An Adaptive Acceleration Structure for Screen-space Ray Tracing

S. Widmer[1,2] D. Pająk[1,3] A. Schulz[4] K. Pulli[1,3] J. Kautz[1] M. Goesele[2] D. Luebke[1]

[1]NVIDIA [2]Graduate School of Computational Engineering, TU Darmstadt [3]Light [4]TU Darmstadt

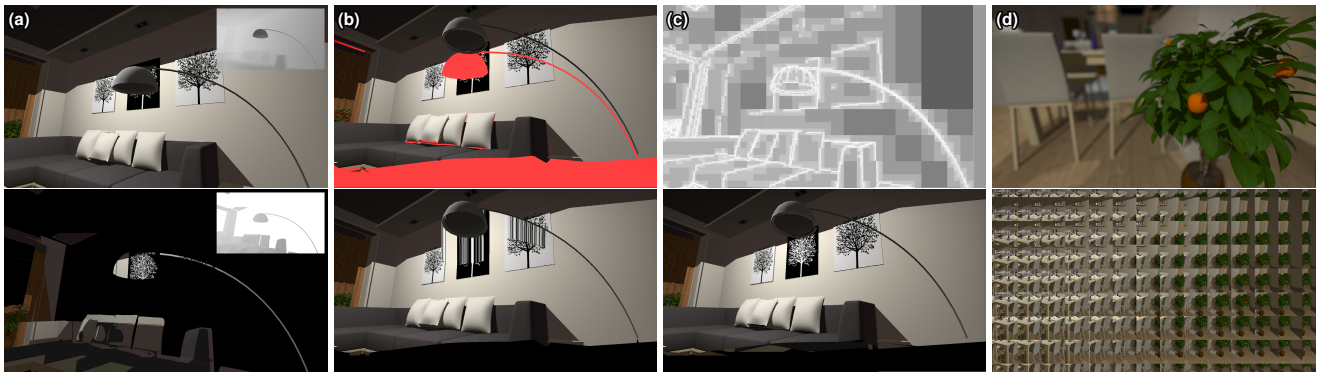

Figure 1: *We perform fast screen-space ray tracing through single- and multi-layered depth representations. Because we efficiently obtain valid hits from occluded geometry, our approach can address many problems of traditional screen-space methods. Examples:* ***(a)*** *Reference view color+depth buffers for the first (top) and second layer (bottom).* ***(b)*** *Large camera motion produces extensive disocclusions (top, red), which are difficult to inpaint by forward warping (bottom).* ***(c)*** *Our compressed depth representation lowers the required ray-AABB intersection count (top, black corresponds to 0 and white to 9 intersections) and allows for an efficient reprojection with only two depth+color layers (bottom).* ***(d)*** *Our ray-tracing approach generalizes well to other real-time applications, including depth-of-field rendering (top) and light-field rendering (bottom).*

Abstract

We propose an efficient acceleration structure for real-time screen-space ray tracing. The hybrid data structure represents the scene geometry by combining a bounding volume hierarchy with local planar approximations. This enables fast empty space skipping while tracing and yields exact intersection points for the planar approximation. In combination with an occlusion-aware ray traversal our algorithm is capable to quickly trace even multiple depth layers. Compared to prior work, our technique improves the accuracy of the results, is more general, and allows for advanced image transformations, as all pixels can cast rays to arbitrary directions. We demonstrate real-time performance for several applications, including depth-of-field rendering, stereo warping, and screen-space ray traced reflections.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: screen-space ray tracing, acceleration structure, GPU

1 Introduction

Many real-time rendering techniques operate in screen-space in order to be computationally efficient. This includes techniques for approximating realistic lighting, such as screen-space ambient occlusion [Mittring 2007], soft shadows [Guennebaud et al. 2006], global illumination effects [Ritschel et al. 2009; Mara et al. 2014], and camera effects such as depth-of-field (DoF) [Lee et al. 2009]. These screen-space techniques trade precision and quality for performance. They approximate algorithms that traditionally work by ray tracing 3D geometry.

In fact, some of those algorithms use screen-space ray tracing, or rather ray marching [Sousa et al. 2011]. Ray marching is attractive as no additional data structure needs to be built, but tracing rays for long distance becomes prohibitively expensive quickly. So precision is often sacrificed for performance by restricting the number of samples along the ray to reduce texture lookups, which can miss geometry – even when combined with a final binary-search refinement step. Classic ray marching methods, like DDA, are prone to over and under-sampling, unless perspective is accounted for [McGuire and Mara 2014]. Most screen-space ray tracing methods use only a single depth layer, as a naïve extension to multiple layers is costly [Mara et al. 2013].

In this paper, we address many of these shortcomings of screen-space ray tracing. Namely, we implement an efficient and scalable screen-space ray tracing algorithm that employs a dynamically created acceleration data structure, which enables efficient empty space skipping. If desired, our algorithm can trade off accuracy for speed and can efficiently handle multi-layered screen-space representations to yield higher quality. The construction of the acceleration structure is extremely fast and can easily be done on a per-frame basis allowing us to also handle dynamic content. Our method's efficiency makes it not only useful for screen-space effects such as depth-of-field, but also for reprojection tasks even into many views, as required for light field displays, that previously mapped very poorly to GPUs. We demonstrate our method in several classic applications (stereo warping, temporal upsampling, depth-of-field rendering, multi-view synthesis, and glossy as well as specular reflections). Our contributions are:

- A novel data structure that stores the depth buffer in a compressed format—a mixture of AABB and planar approxmations—that enables early ray traversal termination and leads to an improved performance. The approximation can be further tuned for specific applications (e.g., depth-of-field) providing a significant performance boost.
- A new algorithm for screen-space ray tracing that is particularly well suited for GPUs. In contrast to state-of-the-art methods (e.g., ray marching [McGuire and Mara 2014]) the ray traversal does not rely on a predefined or maximal number of steps and results in an accurate intersection point (assuming each pixel is planar). Our efficient occlusion handling allows tracing multiple depth layers of our acceleration structure without individually tracing each ray against each layer.
- A real-time implementation of thin-lens-based depth-of-field rendering. We extend previous work by introducing a secondary sampling stage (possible thanks to our compressed scene representation), which significantly reduces noise without sacrificing the defocus blur quality.
- A reprojection application that enables arbitrary existing content to be rendered on light-field displays with many views.
- Efficient multi-bounce specular and glossy reflections, utilizing our adaptive data structure on a (multi-layer) cube map representation of the scene.

2 Related work

2.1 Screen-space ray tracing and applications

In 1986 Fujimoto et al. [1986] proposed the 3D-DDA line traversal algorithm for quickly tracing rays through a regular grid or octree. It inspired the improved 3D-DDA line traversal algorithm by Amanatides and Woo [1987], which serves as the basis for many screen-space ray tracing methods [Sousa et al. 2011; Ganestam and Doggett 2014; McGuire and Mara 2014].

The work by Sousa et al. [2011] was the state-of-the-art for many years. It linearly ray marches a (reflection) ray in 3D, based on 3D-DDA, for a bounded distance. Each 3D point is reprojected into the frame buffer and classified as a hit if it lies behind the depth at the projected pixel. It does not do any space skipping, which was addressed by Ganestam et al. [2014], where an additional BVH is created. Employing a linear 3D-DDA traversal might lead to missed samples in screen-space. McGuire and Mara [2014] address this with a perspective 3D-DDA, ensuring no screen-space samples are skipped. In contrast, we build a screen-space acceleration structure on the fly that allows us to efficiently trace rays without the need for stepping along a line in small increments with 3D-DDA.

A wide range of applications and techniques use screen space ray tracing to simulate effects like ambient occlusion, view interpolation, or reflections. Most of those methods reuse shading information across frames to speed up computation, since this information (e.g., complex material evaluation) is expensive to recompute [Nehab et al. 2007; Sitthi-amorn et al. 2008]. Herzog et al. [2010] combined shading reuse and spatio-temporal upsampling with the focus on reduction of shading cost. In contrast, our method focuses on a general acceleration data structure that allows for fast reprojection.

