$
id buffer	$4 \cdot B + c \cdot n_p \cdot size(node_{id})$	
hit buffer	$4 + size(node_{hit})$	$\forall p_h$
shading buffer	$3 \cdot size(node_{sb})$	
mask texture	2	
$size(node_{id}) = 8,\ size(node_{hit}) = size(node_{sb}) = 32$		

Figure 9: *A single-view setup in conjunction with shadow maps can provide view-dependent, high quality rendering at higher frame rates. Construction/Traversal times (per spp): 5ms/37ms (left) and 9ms/59ms (right) respectively at 1MPixel, $R_l = 512 \times 512$.*

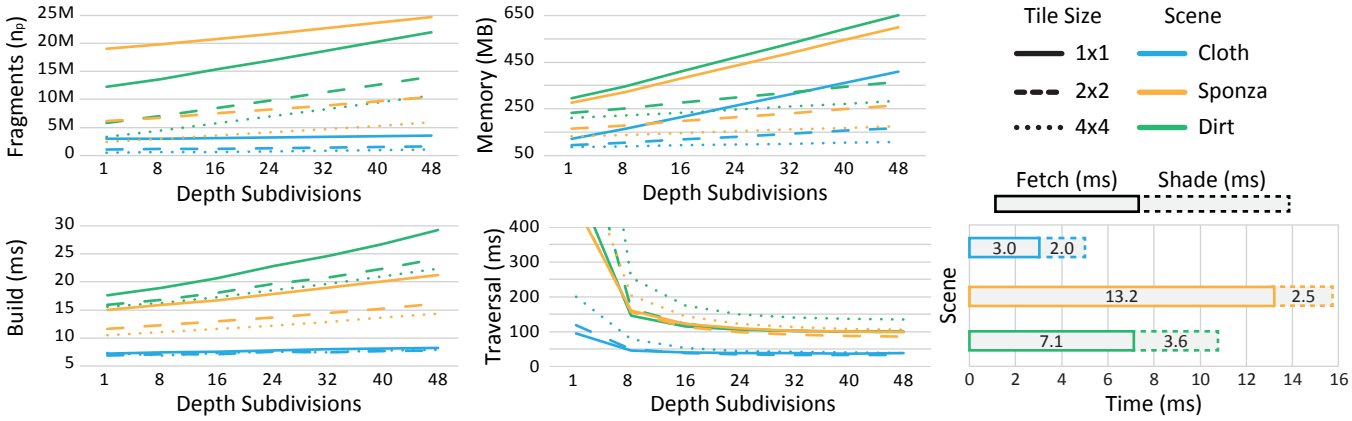


Figure 10: *Detailed measurements regarding memory (top) and performance (bottom) with variable number of depth subdivisions (B) and tile sizes on scenes of different fragment complexity. Separate timings for the Fetch and Shade passes are also shown on the right.*

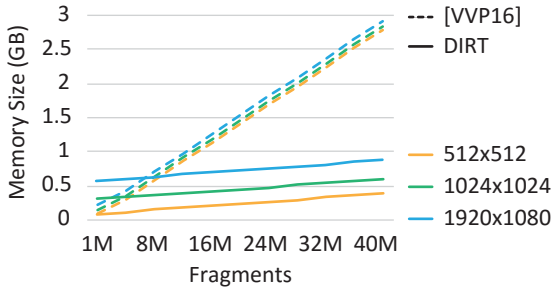


Figure 11: *Correlation of ADS memory cost of [VVP16] and DIRT methods under increasing screen resolutions ($R_h = R_l$, $B = 1$).*

Impact of down-scaling. Figure 10 shows the memory gain when the tile size is increased from 1×1 pixels to a tile of 2×2 and 4×4 pixels respectively on three test cases of different fragment complexity. For a bucket size $B > 8$, which is a practical minimum for our approach to achieve relatively decent performance, we observed that the fragment generation is reduced initially by 56% when moving to a 2×2 tile and a further 37% to a tile size of 4×4 pixels, while the memory required was 50% and 32% respectively. Note that the fragment generation drop is justified by the reduction in the number of clipped primitive samples n_p, when larger tile sizes are used. On the other hand, increasing the number of buckets by 8 corresponds leads to a memory increase of 15%, 8% and 4% for each tile setting. The main observations here are: (i) an increase in the tile size provides similar memory benefits on all scenarios regardless of the fragment complexity, (ii) an increase in the number of depth subdivisions consequently increases the memory required, since more buckets are being intersected per primitive and more head pointers are required for the linked lists, and finally (iii) a combination of tile size 2×2 and depth subdivisions $B \in [20, 40]$ provides a good overall setting in terms of both performance and memory gain.