Another typical screen-space application is depth-of-field. These methods often employ approximations and application-specific algorithms, such as approximate cone tracing [Lee et al. 2009], or bounding the ray footprint [Lee et al. 2010]. While our technique is more general, we show that we also achieve better performance than these methods. Yu et al. [2010] suggested to warp a frame buffer to create a full light field, which is then combined to create depth-of-field effects. They use forward warping and simply splat larger pixels into the target views to prevent holes. Our technique can also be used to create a light field, but is much more efficient, as the run-time is independent on the number of views generated.

2.2 Ray tracing data structures for GPUs

Many different data structures have been proposed for GPU-based ray tracing applications. Bounding volume hierarchies (BVH) have been used extensively on GPUs. Lauterbach et al. [2009] create LBVHs by linearizing primitives along a space filling curve, yielding near optimal hierarchies and good overall performance for both construction and traversal. This was improved upon with a hierarchical version [Pantaleoni and Luebke 2010] and work queues [Garanzha et al. 2011]. Rasterized bounding volume hierarchies (RBVH) [Novák and Dachsbacher 2012], where leafs contain height fields that are ray marched, allow for efficient but approximate ray casting. Similar to our method, it also allows one to trade level of detail for computational efficiency. However, RBVH are not geared towards screen-space ray tracing, as the construction of the data structure is too slow. Very recently, fast parallel construction of high-quality BVHs have been demonstrated on GPUs [Karras and Aila 2013], yielding about 90% of the ray tracing performance of offline methods. K-d trees can also be constructed on GPUs [Zhou et al. 2008], even including the surface area heuristic (SAH) [Wu et al. 2011]. However, construction is generally more costly than for a BVH.

Voxelized scene representations have been used for various applications in real-time rendering. Efficient sparse voxel octrees [Laine and Karras 2010] offer excellent ray casting performance, but can require non-negligible construction time and memory. Voxelized scene representations have been used extensively when high resolution and accuracy is less critical, for instance, in indirect illumination [Crassin et al. 2011].

Unlike these methods, our technique is geared specifically towards ray tracing through layered 2.5D height fields, exploiting their structure for considerable performance gains over standard ray tracing acceleration structures.

2.3 Ray tracing of relief and height fields on GPUs

Many techniques have been proposed since the early 1980's to render height fields, usually using ray tracing [Musgrave 1988; Cohen and Shaked 1993; Cohen-Or et al. 1996]. These methods generally differ in their choice of acceleration data structure. For instance, the early work by Cohen and Shaked [1993] uses a quadtree and is the original inspiration for our method. Using ray tracing to render height fields and reliefs has also become popular in GPU-based real-time rendering. Tracing rays by uniformly stepping through the height field in conjunction with a binary search is a common approach [Policarpo et al. 2005]; Newton iterations is another [Wyman 2005]. While these methods are simple to implement, they can lead to missed intersection points. This can be fixed through the use of safety zones [Donnelly 2005; Baboud and Decoret 2006], but at a higher computational cost and using precomputed data structures.

These methods only support a single layer, the work by Policarpo and Oliveira [2006] adds the ability to render layered height fieldsby packing four layers into a single texture in order to trace through them simultaneously, speeding up rendering. For our use cases, we argue that most of the time only the first layer is hit by a ray, and the other layers are rarely needed. We therefore only trace into deeper layers only if the first layer received no hit (for fewer than 1% of pixels).

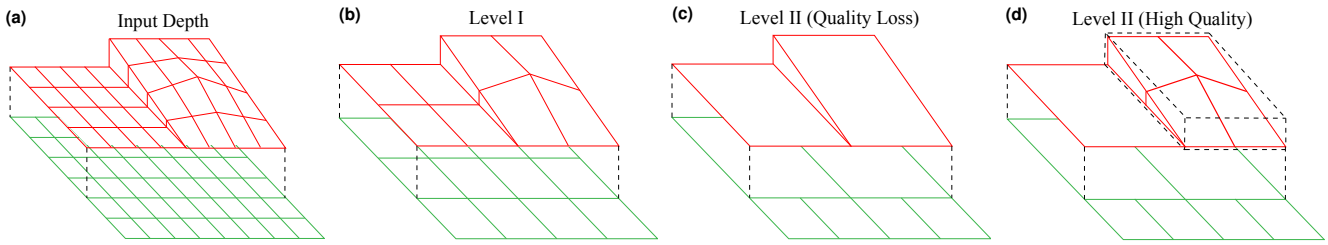

Figure 2: *A toy example of quad-tree construction.* ***(a)*** *We start the construction with two input depth layers (red/green) with per-pixel normal vectors (visualized via plane rotation here).* ***(b)*** *Each* 2×2 *set of adjacent and non-overlapping cells is analysed to generate a parent that best describes them. Depending on the compression setting, slightly misaligned children are replaced by a* ***(c)*** *single parent plane which then becomes a new leaf node (tree branch has been pruned) or a* ***(d)*** *AABB parent node that encompasses the children.*

Tevs et al. [2008] accurately render height fields by creating a min-max mipmap hierarchy over the depth map on the fly, which allows ray tracing with empty space skipping. This improves the pyramidal displacement mapping technique [Oh et al. 2006], sharing the min-max mipmap acceleration data structure with previous work [Kolb and Rezk-Salama 2005; Carr et al. 2006; Guennebaud et al. 2006]. While this data structure—it corresponds to a fully subdivided quad-tree—is attractive for height field rendering, it does not directly support discontinuities and multiple layers. Traversal requires looping to step into the correct hierarchy level, whereas our method uses simple bit patterns to yield the correct level.

The min-max mipmaps [Tevs et al. 2008] have also been used to render volumetric shadows [Chen et al. 2011], where epipolar rectification of the shadow map ensures that a ray traverses along a row, reducing ray traversal to a 1D-problem. Unfortunately, we cannot use this insight, as the construction is slow and the structure is only valid between a single reference view and one novel view, reducing applicability.

Like many of the methods cited here, our method is related to layered depth images (LDIs) [Shade et al. 1998]. Just like LDIs, our scene representation consists of possibly multiple layers of depth plus color and we also support reprojection of the scene. However, LDIs were geared exclusively toward scene reprojection, whereas our method is more general enabling several different applications. Furthermore, the original splatting-based LDI rendering technique was not very GPU-friendly, and has been superseded by much of the work cited in this section.

3 Data structure

In this section, we first introduce our data structure for representing multiple layers of scene geometry, and how it is constructed in real time. We then describe our approach for ray traversal in Sec. 4.

3.1 Compressed depth representation

We start by rendering the reference view into a set of color+depth buffers [Shade et al. 1998], which serves as an over-complete representation of the scene. This kind of rendering workload can be implemented efficiently via *depth peeling* [Mara et al. 2013; Lee et al. 2010; Policarpo and Oliveira 2006]. We decided to use a simple k-buffer algorithm, which turned out to be sufficient and fast enough. Our method, however, is not bound to any particular approach and will work with any depth peeling method.

Next, for every layer from this 2.5D stack, we compute a *quad-tree* ray traversal acceleration structure (see Fig. 2). Each node in the quad-tree stores either a 3D *axis-aligned bounding-box* (AABB) or, at leaf nodes, a 3D plane that represents the geometry. Note that since the quad-tree is built from frame buffer image data, the screen-space 3D AABBs actually correspond to frusta in world-space. The proposed data structure is related to the min-max pyramids [Guennebaud et al. 2006], in a sense that we use non-overlapping (in screen-space) bounding-volume hierarchies (BVH) to accelerate ray tracing through efficient empty space skipping (AABB misses). However, in contrast to this previous work, our quad-tree can be adaptively pruned, as the leaves represent the geometry by a single plane. This has two important consequences for ray traversal. First, since the plane nodes represent the underlying layer geometry, a ray-plane intersection test simultaneously determines if the ray hits *both* the node *and* the geometry. The bounds defined by such a plane are tighter than by the enclosing AABB, which makes skipping empty space during the traversal more efficient. This translates into significant performance gains due to reduction in both GPU memory bandwidth pressure and thread divergence. Furthermore, as the ray-plane intersection test results in an exact intersection point, we do not need a "refinement" stage, such as binary search [Policarpo and Oliveira 2006; Tevs et al. 2008], to remove artifacts. The second feature of our data structure is that we can use it directly to approximate and *compress* the screen-space geometry by controlling when and how a node's geometry is replaced with a *proxy*-plane. This allows us to trade off ray tracing precision for performance.

3.2 Bottom level generation

The quad-tree is built in a bottom-up fashion. The depth of the decomposition could go to $\lceil \log_2 \max(width, height) \rceil$ levels, but our implementation caps it to 9 levels. Both construction and ray tracing are done in a variant of NDC (*Normalized Device Coordinates*) space, where all potential scene points (x, y, z) are inside of a unit cube. The bottom level of the quad-tree is initialized directly from the depth buffer. Each pixel is represented as a tiny plane with a normal vector $\vec{N}$ and plane origin P_{origin}. In practice, we only store the Z coordinate of P_{origin}, as its 2D coordinates are known from the node position in the quad-tree. In addition to $\vec{N}$ and P_{origin}, we store a binary flag O, which indicates whether the plane is close to a depth discontinuity. We use this information during ray tracing to adaptively dilate the screen-space bounding-box to mask tiny cracks between neighbors with different orientations.

Normal vectors can be obtained through deferred rendering (via a *g-buffer*) or computed from depth data directly (see supplemental for details and pseudo code). All our examples use the latter approach, although g-buffer normals should produce slightly better quality.

3.3 Quad-tree generation and planar approximation

The remaining quad-tree levels are generated with our adaptive pruning algorithm demonstrated in Fig. 2 (see supplemental for pseudo code). The idea behind the algorithm is simple. For each output node we consider its 2×2 children nodes, and if they can be approximated *well enough* with a plane, we store the plane, oth-

Scene	Triangle count	Average construction time [*ms*]
SPONZA	227k	0.646 ± 0.042
LIVINGROOM	456k	0.636 ± 0.006
SANMIGUEL	6550k	0.657 ± 0.042

Table 1: *Average quad-tree construction and compression time changes with the output quad-tree node count and is almost invariant to the scene complexity. See Sec. 5 for configuration details.*

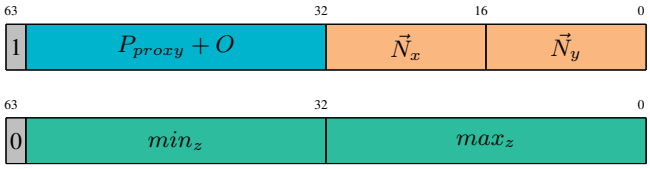

Figure 3: *Quad-tree 64-bit node layout. Top corresponds to the plane node (1x32-bit float + 2x16-bit float) and bottom (2x32-bit float) to the AABB node.*

erwise we define the node as a regular AABB and store its corresponding min/max Z values.

Successfully approximating a subtree by a plane node requires fulfilling three conditions: **(1)** all the children have to be plane nodes, **(2)** the maximum angular difference between proxy-plane and child plane normals has to be less than γ_{norm}, and **(3)** the maximum distance from child plane corners to the proxy-plane has to be less than γ_{dist}. Thresholds γ_{norm} and γ_{dist} can be fixed, or modulated adaptively to implement LOD-like functionality. If the children do not meet these conditions, we output a parent node as AABB that encompasses all of them.

An accurate planar approximation of children requires solving an optimization problem with 4 unknowns ($\vec{N}_{proxy}, P_{proxy}$), which is too slow for real-time applications. We simplify the problem by first estimating normal $\vec{N}_{proxy}$ as the average of child normals, and then finding the plane's Z coordinate P_{proxy} that minimizes the distance of child plane corners to the proxy plane with

$$\underset{P_{proxy}}{\arg\min} \sum_{i=0}^{3} \sum_{j=0}^{3} \left(\frac{P_{proxy} - p_{ij} \cdot \vec{N}_{proxy}}{\vec{v} \cdot \vec{N}_{proxy}} \right)^2 ,$$

where $\vec{v}$ is the view direction we optimize for and p_{ij} is the j^{th} corner of i^{th} child plane represented in the coordinate space of the proxy-plane. By default we set $\vec{v} = (0, 0, 1)$ to maximize the reconstruction quality of the quad-tree for the reference view. In Table 1 we demonstrate the performance of the solver on the GPU. The entire construction algorithm is very fast and does not depend on the geometric complexity of the underlying scene.

3.4 Quad-tree node format

Each quad-tree node can store either an AABB or a 3D-plane. We managed to reduce the node's memory footprint by fitting both structures in just 64 bits (see Fig. 3). Interpretation of the node data is based on the value of the most significant bit of the first word, which corresponds to the *sign* bit in single-precision floating-point number representation. As both min_z and $P_{proxy} + O$ are always known to be positive, we choose to flip the sign of $P_{proxy} + O$ so we can disambiguate whether the node stores a plane or AABB by simply inspecting the sign bit. When encoding the plane, we store two components of a normal vector in half-float precision and recover the third one with $\vec{N}_z \leftarrow \sqrt{1 - \vec{N}_x^2 - \vec{N}_y^2}$.

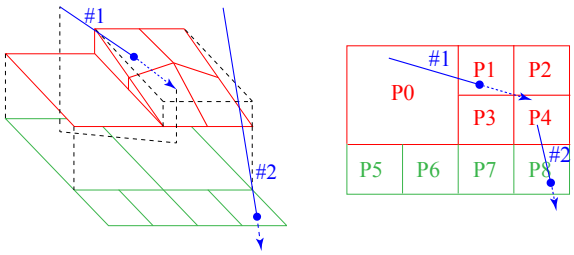

Figure 4: *A toy ray-tracing example. Using the data structure from Fig. 2d (right side shows XY-plane view) we cast two rays into the scene. Ray #1 misses leaf P0 but then hits the AABB node encompassing leaves P1–P4. Based on the AABB intersection point we derive the next intersection candidate (P1 leaf), which produces the final hit point. Ray #2 initially misses top layer completely (no intersection with P1–P4 AABB), but hits AABB encompassing P7–P8. The final hit point is computed via intersection with P8.*

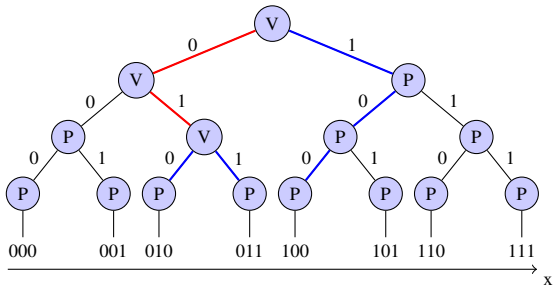

Figure 5: *A 1D-view of ray traversal. Nodes representing planes (P) and AABBs (V) are stored in a hierarchical data structure. Moving from node at position* 011 *to* 010 *is fast as both nodes share most of the path from the root (red edges). In contrast, moving from* 011 *to* 100 *requires going all the way up to the root. Our algorithm computes the position of the last common ancestor between two arbitrary nodes at a given quad-tree level without the use of stack or loops, making it particularly friendly to GPU implementations.*

4 Ray traversal

The core of our ray tracing method is depicted in Fig. 4 (see supplemental for pseudo-code). At a high level, the algorithm performs a classic quad-tree ray traversal [Cohen and Shaked 1993] with several application-specific customizations. For now we describe our method for a single layer and ignore disocclusions, which we detail in the next section.

First, the node type determines the intersection procedure, and we test either for an intersection with the plane or the AABB. Second, to reduce branching and thread divergence, we developed a method for efficient child selection in case of a node hit, and efficient successor selection in case of a node miss. Both functions are numerically stable and resistant to singularities. In case of a node hit, we compute the child position based on the parent node quadrant in which the intersection point landed. In case of a node miss, we generate the successor by evaluating which edge the ray hit when leaving the parent node (see Ray #1 case in Fig. 4).