Impact of decoupling. While the over-fetching issue is present in all variants based on DGB, memory problems are mainly evident in cases where the entire scene geometry is captured [HHZ*14, VVP16]. In the following, we do not compare our method with variants where only a few layers are handled [MM14, WKP*15]) since, while their memory overhead is small, the geometric information and resulting quality is very approximate. Figure 11 illustrates the memory allocation superiority of our method when compared to a recent fragment-based ADS [VVP16]. We were unable to provide a correct comparison against the method by Hu et al. [HHZ*14] since it was not discussed by the authors. However, we assume that the total cost would be similar with the currently compared method due to the three orthographic A-buffers used for capturing the entire scene. Both methods allocate the id and shading data as part of the ADS construction resulting in a huge and wasteful memory demand. While DIRT has an initial overhead due to the fixed memory cost of the decoupled hit and shading buffers, it exhibits a near constant relation with the fragment generation as opposed to a linear one in DGB. This results in reduced memory requirements in most typical cases since the number of generated fragments is usually much higher than the number of pixels. This is even more noticeable when conservative rasterization is employed, since the number generated fragments increases significantly.

5.4. Limitations

While our method is able to achieve high quality results, there are several issues that require further research. The most notable improvement can be achieved in the traversal times, mainly in incoherent rays, when compared to spatial-based data structures. This is an expected outcome of our shallow acceleration structure (one-deep regular subdivision), the overhead caused by the increase of our linked lists due to conservative rasterization and the fact that we do not explore any coherence strategies. Also, an overhead is caused in both construction times and memory consumption due to the duplicate primitive information currently present in multiple buckets of the same tile, which is currently a requirement

for accurate traversal when depth subdivision is involved. We aim to target these issues in the future by exploiting adaptive depth subdivision techniques, clustering algorithms and strategies for ray coherence [MBJ*15].

6. Conclusions

In this work, we have demonstrated that a strong continuum between rasterization and ray-tracing approaches is actually possible. Considering the issues when ray tracing is performed over fully dynamic scenes with undefined motion pattern and unknown topology modification behavior, we have introduced DIRT, a deferred rendering solution to image-based ray tracing. In the heart of our framework lies a novel ADS designed to simultaneously maintain three, commonly conflicting, criteria: (i) fast construction times via (conservative) rasterization in conjunction with (ii) analytic intersection tests using (clipped) primitives as storage nodes and (iii) reduced memory consumption. A wide spectrum of testing scenarios has been explored showcasing the low GPU resources consumed as well as the compelling graphics effects generated when DIRT is demonstrated under a physically-based path tracing platform. Despite the tremendous progress on the landscape of ray tracing field, we believe that our approach provides a novel insight at a key area with renewed research interest, where high potential for software and hardware improvements is feasible in the near future.

7. Acknowledgments

The Cloth-ball simulation and Sponza model were obtained from the UNC Dynamic Scene Benchmarks and Morgan McGuire's Computer Graphics Archive, respectively. The Marie Rose animation model, shown as part of the Candy scene, was downloaded from the Sketchfab website.

References

[BH13] BURNS C. A., HUNT W. A.: The Visibility Buffer: A cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (August 2013), 55–69. 3

[BM15] BITTNER J., MEISTER D.: T-SAH: Animation optimized bounding volume hierarchies. *Computer Graphics Forum 34*, 2 (2015), 527–536. 3

[Gar09] GARANZHA K.: The use of precomputed triangle clusters for accelerated ray tracing in dynamic scenes. In *Proceedings of the Twentieth Eurographics Conference on Rendering* (Aire-la-Ville, Switzerland, 2009), EGSR'09, Eurographics Association, pp. 1199–1206. 3

[GD15] GANESTAM P., DOGGETT M.: Real-time multiply recursive reflections and refractions using hybrid rendering. *The Visual Computer 31*, 10 (2015), 1395–1403. 3