Finally, similar to Frisken and Perry [2002], we observe that quad-tree node x- and y-coordinates Q_x and Q_y encode the traversal stack up to the root node. By simply right bit-shifting Q_x and Q_y by one, we can generate the parent coordinates of the current node. This property allowed us to reduce the number of intersection tests significantly. After finding the coordinates (Q_x^* and Q_y^*) for the successor at the same quad-tree level (e.g., P2 is the successor of P1 in Fig. 4), we could directly proceed to it, but this would provide inefficient fixed-step traversal similar to 2D DDA line-drawing

algorithms. After all, the direct successor or one of its ancestors might be missed, so to maximize the benefit of empty space skipping, we need to select the largest possible parent. This means that an optimal strategy would involve (re-)starting the traversal from the root, which is inefficient (see Fig. 5 for a 1D example); ideally we would like to start from one level below the last common ancestor. Because the node position encodes the full traversal path, we can compute this point via simple bit manipulation. Specifically, the number of levels we need to move up in the hierarchy is defined by the index of the most significant bit at which the coordinates of the current and successor nodes differ. This maps to $findMSB(Q_x \oplus Q_x^*)$ (where $\oplus$ is bit-wise XOR) and can be extended to $findMSB((Q_x \oplus Q_x^*)|(Q_y \oplus Q_y^*))$ for 2D, see supplemental material. The entire procedure maps very well to current GPUs as all the instructions are hardware-accelerated. In fact, this stack-less traversal is 25% faster than stack-based.

4.1 Disocclusion handling

Efficient handling of disocclusions in our 2.5D representation is a non-trivial task. A naïve solution would trace the ray against each quad-tree and pick the nearest hit among all hits. This, however, is slow and does not scale well with increasing number of layers. Some methods [Policarpo and Oliveira 2006] save on the ray traversal time by bundling multiple layers and casting rays through all of them simultaneously. This works much better, but still seems sub-optimal. Most screen-space applications, such as time-warping, DoF rendering, and stereo-warping, have small reprojection requirements, i.e., relatively few pixels end up being fetched from background layers. Our approach efficiently deals with these scenarios by tracing rays individually and indicating not only a valid hit event, but also if the ray has passed through the occlusion volume in the scene.

An occlusion volume describes the 3D space occluded by the data in a depth layer (similar to a shadow volume). Since we do not know what might be hidden in the occlusion volume of the foreground layer, rays that hit it, i.e., rays that would pass between occlusion boundaries, need intersection information from the background layer. This is because after reprojection, the background hit might end up being in the front of the foreground hit. To handle this, we test for intersections with the primitive (plane and AABB) *and* its occlusion volume. We never explicitly create this occlusion volume, but rather extrude a given node during intersection testing.

Knowing if the result of tracing the foreground is final, or whether we still need to trace the background, allows us to tremendously speed up the multi-layer tracing version of our algorithm. We have measured the ray distribution across different layers in the DoF rendering application, and even for large defocus blur, less than 1% of rays end up in the background layer. Note that the decision-making process is cascaded by nature. For example, adding a third layer will only impact rays that have missed both previous layers or hit an occlusion volume in the second layer. Hence the performance of our method scales well with the number of layers. In fact, for typical scenes, adding more than 3 layers has a negligible impact on ray tracing speed, and the overall system performance is limited by initial depth peeling and quad-tree construction stages.

The occlusion volume logic is not useful for single-layer depth+color case, where no information about occluded geometry is available. Usually, the best thing one can do is to inpaint the resulting hole with background data (see Fig. 6). For this particular case, we modify the tracing algorithm to provide a fast inpainting of disocclusions. Specifically, instead of traversing through the occluded part of space, we stop the traversal and return a valid hit at the intersection point with the occlusion volume. To make sure the point belongs to the background image data, we perform the intersection test with a slightly dilated occlusion volume. This effectively turns our algorithm into a fast height-field rendering method, and despite the disocclusion information being hallucinated, as we show in the next section, the method is still quite useful in the context of screen-space ray tracing.

5 Results

We now evaluate the performance and quality of our method for several screen-space applications. All experiments were conducted on a PC running Windows 7 64-bit version and the NVIDIA driver version 347.52. The system is equipped with an Intel Core i7-3930K, 64GB of RAM, and an NVIDIA Geforce GTX 980 with 4GB of RAM. All images were rendered at 1600×900 resolution, unless specified otherwise. The test sequences vary in the amount of camera motion and geometry complexity (see Table 1)—from relatively simple SPONZA to detailed LIVINGROOM (see Fig. 1) and SANMIGUEL scenes. The timings in this section exclude quad-tree construction, which is about 0.6ms per frame, unless otherwise noted; see Table 1 for detailed quad-tree construction timings.

5.1 View synthesis

We compare our approach to efficient implementations of three classes of view synthesis methods: mesh-based forward warping, height-field rendering and screen-space ray tracing.

The mesh-based warping methods represent the reference view as a regular grid mesh and rely on fixed-functionality GPU hardware to warp and rasterize it into a new view [Bowles et al. 2012; Didyk et al. 2010]. These approaches generally trade mesh resolution for performance. To have a fair comparison with our method, which resolves details at sub-pixel resolution, we have set up a mesh with one vertex per input image pixel. This simplifies the implementation as no mesh refining (snapping vertices to the nearest depth-discontinuities) or reprojection point optimization is required.

In Tables 2 and 3 we evaluate the performance for spatial and temporal reprojections. The performance of our approach is strongly correlated with the complexity of per-layer quad-trees. For the SPONZA scene, which has relatively simple geometry and produces few disocclusions, we are faster than mesh-based warping, regardless of the number of layers used. However, with detailed geometry that has large background/foreground depth differences, the speed advantage of our solution decreases. This is demonstrated in the SANMIGUEL sequence, where large camera motion produces lots of disocclusions and occlusion volume hits, which forces our method to shoot rays through the second layer for a significant portion of the image. However, unlike mesh-based forward warping, our method produces images with correctly resolved disocclusions (Fig. 7) and allows for arbitrary per-pixel reprojections, which enables single-pass light-field (Sec. 5.3) and DoF rendering (Sec. 5.2).

Another class of view-synthesis methods is height-field rendering [Musgrave 1988; Policarpo and Oliveira 2006], which aims to efficiently visualize an elevation map from an arbitrary point of view. We have evaluated the performance of our approach with respect to a recent GPU height-field rendering method [Tevs et al. 2008] that uses ray marching in a min-max pyramid followed by a binary search intersection refining step. Their ray traversal routine allows for fast arbitrary per-pixel reprojections, but does not support tracing through multiple depth layers and therefore fails to resolve disocclusions correctly. This makes it equivalent to a single-layer version of our approach. We have used the authors' GLSL implementation and selected their fastest iterative version of the algorithm. To improve the speed further we have disabled bilinear patch interpo-

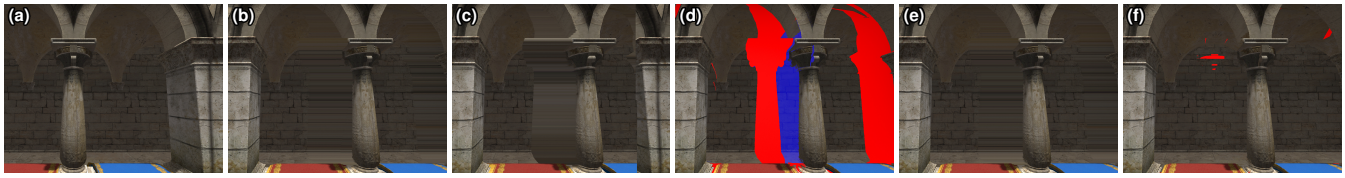

Figure 6: *Comparing view synthesis approaches for large camera motion.* ***(a)*** *Reference view.* ***(b)*** *Mesh-based reprojection fills disocclusions by stretching background triangles.* ***(c)*** *Height-field rendering methods [Tevs et al. 2008] produce less appealing results due to foreground preference during inpainting.* ***(d)*** *Our method tracing through a single depth layer. Apart from the regular hits we get misses (red) and hits of occlusion volume (blue).* ***(e)*** *In the fastest variant we fill both with the nearest background pixels.* ***(f)*** *Another variant performs tracing through the second depth layer and recovers the majority of misses. This can be repeated on subsequent layers to yield a full reconstruction.*

Method	Scene	Average time [ms]			$\frac{Mrays}{s}$
		T_{ref}	T_{eye}	T_{total}	
Mesh-based warping	SPONZA	0.59	1.47	3.56	N/A
	LIVINGROOM	0.76	1.37	3.53	N/A
	SANMIGUEL	2.73	1.37	5.46	N/A
Height-field tracing	SPONZA	0.59	3.5	7.7	405.75
	LIVINGROOM	0.79	3.85	8.5	373.78
	SANMIGUEL	2.75	4.33	11.4	332.41
DDA single layer	SPONZA	0.523	6.06	12.71	237.56
	LIVINGROOM	0.78	5.77	12.33	249.28
	SANMIGUEL	2.75	6.01	14.78	239.48
DDA two layers	SPONZA	1.13	8.26	17.65	174.06
	LIVINGROOM	1.41	7.80	17.01	184.60
	SANMIGUEL	5.51	8.22	21.93	175.05
Our single layer	SPONZA	0.59	1.1	3.43	1309.69
	LIVINGROOM	0.78	1.49	4.35	980.9
	SANMIGUEL	2.74	1.7	6.78	846.56
Our two layers	SPONZA	1.2	1.34	4.52	1076.22
	LIVINGROOM	1.45	1.7	5.45	850.5
	SANMIGUEL	5.52	2.02	10.24	709.8

Table 2: *Performance comparison of our approach in stereo-warping application. We render a central reference view (T_{ref}) and warp it to the left and right eye. T_{eye} is the averaged time for a warp to a single eye and T_{total} corresponds to the total stereo frame render time.*

Method	Scene	Average time [ms]			$\frac{Mrays}{s}$
		T_{ref}	T_{syn}	T_{amt}	
Mesh-based warping	SPONZA	0.54	1.43 ± 0.07	1.2	N/A
	LIVINGROOM	0.78	1.4 ± 0.04	1.25	N/A
	SANMIGUEL	2.75	2.23 ± 0.13	2.36	N/A
Height-field tracing	SPONZA	0.54	2.33 ± 0.1	1.88	617.76
	LIVINGROOM	0.79	2.2 ± 0.15	1.84	655.43
	SANMIGUEL	2.76	3.07 ± 0.73	2.99	468.6
DDA single layer	SPONZA	0.58	3.87 ± 3.67	3.05	371.70
	LIVINGROOM	0.77	1.05 ± 0.4	0.98	1368.36
	SANMIGUEL	2.71	3.11 ± 0.82	3.02	462.13
DDA two layers	SPONZA	1.25	4.08 ± 3.80	3.37	352.5
	LIVINGROOM	1.42	1.53 ± 0.99	1.5	940.56
	SANMIGUEL	5.5	3.55 ± 0.92	4.03	404.95
Our single layer	SPONZA	0.55	0.92 ± 0.24	0.83	1558.44
	LIVINGROOM	0.78	0.89 ± 0.14	0.86	1608.93
	SANMIGUEL	2.71	1.2 ± 0.29	1.57	1201.0
Our two layers	SPONZA	1.15	1.13 ± 0.45	1.13	1269.84
	LIVINGROOM	1.44	1.03 ± 0.73	1.13	1395.34
	SANMIGUEL	5.55	2.18 ± 1.35	3.02	660.85

Table 3: *Performance comparison of our approach for 15Hz to 60Hz conversion application. The mean reference frame rendering time T_{ref} together with new view synthesis time T_{syn} is used to compute amortized frame time T_{amt} for 60Hz rendering.*

lation. This normally produces pixel level staircase artifacts common to voxelization algorithms, but due to relatively small zoom-in factors of our reprojection applications, we have not found this to be an issue. Despite these optimizations, our approach is still $2.5\times$ faster on average (see Table 2 and 3). The performance improvement comes from our compressed depth-representation and more efficient traversal algorithm. Both reduce the overall intersection count and number of nodes visited, which directly maps to reduced texture fetch count and shorter run-times.

Finally, we compare against a recent screen-space GPU ray tracing method that supports tracing through multiple depth layers [McGuire and Mara 2014]. The traversal algorithm is based on the idea of perspective-correct DDA line rasterization, which minimizes the number of duplicated intersection tests and texture fetches. The method does not require pre-computation or any ancillary data structure. However, due to ray marching nature and fixed traversal step size, its performance tends to degrade proportionally to the ray hit distance. To reduce the variance of the performance the authors introduce an upper bound on per ray marching step count. In our experiments we used a minimum value at which the DDA method produced results that are artifact-free and equivalent to ours. Specifically, we set the maximum step count to 200 and 500 for stereo-warping and temporal upsampling applications respectively. For stereo-warping, where the reprojection is relatively small and constant, our method is from $4\times$ to $6\times$ faster (Table 2). The DDA method becomes more competitive for temporal-upsampling (Table 3), which has different (rotation + translation) and varying reprojection requirements. Moving away from the reference frame increases the reprojection magnitude, which maps to longer epipolar lines (rays) and results in high timings variance for SPONZA and SANMIGUEL scenes. The variance for LIVINGROOM case remains relatively small because of the slow camera motion that produces small differences between adjacent reference frames.

5.2 Depth-of-field rendering

Screen-space ray tracing is often used to simulate complex lens effects. Cook et al. [1984] showed that rendering phenomena like motion blur, depth-of-field (DoF), and shadow penumbras is feasible via lens sampling and proper ray distribution. We followed this methodology, and implemented a naïve DoF algorithm that samples the lens aperture [Shirley and Chiu 1997] and traces a fixed number of rays per pixel according to the thin-lens model. The per-pixel ray batches are then accumulated to form the final image.

As we show in Fig. 9, the knowledge of occluded regions in the scene is critical for high-quality DoF simulation. This is especially the case for defocus blur at large depth discontinuities, where rays travel into occluded parts of the scene geometry. Our method addresses this by tracing through multiple layers. Performance-wise, the DoF rendering workload represents the opposite scenario to Sec. 5.1. Here, most of the rays are short, incoherent and few of them end up in background layers. As shown in Table 4, our scheme breaks the 2.16 *billion* Rays-Per-Second (RPS) barrier for a single-layer rendering and 1.96 billion RPS for two layers. Interestingly, the DoF workloads are also handled relatively well by the DDA method [McGuire and Mara 2014]. Setting maximum ray marching step count to 25 produces optimum performance with rendering quality equivalent to ours. While our approach has a significant advantage in terms of RPS, the absolute FPS statistics suggest that

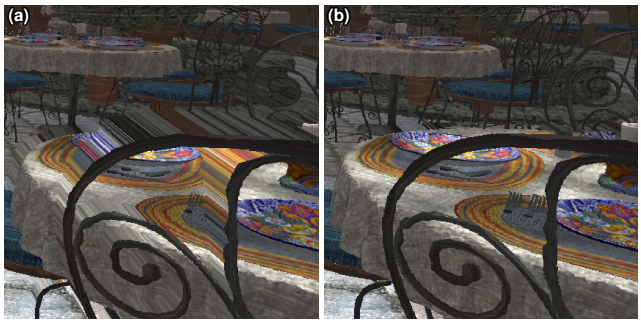

Figure 7: *Time-warping for large camera translation.* ***(a)*** *Mesh-based forward warping produces visible background stretching artifacts.* ***(b)*** *We use information from the background layers, avoiding these artifacts (see supplemental video for animated examples).*

Method	N_{pri}/N_{sec}	T_{ref} [ms]	T_{dof} [ms]	$\frac{M\,rays}{s}$	FPS
DDA single layer	4/1	0.56	4.74	1215.18	188.4
	8/1	0.54	8.15	1412.11	114.85
DDA two layers	4/1	1.25	6.56	876.97	127.87
	8/1	1.23	9.34	1233.4	94.60
Our single layer	4/1	0.54	2.8	2055.67	251.13
	8/1	0.54	5.32	2167.05	152.67
	4/4	0.56	3.47	1657.07	214.5
	8/4	0.54	6.63	1737.55	127.73
Our two layers	4/1	1.22	3.21	1792.72	197.08
	8/1	1.21	5.85	1967.21	129.71
	4/4	1.24	3.84	1496.88	174.58
	8/4	1.22	7.22	1594.46	109.51

Table 4: *Impact of the aperture sampling configuration on performance of depth-of-field rendering in* SPONZA *sequence. Each test case is configured to cast N_{pri} rays and accumulate them with N_{sec} color samples per ray. T_{ref} denotes the rendering time for the reference layer(s) and T_{dof} is the DoF image rendering time. Additionally, we provide the ray tracing speed and the overall application FPS (including quad-tree construction time).*

this improvement is to some extent consumed by the acceleration structure build overhead.