[GN12] GUNTURY S., NARAYANAN P. J.: Ray tracing dynamic scenes on the GPU using grids. *IEEE Transactions on Visualization and Computer Graphics 18*, 1 (Jan 2012), 5–16. 3

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 59–64. 3, 7

[Hav00] HAVRAN V.: *Heuristic ray shooting algorithms*. PhD thesis, Faculty of Electr. Engineering, Czech Techn. University, Prague, 2000. 2

[HHZ*14] HU W., HUANG Y., ZHANG F., YUAN G., LI W.: Ray tracing via GPU rasterization. *The Visual Computer 30*, 6-8 (2014), 697–706. 2, 3, 10

[KA13] KARRAS T., AILA T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 89–99. 3, 7

[Kar12] KARRAS T.: Maximizing parallelism in the construction of BVHs, Octrees, and K-d trees. In *Proceedings of the 4th ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics* (Aire-la-Ville, Switzerland, 2012), HPG'12, EG Associat., pp. 33–37. 3

[KBS11] KALOJANOV J., BILLETER M., SLUSALLEK P.: Two-level grids for ray tracing on GPUs. *Computer Graphics Forum 30*, 2 (2011), 307–314. 3

[KIS*12] KOPTA D., IZE T., SPJUT J., BRUNVAND E., DAVIS A., KENSLER A.: Fast, effective BVH updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2012), ACM, pp. 197–204. 3

[MBJ*15] MATTAUSCH O., BITTNER J., JASPE A., GOBBETTI E., WIMMER M., PAJAROLA R.: CHC+RT: Coherent hierarchical culling for ray tracing. *Computer Graphics Forum 34*, 2 (2015), 537–548. 11

[MM14] MCGUIRE M., MARA M.: Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT) 3*, 4 (December 2014), 73–85. 2, 3, 6, 10

[MMNL14] MARA M., MCGUIRE M., NOWROUZEZAHRAI D., LUEBKE D.: *Fast Global Illumination Approximations on Deep G-Buffers*. Tech. Rep. NVR-2014-001, NVIDIA Corporation, 2014. 3

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: A general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July 2010), 66:1–66:13. 7

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, 2010), HPG '10, Eurographics Association, pp. 87–95. 3, 7

[SD15] SCHIED C., DACHSBACHER C.: Deferred attribute interpolation for memory-efficient deferred shading. In *Proceedings of the 7th Conference on High-Performance Graphics* (New York, NY, USA, 2015), HPG '15, ACM, pp. 43–49. 3

[SKS11] SOUSA T., KASYAN N., SCHULZ N.: Secrets of Cryengine 3 graphics technology. In *ACM SIGGRAPH Talks* (2011). 3

[Ulu14] ULUDAG Y.: Hi-Z screen-space cone-traced reflections. In *GPU Pro 5*, Engel W., (Ed.). CRC Press, 2014, pp. 149–192. 2, 3, 5, 6

[VF13] VASILAKIS A.-A., FUDOS I.: Depth-fighting aware methods for multifragment rendering. *IEEE Transactions on Visualization and Computer Graphics 19*, 6 (2013), 967–977. 5

[VHB15] VINKLER M., HAVRAN V., BITTNER J.: Performance comparison of bounding volume hierarchies and Kd-trees for GPU ray tracing. *Computer Graphics Forum* (2015). 3

[VVP16] VARDIS K., VASILAKIS A. A., PAPAIOANNOU G.: A multi-view and multilayer approach for interactive ray tracing. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2016), I3D '16, ACM, pp. 171–178. 2, 3, 5, 6, 7, 9, 10

[WKP*15] WIDMER S., KAUTZ J., PAJAK D., SCHULZ A., PULLI K., GOESELE M., LUEBKE D.: An adaptive acceleration structure for screen-space ray tracing. In *Proceedings of the 2015 Symposium on High Performance Graphics* (USA, 2015), ACM. 2, 3, 10

[WMG*09] WALD I., MARK W. R., GÃIJNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the art in ray tracing animated scenes. *Computer Graphics Forum 28*, 6 (2009), 1691–1722. 2, 3

[ZRD14] ZIRR T., REHFELD H., DACHSBACHER C.: Object-order ray tracing for fully dynamic scenes. In *GPU Pro 5*, Engel W., (Ed.). CRC Press, 2014, pp. 419–438. 3