Unfortunately, random lens sampling with just a few rays produces visible noise in the final image. To reduce the noise, we have implemented a secondary sampling stage that exploits the characteristics of our quad-tree data structure. Specifically, after hitting the plane leaf-node, we generate additional rays in close proximity to the primary ray (using the same random distribution), but instead of tracing them through the scene, we assume their visibility and simply intersect them with the primary ray hit plane. Some intersection points might land "in the air", therefore we lower their color sample contribution to the final pixel estimate by weighting them by $e^{-(z_i-z_b)^2/\sigma^2}$, where z_i is the intersection z value and z_b is the depth buffer value at the intersection point. Fig. 8 shows the impact this has on DoF rendering noise levels. The proposed sampling strategy only approximates the physically-correct solution, but as we show in Table 4, it allows us to significantly reduce the ray tracing overhead without sacrificing the defocus blur quality.

5.3 Image retargeting for multi-view displays

Glasses-free 3DTVs, particularly those using parallax barriers [Ives 1903] or lenticular arrays [Lippmann 1908], require multiple views of the same scene. Unfortunately, rendering and transmission of dozens of views is expensive both in terms of computation and bandwidth/storage requirements. One way to address this problem is to send/compute only a small subset of all views, so called *reference views*, and use these to synthesize missing in-between views. Unlike existing 3DTVs, near-eye light-field displays [Lanman and Luebke 2013] require rendering from hundreds to thousands of individual scene views. In the original paper, the authors describe two rendering approaches for their display. The first one relies on GPU ray tracing (with NVIDIA OptiX) to produce accurate elemental image array for the display. The other one is a much simpler, where frames from the left and right eye are placed on a virtual plane and sampled to produce an image for micro-lens array. While efficient, this approach significantly under-utilizes the capabilities of a near-eye light-field display, as it cannot reproduce large disparities or accommodation effects this way. Here we implement the first approach: specifically, given color+depth buffers for the left and right eye, we generate a complete elemental image set with our ray tracing approach (see Fig. 1d). This preserves the disparity range of the original stereo content and also enables the eyes to accommodate. Our results are visually indistinguishable from the OptiX solution, and depending on the scene and ray tracer configuration, it takes from 1.1ms to 3ms to reproject the image into an image array for a single eye, about three times faster than OptiX.

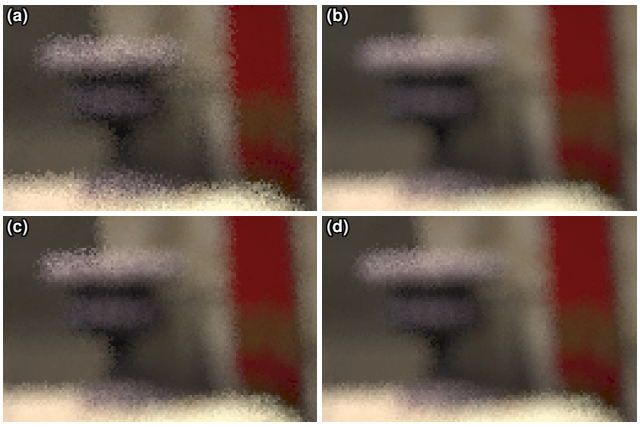

Figure 8: *Depth-of-field rendering noise reduction through oversampling.* ***(a)*** *Integration of only 8rpp (rays per pixel) produces noisy result.* ***(b)*** *32rpp improves the quality, but is 3.9x more costly to compute.* ***(c)*** *4rpp combined with 7 additional samples per ray produces in-between quality while being 25% faster than 8rpp alone.* ***(d)*** *Finally, 8rpp combined with 3 extra samples per ray produces results as in (b) while being only 20% slower than (a).*

5.4 Ray-traced reflections

Many ray tracing algorithms exist to create plausible glossy reflection. As mentioned earlier, screen-space ray tracing is commonly used for this, e.g., the algorithm of Souza et al. [2011] or its more accurate recent variants [Mara et al. 2013; McGuire and Mara 2014] that can also handle multiple layers to solve disocclusions.

As reflections are likely to come from outside the current viewport/screen-space, methods exist to enable this. Umenhoffer et al. [2007] create a cube map at the center of a reflective object, including the respective depth maps and possibly multiple layers. Reflections are rendered by screen-space ray tracing in one or more of the cube map faces. We take a similar approach, but speed up ray tracing using our adaptive data structure for each cube map face.

To show the applicability of our proposed data structure and ray traversal algorithm to non-screen-space effects, we use ray tracing to generate specular (Fig. 11) and glossy reflections (Fig. 13) including multiple bounces and self-reflections on non-planar reflective objects as described by Cook et al. [1984]. This can lead to more incoherent ray tracing workloads, especially when applying

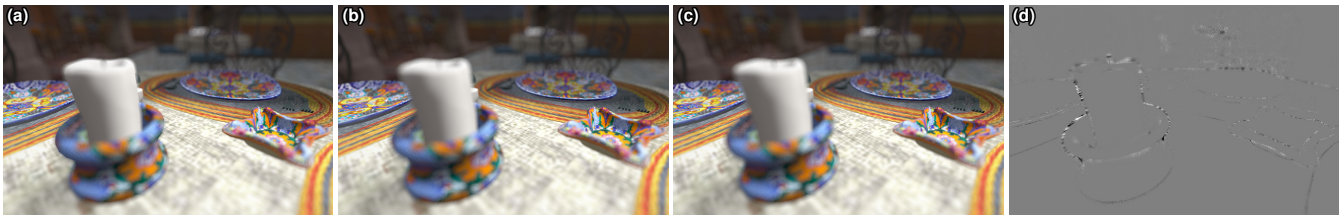

Figure 9: *Depth-of-field rendering using the thin-lens model.* ***(a)*** *With a single depth layer we cannot resolve defocus blur at large depth discontinuities (e.g., the edge of the candle).* ***(b)*** *Our method produces correct image by tracing rays through two layers,* ***(c)*** *or by tracing through only a single depth layer but hallucinating background through inpainting (see Sec. 4.1), which approximates (b) and is faster than (a).* ***(d)*** *5× difference between (b) and (c).*

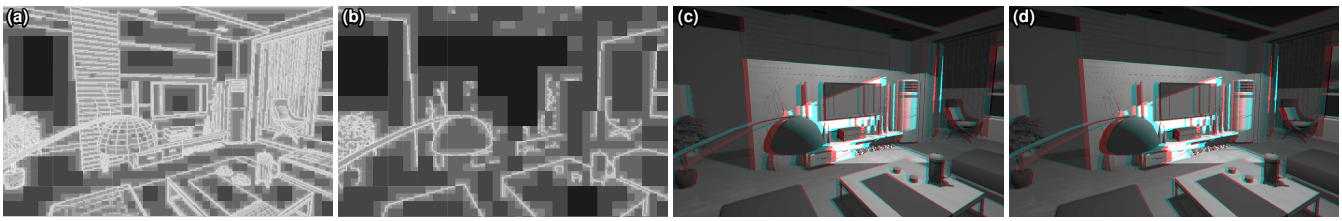

Figure 10: *Impact of quad-tree compression on stereo-warping.* ***(a)*** *AABB intersection count visualization for a close to lossless compression of corresponding depth layers of the reference cyclopean view.* ***(b)*** *Aggressive compression of the quad-tree speeds up ray tracing by 12%.* ***(c,d)*** *Despite the difference in approximation quality between (a) and (b), the resulting stereo image pairs are visually indistinguishable.*

Method	$N_p/N_s/N_b$	T_{cm} [ms]	T_{ref} [ms]	T_{rr} [ms]	FPS
Our single layer	1/8/1	2.38	0.34	1.974	197.31
	3/8/1	2.39	0.41	6.96	98.68
	3/4/3	2.33	0.366	10.25	75.03
	3/8/3	2.35	0.388	12.38	64.64
Our two layers	1/8/1	5.15	0.415	4.91	92.21
	3/8/1	5.20	0.37	13.5	51.40
	3/4/3	5.19	0.34	22.96	34.64
	3/8/3	5.27	0.34	24.75	32.52

Table 5: *Rendering performance of complex multi-bounce glossy reflections. Both single- and two-layer based algorithm is configured to cast N_p primary rays, followed by N_s extra samples per ray. Each primary ray is allowed to bounce N_b times. T_{cm}, T_{ref}, and T_{rr} denote the cube map, input frame, and reflection rendering times respectively. Last column shows the average application FPS.*

small reflection exponents to simulate rough surfaces.

We generate a cube map ($6 \times 1024 \times 1024$ pixels big) at the camera location, and create our adaptive data structure for each face of the cube map. Note that in theory we could support other environment map representations, but only if they do not lead to curved rays. In our implementation, we simply render the scene six times, once to each cube map face. More efficient solutions, such as viewport multi-casting exist, but this is not the focus of our work.

The primary intersection point is computed using rasterization, rendering the reflective objects only. For each intersection point we sample a new direction for the reflection vector by sampling the normalized Blinn-Phong BRDF at this position. If we hit a reflective surface, we generate a new ray until the maximum number of bounces is reached. At a diffuse surface we discontinue tracing.

The core ray traversal algorithm requires very few modifications for supporting tracing in the cube map. We begin by determining the frustum (cube face) the ray originates from. Then we trace the ray in the corresponding acceleration structure. If no valid hit was found, we transition to a new face based on which of the frustum planes of the current view the ray hits. If the ray hits the far plane of any view (ray is leaving the scene) we terminate the trace. We show the performance of our ray tracing approach for various scenarios in Fig. 11, 12 and Table 5.

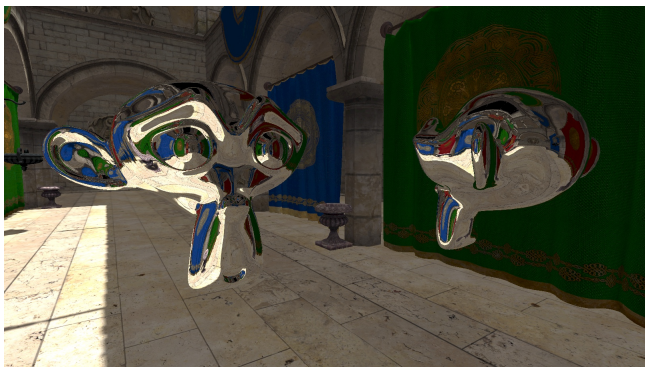

Figure 11: *Single depth+color layer ray-traced specular reflections (no self-reflections). Our per cube map face acceleration data structure is static and build in the first frame in 5.277ms. It takes 1.518ms to render the scene view and 1.062ms to trace the reflections, resulting in 387 average FPS. The two layer version takes 2.55ms to trace, yielding 234 average FPS. Note that since our algorithm effectively* ***reprojects*** *the cube map during tracing, we can use the same cubemap to render* ***correct*** *reflections for both objects.*

6 Discussion

Depth compression impact on performance and quality By increasing γ_{norm} and γ_{dist} the quad-tree construction algorithm coarsens—with gradually increasing tolerance—smoothly-varying geometry in the scene, which in the limit produces a scene filled up with billboard-like objects. However, since we optimize the quad-tree for the reference view, and all the screen-space applications we demonstrate require relatively small parallax shift and ray direction changes, even such a coarse geometry approximation can produce correct results (see Fig. 10).

In our experiments we set $\gamma_{norm} = \cos(3^\circ)$ and $\gamma_{dist} = 10^{-5}$, which provided a 5-10% performance gain (with respect to lossless settings) without introducing any visible reprojection artifacts. The compression thresholds can be further tuned to exploit the nature of perspective projection—the hit precision requirements fall with the distance to the object. We have implemented a distance-adaptive compression by linearly increasing γ_{norm} and γ_{dist} thresholds with

Figure 12: *A comparison with a standard cube-map-based reflections.* ***(a)*** *Cube map rendered at position of the object in the far back produces incorrect reflections for the object in the front that is far from its center of projection.* ***(b)*** *Our method can address this scenario and produce correct reflection by reprojecting the data stored in the acceleration structure.*

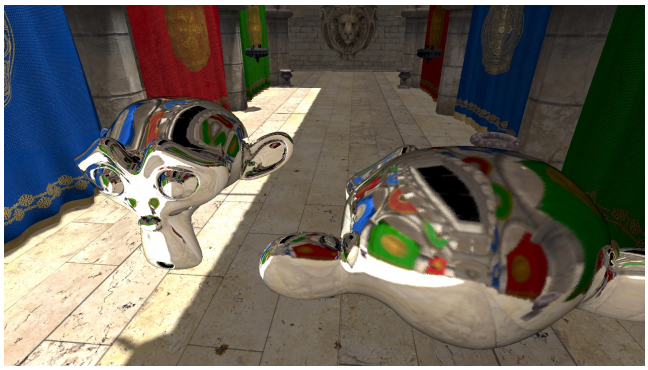

Figure 13: *Ray-traced rough reflection according to the Blinn-Phong BRDF with up to three bounces. Here, the front object has a specular coefficient of $N = 1000$, the object in the back is essentially specular ($N = 100000$).*

the node's mean depth value, which resulted in, on average, a 10% performance gain for view-synthesis applications and 15% for DoF rendering. Fig. 14 and 15 include more detailed evaluation, where we show the impact of γ_{norm}, γ_{dist} and depth-adaptive quantization on performance and quality of the stereo warping application.

Limitations While the single-layer (with background inpainting) variant of our method has good all-around characteristics, the performance of multi-layer version decreases with the number of ray misses and occlusion volume hits. This is because large disocclusions generate long ray traversal paths, which slows down the tracing process, making our method not as efficient for large-displacement view synthesis.

In our current multi-layer traversal implementation, when moving up close to geometry, some rays can pass through tiny cracks between unaligned plane nodes that should otherwise form a continuous surface. This could be fixed by generating more precise input normal vectors (use g-buffers instead of depth-based normal reconstruction) or by combining several neighboring planes to implement a more complex bi-linear patch intersection test.

Finally, we consider only opaque geometry and diffuse lighting models. However, support of multi-sample rendering and deferred shading is feasible and can be added as a straightforward extension.

7 Conclusions

We have presented a novel screen-space ray tracing method tailored for single- and multi-layered depth representations. We have demonstrated its performance and benefits in several screen-space rendering applications. We achieve real-time performance by combining a compact and steerable traversal acceleration structure with an efficient ray tracing algorithm, reaching the level of specialized state-of-the-art approaches for many applications. While our method is not a replacement for general purpose ray tracing frameworks, such as NVIDIA OptiX, it can be thought of as an efficient alternative for problems requiring 2.5D ray tracing capabilities. In the future, we would like to extend this work to support acceleration of volumetric rendering effects and investigate the application to approximate global illumination. We also plan to work on improving the quad-tree compression algorithm, which currently only considers geometric distortions. Accounting for underlying material properties, such as texture contrast or specular highlights, could further improve the performance and quality of our approach.

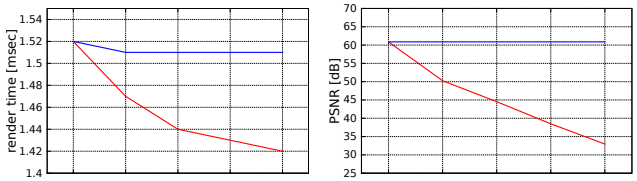

Figure 14: *Impact of γ_{norm} and γ_{dist} on the quality and performance.* ***(left)*** *Frame render time in msec.* ***(right)*** *Frame PSNR in dB.* ***(red)*** *Fixed $\gamma_{norm} = cos(3°)$ while logarithmically decreasing $\gamma_{dist} = 10^{-5}$ to $\gamma_{dist} = 10^{-1}$.* ***(blue)*** *Fixed $\gamma_{dist} = 10^{-5}$ while γ_{norm} varies between $[cos(3°), cos(25°)]$.*

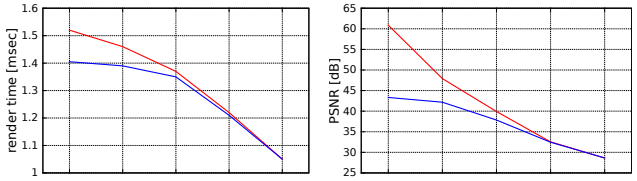

Figure 15: *Impact of depth-adaptive quantization on quality and performance. The compression values γ_{norm} and γ_{dist} vary between $[cos(3°), cos(25°)]$ and $[10^{-5}, 10^{-1}]$ respectively.* ***(left)*** *Frame render time in msec.* ***(right)*** *Frame PSNR in dB. Depth-adaptive quantization disabled* ***(red)*** *and enabled* ***(blue)****.*

Acknowledgments Sven Widmer is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt. Crytek-sponza scene courtesy of Frank Meinl. San-miguel scene courtesy of Guillermo M. Leal Llaguno.

References

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics*.

BABOUD, L., AND DECORET, X. 2006. Rendering geometry with relief textures. In *Proc. Graphics Interface*.

BOWLES, H., MITCHELL, K., SUMNER, R. W., MOORE, J., AND GROSS, M. 2012. Iterative image warping. *CGF 31*, 2.

CARR, N. A., HOBEROCK, J., CRANE, K., AND HART, J. C. 2006. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proc. Graphics Interface*.

CHEN, J., BARAN, I., DURAND, F., AND JAROSZ, W. 2011. Real-time volumetric shadows using 1D min-max mipmaps. In *Proc. I3D*.

COHEN, D., AND SHAKED, A. 1993. Photo-realistic imaging of digital terrains. *CGF 12*, 3.

COHEN-OR, D., RICH, E., LERNER, U., AND SHENKAR, V. 1996. A real-time photo-realistic visual flythrough. *IEEE TVCG 2*, 3.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *SIGGRAPH Computer Graphics 18*, 3.

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive indirect illumination using voxel cone tracing. *CGF 30*, 7.

DIDYK, P., EISEMANN, E., RITSCHEL, T., MYSZKOWSKI, K., AND SEIDEL, H.-P. 2010. Perceptually-motivated real-time temporal upsampling of 3D content for high-refresh-rate displays. *CGF 29*, 2.

DONNELLY, W. 2005. Per-pixel displacement mapping with distance functions. In *GPU Gems 2*. Addison-Wesley.

FRISKEN, S. F., AND PERRY, R. N. 2002. Simple and efficient traversal methods for quadtrees and octrees. *J. Graph. Tools 7*.

FUJIMOTO, A., TANAKA, T., AND IWATA, K. 1986. ARTS: Accelerated ray-tracing system. *IEEE CG & A 6*, 4.

GANESTAM, P., AND DOGGETT, M. 2014. Real-time multiply recursive reflections and refractions using hybrid rendering. *The Visual Computer*.

GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and faster HLBVH with work queues. In *Proc. HPG*.

GUENNEBAUD, G., BARTHE, L., AND PAULIN, M. 2006. Real-time soft shadow mapping by backprojection. In *Proc. EGSR*.

HERZOG, R., EISEMANN, E., MYSZKOWSKI, K., AND SEIDEL, H.-P. 2010. Spatio-temporal upsampling on the GPU. In *Proc. I3D*.

IVES, F., 1903. Parallax stereogram and process of making same, Apr. 14. US Patent 725,567.

KARRAS, T., AND AILA, T. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proc. HPG*.

KOLB, A., AND REZK-SALAMA, C. 2005. Efficient empty space skipping for per-pixel displacement mapping. In *Proc. VMV*.

LAINE, S., AND KARRAS, T. 2010. Efficient sparse voxel octrees. In *Proc. I3D*.

LANMAN, D., AND LUEBKE, D. 2013. Near-eye light field displays. *ACM TOG 32*, 6.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. *CGF 28*, 2.

LEE, S., EISEMANN, E., AND SEIDEL, H.-P. 2009. Depth-of-field rendering with multiview synthesis. *ACM TOG 28*, 5.

LEE, S., EISEMANN, E., AND SEIDEL, H.-P. 2010. Real-time lens blur effects and focus control. *ACM Trans. Graph. 29*, 4.

LIPPMANN, G. 1908. Épreuves réversibles donnant la sensation du relief. *J. Phys. Theor. Appl. 7*, 1.

MARA, M., MCGUIRE, M., AND LUEBKE, D. 2013. Lighting deep g-buffers: Single-pass, layered depth images with minimum separation applied to indirect illumination. Tech. rep., NVIDIA.

MARA, M., NOWROUZEZAHRAI, D., MCGUIRE, M., AND LUEBKE, D. 2014. Fast global illumination approximations on deep g-buffers. Tech. rep., NVIDIA.

MCGUIRE, M., AND MARA, M. 2014. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques*.

MITTRING, M. 2007. Finding next gen: Cryengine 2. In *ACM SIGGRAPH Courses*.

MUSGRAVE, F. K. 1988. Grid tracing: Fast ray tracing for height fields. Tech. Rep. YALEU/DCS/RR-639, Yale University.

NEHAB, D., SANDER, P. V., LAWRENCE, J., TATARCHUK, N., AND ISIDORO, J. R. 2007. Accelerating real-time shading with reverse reprojection caching. In *Proc. Graphics Hardware*.

NOVÁK, J., AND DACHSBACHER, C. 2012. Rasterized bounding volume hierarchies. *CGF 31*, 2.

OH, K., KI, H., AND LEE, C.-H. 2006. Pyramidal displacement mapping: a GPU based artifacts-free ray tracing through an image pyramid. In *Proc. VRST*.

PANTALEONI, J., AND LUEBKE, D. 2010. HLBVH: hierarchical LBVH construction for real-time raytracing of dynamic geometry. In *Proc. HPG*.

POLICARPO, F., AND OLIVEIRA, M. M. 2006. Relief mapping of non-height-field surface details. In *Proc. I3D*.

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. L. D. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *Proc. I3D*.

RITSCHEL, T., GROSCH, T., AND SEIDEL, H.-P. 2009. Approximating dynamic global illumination in image space. In *Proc. I3D*.

SHADE, J., GORTLER, S., HE, L.-W., AND SZELISKI, R. 1998. Layered depth images. In *Proc. SIGGRAPH*.

SHIRLEY, P., AND CHIU, K. 1997. A low distortion map between disk and square. *J. Graph. Tools*.

SITTHI-AMORN, P., LAWRENCE, J., YANG, L., SANDER, P. V., AND NEHAB, D. 2008. An improved shading cache for modern GPUs. In *Proc. Graphics Hardware*.

SOUSA, T., KASYAN, N., AND SCHULZ, N. 2011. Secrets of cryengine 3 graphics technology. In *ACM SIGGRAPH Courses*.

TEVS, A., IHRKE, I., AND SEIDEL, H.-P. 2008. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Proc. I3D*.

UMENHOFFER, T., PATOW, G., AND SZIRMAY-KALOS, L. 2007. Robust multiple specular reflections and refractions. In *GPU Gems 3*, H. Nguyen, Ed.

WU, Z., ZHAO, F., AND LIU, X. 2011. SAH KD-tree construction on GPU. In *Proc. HPG*.

WYMAN, C. 2005. Interactive image-space refraction of nearby geometry. In *Proc. GRAPHITE*.

YU, X., WANG, R., AND YU, J. 2010. Real-time depth of field rendering via dynamic light field generation and filtering. *CGF 29*, 7.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time KD-tree construction on graphics hardware. *ACM TOG 27*, 5.